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 sviluppo di un approccio per la realizzazione di una chatbot RASA con knowledge base memorizzata su graphDB

Design and development of an approach to realize a RASA chatbot with a knowledge base stored on graphDB

Relatore

Prof. Domenico Ursino

Candidato

Alessandro Mentuccia

Anno Accademico 2018-2019

Indice

Introduzione 9

1 Il Natural Language Processing 13
1.1 Cos'è la NLP 13
1.2 Metodi Ruled-Based e Statistical 13
1.3 Varie applicazioni 15
1.4 Ambiti di utilizzo 17

2 I Chatbot 19
2.1 Premessa 19
2.2 L'evoluzione dei chatbot 19
2.3 Le tipologie di chatbot 21
2.3.1 Rule-based chatbot 21
2.3.2 IR-based chatbot 21
2.3.3 Generative chatbot 22
2.3.4 Open e Closed domain 23
2.3.5 Task-oriented chatbot 23
2.4 Scenari d'uso 25
2.5 Limitazioni dei chatbot 26

3 Il framework RASA 29
3.1 Architettura 29
3.2 Rasa NLU 30
3.3 Rasa Core 31
3.4 Panoramica sulle componenti di RASA 31

4 MediciBot: Analisi dei requisiti e generazione dataset di training 35
4.1 Analisi dei requisiti 35
4.2 Knowledge base e Neo4j 38
4.3 Generazione del dataset di training 39
4.3.1 Dataset Generator Platform 40
4.3.2 Definizione file di template 40
4.3.3 Sintassi dei template 41

5 Implementazione di Medici-bot 47
5.1 Panoramica per la configurazione di Medici-bot 47
5.2 Implementazione del file di configurazione del chatbot 47
5.2.1 config.yml 47
5.2.2 credentials.yml 49
5.2.3 domain.yml 50
5.2.4 nlu.md 54
5.2.5 stories.md 55
5.2.6 actions.py 58
5.2.7 neo4jstorage.py 71
5.2.8 utils_bot.py 75
5.2.9 ga_connector.py 82
5.2.10 alexa_connector.py 85
5.3 Problematiche affrontate durante lo sviluppo 86
5.3.1 Disambiguazione tra le mentions e duckling 86
5.3.2 Aggiunta del button con link a google maps 88
5.3.3 Definizione dell'account linking in Google Assistant 89
5.3.4 Salvataggio del dottore preferito tra le informazioni personali dell'utente 90

6 Medici-bot: Fase di validazione 95
6.1 Panoramica sulle procedure di validazione secondo il framework Rasa 95
6.2 Validazione di Medici-bot 98
6.2.1 Definizione della validazione effettuata 98
6.2.2 Risultati della validazione 99
6.3 Test delle funzionalità aggiuntive del chatbot 105

7 Conclusioni e futuri sviluppi 109
7.1 Conclusioni 109
7.2 Sviluppi futuri 110

Riferimenti bibliografici 111

Ringraziamenti 113

Elenco delle figure

1.1 Imitation Game: il partecipante A (essere umano) comunica con B e C senza conoscerli. Se A dopo aver conversato con B e C non riesce a definire chi dei due sia una macchina e chi un uomo, allora la macchina supera il test di Turing 14
1.2 Rappresentazione della differenza tra *Lemmatization* e *Stemming* 16
1.3 Rappresentazione della frase "It is fine if you fine me, but don't rub it in" in un Parse Tree con le relative categorie grammaticali 17

2.1 Un modello Seq2Seq applicato agli agenti di dialogo 22
2.2 Risultato dell'attività di NLU: Intent classification e Slot filling 24

3.1 Schema di elaborazione dei messaggi in Rasa 30
3.2 Caratteristiche degli *entity extractor* che RASA permette di usare . . . 32
3.3 Esempio di Story 33

4.1 Raffigurazione del primo scenario di conversazione da soddisfare 36
4.2 Raffigurazione del secondo scenario di conversazione da soddisfare . . . 37
4.3 Rappresentazione a nodi dei dati sul DB 39
4.4 Interfaccia del *Dataset Generator* 40
4.5 Architettura del servizio *Medici-bot* 46

6.1 Matrice di confusione degli *intent* 100
6.2 Istogramma della distribuzione della confidenza per tutte le previsioni 102
6.3 Matrice di confusione delle *action* rilevate 104
6.4 Esempio di una richiesta degli orari di un dottore 105
6.5 Dialogo contenente il salvataggio del medico preferito 106
6.6 Dialogo contenente l'eliminazione del medico preferito 106
6.7 Dialogo relativo alla richiesta degli ambulatori e risposta contenente la basicCard con link alla posizione su Google Maps 107

Elenco dei listati

4.1 Template per la generazione del dataset di training 41
4.2 Esempi di template ricavati dal file per la generazione 45
5.1 Rappresentazione del file `config.yml` 48
5.2 Rappresentazione del file contenente i sinonimi degli *object_type* 48
5.3 Istruzione da riga di comando per l'avvio del servizio *Duckling* 49
5.4 Rappresentazione del file `credential.yml` 50
5.5 Esempio della definizione del *template* `utter_ask_rephrase` 53
5.6 Rappresentazione del file `domain.yml` 53
5.7 Esempio di frase definita nel file `nlu.md` dell'*intent* `query_knowledge_attribute_of` e relativo template 55
5.8 Sintassi per la definizione di una *story* 55
5.9 Rappresentazione del file `stories.md` 55
5.10 Codice del metodo `utter_objects` 60
5.11 Codice del metodo `utter_attribute_value` 63
5.12 Codice del metodo `run` della classe `ActionPersonaList` 64
5.13 Codice del metodo `_query_objects_my` della classe `ActionPersonaList` 65
5.14 Codice del metodo `run` della classe `ActionAttributoPersona` 66
5.15 Codice del metodo `_query_attribute` della classe `ActionAttributoPersona` 68
5.16 Codice della classe `ActionDetails` 70
5.17 Codice del metodo `get_objects` 72
5.18 Esempio di una query generata dal metodo `get_objects` 73
5.19 Codice del metodo `relation_mapping_function` 73
5.20 Codice del metodo `get_object_type_by_attribute` 75
5.21 Il contenuto del file `file_utils.json` 76
5.22 Codice della classe `Rasa_utils` 78
5.23 Codice della classe `GoogleConnector` 82
5.24 Codice della classe `AlexaConnector` 85
5.25 Codice del metodo `get_date` 87
5.26 Campo `basicCard` definito nel `responce.json` 89
5.27 Sezione di codice contenente l'aggiunta del pulsante nel `responce.json` 89
5.28 Campo `user` definito nel `responce.json` 91
5.29 Codice della classe `ActionPersonalDoctor` 91

5.30 Codice della classe `ActionInformationDoctor` 92
5.31 Codice della classe `ActionDeleteDoctor` 93
6.1 Contenuto del file `intent_report.json` 99
6.2 Parte del contenuto del file `intent_errors.json` 101
6.3 Contenuto del file `CRFEntityExtractor_report.json` 101
6.4 Parte del contenuto del file `CRFEntityExtractor_error.json` 103
6.5 Contenuto del file `failed_stories.md` 105

Introduzione

Le macchine possono pensare? Erano gli anni '50 quando Alan Turing, nel suo articolo *Computing machinery and intelligence*, si chiese se le macchine fossero capaci di concatenare idee e di poterle esprimere, ma, soprattutto, si domandò secondo quale criterio fosse possibile stabilirlo. Il test di Turing, nato, appunto, per rispondere a tale quesito, definisce una macchina "intelligente" se, a seguito di un'interazione con una persona, quest'ultima non è in grado di distinguere se la conversazione sia avvenuta con una macchina o, al contrario, con un interlocutore umano. È così che si iniziò a parlare di chatbot, ovvero dei software in grado di intrattenere delle conversazioni con umani.

Uno dei primi chatbot ad aver superato il test di Turing fu ELIZA, ideato presso il MIT AI Laboratory da Joseph Weizenbaum nel 1966. ELIZA era capace di intrattenere una conversazione testuale, simulando una prima visita psichiatrica con un paziente; dava, inoltre, l'impressione di essere realmente intelligente e di saper rispondere riformulando le risposte dell'utente, anche se, in verità, il software si basava su un meccanismo di pattern matching e sulla sostituzione delle parole chiave con frasi predefinite.

Nel corso dei decenni, molte altre soluzioni furono presentate, come, ad esempio, il progetto ALICE (*Artificial Linguistic Internet Computer Entity*), proposto da Richard Wallace nel 1995, che vinse il Loebner Prize negli anni 2000, 2001 e 2004. ALICE era un chatbot basato sul Natural Language Processing e programmato nel linguaggio AIML (*Artificial Intelligence Markup Language*); quest'ultimo è utilizzato per dichiarare regole di corrispondenza tra le parole e le frasi date in input dall'utente con categorie di argomenti.

Nel 1997, Rollo Carpenter lanciò per la prima volta su Internet il chatbot Jabberwacky, capace di utilizzare un linguaggio ricavato da milioni di interazioni online con persone. Grazie a ciò, si riuscì a definire una innovazione rispetto alle precedenti soluzioni chatbot che, invece, basavano il recupero delle risposte da database statici.

Quello che, però, mancava a questi chatbot era uno scenario di lavoro nel quale poter ricoprire un ruolo assistenziale, fornendo dei servizi utili all'uomo. Ciò cambiò all'inizio del nuovo millenio, quando fu presentato, dalla startup ActiveBuddy Inc., il chatBot SmarterChild. Il progetto, che inizialmente era stato presentato come un gioco di avventura basato sul linguaggio naturale, possedeva la capacità di collegarsi a grandi quantità di informazioni in diversi domini di conoscenza; esso spaziava e

ricercava informazioni riguardanti il meteo, le notizie e gli orari di eventi. Tale caratteristica ha permesso al chatbot di essere riconosciuto come un precursore dei più recenti assistenti vocali, e cioè una macchina capace di fornire un servizio utile all'utente.

Dopodichè, nel corso di quest'ultimo decennio, vi è stato l'avvento degli assistenti personali virtuali, tra cui Siri (sviluppato da Apple), Google Assistant (sviluppato da Google), Cortana (sviluppato da Microsoft) e Alexa (sviluppato da Amazon).

Anche se creati in modi diversi e da aziende differenti, essi si accomunano per delle caratteristiche specifiche. In particolare, le loro funzionalità si appoggiano nella possibilità di avere, attraverso una connessione Internet, un'interazione con un sistema centrale in grado di ricevere ed elaborare i comandi vocali di un'utente.

Nel frattempo, i chatbot, con il passare degli anni, si sono introdotti nella routine giornaliera delle persone e, così, anche nelle loro case. Oggigiorno troviamo assistenti virtuali nei cellulari, computer, televisori, ed anche in alcuni dispositivi domestici. Ad esempio, di mattina, è possibile alzarsi, chiedere al chatbot le temperature, ed avere una chiara idea del meteo della giornata senza neanche aprire le finestre. Inoltre, tali chatbot sono in grado di svolgere compiti sempre più complessi, come leggere ed inviare messaggi, gestire le chiamate, i promemoria, controllare le applicazioni per la riproduzione di contenuti multimediali e i dispositivi connessi (come termostati o luci) attraverso l'*Internet Of Things*. Di conseguenza, con l'obiettivo di rispondere alle richieste più comuni degli utenti, molte aziende hanno iniziato ad implementare nei loro servizi di assistenza al cliente delle soluzioni che comprendessero dei chatbot. Alcuni di questi sono stati distribuiti all'interno delle applicazioni di messaggistica istantanea (ad esempio Telegram, Messenger, etc.), oppure altri attraverso gli assistenti virtuali, i quali permettono di agganciare le singole soluzioni chatbot e fornire all'utente delle piattaforme complete di servizi da poter richiamare.

È proprio all'interno di quest'ultimo contesto applicativo, ovvero l'assistenza al cliente attraverso gli assistenti virtuali, che si colloca il lavoro descritto nella presente tesi che descrive la creazione di un chatbot basato sul framework Rasa. Esso dovrà operare all'interno di uno scenario di assistenza riguardante, nello specifico, le informazioni relative ai medici e pediatri della regione Piemonte. Per di più, si è voluto sviluppare un approccio per la realizzazione di chatbot basati su Rasa gestendo la corrispettiva base di conoscenza su un graphDB Neo4j.

Nel dettaglio, durante la creazione di tale bot riguardante i medici, si sono voluti definire un approccio, un ambiente ed un'architettura generale per lo sviluppo di futuri chatbot basati su Rasa.

Tale progetto, che chiameremo d'ora in avanti *Medici-bot*, è stato sviluppato in collaborazione e per conto del CSI-Piemonte (Consorzio per il Sistema Informativo del Piemonte). Esso presenta una *knowledge base* formata dalle informazioni relative a tre entità, che sono i medici, gli ambulatori e gli orari di ricevimento. Inoltre, il chatbot sarà capace di rispondere, fondamentalmente, a due tipologie di domande:

1. la richiesta di una lista di entità che corrispondono ad un parametro di ricerca, ad esempio, *"Mi diresti i pediatri che lavorano a Torino?"*.
2. la richiesta di un attributo relativo ad una specifica entità, ad esempio, *"Vorrei sapere il telefono del dottor Rossi"*.

Oltre alle due richieste descritte, il chatbot sarà in grado di dialogare attraverso strutture conversazionali naturali e più comuni ai costumi dell'uomo, ad esempio, esso consente di richiedere approfondimenti su un risultato o aggiungere dei campi di ricerca per la richiesta.

Tale chatbot sarà in grado di fornire delle funzionalità ausiliarie al suo operato; l'utente potrà essere reinderizzato sulla posizione degli ambulatori su Google Maps, oppure, potrà salvare il medico di fiducia nel caso in cui, prima di accedere, si è autenticato.

La presente tesi è strutturata come di seguito specificato:

- Nel Capitolo 1 verrà introdotto il Natural Language Processing.
- Nel Capitolo 2 si parlerà dei chatbot, in particolare delle loro caratteristiche e delle loro applicazioni.
- Nel Capitolo 3 si descriverà il framework Rasa, ovvero la tecnologia su cui è basato il chatbot proposto.
- Nel Capitolo 4 verrà illustrata la fase di analisi dei requisiti e la generazione automatica del dataset di training del bot attraverso il servizio *Dataset Generator*.
- Nel Capitolo 5 saranno proposte l'implementazione di Medici-bot e le soluzioni sviluppate.
- Nel Capitolo 6 sarà illustrata la fase di validazione del chatbot Medici-bot.
- Nel Capitolo 7 saranno tratte le conclusioni e verranno presentati alcuni possibili sviluppi futuri di tale progetto.

1

Il Natural Language Processing

In questo primo capitolo verranno introdotti il Natural Language Processing ed alcuni approcci utilizzati per i sistemi di elaborazione del linguaggio

1.1 Cos'è la NLP

Il *Natural Language Processing*, o *NLP*, è una branca dell'Intelligenza Artificiale che si focalizza nella progettazione di sistemi informatici in grado di analizzare e comprendere il linguaggio umano (detto linguaggio naturale) e di elaborare risposte in tale linguaggio. In particolare, l'NLP studia i problemi riguardanti la generazione automatica e la comprensione del linguaggio umano, sia scritto che parlato. Quando si parla di *Natural Language* ci si riferisce a linguaggi nati spontaneamente nel corso della storia umana e usati dalle persone per comunicare (ad esempio, Italiano, Inglese, Cinese, etc); linguaggi, in ogni caso, diversi da quelli usati sui computer (Python, Java, C++).

All'interno del campo di ricerca dell'informatica, l'NLP ha suscitato fin da sempre molta curiosità ed attenzioni. A tal proposito, nel 1950 Alan Turing presenta un test chiamato *The Imitation Game* o *Test di Turing* (Figura 1.1), ideato per valutare il grado di sostitutività comunicativa tra uomo e macchina. Se una macchina passava il test, doveva essere considerata intelligente, dato che era risultata indistinguibile da un essere umano.

1.2 Metodi Ruled-Based e Statistical

I primi sistemi di *Natural Language Processing* erano basati sull'utilizzo di approcci *rule-based* e *statistical*.

Inizialmente, il primo metodo si sviluppa soprattutto grazie a Noam Chomsky e all'influenza che portò, attraverso le sue teorie sintattiche, nella linguistica e nelle grammatiche generative. L'ultime contrappongono le competenze (conoscenze linguistiche che consentono di produrre messaggi verbali) alle esecuzioni (messaggi verbali realizzati); la sola esperienza non è sufficiente per raggiungere una reale padronanza e consapevolezza del linguaggio se ad essa non si affiancano le competenze

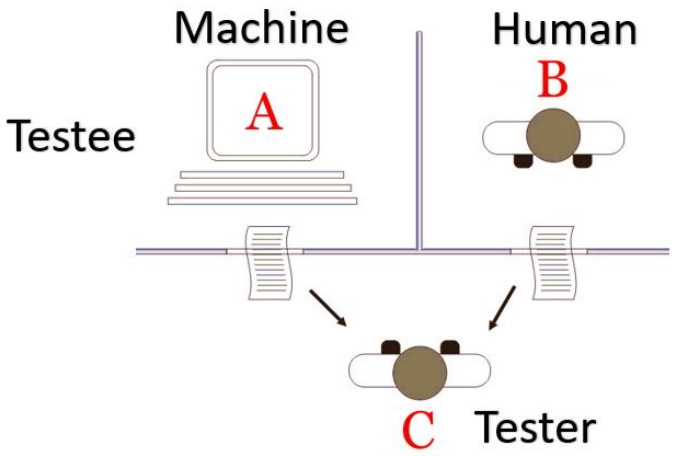

Figura 1.1. Imitation Game: il partecipante A (essere umano) comunica con B e C senza conoscerli. Se A dopo aver conversato con B e C non riesce a definire chi dei due sia una macchina e chi un uomo, allora la macchina supera il test di Turing

innate che gli esseri umani possiedono. L'approccio *rule-based* crea sistemi di elaborazione del linguaggio codificando a mano un insieme di regole, ossia definendo delle grammatiche successivamente tradotte in programmi comprensibili per il calcolatore. Queste regole e relazioni vengono classificati, secondo la linguistica, in *fonetica*, *morfologia*, *sintattica*, *semantica* e *pragmatica*. Grazie a questo metodo, è possibile ottenere degli ottimi risultati. Tuttavia, i programmi tendono a raggiungere un'alta complessità a causa delle numerose regole definite, anche se è doveroso specificare che questo problema di apprendimento è un ostacolo evidente solo per le macchine; l'uomo lo affronta facilmente con capacità innate durante i primi anni di crescita, in modo inconsapevole e naturale. Inoltre, si deve tenere conto della necessità di sviluppare algoritmi diversi per ognuna delle lingue in considerazione in un progetto, [6].

Tra la fine degli anni '80 e la metà degli anni '90, si assiste ad un cambiamento in cui, in contrapposizione ai metodi *ruled based*, gran parte della ricerca sull'elaborazione del linguaggio naturale si basa sull'apprendimento automatico. Questo metodo richiede l'uso dell'inferenza statistica per apprendere automaticamente tali regole attraverso l'analisi di grandi corpora[1] tipici del mondo reale. Diverse classi di algoritmi di apprendimento automatico sono state applicate alle attività di elaborazione del linguaggio naturale. Questi algoritmi prendono come ingresso un ampio set di "caratteristiche" generate dai dati di input. Alcuni dei primi algoritmi utilizzati sono gli alberi decisionali. Tuttavia, questa tipologia di algoritmo produceva regole rigide e, quindi, restituiva risultati simili ai sistemi *ruled based*. Col passare del tempo, però, la ricerca si è concentrata sempre più su modelli statistici, che prendono decisioni "morbide" e probabilistiche, basate sull'attribuzione di pesi di valore reale a ciascuna caratteristica di input.

[1] La forma plurale di corpus, è un insieme di documenti, libri, articoli, possibilmente con annotazioni umane o al computer

Gli approcci statistici sono stati favoriti dal progressivo incremento delle capacità computazionali dei calcolatori e dalla disponibilità di grandi quantità di dati. Questi metodi, definiti anche di Machine Learning e Deep Learning, hanno permesso, indubbiamente, di risolvere problemi complessi e ottenere risultati soddisfacenti in molte applicazioni, come question answering e le chatbot. Tali modelli hanno il vantaggio di poter esprimere la relativa certezza di diverse possibili risposte, piuttosto che di una sola, producendo, così, risultati più affidabili quando sono inclusi come componenti di un sistema più ampio.

I sistemi basati su algoritmi di apprendimento automatico presentano molti vantaggi rispetto alle regole prodotte manualmente. Infatti, le procedure di apprendimento automatico possono fare uso di algoritmi di inferenza statistica per produrre modelli robusti per input non familiari (ad esempio, contenenti parole o strutture che non sono mai state viste prima) e per input errati (ad esempio, con parole errate o parole omesse per errore). Generalmente, gestire tali input con regole scritte a mano, che prendono decisioni "morbide", risulta essere un'operazione estremamente difficile, soggetta ad errori e poco rapida.

I sistemi basati sull'apprendimento automatico delle regole, invece, hanno il vantaggio di poter essere resi più precisi con una semplice operazione: basta, infatti, fornire più dati in ingresso ad essi. Tale operazione, inoltre, richiede soltanto un relativo aumento del numero di ore di lavoro impiegate, senza che ciò comporti un significativo incremento della complessità del processo di elaborazione. Al contrario, per quanto riguarda i sistemi basati su regole scritte a mano, esiste un limite alla loro complessità oltre il quale esse diventano sempre più ingestibili.

1.3 Varie applicazioni

Le tecniche di elaborazione del linguaggio sono utilizzate in varie applicazioni ed in molti contesti. Uno dei più importanti scenari è il web, dove è presente un elevato numero di informazioni testuali, e dove è possibile sviluppare applicazioni che si servono dell'NLP per l'estrazione delle informazioni, per la ricerca di documenti e per la loro classificazione. Ulteriori sistemi che basano il loro funzionamento sull' NLP sono gli assistenti virtuali ed i chatbot. Le fasi che compongono un'applicazione di elaborazione del linguaggio naturale vengono identificate nelle seguenti attività:

- *Tokenizzazione*: La fase di tokenizzazione permette di individuare dei token, ossia delle sottosequenze significative di caratteri, che identificano le parole, i numeri, la punteggiatura ed altri elementi in una frase. La tecnica più semplice è la Whitespace Tokenizer, che individua i token attraverso i caratteri di spazio presenti in una sequenza di caratteri.
- *POS tagging*: Effettua un'analisi della Part Of Speech di ogni parola. Ciò significa che, per ogni token o parola, viene definita la categoria grammaticale a cui appartiene. Nella lingua italiana queste categorie sono: *Nome*, *Verbo*, *Aggettivo*, etc. Questa fase importante permette di effettuare una prima disambiguazione riconoscendo la categoria lessicale di un token e, quindi, riuscendo ad attribuire ad esso un significato all'interno di un contesto.

- *Stemming*: È un metodo di accorpamento di termini che sono varianti della stessa parola. Permette di individuare la parte invariante di una parola, ovvero la radice.
- *Lemmatization*: Questa attività è l'evoluzione dello *Stemming* ed ha l'obiettivo di riportare la parola nella sua forma base, non alla sua radice. A tal proposito, viene effettuata un'analisi del contesto della parola, e ciò offre la possibilità di ricercare la parola lemmatizzata in dizionari.

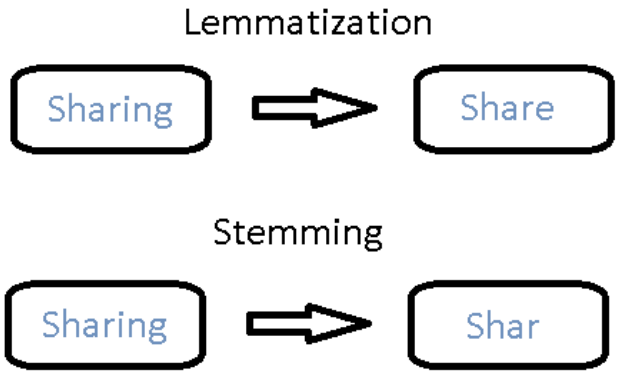

Figura 1.2. Rappresentazione della differenza tra *Lemmatization* e *Stemming*

- *Parsing*: L'analisi delle dipendenze (*dependency marking*) consente di estrarre la struttura della frase per capire quali termini sono in relazione tra di loro. Questa fase permette di identificare i sintagmi e di costruire l'albero sintattico, (Figura 1.3 *Parse tree*). Due approcci possibili sono i seguenti:
  1. *rule based*: richiede la conoscenza delle regole grammaticali della lingua considerata.
  2. *statistico*: richiede un corpus annotato rispetto alle dipendenze.
- *Named Entity Recognition (NER)*: Questa fase, conosciuta anche come *Entity Extraction*, si occupa dell'individuazione e della classificazione di entità predefinite. Queste ultime possono essere i nomi di personaggi famosi, luoghi, brand, organizzazioni, numeri, date, etc.
- *Stop Words Removal*: Si occupa di rimuovere parole di basso contenuto informativo, in quanto molto frequenti all'interno di frasi. Questi token possono essere, ad esempio, articoli oppure verbi ausiliari. Alcune delle tecniche applicate sono le liste di stop words con cui tenere traccia di questi token. In alternativa, si può effettuare una valutazione del valore informativo della parola attraverso la tecnica TF-IDF.

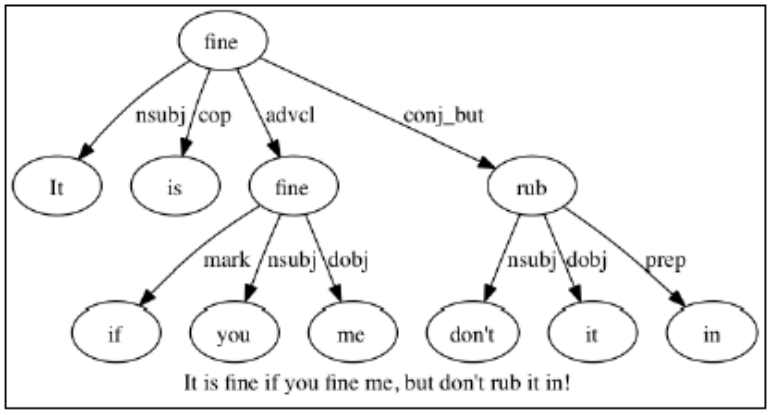

Figura 1.3. Rappresentazione della frase "It is fine if you fine me, but don't rub it in" in un Parse Tree con le relative categorie grammaticali

1.4 Ambiti di utilizzo

Il *Natural Language Processing* ha visto il suo utilizzo in diversi scenari applicativi. Alcuni dei più rilevanti sono riportati di seguito:

- *Sentiment analysis*: si occupa di costruire sistemi per l'estrazione di opinioni da dei testi di riferimento. Si basa sui principali metodi di linguistica computazionale e di text mining. L'analisi del sentiment si rileva utile in molteplici settori, dalla politica ai mercati azionari, dal marketing alla comunicazione, nell'ambito sportivo, dall'analisi dei social media alla valutazione delle preferenze del consumatore.
- *Question answering*: è un sistema di recupero automatico delle informazioni, progettato per rispondere alle domande che gli sono poste nel linguaggio naturale. Per fare ciò, adopera approcci di deep learning e tecniche di information retrieval (per l'individuazione della risposta in uno o più documenti), in grado di analizzare e classificare domande ed estrarre risposte in linguaggio naturale, interrogando basi di conoscenza strutturate di grosse dimensioni e/o collezioni di documenti testuali (anche il Web), [7].
- *Machine translation*: Per Machine Translation (MT) si intende la traduzione di un testo di partenza verso una lingua di arrivo da parte di un computer, senza l'intervento umano. Questi sistemi vengono normalmente impiegati per la traduzione di grandi quantità di informazioni e testi. La Machine Translation Neurale (NMT) è la tecnologia che, al momento, sta riscuotendo maggiore successo, poichè utilizza le reti neurali per processare i dati. Una caratteristica che porta molti vantaggi a tale applicazione è l'isolamento dei segmenti tradotti; tale attività consente di poter riutilizzare i segmenti stessi in ogni momento, con la certezza che saranno precisi e immediatamente disponibili in futuro, [5].
- *Automatic summarization*: Quest'applicazione permette di riassumere un contenuto testuale, preservandone le informazioni principali e la correttezza sintattica. Quest'ultimo è uno dei problemi più impegnativi nell'NLP, ma si rivela utile in

diversi contesti, come, ad esempio, nei motori di ricerca o nella generazione automatica di contenuti.

2

I Chatbot

Nel capitolo corrente si fornirà una panoramica sui chatbot e sulle loro caratteristiche.

2.1 Premessa

I chatbot sono dei software progettati per simulare una conversazione in linguaggio naturale con un essere umano all'interno di una piattaforma di messaggistica istantanea. La parola "chatbot" deriva dall'unione di due parole: "bot" (abbreviazione di robot) ad indicare un software in grado di automatizzare delle attività ripetitive svolte dall'uomo e "chat" (dall'inglese "dialogo") a rappresentare modalità di interazione tra il bot e l'utente. [3]

Questa interfaccia conversazionale viene utilizzata da aziende o enti per erogare dei servizi di dialogo, ma senza la necessità di un'interazione con personale umano. Infatti, i chatbot sono impiegati, ad esempio, in attività di supporto ai clienti, di risposta alle FAQ degli utenti e di esecuzione degli ordini a seguito di una richiesta di un utente.

2.2 L'evoluzione dei chatbot

Dalla formulazione del test di Turing ad oggi, i chatbot hanno avuto una lunga evoluzione, sia dal punto di vista tecnologico che dal punto di vista applicativo. Di seguito, verrà proposta una panoramica sulla loro evoluzione, da Eliza ai più recenti assistenti virtuali:

- *Eliza* (1966): ha rappresentato il primo chatbot della storia. È stato sviluppato da Joseph Weizenbaum per simulare uno psicoterapeuta rogersiano e viene ricordato come il primo programma ad aver mai superato il test di Turing. Nonostante ciò, non è possibile definirlo realmente intelligente, poichè non era capace di "pensare" da solo, ma si serviva di un database contenente funzioni e regole,

usate per rispondere alle domande dell'interlocutore. Eliza basava il suo funzionamento sulla riformulazione delle affermazioni del paziente stesso, facendo uso delle tecniche di pattern matching, [10].

- *Parry* (1972): il chatbot creato dallo psichiatra Kenneth Colby simulava il comportamento di un individuo paranoico e schizofrenico. Una caratteristica importante, che lo rendeva più complesso di Eliza, era la rappresentazione del modello del proprio stato mentale, il quale era variabile durante la conversazione in funzione dei propri livelli di paura o rabbia che aveva raggiunto.
  Inoltre, Parry fu il primo agente di dialogo a passare il test di Turing. Per realizzare tale verifica, furono sottoposte a degli psichiatri delle trascrizioni di interviste di pazienti unite a quelle di Parry. Come risultato si constatò una indistinguibilità tra quelle prodotte da veri pazienti e il chatbot, [12].
- *Jabberwocky* (1981): il programmatore Rollo Carpenter ha sviluppato il chatbot Jabberwacky e, nel 1997, lo ha lanciato per la prima volta su Internet. Tale chatbot era capace di utilizzare un linguaggio ricavato da milioni di interazioni online con persone. Grazie a ciò, proponeva un'innovazione rispetto alle precedenti soluzioni chatbot che, invece, basavano il recupero delle risposte da database statici. Lo scopo con cui fu creato Jabberwacky era di conversare in linguaggio naturale in modo divertente ed umoristico.
- *ALICE, Artificial Linguistic Internet Computer Entity* (1995): il progetto ALICE, proposto da Richard Wallace nel 1995, vinse il Loebner Prize negli anni 2000, 2001 e 2004. ALICE era un chatbot basato sul Natural Language Processing e programmato nel linguaggio AIML (Artificial Intelligence Markup Language) per dichiarare regole di corrispondenza in grado di collegare parole e frasi date in input dall'utente con categorie di argomenti.
- *Smarterchild* (2001): quando la startup ActiveBuddy Inc. presentò il chatbot SmarterChild, inizialmente fu pensato come un gioco di avventura basato sul linguaggio naturale. Esso aveva la possibilità di collegarsi a grandi quantità di informazioni in diversi domini di conoscenza; esso spaziava e ricercava informazioni riguardanti il meteo, le notizie e gli orari di eventi. Tale caratteristica ha permesso al chatbot di essere riconosciuto come un precursore dei più recenti assistenti vocali, cioè una macchina capace di fornire un servizio utile all'utente.
- *Watson* (2006): è un sistema di Intelligenza Artificiale sviluppato dalla IBM che, attraverso un'avanzata elaborazione del linguaggio naturale (rappresentazione della conoscenza, ragionamento automatico e tecnologie di apprendimento automatico) è capace di rispondere automaticamente a delle domande poste dagli utenti. La IBM, dopo che, nel 1997, il sistema Deep Blue sconfisse durante una partita a scacchi il campione Garry Kasparov, iniziò a cercare nuove sfide in cui far competere un'intelligenza artificiale. Questa ricerca portò allo sviluppo di Watson e alla sua partecipazione al famoso quiz statunitense Jeopardy!. In questa sfida, che ha visto il sistema di Intelligenza Artificiale gareggiare contro i campioni Ken Jennings e Brad Rutter, Watson ne è uscito vincitore. Tale vittoria è stata favorita da alcuni importanti fattori; i più rilevanti sono stati i suoi avanzati algoritmi di analisi linguistica e i 4 terabytes di informazioni nel suo database che, anche in assenza di un accesso alla rete, fornivano una vasta conoscenza di base.
- *Siri* (2010): è un assistente vocale intelligente di grande successo sviluppato dalla

Apple. Attualmente è ancora parte integrante dei sistemi iOS e nasce con l'obiettivo di supportare gli utenti rispondendo a domande, cercando informazioni in rete e interfacciandosi direttamente agli altri servizi di base dello smartphone (agenda, rubrica, calendario, messaggistica, etc.).

- *Alexa* (2014): è l'assistente vocale di Amazon basato su cloud ed è l'intelligenza che alimenta Amazon Echo e altri dispositivi compatibili. L'idea è quella di un servizio in continua evoluzione, che può essere continuamente arricchito e che, proprio tramite le interazioni con l'utente, è in grado di apprendere nuove capacità, nuovi tipi di risposte e nuove sfumature dialettiche in grado di aumentare l'empatia tra la macchina e la persona. Alexa è progettata per poter essere inserita in un ecosistema complesso, come, per esempio, i sistemi domotici, essendo essa in grado di gestire e controllare altri dispositivi intelligenti tramite appositi *skill* (software plugin che permettono di riconoscere comandi vocali definiti da aziende terze).

2.3 Le tipologie di chatbot

Per quanto riguarda l'architettura su cui viene basata la progettazione dei chatbot, è possibile effettuare una prima distinzione in due macrocategorie: *rule-based* e *corpus-based*. Questa classificazione ha origine dalle omonime tecniche di NLP utilizzate dai chatbot per dialogare in linguaggio naturale. Di seguito verranno esaminate le caratteristiche delle varie tecniche.

2.3.1 Rule-based chatbot

I chatbot basati su regole fanno uso della tecnica del pattern matching per formulare delle risposte basandosi su un complesso insieme di regole predefinite in fase di sviluppo. Il pattern matching permette di rilevare informazioni significative nei messaggi inviati da un interlocutore.

I chatbot, sviluppati sulla base di questa tecnologia, riescono a rispondere adeguatamente solo all'interno del dominio applicativo in cui vengono progettati, ma non al di fuori di esso; quindi, la bontà di questi sistemi rule-based è fortemente legata alla complessità delle conversazioni che il chatbot dovrà intrattenere, poichè maggiore è tale conversazione e maggiore sarà la complessità delle regole da definire.

La tecnica rule-based è una delle prime ad essere ideata; tant'è che i primi chatbot nella storia, come Eliza o ALICE, furono sviluppati secondo questo metodo. Tuttavia, considerando l'elevata complessità di questa soluzione, negli anni, si è cercato di abbandonare questo approccio, anche se ancora è funzionale in contesti d'uso semplici.

2.3.2 IR-based chatbot

Gli IR-based chatbot, come i rule-based chatbot, fanno parte dei *modelli Retreival-based*. Questi sistemi non generano nuove risposte, ma si limitano a selezionarne una da una lista predefinita.

Nel momento in cui un utente effettua una domanda, il chatbot identifica la risposta migliore rispetto a quelle che custodisce nel corpus (chiamate *candidate responses*). Per rispondere all'utente, essi scelgono all'interno del corpus la risposta appropriata mediante tecniche di IR che analizzano la similarità tra la conversazione in corso e quelle immagazzinate. Questo calcolo è spesso implementato con la funzione coseno applicata sulla frequenza delle parole (TF-IDF[1]) o su word embeddings[2]. Questa tecnica ha visto la sua implementazione nel rinomato Cleverbot, un'evoluzione di Jabberwacky in grado di estendere la sua base di conoscenza andando a memorizzare lo storico delle conversazioni e apprendendo ogni volta.

2.3.3 Generative chatbot

I chatbot di tipo generative-based sono caratterizzati dalla generazione dinamica della risposta che avviene ad ogni nuova interazione con l'utente. La loro più comune architettura è basata sulle reti neurali di tipo *sequence-to-sequence o Seq2Seq* (Figura 2.1), costituite da due RNN (meglio se LSTM), che fungono una da encoder e l'altra da decoder. L'input dell'utente (ossia la domanda dell'utente) passa per l'encoder che lo elabora, parola per parola, e ne codifica uno stato, che rappresenta il contesto. Questo vettore generato dalla prima rete verrà, poi, passato al decoder, che lo utilizza come stato iniziale da cui generare la risposta.

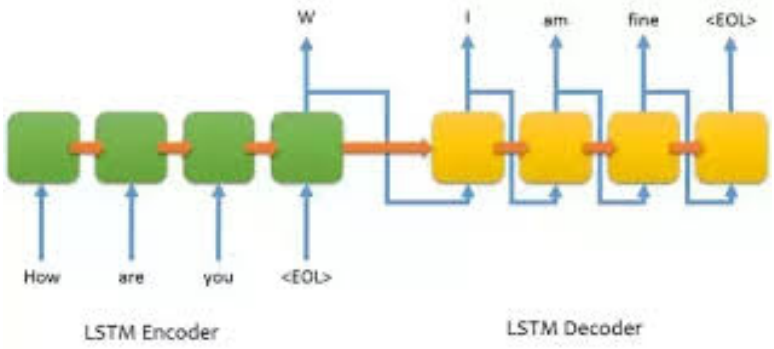

Figura 2.1. Un modello Seq2Seq applicato agli agenti di dialogo

Uno dei maggiori vantaggi che questa architettura permette di avere è l'assenza di un lista di risposte predefinite da cui derivare quella ottima in funzione della domanda. Generando automaticamente le risposte, questa soluzione non necessita nemmeno delle regole, bensì solo delle informazioni relative allo storico delle conversazioni usate nella fase di training della rete neurale.

[1] TF-IDF è una funzione utilizzata in information retrieval per misurare l'importanza di un termine rispetto ad un documento o ad una collezione di documenti. Tale funzione aumenta proporzionalmente al numero di volte che il termine è contenuto nel documento.

[2] La word embeddings è una tecnica impiegata in NLP per rappresentare le parole e le frasi come vettori di numeri reali.

L'utilizzo di questa tecnica comporta, però, altri tipi di problematiche. La principale è la difficolta di realizzazione, sia in termini di costi che di performance. Con questa affermazione si intende la possibilità di generare delle risposte grammaticalmente scorrette, oppure la possibilità di rispondere in modo diverso ad una stessa domanda posta due volte. Inoltre, vi sono delle problematiche legate all'utilizzo di una rete neurale. La necessità di utilizzare un insieme di dati di grandi dimensioni è indispensabile, in quasi tutti i casi, per riuscire a generare delle risposte accurate.

2.3.4 Open e Closed domain

Un ulteriore modo di classificare i chatbot è in base al loro dominio applicativo.

Open Domain

I chatbot che fanno parte di questa tipologia sono in grado di rispondere senza nessun limite negli argomenti e nelle tematiche che un interlocutore può chiedere. Questi sistemi sono progettati per rispondere a domande senza un contesto definito, oppure di un obiettivo voluto.

La realizzazione di questi sistemi è molto complessa; essi si possono ottenere solo attraverso le tecniche generative-based. Non sarebbe possibile avere lo stesso risultato con sistemi rule-based, dove le regole sono predefinite e fanno riferimento a specifici contesti di lavoro del bot.

Closed Domain

I chatbot closed domain sono capaci di svolgere le loro attività solo per uno specifico dominio o problema. Questa tipologia viene sviluppata per rispondere alle domande rivolte al solo dominio di competenza. Ad esempio, un chatbot realizzato per la gestione degli ordini di un ristorante sarà in grado di rispondere alla richiesta "vorrei ordinare delle pizze da asporto", ma non a "come è il tempo oggi?". Il chatbot Medici-bot, che verrà illustrato nel corso di questa trattazione, è di tipo retrieval-based e closed domain. Come verrà approfondito nei capitoli successivi dell'elaborato, il chatbot sarà responsabile di rispondere alle domande rivolte dagli utenti selezionando una delle risposte predefinite.

2.3.5 Task-oriented chatbot

I chatbot *task-oriented* sono una tipologia di agenti di dialogo progettati con l'obiettivo di portare a compimento un'attività richiesta da un utente. Fanno parte di questa famiglia gli assistenti digitali come Siri, Google Now, Cortana e Alexa, i quali ci permettono di svolgere delle attività di diverso genere. É possibile chiedere informazioni (indicazioni stradali, previsioni metereologiche, etc.), far svolgere azioni ad altre applicazioni (chiamate, messaggi, memorizzazione promemoria, etc.) o interagire con altri oggetti interconnessi (smart device).

Architettura

Questi chatbot generalmente sono progettati secondo la seguente struttura a tre componenti:

1. *Natural Language Understanding (NLU)*: questo modulo è uno strumento di elaborazione del linguaggio naturale utile per la comprensione del messaggio dell'utente facendo uso della classificazione degli intenti (Intent classification) e dello slot filling. Gli intent possono essere visti come le intenzioni dell'utente, ovvero le tipologie di richieste ricevibili e comprensibili dall'agente di dialogo. Gli slot, invece, corrispondono alle variabili del chatbot di cui è necessario conoscere il valore per riuscire a formulare una risposta. Prendendo in esempio la frase: "*Sto cercando un ristorante messicano in centro città*", l'intent ottenuto costituisce l'intenzione di prenotare un ristorante, mentre gli slot che dovranno essere riconosciuti sarebbero quelli relativi alla tipologia di ristorante e alla sua posizione. Tutto ciò può essere osservato anche in Figura 2.2.

Figura 2.2. Risultato dell'attività di NLU: Intent classification e Slot filling

2. *Dialog Management (DM)*: si divide in due componenti, dialog state tracker e dialog policy. Il primo si occupa di mantenere lo stato della conversazione, aggiornandolo ad ogni messaggio dell'utente, comprendendo anche gli slot ottenuti in fase di NLU. Il secondo componente, invece, sulla base dell'andamento della conversazione, sceglie la prossima azione da compiere, cercando di capire quale sia la migliore per il completamento del task nel minor numero possibili di turni.
3. *Natural Language Generation (NLG)*: è il processo di generazione della risposta; esso costruisce una frase in linguaggio naturale contenente le informazioni che si vogliono comunicare all'utente. Spesso tale componente è di tipo rule-based; in questi casi esso dispone di una serie di frasi template, da completare con le informazioni opportune.

2.4 Scenari d'uso

I chatbot, oggi giorno, sono utilizzati in molti ambiti applicativi e con attività ed obiettivi molto diversi tra loro. Secondo uno studio presentato dalla Myclever Agency sono tre le categorie in cui essi possono essere classificati, [9]:

1. *Commerce bots:* all'interno di una piattaforma di e-commerce, essi consentono di navigare, selezionare e acquistare prodotti senza mai abbandonare la finestra di messaggistica. In questo scenario, i chatbot hanno permesso di modificare l'esperienza utente, nei processi di acquisto sui negozi online, introducendo il concetto di commercio conversazionale. Alcuni esempi noti di aziende che hanno sviluppato questa soluzione sono H&M e Shopspring; quest'ultima è stata, anche, la prima ad integrare tale innovazione nel proprio sito.
2. *Customer service bots:* sono dei servizi resi disponibili in ogni istante della giornata ed hanno il compito di rispondere alle richieste dei clienti (offrendo informazioni e rispondendo ai reclami) nel minor tempo possibile. Un esempio reale riguarda la compagnia aerea KLM, la quale è stata la prima compagnia aerea a offrire questo servizio, sulla piattaforma Facebook Messenger, per la gestione dei check-in dei biglietti dei passeggeri.
3. *Content bots:* forniscono informazioni e avvisi in tempo reale su determinati argomenti selezionati dai singoli clienti. In verità, in molti casi, non serve nemmeno una esplicita scelta del cliente, ma il chatbot è in grado di promuovere dei contenuti o prodotti in base alle informazioni personali che ricava dal cliente (ad esempio, il *The Guardian News Bot*) e può suggerire articoli a seconda del topic preferito dell'utente).

Oltre a questa classificazione dei chatbot, è possibile categorizzare gli stessi secondo le applicazioni nelle diverse funzioni aziendali, [1]. Le aziende utilizzano i chatbot sia nei loro processi interni, sia nei processi esterni di interazione con il cliente. Di seguito sono riportati i principali campi applicativi all'interno delle aziende.

Servizi

All'interno di questà categoria, i chatbot sono utilizzati per le seguenti attività:

- *Assistenza al cliente post vendita*: sono utilizzati per assistere il cliente durante la gestione del prodotto o del servizio acquistato. Nei settori bancari, assicurativi, finanziari e delle telecomunicazioni i chatbot sono largamente diffusi; questo è proprio il settore in cui essi mostrano il più diffuso ambito di utilizzo.
- *Corporate Knowledge*: questi chatbot sono in grado di rispondere alle domande sull'azienda che vengono poste dal personale o da figure esterne. (ad esempio, "In quale piano lavora l'ing. Luca Rossi?").

Sales & Marketing

All'interno di questà categoria, i chatbot sono utilizzati per le seguenti attività:

- *Shop Assistant*: i chatbot progettati con questa finalità forniscono ai clienti le informazioni riguardanti i prodotti o servizi che l'azienda può offrire. Solitamente, le informazioni riguardano le caratteristiche, la disponibilità in magazzino o i tempi di consegna del prodotto.
- *Guida all'acquisto*: attraverso le informazioni personali che un'azienda (e quindi il chatbot) possiede sui clienti, e identificando le sue intenzioni di acquisto, il chatbot è capace di suggerire il prodotto o il servizio più adatto ai suoi interessi.
- *Brand reputation*: questi chatbot hanno lo scopo di mostrare un'immagine positiva del brand, fornendo ai clienti le informazioni relative all'azienda e a ciò che offre.
- *Supporto alle vendite*: questi chatbot sostengono le attività di vendita del personale aziendale, ad esempio fornendo a loro i dettagli su prodotti o servizi.

Prodotto, Risorse Umane e Ricerca & Sviluppo

All'interno di questà categoria, i chatbot sono utilizzati per le seguenti attività:

- *Prodotto*: i chatbot di questa categoria sono, principalmente, degli assistenti virtuali, poichè sono sviluppati all'interno di un prodotto. Essi sono progettati per gestire da remoto, ad esempio, i dispositivi smart di un'abitazione domotica (ad esempio, illuminazione, regolatori di temperatura e videosorveglianza).
- *Risorse Umane*: i chatbot, progettati con questa finalità, vengono utilizzati, appunto, per supportare le attività di recruiting e di gestione del personale. Possono fornire informazioni contrattuali ai dipendendenti, informazioni sui giorni di ferie maturate durante il periodo lavorativo, o anche rispondere alle richieste su eventuali posizioni aperte nell'azienda.
- *Ricerca e Sviluppo*: i chatbot supportano il personale addetto ai sistemi informatici nelle attività come la gestione del data center o sull'identificazione della fonte di un dato.

2.5 Limitazioni dei chatbot

L'enorme potenziale di questi sistemi è chiaro, ma ovviamente sono presenti anche degli svantaggi e limitazioni. La prima è legata a come un chatbot si identifica durante un'interlocuzione con un utente, dal momento che esso può optare tra il presentarsi come un'intelligenza artificiale oppure come una vera persona.

Detto ciò, il problema sussiste poichè i clienti hanno aspettative diverse in base all'interlocutore presente; quindi, nel momento in cui un chatbot si presenta come umano, i clienti si aspettano un certo livello di conversazione e comprensione da esso. Inoltre, sempre presentandosi come umani, le persone potrebbero esternare i loro sentimenti e la loro emotività nella conversazione ed i chatbot potrebbero non riuscire a gestire tale dialogo, fornendo un esperienza negativa al cliente.

Tuttavia l'imitazione del comportamento umano, concetto diverso dall'impersonificazione, è una caratteristica necessaria per poter offrire un servizio migliore. Inoltre, un utente, si aspetta da un chatbot risposte rapide e accurate ad un livello tale che un essere umano non potrebbe offrire.

Nonostante i molti usi, i benefici per i consumatori e le aziende che attualmente utilizzano questo servizio, l'adozione di massa dei sistemi di conversazione automatizzata nei prossimi anni non è una conclusione scontata. Allo stato attuale, i chatbot presentano, ovviamente, ancora i limiti propri dell'Intelligenza Artificiale. Se grazie al machine learning il tasso di errore dei chatbot diminuisce con l'incremento del loro uso, capita, tuttavia, che alcuni sistemi chatbot fraintendano le domande, forniscano le risposte sbagliate e non raggiungano l'obiettivo. Tutto ciò, anche se può avvenire con frequenza molto bassa, può portare i clienti a giudicare negativamente le loro esperienze con i bot. Inoltre, se ci si aggiungono tutti quei consumatori che non sono intenzionati assolutamente ad abbandonare le interazioni umane, possiamo vedere, come già anticipato, che i chatbot non hanno una evoluzione futura scontata.

Infine, è possibile, per ovvi motivi, osservare che il servizio reso da un chatbot è privo della capacità di stabilire una vera connessione con il cliente, il quale sentirà distacco nella conversazione. Tuttavia, una richiesta che viene presa in carico da un chatbot può essere conclusa con più efficienza ed efficacia rispetto ad una richiesta portata a termine da una persona. Date le caratteristiche esposte in precedenza, l'utilizzo dei chatbot presenta chiaramente sia dei vantaggi sia dei limiti rispetto all'interazione umana, ma il concetto fondamentale è che le due cose non si possono autoescludere: i bot possono essere utilizzati all'occorrenza, ad esempio per rispondere a domande brevi e ripetitive (ad esempio, le FAQ di un sito o servizio), prevedendo l'intervento umano ad hoc in un successivo momento.

Anche se i chatbot offrono miglioramenti concreti a livello di user experience, per il momento possono solo supportare, ma non rimpiazzare del tutto, la presenza umana.

3

Il framework RASA

In questo capitolo si parlerà del framework RASA, il quale è stato scelto in questo progetto come piattaforma per lo sviluppo del chatbot medici-bot

3.1 Architettura

Rasa è un framework open source in Python per la creazione di un chat-bot basato sul machine learning supervisionato.

Esistono altri framework in commercio, come DialogFlow (Google), Lex (Amazon) oppure Luis (Microsoft), ma la scelta è ricaduta su Rasa soprattutto per un motivo principale; ovvero la possibilità di utilizzare un prodotto open-source per il quale si possa avere accesso ai sorgenti del software e modificarlo in base alle esigenze.

L'architettura di Rasa si compone di due moduli chiamati *Rasa NLU* (Natural Language Understanding) e *Rasa Core*, i quali verranno approfonditi nei paragrafi successivi del capitolo. Poichè, in primo luogo è bene, attraverso lo schema in Figura 3.1, descrivere l'architettura con la quale Rasa processa i messaggi (domande) dell'utente, [2]:

1. *Message In* ⇒ *Interpreter*: in questa prima fase, il messaggio inviato dal cliente viene ricevuto dall'interpreter, che costituisce una componente di Rasa NLU. Il suo compito è quello di identificare l'intent (l'intento o l'intenzione che viene espressa nel messaggio) e di estrarre le entità presenti nella domanda;
2. *Interpreter* ⇒ *Tracker*: il Tracker è l'oggetto che tiene traccia dello stato della conversazione; esso riceve le informazioni che un nuovo messaggio è arrivato;
3. *Tracker* ⇒ *Policy*: questo componente riceve lo stato del tracker;
4. *Policy* ⇒ *Action*: il componente Policy sceglie qual è l'azione da intraprendere anche in base all'ultimo messaggio ricevuto;
5. *Policy* ⇒ *Tracker*: l'azione scelta viene conservata nel tracker;
6. *Policy* ⇒ *Message Out*: il messaggio viene rispedito all'utente.

Le componenti di Rasa precedentemente descritte sono in realtà contenute in due moduli distinti: Rasa NLU e Rasa Core; i due moduli possono anche essere utilizzati in modo indipendente l'uno dall'altro.

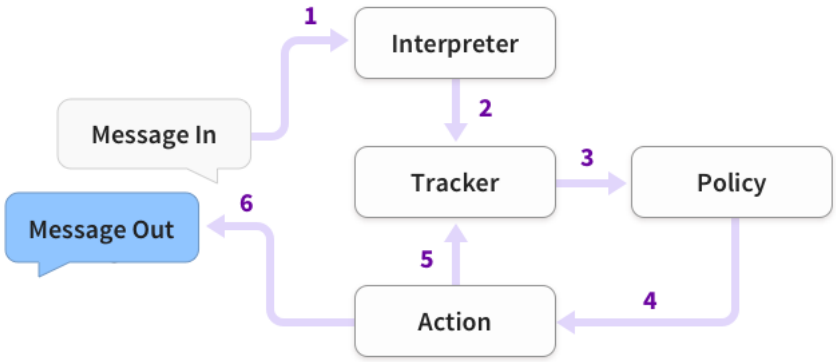

Figura 3.1. Schema di elaborazione dei messaggi in Rasa

3.2 Rasa NLU

Rasa NLU, è il modulo di Rasa che si occupa della comprensione del messaggio inviato dall'utente in linguaggio naturale.

Principalmente, le attività che svolge sono la classificazione degli *Intent* e l'estrazione delle *Entity* (entità, ovvero le variabili del chatbot) dal messaggio attraverso l'utilizzo di un modello generato secondo le tecniche di machine learning. Dunque, per svolgere le attività successive, è necessario addestrare un modello definendo fin dall'inizio *Intent* ed *Entity* che tale modello dovrà riconoscere. Si avrà, altresì, bisogno di un dataset di frasi già annotate secondo una sintassi riconosciuta dal modulo NLU. Segue un esempio di sintassi:

```
1       puoi dirmi i [pediatri](object\_type:medico) che lavorano in [via 1. Vanvitelli](indirizzo) [64 INT. 14](civico
            )?
```

Osservando l'esempio si può notare che i segmenti della frase marcati corrispondono alle *Entity*; queste ultime serviranno per addestrare il modello a riconoscere le parole o i segmenti della frase importanti all'interno dello scenario di utilizzo.

La comprensione del testo, però, segue una pipeline definita dagli sviluppatori e stabilisce la sequenza e i componenti che verranno utilizzati per processare il testo prima e durante la creazione del modello. All'interno della pipeline, tra i tanti componenti, vengono definiti anche i moduli dell'*Intent Classifier* e dell'*Entity Extraction*, che si occupano, per l'appunto, delle attività descritte in precedenza. Dunque, seguendo l'intera pipeline, viene, infine, restituito un modello ottenuto dal dataset di frasi annotate.

3.3 Rasa Core

Rasa Core è il modulo che si occupa della gestione del dialogo e permette di creare delle vere conversazioni scegliendo le risposte secondo i modelli creati durante le fasi di addestramento.

Dopo le attività di NLU svolte dal modulo descritto in precedenza (sulla base dell'*Intent* e delle *Entity* ricavate), Rasa Core ha il compito di decidere le azioni che il chatbot deve portare avanti in base al messaggio inviato dall'utente. Le *Policy* con cui il modulo sceglie le azioni da attivare a seguito di una richiesta sono determinate secondo dei modelli probabilistici (utilizzando, ad esempio, le reti neurali LSTM). Spetta, invece, allo sviluppatore definire le azioni di risposta ad un messaggio utente. Queste ultime possono essere basate su template di messaggi predefiniti, oppure è possibile implementare delle risposte personalizzate, grazie all'opportunità di collegare una base di dati da cui attingere informazioni sul dominio applicativo (utilizzo di una *knowledge base*).

3.4 Panoramica sulle componenti di RASA

Affinchè il chatbot sia in grado di dialogare è necessario definire vari componenti e file per la configurazione, [4].

Il principale file è `domain.yml`. Tale file presenta la definizione del dominio del bot; al suo interno vengono salvati gli elenchi delle risposte, le azioni relative a queste ultime, gli intenti della conversazione, le variabili del chatbot e, anche, tutte le altre componenti utili per riconoscere e rispondere ai messaggi. Di seguito verranno illustrati i componenti che formano il dominio del bot.

Intent

Gli *intent* corrispondono alle intenzioni dell'utente all'interno del dialogo, ovvero costituiscono le tipologie dei messaggi o le richieste che egli può effettuare. Se si prende in considerazione l'esclamazione "Ciao", questa ricade all'interno dell'intent "saluto".

Quando il bot analizza la frase in ingresso, cerca di identificarne lo scopo, riconducendolo a uno degli intent definiti all'interno del file `domain.yml`. Per fare ciò utilizza un modello generato da un algoritmo di machine learning supervisionato che si avvale degli esempi forniti dentro un file chiamato `nlu.md`.

Entity

Le entity rappresentano dei segmenti, all'interno delle frasi, che il chatbot deve essere in grado di riconoscere poichè contengono un'informazione utile per ogni specifico intent. Ad esempio, nella frase: "Sto cercando un ristorante giapponese", "giapponese" è l'entity, ovvero una specifica dell'intent "restaurant_research". Non è detto, però, che ogni intent preveda delle entity. Infatti, se si prende in considerazione il messaggio "Buongiorno", che fa parte dell'intent "saluto", non vi è la presenza di

alcun entity. Al contrario, è possibile che vi siano più entity specificate all'interno di un messaggio.

Le entity vengono rilevate nelle frasi in ingresso attraverso diversi componenti che sono contenute nella pipeline stabilita nel file `config.yml`.

Component	Requires	Model	Notes
`CRFEntityExtractor`	sklearn-crfsuite	conditional random field	good for training custom entities
`SpacyEntityExtractor`	spaCy	averaged perceptron	provides pre-trained entities
`DucklingHTTPExtractor`	running duckling	context-free grammar	provides pre-trained entities
`MitieEntityExtractor`	MITIE	structured SVM	good for training custom entities
`EntitySynonymMapper`	existing entities	N/A	maps known synonyms

Figura 3.2. Caratteristiche degli *entity extractor* che RASA permette di usare

Slot

Gli slot rappresentano la "memoria" del chatbot e permettono di salvare i valori delle entity, nonchè altri importanti parametri tramite cui è possibile di tracciare il filo logico all'interno di un dialogo.

Alla richiesta "Vorrei sapere gli orari di domani del dr Rossi", Il bot potrebbe ribattere "Ho trovato molti risultati, potresti dirmi il suo nome o il comune del suo ambulatorio?". A questo punto, l'utente risponderà alle richieste del bot, che salverà le informazioni ricevute negli slot specifici. Da questo esempio si deduce che gli slot rappresentano i valori che permettono alla conversazione di mantenere il filo del discorso.

Template

I template sono semplici frasi che il bot può utilizzare per rispondere a specifici intent. Ogni template, per convenzione, ha un nome rappresentato nel formato "<utter_nome_template>". Inoltre, ad un singolo template possono corrispondere più risposte, come nell'esempio del template "<utter_greet>" al quale si possono associare "Salve" e "Bentornato": il bot sceglierà in modo casuale una tra le risposte fornite. È importante aggiungere che i template sono risposte statiche, dunque non modificabili in stato di esecuzione.

Action

Le *Actions*, definite all'interno del file `actions.py`, costituiscono una lista delle azioni che il bot può compiere in risposta ad un messaggio dell'utente.

Più precisamente, un'azione è una funzione che il chatbot esegue al manifestarsi di uno o più intent. Tale azione può eseguire una serie di istruzioni anche complesse, come, ad esempio, recuparare i dati da un database, richiedere un servizio attraverso delle API, e tante altre possibili attività che si concludono con un return di una stringa contenente la risposta da fornire.

A differenza dei *template*, che sostanzialmente si limitano ad eseguire un return di una stringa statica, senza alcuna logica aggiuntiva, le *action* restituiscono una stringa generata in modo dinamico. Per necessità architetturali, questi due componenti sono entrambi elencati nel `domain.yml` sotto la voce *actions*, anche se le *action*, per convenzione, vengono dichiarate nel formato action_<nome_azione>".

All'interno del file `actions.py`, ogni azione viene associata a una classe ("class `ActionNomeAzione`") per la quale è necessario definire almeno due metodi: il primo è `name(self)`, il quale esegue un semplice return di una stringa contenente il nome dell'azione (lo stesso nome che abbiamo definito in `domain.yml`); il secondo è il metodo `run (self, dispatcher, tracker, domain)` dentro al quale viene scritto il codice da eseguire quando l'azione viene richiamata dal bot. I tre parametri in ingresso rappresentano, rispettivamente, il *dispatcher*, che permette di mandare messaggi di risposta all'utente, il *tracker*, che tiene in memoria l'intent e l'entity dell'input, e, infine, il *domain*, che rappresenta il percorso del file `domain.yml`.

Le azioni vengono eseguite attraverso un server chiamato `action_endpoints`, il quale viene specificato nel file `endpoints.yml`. Esso viene eseguito di default sull'indirizzo locale `localhost:5055/weebhook`. Dunque, affinché il bot possa eseguire le azioni personalizzate è necessario far partire il server weebhook attraverso il comando "`rasa run actions`".

Story

Le *Story* sono la rappresentazione dei dialoghi che possono intercorrere tra il chatbot ed un utente; esse vengono definite per favorire l'addestramento del modello, che si occupa del dialog management, a scegliere la migliore azione con cui il bot deve rispondere in un dato momento della conversazione.

Le storie vengono definite nel file `stories.md` e vengono rappresentate come uno scambio di messaggi tra gli utenti (espressi come *intent*) e il bot (risposta espressa attraverso l'*action)*. Di seguito è riportato un esempio della sintassi appena descritta:

```
## Happy path 1
* greet
  - utter_greet
* query_knowledge_list
  - action_query_list
* goodbye
  - utter_goodbye
```

Figura 3.3. Esempio di Story

Nell'esempio riportato sopra, la prima riga identifica il nome della storia, le righe che iniziano col carattere `*` rappresentano gli *intent*, e, infine, quelle che iniziano

col carattere - sono le action o i template con cui il bot risponde alle richieste dell'utente.

Pipeline NLU

La pipeline definisce l'insieme dei componenti che hanno l'obiettivo di processare, in modo sequenziale, il testo ricevuto in ingresso. Le attività delineate nella pipeline consentendo, quindi, la classificazione degli intent e l'estrazione delle eventuali entità. Con Rasa sono già disponibili tre pipeline predefinite:

- `pretrained_embeddings_convert`: viene consigliata nel caso in cui le domande fossero brevi e si disponesse di un dataset in inglese e di dimensioni ridotte. Un punto di forza di questa scelta è dovuto al fatto che non tratta in modo indipendente ogni parola del messaggio dell'utente, ma crea una rappresentazione vettoriale contestuale per la frase completa.
- `pretrained_embeddings_spacy`: viene consigliata, come nel caso precedente, nel caso in cui non si disponga di una grande quantità di dati per l'addestramento; essa permette, comunque, di avere dei modelli in grado di rappresentare la similarità tra due parole diverse. Ad esempio, se un utente chiedesse "mi puoi dare due mele", questo modello riesce a mappare la richiesta, la quale non è gestita dal chatbot, nell'intent "ottieni_pere" poichè rileva una similarità tra le due parole.
- `supervised_embeddings`: questa pipeline non utilizza un modello specifico che tiene conto della lingua; quindi funziona con qualsiasi lingua che è possibile tokenizzare (con questo termine si intende il processo di riduzione della frase in segmenti delimitati da caratteri speciali). Inoltre, questo modello viene consigliato nel caso in cui si possieda un dataset con molti esempi.

Sebbene queste pipeline siano consigliate e, generalmente, ben funzionanti per ogni tipo di chatbot, è possibile personalizzarle definendo ed integrando i componenti che si desiderano, ed anche creandone di nuovi. In questo caso, le componenti più importanti da considerare in una pipeline sono l'estrattore delle entità e il classificatore degli intent, che vanno allenati per lo specifico dominio.

4

MediciBot: Analisi dei requisiti e generazione dataset di training

In questo capitolo si parlerà del processo di analisi dei requisiti a partire dalle specifiche fornite dagli stakeholder del CSI Piemonte. Verranno, inoltre, approfondite le funzionalità richieste, le tecnologie usate e verranno illustrate delle osservazioni implementative.

4.1 Analisi dei requisiti

Il CSI Piemonte, nell'ambito dello sviluppo di servizi legati al cittadino e di soluzioni di analytics, ha ritenuto strategico avviare dal 2018 un nucleo di progetti che si avvalessero delle tecniche di machine learning applicate alla realizzazione di soluzioni di tipo conversazionale. Tra queste, in particolare, i chatbot vengono considerati in rapida diffusione tra i diversi ambiti applicativi.

La raccolta dei requisiti si è basata sulle specifiche fornite nel corso di più incontri avvenuti con i responsabili dei servizi informatici legati alla sanità del CSI Piemonte. Nel corso di questi colloqui è stata presentata la richiesta di creazione di un chatbot che fosse in grado di rispondere ai quesiti relativi ai medici e ai pediatri della regione Piemonte.

Il primo requisito necessario era rappresentato dalla capacità del chatbot di fornire informazioni sui medici, sugli ambulatori e sugli orari di visita (entità che definiranno, successivamente, le tipologie di "oggetti" che sarà possibile ricercare) rispondendo a due tipologie di domande che gli utenti ponevano in linguaggio naturale:

- Prima tipologia: richiesta di uno o più medici che soddisfino una condizione di ricerca, con la possibilità di ricevere informazioni riguardanti anche gli ambulatori o gli orari di visita. Chiameremo tale tipologia "*query_list*, in riferimento al nominativo dell'*intent*" che rappresenta.

  Ad esempio, *"Mi puoi dire i medici che operano ad Asti?"*

- Seconda tipologia: richiesta di un attributo di un medico, di un ambulatorio o di un orario (specificando un campo identificativo dell'oggetto ricercato). Chiameremo tale tipologia "*query_attribute_of*".

Ad esempio, *" Vorrei sapere il telefono del dr Luca Rossi"*

Gli stakeholder, oltre alle tipologie di domande a cui il bot dovrà essere capace di rispondere, hanno fornito due scenari di conversazione da rispettare.

In Figura 4.1 è rappresentato il primo dialogo composto da tre fasi. La prima è la richiesta del numero di telefono di uno specifico dottore da parte dell'utente (la domanda è del tipo *query_attribute_of*). Come risposta (seconda fase), il chatbot può fornire diverse informazioni in base al numero di risultati che possiede. Se le informazioni in suo possesso non risultano univoche, esso chiede all'utente di specificare un altro attributo relativo al dottore che sta ricercando. In questo modo il dialogo termina nel momento in cui il bot raggiunge un risultato unico (si ritiene risultato unico anche l'eventualità che il dottore sia in possesso di più cellulari per lo stesso ambulatorio o studio), o nel caso in cui non trovi neanche una corrispondenza (terza fase).

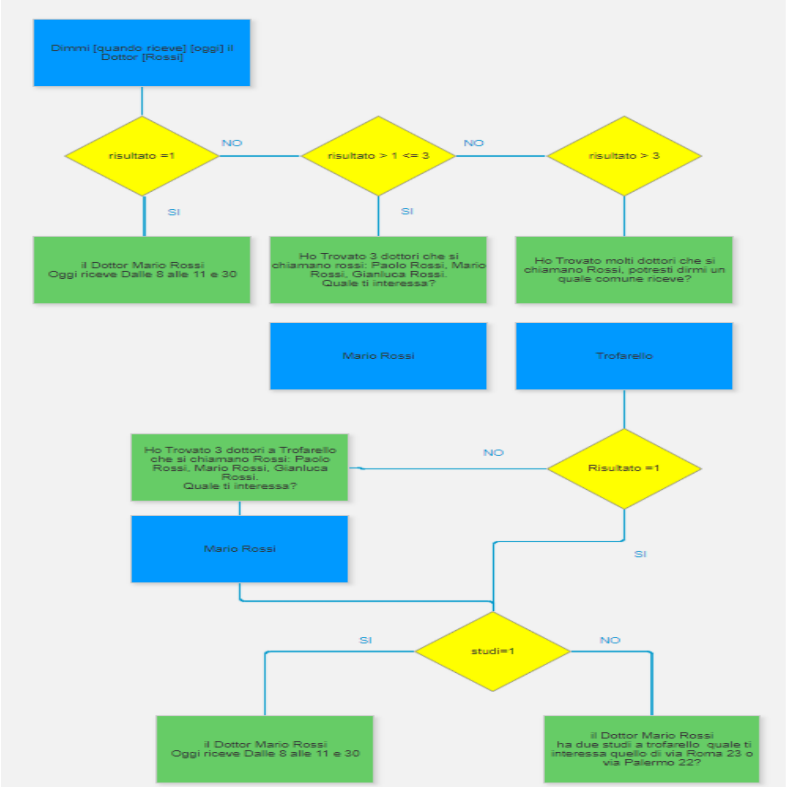

Figura 4.1. Raffigurazione del primo scenario di conversazione da soddisfare

La Figura 4.2 è la rappresentazione di un ipotetico dialogo di tipo *query_list*. In questo caso, viene proposto uno scenario nel quale un utente richiede al bot la lista dei pediatri che operano in un determinato comune. Il bot, come nel dialogo precedente, a seconda del numero di risultati ottenuti dal database, ha la possibilità di rispondere con due differenti esiti. Tuttavia, successivamente, gli stakeholder hanno richiesto la sostituzione dello scenario nel quale il bot risponde inviando una e-mail comprendente la lista dei pediatri, con una richiesta all'utente di fornire maggiori informazioni al fine di filtrare con più successo i risultati.

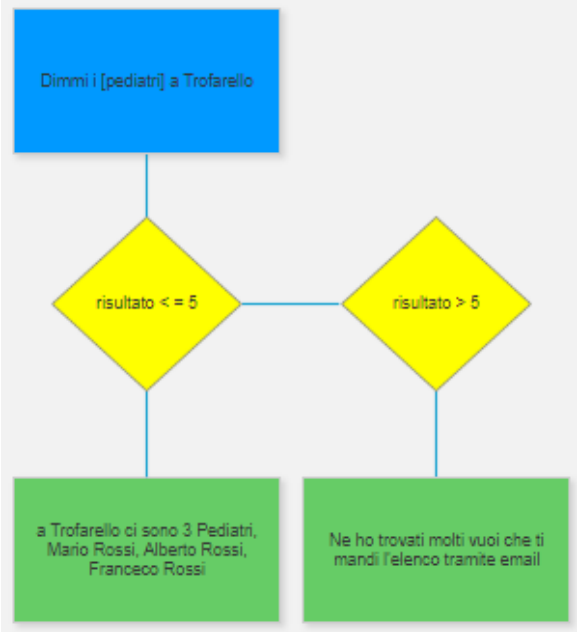

Figura 4.2. Raffigurazione del secondo scenario di conversazione da soddisfare

La richiesta di produzione di risultati limitati è dovuta agli stakeholder, i quali ritenevano necessario che il chatbot avesse la capacità di rispondere attraverso frasi brevi, al fine di garantire una conversazione più fluida all'interno di uno scenario di distribuzione come le piattaforme di assistenza virtuale (ad esempio, Google assistant e Alexa).

Oltre agli scenari di dialogo da soddisfare, è stato richiesto che il bot fosse progettato su una struttura software generale, in grado di permettere la creazione di successivi chatbot che avessero Medici-bot come modello e che, quindi, fossero sviluppati a partire dalla sua architettura, modificando soltanto la knowledge base e un file JSON per la definizione delle caratteristiche personali.

Di fatto, l'obiettivo degli stakeholder non era soltanto la creazione di un chatbot, bensì lo sviluppo di un approccio e di un sistema che supportassero la generazione semi-automatica dei chatbot, fornendo loro soltanto la knowledge base del contesto applicativo e un file di configurazione (in formato `.json`) da definire all'inizio. Questo approccio doveva comprendere anche la fase di generazione degli esempi

delle domande su cui il chatbot venisse addestrato per riconoscere le richieste dell'utente in linguaggio naturale. Ciò ha portato alla realizzazione di una piattaforma per la generazione automatica del dataset di esempi di training, necessari per la creazione del modello NLU (Natural Language Understanding). Il software è stato denominato *Dataset Generator*.

4.2 Knowledge base e Neo4j

Con il termine "knowledge base", in generale, si fa riferimento ad un ambiente volto a facilitare la raccolta, l'organizzazione e la distribuzione della conoscenza, ovvero una tipologia speciale di database. Dunque, le informazioni archiviate nella knowledge base rappresentano la "memoria" del chatbot e determinano anche la conoscenza che esso possiede in merito allo scenario in cui opera.

Per quanto riguarda la tecnologia utilizzata per la memorizzazione dei dati, la scelta è ricaduta sul DBMS *Neo4j*, una base di dati non relazionale open-source che permette una rappresentazione a grafo dei dati. Questo DBMS è stato scelto grazie alle caratteristiche sopracitate e al suo linguaggio di interrogazione dei dati.

Il suo essere non relazionale consente una raffigurazione dei dati basata su nodi di tipologia diversa ed in relazione tra loro. Quindi, l'interrogazione e la gestione dei dati all'interno di questo caso di studio sono favorite da questi fattori, poichè consentono di recuperare le informazioni dal database senza avere conoscenza della precisa alberatura dei dati sul grafo. Dunque, questa tecnologia ha permesso di usufruire di una soluzione scalabile sia in termini dimensionali dei dati, sia nel modello della loro rappresentazione, senza dover ogni volta modificare le funzioni di interrogazioni del database.

Pertanto, le informazioni, fornite dagli stessi stakeholder, sono raccolte in tre file in formato `.csv` e rappresentano ciascuno una tabella.

- `j25_medici_new.csv`: contiene le informazioni personali dei medici della regione Piemonte. I principali campi sono: *"nome"*, *"cognome"*, *"codice_fiscale"*, *"sesso"*, *"data_nascita"*, *"desc_categoria"* (rappresenta la tipologia di medico; può assumere i valori "MMG", per i medici generali, e "PLS" per i pediatri), *"posti-disponibili"* (il campo rappresenta il numero dei posti disponibili per i pazienti che decidono di avere quel dottore come medico di famiglia). Contiene 3425 record.
- `j25_medici_ambulatori_new`: contiene i dati di tutti gli ambulatori della regione. Ogni dottore può avere più di un ambulatorio o luogo di lavoro. Nel file, i campi più rilevanti sono: *"indirizzi"*, *"telefono"*, *"indirizzo_email"*, *"comune"*, *"civico"*, *"cap"*, *"descrizione"* (quest'ultimo campo assume i valori "ambulatorio" o "studio medico"). Contiene 5750 record.
- `j25_medici_orari_new`: contiene gli orari settimanali per ciascuno dei medici e per ciascuno dei luoghi in cui egli esercita la sua professione. I campi più importanti sono: *"ora_fine"*, *"ora_inizio"*, *"giorno"*. Contiene 27639 record.

Di seguito, i file sono stati importati all'interno della base di dati Neo4j. Una volta inseriti, i tre file rappresentano i rispettivi tipi di nodi, ovvero `Medici,`

`Ambulatori e Orari`. Dopodichè, le tre classi di nodi sono state unite attraverso la definizione di due relazioni, denominate `WORK-IN e Orario`.

Lo schema dei dati che si è formato è il seguente:

"(Medici)-[WORK_IN]-(Ambulatori)-[Orario]-(Orari)"

In Figura 4.3 è possibile osservare, sotto forma di nodi e relazioni, una sezione rappresentativa dei dati sul database (nodo arancione: "Medici", nodo azzuro: "Ambulatori", nodo rosso: "Orari").

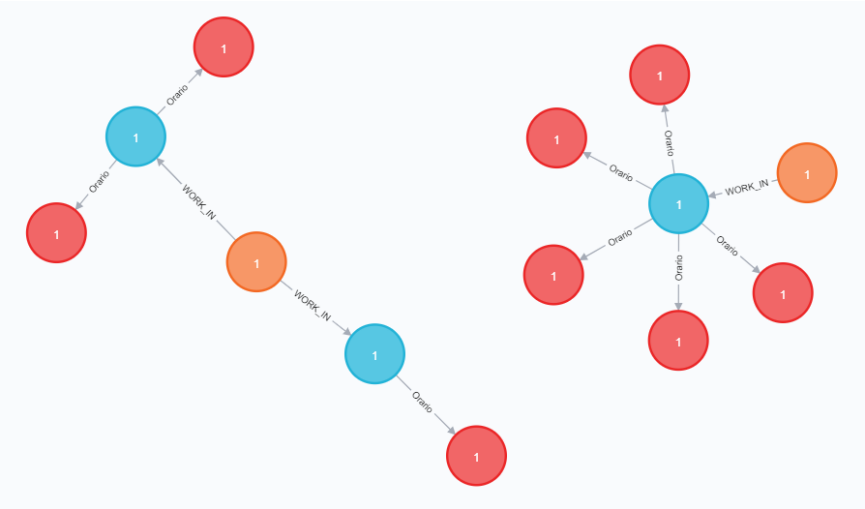

Figura 4.3. Rappresentazione a nodi dei dati sul DB

4.3 Generazione del dataset di training

Dopo il caricamento dei dati sul database, si è passati alla fase di generazione del dataset di training del modello NLU. Esso è composto da un insieme di frasi di esempio, suddivise nei diversi *intent*, che saranno utilizzate per definire un modello per il riconoscimento del linguaggio naturale. Inoltre, nel modello, per ogni frase, vengono evidenziate ed etichettate le eventuali *entity* allo scopo di addestrare il chatbot ad estrarre tali informazioni dalle richieste degli utenti.

Questa fase essenziale per la costruzione del chatbot è stata realizzata attraverso l'utilizzo di un servizio costruito appositamente per generare automaticamente tali training dataset.

4.3.1 Dataset Generator Platform

Il *Dataset Generator* è un servizio sviluppato durante la progettazione del chatbot Medici-bot. Esso consente di generare un dataset di frasi per il training del bot a partire da un file `.json` contenente i template[1] delle frasi.
Dopo aver eseguito l'upload del file dei template sul servizio dedicato alla generazione, ogni singolo template viene elaborato, e per esso, vengono generate automaticamente un massimo di 25 frasi. Al fine di garantirne la diversità costruttiva, per ogni frase viene utilizzato un sinonimo che la rappresenti e si inseriscono, di volta in volta, informazioni diverse prelevate dalla *knowledge base*.

Come si può osservare dall'interfaccia del *Dataset Generator* (Figura 4.4), esso permette di caricare un file `.json` che, dopo la fase di elaborazione, viene scaricato ed il cui contenuto viene copiato nel file `nlu.md`

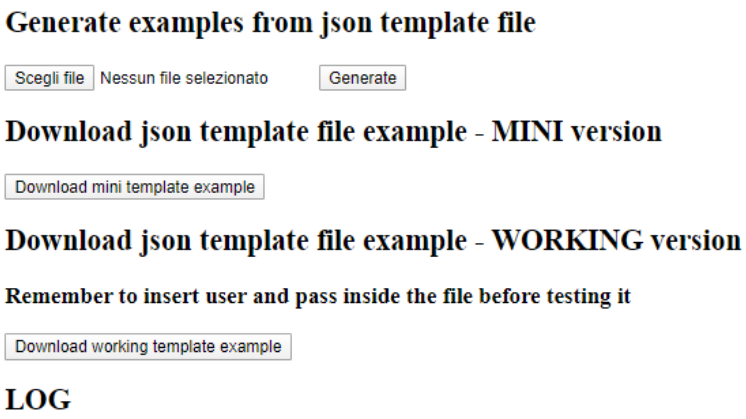

Figura 4.4. Interfaccia del *Dataset Generator*

4.3.2 Definizione file di template

Nel file contenente i template i campi definiti sono tre: *database_entities*, *local_entities*, *intent_template*. Il primo campo è stato ideato per rappresentare le variabili delle informazioni contenute nel database e, perciò, è "valorizzato" dalle seguenti proprietà:

- *database_entities*: contiene, a sua volta, tre diverse chiavi: *uri*, l'indirizzo dove ricercare il database Neo4j; *user*, lo username dell'account Neo4j; *password*.

[1] con questo termine si intendono le strutture sintattiche delle frasi rappresentate dai token che le compongono. Inoltre, tramite essi vengono etichettati le *entity* e vengono definiti anche i sinonimi delle parole.

- *entities*: contiene i riferimenti ai valori degli attributi dei *data_node* che il generatore scarica dal database per formare le frasi di training. Dunque, al suo interno, per ogni attributo dei nodi sul database da cui si vogliono prelevare i valori, vengono specificati *nodo_type* e *attribute_name*. Il primo servirà a definire il tipo del nodo (l'equivalente di tabella nei DBMS relazionali), il secondo per specificare l'attributo a cui si fa riferimento.

Nel secondo campo, *local_entities*, sono presenti le variabili utilizzate per semplificare la rappresentazione degli attributi nei template delle frasi da generare. Un esempio è la variabile contenente i vari sinonimi della parola "dottore" con cui, poi, vengono generate le frasi. Si avrà, quindi, la possibilità di generare frasi con tutti i sinonimi descritti nelle variabili *local_entities*. Inoltre, è possibile definire variabili contenenti al loro interno delle *local_entities* o *database_entities*, così da permettere una generazione delle frasi il più possibile casuale.

Sempre all'interno di questo campo sono definite le *mention*, aggettivi ordinali utilizzati per identificare un risultato posizionato ordinatamente secondo una numerazione crescente (ad esempio, l'aggettivo "primo" nella frase "Vorrei sapere il telefono del primo dottore"). Il modello NLU da addestrare dovrà essere in grado di riconoscere tali elementi usati nel linguaggio umano e di assegnare loro l'oggetto a cui si riferiscono.

Infine, nel seguente file vengono definiti i diversi template, suddivisi a seconda degli *intent* che si ha la volontà di rappresentare; questi sono contenuti nel campo *intent_templates*. In questo caso, le suddivisioni sono tre, e riportano il nome degli *intent* a cui fanno riferimento. Tale campo verrà discusso nel dettaglio nella prossima sottosezione.

Successivamente alla definizione del file e del suo caricamento sulla piattaforma Dataset Generator, viene fornito in output un file `.json` contenente le frasi generate attraverso le proprietà appena descritte. Esse vengono trasferite all'interno del file `nlu.md`, il quale risulta indispensabile nell'architettura Rasa. Il file appena accennato, oltre alle frasi per il training del modello, contiene tutte le informazioni necessarie per la creazione del modello NLU, ovvero le *lookup-table*[2], le RegEx[3] ed i sinonimi delle parole più significative.

4.3.3 Sintassi dei template

La sintassi del dataset degli esempi viene definita all'interno di un file `.json` seguendo la struttura proposta nel Listato 4.1.

```
1
2   {
3       "database_entities": {
4           "database": {
5               "uri": "bolt://int-sdnet-convplat1.sdp.csi.it:7687",
6               "user": "neo4j",
7               "password": "test"
8           },
```

[2] Con questo termine si intende una tabella di ricerca contenente i nomi ed i cognomi dei medici. È utilizzata dal modello NLU per riconoscere tali elementi in una frase.

[3] Con questo termine si intende una Regular Expression, ovvero espressioni regolari, come, ad esempio, "@","#","&", etc.

```
        "entities": {
            "nome": {
                "node_type": "Medici",
                "attribute_name": "nome"
            },
            "cognome": {
                "node_type": "Medici",
                "attribute_name": "cognome"
            },
            "codice_fiscale": {
                "node_type": "Medici",
                "attribute_name": "codice_fiscale"
            },
            "cap": {
                "node_type": "Ambulatori",
                "attribute_name": "cap"
            },
            "indirizzo": {
                "node_type": "Ambulatori",
                "attribute_name": "indirizzo"
            },
            "comune": {
                "node_type": "Ambulatori",
                "attribute_name": "comune"
            },
            "civico": {
                "node_type": "Ambulatori",
                "attribute_name": "civico"
            },
            "orariodichiusura": {
                "node_type": "Orari",
                "attribute_name": "orariodichiusura"
            },
            "orariodiapertura": {
                "node_type": "Orari",
                "attribute_name": "orariodiapertura"
            },
            "giorno": {
                "node_type": "Orari",
                "attribute_name": "giorno"
            }
        }
    },
    "local_entities": {
        "medico": [
            "[dottore](object_type:medico)",
            "[dottoressa](object_type:medico)",
            "[pediatra](object_type:medico)",
            "[dottori](object_type:medico)",
            "[dottoresse](object_type:medico)",
            "[pediatri](object_type:medico)"
        ],
        "ambulatorio": [
            "[ambulatorio](object_type:ambulatorio)",
            "[ambulatori](object_type:ambulatorio)",
            "[ospedale](object_type:ambulatorio)",
            "[studio](object_type:ambulatorio)"
        ],
        "orario": [
            "[orario](object_type:orario)",
            "[orari](object_type:orario)",
            "[ora](object_type:orario)",
            "[quando riceve](object_type:orario)"
        ],
        "telefono_attribute": [
            "[telefono](attribute)",
            "[numero di telefono](attribute:telefono)",
            "[cellulare](attribute:telefono)",
            "[numero di cellulare](attribute:telefono)",
            "[numero](attribute:telefono)"
        ],
        "comune_attribute": [
            "[comune](attribute:comune)",
            "[città](attribute:comune)",
            "[citta](attribute:comune)",
            "[località](attribute:comune)",
            "[localita](attribute:comune)"
        ],
        "cap_attribute": [
            "[cap](attribute:cap)",
            "[CAP](attribute:cap)"
        ],
        "indirizzo_attribute": [
            "[indirizzo](attribute)",
            "[via](attribute:indirizzo)"
        ],
        "orariodichiusura_attribute": [
            "[ora di chiusura](attribute:orariodichiusura)",
            "[orario di chiusura](attribute:orariodichiusura)",
            "[ora chiude](attribute:orariodichiusura)"
        ],
        "orariodiapertura_attribute": [
            "[ora di apertura](attribute:orariodiapertura)",
            "[orario di apertura](attribute:orariodiapertura)",
```

```
103             "[ora apre](attribute:orariodiapertura)"
104         ],
105         "giorno_attribute": [
106             "[giorni di apertura](attribute:giorno)",
107             "[giorni](attribute:giorno)",
108             "[giorni aperti](attribute:giorno)",
109             "[giorno](attribute)"
110         ],
111         "email_attribute": [
112             "[e-mail](attribute:email)",
113             "[mail](attribute:email)",
114             "[email](attribute:email)",
115             "[indirizzo mail](attribute:email)",
116             "[indirizzo email](attribute:email)"
117         ],
118         "civico_attribute": [
119             "[civico](attribute:civico)",
120             "[numero civico](attribute:civico)"
121         ],
122         "tutti_attribute": [
123             "$telefono_attribute",
124             "$comune_attribute",
125             "$cap_attribute",
126             "$indirizzo_attribute",
127             "$orariodichiusura_attribute",
128             "$orariodiapertura_attribute",
129             "$giorno_attribute",
130             "$email_attribute",
131             "$civico_attribute"
132         ],
133         "tutti_object": [
134             "$medico",
135             "$ambulatorio",
136             "$orario"
137         ],
138         "mention_numeri": [
139             "[1](mention)",
140             "[2](mention)",
141             "[3](mention)",
142             "[4](mention)",
143             "[5](mention)"
144             "[ultimo](mention:LAST)",
145             "[primo](mention:1)"
146         ],
147         "mention_ordinali": [
148             "[primo](mention:1)",
149             "[prima](mention:1)",
150             "[secondo](mention:2)",
151             "[seconda](mention:2)",
152             "[terzo](mention:3)",
153             "[terza](mention:3)",
154             "[quarto](mention:4)",
155             "[quarta](mention:4)",
156             "[quinto](mention:5)",
157             "[quinta](mention:5)"
158             "[1o](mention:1)",
159             "[1a](mention:1)",
160             "[2o](mention:2)",
161             "[2a](mention:2)",
162             "[3o](mention:3)",
163             "[3a](mention:3)",
164             "[4o](mention:4)",
165             "[4a](mention:4)",
166             "[5o](mention:5)",
167             "[5a](mention:5)"
168         ],
169         "mention_speciali": [
170             "[penultimo](mention:-2)",
171             "[ultimo](mention:-1)"
172         ],
173         "mention_suo": [
174             "[suo](mention:1)"
175         ],
176         "mention": [
177             "$mention_ordinali",
178             "$mention_numeri"
179         ]
180     },
181     "intent_templates": {
182         "query_knowledge_list": [
183             "[quali|chi|cerco i|cerco le|dimmi|puoi dirmi|vorrei sapere] $tutti_object [sono|lavorano|conosci|
                    elenco|elencami] a $comune ?",
184             "[quali|chi|cerco i|cerco le|dimmi|puoi dirmi|vorrei sapere] $tutti_object [sono|lavorano|conosci|
                    elenco|elencami] in $indirizzo $civico?",
185             "[quali|chi|cerco i|cerco le|dimmi|puoi dirmi|vorrei sapere] $ambulatorio [sono|lavorano|conosci|elenco
                    |elencami] [a|nel comune di] $comune ?",
186             "[quali|chi|cerco i|cerco le|dimmi|puoi dirmi|vorrei sapere] $medico [sono|lavorano|conosci|elenco|
                    elencami] $giorno ?",
187             "[quali|chi|cerco i|cerco le|dimmi|puoi dirmi|vorrei sapere] $medico [sono|lavorano|conosci|elenco|
                    elencami] $comune ?",
188             "[quali|chi|cerco i|cerco le|dimmi|puoi dirmi|vorrei sapere] $medico che [lavorano|fanno|aprono|
                    iniziano] [dalle|alle] $orariodiapertura ?",
189             "[quali|chi|cerco i|cerco le|dimmi|puoi dirmi|vorrei sapere] $medico che [lavorano|fanno|chiudono|
                    restano aperti] [alle|fino alle] $orariodichiusura ?",
```

```
190             "[quali|chi|cerco i|cerco le] $ambulatorio [sono|lavorano|conosci|elenco|elencami|si trova] a $comune
                    in $indirizzo ?",
191             "[Qual'e|dimmi|vorrei sapere|cerco] [l'|gli] $orario [di|del dottor] $cognome a $comune ?",
192             "[Qual'e|dimmi|vorrei sapere|cerco] [l'|gli] $orario [di|del dottor] $cognome",
193             "[|i|gli] $tutti_object [|a|di] $comune",
194             "[|mi dici|vorrei sapere] [quando riceve](object_type:orario) [al|di] $giorno [|il dottor|la dottoressa
                    ] $nome $cognome",
195             "[|mi dici|vorrei sapere] [quando riceve](object_type:orario) [al|di] $giorno [|il dottor|la dottoressa
                    ] $cognome",
196             "[Qual'e|dimmi|vorrei sapere|cerco] [l'|gli] $ambulatorio [aperti|aperto] $giorno a $comune",
197             "[mi dici|vorrei sapere|dimmi|cerco|fammi] la lista $tutti_object [nel comune di|a] $comune ?",
198             "[mi dici|vorrei sapere|dimmi|cerco|fammi] la lista $tutti_object [ricevono|operano|sono aperti]
                    $giorno ?",
199             "[dov'è|dove sono| in che comune è] $ambulatorio [|del dottor|della dottoressa] $nome $cognome",
200             "dove [lavora](object_type:ambulatorio) [|il dottor|la dottoressa] $nome $cognome"
201
202         ],
203         "query_knowledge_attribute_of": [
204             "[qual è|dimmi|vorrei sapere] [|il|lo|la|l'] $telefono_attribute [di|del dottore|della dottoressa]
                    $cognome",
205             "[qual è|dimmi|vorrei sapere] [|il|lo|la|l'] $tutti_attribute [di|del dottor|del dottore|della
                    dottoressa] $nome $cognome",
206             "[|il dr|la dottoressa] $cognome [quale|che|in quale] $tutti_attribute [ha|possiede]",
207             "[|il|la] $mention [quale|che|in quale] $tutti_attribute [ha|possiede]",
208             "[Qual'e|in quale|dimmi|vorrei sapere|cerco] i $giorno_attribute dell'ambulatorio a $comune?",
209             "[Qual'e|in quale|dimmi|vorrei sapere|cerco] [la|il|l'] $indirizzo_attribute [di|del dottor|del dottore
                    |della dottoressa] $cognome ?",
210             "[Qual'e|in quale|dimmi|vorrei sapere|cerco] $tutti_attribute [del|di|è] $mention ?",
211             "$tutti_attribute [|di|del dottor|del dottore|della dottoressa] $cognome",
212             "$tutti_attribute [|di|del dottor|del dottore|della dottoressa] $nome $cognome",
213             "$tutti_attribute del $mention",
214             "il $mention_suo $tutti_attribute ?"
215         ],
216         "query_details": [
217             "$nome",
218             "$nome $cognome",
219             "$comune",
220             "il suo nome è$nome",
221             "il suo comune è$comune",
222             "il suo cognome è$cognome",
223             "si chiama $nome",
224             "lavora a $comune"
225         ]
226     }
227 }
```

Listato 4.1. Template per la generazione del dataset di training

Il template per la generazione del dataset presenta una sintassi precisa e ben definita. Dunque, analizzando singolarmente i tre campi principali del file di template (Figura 4.1), è possibile osservarne le seguenti caratteristiche:

1. Le variabili possono essere nominate a piacere.
2. Le variabili, nel momento in cui vengono inserite tra i valori che una nuova variabile può assumere, devono essere richiamate con l'aggiunta del carattere "$" a monte del loro nominativo.
3. Al fine di rispettare la sintassi di Rasa per la rappresentazione delle frasi nel training dataset NLU, le *entity* e l'*object_type*[4] sono etichettati secondo un linguaggio di markup[5] preciso. Di seguito, vi è riportato un esempio del formato accettato da Rasa e usato per la definizione dei template.

Il modello prodotto da Rasa per l'identificazione degli *intent*, delle *entity* e degli *object_type* viene addestrato utilizzando dei markdown ad etichettare e rilevare il significato dei termini, nel caso questi ne avessero uno. Di seguito, saranno illustrate le diverse etichette:

[4] Termine rilevato nell'*Entity Extraction* e utile ad identificare l'oggetto di una richiesta. Ad esempio, nella frase "Mi dici i dottori a Torino?" il termine "dottori" identifica la precisa richiesta dell'oggetto dottori. In questo caso avremo *object_type* = "medico". Nel caso di studio, tale campo può assumere i valori "medico", "ambulatorio" e "orario".

[5] È un linguaggio che permette di descrivere i dati attraverso una formattazione specifica che utilizza i cosiddetti tag, che non sono altro che dei marcatori.

- *entity*: "`[<token>](attribute)`" (ad esempio, "`[giorno](attribute)`"). Nel caso il token fosse un sinonimo dell'*entity*, si avrebbe "`[<token>](attribute:<entity>)`" (ad esempio, "`[giorni di apertura](attribute:giorno)`").
- *object_type*: "`[<token>](object_type)`" (ad esempio, "`[medico](object_type)`"). Nel caso di sinonimo dell'*object_type*, si avrebbe "`[<token>](object_type:<object_type>)`" (ad esempio, "`[dottoressa](object_type:medico)`").

Dopo aver definito le etichette con le quali le frasi vengono generate, nel template rimangono da illustrare i *database_entities* . Questi, come accennato in precedenza, sono delle variabili che, durante la generazione, vengono sostituite con dei token ottenuti dal database e che rappresentano gli attributi di cui prendono il posto.

A seguire, verranno illustrati il metodo di creazione del dataset ed il formato di rappresentazione dei template attraverso l'ausilio di due esempi derivati dal Listato 4.1.

```
1  8220
2      [Qual'e|dimmi|vorrei sapere|cerco] [l'|gli] $orario [di|del dottor] $cognome a $comune ?";8220
3      [qual è|dimmi|vorrei sapere] [|il|lo|la|l'] $telefono\_attribute [di|del dottore|della dottoressa] $cognome".
```

Listato 4.2. Esempi di template ricavati dal file per la generazione

Il primo esempio è tratto dai template dell'*intent "query_knowledge_list"*, mentre il secondo dai template *"query_knowledge_attribute_of"*. Durante la fase di generazione, vengono riconosciuti i termini tra le parentesi quadre come possibili alternative e solo una di queste viene casualmente selezionata. Oltre a ciò, attraverso il carattere "$", il generatore è in grado di riconoscere le variabili definite nei campi *local_entities* e *database_entities* e a sostituirle nella frase da creare, associando a ciascuna di queste il corrispondente valore che la variabile rappresenta. Un ipotetico risultato potrebbe essere il seguente:

- "Vorrei sapere gli orari del dottor Rossi a Torino";
- "Qual è il telefono del dottore Bianchi?".

È importante notare che la fase di generazione produce anche frasi grammaticalmente inesatte, anche se ciò non inficia eccessivamente la bontà del modello che si andrà ad addestrare a partire dal dataset generato. Comunque sia, è sempre possibile migliorare i template affinchè essi producano frasi grammaticalmente corrette.

Osservazioni risultate dall'analisi dei requisiti

Nel corso della realizzazione del chatbot, si sono aggiunti dei requisiti a quelli inizialmente previsti dallo stakeholder e a quelli rilevati durante la fase di progettazione.

Il primo requisito aggiunto rispondeva alla necessità del chatbot Medici-bot di essere in grado di garantire una conversazione fluida e il più naturale possibile; perciò, oltre ai due principali *intent* (*query_knowledge_list* e *query_attribute_of*), ne è stato creato un terzo chiamato *query_details*. In quest'ultimo vengono identificati i messaggi che un utente invia al bot nei quali viene riconosciuto l'obiettivo di

specificare una richiesta posta precedentemente. Questa circostanza si verifica nel momento in cui il chatbot restituisce all'utente delle risposte multiple (ad esempio, potrebbe aver trovato più medici residenti in un comune). Detto ciò, attraverso questo nuovo *intent*, il bot può permettere all'utente di definire dei campi di ricerca aggiuntivi, al fine di ottenere risultati ancor più precisi o univoci.

Per implementare questa funzionalità, il bot ha necessità di mantenere in memoria il messaggio che ha ricevuto precedentemente e deve avere la capacità di aggiungere i nuovi parametri di ricerca a quelli relativi al messaggio precedente.

Dopodichè, un requisito voluto dallo stakeholder è risultato essere la capacità di distribuzione su canali di assistenza virtuale ben inseriti sul mercato. Per questa ragione è stato richiesto di connettere il chatbot alle piattaforme Google Assistant ed Alexa, insieme alla possibilità di salvataggio del medico di fiducia da parte dell'utente attraverso la funzione di *Account Linking* (Autentificazione dell'utente per mezzo della piattaforma ospitante, ad esempio, Google SIGN-In e Alexa SIGN-IN).

In Figura 4.5, si può osservare l'architettura completa del sistema che è stata definita a valle dell'analisi dei requisiti. In aggiunta alle componenti di cui si è parlato, nella figura è presente un modulo denominato `GoogleConnector & AlexaConnector`. Quest'ultimo servirà al bot per interfacciarsi con le due piattaforme di assistenza virtuale.

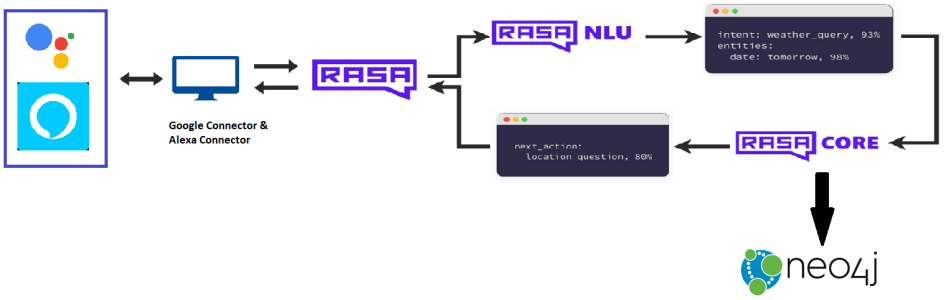

Figura 4.5. Architettura del servizio *Medici-bot*

Come ultima specifica rilevata in fase di analisi, al fine di rendere il bot una soluzione generale e riutilizzabile in contesti diversi, è stato necessario definire un file di supporto contenente le informazioni sul bot che rendessero il suo funzionamento il più astratto possibile. Di conseguenza, è stata rilevata la necessità di possedere una classe per interfacciare il bot e le informazioni per il corretto funzionamento. Inoltre, all'interno del file sono state ipotizzate delle informazioni utili per personalizzare ed esprimere il corretto output del chatbot in relazione ad ogni singola richiesta dell'utente.

5

Implementazione di Medici-bot

Dopo aver discusso dei requisiti e della generazione del dataset di training del chatbot Medici-bot, verrà esposta l'implementazione delle parti principali mettendo in evidenza le varie classi realizzateolte.

5.1 Panoramica per la configurazione di Medici-bot

In accordo con quanto dichiarato nel Capitolo 3, per configurare il framework Rasa e beneficiare delle sue funzionalità è necessario implementare le seguenti classi:

- `config.yml`
- `credentials.yml`
- `domain.yml`
- `nlu.md`
- `stories.md`
- `actions.py`

È importante specificare che i file `config.yml` e `credentials.yml` vengono utilizzati da Rasa per configurare i vari moduli del framework, mentre i restanti file sono utilizzati per definire la logica e tutte le funzionalità di cui è composto.

5.2 Implementazione del file di configurazione del chatbot

In questa sezione verrà proposta l'implementazione effettuata sui file `config.yml` e `credenzials.yml`.

5.2.1 config.yml

Il file `config.yml` presenta, al suo interno, le definizioni dei parametri della configurazione della fase di training del modello NLU.

Di fatto, come si può osservare dal Listato 5.1, il primo campo definito è la lingua che il modello verrà riconoscere, mentre nel secondo viene caratterizzata la pipeline di tipo *supervised_embeddings* (tale pipeline è stata trattata nella sottosezione 3.4).

È possibile definire la pipeline semplicemente attraverso il comando `pipeline supervised_embeddings`. Tuttavia, al fine di ottenere un modello personalizzato, si è optato di riportare, passo dopo passo, le fasi che costituiscono la pipeline desiderata.

```
language: it
#pipeline: supervised_embeddings
pipeline:
- name: "WhitespaceTokenizer"
  case_sensitive: false
- name: "src.my_synonym_mapper.SynonymMapper"
- name: "RegexFeaturizer"
- name: "CRFEntityExtractor"
- name: "EntitySynonymMapper"
- name: "CountVectorsFeaturizer"
  analyzer: "char_wb"
  min_ngram: 1
  max_ngram: 4
- name: "EmbeddingIntentClassifier"
- name: "DucklingHTTPExtractor"
  url: "http://localhost:8000"
  locale: "it_IT"
  dimensions: ["time", "email"]
  timezone: "Europe/Rome"

policies:
  - name: MemoizationPolicy
  - name: KerasPolicy
```

Listato 5.1. Rappresentazione del file `config.yml`

La pipeline è formata da un primo modulo *WhitespaceTokenizer*, che si occupa di riconoscere i token di una frase attraverso il rilevamento degli spazi tra una parola e la successiva. Il secondo modulo rappresenta una delle due fasi aggiunte rispetto alla *supervised_embeddings* standard, le quali hanno motivato la scelta di definire una pipeline personalizzata. Tale fase utilizza il modulo denominato `src.my_synonym_mapper.SynonymMapper`, il quale viene usato allo scopo di riconoscere i sinonimi dei termini durante la fase di estrazione degli `object_type`: Per svolgere la sua attività utilizza un file *JSON* i cui valori sono riportati nel seguente listato (rappresentati dal pattern “sinonimo”:“elemento”):

```
{
    "dottore": "medico",
    "dottoressa": "medico",
    "dottori": "medico",
    "dottoresse": "medico",
    "medico": "medico",
    "medici": "medico",
    "pediatra": "medico",
    "pediatri": "medico",
    "pediatre": "medico",
    "centro": "ambulatorio",
    "centri": "ambulatorio",
    "ospedale": "ambulatorio",
    "ospedali": "ambulatorio",
    "clinica": "ambulatorio",
    "cliniche": "ambulatorio",
    "struttura": "ambulatorio",
    "strutture": "ambulatorio",
    "studi": "ambulatorio",
    "studio": "ambulatorio",
    "ambulatori": "ambulatorio",
    "orario": "orario",
    "orari": "orario"
}
```

Listato 5.2. Rappresentazione del file contenente i sinonimi degli *object_type*

Di seguito, il *RegexFeaturizer* permette di creare un modello capace di individuare le Regex all'interno di un messaggio dell'utente.

CRFEntityExtractor è un componente utilizzato per eseguire il riconoscimento delle entity attraverso l'implementazione dei *Conditional Random Fields* (CRF)[1].

Il componente *EntitySynonymMapper* è il modulo di Rasa che, a differenza del precedente "SynonymMapper", permette l'individuazione dei sinonimi delle entità.

Di seguito, il modulo *CountVectorsFeaturizer* implementa una classificazione degli intenti e la selezione della risposta. A tale scopo, esso realizza una rappresentazione bag-of-word del messaggio, dell'intent e della risposta dell'utente utilizzando la funzione `CountVectorizer`[2] di *sklearn*. Tutti i token sono costituiti solo da cifre (come, ad esempio, 123 e 99, ma non a123d) che verranno, poi, assegnate alla stessa "feature".

L'algoritmo *EmbeddingIntentClassifier* incorpora gli input dell'utente e gli intent etichettati nello stesso spazio. In questa fase è permessa la crezione di modelli addestrati massimizzando la somiglianza tra elementi etichettati; tale algoritmo fornisce anche delle classifiche di somiglianza tra le etichette che non risultano "vincenti".

L'ultima fase rappresenta un estrattore di entity denominato *Duckling*, il quale affianca gli altri moduli per l'estrazione dei token. All'interno del file `config.yml`. quindi, viene definito il collegamento al suddetto servizio che permette di estrarre tutti i riferimenti agli elementi della dimensione temporale (ad esempio, domani, oggi, etc.) che si possono trovare in una frase, e mapparli in formato `datetype`. Inoltre, Duckling è fornito come servizio all'interno di un container Docker, il quale si avvia attraverso il comando mostrato nel seguente listato:

```
1    docker run -p 8000:8000 rasa/duckling
```

Listato 5.3. Istruzione da riga di comando per l'avvio del servizio *Duckling*

5.2.2 credentials.yml

Al fine di concedere al chatbot l'accesso alle piattaforme di messaggistica, è possibile utilizzare il file `credential.yml` per definire le credenziali delle loro connessioni. Lo stesso framework Rasa fornisce una *shell* di comando per permettere di testare ed addestrare il chatbot durante le fasi di sviluppo.

Le connessioni stabilite per il seguente chatbot sono osservabili nel Listato 5.4 e sono tre.

[1] I CRF, ovvero i *campi aleatori condizionali*, possono essere pensati come una catena di Markov non orientata in cui le transizioni sono parole e gli stati sono classi di entità. Le caratteristiche delle parole (lettere maiuscole, tag POS, etc.) definiscono la probabilità di rientrare in determinate classi di entità, così come le transizioni tra entità vicine; il set più probabile di entità etichettate viene, quindi, calcolato e restituito come risultato.

[2] Una funzione della libreria sklearn per la conversione da collezione di documenti di testi in matrici valorizzate dal conteggio del token.

```
1   # This file contains the credentials for the voice & chat platforms
2   # which your bot is using.
3   # https://rasa.com/docs/rasa/user-guide/messaging-and-voice-channels/
4
5   rasa:
6     url: "http://localhost:5002/api"
7
8   #googleassistant:
9   ga_connector.GoogleConnector:
10
11  #alexa:
12  alexa_connector.AlexaConnector:
```

Listato 5.4. Rappresentazione del file `credential.yml`

La prima riga del Listato 5.4 rappresenta il riferimento all'indirizzo del servizio da dove è possibile raggiungere la *shell* di Rasa per effettuare delle richieste al chatbot. Di seguito, il secondo e il terzo elemento della lista definiscono le relative classi in cui all'interno sono implementati i connettori delle piattaforme Google Assistant e Alexa; entrambe le classi offrono una connessione di tipo *webhook*[3] ai due assistenti virtuali.

All'interno del file `credential.yml`, sono specificate esclusivamente le piattaforme di messaggistica tramite cui l'utente può interagire con Medici-bot, mentre la logica dietro alla base delle connessioni verrà illustratata nelle sezioni 5.2.9 e 5.2.10.

5.2.3 domain.yml

Il file `domain.yml` definisce il contesto del chatbot; al suo interno sono contenute le `action`, le `entity`, gli `intent`, gli `slot` e i `templates` (questi componenti sono stati introdotti nel Capitolo 3). Sostanzialmente, all'interno del file di cui sopra sono definite tutte le componenti che caratterizzano il chatbot.

Nel Listato 5.6, come primo elemento è possibile osservare il campo `actions`, costituito da dieci *action* al suo interno. Queste ultime sono classificabili in due tipologie; alla prima appartengono le 6 *action*:

- `action_query_attribute`
- `action_query_list`
- `action_query_details`
- `action_personal_doctor`
- `action_information_personal_doctor`
- `action_delete_personal_doctor`

La seconda tipologia è composta dalle restanti quattro *action*. L'unica differenza che contraddistingue le due tipologie è l'implementazione della prima all'interno del file `actions.py`, dove, per ciascuna di esse, viene elaborata la logica con la quale il bot formula le risposte alle richieste dell'utente. Diversamente, la seconda tipologia costituisce l'insieme dei *templates* del bot (con tale termine si sta facendo riferimento ad elementi diversi rispetto agli omonimi discussi nel capitolo precedente), ovvero i messaggi che, in funzione dell'intent, formano le risposte statiche rilevate dall'utente.

[3] I webhook sono delle callback HTTP definite dall'utente per richiedere certe informazioni allo scatenarsi di un evento. Nel caso di studio, le piattaforme di assistenza virtuale inviano una richiesta al chatbot, il quale risponde al mittente con l'informazione ricercata.

Le frasi che appartengono a questo set di risposte sono, ad esempio, i ringraziamenti, i saluti, etc.

Successivamente, all'interno del file, sono state definite le *entity* che il chatbot dovrà essere capace di estrarre dalle domande poste dall'utente.

Tra gli elementi definiti in tale campo, è possibile osservare `feedback-value` (riga 14 del Listato 5.6); questa *entity* rappresenta le risposte, affermative o negative, che l'utente esprimerà ad una richiesta a lui posta. Come sarà possibile notare di seguito, tale entità sarà riconosciuta esclusivamente se il bot riceverà in ingresso i token "Sì", "Ok" e "No". Inoltre, alla riga 33 del file, è possibile osservare l'*object_type* il quale, anche se a livello semantico rappresenta un diverso elemento, dal punto di vista del modello NLU risulta essenzialmente una *entity*.

Di seguito, sono definiti gli *intent* che caratterizzano gli scenari di dialogo ai quali il bot dovrà saper rispondere (alla riga 39 del file).

- `query_knowledge_list`: esso rappresenta la richiesta di un elemento *object_type* a seconda di un attributo di ricerca.
- `query_knowledge_attribute_of`: esso rappresenta la richiesta di un'*entity* per mezzo di un attributo di ricerca.
- `query_details`: all'interno di tale *intent* vengono mappate le frasi del bot in cui viene precisata una domanda attraverso la definizione di un ulteriore attributo.
- `personal_doctor`: al suo interno vengono rappresentate le risposte positive o negative dell'utente in relazione ad una richiesta posta dal chatbot; ad esempio, la richiesta di conferma del salvataggio del medico preferito tra le informazioni personali dell'utente.
- `delete_personal_doctor`: esso definisce l'intent nel quale vengono mappate le richieste di eliminazione dell'informazione sul medico preferito dall'utente.
- `information_personal_doctor`: essa rappresenta una richiesta di informazioni riguardanti il medico personale salvato in quell'istante.
- `greet`: al suo interno vengono mappate le frasi del bot nel quale vengono rilevati dei ringraziamenti.
- `goodbye`: al suo interno vengono riconosciuti i saluti espressi da un utente.
- `bot_challenge`: essa rappresenta l'*intent* nel quale l'utente chiede se stia interagendo con un bot o un utente.
- `deny+ringraziamento`: al suo interno vengono mappate le frasi del bot composte da un'affermazione negativa accompagnata da un ringraziamento (ad esempio, "no, grazie")
- `ringraziamento`: al suo interno vengono mappate le frasi del bot nel quale si rilevano dei ringraziamenti.
- `affirm`: al suo interno vengono mappate le frasi del bot nel quale sono rilevate delle frasi di affermazioni.
- `affirm+ringraziamento`: al suo interno vengono mappate le frasi del bot composte da un'affermazione positiva accompagnata da un ringraziamento (ad esempio, "sì, grazie").
- `deny`: al suo interno vengono mappate le frasi del bot nel quale vengono rilevate delle negazioni.
- `saluto`: al suo interno vengono riconosciuti i saluti di benvenuti di un utente.

Nel campo `Slots`, alla riga 56, sono definiti gli *slot* che il bot utilizza come variabili all'interno della costruzione della risposta all'utente.

Gli *slot* possono essere suddivisi in due categorie; quelli della prima categoria sono utilizzati per mappare le *entity* che vengono rilevate durante la fase di estrazione, mentre quelli della seconda sono i seguenti:

- `feedback_value`: esso viene utilizzato all'interno dell'*action* che risponde all'*intent* `personal_doctor`. Dunque, i valori che lo *slot* può assumere sono esclusivamente "Si" e "No", poichè il modello è stato addestrato a rilevare l'*entity* riconoscendo tali valori dalle espressioni dell'utente. Inoltre, `feedback_value` rappresenta l'unico *slot* valorizzato da una tipologia di dato specifica, ovvero `categorical`.
- `personal_doctor`: contiene l'*id* identificativo del medico preferito dell'utente, il quale viene, poi, memorizzato tra le informazioni personali. Nello specifico, tale *slot* viene valorizzato all'interno dell'*action* `action_query_list` ed utilizzato in `action_personal_doctor`.
- `attribute`: al suo interno si mappa l'attributo di una tabella del database che l'utente vuole ottenere. In altre parole, con tale *slot* viene definito l'attributo di un `object_type`; ad esempio, nel caso del termine "telefono", esso viene rilevato come *entity* di un `object_type` "medico" e, di conseguenza, viene salvato in tale *slot*.
- `knowledge_base_last_object`: tale *slot* mantiene nella memoria della conversazione l'identificativo del risultato dell'ultima richiesta dell'utente.
- `knowledge_base_last_object_type`: esso mantiene nella memoria della conversazione l'`object_type` dell'ultima richiesta dell'utente. Questa variabile viene utilizzata per mantenere il contesto dell'oggetto con cui il bot risponderà. Ad esempio, nel momento in cui sarà richiesto all'utente di fornire maggiori precisazioni su una particolare richiesta, tale *slot* risulterà utile al fine di mantenere il tipo di oggetto a cui si riferisce la richiesta precedentemente posta.
- `knowledge_base_listed_object`: all'interno di tale *slot* verranno conservati e rappresentati, sotto forma di lista, gli identificativi dei risultati di una richiesta dell'utente.
- `last_attributes`: la variabile rappresenta l'intero risultato di una query al database. Tale valore, una volta memorizzato, viene utilizzato per poter prelevare informazioni utili all'interno di successive interazioni con l'utente (l'interazione si dovrà consumare imprescindibilmente all'interno di un'unica conversazione).
- `mention`: al suo interno sono salvati degli identificativi numerici rilevati precedentemente come aggettivi ordinali dal modello NLU. Questi termini verranno, in seguito, utilizzati come dei riferimenti posizionali per gli elementi all'interno della lista rappresentata nello *slot* `knowledge_base_listed_object`.

In conclusione, all'interno del campo *Templates*, sono elencate le risposte statiche che il chatbot è in grado di fornire nel momento in cui il relativo *intent* si presenta. Come già detto nell'illustrazione del campo *actions*, al suo interno vi sono dieci *action*, quattro delle quali risultano essere sostanzialmente i seguenti *template*.

Detto ciò, è possibile affermare che il chatbot è in grado di rispondere con delle frasi statiche (ovvero, non generate dinamicamente dalla classe `actions.py`) a quattro *intent*, adoperando una scelta casuale tra le frasi che sono riportate nei singoli *template*.

Un esempio, tratto dal `domain.yml`, è riportato nel Listato 5.5. In esso è possibile osservare il *template* `utter_ask_rephrase` con al suo interno due opzioni che il bot può selezionare come risposta da fornire all'utente. Come si può intuire, nel seguente esempio sono riportate le risposte nel caso in cui il modello NLU non avesse avuto successo con la comprensione del messaggio dell'utente.

```
1   utter_ask_rephrase
2       - text: Scusa non capisco, puoi rifrasare?
3       - text: Scusa, potresti riesprimere il concetto?
```

Listato 5.5. Esempio della definizione del *template* `utter_ask_rephrase`

```
1   %YAML 1.1
2   ---
3   actions:
4   - action_personal_doctor
5   - action_information_personal_doctor
6   - action_delete_personal_doctor
7   - action_query_attribute
8   - action_query_details
9   - action_query_list
10  - utter_ask_rephrase
11  - utter_goodbye
12  - utter_greet
13  - utter_iamabot
14  entities:
15  - feedback_value
16  - ambulatorio
17  - appunto_ricevimento
18  - attribute
19  - cap
20  - civico
21  - codice_fiscale
22  - cognome
23  - comune
24  - date
25  - descrizione
26  - distretto
27  - email
28  - giorno
29  - indirizzo
30  - medico
31  - mention
32  - nome
33  - object_type
34  - orariodiapertura
35  - orariodichiusura
36  - postidisponibili
37  - telefono
38  - telefonosecondario
39  intents:
40  - personal_doctor
41  - delete_personal_doctor
42  - information_personal_doctor
43  - query_knowledge_list
44  - query_knowledge_attribute_of
45  - query_details
46  - greet
47  - goodbye
48  - bot_challenge
49  - input_date
50  - deny+ringraziamento
51  - ringraziamento
52  - affirm
53  - affirm+ringraziamento
54  - deny
55  - saluto
56  slots:
57    feedback_value:
58      type: categorical
59      values:
60      - positive
61      - negative
62      - si
63      - no
64    personal_doctor:
65      type: unfeaturized
66    ambulatorio:
67      type: unfeaturized
68    appunto_ricevimento:
```

```
    type: unfeaturized
  attribute:
    type: unfeaturized
  cap:
    type: unfeaturized
  civico:
    type: unfeaturized
  codice_fiscale:
    type: unfeaturized
  cognome:
    type: unfeaturized
  comune:
    type: unfeaturized
  date:
    type: unfeaturized
  descrizione:
    type: unfeaturized
  distretto:
    type: unfeaturized
  email:
    type: unfeaturized
  giorno:
    type: unfeaturized
  indirizzo:
    type: unfeaturized
  knowledge_base_last_object:
    type: unfeaturized
  knowledge_base_last_object_type:
    type: unfeaturized
  knowledge_base_listed_objects:
    type: unfeaturized
  knowledge_base_object:
    type: unfeaturized
  last_attributes:
    type: unfeaturized
  medico:
    type: unfeaturized
  mention:
    type: unfeaturized
  nome:
    type: unfeaturized
  object_type:
    type: unfeaturized
  orariodiapertura:
    type: unfeaturized
  orariodichiusura:
    type: unfeaturized
  postidisponibili:
    type: unfeaturized
  telefono:
    type: unfeaturized
  telefonosecondario:
    type: unfeaturized

templates:
  utter_ask_rephrase:
  - text: Scusa non capisco, puoi rifrasare?
  - text: Scusa, potresti riesprimere il concetto?
  utter_goodbye:
  - text: Alla prossima
  - text: Arrivederci
  utter_greet:
  - text: Ciao!
  - text: Buongiorno! Come posso aiutarti?
  utter_iamabot:
  - text: Si, sono un bot
```

Listato 5.6. Rappresentazione del file `domain.yml`

5.2.4 nlu.md

Il file `nlu.md` contiene il dataset di frasi, generate dal servizio *Dataset Generator*, ed utilizzate da Rasa per effettuare il training del modello NLU del chatbot[4].

Al suo interno, oltre alle frasi di esempio definite per ogni *intent* del chatbot, sono presenti anche alcuni sinonimi di termini utilizzati in tali esempi, oltre alla definizione dei path dove raggiungere e recuperare le *lookup table* che saranno utilizzate dal modello NLU.

[4] Il training del modello NLU viene eseguito utilizzando i file `nlu.md`, `stories.md`, `domain.yml` e `config.yml`.

```
1   ## intent:query_knowledge_attribute_of
2   <!-- [|il dr|la dottoressa] $cognome [quale|che|in quale] $tutti_attribute [ha|possiede] -->
3   - la dottoressa [Rossi](cognome) che [indirizzo mail](attribute:email) possiede?
```

Listato 5.7. Esempio di frase definita nel file `nlu.md` dell'*intent* `query_knowledge_attribute_of` e relativo template

5.2.5 stories.md

In linea con quanto introdotto nel Capitolo 3 riguardante il framework Rasa, le *story*, ritenute componenti essenziali per la definizione del bot, vengono espresse nel file `stories.md`. Al suo interno, esse sono definite seguendo un pattern sintattico ben preciso, come si può osservare nel Listato 5.8.

Come è possibile notare, la sintassi è composta dal nome della *story* e, a seguire, da una serie di *intent*, con la relativa *action*, con cui si prevede che il chatbot risponda. Il numero degli *intent* definibili dipenderà esclusivamente dallo scenario di dialogo che la *story* rappresenterà. Inoltre, l'importanza del file `stories.md` è dovuta anche al fatto che al suo interno siano stabilite le *action* o i *template* che dovranno rispondere ad un determinato *intent*; tutto ciò avviene all'interno di un determinato scenario conversazionale con cui il modello NLU verrà addestrato per riconoscere il contesto del dialogo nonchè la modalità con cui esso si potrà sviluppare.

```
1   ## <<nome story>>
2   * <<intent>>
3     - <<template>>
4   * <<intent>>
5     - <<action>>
6   * <<intent>>
7     - <<action>>
8   * <<intent>>
9     - <<template>>
```

Listato 5.8. Sintassi per la definizione di una *story*

Per maggiore chiarezza, utilizzando la *story* nel Listato 5.9 alla riga 9, è possibile analizzare un esempio di una sua dichiarazione. Perciò, un chatbot che riceve un saluto (e cioè l'*intent* `greet`), risponderà con una frase del template `utter_greet`. A seguire, poichè il bot è stato addestrato sulla base di questa storia, nel caso la conversazione continuasse, esso si aspetterà un *intent* `query_knowledge_list` e, di conseguenza, il sistema risponderà attraverso l'*action* `action_query_list`. Allo stesso modo, ad una richiesta `query_knowledge_attribute_of`, Medici-bot risponderà con la risposta opportuna generata dall'`action_query_attribute`.

Per concludere, l'utente che saluterà il bot in questione riceverà una risposta selezionata dalla lista dei *template* di `utter_goodbye`.

```
1   ## Happy path 1
2   * greet
3     - utter_greet
4   * query_knowledge_list
5     - action_query_list
6   * goodbye
7     - utter_goodbye
```

```
 8
 9  ## Happy path 2
10  * greet
11    - utter_greet
12  * query_knowledge_list
13    - action_query_list
14  * query_knowledge_attribute_of
15    - action_query_attribute
16  * goodbye
17    - utter_goodbye
18
19  ## Happy path 3
20  * greet
21    - utter_greet
22  * query_knowledge_list
23    - action_query_list
24  * query_details
25    - action_query_details
26  * goodbye
27    - utter_goodbye
28
29  ## Happy path 4
30  * greet
31    - utter_greet
32  * query_knowledge_list
33    - action_query_list
34  * query_knowledge_attribute_of
35    - action_query_attribute
36  * query_details
37    - action_query_details
38  * query_details
39    - action_query_details
40  * goodbye
41    - utter_goodbye
42
43  ## Delete personal doctor
44  * delete_personal_doctor
45    - action_delete_personal_doctor
46
47  ## Information personal doctor
48  * information_personal_doctor
49    - action_information_personal_doctor
50
51  ## Query Knowledge Base
52  * query_knowledge_list
53    - action_query_list
54  * affirm
55    - action_personal_doctor
56    - slot{"feedback_value" : "si"}
57
58  ## Query Knowledge Base
59  * query_knowledge_list
60    - action_query_list
61  * deny
62    - action_personal_doctor
63    - slot{"feedback_value" : "no"}
64
65  ## Query Knowledge Base
66  * query_knowledge_list
67    - action_query_list
68  * query_details
69    - action_query_details
70  * personal_doctor
71    - action_personal_doctor
72
73  ## Query Knowledge Base
74  * query_details
75    - action_query_details
76
77  ## Hello
78  * greet
79  - utter_greet
80
81  ## Bye
82  * goodbye
83  - utter_goodbye
84
85  ## bot challenge
86  * bot_challenge
87    - utter_iamabot
88
89  ## Query Knowledge Base
90  * query_knowledge_list
91    - action_query_list
92    - slot{"object_type": "ambulatorio"}
93
94  ## Query Knowledge Base
95  * query_knowledge_attribute_of
96    - action_query_attribute
97    - slot{"object_type": "medico"}
98
99  ## Query Knowledge Base
100 * query_knowledge_attribute_of
101   - action_query_attribute
```

```
102     - slot{"object_type": "ambulatorio"}
103
104   ## Query Knowledge Base
105   * query_knowledge_attribute_of
106     - action_query_attribute
107     - slot{"object_type": "orario"}
108
109   ## Query Knowledge Base
110   * query_knowledge_attribute_of{"attribute": "nome", "cognome": "Torino"}
111       - slot{"attribute": "nome"}
112       - slot{"cognome": "Torino"}
113       - action_query_attribute
114       - slot{"object_type": "medico"}
115       - slot{"attribute": null}
116       - slot{"mention": null}
117       - slot{"knowledge_base_last_object": "TORINO"}
118       - slot{"knowledge_base_last_object_type": "medico"}
119       - slot{"cognome": null}
120   * query_knowledge_list{"object_type": "medico", "comune": "Torino"}
121       - slot{"comune": "Torino"}
122       - slot{"object_type": "medico"}
123       - action_query_list
124       - slot{"object_type": "medico"}
125       - slot{"mention": null}
126       - slot{"attribute": null}
127       - slot{"knowledge_base_last_object": null}
128       - slot{"knowledge_base_last_object_type": "medico"}
129       - slot{"knowledge_base_listed_objects": ["D ADDONA", "D ADDONA", "D ADDONA", "D ADDONA", "D ADDONA", "POGLIANO"
                , "POGLIANO", "POGLIANO", "POGLIANO", "POGLIANO", "BUSCA", "BUSCA", "BUSCA", "BUSCA", "BUSCA", "
                GRASSINO SANTORO", "GRASSINO SANTORO", "GRASSINO SANTORO", "GRASSINO SANTORO", "GRASSINO SANTORO", "
                MOSCA", "MOSCA", "MOSCA", "MOSCA", "MOSCA"]}
130       - slot{"comune": null}
131   * query_knowledge_list{"object_type": "ambulatorio", "comune": "Torino"}
132       - slot{"comune": "Torino"}
133       - slot{"object_type": "ambulatorio"}
134       - action_query_list
135       - slot{"object_type": "ambulatorio"}
136       - slot{"mention": null}
137       - slot{"attribute": null}
138       - slot{"knowledge_base_last_object": null}
139       - slot{"knowledge_base_last_object_type": "ambulatorio"}
140       - slot{"knowledge_base_listed_objects": ["TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO",
                "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "
                TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO", "TORINO"]}
141       - slot{"comune": null}
142   * query_knowledge_list{"object_type": "medico", "comune": "Torino"}
143       - slot{"comune": "Torino"}
144       - slot{"object_type": "medico"}
145       - action_query_list
146       - slot{"object_type": "medico"}
147       - slot{"mention": null}
148       - slot{"attribute": null}
149       - slot{"knowledge_base_last_object": null}
150       - slot{"knowledge_base_last_object_type": "medico"}
151       - slot{"knowledge_base_listed_objects": ["D ADDONA", "D ADDONA", "D ADDONA", "D ADDONA", "D ADDONA", "POGLIANO"
                , "POGLIANO", "POGLIANO", "POGLIANO", "POGLIANO", "BUSCA", "BUSCA", "BUSCA", "BUSCA", "BUSCA", "
                GRASSINO SANTORO", "GRASSINO SANTORO", "GRASSINO SANTORO", "GRASSINO SANTORO", "GRASSINO SANTORO", "
                MOSCA", "MOSCA", "MOSCA", "MOSCA", "MOSCA"]}
152       - slot{"comune": null}
```

Listato 5.9. Rappresentazione del file `stories.md`

I nomi con cui vengono dichiarate le *story* non devono rispettare una particolare logica, tanto che è possibile che diverse *story* abbiano lo stesso nominativo.

All'interno del file `stories.md` è possibile osservare come le *story* possano contenere degli *slot* gia valorizzati. Di fatto, alla riga 89, si ha un esempio di quanto accennato: è stato definito lo *slot* `object_type` col valore "medico". Questa soluzione è stata sviluppata con lo scopo di riuscire a popolare lo *slot* `object_type` su ogni conversazione `query_knowledge_attribute_of`, dal momento che tale soluzione risulta l'unico metodo per trasferire informazioni alle relative *action*.

Di conseguenza, come per la *story* appena descritta, la motivazione è valida anche per le altre equivalenti che presentano *slot* valorizzati.

A questo punto è doveroso porre l'attenzione su una soluzione sviluppata alla riga 109 del file dove, attraverso la funzionalità di addestramento interattivo messo a disposizione dal framework Rasa, è stato possibile disambiguare una problematica riguardante il doppio significato del termine "Torino". Quest'ultimo veniva rilevato dal bot in due differenti modalità: come una *entity* di tipo "comune", e come una di

tipo “cognome”. Dunque, tale funzionalità di Rasa ha permesso di risolvere questo caso consentendo di correggere le errate interpretazioni delle richieste dell'utente e, in seguito, riportando la soluzione esatta all'interno di una nuova *story*. Di conseguenza, il bot, a seguito della correzione, è venuto in possesso di un nuovo modello aggiornato contenente una soluzione di disambiguazione sul termine problematico.

5.2.6 actions.py

Il file `actions.py` rappresenta, volendo semplificare, la logica con cui il chatbot elabora le risposte da fornire all'utente. Al suo interno, sono definite le sei *action* di cui si è anticipato nella Sottosezione 5.2.3, ovvero:

1. `action_query_list`
2. `action_query_attribute`
3. `action_query_details`
4. `action_personal_doctor`
5. `action_information_personal_doctor`
6. `action_delete_personal_doctor`

Tali *action* ereditano le loro funzionalità dalla superclasse `ActionPersona`, la quale è definita all'interno del file `actions.py`.

La seguente classe presenta, principalmente, l'implementazione di due metodi indispensabili per la visualizzazione della risposta del bot, ovvero il metodo `utter_objects`, il quale è definito per la visualizzazione della lista degli `object_type` ricercati (ad esempio, i medici), ed il metodo `utter_attribute_value`, il quale si occupa della visualizzazione degli `entity` richiesti dall'utente (ad esempio, il telefono di un dottore).

Il metodo `utter_objects`, mostrato nel Listato 5.10, alla riga 12 esegue una prima verifica affinchè le variabili `attributes`, che contengono gli attributi che devono essere soddisfatti dal risultato della richiesta dell'utente, e `objects`, che contengono i risultati della query sul database, non risultino comtemporaneamente nulle. Nel caso non lo fossero, alla riga 18 viene creata la prima sezione del messaggio di risposta; questo sarà poi rappresentato all'interno della stringa `result_string`, la quale si concatena, passo dopo passo, alle altri parti.

Successivamente, la seconda parte del messaggio generato sarà condizionata dal numero dei risultati ottenuti interrogando il database. Più dettagliatamente, nel caso in cui si ottengono da uno a sei risultati, questi verrebbero restituiti nel messaggio all'utente; se, invece, si ottengono un numero maggiore di sei, si ha un messaggio contenente la richiesta all'utente di definire dei parametri di ricerca aggiuntivi.

Per quanto concerne la rappresentazione dei risultati ricercati, è stato definito un metodo in grado di restituire un pattern per ognuno degli `object_type` gestiti dal chatbot. Al suo interno sono definite la sintassi della risposta e le informazioni da visualizzare; la funzione preposta a tale compito è denominata `get_representation_function_of_object` e appartiene alla classe `Neo4jKnowledgeBase` (riga 37).

I pattern sono modificati all'avvio di una *action* attraverso il metodo denominato `reset_entities_parameter` della classe `file_util`, il quale utilizza

al suo interno la funzione `set_representation_function_of_object` della classe `Neo4jKnowledgeBase`. Quest'ultima, a seconda dell'`object_type` della richiesta, stabilisce una delle seguenti rappresentazioni dei risultati:

- "medico": "nome" "cognome";
- "ambulatorio": "indirizzo" "civico" a "comune";
- "orario": [il dottore|la dottoressa|il pediatra|la pediatra] "nome" "cognome" riceve "giorno" dalle "orariodiapertura" alle "orariodichiusura" a "comune".

Inoltre, nella condizione di risultato singolo, a seconda dell'`object_type` rilevato, viene generata una risposta personalizzata (righe 39-70). Ad essa, nel caso dell'`object_type` "medico", è stato aggiunto un `button` che, nel caso in cui viene premuto, salva il risultato come medico di fiducia dell'utente (riga 45-56). Invece, se si ha un singolo risultato e l'`object_type` è di tipo "ambulatorio", viene definito e aggiunto un `button` alla risposta del chatbot (riga 61-70). Il caso che presenta l'`object_type` di tipo "orario" è stato gestito di seguito nel codice.

Come si può notare, alle righe 70, 75 e 79 si utilizza una variabile intera, denominata `result_type`, allo scopo di facilitare l'implementazione dei tre casi di risposta a seconda del numero di risultati.

1. `result_type` = 1: risultato univoco nella variabile `objects`;
2. `result_type` = 2: da 2 a 6 risultati nella variabile `objects`;
3. `result_type` = 3: maggiore di 6 risultati nella variabile `objects`.

Dalla riga 81 alla 105, è stata sviluppata la gestione dell'`object_type` "orario", ignorato fino ad ora a causa della sua complessa rappresentazione.

Innanzitutto, prima dell'inizio di tale implementazione, la variabile `objects`, definita come *List* di *Dictionary*, è stata riordinata secondo il campo "giorno" di ogni item. Questo passaggio è stato eseguito attraverso il metodo `sort_date(objects)` della classe `file_util`.

Dopodichè, ogni campo "giorno" di ogni elemento della lista `objects` è stato modificato nel caso in cui tali attributi fossero definiti con i valori del giorno odierno o di quello successivo. A tali condizioni, questi valori verrebbero sostituiti con i termini "oggi" e "domani" attraverso il metodo `file_util.get_list_date(objects)`.

Dalla riga 81, ha inizio la creazione del messaggio di risposta nella stringa `result_string`; la sua composizione complessa è dovuta alla volontà di rappresentare gli orari settimanali di ogni medico riportando nel messaggio un'unica volta il dottore a cui si riferiscono tali risultati.

Tale problematica ha origine dallo stato della rappresentazione degli orari sul database, perciò ogni elemento della lista `objects` viene inserito all'interno di un pattern di rappresentazione, diverso a seconda dell'`object_type`. Ad esempio, nel caso di `object_type` equivalente a "medico", il pattern di rappresentazione del risultato è formato dalla concatenazione dei campi "nome" e "cognome". Da tutto ciò, ne consegue l'obiettivo di evitare che tale pattern di rappresentazione del risultato venga inserito nel messaggio di risposta per ogni orario riferito al singolo dottore, ma piuttosto venga inserito un'unica volta.

Il pattern di rappresentazione degli orari viene utilizzato nel singolo caso in cui sia richiesto un unico orario di uno specifico dottore in una precisa struttura ricettiva (riga 86). Nei restanti casi, come accennato in precedenza, per ovviare alle

problematiche, la rappresentazione del risultato è definita direttamente concatenando gli attributi degli orari all'interno della stringa `result_string` a, seconda dei casi.

Dunque, alla riga 97 è possibile riscontrare il primo caso considerato, ovvero la condizione nella quale la query sul database ha fornito diversi risultati per diversi dottori, nel quale un dottore possiede un singolo orario da far visualizzare. Perciò, alla stringa `result_type` viene concatenata la rappresentazione dell'`object_type` "orario" con l'ausilio dei metodi `get_result_number`, in grado di convertire in aggettivo ordinale il numero relativo al risultato corrente, e il metodo `get_doctor_type`, il quale permette di ottenere il titolo professionale e il genere da associare al nominativo del dottore. Mentre, per quanto concerne le informazioni con cui generare il `result_type`, si è utilizzata una variabile di tipo *dictionary*, denominata `_entities`, la quale è stata precedentemente popolata con le informazioni contenute in file `.json` chiamato `file_utils.json`. Tale file verrà presentato nella successiva sottosezione 5.2.8.

La seconda situazione trattata si verificherebbe nel momento in cui i risultati da rappresentare fossero i diversi orari appartenenti ad un singolo dottore.

All'interno di questo contesto, il nominativo del dottore è riportato una singola volta (riga 100), e ad esso è stato concatenato il primo risultato a lui associato (riga 101). Dopodichè, i restanti orari sono stati aggiunti alla riga 104.

In seguito, nel codice, si sono gestiti i casi con `result_type` equivalente a "2" e "3", nei quali sono state definite delle frasi aggiuntive da far visualizzare all'utente.

Infine, il messaggio generato a seguito dei passaggi illustrati precedentemente, viene esposto in output attraverso la classe `dispatcher`, la quale incapsula il messaggio all'interno di un `dictionary` che viene passato ai connettori `GoogleConnector` e `AlexaConnector` per essere inviato alle piattaforme di visualizzazione.

```
1       def utter_objects(self, dispatcher: CollectingDispatcher, object_type: Text, objects: List[Dict[Text, Any]],
2           attributes: List[Dict[Text, Text]]) -> None:
3           """
4           Utters a response to the user that lists all found objects.
5           Args:
6               dispatcher: the dispatcher
7               object_type: the object type
8               objects: the list of objects
9           """
10          personal_doctor = ""
11          result_string = ""
12          if (not attributes or len(attributes)==0) and not objects:
13              dispatcher.utter_message(
14                  "Non ho capito a chi fossi interessato, prova a spiegarmi diversamente cosa cerchi."
15              )
16              return
17          else:
18              if attributes and (len(objects) != 1):
19                  if object_type != self.file_util.get_object_types()[2]:
20                      result_string = "Per la condizione "
21                      lis = []
22                      for attribute in attributes:
23                          synfound = False
24                          for attribute_syn_list in self.knowledge_base.attribute_syn:
25                              if attribute["name"] in attribute_syn_list:
26                                  synfound = True
27                                  synlist = []
28                                  for syn in attribute_syn_list:
29                                      synlist.append(syn)
30                                  lis.append("{}".format(", ".join(synlist)))
31                          if not synfound:
32                              lis.append("{}".format(attribute["value"]))
33                      result_string += "{} ".format(", ".join(lis))
34
35          buttons = []
36          button_text = ""
37          repr_function = self.knowledge_base.get_representation_function_of_object(object_type)
```

```
38          result_type = 1 #se 1 allora nulla, se 2 richiede quale risultato l'utente preferisce, se 3 richiede di
                definire un altro parametro
39          if objects:
40              logger.debug("length: " + str(len(objects)))
41              if (len(objects) == 1):
42                  if object_type != "orario":
43                      _ob = self.file_util.get_references(object_type, False)
44                      #logger.debug("ottenuto _ob")
45                      if object_type == "medico":
46                          _ob = self.file_util.get_doctor_type(object_type, objects[0], "singolare", "articolo") + " "
47                          button_text = ".\nVuoi salvare il dottore come medico di riferimento?"
48                          buttons = [{
49                              "title" : "chooseYes",
50                              "text" : "Si",
51                              "payload" : '/personal_doctor{"feedback_value": "si"}'
52                              },
53                              {"title": "chooseNo",
54                              "text" : "No",
55                              "payload" : '/personal_doctor{"feedback_value": "no"}'
56                              }]
57                      else:
58                          _ob = self.file_util.get_doctor_type(object_type, objects[0], "singolare", "articolo") + " in
                                "
59                      result_string = "Ho trovato " + _ob + repr_function(objects[0])
60
61                      if object_type == "ambulatorio":
62                          result_string += ".\nSe mi hai contattato da uno smartphone o PC, puoi trovare la sua
                                posizione a questo link "
63                          buttons = [{
64                              "type":"web_url",
65                              "title":"link",
66                              "text": "Google Maps",
67                              "payload": self.file_util.google_maps(str(str(objects[0]["indirizzo"]).replace(" ", "+")
                                    +"+"+ objects[0]["civico"] +"+"+ objects[0]["cap"] +"+"+ objects[0]["comune"]))
68                          }]
69                      #logger.debug("definito primo segmento di risposta: " + result_string)
70                      result_type = 1
71                  elif (len(objects) in range(2,6)):
72                      if object_type != "orario":
73                          _ob = self.file_util.get_references(object_type, True)
74                          result_string += "ho trovato i seguenti "+ _ob + ": \n"
75                      result_type = 2
76                  elif (len(objects) >= 6 ):
77                      _ob = self.file_util.get_references(object_type, True)
78                      result_string += "ho trovato molti "+ _ob + ", "
79                      result_type = 3
80
81                  repr_list = [] #per non ripetere sempre la dr nome cognome riceve ...
82                  if object_type == "orario":
83                      objects = self.file_util.sort_date(objects)
84                      objects = self.file_util.get_list_date(objects)
85                      if result_type == 1: #stampo risultato preciso
86                          result_string += "{} \n".format(repr_function(objects[0]))
87                      elif result_type == 2: #stampo massimo 7 risultati
88                          for i, obj in enumerate(objects, 1):
89                              if i <= 5: #limite di risultati posto a 5
90                                  if obj[self._entities[object_type]["param6"]]+obj[self._entities[object_type]["param7"]]+
                                        obj[self._entities[object_type]["param5"]] not in repr_list: #dottore ancora da
                                        presentare nei risultati
91                                      object_len = 0
92                                      for ob in objects: #conto gli orari del singolo dottore
93                                          if obj[self._entities[object_type]["param6"]] == ob[self._entities[object_type]["
                                                param6"]]: #obj == ob
94                                              object_len += 1
95
96                                      if object_len == 1: #se si ha un risultato per dottore
97                                          result_string += "{} risultato è{} {} {} e riceve \n".format(self.file_util.
                                                get_result_number(str(i)), self.file_util.get_doctor_type(object_type, obj
                                                , "singolare", "articolo"), obj[self._entities[object_type]["param6"]],
                                                obj[self._entities[object_type]["param7"]])
98                                          result_string += " {}; \n\n".format(obj[self._entities[object_type]["param3"]] +"
                                                dalle " + obj[self._entities[object_type]["param1"]] + " alle " + obj[self
                                                ._entities[object_type]["param2"]] + " a " + obj[self._entities[
                                                object_type]["param5"]])
99                                      elif object_len > 1: #se si hanno più risultati per dottore
100                                         result_string += "{} {} {} riceve nei seguenti orari a {}:\n".format(self.
                                                file_util.get_doctor_type(object_type, obj, "singolare", "articolo"), obj[
                                                self._entities[object_type]["param6"]], obj[self._entities[object_type]["
                                                param7"]], obj[self._entities[object_type]["param5"]])
101                                         result_string += "{} risultato è{}, \n".format(self.file_util.get_result_number(
                                                str(i)), obj[self._entities[object_type]["param3"]] +" dalle " + obj[self.
                                                _entities[object_type]["param1"]] + " alle " + obj[self._entities[
                                                object_type]["param2"]])
102                                         repr_list.append(obj[self._entities[object_type]["param6"]]+obj[self._entities[
                                                object_type]["param7"]]+obj[self._entities[object_type]["param5"]])
103                                     else: #stampo l'orario in caso si avesse già presentato il dottore
104                                         result_string += "{} è{}, \n".format(self.file_util.get_result_number(str(i)), obj[
                                                self._entities[object_type]["param3"]] +" dalle " + obj[self._entities[
                                                object_type]["param1"]] + " alle " + obj[self._entities[object_type]["param2"
                                                ]])
105                             result_string += "che altro ti interessa sapere? "
106
107                 elif result_type == 2:
108                     for i, obj in enumerate(objects, 1):
```

```
109                 result_string += "{} è{}, \n".format(self.file_util.get_result_number(str(i)), repr_function(obj
                        ))
110             result_string += "che altro ti interessa sapere? "
111
112         if result_type == 3:
113             result_string += "potresti darmi altre informazioni?"
114         logger.debug("stampo result_string: " + result_string)
115
116         if buttons != []:
117             result_string += button_text
118             dispatcher.utter_button_message(result_string, buttons)
119         else:
120             dispatcher.utter_message(result_string)
121     else:
122         dispatcher.utter_message(
123             " Mi spiace, non ho trovato nessun risultato. "
124         )
```

Listato 5.10. Codice del metodo `utter_objects`

La seconda funzione essenziale per la visualizzazione del risultato di Medici-bot è il metodo `utter_attribute_value`, il quale definisce la rappresentazione dell'*intent* query_knowledge_attribute_of (Listato 5.11).

Il seguente metodo non effettua alcuna distinzione sull'`object_type` a cui fa riferimento l'attributo ricercato dall'utente; pertanto, la prima parte (righe 13-32) si pone come obiettivo quello di definire il pattern di rappresentazione del risultato in relazione all'`object_type`.

Le variabili `object_representation` e `name_surname_representation` contengono le definizioni di due pattern di rappresentazione. Il primo, per ogni elemento della lista `objects` e attraverso il metodo `get_utter_attribute_rappresentation` della classe `file_util`, permette di definire il seguente pattern per i tipi di oggetti "medico" e "orario".

- "medico" e "orario": [del dottore|della dottoressa|il pediatra|la pediatra] "nome" "cognome".

Il secondo pattern di rappresentazione è costituito esclusivamente dalla concatenazione dei campi "nome" e "cognome" per ciascuno degli elementi della lista `objects`.

La seconda parte del metodo (righe 34-47) si concentra nell'effettuare una suddivisione della rappresentazione secondo il numero dei risultati ottenuti dal database; perciò, con la stessa modalità usata dal metodo precedente, è stata definita la variabile `result_type`. È importante osservare come il risultato in output verrà trasmesso solo nel caso di `result_type` uguale a "1". Oltre a valorizzare tale variabile, all'interno di questa parte del codice viene popolata la prima sezione della stringa `output_string`, che contiene il messaggio destinato all'utente.

La terza ed ultima parte del metodo (righe 50-93) si occupa della definizione della restante porzione della stringa di risposta.

Come è possibile osservare dalla riga 52 del codice, nel caso di `result_type` uguale a "2", possiamo notare come la stringa `output_string` venga concatenata con i risultati rappresentati dal pattern `name_surname_representation`. Inoltre, all'interno della stringa di output, viene aggiunta la richiesta all'utente di dichiarare quale dei risultati avesse richiesto.

Al contrario, nel caso di `result_type` uguale a "1", la stringa di output è valorizzata in base al fatto che gli attributi da rappresentare siano singoli o maggiori dell'unità (riga 59). In entrambe le casistiche, la stringa `output_string` viene ridefinita da capo utilizzando il pattern di rappresentazione `object_representation`.

Infine, nel caso di `result_type` uguale a "3", il metodo non restituisce alcuno dei risultati ottenuti dal database, bensì valorizza la stringa di output con una richiesta di specificare maggiori informazioni su quanto l'utente stesse cercando.

```
1       def utter_attribute_value(self, dispatcher: CollectingDispatcher, object_name: Text, object_type: Text,
2           objects: List[Dict[Text, Any]], attribute: Text, attributes: List[Dict[Text,Text]],
3           key_attribute: Text) -> None:
4           """
5           Utters a response that informs the user about the attribute value of the
6           attribute of interest.
7           Args:
8               dispatcher: the dispatcher
9               object_name: the name of the object
10              attribute_name: the name of the attribute
11              attribute_value: the value of the attribute
12          """
13          output_string = ""
14          list_value = []
15          list_object = []
16          object_rep = []
17          name_surname_rep = []
18          repr_function = self.knowledge_base.get_representation_function_of_object(object_type)
19          for object_of_interest in objects:
20              value = object_of_interest[attribute]
21              #logger.debug("valori oggetti : " + value)
22              if value not in list_value:
23                  list_value.append(value)
24                  list_object.append(object_of_interest)
25                  object_representation = self.file_util.get_utter_attribute_rappresentation(object_type,
                            object_of_interest)
26                  name_surname_representation = self.file_util.get_doctor_type(object_type, object_of_interest, "
                            singolare" , "articolo") + " " + object_of_interest["nome"] + " " + object_of_interest["
                            cognome"]
27                  if not object_representation:
28                      object_representation = repr_function(object_of_interest)
29                  if  object_representation not in object_rep:
30                      object_rep.append(object_representation)
31                      name_surname_rep.append(name_surname_representation)
32                  #logger.debug("valori oggetti in lista: " + value + "object_representation" + object_representation)
33
34          att_val = []
35          att = []
36          for a in attributes:
37              att_val.append(a["value"])
38              att.append(a["name"] + " " + a["value"])
39          #definisco il result_type della risposta
40          result_type = 1
41          logger.debug("numero dei risultati: " + str(len(list_value)) + " per " + object_type)
42          if  len(object_rep) in range(2,3) :
43              output_string += "Ho trovato " + str(len(object_rep)) + " " + self.file_util.get_references(object_type
                        , True) + " con " + " e ".join(att) + ", "
44              result_type = 2
45          elif len(object_rep) > 3:
46              output_string += "Ho trovato " + str(len(object_rep)) + " " + self.file_util.get_references(object_type
                        , True) + " con " + " e ".join(att) + ". "
47              result_type = 3
48
49          logger.debug("stampo i risultati e result_type = " + str(result_type) + " e object_type: " + object_type)
50          list_value = list(filter(None,list_value))
51          if list_value:
52              if result_type == 2:
53                  for ob in name_surname_rep:
54                      logger.debug("object_representation: %s", ob)
55                      output_string += "{}, ".format(ob)
56                  output_string += "\n"
57                  output_string += "a quale dei risultati ti stai riferendo? \n"
58              elif result_type == 1:
59                  if len(list_value) > 1:
60                      output_string = "Ho trovato {} {} {} e ".format(len(list_value), attribute, object_rep[0])
61                      for value in list_value:
62                          if value != "":
63                              output_string += "{} è{}, ".format(self.file_util.get_result_number(str(list_value.index(
                                        value)+1)), value)
64                      output_string += "\n"
65                  elif len(list_value) == 1:
66                      if attribute[0] in ["a","e","i","o","u"]: #definisco l'articolo da associare all'attributo
67                          articolo = "l'"
68                      else:
69                          articolo = "il "
70                      output_string ="{}{} di {} è{}.\n".format(articolo, attribute, object_representation, list_value
                                [0])
71                      if object_type == "orario" or object_type == "medico":
72                          output_string ="{}{} {} è{}.\n".format(articolo, attribute, object_representation, list_value
                                    [0])
73              elif result_type == 3:
74                  output_string += "Potresti precisare la tua richiesta? \n"
75                  if attributes:
76                      att = attributes[0]["name"]
```

```
77                elif attribute:
78                    att = attribute
79                else:
80                    att = ""
81                ob_type = self.knowledge_base.get_object_type_by_attribute(att)
82                logger.debug("ob_type" + ob_type)
83                ob_type = self.file_util.get_table(ob_type)
84                list_param = self.file_util.get_param_object(ob_type)
85                output_string += "Prova a dirmi " + ", ".join(list_param)
86            #stampo il messaggio finale
87            dispatcher.utter_message(output_string)
88        else:
89            dispatcher.utter_message(
90                "Non ho trovato nessun {} di {} .".format(
91                    attribute, object_name
92                )
93            )
```

Listato 5.11. Codice del metodo `utter_attribute_value`

I due metodi appena descritti sono contenuti nella superclasse `ActionPersona`, mentre, nella successiva parte della sottosezione, saranno trattate le sei classi in cui sono implementate le *action* del chatbot.

La prima *action* che sarà presentata è l'`action_query_list`, la quale è definita nella classe `ActionPersonaList`, con le funzionalità ereditate dalla superclasse `ActionPersona`.

Al suo interno, i metodi definiti sono `run` e `_query_objects_my`; il primo è la funzione richiamata nel momento in cui l'*action* viene invocata, mentre il secondo è il metodo dedicato all'elaborazione del messaggio destinato all'utente.

Nel Listato 5.12 è presente il codice del metodo `run`, nel quale è stata definita la variabile `object_type` con le informazioni del relativo *slot*. Quest'ultimo è contenuto all'interno della classe *tracker*, la quale si occupa di trasferire e tenere traccia delle informazioni rilevate nella fase di classificazione dell'*intent* e di estrazione delle *entity*.

Di seguito, il metodo effettua una modifica di alcune funzionalità del chatbot adattandole all'`object_type` rilevato durante la sua esecuzione (riga 7). A tale scopo viene utilizzata la funzione `reset_entities_parameter` e, grazie ad essa, i pattern di rappresentazione dell'`object_type` e il campo univoco con cui identificare i singoli oggetti del tipo `object_type` rilevato dalla richiesta dell'utente subiscono una ridefinizione.

Infine, al metodo `_query_objects_my` sarà trasferito il compito di generare il messaggio destinato all'utente (riga 29).

```
1       def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text,
              Any]]:
2
3           logger.info("action_query_list")
4           object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
5
6           if object_type:
7               self.file_util.reset_entities_parameter(object_type)
8
9           #logger.info('query objects attr:'+str(attribute) +' new_req:'+str(new_request))
10          return self._query_objects_my(dispatcher, tracker)
11
12          dispatcher.utter_template("utter_ask_rephrase", tracker)
13          return []
```

Listato 5.12. Codice del metodo `run` della classe `ActionPersonaList`

Il metodo `_query_objects_my` può essere osservato all'interno del Listato 5.13 ed è la funzione designata all'interrogazione della knowledge base con il fine di ottenere gli oggetti richiesti e filtrati secondo gli attributi dichiarati dall'utente.

La prima sezione del codice si occupa della gestione delle "mention" (righe 24-46). Per iniziare, si esegue una controllo per verificare se effettivamente il valore riportato nello *slot* `SLOT_MENTION` sia senza alcun dubbio un numero che rappresenti la posizione dell'elemento che sostituisce nella lista `SLOT_LISTED_OBJECTS` (righe 25-28), e non una errata interpretazione di eventuali altri termini (ad esempio, considerando la frase "mi dici gli ambulatori aperti il primo del mese", il termine "primo" potrebbe essere erroneamente scambiato con una "mention").

Dopodichè, la funzione `knowledge_base.ordinal_mention_mapping` esegue la mappatura della "mention" con l'oggetto a cui fa riferimento; da cui è ricavato il campo identificativo utilizzato per la costruzione della query sul database. Di conseguenza, il metodo appena descritto è utilizzato per riferirsi ad un oggetto e rappresenta una delle modalità con cui ottenere gli attributi dell'oggetto voluto dall'utente. Il campo identificato dell'oggetto viene memorizzato nella variabile `object_identifier`.

Tuttavia, nel caso in cui l'utente non avesse espresso una "mention", il campo con cui identificare l'oggetto e da cui ricavare l'attributo ricercato dall'utente, verrà individuato all'interno del contenuto della variabile `attributes`. Essa è costituita dalle eventuali condizioni di ricerca espresse dall'utente.

Dunque, dopo aver definito i parametri con cui verrà eseguita la query sul database (la "mention" o gli `attributes`), si richiama il metodo `get_objects` della classe `Neo4jKnowledgeBase` (riga 49) per l'esecuzione della query sul DBMS, il quale restituisce i risultati all'interno di una lista di *dictionary*.

A questo punto (riga 66), i risultati contenuti nella variabile `ob` sono inseriti all'interno di un pattern di rappresentazione, e a seguito di tale passaggio, la gestione dell'elaborazione del messaggio di risposta all'utente viene trasferita al metodo `utter_objects` (tale funzione è stata trattata precedentemente).

Infine, all'interno di questo metodo si definiscono gli *slot* che saranno mantenuti durante la successiva richiesta dell'utente. In particolare, viene valorizzato lo *slot* `SLOT_LAST_ATTRIBUTE` con le informazioni riguardanti l'attuale conversazione (le informazioni sono salvate all'interno del *dictionary* `dict_object`) nell'eventualità che in futuro possa essere invocato l'*intent* `query_details`; di conseguenza, sarà necessario possedere la totalità dei dati attinenti alla corrente interazione.

```
1       def _query_objects_my(self, dispatcher: CollectingDispatcher, tracker: Tracker) -> List[Dict]:
2           """
3           Queries the knowledge base for objects of the requested object type and outputs those to the user. The
                    objects are filtered by any attribute the user mentioned in the request.
4           Args:
5               dispatcher: the dispatcher
6               tracker: the tracker
7           Returns: list of slots
8           """
9           object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
10          #listo tutti i nomi degli attributi di un object_type
11          object_attributes = self.file_util.get_attributes_of_object(object_type)
12
13          if not object_type:
14              logger.debug("not object_type")
15              self.utter_objects(dispatcher, object_type, None, None)
16              return reset_attribute_slots(tracker, object_attributes)
17
18          # get all set attribute slots of the object type to be able to filter the list of objects
19          attributes = get_attribute_slots(tracker, object_attributes)
```

```
20          logger.debug("attributes before: %s", attributes)
21          attributes = self.get_date(dispatcher, tracker, attributes, object_type) #merge the 2 list of attributes
22          object_identifier = None
23          logger.debug("attributes after: %s", attributes)
24          mention = tracker.get_slot(SLOT_MENTION)
25          for e in tracker.latest_message['entities']:
26              if e["entity"] == "mention" and (int(e["end"])-int(e["start"])) <= 2:
27                  mention = ""
28                  logger.debug(" ELIMINATA LA MENTION")
29          last_object = tracker.get_slot(SLOT_LAST_OBJECT)
30          if not attributes and mention:
31              listed_items = tracker.get_slot(SLOT_LISTED_OBJECTS)
32              last_object_type = tracker.get_slot(SLOT_LAST_OBJECT_TYPE)
33              current_object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
34              ordinal_mention_mapping = self.knowledge_base.ordinal_mention_mapping
35              if listed_items and mention in ordinal_mention_mapping:
36                  idx_function = ordinal_mention_mapping[mention]
37                  object_identifier = idx_function(listed_items)
38                  logger.debug("object_identifier: " + str(object_identifier))
39                  if current_object_type != last_object_type:
40                      ent = self.file_util.get_entities(last_object_type)
41                      last_object = ent['discriminante']
42                      logger.debug("last_object: " + str(last_object))
43                  else:
44                      last_object = None
45          else:
46              object_identifier = None
47
48          # query the knowledge base
49          ob = self.knowledge_base.get_objects(object_type, attributes, object_identifier, last_object)
50          repr_function = self.knowledge_base.get_representation_function_of_object(object_type)
51
52          list_object = []
53          objects = []
54          if ob:
55              for obj in ob:
56                  if repr_function(obj) not in list_object:
57                      list_object.append(repr_function(obj))
58                      objects.append(obj)
59
60          self.utter_objects(dispatcher, object_type, objects, attributes)
61
62          if not objects:
63              logger.debug("Objects not find")
64              return reset_attribute_slots(tracker, object_attributes)
65
66          key_attribute = self.knowledge_base.get_key_attribute_of_object(object_type)
67          listed = []
68          if object_type == "orario":
69              sorted_objects = self.file_util.sort_date(objects)
70              listed_objects = self.file_util.get_list_date(sorted_objects)
71              for ob in listed_objects:
72                  listed.append(ob["id_operatore"]+"#"+ob["id_raplav_ambulatorio"]+"#"+ob["giorno"])
73              #key_attribute = "id_operatore#id_raplav_ambulatorio#giorno"
74          else:
75              listed = list(map(lambda e: e[key_attribute], objects))
76
77          last_object = None if len(objects) > 1 else objects[0][key_attribute]
78
79          dict_object = {"object_name" : "",
80                         "object_type" : object_type,
81                         "objects" : objects,
82                         "attribute" : "no",
83                         "attributes" : attributes,
84                         "key_attribute" : ""}
85
86          slots = [
87              SlotSet(SLOT_OBJECT_TYPE, None),
88              SlotSet(SLOT_MENTION, None),
89              SlotSet(SLOT_ATTRIBUTE, None),
90              SlotSet(SLOT_LAST_ATTRIBUTE, dict_object),
91              SlotSet(SLOT_LAST_OBJECT, last_object),
92              SlotSet(SLOT_LAST_OBJECT_TYPE, object_type),
93              SlotSet(SLOT_LISTED_OBJECTS, listed),
94          ]
95          return slots + reset_attribute_slots(tracker, object_attributes)
```

Listato 5.13. Codice del metodo `_query_objects_my` della classe `ActionPersonaList`

La seconda *action* che prendiamo in considerazione è l'`action_query_attribute` ed è definita all'interno della classe `ActionAttributoPersona`; essa è presentata a partire dal suo metodo di invocazione denominato `run` (Listato 5.14).

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

```
2           """
3           Executes this action. If the user ask a question about an attribute, the knowledge base is queried for that
                attribute. Otherwise, if no attribute was detected in the request or the user is talking about a
               new object type, multiple objects of the requested type are returned from the knowledge base.
4           Args:
5               dispatcher: the dispatcher
6               tracker: the tracker
7               domain: the domain
8           Returns: list of slots
9           """
10          logger.info("action_query_attribute_of")
11          object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
12          last_object_type = tracker.get_slot(SLOT_LAST_OBJECT_TYPE)
13          attribute = tracker.get_slot(SLOT_ATTRIBUTE)
14
15          #new_request = object_type != last_object_type
16
17          if not object_type and tracker.get_slot(SLOT_MENTION): #in caso di mention, object_type uguale a quello
                precedente
18              object_type = last_object_type
19              logger.debug("ridefinisco object_type per mention: " + object_type)
20
21          object_attributes = self.file_util.get_attributes_of_object(object_type) #object_type non serve in realtà
22          # list of objects
23          attributes = get_attribute_slots(tracker, object_attributes)
24          #merge the 2 list of attributes
25          logger.debug("ATTRIBUTES dopo aggiunta: %s", attributes)
26          logger.debug("attribute: %s", attribute)
27          if not object_type:
28              if attributes:
29                  att = attributes[0]["name"]
30              elif attribute:
31                  att = attribute
32              else:
33                  att = ""
34              ob_type = self.knowledge_base.get_object_type_by_attribute(att)
35              logger.debug("ob_type: " + ob_type)
36              object_type = self.file_util.get_table(ob_type)
37
38              logger.debug("object_type before definito in _query_attribute: " + object_type)
39
40          if object_type:
41              self.file_util.reset_entities_parameter(object_type)
42
43          logger.info('query attribute attr: ' + str(attribute))
44          return self._query_attribute(dispatcher, tracker, object_type, attributes)
45
46          dispatcher.utter_template("utter_ask_rephrase", tracker)
47          return []
```

Listato 5.14. Codice del metodo `run` della classe `ActionAttributoPersona`

All'interno di tale metodo, la prima sezione del codice ha il compito di recuperare le informazioni necessarie alla corretta generazione del messaggio destinato all'utente. A tale scopo vengono memorizzati i valori degli *slot* `SLOT_OBJECT_TYPE`, `SLOT_LAST_OBJECT_TYPE` e `SLOT_ATTRIBUTE`. Tuttavia, la maggioranza delle volte, il primo *slot* non viene mai definito dall'utente; basti pensare alla richiesta "vorrei sapere il telefono del dottor Rossi" la quale non contiene alcun riferimento all'`object_type` della frase, ma esso è indispensabile dal punto di vista del funzionamento di Rasa.

A tal proposito, la problematica è stata risolta attraverso due possibili soluzioni per il recupero della suddetta informazione: la prima consiste nell'utilizzo dello *slot* `SLOT_LAST_OBJECT_TYPE`, recuperando così il dato memorizzato dell'interazione precedente (riga 17), mentre la seconda consiste nel recuperare l'`object_type` dal nominativo della tabella del database attraverso l'utilizzo del campo dell'attributo richiesto dall'utente. Ad esempio, nel caso della richiesta "vorrei sapere il telefono del dottor Rossi", verrebbe utilizzata l'*entity* con cui il termine "Rossi" è stato identificato, ovvero "cognome". Dopodichè, esso verrebbe usato per ricercare il nominativo del *nodo_type* che lo rappresenta nel database e verrebbe convertito, attraverso un mapping tra i due valori, con l'*object_type* a lui corrisposto. Il metodo utilizzato è `get_object_type_by_attribute`, il quale è definito all'interno della classe `Neo4jKnowledgeBase` (riga 34).

Di seguito, viene richiamato il metodo `set_entities_parameter` di cui si è già discusso precedentemente, e l'elaborazione del messaggio destinato all'utente viene passata alla funzione `_query_attribute`. Tale metodo, nella prima parte, recupera due importanti informazioni: la prima è l'attributo che è stato richiesto dall'utente, il quale viene ottenuto dallo *slot* `SLOT_ATTRIBUTE` (riga 10), mentre la seconda informazione risulta essere l'`object_name` (riga 11), ovvero il parametro utilizzato per riconoscere l'oggetto, o (nel caso fossero più di uno) gli oggetti, da cui ricavare l'attributo di cui si è appena accennato.

Nel dettaglio, l'`object_name` è ottenuto attraverso l'utilizzo della funzione `_get_object_name`; essa permette di ricavare il campo identificativo dell'oggetto in cui ricercare l'attributo richiesto al chatbot; questo è rappresentato mediante una *mention* dichiarata all'interno del messaggio dell'utente.

Inoltre, l'oggetto in cui ricercare l'attributo può essere ottenuto attraverso un ulteriore mezzo, ovvero servendosi della possibilità dell'utente di dichiarare nella sua richiesta una *entity*, la quale verrebbe interpretata dal metodo `_query_attribute` come un parametro su cui filtrare gli elementi all'interno del database. Detto ciò, nel momento in cui viene richiamato il metodo `get_objects`, il quale è parte della classe `Neo4jKnowledgeBase` ed è utilizzato per effettuare la query sul database, vengono passati, oltre ai parametri `object_type` e `object_name`, anche la variabile `attributes`, la quale contiene le informazioni riguardanti le *entity* estratte dal modello NLU.

Dopodichè, gli oggetti ottenuti dall'interrogazione del database vengono utilizzati dal metodo `utter_attribute_value` per la generazione del messaggio di risposta all'utente.

Infine, nell'ultima parte del codice sono definiti gli *slot* che il chatbot dovrà mantenere durante la sua prossima interrogazione; tra questi è possibile osservare nuovamente `SLOT_LAST_ATTRIBUTE` presente per le stesse ragioni viste per la *action* presentata.

```
1       def _query_attribute(self, dispatcher: CollectingDispatcher, tracker: Tracker, object_type: Text, attributes:
            List) -> List[Dict]:
2           """
3           Queries the knowledge base for the value of the requested attribute of the
4           mentioned object and outputs it to the user.
5           Args:
6               dispatcher: the dispatcher
7               tracker: the tracker
8           Returns: list of slots
9           """
10          attribute = tracker.get_slot(SLOT_ATTRIBUTE)
11          object_name = self._get_object_name(
12              tracker,
13              self.knowledge_base.ordinal_mention_mapping,
14              self.use_last_object_mention,
15          )
16
17          logger.info("_query_attribute [object_type]:"+str(object_type) + " [attribute]:"+str(attribute)+" [
                object_name]:"+str(object_name))
18
19          object_attributes = self.file_util.get_attributes_of_object(object_type)
20
21          if not object_name or not attribute:
22              logger.info("object_name or attribute not available")
23              dispatcher.utter_template("utter_ask_rephrase", tracker)
24              return [SlotSet(SLOT_MENTION, None)] + reset_attribute_slots(tracker, object_attributes)
25
26          # get all set attribute slots of the object type to be able to filter the
27          # list of objects
28          #attributes = get_attribute_slots(tracker, object_attributes)
29          #merge the 2 list of attributes
30
31          # query the knowledge base
32          objects = self.knowledge_base.get_objects(object_type,attributes,object_name, None)
```

```
33          #logger.debug("object_type before get_key_attribute " + object_type)
34          self.key_attribute = self.knowledge_base.get_key_attribute_of_object(object_type)
35          #logger.debug("key_attribute after get_key_attribute" + self.key_attribute)
36
37          if not objects or attribute not in objects[0]:
38              logger.info("object not found or attribute not in objects[0]")
39              dispatcher.utter_template("utter_ask_rephrase", tracker)
40              return [SlotSet(SLOT_MENTION, None)] + reset_attribute_slots(tracker, object_attributes)
41          self.utter_attribute_value(dispatcher, object_name, object_type, objects, attribute, attributes, self.
                key_attribute)
42
43          list_value = []
44          for object_of_interest in objects:
45              value = object_of_interest[attribute]
46              if value not in list_value:
47                  object_identifier = object_of_interest[self.key_attribute]
48                  list_value.append(value)
49
50          dict_object = {"object_name" : object_name,
51                         "object_type" : object_type,
52                         "objects" : objects, #passo i vecchi objects
53                         "attribute" : attribute,
54                         "attributes" : attributes,
55                         "key_attribute" : self.key_attribute}
56
57          slots = [
58              SlotSet(SLOT_OBJECT_TYPE, None),
59              SlotSet(SLOT_ATTRIBUTE, None),
60              SlotSet(SLOT_MENTION, None),
61              SlotSet(SLOT_LAST_OBJECT, object_identifier),
62              SlotSet(SLOT_LAST_OBJECT_TYPE, object_type),
63              SlotSet(SLOT_LAST_ATTRIBUTE, dict_object)
64          ]
65
66          return slots + reset_attribute_slots(tracker, object_attributes)
```

Listato 5.15. Codice del metodo `_query_attribute` della classe `ActionAttributoPersona`

A seguire, verrà presentata l'*action_query_details*, la quale è definita all'interno della classe `ActionDetails` (Listato 5.16).

La seguente classe, equivalentemente alle altre precedentemente illustrate, elabora le risposte delle richieste dell'utente all'interno del metodo `run`. Esso, nella sua prima parte, recupera le già discusse informazioni iniziali, definendo, tuttavia, una nuova variabile contenente i dati memorizzati all'interno dello *slot* `SLOT_LAST_ATTRIBUTE`. Quest'ultimo contiene le informazioni riguardanti la precedente richiesta elaborata (riga 18).

A questo punto, gli attributi precedenti sono aggiunti nella lista *attributes* insieme a quelli rilevati durante la relativa fase di estrazione delle *entity*, e per ognuno di essi viene verificata la loro presenza all'interno dei risultati elaborati dal precedente *intent* (riga 52-56).

La successiva parte del *metodo* rappresenta il fulcro dell'attività svolta dall'*action*, e ciò è dovuto al fatto che *action_query_details* costituisce null'altro che la continuazione, dal punto di vista del progresso di una conversazione e dal punto di vista dell'elaborazione del risultato, delle *action* dei due *intent* illustrati precedentemente. Di conseguenza, il processo di creazione della visualizzazione del risultato viene tresferito ai due metodi utilizzati dalle classi `ActionAttributoPersona` ed `ActionPersonaList`, i quali, rispettivamente, sono `utter_attribute_value` e `utter_objects`.

A seguito di quanto illustrato, la decisione su dove indirizzare l'elaborazione del risultato è determinata dal valore del campo `key_attribute` del *dictionary* `last_attribute`. Nel caso esso contenesse il valore nullo, il metodo che verrà utilizzato risulterebbe `utter_attribute_value` (riga 72), mentre, nel caso contrario, verrebbe richiamata la funzione `utter_objects`.

Una sezione significativa del codice, la quale viene riportata anche all'interno delle *action* precedentemente descritte, risulta essere la gestione riguardante l'identifi-

cativo degli oggetti di tipo “orario”, i quali non posseggono una chiave univoca all’interno del database. Tale problematica viene risolta definendo una nuova chiave composta dalla concatenazione degli attributi “id_operatore”, “id_raplav_ambulatorio” e “giorno”, con l’aggiunta del carattere “#” con funzione separatoria (righe 87-92).

In conclusione, a seguito delle attività sopra descritte, nell’ultima parte del codice, vengono dichiarati gli *slot* da azzerare e quelli da aggiornare con i relativi valori al fine di garantire la continuità della conversazione con l’utente.

```
1       class ActionDetails(ActionPersona):
2
3       def name(self):
4           return "action_query_details"
5
6       def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text,
            Any]]:
7           logger.info("action_query_details")
8
9           object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
10          if not object_type:
11              object_type = tracker.get_slot(SLOT_LAST_OBJECT_TYPE)
12
13          #try:
14          if object_type:
15              self.file_util.reset_entities_parameter(object_type)
16
17          attribute = tracker.get_slot(SLOT_ATTRIBUTE)
18          last_attribute = tracker.get_slot(SLOT_LAST_ATTRIBUTE) #il risultato della domanda precedente
19
20          #listo tutti i nomi degli attributi di un object_type
21          object_attributes = self.file_util.get_attributes_of_object(object_type)
22          attributes = get_attribute_slots(tracker, object_attributes) #nuovo attributo con cui filtrare i dati
23          logger.debug("modify list of attributes") #serve per stoppare di fare query_detail annidate
24          if not last_attribute:
25              logger.debug("NESSUN ATTRIBUTO LAST E' STATO TROVATO")
26              slots = [SlotSet(SLOT_LAST_OBJECT_TYPE, object_type)]
27              slot_detail = []
28              for ab in attributes: #elimino gli slot con gli attributi settati
29                  slot_detail.append(SlotSet(ab["name"], None))
30                  #logger.debug("slot da eliminare:" + ab["name"])
31              slots += slot_detail
32              dispatcher.utter_template("utter_ask_rephrase", tracker)
33              return slots + reset_attribute_slots(tracker, object_attributes)
34
35          attributes = attributes + last_attribute["attributes"]
36          #filtro gli attributi in caso ci fossero attributi doppi
37          output_att = []
38          z = []
39          for x in attributes:
40              if x["name"] not in z:
41                  z.append(x["name"])
42                  output_att.append(x)
43          attributes = output_att
44          logger.debug("attributes: %s", attributes)
45
46          object_identifier = ""
47          list_o = []
48          list_value = []
49          if last_attribute:
50              for object_of_interest in last_attribute["objects"]:
51                  #logger.debug("object_of_interest: %s", object_of_interest)
52                  flag_att = True
53                  for attributes_for in attributes:
54                      if attributes_for["value"].upper() != object_of_interest[attributes_for["name"]].upper():
55                          #logger.debug("flag false")
56                          flag_att = False
57                  if flag_att:
58                      list_o.append(object_of_interest)
59                      if last_attribute["attribute"] != "no":
60                          #logger.debug("value")
61                          value = object_of_interest[last_attribute["attribute"]]
62                          #logger.debug("value: " + value)
63                          if value not in list_value: #per query_list non serve value
64                              object_identifier = object_of_interest[last_attribute["key_attribute"]]
65                              list_value.append(value)
66                              #logger.debug("valori object_identifier in lista: " + object_identifier)
67                      else:
68                          logger.debug("no value")
69                  #else:
70                      #logger.debug("non sono stati rispettati i parametri")
71          if last_attribute["key_attribute"] != "":
72              self.utter_attribute_value(dispatcher, last_attribute["object_name"], object_type, list_o,
                    last_attribute["attribute"], attributes, last_attribute["key_attribute"])
73          else:
74              self.utter_objects(dispatcher, object_type, list_o, attributes)
```

```
75          logger.debug("lunghezza risultati: %s", len(list_value))
76          if len(list_value) < 2:
77              dict_object = None
78              logger.debug("elimino dati last")
79          else:
80              dict_object = {"object_name" : last_attribute["object_name"],
81                         "object_type" : object_type,
82                         "objects" : list_o,
83                         "attribute" : last_attribute["attribute"],
84                         "attributes" : attributes,
85                         "key_attribute" : last_attribute["key_attribute"]}
86          listed = []
87          if object_type == "orario":
88              sorted_objects = self.file_util.sort_date(list_o)
89              listed_objects = self.file_util.get_list_date(sorted_objects)
90              for ob in listed_objects:
91                  listed.append(ob["id_operatore"]+"#"+ob["id_raplav_ambulatorio"]+"#"+ob["giorno"])
92              key_attribute = "id_operatore#id_raplav_ambulatorio#giorno"
93          #else:
94              #logger.debug("key_attribute: " + last_attribute["key_attribute"])
95              #key_attribute = self.knowledge_base.get_key_attribute_of_object(object_type)
96              #listed = list(map(lambda e: e[key_attribute], list_o))
97              #listed = []
98
99          slots = [
100             SlotSet(SLOT_OBJECT_TYPE, None),
101             SlotSet(SLOT_ATTRIBUTE, None),
102             SlotSet(SLOT_MENTION, None),
103             SlotSet(SLOT_LAST_OBJECT, None),
104             SlotSet(SLOT_LAST_OBJECT_TYPE, object_type),
105             SlotSet(SLOT_LAST_ATTRIBUTE, dict_object),
106             SlotSet(SLOT_LISTED_OBJECTS, listed),
107         ]
108         slot_detail = []
109         for ab in attributes: #elimino gli slot con gli attributi settati
110             slot_detail.append(SlotSet(ab["name"], None))
111             logger.debug("slot da eliminare:" + ab["name"])
112         slots += slot_detail
113
114         return slots + reset_attribute_slots(tracker, object_attributes)
115         #except:
116         dispatcher.utter_template("utter_ask_rephrase", tracker)
117         return []
```

Listato 5.16. Codice della classe `ActionDetails`

I tre *intent* ancora da illustrare verranno proposti nella Sottosezione 5.3.4 a causa di alcune funzionalità che necessitano di essere introdotte preliminarmente.

5.2.7 neo4jstorage.py

Il file `neo4jstorage.py` contiene al suo interno la classe `Neo4jKnowledgeBase`, la quale eredita le sue funzioni dalla superclasse `KnowledgeBase` della libreria `rasa_sdk.knowledge_base.storage`; questa classe è stata creata con l'obiettivo di gestire la connessione con il DBMS e di definire le funzioni per la costruzione e l'elaborazioneses delle query sul database.

All'interno della classe `Neo4jKnowledgeBase` sono definiti tre metodi essenziali per il corretto funzionamento di Medici-bot:

- `get_objects`: si occupa della creazione della query e della sua esecuzione per l'ottenimento dei risultati richiesti dall'utente.
- `relation_mapping_function`: viene richiamato all'avvio del chatbot e permette di costruire lo schema dei dati contenuti sul DBMS. Esso consente di modificare lo schema sul database senza la necessità di modificare le query da eseguire.
- `get_object_type_by_attribute`: consente di ottenere l'`object_type` a partire da un suo attributo. È opportuno ricordare che ogni `object_type` è associato ad un *nodo_type* nel database.

Il metodo `get_objects`, illustrato nel Listato 5.17, si occupa di definire e di eseguire le query sul database, permettendo, di conseguenza, di ricavare le informazioni richieste dall'utente.

```
1   def get_objects(
2           self, object_type: Text, attributes: List[Dict[Text, Text]],object_identifier: Text, last_object: Text,
                limit: int = 5
3       ) -> List[Dict[Text, Any]]:
4
5           #fase di associazione delle relazioni alla query
6           self.set_entity_attribute_mapping(object_type, attributes)
7           #logger.debug("dict_attr: %s", self.dict_attr)
8
9           logger.debug("object_type: %s ", object_type)
10          #preparazione della fase match della query
11          query = "MATCH p = ("+ self.initial_object + ":" + self.initial_object +")" + self.relation_object
12
13          #definire il where in caso ci fosse una condizione esplicitata
14          if object_identifier or attributes:
15              logger.debug("attributes: %s ", attributes)
16              query += ' WHERE EXISTS('+ self.entity_mapping_function(object_type) +'.'+ self.
                        get_key_attribute_of_object(object_type) +') AND '
17              wherecond = []
18              #logger.debug("query parziale 1: " + query)
19          else:
20              return None
21
22          # fase di filtraggio degli oggetti in base agli attributi, preparazione query
23          if attributes:
24              for attribute in attributes:
25                  synfound = False
26                  for attribute_syn_list in self.attribute_syn:
27                      if attribute["name"] in attribute_syn_list:
28                          synfound = True
29                          synwherecond = []
30                          for syn in attribute_syn_list:
31                              synwherecond.append(self.attribute_field_function(object_type,syn)+
32                              "("+self.entity_mapping_function(object_type)+"."+syn+") "+ self.
                                        attribute_operator_function(object_type,syn) +" "+
33                              self.attribute_value_function(object_type,syn,attribute["value"].replace("'","\\'"))
34                              +" " )
35                          wherecond.append("("+ " OR ".join(synwherecond) +")")
36                          logger.info("ATTRIBUTE1: %s", wherecond)
37                  if not synfound:
38                      #logger.debug("list_dict: " + self.dict_attr[attribute["name"]])
39                      wherecond.append(self.attribute_field_function(object_type,attribute)+
40                      "("+self.dict_attr[attribute["name"]]+") "+ self.attribute_operator_function(object_type,
                                attribute["name"]) +" "+
41                      self.attribute_value_function(object_type,attribute["name"],attribute["value"].replace("'","\\'"
                                ))
42                      +" " )
43                      #logger.info("ATTRIBUTE2: %s", wherecond)
44          else: #caso in cui si ha delle mention
45              if "#" not in object_identifier:
46                  self.set_entity_ob_id_mapping(last_object)
47                  if last_object:
48                      tabella = self.dict_attr[last_object]
49                      logger.debug("tabella definito: " + tabella)
50                  else:
51                      tabella = self.entity_mapping_function(object_type) +"."+ self.get_key_attribute_of_object(
                                object_type)
52                  #logger.debug("object_identifier definito: " + object_identifier)
53                  #logger.debug("key_attribute_of_object definito: " + self.get_key_attribute_of_object(object_type))
54                  wherecond.append(
55                      "TOUPPER("+ tabella +
56                      ") = TOUPPER('" + object_identifier +"') ")
57                  #logger.debug("query parziale 1,1: " + query)
58              elif "#" in object_identifier:
59                  object_identifier = object_identifier.split("#")
60                  #"id_operatore#id_raplav_ambulatorio#giorno"
61                  attr_l = ["id_operatore", "id_raplav_ambulatorio", "giorno"]
62                  for cont_l, ob in enumerate(object_identifier, 0):
63                      #logger.debug("object_identifier separato " + str(cont_l) + ": " + ob)
64                      tabella = self.get_object_type_by_attribute(attr_l[cont_l]) +"."+ attr_l[cont_l]
65                      wherecond.append(
66                          "TOUPPER("+ tabella +
67                          ") = TOUPPER('" + ob +"') ")
68          query += " AND ".join(wherecond)
69          #logger.debug("query parziale 2: " + query)
70
71          query += ' RETURN nodes(p) As Result LIMIT 70'
72
73          #esecuzione della query
74          result = self.query_db(query)
75
76          objects = []
77
78          for record in result:
```

```
79              entity = {}
80              for rec in record['Result']:
81                  for k, v in rec.items():
82                      entity[k] = str(v)
83              objects.append(entity)
84
85          print(objects)
86
87          return objects
```

Listato 5.17. Codice del metodo `get_objects`

Un esempio della query generata da questo metodo è mostrato nel seguente Listato:

```
1       MATCH p = (Medici:Medici)-[WORK_IN]-(Ambulatori)-[Orario]-(Orari) WHERE EXISTS(Medici.id_operatore) AND TOUPPER
                (Medici.cognome) contains TOUPPER('rossi') AND TOUPPER(Medici.nome) contains TOUPPER('mauro') AND
                TOUPPER(Ambulatori.comune) contains TOUPPER('cigliano') RETURN nodes(p) As Result LIMIT 70
```

Listato 5.18. Esempio di una query generata dal metodo `get_objects`

Come è possibile osservare, la query è composta da una prima parte nella quale viene esplicitato l'intero schema dei dati e poi, di seguito, da una condizione `WHERE EXISTS` utilizzata per la verifica dell'identificativo del *nodo_type* in cui risiede l'informazione voluta. Dopodichè vengono espressi i filtri della ricerca attraverso il termine `contains`

La query viene costruita passo dopo passo concatenando le varie sezioni che compongono la stringa finale e, al termine di tale fase, viene eseguita ed i risultati vengono restituiti all'opportuna *action*.

Il secondo metodo definito è `relation_mapping_function` (Listato 5.19), il quale permette di creare lo schema di *relazioni* e *nodo_type* collegati a partire da un nodo inziale (nel caso di studio il tipo di nodo iniziale è "Medici"). Il parametro iniziale, denominato `init_object`, è l'unico necessario al metodo per ricavare lo schema dei dati.

Il risultato finale di questa fase è il seguente:

(Medici)-[WORK_IN]-(Ambulatori)-[Orario]-(Orari)

Questo processo si rende necessario poichè la classe `Neo4jKnowledgeBase` è stata progettata affinchè la si possa riutilizzare in futuri progetti di nuovi chatbot con knowledge base basata su Neo4j. Inoltre, un ulteriore vantaggio a favore di questa soluzione, è dato dalla sua capacità di adattarsi a nuovi schemi rappresentativi dei dati.

Il metodo per la creazione dello schema dei dati viene eseguito durante l'inizializzazione del servizio e si divide in due fasi: la prima, a partire dal tipo `initial_object`, esegue una query al fine di derivare le relazioni ed i `nodo_type` collegati ad esso; mentre la seconda fase, per ogni risultato ottenuto, esegue iterativamente una query fin quando non vengono più rilevate relazioni nel DBMS da mappare all'interno dello schema dei dati fin lì ottenuto.

```
1   """funzione che crea la mappa delle relazioni a partire da quella definita da principio"""
2       def relation_mapping_function(self, init_object : Text) -> Text:
3           s_entity = []
4           s_relation = []
```

```
5           s_result = []
6           q_relation = ""
7           logger.debug("initial object: " + init_object)
8           query = "match (n:"+ init_object +")-[relation]-(medium) where relation is NOT null RETURN DISTINCT type(
                relation) As relation, labels(medium) As medium"
9           result = self.query_db(query)
10          try: #try se ci sono relazioni, se non ci sono fallisce e except
11              for s in result:
12                  if s['relation'] != None and s['relation'] not in s_relation:
13                      s_result.append([str(s['relation']),str(s['medium'][0])])
14                      s_entity.append(s['medium'][0])
15                      s_relation.append(s['relation'])
16                      print(str(s['relation']))
17              s_result = s_result.pop(0) #elimino una lista con
18
19              logger.debug('fase 2')
20
21              popentity = s_entity.pop(0)
22              #relations = ""
23              while popentity != "":
24                  #if relations == "":
25                  #   relations = "relation"
26                  logger.debug("popentity: " + str(popentity))
27                  subquery = "MATCH (n:"+ self.entity_mapping_function(popentity) +")-[relation]-(last) where relation
                        is NOT null RETURN DISTINCT type(relation) As relation , labels(last) As last"
28                  subresult = self.query_db(subquery)
29                  logger.debug('subquery')
30
31                  subsub = [] #lista degli elementi che dovranno popolare sResult
32                  for sub in subresult:
33                      if sub['relation'] != None and sub['last'][0] != None:
34                          if sub['relation'] not in s_relation:
35                              logger.debug('inizio del for: %s', str(sub['relation']) + ' -> ' + str(sub['last'][0]))
36                              subsub.append(s_result) #mi creo una riga in subsub
37                              logger.debug('subsub: %s', subsub)
38                              s_relation.append(sub['relation'])
39                              s_entity.append(sub['last'][0])
40                              for rr in subsub: #per ogni vecchia relazione, viene ampliata la sua relazione se ne è
                                        provvista
41                                  logger.debug('rr: %s', rr)
42                                  if rr[1] == popentity:
43                                      logger.debug('rr[1]: ' + str(rr[1]))
44                                      rr.append(sub['relation'])
45                                      rr.append(sub['last'][0])
46                                      logger.debug('rel: ' + str(sub['relation']) + ', ent: ' + str(sub['last'][0]))
47                              logger.debug('subsub modificato: %s', subsub)
48                          else:
49                              logger.debug("la relazione e' gia stata inserita")
50                      else:
51                          logger.debug('La relazione o la tabella non sono state trovate')
52
53                  if s_entity != []: #se ci sono ancora tabelle nella lista delle tabelle da esplorare
54                      logger.debug('s_entity: %s', s_entity)
55                      popentity = s_entity.pop(0) #estrarre nome entity e poi di seguito ricomincio ciclo while
56                      s_result = subsub #reinserisco il risultato aggiornato in s_result
57                  else:
58                      logger.debug('sono finite le tabelle in relazione')
59                      popentity = ""
60              logger.debug('risultato query relazioni: %s', s_result)
61
62              #ciclo di costruzione della query
63              #q_relation += "("+ self.initial_object +")"
64              for s_res in s_result:
65                  if s_res != None:
66                      rel = []
67                      tab = []
68                      for sr in range(len(s_res)):
69                          if sr%2 == 0:
70                              rel.append(s_res[sr])
71                              logger.debug("1: "+str(sr) + ", " + s_res[sr])
72                              q_relation += "-["+s_res[sr]+"]"
73                          elif sr%2 == 1:
74                              tab.append(s_res[sr])
75                              logger.debug("2: "+str(sr) + ", " + s_res[sr])
76                              q_relation += "-("+s_res[sr]+")"
77                      logger.debug("q_relation: " + q_relation)
78              self.relation_name = rel
79              self.table_name = tab
80          except:
81              logger.debug("nessuna relazione trovata")
82          return q_relation
```

Listato 5.19. Codice del metodo `relation_mapping_function`

Infine, il terzo metodo illustrato è `get_object_type_by_attribute` (Listato 5.20). Esso permette di restituire il nome della tabella dell'attributo passato come parametro; tale metodo viene utilizzato nel momento in cui si vuole forzare la definizione dell'`object_type` all'interno di un'*action* in cui non è stata rilevata

durante la fase di *Entity Extraction*.

```
1       """funzione che restituisce il nome della tabella dell'attributo"""
2       def get_object_type_by_attribute(self, attribute) -> Text:
3           oggetto = ""
4           if attribute:
5               query = "MATCH (n) WHERE EXISTS(n."+ attribute +") RETURN DISTINCT LABELS(n) AS Entity"
6
7               result = self.query_db(query)
8               if result:
9                   for res in result:
10                      if res['Entity'][0] != None and (res['Entity'][0] in self.relation_object or res['Entity'][0] in
                                self.initial_object):
11                          oggetto = res['Entity'][0]
12          return oggetto
```

Listato 5.20. Codice del metodo `get_object_type_by_attribute`

5.2.8 utils_bot.py

Al fine di generalizzare il più possibile la soluzione chatbot ideata, sono state create delle funzioni a supporto delle action definite nel file `actions.py`. All'interno di `utils_bot.py` è stata definita la classe `Rasa_utils`, la quale utilizza un file `.json` denominato `file_utils.json` e contenente le informazioni essenziali per la personalizzazione del chatbot e per la corretta elaborazione delle richieste dell'utente.

Nel `JSON` sono contenuti dieci campi principali e sono:

1. `object_types`: esso contiene una lista dei nominativi degli `object_type` del bot.
2. `entities`: esso contiene gli attributi significativi e identificativi di ogni `object_type` e il nome della corrispondente tabella nel DBMS.
3. `initial_entity`: esso rappresenta il campo con l'`object_type` iniziale utilizzato per creare lo schema dei dati sul DBMS.
4. `reference_singolar`: esso contiene i riferimenti al singolare in linguaggio naturale degli `object_type` che saranno visualizzati all'interno della risposta all'utente.
5. `reference_plural`: esso contiene i riferimenti al plurale in linguaggio naturale degli `object_type` che saranno visualizzati all'interno della risposta all'utente.
6. `reference`: esso rappresenta il campo costituito dalle informazioni per la visualizzazione dei dottori ed effettua una distinzione tra medici di medicina generale e pediatri.
7. `result_number`: esso consente di mappare i numeri da "1" a "12" con i rispettivi aggettivi ordinali.
8. `convert_date`: esso consente di ottenere i giorni della settimana attraverso una chiave numerica.
9. `giorni`: esso permette di mappare due diverse rappresentazioni dei giorni della settimana; è necessario per via degli errati valori presenti nel database, nei quali non sono presenti le lettere accentate.
10. `plural_attribute`: esso effettua un mapping tra gli attributi e il corrispettivo termine al plurale.

Il Listato 5.21 riporta il contenuto del file `utils_bot.py`.

```
{
   "object_types" : [ "medico", "ambulatorio", "orario" ]
   ,
   "entities" : {
      "medico": {
         "table": "Medici",
         "discriminante": "id_operatore",
         "param1": "nome",
         "param2": "cognome",
         "param3": "comune",
         "sesso": "sesso"},
      "ambulatorio": {
         "table": "Ambulatori",
         "discriminante": "id_raplav_ambulatorio",
         "param1": "indirizzo",
         "param2": "civico",
         "param3": "comune"},
      "orario": {
         "table": "Orari",
         "discriminante": "giorno",
         "param1": "orariodiapertura",
         "param2": "orariodichiusura",
         "param3": "giorno",
         "param4": "indirizzo",
         "param5": "comune",
         "param6": "nome",
         "param7": "cognome",
         "sesso": "sesso",
         "discriminante2": "id_operatore#id_raplav_ambulatorio#giorno"
      }
   },
   "initial_entity": "medico",
   "reference_singolar": {
      "medico": "il dottore",
      "ambulatorio": "lo studio",
      "orario": "l'orario",
      "pediatra": "il pediatra"
   },
   "reference_plural": {
      "medico": "dottori",
      "ambulatorio": "luoghi",
      "orario": "orari",
      "pediatra": "pediatri"
   },
   "reference": {
      "MMG": {
         "singolare": {
            "M": {
               "entita": "dottore",
               "articolo": "il",
               "preposizione": "del"
            },
            "F": {
               "entita": "dottoressa",
               "articolo": "la",
               "preposizione": "della"
            }

         },
         "plurale": {
            "M": {
               "entita": "dottori",
               "articolo": "i",
               "preposizione": "dei"
            },
            "F": {
               "entita": "dottoresse",
               "articolo": "le",
               "preposizione": "delle"
            }
         }
      },
      "PLS": {
         "singolare": {
            "M": {
               "entita": "pediatra",
               "articolo": "il",
               "preposizione": "del"
            },
            "F": {
               "entita": "pediatra",
               "articolo": "la",
               "preposizione": "della"
            }
         },
         "plurale": {
            "M": {
               "entita": "pediatri",
               "articolo": "i",
               "preposizione": "dei"
            },
            "F": {
               "entita": "pediatri",
```

```
                    "articolo": "i",
                    "preposizione": "dei"
                }
            }
        }
    },
    "result_number": {
        "1": "il primo",
        "2": "il secondo",
        "3": "il terzo",
        "4": "il quarto",
        "5": "il quinto",
        "6": "il sesto",
        "7": "il settimo",
        "8": "l'ottavo",
        "9": "il nono",
        "10": "il decimo",
        "11": "l'undicesimo",
        "12": "il dodicesimo"
    },
    "convert_date": {
        "1": "oggi",
        "2": "domani",
        "11" : "Lunedì",
        "22" : "Martedì",
        "33" : "Mercoledì",
        "44" : "Giovedì",
        "55" : "Venerdì",
        "66" : "Sabato",
        "77" : "Domenica"
    },
    "giorni" : {
        "lunedì" : "Lunedì",
        "martedì" : "Martedì",
        "mercoledì" : "Mercoledì",
        "giovedì" : "Giovedì",
        "venerdì" : "Venerdì",
        "sabato" : "Sabato",
        "domenica" : "Domenica"
    },
    "plural_attribute" : {
        "telefono" : "telefoni",
        "indirizzo" : "indirizzi",
        "comune" : "comuni",
        "giorno" : "giorni",
        "orariodichiusura" : "orari di chiusura",
        "orariodiapertura" : "orario di apertura"
    }
}
```

Listato 5.21. Il contenuto del file `file_utils.json`

Dopo aver descritto `file_utils.json`, verrà ora illustrata la classe `Rasa_utils` nella quale sono definiti i metodi riportati di seguito:

- `get_object_types`: esso restituisce la lista degli `object_type` gestiti dal bot.
- `get_entities_parameters`: esso restituisce il campo *entities*, nel quale sono contenuti i parametri e i campi di ogni `object_type`.
- `get_entities`: esso restituisce il campo *entities* relativo di uno specifico `object_type`.
- `get_table`: esso converte il nome della tabella del DBMS con il nominativo del relativo `object_type`, ad esempio, "Medici : medico".
- `get_list_object`: esso restituisce la lista degli `object_type` ad esclusione di quello passato come parametro.
- `get_param_object`: esso restituisce i campi definiti all'interno della chiave `entities` di un *object_type* con i nominativi che iniziano con i caratteri "param" e concatenati ad un numero (ad esempio, "param1", "param2", etc).
- `get_knowledge_base`: esso restituisce l'istanza dell'oggetto `Neo4jKnowledgeBase` nel quale viene definita la connessione con Neo4J.
- `get_attributes_of_object`: esso restituisce gli attributi di tutti gli `object_type` all'interno del database.

- `get_references`: esso restituisce la stringa plurale o singolare in rappresentanza di un `object_type` specificato. Esso verrà di seguito inserito nel messaggio di risposta all'utente.
- `get_distinct_list`: esso viene usato per restituire una lista distinta, ovvero una lista che possiede un'occorrenza per ogni suo elemento.
- `get_result_number`: esso consente di mappare i numeri con gli equivalenti aggettivi ordinali, ad esempio il termine "1" con la stringa "il primo", "2" con il termine "il secondo", etc. Essi saranno in seguito utilizzati per rappresentare un'ipotetica lista dei risultati del chatbot.
- `get_utter_attribute_rappresentation`: esso restituisce la rappresentazione dell'`object_type` personalizzata diversamente rispetto alla versione definita di default dal metodo `reset_entities_parameter`. Al suo interno viene utilizzato il metodo `get_doctor_type` per riuscire a rappresentare il risultato a seconda che sia un medico oppure un pediatra, o che il termine sia singolare o plurale.
- `representsInt`: esso verifica se una stringa rappresenta un intero.
- `find`: esso trova un attributo all'interno di una lista di dizionari, senza tenere conto della profondità del dizionario o se esso contiene o meno una lista.
- `get_sesso_type`: il metodo restituisce il campo del database contenente l'informazione riguardante il genere dei dottori.
- `get_doctor_type`: esso restituisce una rappresentazione testuale del titolo lavorativo del dottore effettuando una distinzione tra medico o pediatra, tra dottore/pediatra o dottoressa/pediatra, e infine inserendo l'articolo più indicato a seconda del caso.
- `get_list_date`: esso restituisce la lista dei risultati di tipo "orario" con la sostituzione dei valori del campo "giorno", espressi attraverso i giorni della settimana, con i termini "oggi" e "domani" nei rispettivi risultati.
- `get_dict_giorni`: esso restituisce un dizionario contenente la mappatura tra i valori dei giorni espressi con la prima lettera minuscola e senza accenti e la loro rappresentazione contenente la prima lettera maiuscola e l'accento dove è necessario.
- `sort_date`: esso restituisce la lista dei risultati con *object_type* "orario" a partire dalla giornata corrente.
- `reset_entities_parameter`: questo metoto è stato già descritto nella sezione 5.2.6, in relazione all'*action* `action_query_list`. Esso permette di ridefinire i pattern di rappresentazione dell'*object_type* e il campo univoco con cui vengono identificati i singoli oggetti del tipo *object_type* rilevato a seguito della richiesta dell'utente.

```
1   UTILS_FILE = 'source/file_utils.json'
2   logger = logging.getLogger("rasa_sdk."+__name__)
3   class Rasa_utils():
4       file_utils_dict = []
5       initial_object = ""
6       dict_attributes = {}
7
8       def __init__(self):
9           self.file_utils_dict = self._load_file()
10          self.initial_object = self.file_utils_dict["initial_entity"]
11          self.knowledge_base = Neo4jKnowledgeBase("bolt://int-sdnet-convplat1.sdp.csi.it:7687", "neo4j", "test", "
                    Medici")
12          self.dict_attributes = self.set_attributes_of_object()
13
14
```

```
15      def _load_file(self) -> List:
16          """Try to load file_utils from json.
17          The file must be in the same dir of this file"""
18          dir_path = os.path.dirname(os.path.realpath(__file__))
19          file_path = os.path.join(dir_path, UTILS_FILE)
20          if os.path.isfile(file_path):
21              with open(file_path, 'r', encoding = 'utf-8') as fp:
22                  file_utils_dict = json.load(fp)
23          else:
24              file_utils_dict = None
25              warnings.warn(
26                  f"Failed to load file_utils from '{file_path}' \n The '{UTILS_FILE}' \
27                  must be in the same directory of utils.py."
28              )
29          return file_utils_dict
30
31      def reset_entities_parameter(self, object_type: Text) -> None:
32          # overwrite the representation function of the objects
33          # by default the representation function is just the name of the object medico
34
35          logger.debug("entità definita dopo reset: " + object_type)
36          entita = self.get_entities(object_type)
37
38          if entita["table"] == "Medici":
39              self.knowledge_base.set_representation_function_of_object(
40                  object_type, lambda obj: obj[entita["param1"]] + " " + obj[entita["param2"]]
41              )
42          elif entita["table"] == "Ambulatori":
43              self.knowledge_base.set_representation_function_of_object(
44                  object_type, lambda obj: obj[entita["param1"]] + " " + obj[entita["param2"]] + " a " + obj[entita["
                        param3"]]
45              )
46          elif entita["table"] == "Orari":
47              #doctor_type = self.file_util.get_doctor_type(object_type, obj, False, True)
48              self.knowledge_base.set_representation_function_of_object(
49                  object_type, lambda obj: self.get_doctor_type(object_type, obj, "singolare", "articolo") + " " + obj
                        [entita["param6"]] + " " + obj[entita["param7"]]+" riceve "+ obj[entita["param3"]] +" dalle
                        " + obj[entita["param1"]] + " alle " + obj[entita["param2"]] + " a " + obj[entita["param5"]]
50              )
51          else:
52              logger.error("l'entity non èin lista")
53
54          self.knowledge_base.set_entity_mapping(
55              object_type, lambda entity: entita["table"] if (entity == object_type) else entity
56          )
57
58          self.knowledge_base.set_key_attribute_of_object(object_type, entita["discriminante"])
59          logger.debug("key attribute settata: " + entita["discriminante"])
60
61          # definition of function to field and value for query
62          self.knowledge_base.set_attribute_operator(
63              lambda entity,attribute: 'contains' if (entity == object_type and attribute in (entita["param1"],
                        entita["param2"], entita["param3"])) else '='
64          )
65
66          #self.knowledge_base.set_attribute_value(
67          #    lambda entity,attribute, value: value if (entity == entita["table"] and attribute in (entita["param1"
                    ], entita["param2"], entita["param3"])) else "TOUPPER('"+value+"')"
68          #)
69
70      #restituisce la lista degli object_type che il bot gestisce
71      def get_object_types(self) -> List[Text]:
72          _list = self.file_utils_dict["object_types"]
73          logger.debug("_list: %s", _list)
74          return _list
75
76      #restituisce il campo 'entities", il quale al suo interno tiene traccia dei parametri e i campi di ogni
                object_type
77      def get_entities_parameters(self):
78          return self.file_utils_dict["entities"]
79
80       #restituisce il campo 'entities' riferito ad uno specifico object_type
81      def get_entities(self, ent: Text):
82          _entities = self.get_entities_parameters()
83          return _entities[ent]
84
85      #restituisce l'object_type iniziale da cui costruire lo schema di relazioni dei dati sul DB
86      def get_initial_object(self) -> Text:
87          return self.initial_object
88
89      #funzione che converte il nome della tabella con il nome dell'object_type. ES. Medici -> medico
90      def get_table(self, ob_type) -> Text:
91          for key, value in self.file_utils_dict["entities"].items():
92              #logger.debug(key)
93              #logger.debug(value)
94              for k, v in value.items():
95                  if v == ob_type:
96                      #logger.debug(key)
97                      object_type = key
98          return object_type
99
100     #restituisce la lista degli object_type a meno di quello specificato
101     def get_list_object(self, ob_type) -> List[Text]:
102         list_ob = []
```

```
103             for u in self.get_object_types():
104                 if u != ob_type:
105                     list_ob.append(u)
106             return list_ob
107 
108         #mi prende i parametri riferiti ad una entità definita nel file_utils.json in param
109         def get_param_object(self, object_type):
110             list_param = []
111             dict_param = self.get_entities(object_type)
112             logger.debug(dict_param)
113             for key,value in dict_param.items():
114                 if "param" in key:
115                     list_param.append(value)
116             logger.debug("parametri da suggerire: %s", list_param)
117             return list_param
118 
119         #restituisce il riferimento al connettore della knowledgeBase Neo4J
120         def get_knowledge_base(self):
121             return self.knowledge_base
122 
123         #restituisce tutti gli attributi di tutti gli object_type all'interno del DB
124         def get_attributes_of_object(self, object_type: Text) -> List[Text]:
125             return self.dict_attributes[self.initial_object]
126 
127         #restituisce un dizionario degli attributi nel DB
128         def set_attributes_of_object(self):
129             dict_att = {}
130             list_of_ob = self.file_utils_dict["initial_entity"]
131             attributes_of_object = self.knowledge_base.get_attributes_of_object(list_of_ob)
132             #for item_object in list_of_ob:
133             dict_att.update( {list_of_ob : attributes_of_object} )
134             return dict_att
135 
136         #restituisce il plurale o singolare di un object_type specificato al fine di inserirlo nel messaggio di
                risposta;
137         def get_references(self, object_type: Text, occurance) -> Text:
138             if occurance == True: #riferimento plurale
139                 _ob = self.file_utils_dict["reference_plural"][object_type]
140             else: #riferimento singolare
141                 _ob = self.file_utils_dict["reference_singolar"][object_type]
142             return _ob
143 
144         #restituisce una lista distinta (un'occorrenza per ogni elemento)
145         def get_distinct_list(self, _list: List[Text]) -> List[Text]:
146             output = []
147             for x in _list:
148                 if x not in output:
149                     output.append(x)
150             return output
151 
152         #mappa i numeri con gli equivalenti 1=="il primo", 2=="il secondo"....
153         def get_result_number(self, num: Text) -> Text:
154             return self.file_utils_dict["result_number"][num]
155 
156         #restituisco la rappresentazione dell'objcet_type personalizzata rispetto a quella settata di default dal
                metodo reset_entities_parameter().
157         #Al suo interno viene utilizzato il metodo get_doctor_type() per riuscire a rappresentare il risultato, a
                seconda che sia un medico oppure un pediatra, o che il termine debba essere singolare o plurale.
158         def get_utter_attribute_rappresentation(self, object_type: Text, object_of_interest: List) -> Text:
159             dict_param = self.get_entities(object_type)
160             if object_type == "orario":
161                 object_representation = self.get_doctor_type(object_type, object_of_interest, "singolare" , "
                        preposizione") + " " + object_of_interest[dict_param["param6"]] + " " + object_of_interest[
                        dict_param["param7"]]
162             elif object_type == "medico":
163                 object_representation = self.get_doctor_type(object_type, object_of_interest, "singolare" , "
                        preposizione") + " " + object_of_interest[dict_param["param1"]] + " " + object_of_interest[
                        dict_param["param2"]]
164             else:
165                 object_representation = ""
166             return object_representation
167 
168         #verifica se èun intero
169         def representsInt(self, s):
170             try:
171                 int(s)
172                 return True
173             except ValueError:
174                 return False
175 
176         #trova un attributo all'interno di una lista di dizionari,
177         #senza tenere conto della profondità del dizionario o se contenga una lista
178         def find(self, key, dictionary):
179             for k, v in dictionary.items():
180                 if k == key:
181                     yield v
182                 elif isinstance(v, dict):
183                     for result in self.find(key, v):
184                         yield result
185                 elif isinstance(v, list):
186                     for d in v:
187                         for result in self.find(key, d):
188                             yield result
189 
```

```
    #caratterizza la risposta in base al sesso del dottore
    def get_sesso_type(self, object_type, object_of_interest:List) -> Text:
        return object_of_interest["sesso"]

    #restituisce una rappresentazione testuale del dottore distinguendo tra medico o pediatra,
    #tra dottore/pediatra o dottoressa/pediatra e infine inserendo l'articolo più indicato a seconda del caso
    def get_doctor_type(self, object_type, object_of_interest:List, occorence, article_flag) -> Text:
        doctor_type = ""
        article = ""
        dict_param = self.file_utils_dict

        type_doc = object_of_interest["desc_categoria"]
        sesso = object_of_interest["sesso"]

        doctor_type = dict_param["reference"][type_doc][occorence][sesso]["entita"]

        article = dict_param["reference"][type_doc][occorence][sesso][article_flag]

        doctor_type = article + " " + doctor_type

        return doctor_type


    def get_list_date(self, objects) -> List:
        today = date.today()
        this_day_number = today.weekday() + 1
        for obj in objects:
            if int(obj["id_giorno"]) == this_day_number and obj["giorno"] != "oggi":
                obj["giorno"] = "oggi"
            elif int(obj["id_giorno"]) == this_day_number + 1 and obj["giorno"] != "domani":
                obj["giorno"] = "domani"
            elif obj["giorno"] == "oggi":
                obj["giorno"] = self.file_utils_dict["convert_date"][str(this_day_number)+str(this_day_number)]
                #logger.debug(str(this_day_number)+str(this_day_number) + " è" + self.file_utils_dict["convert_date"
                        ][str(this_day_number)+str(this_day_number)])
            elif obj["giorno"] == "domani":
                obj["giorno"] = self.file_utils_dict["convert_date"][str(this_day_number+1)+str(this_day_number+1)]
                #logger.debug(str(this_day_number)+str(this_day_number) + " è" + self.file_utils_dict["convert_date"
                        ][str(this_day_number+1)+str(this_day_number+1)])
        return objects

    #restituisce un dizionario contenente la mappatura tra i giorni scritti con prima lettera minuscola
    #e senza accenti in una rappresentazione con la prima lettera maiuscola e l'accento dove ènecessario
    def get_dict_giorni(self) -> Dict:
        return self.file_utils_dict["giorni"]

    #restituisce la lista dei risultati a partire dalla giornata odierna
    def sort_date(self, objects):
        today = date.today()
        this_day_number = today.weekday() + 1
        objects.sort(key = lambda i: int(i['id_giorno'])) #object sort by id_giorno
        if len(objects) > 1:
            obje = [o for o in objects if int(o["id_giorno"]) < this_day_number] #giorni precedenti
            cts = [o for o in objects if int(o["id_giorno"]) >= this_day_number] #giorni successivi
            objects = cts + obje
        return objects

    def google_maps(self, req):

        req.strip()
        uri = 'https://www.google.com/maps/place/'
        url = uri+req
        """ret = requests.get(url).text
        scrape = '['+ret.split('cacheResponse([[')[1].split(',[')[0].split(',',1)[1]
        location = ast.literal_eval(scrape)
        print("coordinate: " + location)"""
        return url

    def get_personal_doctor(self, latest_message): #tracker.latest_message.text
        doctor = ""
        return doctor

    def get_utter_massage_rappresentation(self, object_type, articolo, attribute, object_representation, list_value
            ):
        _output = ""

        if object_type == "orario" or object_type == "medico":
            _output ="{}{} {} è{}.\n".format(articolo, attribute, object_representation, list_value)
        elif object_type == "ambulatorio":
            _output ="{}{} di {} è{}.\n".format(articolo, attribute, object_representation, list_value)

        return _output

    def define_ob_for_output(self, object_type, object_, repr_function):
        _ob = ""
        if object_type == "medico":
            _ob = self.get_doctor_type(object_type, object_, "singolare", "articolo") + " " + repr_function(object_
                    )
        elif object_type == "ambulatorio":
            _ob = self.get_doctor_type(object_type, object_, "singolare", "articolo") + " in " + repr_function(
                    object_)
            _ob += ".\nSe mi hai contattato da uno smartphone o PC, puoi trovare la sua posizione a questo link "

        result_string = "Ho trovato " + _ob
```

```
279
280         if object_type == "orario": #stampo risultato preciso
281                 result_string = "{} \n".format(repr_function(object_))
282
283         return result_string
284
285     def get_button_response(self, object_type, object_):
286         buttons = []
287         if object_type == "medico":
288             button_text = ".\nVuoi salvare il dottore come medico di riferimento?"
289             buttons = [{
290                 "title" : "chooseYes",
291                 "text" : "Si",
292                 "payload" : '/personal_doctor{"feedback_value": "si"}'
293                 },
294                 {"title": "chooseNo",
295                 "text" : "No",
296                 "payload" : '/personal_doctor{"feedback_value": "no"}'
297                 }]
298         elif object_type == "ambulatorio":
299             buttons = [{
300                 "type":"web_url",
301                 "title":"link",
302                 "text": "Google Maps",
303                 "payload": self.google_maps(str(str(object_["indirizzo"]).replace(" ", "+") +"+"+ object_["civico"]
                        +"+"+ object_["cap"] +"+"+ object_["comune"]))
304             }]
305         return buttons
```

Listato 5.22. Codice della classe `Rasa_utils`

5.2.9 ga_connector.py

Il file `ga_connector.py` contiene la definizione della classe `GoogleConnector`, la quale viene utilizzata per presentare una connessione con la piattaforma di Google Assistant. La classe permette di trasferire le richieste dall'assistente virtuale al chatbot, e viceversa, attraverso lo scambio di informazioni in formato `JSON` e grazie al protocollo di comunicazione *webhook*[5], il quale viene gestito all'interno della stessa classe `GoogleConnector`.

La classe, illustrata all'interno del Listato 5.23, presenta due metodi; il primo restituisce il nominativo con cui il *webhook* è richiamato, mentre il secondo gestisce la ricezione della richiesta da parte dell'utente e l'elaborazione del `JSON` di risposta.

```
1   class GoogleConnector(InputChannel):
2       """A custom http input channel.
3       This implementation is the basis for a custom implementation of a chat
4       frontend. You can customize this to send messages to Rasa Core and
5       retrieve responses from the agent."""
6
7       @classmethod
8       def name(cls):
9           return "google_assistant"
10
11      def blueprint(self, on_new_message):
12
13          google_webhook = Blueprint('google_webhook', __name__)
14
15          @google_webhook.route("/", methods=['GET'])
16          async def health(request):
17              return response.json({"status": "ok"})
18
19          @google_webhook.route("/webhook", methods=['POST'])
20          async def receive(request):
21              payload = request.json
22              #sender_id = payload['user']['userId']
23              intent = payload['inputs'][0]['intent']
24              message = ""
25              button = ""
```

[5] I webhook sono delle callback HTTP definite dallo sviluppatore. Solitamente sono scatenati da un evento, a seguito del quale essi inviano delle richieste al chatbot, il quale risponderà con il risultato desiderato.

```
26              doctor = ""
27              doctor_str = ""
28              buttons = []
29              resp = ""
30              stored_doctor = ""
31              try:
32                  stored_doctor_string = payload['user']['userStorage']
33                  stored_doctor_string = stored_doctor_string.replace("\\",'')
34                  stored_doctor_string = json.loads(stored_doctor_string)
35                  stored_doctor = stored_doctor_string["data"]["doctor"]
36              except:
37                  logger.debug("doctor not find")
38
39              if intent == 'actions.intent.MAIN' or intent == 'actions.intent.SIGN_IN':
40                  message = os.environ.get(
41                      'WELCOME_MESSAGE', "Ciao sono Sara il bot dei medici della regione Piemonte. Dimmi che
                            informazioni vuoi ed io te le fornirò! Hai bisogno di conoscere l'orario di visita del
                            tuo dottore? o la lista dei dottori nel tuo comune?")
42              else:
43                  try:
44                      text = payload['inputs'][0]['rawInputs'][0]['query'] #messaggio che arriva da GA {"doctor" :
                            stored_doctor}
45                      out = CollectingOutputChannel()
46                      await on_new_message(UserMessage(text, out, metadata={"doctor" : stored_doctor}))
47                      logger.debug("out: %s", out.messages)
48                      responses = [m["text"] for m in out.messages] #messaggi da inviare su GA
49                      buttons = [m["buttons"] for m in out.messages if "buttons" in m] #bottoni da inviare su GA
50                      message = responses[0]
51                      button = buttons[0][0]
52
53                      if button["title"] == "doctor":
54                          doctor = button["text"]
55                          #doctor_str = { "data": { "doctor": doctor }}
56                          doctor_str = "{\"data\":{\"doctor\": "+doctor+"}"
57                          doctor_str += "}"
58                          #message += ". il dottore esiste " + doctor
59                      elif button["title"] == "delete_doctor":
60                          doctor_str = "{\"data\":{}}"
61                      else:
62                          logger.debug("doctor not assigned")
63                  except:
64                      logger.debug("text non identificato")
65
66              r = {
67                  "expectUserResponse": 'true',
68                  "expectedInputs": [
69                      {
70                          "possibleIntents": [
71                              {
72                                  "intent": "actions.intent.TEXT"
73                              }
74                          ],
75                          "inputPrompt": {
76                              "richInitialPrompt": {
77                                  "items": [
78                                      {
79                                          "simpleResponse": {
80                                              "textToSpeech": message,
81                                              "displayText": message
82                                          }
83                                      }
84                                  ]
85                              }
86                          }
87                      }
88                  ],
89              }
90
91              if intent == 'actions.intent.MAIN':
92                  r["expectedInputs"][0]["possibleIntents"].append({
93                                  "intent": "actions.intent.SIGN_IN",
94                                  "inputValueData": {
95                                      "@type": "type.googleapis.com/google.actions.v2.SignInValueSpec"
96                                  }
97                              })
98
99              if doctor_str != "":
100                 r.update({"userStorage" : doctor_str})
101
102             if button:
103                 if button["title"] == "link":
104                     r["expectedInputs"][0]["inputPrompt"]["richInitialPrompt"]["items"].append(
105                                                                             {
106                                                                                 "basicCard": {
107                                                                                     "title": "Indirizzo",
108                                                                                     "subtitle": "clicca il link per
                                                                                             essere reindirizzato a
                                                                                            google maps",
109                                                                                     "formattedText": button["
                                                                                            payload"],
110                                                                                     "buttons": [
111                                                                                         {
112                                                                                             "title": "Indirizzo",
113                                                                                             "openUrlAction": {
```

```
114                                                                                    "url": button["payload"]
115                                                                                    }
116                                                                                }
117                                                                            ]
118                                                                        }
119                                                                    })
120             try:
121                 if buttons[0][1]:
122                     r["expectedInputs"][0]["inputPrompt"]["richInitialPrompt"].update(
123                                                                    {"suggestions":
124                                                                    [
125                                                                        {
126                                                                            "title": "si"
127                                                                        },
128                                                                        {
129                                                                            "title": "no"
130                                                                        }
131                                                                    ]})
132             except:
133                 logger.debug("suggestion not implemented")
134
135             return response.json(r)
136         return google_webhook
```

Listato 5.23. Codice della classe `GoogleConnector`

Il template su cui è stata basata la classe è fornito direttamente dalla community di sviluppatori di Rasa e, come osservabile alla riga 39 all'interno del metodo `blueprint`, sono gestiti solo tre *intent* con i quali Google Assistant etichetta il messaggio dell'utente inviato al chatbot; essi sono `actions.intent.MAIN`, `actions.intent.SIGN_IN` e `actions.intent.TEXT`.

All'interno di `blueprint` sono definiti due metodi asincroni; il primo risponde ad una richiesta di tipo *GET* riguardante la raggiungibilità del chatbot (riga 16), mentre il secondo, denominato `receive`, risponde ad una richiesta *POST* ricevendo le richieste dell'utente incapsulate all'interno di un file `JSON`.

Il metodo `receive` gestisce gli *intent* ricevuti da Google Assistant. Il primo *intent*, che solitamente il bot riceve all'inizio di una conversazione, è `actions.intent.MAIN`. Esso è associato al messaggio di invocazione del bot all'interno della piattaforma di Google Assistant; ad esso il bot risponde con il messaggio di presentazione di Medici-bot (riga 41). Inoltre, unitamente ad esso, il chatbot presenta all'utente la richiesta di effettuare l'autentificazione *SIGN-IN* nel caso in cui non avessse effettuato l'*account linking*. Tale procedura è gestita interamente dalla piattaforma di Google Assistant.

Il messaggio definito per l'invocazione di Medici-bot è "Parla con Medici Piemonte", mentre il messaggio di risposta è presentato alla riga 40. Di seguito, dopo aver effettuato l'accesso all'interno del chatbot, ogni messaggio espresso dall'utente viene etichettato secondo l'intent `actions.intent.TEXT`, e la risposta viene gestita all'interno del codice, dalla riga 42 fino alla 135.

In tale sezione di codice viene formato il `JSON` di risposta il quale contiene il messaggio generato attraverso le *action* presentate in precedenza.

In particolare, `(UserMessage` (riga 46) è il metodo che permette di trasferire il messaggio dell'utente e altre informazioni ai vari moduli dell'architettura di Rasa per effettuare l'intent recognition, l'entity extraction e la generazione del relativo messaggio di risposta destinato all'utente; tale messaggio sarà restituito a questo connettore per la sua trasmissione alla piattaforma di Google. Contemporaneamente, il metodo `receive` attende in stato di *wait* la conclusione dell'elaborazione della risposta da parte di Rasa e, successivamente, costruisce il `JSON` popolandolo con le informazioni appena ricevute (righe 48-135).

5.2.10 alexa_connector.py

Il file `alexa_connector.py`, al pari di `ga_connector.py`, è utilizzato per definire la connessione con la piattaforma di assistenza virtuale Alexa attraverso un protocollo di comunicazione *webhook*.

All'interno del file è presente la classe `AlexaConnector`, la quale è illustrata nel Listato 5.24.

```
1
2   class AlexaConnector(InputChannel):
3       """A custom http input channel for Alexa.
4       You can find more information on custom connectors in the
5       Rasa docs: https://rasa.com/docs/rasa/user-guide/connectors/custom-connectors/
6       """
7
8       @classmethod
9       def name(cls):
10          return "alexa_assistant"
11
12      # Sanic blueprint for handling input. The on_new_message
13      # function pass the received message to Rasa Core
14      # after you have parsed it
15      def blueprint(self, on_new_message):
16
17          alexa_webhook = Blueprint("alexa_webhook", __name__)
18
19          # required route: use to check if connector is live
20          @alexa_webhook.route("/", methods=["GET"])
21          async def health(request):
22              return response.json({"status": "ok"})
23
24          # required route: defines
25          @alexa_webhook.route("/webhook", methods=["POST"])
26          async def receive(request):
27              # get the json request sent by Alexa
28              payload = request.json
29              # check to see if the user is trying
30              # to launch the skill
31              intenttype = payload["request"]["type"]
32
33              # if the user is starting the skill, let them
34              # know it worked & what to do next
35              if intenttype == "LaunchRequest":
36                  message = "Ciao sono Sara il bot dei medici della regione Piemonte. Dimmi che informazioni vuoi ed
                            io te le fornirò! Hai bisogno di conoscere l'orario di visita del tuo dottore? o la lista
                            dei dottori nel tuo comune?"
37                  session = "false"
38              else:
39                  # get the Alexa-detected intent
40                  intent = payload["request"]["intent"]["name"]
41
42                  # makes sure the user isn't trying to
43                  # end the skill
44                  if intent == "AMAZON.StopIntent":
45                      session = "true"
46                      message = "Talk to you later"
47                  else:
48                      # get the user-provided text from
49                      # the slot named "text"
50                      text = payload["request"]["intent"]["slots"]["text"]["value"]
51
52                      # initialize output channel
53                      out = CollectingOutputChannel()
54
55                      # send the user message to Rasa &
56                      # wait for the response
57                      await on_new_message(UserMessage(text, out))
58                      # extract the text from Rasa's response
59                      responses = [m["text"] for m in out.messages]
60                      message = responses[0]
61                      session = "false"
62
63                  # Send the response generated by Rasa back to Alexa to
64                  # pass on to the user.
65
66              r = {
67                  "version": "1.0",
68                  "sessionAttributes": {"status": "test"},
69                  "response": {
70                      "outputSpeech": {
71                          "type": "PlainText",
72                          "text": message,
73                          "playBehavior": "REPLACE_ENQUEUED",
74                      },
75                      "reprompt": {
```

```
76                      "outputSpeech": {
77                          "type": "PlainText",
78                          "text": message,
79                          "playBehavior": "REPLACE_ENQUEUED",
80                      }
81                  },
82                  "card": {
83                      "type": "LinkAccount"
84                  },
85                  "shouldEndSession": session,
86              },
87          }
88
89          return response.json(r)
90
91      return alexa_webhook
```

Listato 5.24. Codice della classe `AlexaConnector`

Il codice della classe `AlexaConnector` è definito con lo stesso obiettivo e seguendo la medesima logica dichiarata per la classe `GoogleConnector`. Tuttavia, la modalità con cui Alexa gestisce le connessioni con servizi esterni è molto diversa da quella utilizzata da Google Assistant, anzi, nella realtà dei fatti, Alexa non permetterebbe di utilizzare i modelli NLU per il riconoscimento del linguaggio naturale al di fuori di quello disponibile sulla stessa piattaforma, ma solo unicamente di collegare delle "skill" per la generazione della risposta all'utente.

Perciò, al fine di riuscire ad utilizzare il modello NLU definito all'interno di Medici-bot, è stata utilizzata una particolare soluzione fornita dalla community di Rasa. Tale soluzione prevede la possibilità di incapsulare ogni messaggio dell'utente all'interno di un unico e solo *slot* in riferimento ad un unico *intent* definito nel modello NLU di Alexa. Tale *intent* è denominato `ReturnUserInput`.

5.3 Problematiche affrontate durante lo sviluppo

5.3.1 Disambiguazione tra le mentions e duckling

Durante alcuni test con l'*object_type* "Orario", da parte del chatbot si sono osservate delle complicazioni nel distinguere un termine che poteva essere identificato come "mention" oppure come un'informazione temporale. Nello specifico, i termini in questione sono gli aggettivi ordinali con cui viene richiamato il risultato di una richiesta precedente per formulare una "mention" (ad esempio, "mi dici il telefono del primo?"). Tali aggettivi si possono confondere con i modi comuni per riferirsi alle date (ad esempio, il termine "primo" nella frase "mi dici gli ambulatori aperti il primo di gennaio?"). Riguardo a quest'ultimo caso, è il servizio *Duckling* ad occuparsi dell'estrazione dell'informazione. Tuttavia, oltre ad esso, si attivavano più processi di estrazione e il bot restituiva lo stesso aggettivo ordinale mappato sia come "mention" sia come informazione temporale.

Per risolvere tale problema, è stato definito un metodo all'interno dell classe `ActionPersona`, chiamato `get_date`. Tale metodo si occupa anche della disambiguazione tra le entità "mention" e l'entità "comune", nonchè della gestione delle informazioni temporali multiple, ad esempio:

Mi dici gli ambulatori aperti "domani" dalle "9" a Torino?

Nel Listato 5.25 è possibile osservare l'implementazione del metodo `get_date`.

```
1   def get_date(self, dispatcher: CollectingDispatcher, tracker: Tracker, attributes: List[Dict[Text,Any]],
          object_type: Text) -> List[Dict[Text, Any]]:
2           dict_giorni = self.file_util.get_dict_giorni()
3           logger.debug(dict_giorni)
4           flag_time_response = False
5           date_of = [] #serve per disambiguare le date con le mention. se ci sono le date, non considero le mention.
                Lista che contiene i valori estratti e la sua "grain"
6           dict_time = {"start1" : None, "end1" : None, "start2" : None, "end2" : None, "start3" : None, "end3" : None
                }
7           for e in tracker.latest_message['entities']:
8               #attributi.append({'name' : e["entity"], 'value' : e["value"]})
9               if e["entity"] == "time":
10                  grain = list(self.file_util.find('grain', e["additional_info"]))[0]
11
12                  value = e["value"]
13                  if isinstance(value, dict):
14                      if value["to"] != None:
15                          value = value["to"]
16                      elif value["from"] != None:
17                          value = value["from"]
18                  date_of.append({ "grain" : grain, "date" : value })
19                  logger.debug("latest message value 1 %s", e["value"])
20                  if not (e["additional_info"]["type"] == "interval" and grain == "day"):
21                      dict_time.update({"start1" : e["start"], "end1" : e["end"]})
22
23              #controllo se l'NLU ha scambiato il date per una mention
24              elif e["entity"] == "mention" and (int(e["end"])-int(e["start"])) > 2 : #controllo se èuna mention
25                  logger.debug("latest message start 2 %s", e["value"])
26                  dict_time.update({"start2" : e["start"], "end2" : e["end"]})
27              #controllo se èstato inserito erroneamente il comune tra i date
28              elif e["entity"] == "comune":
29                  logger.debug("latest message start 3 %s", e["value"])
30                  dict_time.update({"start3" : e["start"], "end3" : e["end"]})
31              elif e["entity"] == "object_type" and object_type == "orario":
32                  if str(int(e["end"])-int(e["start"])) == "6":
33                      logger.debug("numero di lettere object_type:" + str(int(e["end"])-int(e["start"])))
34                      flag_time_response = True
35
36          if dict_time["start1"] == dict_time["start2"] or dict_time["end1"] == dict_time["end2"]:
37              date_of = []
38          if dict_time["start1"] == dict_time["start3"] or dict_time["end1"] == dict_time["end3"]:
39              attributes = [a for a in attributes if not (a['name'] == "comune")]
40          #logger.debug("latest message 1 %s", tracker.latest_message["entities"])
41
42          humanOrario = ""
43          humanGiorno = ""
44          if date_of:
45              for att in attributes:
46                  if att["name"] == "giorno":
47                      attributes.remove(att)
48                      logger.debug("humangiorno" + att['value'])
49                  if att["name"] == "orariodiapertura":
50                      attributes.remove(att)
51                      logger.debug("humangiorno" + att['value'])
52                  if att["name"] == "orariodichiusura":
53                      attributes.remove(att)
54                      logger.debug("humangiorno" + att['value'])
55
56              for typee in date_of:
57                  datetime_obj = dateutil.parser.parse(typee["date"])
58                  if typee["grain"] == "day":
59                      humanGiorno = datetime_obj.strftime('%A')
60                  elif typee["grain"] == "hour":
61                      humanGiorno = datetime_obj.strftime('%A')
62                      humanOrario = datetime_obj.strftime('%H:%S')
63                  logger.debug("Giorno e Orario: " + humanGiorno + " " + humanOrario)
64                  #dispatcher.utter_message("Questi sono i giorni: " + dict_giorni[humanGiorno])
65
66
67                  if humanGiorno != "": #valido giorno
68                      attributes.append({'name' : "giorno", 'value' : dict_giorni[humanGiorno]})
69
70                  if humanOrario != "00:00" and humanOrario != "": #valido orario
71                      #if att["value"] in humanOrario:
72                      attributes.append({'name' : "orariodiapertura", 'value' : humanOrario})
73                      logger.debug("humanorario: %s",attributes)
74          logger.debug("latest message 3 %s", attributes)
75          logger.debug("flag_time_response" + str(flag_time_response))
76          if humanGiorno == "" and flag_time_response == True:
77              today = date.today()
78              this_day_number = today.weekday()
79              day = list(dict_giorni.values())
80              logger.debug("giorno: " + day[this_day_number])
81              attributes.append({'name' : "giorno", 'value' : day[this_day_number]})
82          return attributes
```

Listato 5.25. Codice del metodo `get_date`

Inoltre, a causa del servizio *Duckling*, il quale restituisce diversi risultati in funzione della comprensione che ha avuto della frase, il metodo si rende indispensabile per l'uniformità della rappresentazione dei dati temporali.

Di fatto, riprendendo come esempio la frase "mi dici gli ambulatori aperti domani dalle 9 a Torino?", il primo output restituito da *Duckling* può risultare un'unico *datetype* contenente entrambe le informazioni "domani" e "9". Perciò, la funzione `get_date` ha il compito di eseguire la divisione del *datetype* in due informazioni separate, le quali saranno considerate al pari di due parametri distinti durante la costruzione della query sul database.

Un secondo possibile caso consiste nell'estrazione effettuata da *Duckling* delle informazioni temporali in formato *datetype*, congiuntamente ad un ulteriore servizio di estrazione, il quale fornisce un risultato ridondante ed errato. In questo caso l'esito fornito da *Duckling* sarà l'unico preso in considerazione, mentre l'altro, nel caso contenesse delle informazioni duplicate, verrebbe eliminato.

Infine, tenendo sempre in considerazione l'esempio, come ultimo caso è possibile riscontrare un risultato con il seguente contenuto: un *datetype* con entrambe le informazioni temporali, il token "9" estratto come "mention" e il token "domani" erroneamente individuato come entità di tipo "comune".

Nel caso appena riportato, come nel precedente, il metodo `get_date` permette di disambiguare la mention ed eliminare tale informazione, per poi, di seguito, riuscire ad identificare il token "domani" erroneamente identificato di tipo "comune". Di conseguenza, quest'ultima informazione sarà eliminata.

5.3.2 Aggiunta del button con link a google maps

Al chatbot Medici-bot è stata aggiunta la possibilità di fornire un link per Google Maps per l'individuazione geografica degli ambulatori ricercati dall'utente. Il link verrà reso fruibile esclusivamente nel caso in cui il risultato di una richiesta di un ambulatorio fosse unico.

Questa funzionalità è stata aggiunta esclusivamente per la piattaforma Google Assistant, poichè Alexa non consente l'utilizzo di bottoni che permettano il reindirizzamento, per mezzo di URL, su siti o servizi esterni.

Dunque, per la sua implementazione si è fatto uso delle *basicCard*, le quali sono dei contenitori configurabili nel file `.json` utilizzato per passare informazioni dall'istanza di `GoogleConnector` a Google Assistant. Tali *basicCard* vengono mostrate congiuntamente alla risposta del chatbot e permettono di raggruppare e contenere, a loro volta, oggetti che possono essere, ad esempio, delle immagini, delle mappe, delle liste o dei bottoni (ovvero, gli elementi di tipo `button`).

Perciò, all'interno di questa *basicCard* è stato definito un `button` contenente un link generato dinamicamente dall'`action_query_list`; nel momento in cui tale link viene premuto, esso è in grado di reindirizzare l'utente alla visualizzazione della mappa sull'applicazione o sul sito di Google Maps.

Di seguito è riportata l'aggiunta del `button` nella definizione del `responce.json` nel file `ga_connector.py`.

```
if button != "":
                r["expectedInputs"][0]["inputPrompt"]["richInitialPrompt"]["items"].append(
                    {
                        "basicCard": {
                            "title": "Indirizzo",
                            "subtitle": "clicca il link per essere reindirizzato a google maps",
                            "formattedText": button,
                            "buttons": [
                            {
                                "title": "Indirizzo",
                                "openUrlAction": {
                                "url": button
                                }
                            }
                            ]
                        }
                    })
```

Listato 5.26. Campo `basicCard` definito nel `responce.json`

Dopodichè, nel file `actions.py` è stato modificato il metodo `utter_objects` per aggiungere il `button` alla risposta fornita per l'*intent* `action_query_list`. Ovviamente, l'`object_type` rilevato dovrà risultare di tipo"ambulatorio"; in seguito a ciò verrà generato un *button* con le caratteristiche mostrate nel seguente listato:

```
buttons = [{
            "type":"web_url",
            "title":"link",
            "text": "Google Maps",
            "payload": self.file_util.google_maps(str(str(objects[0]["indirizzo"]).replace(" ", "+") +"+"+ objects
                    [0]["civico"] +"+"+ objects[0]["cap"] +"+"+ objects[0]["comune"]))
        }]
```

Listato 5.27. Sezione di codice contenente l'aggiunta del pulsante nel `responce.json`

Il campo `payload` contiene il link a Google Maps definito attraverso il concatenamento dei vari parametri che formano un indirizzo valido (ovvero, "comune", "indirizzo", "civico", "cap").

Dopo la creazione del link, esso viene passato al `GoogleConnector` attraverso il metodo `dispatcher.utter_button_message`, il quale aggiunge il pulsante agli oggetti da stampare nella risposta.

5.3.3 Definizione dell'account linking in Google Assistant

All'interno del seguente progetto è stata inserita la possibilità, da parte dell'utente, di associare e collegare il proprio account Google al chatbot Medici-bot. Eseguendo tale azione, l'utente beneficia di una esperienza personalizzata e con la possibilità di ottenere informazioni personali da lui stesso precedentemente salvate all'interno del proprio account. Allo stato attuale, l'unica informazione che è possibile definire risulta il "medico di fiducia" e sarà trattata nella successiva sottosezione.

L'account linking messo a disposizione all'utente è di tipo "Google SIGN-IN" ovvero, al momento del suo primo accesso, all'utente viene mostrata una schermata Popup nella quale si richiede di effettuare l'autentificazione sulla piattaforma Google inserendo le relative credenziali di accesso. A seguito di tale azione, l'utente è reindirizzato di nuovo alla schermata del chatbot, ed il seguente messaggio viene fornito come risposta dell'autentificazione conclutasi con successo:

> "Perfetto. Il tuo account di Medici Piemonte ora è collegato a Google. Ciao sono Sara, il bot dei medici della regione Piemonte. Dimmi che informazioni

vuoi ed io te le fornirò! Per caso hai bisogno di conoscere l'orario di visita del tuo dottore? oppure preferisci l'elenco dei dottori nel tuo comune?"

A livello di codice, l'account linking richiede la definizione dell'*intent* (messo a disposizione da Google Assistant) `actions.intent.SIGN_IN` all'interno del `JSON` fornito in risposta a Google Assistant. Così facendo, tale *intent* verrà attivato a seguito della risposta che il chatbot fornirà all'utente. Il passaggio citato può essere osservato all'interno della classe `GoogleConnector` alla riga 131 del Listato 5.23.

5.3.4 Salvataggio del dottore preferito tra le informazioni personali dell'utente

Alla soluzione Medici-bot è stata aggiunta la funzione di salvataggio del dottore preferito tra le informazioni personali, e ciò incrementa la personalizzazione dell'interazione tra utente e bot. Innanzitutto, sono stati creati tre nuovi *intent*, ovvero:

1. `personal_doctor`: questo è l*intent* di risposta alla conferma o alla negazione dell'utente di salvare il dottore. L'utente risponde con "Si" o "No".
2. `delete_personal_doctor`: attraverso questo *intent*, l'utente può richiedere l'eliminazione del dottore dai suoi dati personali. L'utente può chiedere "elimina le informazioni sul mio dottore".
3. `information_personal_doctor`: *intent* per chiedere al bot quale medico attualmente è memorizzato. L'utente può chiedere "mi dici le informazioni sul mio medico".

La richiesta di salvataggio del dottore preferito può avvenire in seguito alla presentazione di un risultato unico da parte di una ricerca. Ad esempio, nel caso in cui un utente richieda un dottore nel suo comune, nel momento in cui questo fosse l'unico, il bot domanda all'utente se vuole salvare tra i suoi dati il risultato come medico preferito. È importante ricordare che è possibile arrivare ad una soluzione unica anche fornendo informazioni aggiuntive rispetto ad una richiesta precedente, filtrando il risultato su più campi di ricerca. Inoltre, è di vitale importanza ricordare che la possibilità di salvare informazioni personali è strettamente legato alla funzionalità di account-linking, ovvero essa può essere implementata solo nel caso in cui l'utente si fosse precedentemente autenticato sulla piattaforma.

Google Assistant, per avere memoria dei dati personali di ogni singolo utente, permette di associare ad ogni account dei campi aggiuntivi; l'importante è che questi siano definiti all'interno di un campo chiamato "UserStorage". Tale attributo viene conservato e passato all'interno dei file `JSON` di `request` e di `responce` con cui Google invia e riceve i dati dal chatbot.

Di seguito è riportato un esempio del campo "user" del `responce.json` che Google Assistant gestisce. È possibile notare come, in associazione ad un token identificativo dell'utente, è presente il campo `userStorage`, contenente l'*id* di riferimento del dottore settato precedentemente (più precisamente, il campo `userStorage` presenta un'ulteriore campo `data`, al cui interno è memorizzato il campo `doctor` valorizzato con l'*id* del dottore).

La rappresentazione delle informazioni relative ad un singolo utente all'interno del `responce.json` viene mostrato nel seguente listato:

```
1   "user": {
2         "accessToken": "6246d9ea-72a6-4ef4-93e4-d107b069cbbe",
3         "locale": "it-IT",
4         "lastSeen": "2020-04-28T10:25:20Z",
5         "userStorage": "{\"data\":{\"doctor\": 13651}}",
6         "userVerificationStatus": "VERIFIED"
7      },
```

Listato 5.28. Campo `user` definito nel `responce.json`

Per quanto riguarda il recupero dell'informazione da `responce.json` e il suo utilizzo nell'action opportuno (in questo caso sono `delete_personal_doctor` e `information_personal_doctor`), i dati di interesse sono stati inseriti tra i metadati che Rasa consente di avere all'interno del `Tracker`. Questa entità contiene i dati ricavati in fase NLU ed altre informazioni utili alla corretta generazione della risposta all'utente. Per inserire i metadati nel `Tracker` è necessario valorizzare un parametro del metodo `UserMessage`, il quale prevede tale funzionalità (riga 46 della classe `GoogleConnector` nel Listato 5.23).

Avendo definito il procedimento e le funzionalità messe in campo per ottenere il salvataggio del medico di fiducia dell'utente, sarà ora possibile illustrare le tre *action* accennate precedentemente e rimandate al seguente punto della trattazione dalla Sottosezione 5.2.6.

La prima che verrà presentata è l'*action* `action_personal_doctor`, definita all'interno della classe `ActionPersonalDoctor` la quale risponde all'*intent personal_doctor* (nel Listato 5.29).

Il codice definito all'interno della classe è basato sulle informazioni prelevate dagli *slot* `SLOT_LAST_OBJECT` e `feedback_value`; il primo contiene l'identificativo del medico che si vuole salvare, mentre il secondo contiene la scelta espressa dallo stesso utente.

Perciò, se la variabile `choose`, contenente l'informazione passata dallo *slot* `feedback_value`, risultasse valorizzata dalla stringa "Si", l'identificativo del medico verrebbe passato al connettore `GoogleConnectore` e salvato tra i dati dell'utente. Per concludere, all'utente viene inviato un messaggio a conferma dell'avvenuto salvataggio.

```
1   class ActionPersonalDoctor(ActionPersona):
2
3       def name(self):
4           return "action_personal_doctor"
5
6       def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text,
                Any]]:
7           logger.info("action_personal_doctor")
8
9           object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
10          if not object_type:
11              object_type = tracker.get_slot(SLOT_LAST_OBJECT_TYPE)
12
13          if object_type:
14              self.file_util.reset_entities_parameter(object_type)
15
16          personal_doctor = tracker.get_slot(SLOT_LAST_OBJECT) #recupero dottore dallo slot last_object
17          choose = tracker.get_slot("feedback_value")
18
19          #listo tutti i nomi degli attributi di un object_type
20          object_attributes = self.file_util.get_attributes_of_object(object_type)
21          attributes = get_attribute_slots(tracker, object_attributes) #nuovo attributo con cui filtrare i dati
```

```
22
23          slots = [
24              SlotSet(SLOT_OBJECT_TYPE, None),
25              SlotSet(SLOT_ATTRIBUTE, None),
26              SlotSet(SLOT_MENTION, None),
27              SlotSet(SLOT_LAST_OBJECT, None),
28              SlotSet(SLOT_LAST_OBJECT_TYPE, object_type),
29              SlotSet("feedback_value", None),
30              SlotSet(SLOT_LAST_ATTRIBUTE, None),
31              SlotSet(SLOT_LISTED_OBJECTS, None),
32          ]
33          if choose == "si":
34              buttons = [{
35                          "title" : "doctor",
36                          "text" : personal_doctor,
37                          "payload" : '/personal_doctor{"feedback_value": "si"}'
38                          }]
39              slots.append(SlotSet("personal_doctor", personal_doctor))
40              dispatcher.utter_button_message("Il dottor " + personal_doctor + " èstato salvato come preferito",
                        buttons)
41              #dispatcher.utter_message(personal_doctor)
42              #doctor_dict = {"text" : str(personal_doctor)}
43              #dispatcher.utter_elements(doctor_dict)
44
45          else:
46              dispatcher.utter_message("vabene, il dottore non èstato salvato")
47
48          return slots + reset_attribute_slots(tracker, object_attributes)
49
50          dispatcher.utter_template("utter_ask_rephrase", tracker)
51          return []
```

Listato 5.29. Codice della classe `ActionPersonalDoctor`

La seconda è l'*action* `action_information_personal_doctor`, definita all'interno della classe `ActionInformationDoctor`, la quale risponde all'*intent information_personal_doctor* (nel Listato 5.30).

In questa *action*, al fine di presentare all'utente il medico preferito salvato sui suoi dati personali, viene recuperato l'identificativo dell'oggetto di tipo "medico" dai metadati trasferiti dal connettore di Google Assistant alla *action* per mezzo dell'istanza del *tracker*. Di fatto, al suo interno è stato definito un metadato e che viene recupato attraverso la chiave "doctor" (riga 26).

Dopodichè viene effettuata una query sul database al fine di recuperare le informazioni essenziali per la visualizzazione del risultato all'utente, ovvero si vogliono ottenere i campi "nome" e "cognome" per definire il pattern di rappresentazione del risultato.

Infine, il risultato viene trasferito al connettore Google attraverso il metodo `utter_message` della classe `dispatcher`, il quale permette di stampare il risultato senza dover richiamare i metodi `utter_object` e `utter_attribute_value`.

```
1   class ActionInformationDoctor(ActionPersona):
2
3       def name(self):
4           return "action_information_personal_doctor"
5
6       def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text,
            Any]]:
7           logger.info("action_information_personal_doctor")
8
9           object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
10          if not object_type:
11              object_type = tracker.get_slot(SLOT_LAST_OBJECT_TYPE)
12          if not object_type:
13              object_type = self.knowledge_base.get_object_type_by_attribute("id_operatore")
14              object_type = self.file_util.get_table(object_type)
15          if object_type:
16              self.file_util.reset_entities_parameter(object_type)
17
18          personal_doctor = tracker.get_slot(SLOT_LAST_OBJECT) #recupero dottore dallo slot last_object
19
20          events = tracker.current_state()['events']
21          #logger.debug("events: %s", events)
22          user_events = []
```

```
        for e in events:
            if e['event'] == 'user':
                user_events.append(e)
        personal_doctor = user_events[-1]['metadata']['doctor']
        #logger.debug("user_events: %s", user_events)
        #logger.debug("personal_doctor: %s", personal_doctor)

        #listo tutti i nomi degli attributi di un object_type
        object_attributes = self.file_util.get_attributes_of_object(object_type)
        attributes = [{"name" : "id_operatore", "value" : str(personal_doctor)}]
        repr_function = self.knowledge_base.get_representation_function_of_object(object_type)
        object_identifier = resolve_mention(tracker, self.knowledge_base.ordinal_mention_mapping)
        ob = self.knowledge_base.get_objects(object_type, attributes, object_identifier, None)
        list_object = []
        objects = []
        if ob:
            for obj in ob:
                if repr_function(obj) not in list_object:
                    list_object.append(repr_function(obj))
                    objects.append(obj)

        _ob = self.file_util.get_doctor_type(object_type, objects[0], "singolare", "articolo") + " "
        result_string = _ob + "preferita è" + repr_function(objects[0])

        if not objects:
            logger.debug("Objects not find")
            return reset_attribute_slots(tracker, object_attributes)

        slots = [
            SlotSet(SLOT_OBJECT_TYPE, None),
            SlotSet(SLOT_ATTRIBUTE, None),
            SlotSet(SLOT_MENTION, None),
            SlotSet(SLOT_LAST_OBJECT, None),
            SlotSet(SLOT_LAST_OBJECT_TYPE, object_type),
            SlotSet("feedback_value", None),
            SlotSet(SLOT_LAST_ATTRIBUTE, None),
            SlotSet(SLOT_LISTED_OBJECTS, None),
        ]

        dispatcher.utter_message(result_string)

        return slots + reset_attribute_slots(tracker, object_attributes)

        dispatcher.utter_template("utter_ask_rephrase", tracker)
        return []
```

Listato 5.30. Codice della classe `ActionInformationDoctor`

Infine, per concludere, verrà presentata l'ultima l'*action* del chatbot, ovvero l'`action_delete_personal_doctor`, la quale è definita all'interno della classe `ActionDeleteDoctor` e che risponde all'*intent delete_personal_doctor* (nel Listato 5.31).

In questa *action*, in realtà, non viene effettuata l'eliminazione di nessuna informazione personale riguardante il medico preferito dell'utente, poichè tale informazione è contenuta all'interno del `responce.json`. Dunque, tale classe genera essenzialmente un messaggio di risposta all'utente con la conferma dell'avvenuta cancellazione dell'informazione.

```
class ActionDeleteDoctor(ActionPersona):

    def name(self):
        return "action_delete_personal_doctor"

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

        object_type = tracker.get_slot(SLOT_OBJECT_TYPE)
        if not object_type:
            object_type = tracker.get_slot(SLOT_LAST_OBJECT_TYPE)

        if object_type:
            self.file_util.reset_entities_parameter(object_type)

        events = tracker.current_state()['events']
        logger.debug("events: %s", events)
        user_events = []
        for e in events:
            if e['event'] == 'user':
```

```
21                 user_events.append(e)
22         personal_doctor = user_events[-1]['metadata']
23         logger.debug("latest_message: %s", personal_doctor)
24
25         #listo tutti i nomi degli attributi di un object_type
26         object_attributes = self.file_util.get_attributes_of_object(object_type)
27
28         slots = [
29             SlotSet(SLOT_OBJECT_TYPE, None),
30             SlotSet(SLOT_ATTRIBUTE, None),
31             SlotSet(SLOT_MENTION, None),
32             SlotSet(SLOT_LAST_OBJECT, None),
33             SlotSet(SLOT_LAST_OBJECT_TYPE, object_type),
34             SlotSet("feedback_value", None),
35             SlotSet(SLOT_LAST_ATTRIBUTE, None),
36             SlotSet(SLOT_LISTED_OBJECTS, None),
37         ]
38
39         buttons = [{
40                       "title" : "delete_doctor",
41                       "text" : personal_doctor,
42                       "payload" : '/personal_doctor{"feedback_value": "si"}'
43                       }]
44         slots.append(SlotSet("personal_doctor", personal_doctor))
45         dispatcher.utter_button_message("Il tuo dottore preferito èstato eliminato dalle tue informazioni personali
                  ", buttons)
46
47         return slots + reset_attribute_slots(tracker, object_attributes)
48
49         dispatcher.utter_template("utter_ask_rephrase", tracker)
50         return []
```

Listato 5.31. Codice della classe `ActionDeleteDoctor`

Il comando di cancellazione dell'informazione viene passato all'interno della classe `GoogleConnector` attraverso la definizione di un "button" (ovvero la definizione di un bottone visibile in output, riga 44), nel quale è dichiarato il campo "title" contenente la stringa "delete_doctor".

Dunque, tale stringa verrà utilizzata all'interno della classe `GoogleConnector` (Listato 5.23 alla riga 60), per soddisfare una condizione che permetterà di eliminare il contenuto del campo "data" del `responce.json`.

6

Medici-bot: Fase di validazione

In questo capitolo si discuterà l'ultima fase del lavoro finora illustrato, ovvero la validazione delle funzionalità del chatbot. Verranno, dapprima, analizzate le tecniche fornite dal framework Rasa ed utilizzate per valutare le performance di Medici-bot. Successivamente, saranno presentati i risultati ottenuti riguardo il modello NLU e il modello Core del chatbot

6.1 Panoramica sulle procedure di validazione secondo il framework Rasa

Il framework Rasa, attraverso vari metodi "built-in" e metriche da poter consultare, consente di effettuare diverse valutazioni sulle performance del chatbot. Possono essere verificati, in modo distinto, diversi componenti, come, ad esempio, il *Natural Language Understanding model* (detto NLU model) o il *Core model*. È bene sottolineare come il primo modello, il quale consente di identificare l'*intent* ed estrarre le *entity*, sia composto da molteplici moduli, ciascuno dei quali è valutabile servendosi di precise metriche.

Dunque, il fulcro della fase di validazione del bot è composto dall'analisi dei due modelli che fungono da "motore" per l'intera logica del chatbot.

Saranno, ora, descritti i metodi di valutazione del chatbot, ognuno indispensabile per alcuni aspetti:

- *End-to-End Testing*: Rasa consente di valutare i dialoghi end-to-end, ovvero permette di eseguire conversazioni di test al fine di assicurarsi che sia il modello NLU e sia il modello Core facciano predizioni corrette. A tale scopo, si ha la necessità di definire alcune storie nel formato end-to-end, ovvero capaci di includere sia l'output del modello NLU (le *intent* e gli *slot*) e sia il testo originale dell'utente. Tali storie saranno contenute all'interno di un file denominato `tests/conversation_tests.md` e rappresentate dal seguente template:

```
1  ## End-to-End tests where a custom action appends events
2  * intent1: <<testo originale>>
3      - action
```

```
4       <!-- Gli eventi successivi hanno origine dall'action -->
5       - slot{"nome_slot": "<<valori aggiunti dall'action>>"}
6       - template1
7    * intent2: <<testo originale>>
8       - template2
```

Il test si avvia da riga di comando eseguendo lo script `rasa test`.

- *NLU Model Evaluation*: la seconda tipologia effettua esclusivamente delle valutazioni sulle componenti che costituiscono il modello NLU, ovvero la *pipeline*, l'*Intent Classificator* e l'*Entity Extractor*.
  Tale metodo consente la valutazione del modello NLU attraverso delle tecniche e metriche tipiche del machine learning; al fine di verificare la bontà del modello, possono essere utilizzate sia la *split validation* che la *cross validation*. Entrambe sono supportate dal framework Rasa, il quale consente di realizzare una valutazione veloce, automatica e con molteplici risultati da poter consultare.
  La prima delle tecniche che verrà illustrata è la *split validation*; essa effettua una suddivisione del dataset delle frasi di training (il quale è contenuto nel file `nlu.md`) in due sottoinsiemi di dati, dove il primo è necessario per l'addestramento del modello, mentre il secondo serve per realizzare il test di esso.
  Il comando Rasa è il seguente:

```
1       rasa data split nlu
```

Come risultato si avrà la divisione materiale del dataset in due insiemi distinti di dati.
A seguito di ciò, a partire dal primo dataset, è possibile effettuare il test del modello creato utilizzando il comando:

```
1       rasa test nlu -u train_test_split/test_data.md --model models/nlu-model.tar.gz
```

Nell'eventualità che uno sviluppatore non abbia la volontà di effettuare uno "split" dei dati e voglia evitare il problema dell'*overfitting*, il framework Rasa mette a disposizione la seconda tecnica accennata in precedenza, ovvero la *cross validation*. Tale tecnica può essere utilizzata attraverso il seguente comando da "shell":

```
1       rasa test nlu -u data/nlu.md --config config.yml --cross-validation
```

Nel comando, oltre al "flag" `--cross-validation`, sono dichiarati il dataset `nlu.md` e il `config.yml`, contenente le caratteristiche che il modello dovrà rispettare.
L'esecuzione di tale script restituisce diversi risultati per ciascuno degli aspetti che caratterizzano il modello; essi sono la *pipeline*, l'*Intent Classification*, e l'*Entity Extraction*.
Per quanto concerne l'*Intent Classification*, vengono prodotti diversi file di report:

- `intent_report.json`: esso contiene, per ogni *intent* del chatbot, i valori delle principali metriche di valutazione di un modello, ovvero la *precision*, il *recall*, l'*f1-score* e il *support*.
- `confmat.png`: al suo interno è riportata la figura della matrice di confusione, la quale contiene la rappresentazione dei risultati della classificazione degli *intent* del bot.
- `hist.png`: all'interno dell'immagine è rappresentato un istogramma della confidenza del modello analizzato.
- `intent_errors.json`: il file contiene le frasi che sono state classificate erratamente rispetto al loro corretto *intent*. Ciascuna delle frasi è caratterizzata dal valore della `confidence` e contiene i nominativi dei due *intent* (quello corretto e quello erroneamente rilevato).

I risultati ottenuti per la *Entity Extraction* comprendono unicamente la valutazione del modulo *CRFEntityExtractor*, poichè esso è l'unico estrattore di entità addestrato utilizzando i dati del file `nlu.md`. Di conseguenza, non saranno valutati alcuni servizi, come *Duckling* e *SpaCy* (entrambi utilizzano modelli preaddestrati). Detto ciò, Rasa fornirà come risultato le metriche *f1-score*, *recall* e *precision* per ciascun tipo di entità che *CRFEntityExtractor* è addestrato a classificare.
Inoltre, Rasa consente di effettuare un'analisi specifica sulle prestazioni ottenute dalla comparazioni di due *pipeline* definite, così da poter generare delle informazioni su quale soluzione è opportuno scegliere per il chatbot.
È possibile effettuare tale tipologia di analisi eseguendo, ad esempio, il seguente comando:

```
1       rasa test nlu --config pretrained_embeddings_spacy.yml supervised_embeddings.yml --nlu data/nlu.md --runs
            3 --percentages 0 25 50 70 90
```

Tale comando crea una suddivisione train/test dei dati posseduti, ed addestra più volte diversi modelli per ogni pipeline utilizzando lo 0, 25, 50, 70 e 90% dei dati degli **intent** inclusi nel dataset di train. A seguito di ciò, i modelli vengono valutati attraverso l'insieme di test e si registra il valore della metrica *f1-score* per ciascuna delle percentuali illustrate precedentemente.
Tale processo viene eseguito tre volte (ovvero con 3 dataset di test in totale) e, di conseguenza, viene delineato un grafico utilizzando le medie e le deviazioni standard dei valori della metrica *f1-score*.

- *Core Model Evaluation*: In alternativa alle altre tipologie di test, questa permette di valutare esclusivamente il modello Core del chatbot creato. Di fatto, è possibile analizzare il modello addestrato attraverso l'utilizzo di storie create appositamente per il test e, quindi, di verificarne il loro compimento.
Il comando utilizzato per eseguire la verifica è il seguente:

```
1       rasa test core --stories test_stories.md --out results
```

Tale test è in grado di analizzare il corretto svolgimento di una *story* o se essa fallisce. Una qualsiasi *story* è considerata fallita se almeno una delle *action*, in essa richiamate, è stata predetta in modo errato.

Le storie etichettate come fallite sono restituite all'interno di un file denominato `results/failed_stories.md`; in aggiunta ad esse viene creata una matrice di confusione per mostrare le occorrenze con cui è stata predetta correttamente l'*action* e con quale frequenza è stata, invece, prevista un'*action* errata.

6.2 Validazione di Medici-bot

6.2.1 Definizione della validazione effettuata

La validazione di Medici-bot è stata effettuata in due fasi, ovvero l'*NLU Model Evaluation* e la *Core Model Evaluation*.

Per quanto concerne la prima fase, è stato deciso di utilizzare la tecnica di *cross validation* per effettuare l'opportuna validazione del modello NLU.

La principale motivazione che ha favorito tale scelta è stata la considerazione per la quale il dataset nel file `nlu.md` contiene sempre un numero di frasi bilanciate rispetto ad ogni template utilizzato durante la generazione del dataset. Perciò, effettuare lo split del dataset in due sottoinsiemi non avrebbe favorito la validazione del chatbot, poichè l'ipotetico dataset di test non avrebbe mai potuto contenere degli elementi sconosciuti al modello generato attraverso il dataset di train e, quindi, sarebbe venuto meno al suo scopo.

Tuttavia, si è optato di eseguire una fase di *cross validation* per avere una stima delle performance del modello e la sua comprensione al fine di migliorarlo

Inoltre, una seconda motivazione è stata la possibilità di effettuare delle stime sulla bontà del chatbot senza la necessità di suddividere fisicamente il dataset in due sotto-insiemi (per il train ed il test), così favorendo l'agilità nello sviluppo di nuove funzionalità richieste. Tornando a discutere del bilanciamento del dataset in `nlu.md`, come già accennato, al suo interno è contenuto, per ogni frase, un numero sempre equivalente di frasi generate dallo stesso template dal servizio *Dataset Generator*. Tuttavia, l'unica accortezza indispensabile al mantenimento di tale risultato riguarda il numero di template definiti per ciascuno degli *intent*, poichè tale fattore risulterebbe l'unico che provocherebbe uno sbilanciamento nel dataset. Un'intent con più template da cui poter generare avrà più frasi rispetto agli altri.

Per avviare la validazione del modello si è eseguito il seguente comando:

```
1       rasa test nlu -u data/nlu.md --config config.yml --cross-validation
```

Come già illustrato, tale comando esegue una validazione incrociata utilizzando esclusivamente il dataset `nlu.md` e la *pipeline* definita in `config.yml`.

Dopodichè, è stata eseguita la seconda tipologia di validazione del chatbot, ovvero la *Core Model Evaluation*. In questa fase, è stato utilizzato il seguente comando:

```
1       rasa test core --stories test_stories.md --out results
```

Come è possibile osservare, viene definita la cartella `results` per contenere l'output e il file contenente le *story* con cui valutare il modello (nel caso esso non venisse specificato, verrebbe preso per default l'ultimo che è stato addestrato).

Il file contenente le *story* di test è denominato `test_stories.md` ed è stato definito dalle storie con il nominativo *happy path* prelevate dal file `stories.md`.

6.2.2 Risultati della validazione

I risultati della validazione saranno illustrati rispettando la suddivisione secondo le due fasi sopra descritte.

Per quanto concerne l'*NLU Model Evaluation* vengono restituiti sei file contenenti i risultati della validazione.

Il primo che sarà analizzato è l'`intent_report.json`, il quale contiene le metriche di ogni intent. Esso viene mostrato nel Listato 6.1:


```
{
  "query_details": {
    "precision": 0.990990990990991,
    "recall": 0.99,
    "f1-score": 0.9904952476238119,
    "support": 1000
  },
  "query_knowledge_list": {
    "precision": 0.9933774834437086,
    "recall": 1.0,
    "f1-score": 0.9966777408637874,
    "support": 1800
  },
  "query_knowledge_attribute_of": {
    "precision": 0.9932301740812379,
    "recall": 0.9827751196172249,
    "f1-score": 0.9879749879749881,
    "support": 1045
  },
  "micro avg": {
    "precision": 0.9927178153446033,
    "recall": 0.9927178153446033,
    "f1-score": 0.9927178153446033,
    "support": 3845
  },
  "macro avg": {
    "precision": 0.9925328828386458,
    "recall": 0.9909250398724083,
    "f1-score": 0.9917159921541958,
    "support": 3845
  },
  "weighted avg": {
    "precision": 0.9927167732391572,
    "recall": 0.9927178153446033,
    "f1-score": 0.9927045627080602,
    "support": 3845
  }
}
```


Listato 6.1. Contenuto del file `intent_report.json`

Dall'analisi di questo listato, è possibile constatare un valore di *precision*[1] intorno allo 0,99 per ciascuno degli *intent* classificati. Lo stesso risultato è osservabile per la misura del *recall*[2], la quale, per ogni intent, risulta mediamente sopra lo 0,99. Di conseguenza, poichè la *precision* e il *recall* risultano di valore elevato, per ogni

[1] La *precision* misura il rapporto tra le istanze classificate correttamente e le istanze totali classificate con la medesima classe di oggetti.

[2] Il *recall*, detto anche *true positive rate*, misura il rapporto di istanze positive correttamente etichettate dal sistema di machine learning rispetto al totale delle istanze della classe stessa.

intent, anche la *f1-score*, che si ottiene a partire da queste due misura, risulta essere sopra lo 0,99 su un valore massimo di 1 (99%). Tutto ciò comporta una valutazione positiva sulla capacità del modello di identificare gli *intent* del chatbot; tale risultato è avvalorato dalla matrice di confusione (Figura 6.1), nella quale è possibile osservare come la classificazione degli *intent* abbia prodotto pochi risultati errati (minori di 10 per ogni intent); questo vuol dire che nessun *intent* ha problemi a distinguersi dagli altri *intent*.

Tuttavia, nell'analizzare tali risultati, è impossibile non tener conto che, anche eseguendo una *cross validation*, tutti i dataset di train e test che vengono creati ad ogni iterazione contengono mediamente sempre le medesime frasi per ogni tipologia; perciò, non è possibile definire con certezza se il modello del chatbot Medici-bot sia in grado di generalizzare.

Di fatto, nel momento in cui Medici-bot è stato fornito in esame a degli utenti per eseguire dei test, esso in rare occasioni non è riuscito perfettamente a identificare le richieste espresse in linguaggio naturale e con delle sintassi delle frasi diverse da quelle utilizzate durante il train del modello.

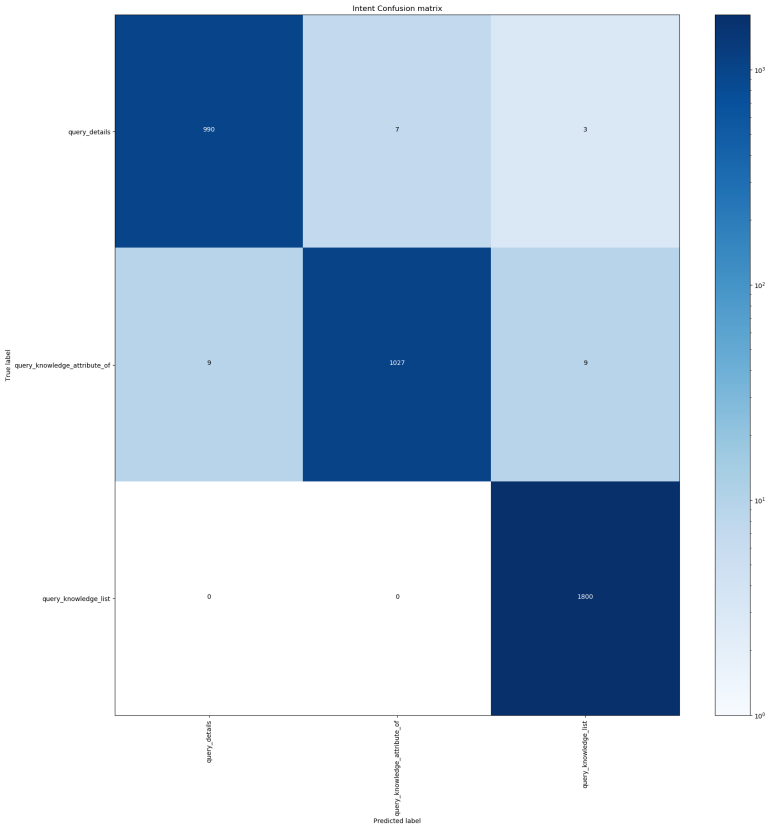

Figura 6.1. Matrice di confusione degli *intent*

Il terzo risultato della fase di validazione del modello NLU risulta essere il file

`intent_errors.json` (Listato 6.2); in esso è contenuta la totalità delle frasi che hanno rappresentato un errore di classificazione durante la validazione del modello NLU.

All'interno del listato sono presenti solo alcuni dei 28 casi di errore, e come è possibile notare, per ognuno di essi, è riportata la misura della *confidence* con cui è stata effettuata la classificazione. Inoltre, sono riportati i campi dell'*intent* corretto e dell'*intent* predetto.

Infine, al suo interno i casi di errore riguardano maggiormente le frasi di tipo *query_details* e *query_knowledge_attribute_of*.


```
{
    "text": "il suo comune èMEZZANA MORTIGLIENGO",
    "intent": "query_details",
    "intent_prediction": {
      "name": "query_knowledge_list",
      "confidence": 0.5991377234458923
    }
},
{
    "text": "il suo comune ?",
    "intent": "query_knowledge_attribute_of",
    "intent_prediction": {
      "name": "query_details",
      "confidence": 0.9442260265350342
    }
},
  {
    "text": "SAN MAURIZIO D OPAGLIO",
    "intent": "query_details",
    "intent_prediction": {
      "name": "query_knowledge_list",
      "confidence": 0.596102237701416
    }
},
  {
    "text": "dimmi tel del dottore GRAZIOLI",
    "intent": "query_knowledge_attribute_of",
    "intent_prediction": {
      "name": "query_knowledge_list",
      "confidence": 0.9371774792671204
    }
},
{
    "text": "la dottoressa SERENO quale comune possiede",
    "intent": "query_knowledge_attribute_of",
    "intent_prediction": {
      "name": "query_knowledge_list",
      "confidence": 0.7985691428184509
    }
}
```


Listato 6.2. Parte del contenuto del file `intent_errors.json`

Dopodichè, in Figura 6.2, è riportato l'istogramma che evidenzia la distribuzione di confidenza per tutte le predizioni.

Gli ultimi due file risultati dalla valutazione del modello NLU riguardano l'estrazione delle *entity*, e sono, rispettivamente, `CRFEntityExtractor_report.json` e `CRFEntityExtractor_errors.json`.

Come è possibile osservare nel Listato 6.3, le *entity* che hanno riportato le valutazioni più basse sono state *orariodiapertura*, *civico* e *orariodichiusura*.


```
{
  "giorno": {
    "precision": 0.998003992015968,
    "recall": 1.0,
    "f1-score": 0.9990009990009989,
    "support": 500
  },
  "nome": {
    "precision": 0.9559300064808814,
```

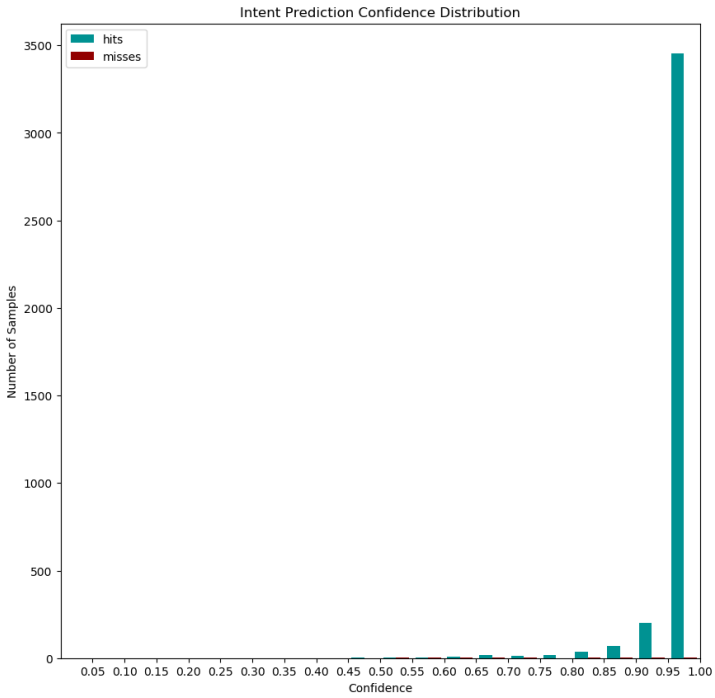

Figura 6.2. Istogramma della distribuzione della confidenza per tutte le previsioni


```
    "recall": 0.9282567652611705,
    "f1-score": 0.941890166028097,
    "support": 1589
  },
  "orariodiapertura": {
    "precision": 0.7415730337078652,
    "recall": 0.66,
    "f1-score": 0.6984126984126984,
    "support": 100
  },
  "attribute": {
    "precision": 0.9973118279569892,
    "recall": 0.9951716738197425,
    "f1-score": 0.9962406015037594,
    "support": 1864
  },
  "mention": {
    "precision": 0.9794721407624634,
    "recall": 0.9681159420289855,
    "f1-score": 0.9737609329446064,
    "support": 345
  },
  "indirizzo": {
    "precision": 0.9643493761140820,
    "recall": 0.9092436974789916,
    "f1-score": 0.9359861591695502,
    "support": 595
  },
  "comune": {
    "precision": 0.8746031746031746,
    "recall": 0.9429549343981746,
    "f1-score": 0.9074938237716167,
    "support": 1753
  },
  "cognome": {
    "precision": 0.8983164983164983,
    "recall": 0.8280571073867163,
    "f1-score": 0.8617571059431525,
    "support": 1611
  },
  "orariodichiusura": {
    "precision": 0.6936936936936937,
    "recall": 0.77,
```

```
    "f1-score": 0.7298578199052133,
    "support": 100
  },
  "object_type": {
    "precision": 1.0,
    "recall": 0.9937163375224417,
    "f1-score": 0.9968482665466007,
    "support": 1114
  },
  "civico": {
    "precision": 0.9361702127659575,
    "recall": 0.6423357664233577,
    "f1-score": 0.7619047619047619,
    "support": 137
  },
  "micro avg": {
    "precision": 0.9423919849718222,
    "recall": 0.9301606922126081,
    "f1-score": 0.9362363919129083,
    "support": 9708
  },
  "macro avg": {
    "precision": 0.9126749051288702,
    "recall": 0.8761683840290527,
    "f1-score": 0.891195757739187,
    "support": 9708
  },
  "weighted avg": {
    "precision": 0.9430180291229087,
    "recall": 0.9301606922126081,
    "f1-score": 0.9356036346297533,
    "support": 9708
  }
}
```


Listato 6.3. Contenuto del file `CRFEntityExtractor_report.json`

Nel Listato 6.4 è presente una parte, illustrata a scopo esplicativo, del contenuto del file `CRFEntityExtractor_error.json`; al suo interno sono riportate le classificazioni errate delle *entity*, con la relativa misura di confidenza, e le tipologie di *entity* predette erroneamente in ognuna delle frasi.


```
{
   "text": "il suo nome èVIOLA",
   "entities": [
     {
       "start": 14,
       "end": 19,
       "value": "VIOLA",
       "entity": "nome"
     }
   ],
   "predicted_entities": [
    {
      "start": 14,
      "end": 19,
      "value": "viola",
      "entity": "comune",
      "confidence": 0.41997210503415083,
      "extractor": "CRFEntityExtractor"
    }
   ]
  },
  {
   "text": "si chiama VALERIA IRMA",
   "entities": [
     {
       "start": 10,
       "end": 22,
       "value": "VALERIA IRMA",
       "entity": "nome"
     }
   ],
   "predicted_entities": [
    {
      "start": 10,
      "end": 17,
      "value": "valeria",
      "entity": "nome",
      "confidence": 0.9908318602727217,
      "extractor": "CRFEntityExtractor"
    },
    {
      "start": 28,
```

```
43          "end": 22,
44          "value": "irma",
45          "entity": "cognome",
46          "confidence": 0.693444435457363,
47          "extractor": "CRFEntityExtractor"
48        }
49      ]
50    },
```


Listato 6.4. Parte del contenuto del file `CRFEntityExtractor_error.json`

I casi più frequenti nei quali si sono riscontrati dei malintesi tra le *entity* sono risultati nelle frasi con entità composte da più termini, ad esempio un doppio nome ("Mattia Alberto") o nominativi di comuni composti da due parole ("Aqui Terme").

Di seguito, saranno illustrati i risultati ottenuti dalla valutazione del modello Core. Rasa, come anticipato nella Sottosezione 6.1, fornisce due file; il primo è il file `story_confmat.pdf`, dove, al suo interno, è presente la *matrice di confusione* delle *action* (Figura 6.3.

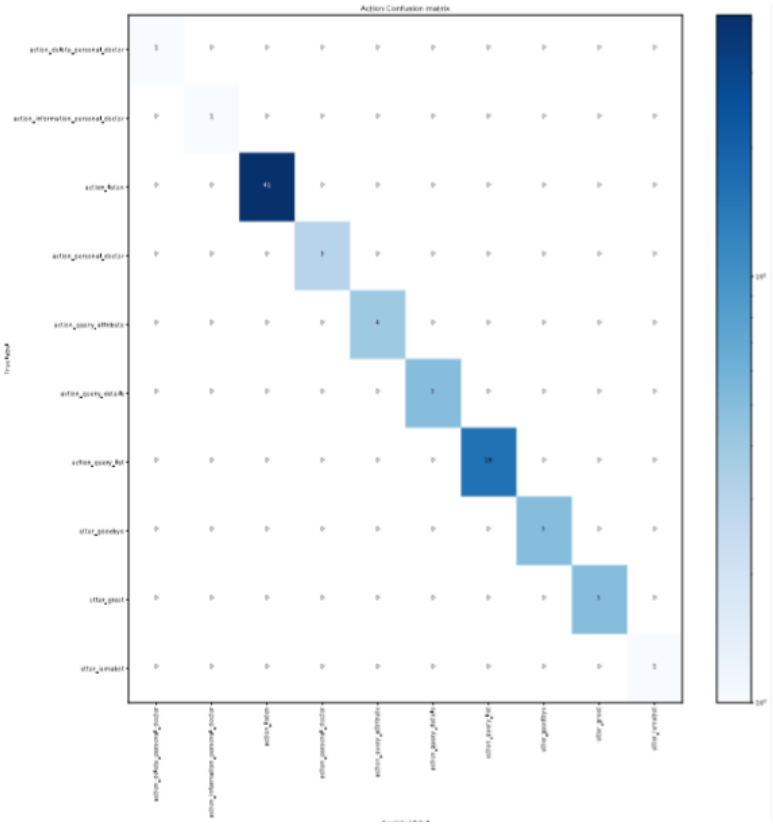

Figura 6.3. Matrice di confusione delle *action* rilevate

Come è possibile constatare, la validazione del modello Core ha prodotto una classificazione priva di errori.

Tale risultato è constatabile all'interno del secondo file, `failed_stories.md`, nel quale non sono state riscontrate storie fallite; al suo interno è riportato il seguente messaggio (Listato 6.5):

```
1   <!-- All stories passed -->
```

Listato 6.5. Contenuto del file `failed_stories.md`

6.3 Test delle funzionalità aggiuntive del chatbot

Al termine della validazione dei modelli, è stato effettuato un test sulle funzionalità aggiunte del chatbot, ovvero l'*account linking*, il salvataggio e l'eliminazione del dottore preferito ed il link alla posizione dell'ambulatorio su Google Maps.

Tale verifica può essere considerata un `beta test`, poichè è stato chiesto ad alcuni utenti di effettuare delle verifiche sulle funzionalità di Medici-bot.

Perciò, dopo il loro esame di tutte le tipologie di conversazione attuabili (ad esempio, il dialogo in Figura 6.4), si è verificata l'autenticazione dell'utente attraverso il proprio account Google e, a seguire, il salvataggio del medico di fiducia tra le informazioni personali dell'utente (Figura 6.5).

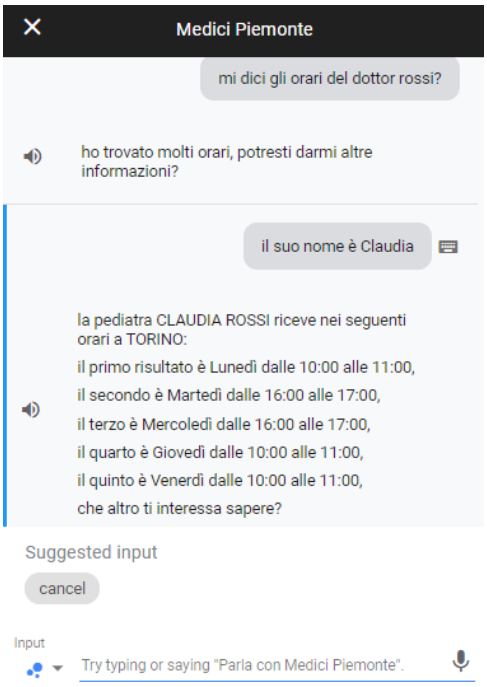

Figura 6.4. Esempio di una richiesta degli orari di un dottore

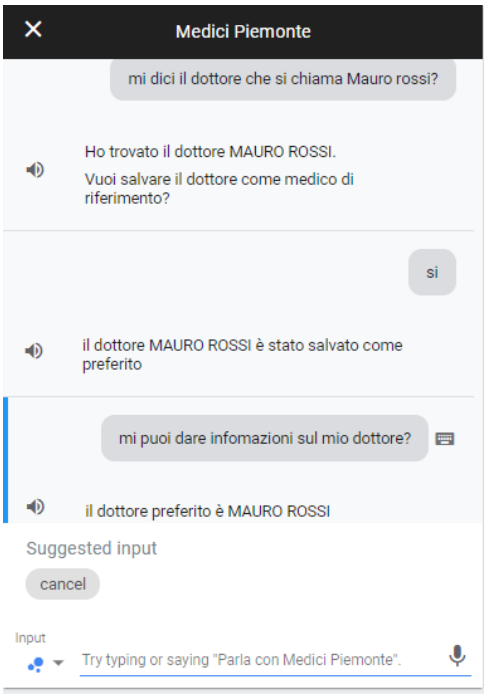

Figura 6.5. Dialogo contenente il salvataggio del medico preferito

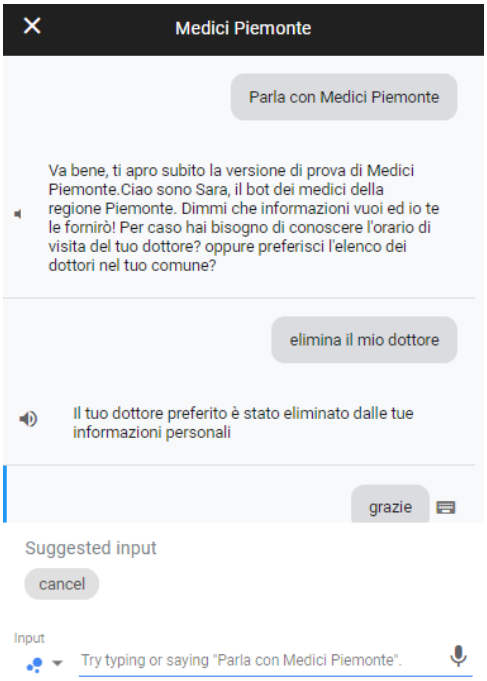

Figura 6.6. Dialogo contenente l'eliminazione del medico preferito

Dopodichè, è stata verificata la richiesta dell'ambulatorio ed il pulsante contenente il link sulla posizione di tale ambulatorio su Google Maps (Figura 6.7).

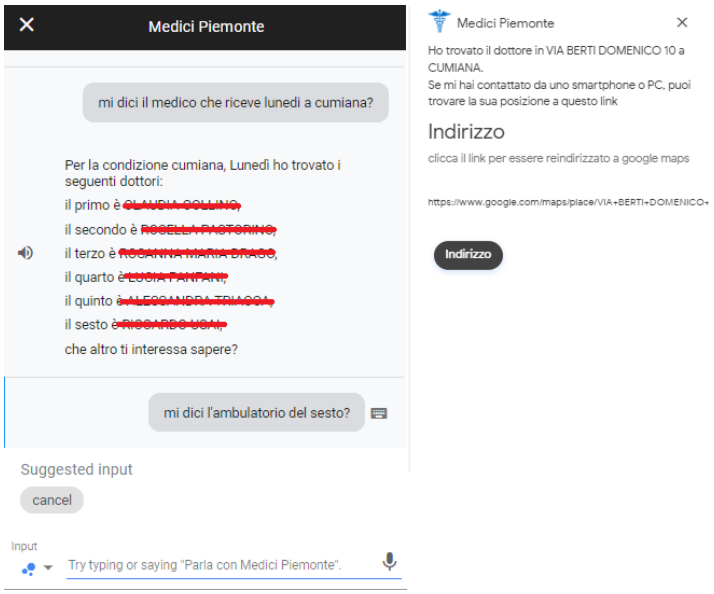

Figura 6.7. Dialogo relativo alla richiesta degli ambulatori e risposta contenente la basicCard con link alla posizione su Google Maps

Infine è stato riproposto l'equivalente test su Alexa; tenendo conto che tali funzionalità non sono state implementate per questa piattaforma (Alexa non garantisce la possibilità di fornire link esterni o di associare all'utente delle informazioni aggiuntive), è stata effettuata una verifica delle sole conversazioni di base del chatbot.

7

Conclusioni e futuri sviluppi

In questo ultimo capitolo saranno tratte le conclusioni riguardanti il lavoro svolto; verranno, inoltre, illustrati alcuni possibili sviluppi futuri già richiesti dagli stakeholder.

7.1 Conclusioni

In questa tesi è stato proposto lo sviluppo del chatbot Medici-bot e delle soluzioni sviluppate per la creazione di chatbot basati su Rasa e knowledge base Neo4j. Come già descritto nel corso della trattazione, l'obiettivo principale era la realizzazione di software riutilizzabili in altri contesti e la definizione di uno standard procedurale per la costruzione di chatbot che volessero rispettare la medesima architettura.

Nella prima parte si è esaminata la generazione del dataset per il train del modello NLU, illustrando il *Dataset Generator*, con il quale, attraverso la sola definizione dei template ed i dati della knowledge base, è possibile ottenere in output il dataset per il training del bot. Inoltre, tale software è stato modificato per adattarsi ad ogni grafo dei dati presenti nel database.

Nella seconda parte, si sono esaminate le classi del chatbot; in esse si è cercato di astrarre, il più possibile, il contesto dalla sua implementazione, cercando, inoltre, di generalizzare i metodi per ogni scenario di utilizzo. Ne sono stati degli esempi la creazione del file `file_utils.json` e i metodi di supporto. Tali metodi, all'interno di nuovi progetti, potranno essere riutilizzati semplicemente modificando i dati del file `file_utils.json`.

Dopo di ciò, sono stati presentati i risultati ottenuti dall'analisi delle performance attuali del chatbot, e si è visto che quest'ultimo presenta margini di miglioramento per il futuro.

Per concludere, è bene chiarire che il risultato della corrente elaborazione, con elevate probabilità, non andrà mai in produzione, poichè è stato concepito principalmente come una soluzione "muletto" all'interno di un progetto di ricerca e sviluppo, nel quale il CSI-Piemonte tenta di accumulare conoscenze da poter inserire in progetti più concreti.

7.2 Sviluppi futuri

Il lavoro esaminato in questa trattazione rappresenta una piccola parte dei progetti chatbot che sono sviluppati dal team con cui si è lavorato durante il tirocinio connesso alla presente tesi. Ognuno di questi bot intercetta esigenze diverse, ed i più vari scenari di utilizzo, al fine di accrescere la conoscenza dell'Azienda ed avere a disposizione delle architetture software utilizzabili a seconda dell'esigenza.

Gli sviluppi maggiormente probabili, che potrebbero nascere dall'architettura creata per Medici-bot sono delle finestre di dialogo all'interno dei siti regionali e comunali del Piemonte.

A tal fine, sono presenti numerosi aspetti e funzionalità ancora sviluppabili o in corso di approfondimento. Tra questi, troviamo la possibilità di utilizzare il riferimento al dottore preferito all'interno di una richiesta formulata dall'utente. Un ulteriore *intent* in fase di studio consiste nel fornire una descrizione di un dottore nell'istante in cui essa venga richiesta (possono essere utilizzate le *card*, le quali vengono gestite e implementate sia da Google Assistant che da Alexa).

Dopodichè, al momento attuale, è in studio la possibilità di autenticarsi utilizzando non più le credenziali di Google, bensì quelle per la piattaforma *TorinoFacile*; ciò permetterebbe un'autenticazione attraverso *SPID* (l'acronimo di Sistema Pubblico di Identità Digitale).

Mentre, per quanto concerne la distribuzione del bot sulla piattaforma Alexa, il primo passo sarà costituito dalla realizzazione dell'*Account Linking*, sempre attraverso l'autenticazione di *TorinoFacile*, per poi proseguire sperimentando la possibilità di definire le funzionalità aggiuntive descritte in questo elaborato.

Tuttavia, lo sviluppo che sarà portato avanti con più urgenza riguarderà il miglioramento del modello NLU e la totale decontestualizzazione dei metodi implementati in Medici-bot, così da soddisfare l'obiettivo principale di tale progetto, ovvero la realizzazione di una procedura per la creazione di chatbot basati sull'architettura di Medici-bot.

Riferimenti bibliografici

1. Redazione Osservatori Digital Innovation, Cosa sono i Chatbot e come possono essere sfruttati dalle aziende. https://blog.osservatori.net/it_it/chatbot-cosa-sona-come-utilizzarli.
2. Architettura di rasa. https://rasa.com/docs/rasa/user-guide/architecture/.
3. chatbot. https://it.wikipedia.org/wiki/Chat_bot.
4. Guida all'utilizzo del framework Rasa e sviluppo applicativo del chatbot all'interno di un progetto. https://www.qi-lab.it/2019/10/12/sviluppo-un-chatbot-framework-rasa/.
5. Machine Translation. http://it.creative-words.com/blog/che-cose-la-machine-translation-e-come-funziona/.
6. Natural language Processing. https://en.wikipedia.org/wiki/Natural_language_processing#Rule-based_vs._statistical_NLP.
7. Question Answering. https://www.agendadigitale.eu/cultura-digitale/linguaggio-naturale-e-intelligenza-artificiale-a-che-punto-siamo/.
8. Rasa, Testing your assistant. https://rasa.com/docs/rasa/user-guide/testing-your-assistant/.
9. Myclever Agency: Chatbots. A consumer research study. https://www.mycleveragency.com/media/download/0c44f0c083879818a0d2347ab948752b, 2016.
10. Chatbot. Linee guida alla progettazione per le aziende. Il caso Wind. http://hdl.handle.net/10589/138320, 2017.
11. Allison Ragan. Taking the Confusion Out of Confusion Matrices. https://towardsdatascience.com/taking-the-confusion-out-of-confusion-matrices-c1ce054b3d3e?gi=d27279682f47, 2018.
12. Natural Language Processing: Chatbot per gli studenti del campus di Siena. https://amslaurea.unibo.it/19555/1/tesi_andrea_lavista_studio_unibo_it.pdf, 2019.
13. Timothy W Bickmore, Ha Trinh, Stefan Olafsson, Teresa K O'Leary, Reza Asadi, Nathaniel M Rickles, and Ricardo Cruz. Patient and consumer safety risks when using conversational assistants for medical information: an observational study of siri, alexa, and google assistant. *Journal of medical Internet research*, 20(9):e11510, 2018.
14. Tom Bocklisch, Joey Faulkner, Nick Pawlowski, and Alan Nichol. Rasa: Open source language understanding and dialogue management. *arXiv preprint arXiv:1712.05181*, 2017.
15. J Castaño, H Park, P Ávila, D Pérez, H Berinsky, L Gambarte, C Otero, and D Luna. Evaluating the performance of a terminology search engine using historical data. *Studies in health technology and informatics*, 264:1439–1440, 2019.

16. Srini Janarthanam. *Hands-on chatbots and conversational UI development: build chatbots and voice user interfaces with Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio, and Alexa Skills*. Packt Publishing Ltd, 2017.
17. Anran Jiao. An intelligent chatbot system based on entity extraction using rasa nlu and neural network.
18. Andrew Rafla and Casey Kennington. Incrementalizing rasa's open-source natural language understanding pipeline. *arXiv preprint arXiv:1907.05403*, 2019.
19. Andreas Reiswich and Martin Haag. Evaluation of chatbot prototypes for taking the virtual patient's history. *Studies in health technology and informatics*, 260:73–80, 2019.
20. Dipanjan Sarkar. *Text analytics with Python: a practitioner's guide to natural language processing*. Apress, 2019.
21. Abhishek Singh, Karthik Ramasubramanian, and Shrey Shivam. *Building an enterprise chatbot: Work with protected enterprise data using open source frameworks*. Springer, 2019.
22. Jalaj Thanaki. *Python natural language processing*. Packt Publishing Ltd, 2017.
23. Jim Webber and Ian Robinson. *A programmatic introduction to neo4j*. Addison-Wesley Professional, 2018.

Ringraziamenti

Il primo ringraziamento va alla mia famiglia, che mi ha sostenuto e mi ha dato la possibilità di intraprendere e portare a termine questo percorso universitario.

Vorrei ringraziare il prof. Ursino, per aver accettato di essere il mio relatore, per essersi speso nell'aiutarmi a trovare il percorso di tirocinio più adatto alle mie esigenze — scaturito, poi, in quella che è la mia attuale occupazione — e per la straordinaria disponibilità dimostrata nel corso della stesura di questa tesi.

Ringrazio i miei amici di sempre, per il sostegno, per l'affetto dimostratomi in questi lunghi anni e per aver reso questo periodo più lieto.

Ringrazio i miei compagni di università, soprattutto Enrico ed Edoardo, che mi hanno accompagnato in tutti i progetti in cui siamo stati coinvolti.

Per ultima, ma non per importanza, vorrei ringraziare la mia compagna, per avermi sostenuto e sopportato sempre, per avermi seguito nel mio rischioso progetto di vita e per aver trasformato una quarantena in un periodo lieto e colmo di gioie.