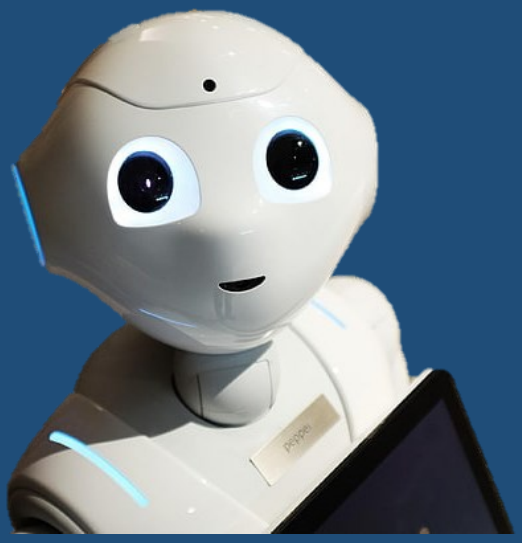
Università degli Studi di Salerno
DIEM, a. a. 2023-24
Corso: COGNITIVE ROBOTICS
Professoressa: Alessia Saggese



Pepper: il guardiano robotico del centro commerciale

Salvatore Grimaldi 0622701742 s.grimaldi29@studenti.unisa.it
Mariapia Lombardi 0622701957 m.lombardi59@studenti.unisa.it
Lorenzo Mignone 0622701866 l.mignone@studenti.unisa.it
Carlo Santonicola 0622702027 c.santonicola7@studenti.unisa.it

Sommario

1. Introduzione

Il progetto finale del corso di **Cognitive Robotics** dell'a.a. 2023-24, tenuto dalla professoressa Alessia Saggese presso il Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica Applicata (**DIEM**) dell'Università degli Studi di Salerno, consiste nella realizzazione di un *guardiano robotico per un centro commerciale*. Il presente report fornisce un'analisi dettagliata delle scelte progettuali ed implementative del sistema.

Il robot adoperato nell'ambito del progetto è **Pepper**. Presentato nel 2014 dalla *SoftBank Robotics*, esso è ad oggi uno dei più famosi e popolari robot sociali in commercio. L'obiettivo fondamentale del progetto è realizzare un'applicazione robotica che consenta a Pepper di svolgere il ruolo di guardiano di un centro commerciale. In particolare, sulla base di informazioni precedentemente acquisite da un'applicazione di analisi video, Pepper è in grado di rispondere a domande come quesiti circa il numero totale di persone presenti, il numero di persone con determinati attributi presenti in un certo negozio, la localizzazione di un soggetto con determinate caratteristiche, ecc. Il robot comunica con l'interlocutore, che si assume essere un cliente del centro di acquisti, attraverso linguaggio naturale, sfruttando una camera, un microfono, un tablet ed un sintetizzatore vocale. È lo stesso sistema che ingaggia l'utente, quando ne rileva il volto. Inoltre, l'interazione avviene in modo socialmente accettabile e piacevole: grazie ad un modulo di Natural Language Processing (**NLP**) basato su **Rasa**, il robot è capace di comprendere le richieste avanzate dal cliente e rispondervi in modo adeguato.

Prima di entrare nel merito della descrizione del sistema, è opportuno soffermarsi sugli *environmental constraints* stabiliti dalla committenza. Si assume che l'utente si collochi davanti al robot ad una distanza ragionevolmente limitata in un ambiente ragionevolmente poco rumoroso. Più nello specifico, si assume che la distanza tra l'utente ed il robot sia minore di 3 metri, che non vi siano eccessivi rumori ambientali durante la conversazione, e che il robot interagisca con una sola persona alla volta.

Il report si sviluppa nel modo seguente: il *Capitolo 2* fornisce una descrizione dell'architettura **ROS**; il *Capitolo 3* è incentrato su una spiegazione chiara e puntuale dei nodi ROS e delle relative scelte implementative; il *Capitolo 4* descrive la progettazione del Task Oriented Dialogue System (**TOD**) ed il suo sviluppo in Rasa; il *Capitolo 5* parla dettagliatamente dei moduli di **face detection** e **speech to text**; il *Capitolo 6* descrive l'intera pianificazione dei **test**; infine, il *Capitolo 7* riporta i risultati di questi ultimi, opportunamente commentati.

Figura 1. Pepper in un centro commerciale [immagine generata da AI]

2. WP1: Progettazione dell'architettura basata su ROS

Lo sviluppo di questa applicazione robotica è stato eseguito utilizzando il framework ROS, ampiamente diffuso nella programmazione dei robot. Il linguaggio di programmazione impiegato è Python. Dalla seguente immagine si evince l'architettura adoperata:

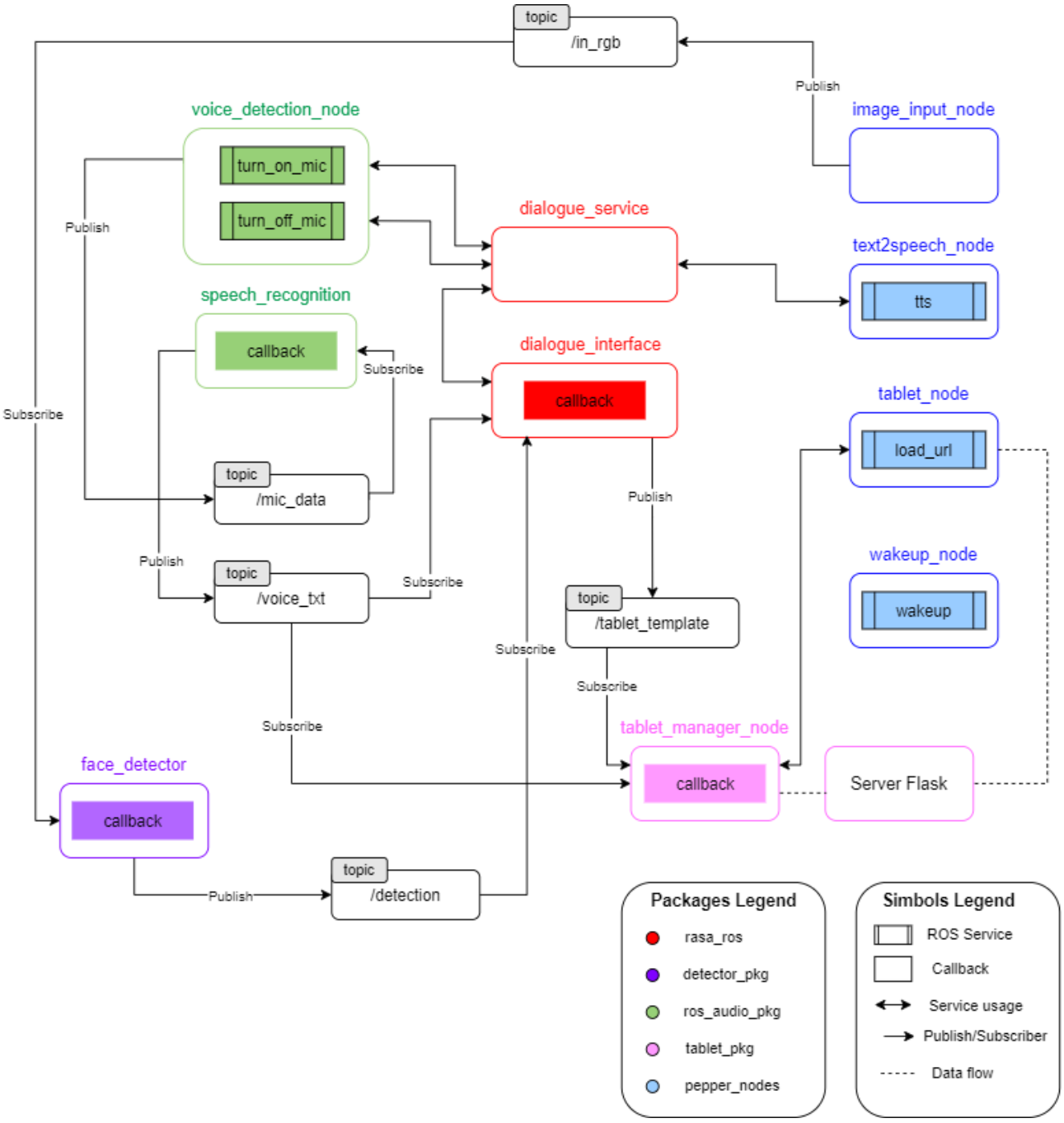


Figura 2. Architettura ROS del progetto

Dall'immagine si può apprezzare come i nodi realizzati sfruttino le due modalità di comunicazione messe a disposizione da ROS: i **servizi** basati sull'architettura client/server e la comunicazione basata sull'architettura **publisher/subscriber**. Per una descrizione più approfondita dei pacchetti ROS e dei loro nodi si rimanda al Capitolo 3.

3. WP2: Implementazione dell'architettura basata su ROS

Questo pacchetto descrive approfonditamente i pacchetti ROS adoperati, *detector_pkg*, *ros_audio_pkg*, *pepper_nodes*, *rasa_ros*, ed i nodi in essi contenuti.

3.1 Pacchetto *detector_pkg*

Nel pacchetto *detector_pkg* sono presenti il nodo ed i pesi della rete neurale che effettuano la face detection. Di seguito, ci concentreremo maggiormente sull'integrazione di tali componenti all'interno del framework ROS. Per maggiori dettagli circa la rete di face detection utilizzata si rimanda al Capitolo 5.

3.1.1 *face_detector*

Il nodo *face_detector* si sottoscrive al topic */in_rgb*, da cui riceve i frame della telecamera, e pubblica sul topic */detection* nel caso in cui vengano rilevati volti.

Dopo diversi tentativi, si è deciso di fissare la frequenza di pubblicazione della telecamera di Pepper a **20 fotogrammi al secondo (FPS)**. Tale decisione rappresenta un equilibrato compromesso tra la complessità computazionale e l'accuratezza dei risultati. In termini pratici, la rete neurale dimostra di essere efficiente a gestire 20 fps, consentendo dunque un numero elevato di inferenze al secondo, il che è fondamentale per garantire un'elevata velocità di elaborazione del sistema nel suo complesso. Al fine di garantire una valutazione più robusta della presenza di un utente di fronte al robot, si è deciso di realizzare un meccanismo di sogliatura, tale per cui *face_detector* pubblica 1 sul topic */detection* solo se almeno la metà delle ultime **MAXLEN** inferenze effettuate dalla rete hanno rilevato la presenza di almeno un volto. In caso contrario, *face_detector* non pubblica nulla sul topic */detection*. In particolare, il nodo decide se pubblicare o meno sul topic /detection ogni **INTERVAL_TIME** secondi, che si è fissato uguale a 0.5. Tale scelta è un compromesso ragionevole tra reattività desiderata per l'engagement e la robustezza delle informazioni provenienti da *face_detector*. Si osservi che MAXLEN è, di fatto, la dimensione di una coda, ed è uguale al prodotto **FPS * INTERVAL_TIME = 10**. In tal modo, ogni pubblicazione tiene conto delle inferenze effettuate sugli ultimi 10 frames, e la coda viene del tutto aggiornata tra una pubblicazione e la successiva.

3.2 Pacchetto *ros_audio_pkg*

Il pacchetto *ros_audio_pkg* contiene i nodi che si occupano della gestione del microfono e del task di speech2text. Di seguito, ci concentreremo maggiormente sull'integrazione di tali componenti all'interno del framework ROS. Per maggiori dettagli circa lo speech2text si rimanda al Capitolo 5.

3.2.1 *voice_detection_node*

Lo scopo del nodo *voice_detection_node* consiste nell'inizializzare ed acquisire l'audio dal microfono esterno **ReSpeaker Mic Array**, sfruttando un VAD (Voice Activity Detection) basato su energia messo a disposizione dalla libreria Python **SpeechRecognition**. Si osservi che si è scelto di adoperare il microfono sopracitato invece del microfono di uno dei laptop dei componenti del team poiché il Respeaker Mic Array effettua abbattimento del rumore, consentendo quindi di passare al nodo deputato allo speech to text una traccia audio senza particolari disturbi. Dopo numerosi esperimenti in scenari conformi agli *environmental constraints*, si è deciso di fissare la soglia di energia *energy_threshold* del VAD a 100. Infatti, la calibrazione dinamica resa disponibile da SpeechRecognition non ha dato sufficienti garanzie, comportando spesso un eccessivo aumento della *energy_threshold*.
Ad ogni rilevazione di audio che supera la soglia di energia viene attivata una *callback* che pubblica sul topic */mic_data.* Inoltre, è stato configurato il parametro ***pause_threshold*** a **1 secondo**, in modo che ogni frammento di audio possa considerarsi concluso dopo un secondo di "silenzio". Durante alcuni scenari di prova, è emerso che Pepper può rilevare la propria voce durante la conversazione, pertanto sono stati aggiunti i servizi *turn_on_mic* e *turn_off_mic* rispettivamente per **attivare** e **disattivare** il **microfono** quando il robot comunica con l'utente. La scelta di implementarli come servizi è dovuta alla necessità di sincronizzazione nell'utilizzo del microfono. Altrimenti, utilizzando un'architettura publisher/subscriber, si otterrebbero sovrapposizioni indesiderate di callback causando accessi concorrenti alla medesima risorsa.

3.2.2 *speech_recognition*

Il nodo *speech_recognition* si sottoscrive al topic */mic_data* e, alla ricezione di una traccia audio, attiva una *callback*, la cui funzione è quella di utilizzare **Google Speech Recognition** per la trascrizione della traccia audio, per poi pubblicare

il risultato in modo opportuno sul topic */voice_txt*. Questo processo prepara la risposta dell'utente ad essere elaborata da altri nodi.

3.3 Pacchetto *pepper_nodes*

Nel pacchetto *pepper_nodes* sono presenti tutti i nodi che consentono sia il funzionamento di Pepper e sia l'utilizzo di accessori che facilitano l'interazione dell'utente con esso. L'implementazione di questi nodi è stata fornita direttamente dal DIEM.

Dei nodi forniti solo alcuni sono stati utilizzati e sono:

- ***wake_up_node:*** implementa il servizio *wakeup*, il quale consente il *bringup* di Pepper.
- ***text2speech_node:*** implementa il servizio *tts*, consentendo l'utilizzo della voce di Pepper.
- ***tablet_node:*** implementa il servizio *load_url*, consentendo la visualizzazione delle pagine html sul browser del tablet a disposizione del robot.
- ***image_input_node:*** usa l'architettura publisher/subscriber per pubblicare sul topic */in_rgb* i frame acquisiti dalla telecamera con una frequenza di **20 fps**.

3.4 Pacchetto *tablet_pkg*

Il pacchetto *tablet_pkg* contiene tutti i moduli e i nodi necessari al funzionamento del tablet di Pepper.

3.4.*1 server_flask*

Il modulo ***server_flask*** implementa un *server web* in esecuzione sulla rete locale per consentire al tablet di caricare le *pagine HTML* definite nella cartella *templates*. *Flask* è il framework utilizzato per lo sviluppo web. Il server presenta tre possibili **percorsi**:

1. ***/static/index***: permette la visualizzazione della pagina *index.html* durante il periodo di attesa di interazione con l'utente.
2. ***/engagement***: consente la visualizzazione della pagina *engagement.html* durante la procedura di engagement.
3. ***/dialogue***: consente la visualizzazione della pagina *dialogue.html,* la quale riporta il testo trascritto dal nodo *speech_recognition*. Il testo viene passato come argomento all'*url*.

I templates sono stati realizzati utilizzando i linguaggi per lo sviluppo web Jinja, HTML, CSS e JavaScript. Sono presenti tre templates, *index.html, engagement.html, dialogue.html*, ciascuno corrispondente a uno dei percorsi precedentemente menzionati.

Nel file ***index.html***, viene eseguito il rendering di un'immagine specifica, ovvero *index.png*, la quale viene passata dinamicamente alla pagina HTML utilizzando Jinja. Seguono istruzioni per la centratura dell'immagine, garantendo così una visualizzazione adatta al tablet.

Per ***engagement.html***, vengono eseguite operazioni di centratura per l'animazione *cartoon-txt* la quale mostra la frase di *engagement* durante la procedura corrispondente.

Infine, per ***dialogue.html***, oltre alle operazioni di centratura in CSS, è presente uno script JavaScript per l'esecuzione di un'animazione che compone una frase inserendo un carattere alla volta. Lo script inizia acquisendo il testo dell'user message, fornito come argomento nell'url. Successivamente, viene creato un oggetto rappresentante la pagina HTML, a cui viene associata la funzione di animazione, la cui velocità di elaborazione è regolata dal parametro *delay.* Inoltre, è da evidenziare la presenza di un'icona di un altoparlante.

3.4.2 *tablet_manager_node*

La gestione della visualizzazione delle pagine HTML sul tablet, mediante la chiamata al servizio *load_url* fornito dal nodo *tablet_node* nel pacchetto *pepper_nodes*, è affidata al nodo *tablet_manager_node*. Quest'ultimo è sottoscritto a due topic, precisamente */tablet_template* e */voice_txt.*

Inizialmente, tale nodo effettua una richiesta di visualizzazione della pagina di default *index.html*.
La callback che si attiva in risposta al topic */tablet_template* consente di effettuare due richieste di visualizzazione: la prima per la pagina *engagement.html* all'inizio di una nuova conversazione con l'utente, e la seconda per la riproposizione della pagina di default al termine della conversazione.

D’altra parte, la callback associata al secondo topic */voice_txt* inoltra una richiesta di visualizzazione del testo trascritto dalla voce dell’utente, passato come argomento nell’URL.

Sebbene l’architettura basata su servizi sia generalmente più indicata per la gestione di risorse come il tablet, la scelta implementativa è stata orientata verso l’impiego dell’architettura publisher/subscriber per le seguenti ragioni:

1. **Bassa latenza:** il tempo di esecuzione dei servizi predisposti alla visualizzazione sul tablet non è tale da causare un disallineamento tra l’interazione utente-Pepper ed il tablet stesso.
2. **Frequenza della pubblicazione sul topic */voice_txt:*** l’architettura scelta si adatta bene in contesti con elevata frequenza di pubblicazione.
3. **Ruolo limitato del tablet:** il tablet non svolge un ruolo centrale all'interno del sistema, essendo utilizzato esclusivamente come dispositivo di visualizzazione. La sua funzione principale è facilitare la comprensione dell'utente rispetto a ciò che Pepper ha compreso. Non viene impiegato come elemento di input, non richiedendo così la necessità di un meccanismo di sincronizzazione specifico per gestire disallineamenti tra le visualizzazioni sul tablet e le azioni dell'utente.

3.5 Pacchetto *rasa_ros*

Nel pacchetto *rasa_ros* sono contenuti i nodi *dialogue_service* e *dialogue_interface*, i quali realizzano l’integrazione del chatbot Rasa nel framework ROS.

3.5.1 *dialogue_service*

Il servizio ROS *dialogue_server*, implementato nel nodo *dialogue_service*, permette l'invio di una stringa al chatbot di Rasa per ottenere una risposta in formato testuale. L'implementazione mira a garantire maggiore flessibilità e riutilizzo del nodo. In particolare, il nodo è stato reso adattabile a diverse applicazioni tramite l'opzione definita dalla costante booleana **PEPPER**. Questa costante consente **due modalità di funzionamento**:

1. **Prima modalità**: ricezione e invio di messaggi testuali da e verso il chatbot.
2. **Seconda modalità (scelta corrente)**: in aggiunta alle funzionalità della prima modalità, abilita la comunicazione verbale di Pepper. Ciò è reso possibile attraverso l’utilizzo dei servizi *turn_on_mic* e *turn_off_mic* del nodo *voice_detection_node*, appartenente al pacchetto *ros_audio_pkg*, insieme al servizio *tts* del nodo *text2speech_node* nel pacchetto *pepper_nodes.*

L'utilizzo dei servizi è motivato, come menzionato nei paragrafi precedenti, dal fatto che Pepper può ascoltare la propria voce durante le interazioni vocali. Per risolvere questa problematica, si è scelto di **disattivare il microfono** per un periodo di tempo almeno pari a quello necessario a Pepper per pronunciare una frase, definito da ***time_last_utterance***. Questo tempo può essere ulteriormente esteso di una quantità pari al tempo di esecuzione dei servizi. Successive ricerche sulla documentazione, supportate da sessioni di cronometraggio, hanno dimostrato che Pepper è in grado di pronunciare circa 100 parole in 25 secondi. Pertanto, considerando la lunghezza di una specifica frase, il tempo necessario a pronunciarla può essere calcolato come il **prodotto tra il numero di parole e 0.25**. Questa strategia è stata adottata per mitigare con successo il suddetto problema. Infine, è importante sottolineare che il nodo in questione ha anche il compito di disattivare il microfono ad ogni ricezione del messaggio */restart*, che si verifica al termine di ciascuna conversazione.

3.5.2 *dialogue_interface*

Il nodo *dialogue_interface* gestisce l’inizio e la fine delle conversazioni con l’utente, utilizzando informazioni fondamentali provenienti da altri nodi. Utilizza il servizio *dialogue_server* per la comunicazione con Rasa, si sottoscrive rispettivamente ai topic */detection* e */voice_txt* per monitorare la presenza di una persona e processare le richieste verbali dell'utente. Inoltre, pubblica sul topic */tablet_template* per visualizzare la pagina di *engagement.html* durante l’engagement e la pagina di default *index.html* quando è in attesa di nuove conversazioni.

La **logica operativa** dell'applicazione può essere delineata come segue: quando Pepper rileva una persona, avvia la fase di engagement, salutando e mostrando la pagina *engagement.html.* Da questo momento in poi, Pepper accende il microfono e avvia una conversazione. Dopo aver ottenuto il testo trascritto della richiesta verbale dell'utente, effettua una richiesta al chatbot tramite il servizio *dialogue_service*, il quale fornisce e pronuncia la risposta mantenendo il microfono spento. In aggiunta, mostra la pagina *dialogue.html* contenente quanto trascritto precedentemente. A questo punto, l'utente può presentare una nuova richiesta.

È importante sottolineare che la conversazione termina se l'utente non viene più rilevato dal modulo di face detection per più di **5 secondi (TIMEOUT_DETECTOR)** oppure se impiega più di **30 secondi (TIMEOUT_VOICE)** per rispondere a Pepper. La scelta di utilizzare un periodo di 30 secondi è motivata dal fatto che a volte il nodo di *speech_recognition* potrebbe richiedere del tempo prima di estrarre il testo dalla traccia audio.

Allo scadere di uno di questi *timeout*, Pepper ritiene conclusa la conversazione, quindi visualizza la pagina predefinita *index.html* sul tablet, resetta gli *slots* in Rasa, spegne il microfono e resta in attesa di un'interazione con un nuovo utente.

4. WP3: Integrazione del modulo di dialogue management

Questo capitolo descrive la progettazione del **chatbot** ed il suo sviluppo in **Rasa**, un framework *open source* molto popolare per la realizzazione di assistenti basati sull'*Intelligenza Artificiale*. Di seguito ci soffermeremo soprattutto sulle scelte progettuali riguardanti **intenti**, **entità**, **slots**, **forms**, **pipeline**, **policies**, **regole** e **custom actions**.

Il nostro chatbot è un *contextual assistant*, ovvero un Task-Oriented Dialogue (**TOD**) system. Esso, infatti, ha l'obiettivo di supportare l'interlocutore per obiettivi ben specifici in un contesto ben definito, in particolare *il conteggio e l'individuazione di persone, sulla base di alcuni attributi fisici, all'interno di un centro commerciale*.

È doveroso evidenziare che la maggior parte delle scelte progettuali del TOD system dipendono strettamente dalla tipologia di informazioni a disposizione del chatbot riguardanti i clienti del centro commerciale. Nello specifico, si ipotizza che all'interno della galleria commerciale in cui viene collocato il robot sia in esecuzione un sistema di analisi video che restituisce in output un file *database.json* formattato nel modo seguente:

```
1. {
2.     “people” : [
3.         {
4.             “id”: 1,
5.             “gender” : “male”,
6.             “bag” : true,
7.             “hat” : false,
8.             “upper_color” : “red”,
9.             “lower_color” : “blue”,
10.            “roi1_passages” : 1,
11.            “roi1_persistence_time” : 32,
12.            “roi2_passages” : 2,
13.            “roi2_persistence_time” : 45
14.        },
15.        {
16.            “id” : 2,
17.            “gender” : “female”,
18.            “bag” : false,
19.            “hat” : true,
20.            “upper_color” : “black”,
21.            “lower_color” : “white”,
22.            “roi1_passages” : 0,
23.            “roi1_persistence_time” : 0,
24.            “roi2_passages” : 1,
25.            “roi2_persistence_time” : 12
26.        }
27.    ]
28. }
```

In sostanza, il file fornisce una lista di persone, ciascuna delle quali rappresentata per mezzo di un dizionario che riporta le seguenti informazioni:

- *id:* identificativo univoco
- *gender:* male o female
- *bag:* true o false
- *hat:* true o false
- *upper_color:* black, blue, brown, gray, green, orange, pink, purple, red, white, yellow

- *lower_color:* black, blue, brown, gray, green, orange, pink, purple, red, white, yellow
- *roi1_passages:* numero di passaggi in ROI1
- *roi1_persistence_time:* tempo totale speso in ROI1, espresso in secondi
- *roi2_passages:* numero di passaggi in ROI2
- *roi2_persistence_time:* tempo totale speso in ROI2, espresso in secondi

Si osservi che per ROI1 e ROI2 si intendono due regioni di interesse collocate all'interno del centro commerciale. Si ipotizza per semplicità che ROI1 corrisponda ad un supermercato *Walmart* e che ROI2 corrisponda ad un una caffetteria *Starbucks*.

4.1 Intenti ed entità

La progettazione di un TOD System parte necessariamente dall'individuazione degli intenti e delle entità significativi per il dominio applicativo e gli obiettivi dell'interazione. Dato uno *user input*, l'intento ne rappresenta lo scopo, mentre le entità sono degli elementi informativi rilevanti al suo interno.

Gli intenti scelti sono i seguenti:

- ***greet**:* intenzione di iniziare una conversazione.
- ***goodbye**:* intenzione di concludere una conversazione.
- ***affirm**:* conferma o approvazione
- ***deny**:* negazione o rifiuto.
- ***ask_functions**:* richiesta di informazioni circa le capacità del sistema.
- ***ask_count**:* domanda riguardante il numero di persone presenti all'interno del centro commerciale o in un suo negozio specifico. Può contenere eventuali attributi di interesse, quali il genere, il colore della parte superiore, il colore della parte inferiore, la presenza o assenza di cappello, la presenza o assenza di borsa, la durata della permanenza.
- ***ask_location**:* intenzione di localizzare una persona all'interno del centro commerciale. Può contenere eventuali attributi di interesse, quali il genere, il colore della parte superiore, il colore della parte inferiore, la presenza o assenza di cappello, la presenza o assenza di borsa.
- ***inform**:* intenzione di fornire informazioni utili al completamento del task di ricerca o di localizzazione, come la specificazione del genere, del colore della parte superiore, del colore della parte inferiore, ecc.
- ***unknown**:* rappresenta la risposta dell'utente quando questi non ha una preferenza specifica circa una caratteristica atta a filtrare le persone da conteggiare oppure quando non ricorda/conosce un attributo utile a localizzare una persona.
- ***thanks**:* espressione di gratitudine o ringraziamento.

Gli intenti *ask_count* ed *ask_location* sono fondamentali affinché il chatbot possa stabilire quale sia lo specifico task da svolgere (conteggio o localizzazione). *inform* ed *unknown* sono stati introdotti per gestire opportunamente le risposte dell'utente all'interno dei forms *count_form* e *location_form*, che in seguito saranno descritti dettagliatamente. *greet* e *goodbye* consentono al chatbot di stabilire quando l'utente intende iniziare la conversazione e quando intende terminarla, così da rispondergli in modo appropriato. *affirm* e *deny* sono utilissimi per gestire semplici affermazioni o negazioni, ampiamente sfruttate dalla parte di dialogue management per stabilire come portare avanti la conversazione. L'intento *ask_functions* consente al chatbot di identificare domande relative alle sue capacità, permettendo così di informare l'utente sulle azioni specifiche che può effettivamente svolgere. Non è garantito che l'utente sia già a conoscenza delle funzionalità del chatbot, pertanto è probabile che ponga domande per ottenere maggiori dettagli a riguardo. L'intento *thanks* permette di rilevare eventuali espressioni di gratitudine dell'utente, molto probabili al termine dello svolgimento del task di conteggio o di localizzazione da parte del sistema.
Si osservi che, al fine di gestire opportunamente messaggi utente con una bassa NLU confidence, si è deciso di introdurre l'ulteriore intento *nlu_fallback*. Per ulteriori dettagli in merito si rimanda alle sezioni riguardanti pipeline e regole.

Le entità scelte sono le seguenti:

- ***not**:* un termine che indica una negazione, come *not, don't, without*, ecc.
- ***more**:* un termine che indica il superamento di una durata temporale, come *more, higher, exceeds*, ecc.
- ***less**:* un termine che indica fatto di essere al di sotto di un limite temporale, come *less, shorter, below*, ecc.
- ***upper_body**:* un indumento per la parte superiore del corpo, come *jacket, t-shirt, hoodies*.

- ***lower_body**:* un indumento per la parte inferiore del corpo, come *skirt, leggins, trousers*, ecc.
- ***colour**:* un colore tra *black, blue, brown, gray, green, orange, pink, purple, red, white, yellow*, non necessariamente in lowercase.
- ***mall**:* un termine che indica un centro commerciale, come *mall*, *shopping center*, *commercial plaza*, ecc.
- ***people**:* un termine che indica uno o più individui senza specificazione del genere, come *people, children, customer*, ecc.
- ***hat**:* un termine che indica un cappello, come *hat, berret, cap*, ecc.
- ***bag**:* un termine che indica una borsa, come *bag, schoolbag, purse*, ecc.
- ***male**:* un termine che indica uno o più individui di genere maschile, come *boys, dudes, male*, ecc.
- ***female**:* un termine che indica uno o più individui di genere femminile, come *girls, female, daughters*, ecc.
- ***shop**:* un negozio tra *starbucks* e *walmart*, non necessariamente in lowercase.
- ***duration**:* un'espressione di durata, come *two minutes*, *5 hours*, *half an hour*, ecc.

Tutte le entità sono sostanzialmente atte al perseguimento dei task di conteggio e localizzazione. In particolare, *mall* e *shop* sono utili a determinare l'area di interesse. *people*, *male* e *female* concorrono a stabilire se il task debba considerare il genere e, in tal caso, quale. *colour*, *upper_body* e *lower_body* consentono di determinare quali debbano essere il colore della parte superiore e della parte inferiore dei soggetti di interesse. *hat*, *bag* e *not* permettono di comprendere se il task debba concentrarsi su persone con o senza cappello e con o senza borsa. *more*, *less* e *duration* consentono di individuare eventuali constraints relativi alla durata della permanenza nell'area di interesse, i quali sono particolarmente utili per il task di conteggio. Si osservi che l'entità *not* viene adoperata anche per rilevare l'eventuale interesse dell'utente a contare/localizzare soggetti il cui colore della parte superiore/inferiore è diverso da uno specifico colore.

Tutte le entità elencate concorrono alla valorizzazione di alcuni slots, che vengono successivamente sfruttati per interrogare il file *database.json* ed ottenere le informazioni desiderate.

L'entità *duration* viene rilevata sfruttando il componente *DucklingEntityExtractor*. Le entità *colour* e *shop*, poiché si ipotizza che possano assumere valori appartenenti ad un set noto, specificati all'interno di apposite lookup tables, vengono estratte con l'ausilio del componente *RegexEntityExtractor*. Pertanto, all'interno del dataset le entità *duration*, *colour* e *shop* non sono etichettate. Tutte le altre entità, poiché estratte sfruttando il componente *DietClassifier*, sono invece accuratamente etichettate. Dettagli circa i componenti sopra citati saranno forniti nella sezione riguardante la pipeline.

Nelle fasi iniziali della progettazione si è deciso di realizzare il chatbot in lingua inglese. La scelta è motivata dal fatto che le reti più performanti che forniscono ***pre-trained word embeddings*** sono addestrate su dataset in lingua inglese. L'utilizzo di pre-trained word embeddings è fondamentale per ottenere elevate accuracy nei task di classificazione degli intenti e riconoscimento delle entità, anche con pochi dati di addestramento, come nel nostro caso.

Il dataset su cui è allenata la versione finale del chatbot è stato ottenuto mediante l'aggiunta graduale di nuovi esempi e una serie di miglioramenti progressivi atti ad ottenere ottime performance per quanto riguarda la classificazione degli intenti e il riconoscimento delle entità. All'interno del dataset sono stati inseriti anche alcuni esempi che simulano errori di battitura dell'utente o di trascrizione da parte del modulo di speech to text, cosicché il chatbot possa essere robusto anche rispetto ad essi. Tre dei quattro componenti del team si sono occupati prevalentemente della scrittura o della raccolta degli esempi, mentre il quarto componente ha assunto principalmente il ruolo di tester. In tal modo, ci si è assicurati che non venissero introdotti bias significativi nella fase di test e che il chatbot fosse effettivamente in grado di generalizzare rispetto ad intenti ed entità. Inoltre, si è deciso di far testare il chatbot anche da parte di persone esterne al team, così da raccogliere ulteriori feedback, i quali sono stati particolarmente utili non solo per l'aggiunta di nuovi esempi, ma anche per l'introduzione di intenti inizialmente non considerati, come *ask_functions* e *thanks*, e di regole importanti per rendere l'interazione più naturale. Si osservi che, soprattutto nelle fasi iniziali, ci si è affidati al *Wizard of Oz* approach, simulando conversazioni human-bot tra i componenti del gruppo e talvolta coinvolgendo anche soggetti esterni al team. Così facendo, è stato molto più semplice individuare gli intenti potenzialmente più frequenti e le entità realmente significative.

4.2 Slots

Gli slots rappresentano la memoria di un bot. Essi fungono da contenitori key-value e possono essere usati per memorizzare informazioni significative fornite dall'utente. Di seguito si descrivono gli slots adoperati nel sistema e le procedure attuate per estrarli dagli user inputs all'esterno dei forms.

4.2.1 Definizione degli slots

L'individuazione degli slots è stata una fase cruciale nella progettazione del nostro chatbot. Essi, di fatto, rappresentano il punto di convergenza tra la parte di comprensione del linguaggio naturale (NLU) e le *custom actions* finalizzate alla ricerca all'interno del file *database.json*. In pratica, gli slots contengono informazioni fondamentali che agiscono come vincoli di ricerca dei soggetti rilevanti per i compiti di conteggio e localizzazione.

Gli slots sono definiti come segue:

- ***upperColour**:* rappresenta il colore della parte superiore dei soggetti di interesse; è di tipo *text* poiché il colore che vi viene inserito è semplicemente una stringa.
- ***lowerColour**:* rappresenta il colore della parte inferiore dei soggetti di interesse; è di tipo *text* poiché il colore che vi viene inserito è semplicemente una stringa.
- ***kindOfPeople**:* rappresenta il genere dei soggetti di interesse; è di tipo *categorical* e può assumere i valori M, F, A, che stanno rispettivamente per Maschio, Femmina, Altro (genere non specificato).
- ***hatSlot**:* specifica se si è interessati a soggetti con il cappello, senza, oppure a entrambi; è di tipo *categorical* e può assumere i valori *with*, *without* e *both*.
- ***bagSlot**:* specifica se si è interessati a soggetti con la borsa, senza, oppure a entrambi; è di tipo *categorical* e può assumere i valori *with*, *without* e *both*.
- ***place**:* rappresenta il luogo di interesse; è di tipo *categorical* e può assumere i valori *mall*, *walmart*, *starbucks*.
- ***not_upperSlot**:* è di tipo *bool* e determina come interpretare il contenuto dello slot *upperColour*. In particolare, se *not_upperSlot* assume valore *False*, allora si è interessati esclusivamente a soggetti il cui colore della parte superiore corrisponde al contenuto di *upperColour*; invece, se *not_upperSlot* assume valore *True*, allora si è interessati a tutti i soggetti il cui colore della parte superiore è diverso dal contenuto di *upperColour*.
- ***not_lowerSlot**:* è di tipo *bool* e determina come interpretare il contenuto dello slot *lowerColour*. In particolare, se *not_lowerSlot* assume valore *False*, allora si è interessati esclusivamente a soggetti il cui colore della parte inferiore corrisponde al contenuto di *lowerColour*; invece, se *not_lowerSlot* assume valore *True*, allora si è interessati a tutti i soggetti il cui colore della parte inferiore è diverso dal contenuto di *lowerColour*.
- ***duration**:* rappresenta una durata temporale di interesse; è di tipo *any* al fine di svincolarsi dal tipo di dato normalizzato contenuto nel json restituito da Duckling.
- ***compareSlot**:* specifica il tipo di confronto da effettuare con il contenuto dello slot *duration*; è di tipo *categorical* e può assumere i valori *more*, *less*, *equal*.

Si è scelto di estrarre tutti gli slots in modo *custom*, in quanto tale tipologia di *slot mapping* assicura grande flessibilità. Per via di una chiara differenza tra le tipologie di user inputs attesi fuori dai forms e al loro interno, si è deciso di differenziare l'estrazione degli slots in base al fatto di trovarsi all'esterno dei forms oppure all'interno di essi. In quest'ultimo caso si è effettuata un'ulteriore distinzione in base allo specifico form, *location_form* o *count_form*.

4.2.2 Estrazione degli slots fuori dai forms

Per l'estrazione custom degli slots al di fuori dei forms si è sfruttata una *helper class* messa a disposizione da Rasa che prende il nome di *ValidationAction*: è stato sufficiente estenderla ed implementare tanti metodi estrattori quanti gli slot da estrarre. Ciascun metodo, prima di eseguire la propria logica, si assicura che nessuno dei due forms sia attivo. Nel caso degli slots *upperColour, lowerColour, kindOfPeople, hatSlot, bagSlot, not_upperSlot e not_lowerSlot*, i metodi estrattori realizzano la propria logica solo se l'intento predetto per l'ultimo messaggio utente è *ask_count* oppure *ask_location*. Nel caso degli slots *place, duration e compareSlot*, invece, i metodi estrattori realizzano la propria logica solo se l'intento predetto per l'ultimo messaggio utente è *ask_count*, in quanto tali slots non si ritengono di interesse per il task di localizzazione.

L'estrattore di *upperColour* individua un *upper_body* e, se rileva un *colour* nelle sue immediate vicinanze, lo estrae. L'estrattore di *not_upperSlot* verifica se nelle vicinanze di tale *upper_body* vi sia o meno un *not*: nel primo caso inserisce nello slot *True*, nel secondo *False*. Qualora all'interno dello user message ci fosse più di un *upper_body*, allora verrebbe considerato per semplicità solo l'ultimo. L'estrattore di *lowerColour* e l'estrattore di *not_lowerSlot* implementano una logica simile.
L'estrattore di *hatSlot* individua un *hat* e, se non rileva alcun *not* nelle sue immediate vicinanze, inserisce nello slot *with*, altrimenti *without*. Qualora all'interno dello user message ci fosse più di un *hat*, allora verrebbe considerato per semplicità solo l'ultimo. L'estrattore di *bagSlot* implementa una logica simile.
L'estrattore di *kindOfPeople* inserisce nello slot *M* se nello user message vengono rilevati più *male* che *female*, vi

inserisce *F* nel caso opposto, lo lascia vuoto se *male* e *female* sono in egual numero. L'estrattore di *place*, se c'è un solo *shop* e nessun *mall*, inserisce lo *shop* nello slot; se c'è un solo *mall* e nessuno *shop*, inserisce *mall* nello slot; altrimenti lo lascia vuoto. L'estrattore di *duration* sfrutta l'output di *Duckling*, inserendo nello slot l'espressione di durata temporale rilevata convertita in secondi. L'estrattore di *compareSlot* individua un'espressione temporale di durata e, in base alla presenza nelle sue immediate vicinanze di un *more*, di un *less* o di nessuno dei due, inserisce nello slot rispettivamente *more*, *less* o *equal*.

Si osservi che ogni volta che un messaggio dell'utente viene predetto come *ask_count* o *ask_location*, tutti gli slots che non vengono aggiornati da tale messaggio utente vengono resettati. In tal modo ci si assicura che un task di ricerca/localizzazione precedente non condizioni erroneamente il task di ricerca/localizzazione corrente.

Per la descrizione dell'estrazione degli slots all'interno dei forms si rimanda alla sezione successiva.

4.3 Forms

I forms costituiscono un *conversation pattern* molto comune per la raccolta di slots. Essi, infatti, consentono di raccogliere informazioni utili attraverso domande mirate. Questa sezione descrive i forms presenti nel progetto e come vengono estratti gli slots quando uno di essi risulta attivo.

4.3.1 Definizione dei forms

Al fine di completare e semplificare la raccolta di informazioni utili allo svolgimento dei task di conteggio e localizzazione, sono stati introdotti i seguenti forms:

- ***count_form***: si attiva quando viene rilevata l'intenzione da parte dell'utente di svolgere un conteggio dettagliato. Tale form ha l'obiettivo di riempire gli slots *kindOfPeople, upperColour, lowerColour, hatSlot, bagSlot, not_lowerSlot, not_upperSlot* e *place*, cosicché il task di conteggio possa essere svolto considerando un numero rilevante di constraints.
- ***location_form***: si attiva in seguito alla rilevazione dell'intento *ask_location* e ha l'obiettivo di riempire gli slots *kindOfPeople, upperColour, lowerColour, hatSlot, bagSlot, not_lowerSlot* e *not_upperSlot*, cosicché il task di localizzazione possa essere svolto correttamente.

Si osservi che il *count_form* non si attiva immediatamente dopo la rilevazione di un *ask_count*. In seguito ad uno user input classificato come *ask_count*, il chatbot svolge il task di conteggio sfruttando le sole informazioni contenute nel messaggio, per poi chiedere all'utente se desideri effettuare un conteggio più dettagliato. Se la risposta è affermativa, allora il *count_form* si attiva. Questa scelta progettuale deriva dal fatto che è molto probabile che l'utente si aspetti di ricevere una risposta veloce ad una richiesta di conteggio. In sostanza, il *count_form* si attiva solo se l'utente manifesta l'intenzione di rispondere ad alcune domande atte a dettagliare la sua richiesta di conteggio.

4.3.2 Estrazione degli slots all'interno dei forms

Per l'estrazione custom degli slots all'interno dei forms si è estesa una *helper class* messa a disposizione da Rasa che prende il nome di *FormValidationAction*. Ciascun estrattore esegue la propria logica esclusivamente se l'ultima domanda del form ha l'obiettivo di riempire lo slot corrispondente.

L'estrattore di *upperColour*, se rileva l'intento *unknown*, inserisce nello slot la parola chiave *free*; se invece rileva l'intento *inform*, cerca all'interno dello user message un *colour* e stabilisce se esso sia significativo o meno per lo slot. Nel primo caso ve lo inserisce e valuta anche se nelle sue immediate vicinanze sia presente o meno un *not*, quindi valorizza opportunamente lo slot *not_upperSlot*. L'estrattore di *lowerColour* implementa una logica simile.
L'estrattore di *kindOfPeople*, se rileva l'intento *unknown*, inserisce nello slot *A*; se invece rileva l'intento *inform*, cerca nello user message le entità *people, male* e *female* e, in base al loro numero, decide se inserire all'interno dello slot *M*, *F* o *A*. L'estrattore di *hatSlot*, se rileva l'intento *unknown*, inserisce nello slot *both*; se invece rileva l'intento *inform*, cerca all'interno dello user message un *hat* e, in base alla sua presenza o meno e all'eventuale presenza di un *not* nelle sue vicinanze, inserisce nello slot *with* o *without*. L'estrattore di *bagSlot* implementa una logica simile. L'estrattore di *place*, se rileva l'intento *unknown*, inserisce nello slot *mall* poiché assume che la ricerca riguardi l'intero centro commerciale; se invece rileva l'intento *inform*, cerca all'interno dello user message le entità *mall* e *shop* e, in base al loro numero e al loro valore, decide se inserire all'interno dello slot *mall*, *walmart* o *starbucks*.

Si osservi che l'estrazione degli slots avviene in modo leggermente diverso in base a quale form risulti attivo. Nel caso di *location_form*, la logica descritta in seguito alla rilevazione di un *inform* viene eseguita anche nel caso in cui venisse

rilevato un *ask_location*. Infatti, è probabile che l'utente risponda alle domande del form con espressioni del tipo *"I'm looking for a girl", "I'm searching a person with a bag*", ecc. Esse non verrebbero interpretate come *inform* bensì come *ask_location*, tuttavia è importante che le informazioni in esse contenute vengano usate per valorizzare opportunamente gli slots. Nel caso di *count_form*, la logica descritta in seguito alla rilevazione di un *inform* viene eseguita anche nel caso in cui venisse rilevato un *ask_count* oppure un *ask_location*. Infatti, è probabile che l'utente risponda alle domande del form con espressioni del tipo *"I want to count the ones with blue shirt", "I'm looking for girls",* ecc. Alcune di esse (come la prima) verrebbero interpretate come *ask_count*, mentre altre (come la seconda) come *ask_location*, tuttavia è importante che le informazioni in esse contenute vengano usate per valorizzare opportunamente gli slots.

4.4 Pipeline e Policies

Questa sezione è dedicata alle scelte progettuali riguardanti NLU pipeline e policies.

4.4.1 Pipeline

In Rasa, la NLU pipeline rappresenta la sequenza di componenti che processano l'input dell'utente con l'obiettivo di classificare l'intento e individuare le entità. La pipeline adoperata, frutto di molteplici prove atte ad individuare i componenti più adatti, è riportata di seguito:

```yaml
pipeline:
  - name: SpacyNLP
    model: "en_core_web_md"
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: "char_wb"
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    constrain_similarities: True
    epochs: 100
  - name: FallbackClassifier
    threshold: 0.4
  - name: "DucklingEntityExtractor"
    url: "http://localhost:8000"
    dimensions: ["duration"]
    timezone: "Europe/Rome"
  - name: RegexEntityExtractor
    case_sensitive: False
    use_lookup_tables: True
```

Come già accennato, considerando la limitata disponibilità di dati di addestramento, l'impiego di pre-trained word embeddings è fondamentale per conseguire un'elevata accuracy, al contempo assicurando tempi di training ridotti. La scelta è ulteriormente supportata dal contesto operativo del chatbot: un centro commerciale non è di certo un ambiente caratterizzato da un lessico domain specific, pertanto l'adozione di un modello di linguaggio pre-addestrato su un'elevata quantità di dati generici in lingua inglese si configura come un'opzione particolarmente efficace. Nello specifico, dopo alcune sperimentazioni con i modelli *GPT-2*, *en_core_web_sm*, *en_core_web_lg* ed *en_core_web_md*, è stato selezionato quest'ultimo per via delle sue dimensioni ridotte e delle ottime performance raggiunte. Si osservi che il modello *en_core_web_md* fa capo alla libreria open-source *spacy*.

La scelta appena discussa ha comportato l'introduzione nella pipeline anche del tokenizer *SpacyTokenizer* e del featurizer *SpacyFeaturizer*. Gli ulteriori featurizers inseriti sono: il *RegexFeaturizer*, il quale estrae le regex features necessarie a *RegexEntityExtractor* per individuare i *colour* e gli *shop* specificati nelle lookup tables; il *LexicalSyntacticFeaturizer*, che crea features basate sul lessico e sulla sintassi, particolarmente utili per supportare l'estrazione delle entità; il *CountVectorsFeaturizer*, che crea bag-of-words features funzionali soprattutto alla classificazione degli intenti. I modelli deputati all'estrazione delle entità di interesse sono: *RegexEntityExtractor*, necessario ad estrarre *colour* e *shop*; *DucklingEntityExtractor*, necessario ad estrarre *duration*; *DIETClassifier*, il miglior modello messo a disposizione da Rasa, si occupa dell'estrazione di tutte le altre entità. I modelli deputati alla classificazione degli intenti sono: *DIETClassifier*, che di fatto è un'architettura multi-task capace di realizzare sia intent

classification sia entity extraction con performance allo stato dell'arte; *FallbackClassifier*, introdotto per gestire opportunamente messaggi utente con una bassa NLU confidence, e la cui threshold è stata impostata a 0.4 dopo vari esperimenti. In sostanza, dato un input utente, se nessuno degli intenti viene predetto con una confidence superiore a 0.4, allora l'intento predetto è *nlu_fallback*.

Si osservi che, per quanto concerne il *LexicalSyntacticFeaturizer* ed il *CountVectorsFeaturizer*, la configurazione scelta è quella suggerita dalla documentazione di Rasa; per quanto riguarda *DIETClassifier*, l'impostazione `constrain_similarities: True` aiuta il modello a generalizzare meglio, mentre il numero di epoche posto uguale a 100 consente di raggiungere ottime performance in tempi bevi, senza andare in overfitting; infine, per quanto riguarda *RegexEntityExtractor*, è stata adottata l'impostazione `case_sensitive: False` poiché l'estrazione delle entità *colour* e *shop* deve essere case insensitive. Infatti, le lookup tables adoperate contengono solo esempi in lower case, mentre gli user messages potrebbero contenere nomi di colori o negozi con diverse combinazioni di maiuscole e minuscole.

4.4.2 Policies

In Rasa, le policies definiscono il **dialogue management**, ovvero il comportamento del chatbot durante una conversazione. Nello specifico, ad ogni step della conversazione, le policies concorrono a decidere quale azione il chatbot debba intraprendere. Le policies adoperate nel progetto sono riportate di seguito:

```
policies:
  - name: RulePolicy
    restrict_rules: False
    core_fallback_threshold: 0.4
    core_fallback_action_name: "action_default_fallback"
    enable_fallback_prediction: True
  - name: TEDPolicy
    max_history: 5
    epochs: 100
    constrain_similarities: true
```

La *RulePolicy* effettua predizioni basandosi sulle regole presenti nei dati di addestramento. L'opzione `restrict_rules: False` consente di definire regole con più di un singolo user turn, ed è stata impostata per poter definire la regola *ask confirmation after detecting ask_count, if confirmation is good, give answer, then ask for possible other details*. In particolare, tale regola permette di stabilire che, se il task di conteggio viene completato senza l'aggiunta di dettagli mediante form, allora il chatbot debba chiedere all'utente se intende effettuare un conteggio più dettagliato.
Le ulteriori opzioni che caratterizzano *RulePolicy* consentono di gestire casi in cui l'azione successiva non venga predetta da alcuna policy con una confidence superiore ad una certa threshold, che, nel caso specifico, è stata posta uguale a 0.4 dopo vari esperimenti. In uno scenario del genere, il chatbot risponde all'utente con la *utter_default*, che è *"I'm sorry, I didn't understand you. Repeat, please"*. Per ulteriori dettagli sulle regole si rimanda alla sezione successiva.
La *TEDPolicy* consiste in un'architettura multi-task per next action prediction ed entity recognition. Essa è in grado di prevedere l'azione successiva sulla base degli esempi contenuti nel training set, degli intenti e delle entità degli user messages precedenti, delle azioni e delle utterances precedenti. La *TEDPolicy* è particolarmente utile per gestire tutti gli scenari non contemplati dalle regole, pertanto offre significativi vantaggi in termini di adattabilità e gestione di input più complessi. Si osservi che la configurazione scelta per *TEDPolicy* è quella suggerita dalla documentazione di Rasa.

4.5 Regole

Le regole sono un tipo di training data utilizzato per addestrare il dialogue management del chatbot. Nello specifico, le regole descrivono brevi pezzi di conversazione che dovrebbero seguire sempre lo stesso percorso. Dato che le specifiche prevedevano che il chatbot non dovesse gestire conversazioni particolarmente complesse, si è deciso di inserire all'interno dei dati di addestramento solamente regole, quindi di non adoperare nessuna storia.

La maggior parte delle regole ha lo scopo di gestire le diverse fasi dei task di conteggio e di localizzazione, come: l'avvio e la conclusione dei rispettivi forms, *count_form* e *location_form*, la conferma dell'utente della corretta comprensione dei constraints da parte del chatbot e, nel caso di mancata conferma, la sollecitazione rivolta all'utente affinché riformuli la propria richiesta.
Si osservi che, in caso di rilevazione di un *ask_count*, il chatbot, dopo aver chiesto ed ottenuto conferma degli attributi compresi, fornisce direttamente la risposta, quindi chiede all'utente se sia sua intenzione imporre ulteriori constraints per effettuare un nuovo conteggio. Se l'utente risponde affermativamente, allora il *count_form* si attiva. Questa scelta progettuale fa sì che l'utente possa ricevere velocemente una risposta alla propria richiesta di conteggio, senza dover

necessariamente rispondere alle domande di un form. Successivamente, è l'utente medesimo a decidere se attivare il form al fine di dettagliare ulteriormente la propria richiesta. Nel caso di rilevazione di un *ask_location*, invece, l'attivazione del *location_form* è praticamente immediata. Esso, infatti, ha il fondamentale obiettivo di aiutare l'utente a definire velocemente dei constraints che facilitino la localizzazione della persona che sta cercando.

Le altre regole hanno principalmente lo scopo di rendere la conversazione più piacevole e realistica. Ad esempio, l'assistente è in grado di rispondere opportunamente ad un'espressione di gratitudine, spiegare quali siano le proprie capacità qualora l'utente dovesse fare una domanda in proposito, salutare quando è opportuno farlo, ammettere di non aver capito quando il messaggio dell'utente risulta ambiguo, ecc. Tutte le regole sono contenute all'interno del file *rules.yml*.

4.6 Custom actions

In Rasa sono state implementate le custom actions: *ActionCount*, *ActionLocation*, *ActionConfirmationCount* e *ActionConfirmationLocation*. Di seguito, è fornita una loro descrizione.

4.6.1 *ActionCount* e *ActionLocation*

ActionCount ed ***ActionLocation*** sono azioni progettate per eseguire ricerche sul file *database.json* tramite le funzioni definite nel modulo *readerJson.py*, rispettando i vincoli di ricerca memorizzati negli slots. Entrambe le classi estendono la classe nativa di Rasa nota come *Action.*

Dopo aver ottenuto i risultati della ricerca nel database, mentre nella classe *ActionCount* si procede con la creazione di una *utterance* la cui struttura dipende dal numero di persone individuate, nella classe *ActionLocation* si genera una *utterance* che tiene conto sia del numero di persone coinvolte sia delle informazioni relative alla localizzazione di una persona all'interno del centro commerciale. In tal caso, infatti, per ogni individuo identificato, vengono forniti dettagli come il numero di transizioni effettuate e la quantità di tempo trascorsa nei due possibili negozi: *walmar*t e *starbucks*. Si osservi che nel caso in cui il task di conteggio riguardi mall e non il singolo negozio allora la ricerca si basa sulla somma dei tempi trascorsi dei 2 possibili negozi che compongono il centro commerciale.

È da presupporre che il file *database.json* sia soggetto ad aggiornamenti provenienti da un sistema di analisi video non appartenente al dominio di questo progetto. Dunque, il **modulo readerJson.py** ha un ruolo fondamentale nell'interazione con questi dati.

La ricerca nel database, implementata in *readerJson.py*, può essere suddivisa nelle seguenti **tre fasi**:

- Durante la **fase di creazione dei vincoli**, si procede con l'estrazione delle informazioni dagli slots. Vengono escluse le informazioni che non rappresentano veri vincoli di ricerca. Le informazioni considerate non rilevanti comprendono gli slots non valorizzati, cioè *None*, e quelli con valori come *free* o *both*. In seguito a questa selezione, i dati vengono organizzati in un dizionario Python, le cui chiavi corrispondono ai nomi degli slots dei vincoli di ricerca. I valori associati a ciascuna di queste chiavi rispettano la coerenza con i dati presenti nel file JSON.
- Durante la **fase di trasformazione dei vincoli**, l'obiettivo è rielaborare il dizionario precedentemente creato in modo che rispetti le chiavi richieste dal file JSON, facendo un mapping tra questi ultimi e gli slots. In aggiunta, si procede rispettivamente con la "traduzione" dei comparativi temporali (l*ess, more, equal*) in espressioni logiche e con la mappatura degli shop *walmart* e *starbucks* nei valori *roi1* e *roi2*.
- La **fase di ricerca su file** si basa sulla **teoria degli insiemi**. Essendo ogni persona identificata univocamente da un *id*, la procedura prevede una ricerca nel file tenendo conto di un vincolo alla volta. Ad ogni iterazione, le persone identificate vengono aggiunte all'insieme corrispondente al vincolo considerato. Il risultato finale consiste nell'intersezione di tutti gli insiemi, cosicché al termine possa essere restituita la lista delle persone il cui *id* è stato selezionato in tutte le ricerche soggette a vincoli.

4.6.2 *ActionConfirmationCount* e *ActionConfirmationLocation*

Le custom actions *ActionConfirmationCount* e *ActionConfirmationLocation* hanno semplicemente l'obiettivo di restituire all'utente rispettivamente una richiesta di conferma del task di conteggio e del task di localizzazione, sulla base degli slots estratti a partire dagli user messages. Pertanto, i metodi *run* delle classi *ActionConfirmationCount* e *ActionConfirmationLocation* restituiscono rispettivamente stringhe del tipo:

"So, if I understand well, you want to count males with upper clothes of any colour, with red lower clothes, without a hat, who have been in walmart for more than 20 minutes. Is it right?"

"So, if I understand well, you want to localize a person with a hat, without a bag, with green upper clothes, with blue lower clothes. Is it right?"

La richiesta di conferma è essenziale per garantire all'utente la corretta comprensione da parte del robot di tutti i vincoli da lui specificati.

5. WP4: Integrazione del modulo di face detection e del modulo di speech to text

Questo breve capitolo è incentrato sulla descrizione dei moduli di face detection e di speech to text integrati all'interno dell'architettura ROS. In particolare, se ne descrivono le caratteristiche e se ne motiva l'impiego nel progetto.

5.1 Face detection

Si è preferito usare un modulo di face detection invece di un modulo di people detection, in quanto quest'ultimo, senza particolari accorgimenti a valle riguardanti la dimensione dei bounding boxes restituiti, avrebbe rilevato anche individui situati a considerevole distanza dal robot. Ciò avrebbe comportato attivazioni frequenti e inopportune della procedura di engagement.
Il face detector adoperato è quello *deep-learning based* messo a disposizione da **OpenCV**. Esso si basa sul framework *Single Shot Detector (SSD)* ed ha *ResNet* come backbone. Di fatto, la sua architettura è progettata per massimizzare l'efficienza computazionale, mantenendo al contempo un'elevata accuracy. La sua capacità di lavorare in tempo reale lo rende una scelta particolarmente popolare in applicazioni di analisi video, come nel nostro caso. Esso, inoltre, è capace di rilevare anche volti parzialmente occlusi.
Dopo molteplici esperimenti, si è deciso di settare la sua *confidence threshold* uguale a 0.7, poiché è il valore che ha dato maggiori garanzie in condizioni di luminosità non perfette e ad una distanza compresa tra 1 m e 3 m dal robot.

5.2 Speech to text

Il modulo di speech to text ha il compito di trascrivere le frasi pronunciate dall'utente, in modo che queste possano essere somministrate al chatbot. All'interno del progetto, per realizzare questa funzionalità, è stata impiegata la libreria Python **SpeechRecognition**. Essa si distingue per il supporto a diversi motori di riconoscimento vocale e la presenza di numerose opzioni di personalizzazione, come quelle che consentono di impostare e gestire una *energy threshold*. Ciò la rende particolarmente versatile e adatta anche a contesti rumorosi, come un centro commerciale. Nello specifico, si è deciso di utilizzare l'API di riconoscimento vocale di Google, **Google Speech Recognition**, per via della sua elevata accuratezza, della sua velocità (la latenza massima è di circa 2 s) e della sua accessibilità. Infatti, a differenza di Whisper, la cui API è ormai a pagamento, Google Speech Recognition è un servizio completamente gratuito.

6. WP5: Pianificazione dei test

In questo capitolo sono descritti i test che sono stati eseguiti al fine di verificare il corretto funzionamento dei singoli nodi realizzati e la loro corretta integrazione.

6.1 Test Rasa

I primi test hanno riguardato il funzionamento del chatbot sviluppato in Rasa.

Rasa fornisce nativamente delle procedure per la valutazione del modello NLU realizzato. A tale scopo, nel nostro progetto è stata utilizzata la tecnica di *cross-validation*, più specificamente la k-fold cross-validation. Questa procedura prevede la suddivisione in maniera automatica del dataset in k sottoinsiemi (detti 'folds'). Ad ogni iterazione, il modello viene addestrato su k-1 folds e testato su quello restante; questo procedimento si ripete k volte e ogni volta viene usato un fold diverso per il test. Successivamente, i risultati ottenuti ad ogni iterazione vengono aggregati per ottenere una valutazione complessiva delle prestazioni del modello.

Per la fase di test del chatbot è stato eseguito il seguente comando:

```
rasa test nlu --nlu data/nlu --cross-validation --folds 3
```

Abbiamo scelto di adottare un numero limitato di folds, precisamente 3, al fine di suddividere il dataset in blocchi non troppo piccoli. Tale decisione mira a garantire una rappresentazione significativa di ciascun intento nell'ambito della

valutazione del modello. La scelta di un valore più grande, infatti, avrebbe portato ad un numero troppo piccolo di esempi per ciascun intento nel fold di test.

6.2 Test di Integrazione

I test di integrazione mirano ad eseguire un check della corretta integrazione tra i vari nodi ROS realizzati. Si riportano di seguito tutti i test effettuati.

Innanzitutto, è stata verificata la corretta integrazione tra Rasa e ROS semplicemente conversando con il chatbot da terminale, sfruttando i nodi presenti nel pacchetto *rasa_ros*.

Successivamente, è stato eseguito un test per valutare il corretto funzionamento del modulo di face detection. Il nodo coinvolto in questo test è *face_detector*, presente all'interno del package *detector_pkg*. Di seguito si riportano le fasi di tale test:

- lettura e visualizzazione dell'immagine dal topic */in_rgb*;
- applicazione del face detector sull'immagine acquisita e pubblicazione di un evento di rilevamento di un volto sul topic */detection*.

É stato eseguito un test sul *voice_detection_node*, al fine di verificare la corretta calibrazione della *energy_threshold* del microfono all'interno di un generico ambiente, e sul nodo *speech_recognition* per verificare il corretto funzionamento del riconoscimento vocale da parte del servizio cloud Google Speech Recognition.

Successivamente, è stato testato il nodo *text2speech_node* avviando una conversazione con il chatbot tramite Pepper. In questa fase, sono state poste delle richieste di conteggio e localizzazione al chatbot attraverso il microfono esterno.

Infine, è stato testato con successo il funzionamento del tablet di Pepper per la visualizzazione delle pagine web da noi realizzate, mediante il nodo *tablet_manager_node*. In particolare, il robot ha visualizzato correttamente sul proprio tablet le schermate di default, engagement e dialogo.

6.3 Test Finali

I test finali sono finalizzati a verificare l'integrità dell'intera logica di sistema. Per ciascuno di essi è stata fornita una breve descrizione, sono stati poi definiti un obiettivo da soddisfare e la condizione che lo rende superato con successo.

- **Test 1:** Il test assume che il robot non stia effettuando alcuna conversazione e che una persona si collochi davanti al robot. L'obiettivo è verificare che la procedura di engagement funzioni correttamente.
  Il test si considera superato se, in seguito alla rilevazione del volto, il robot pronuncia la frase di engagement e mostra sul tablet la pagina web *engagement.html*.

- **Test 2:** Il test assume che sia in corso una conversazione e la persona non parli per più di 30 secondi. L'obiettivo è verificare che, sotto tali condizioni, il robot concluda la conversazione corrente e si ponga in attesa di un nuovo utente. Il test si considera superato se il robot invia un messaggio di *restart* a Rasa e mostra sul tablet la pagina web *index.html*.

- **Test 3:** Il test assume che sia in corso una conversazione e la persona non compaia più davanti alla camera del robot per più di 5 secondi. L'obiettivo è verificare che, sotto tali condizioni, il robot concluda la conversazione corrente e si ponga in attesa di un nuovo utente.
  Il test si considera superato se il robot invia un messaggio di *restart* a Rasa e mostra sul tablet la pagina web *index.html*.

- **Test 4:** Il test assume che sia in corso una conversazione. L'obiettivo è verificare che sul tablet vengano riportate correttamente le frasi dell'utente.
  Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono trascritte e visualizzate correttamente sulla pagina web *dialogue.html* mostrata sul tablet.

- **Test 5:** Il test prevede uno scenario di conversazione in cui l'utente interagisce con il robot, ponendogli una richiesta di conteggio. Si riporta la sequenza degli eventi del test:

    1. L'utente si avvicina al robot.

2. Il robot ingaggia l'utente.
3. L'utente chiede al robot quali sono le sue capacità o cosa è in grado di fare.
4. Il robot risponde elencando le varie funzionalità che può eseguire.
5. L'utente presenta al robot una richiesta di conteggio.
6. Il robot analizza la richiesta dell'utente, raccogliendo le informazioni presenti nella richiesta e chiedendo se quanto interpretato è corretto.
7. L'utente conferma la correttezza della comprensione del robot.
8. Il robot fornisce la risposta al conteggio e chiede all'utente se desidera aggiungere ulteriori dettagli alla richiesta.
9. L'utente risponde negativamente e ringrazia.
10. Il robot risponde con una frase di cortesia.
11. L'utente si congeda salutando.
12. Il robot risponde con un saluto.

L'obiettivo è verificare che il robot sia capace di comprendere le richieste di conteggio dell'utente e di rispondere a queste in maniera corretta. Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono comprese correttamente e Pepper vi risponde correttamente.

- **Test 6:** Il test prevede che venga eseguita una conversazione in cui l'utente pone una richiesta di conteggio al robot, e che venga avviata l'esecuzione del form corrispondente. Si riporta la sequenza degli eventi del test:

  1. L'utente si avvicina al robot.
  2. Il robot ingaggia l'utente.
  3. L'utente presenta al robot una richiesta di conteggio.
  4. Il robot analizza la richiesta dell'utente, raccogliendo le informazioni presenti in essa e chiedendo se ciò che ha interpretato è corretto.
  5. L'utente risponde positivamente.
  6. Il robot fornisce la risposta al conteggio e chiede all'utente se vuole aggiungere alla sua richiesta ulteriori dettagli.
  7. L'utente risponde positivamente.
  8. Il robot esegue il form di conteggio, raccogliendo le informazioni fornite dall'utente e chiedendo se ciò che ha compreso è corretto.
  9. L'utente risponde positivamente.
  10. Il robot fornisce la risposta derivante dall'esecuzione del form.
  11. L'utente ringrazia.
  12. Il robot risponde con una frase di cortesia.
  13. L'utente si congeda salutando.
  14. Il robot risponde con un saluto.

  L'obiettivo è verificare che il robot sia in grado di comprendere le richieste di conteggio ed eseguire correttamente il form corrispondente. Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono comprese correttamente e Pepper vi risponde correttamente.

- **Test 7:** Il test prevede che venga eseguita una conversazione in cui l'utente pone una richiesta di localizzazione al robot, avviando l'esecuzione del form corrispondente. Si riporta la sequenza degli eventi del test:

  1. L'utente si avvicina al robot.
  2. Il robot ingaggia l'utente.
  3. L'utente effettua una richiesta di localizzazione.
  4. Il robot esegue il form di localizzazione, raccogliendo le informazioni fornite dall'utente e chiedendo se ciò che ha compreso è corretto.
  5. L'utente risponde positivamente.
  6. Il robot fornisce la risposta derivante dall'esecuzione del form.
  7. L'utente ringrazia.
  8. Il robot risponde con una frase di cortesia.

9. L'utente si congeda salutando.
10. Il robot risponde con un saluto.

L'obiettivo è verificare che il robot sia in grado di comprendere le richieste di localizzazione ed eseguire correttamente il form corrispondente. Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono comprese correttamente e Pepper vi risponde correttamente.

- **Test 8:** Il test prevede l'esecuzione di una conversazione durante la quale l'utente esprime una richiesta di conteggio, che viene interpretata in maniera errata dal robot. Si riporta la sequenza di eventi del test:

  1. L'utente si avvicina al robot.
  2. Il robot ingaggia l'utente.
  3. L'utente chiede al robot quali sono le sue capacità o cosa è in grado di fare.
  4. Il robot risponde elencando le varie funzionalità che può eseguire.
  5. L'utente presenta al robot una richiesta di conteggio.
  6. Il robot analizza la richiesta dell'utente, raccogliendo le informazioni presenti in essa e chiedendo se quanto interpretato è corretto.
  7. L'utente risponde negativamente.
  8. Il robot chiede di formulare una nuova richiesta.
  9. L'utente formula una nuova richiesta di conteggio.
  10. Il robot raccoglie le informazioni presenti nella nuova richiesta e chiede all'utente se ciò che ha compreso è corretto.
  11. L'utente risponde positivamente.
  12. Il robot fornisce la risposta e chiede all'utente se vuole aggiungere alla sua richiesta ulteriori dettagli.
  13. L'utente risponde negativamente e ringrazia.
  14. Il robot risponde con una frase di cortesia.
  15. L'utente si congeda salutando.
  16. Il robot risponde con un saluto.

  L'obiettivo è assicurare che la conversazione proceda correttamente, concentrandosi soprattutto sulla capacità del robot di gestire situazioni in cui la risposta fornita all'utente risulta non corretta. Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono comprese correttamente e Pepper vi risponde correttamente.

- **Test 9:** Il test assume che venga eseguita una conversazione durante la quale l'utente esprime una richiesta di localizzazione, che viene interpretata in maniera errata dal robot. In seguito alla domanda del robot di formulare una nuova richiesta, l'utente risponde negativamente. Si riporta la sequenza di eventi del test:

  1. L'utente si avvicina al robot.
  2. Il robot ingaggia l'utente.
  3. L'utente esegue una richiesta di localizzazione.
  4. Il robot esegue il form di localizzazione, raccogliendo le informazioni fornite dall'utente e chiedendo se ciò che ha compreso è corretto.
  5. L'utente risponde negativamente.
  6. Il robot chiede all'utente di formulare una nuova richiesta.
  7. L'utente risponde negativamente.
  8. Il robot risponde con una frase di cortesia
  9. L'utente si congeda salutando.
  10. Il robot risponde con un saluto.

  L'obiettivo è verificare che il robot sia in grado di gestire correttamente il caso in cui la risposta che ha fornito non è corretta e l'utente non intende formulare una nuova richiesta. Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono comprese correttamente e Pepper vi risponde correttamente.

- **Test 10:** Il test assume che venga eseguita una conversazione durante la quale l'utente esprime una richiesta di localizzazione, che viene interpretata in maniera errata dal robot. In seguito alla domanda del robot di formulare una nuova richiesta, l'utente risponde positivamente, senza però formulare alcuna nuova richiesta. Il

robot, quindi, sollecita l'utente a esprimere una nuova richiesta. Si riporta la sequenza di eventi del test:

1. L'utente si avvicina al robot.
2. Il robot ingaggia l'utente.
3. L'utente effettua una richiesta di localizzazione.
4. Il robot esegue il form di localizzazione, raccogliendo le informazioni fornite dall'utente e chiedendo se ciò che ha compreso è corretto.
5. L'utente risponde negativamente.
6. Il robot chiede all'utente di riformulare la sua richiesta.
7. L'utente risponde positivamente, ma senza esprimere una nuova richiesta.
8. Il robot, quindi, sollecita l'utente a formulare una nuova richiesta.
9. L'utente esegue una nuova richiesta di localizzazione.
10. Il robot esegue il form di localizzazione, raccogliendo le informazioni fornite dall'utente e chiedendo se ciò che ha compreso è corretto.
11. L'utente risponde positivamente.
12. Il robot fornisce la risposta derivante dall'esecuzione del form.
13. L'utente ringrazia.
14. Il robot risponde con una frase di cortesia.
15. L'utente si congeda salutando.
16. Il robot risponde con un saluto.

L'obiettivo è verificare che il robot sia in grado di gestire correttamente il caso in cui la risposta che ha fornito non è corretta e l'utente non esprime immediatamente una nuova richiesta, portando il robot a sollecitarlo. Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono comprese correttamente e Pepper vi risponde correttamente.

- **Test 11:** Il test assume che venga eseguita una conversazione durante la quale l'utente esprime una richiesta di localizzazione, che viene interpretata in maniera errata dal robot. In seguito alla domanda del robot di formulare una nuova richiesta, l'utente risponde positivamente, formulando però una richiesta di conteggio. Si riporta la sequenza di eventi del test:

  1. L'utente si avvicina al robot.
  2. Il robot ingaggia l'utente.
  3. L'utente effettua una richiesta di localizzazione.
  4. Il robot esegue il form di localizzazione, raccogliendo le informazioni fornite dall'utente e chiedendo se ciò che ha compreso è corretto.
  5. L'utente risponde negativamente.
  6. Il robot chiede all'utente di riformulare la sua richiesta.
  7. L'utente esegue una richiesta di conteggio.
  8. Il robot analizza la richiesta dell'utente, raccogliendo le informazioni presenti nella richiesta e chiedendo se quanto interpretato è corretto.
  9. L'utente risponde positivamente.
  10. Il robot fornisce la risposta alla richiesta di conteggio e chiede all'utente se vuole aggiungere alla sua richiesta ulteriori dettagli.
  11. L'utente risponde negativamente e ringrazia.
  12. Il robot risponde con una frase di cortesia.
  13. L'utente si congeda salutando.
  14. Il robot risponde con un saluto.

  L'obiettivo è verificare che il robot sia in grado di gestire correttamente il caso in cui l'utente, all'interno della stessa conversazione, effettua due richieste di natura diversa. Il test si considera superato se almeno il 70% delle frasi pronunciate dall'utente vengono comprese correttamente e Pepper vi risponde correttamente.

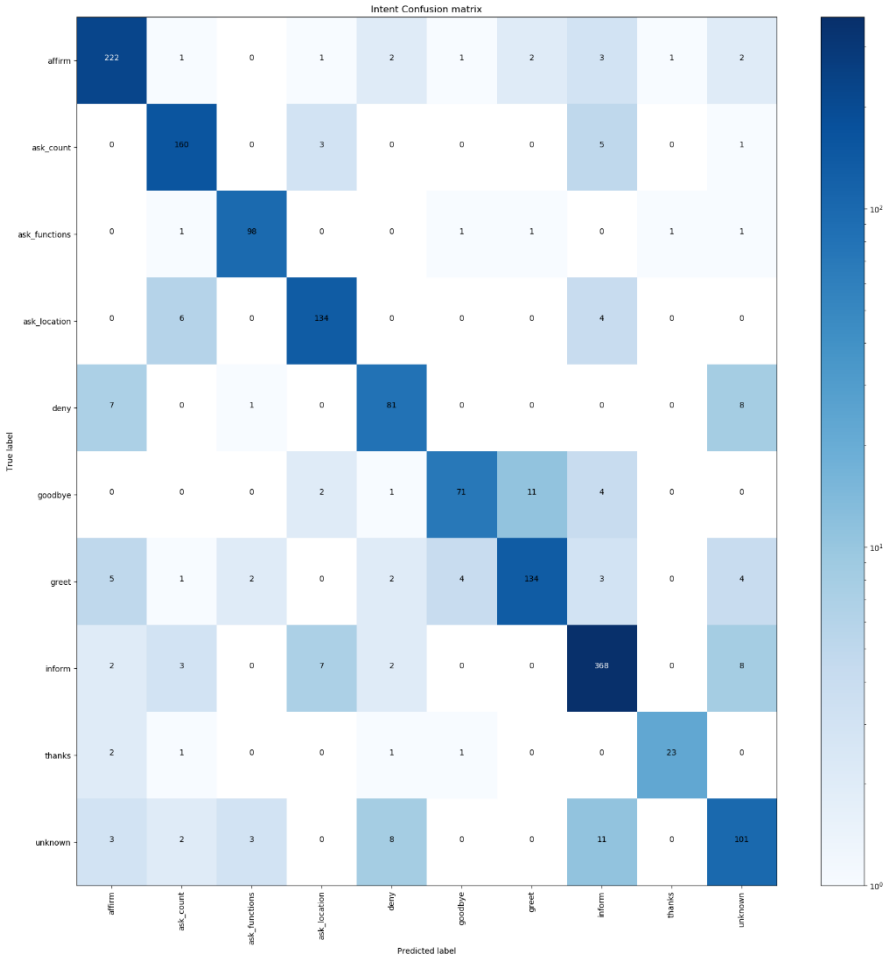
7. WP6: Esecuzione dei test

In questo capitolo sono riportati e commentati i risultati dei test Rasa e dei test finali descritti nel capitolo precedente.

7.1 Risultati dei test Rasa

Al fine di evidenziare le performance ottenute per il chatbot con la pipeline scelta, sono state calcolate la matrice di confusione degli intenti e la matrice di confusione delle entità.

Di seguito si riporta la matrice di confusione degli intenti:



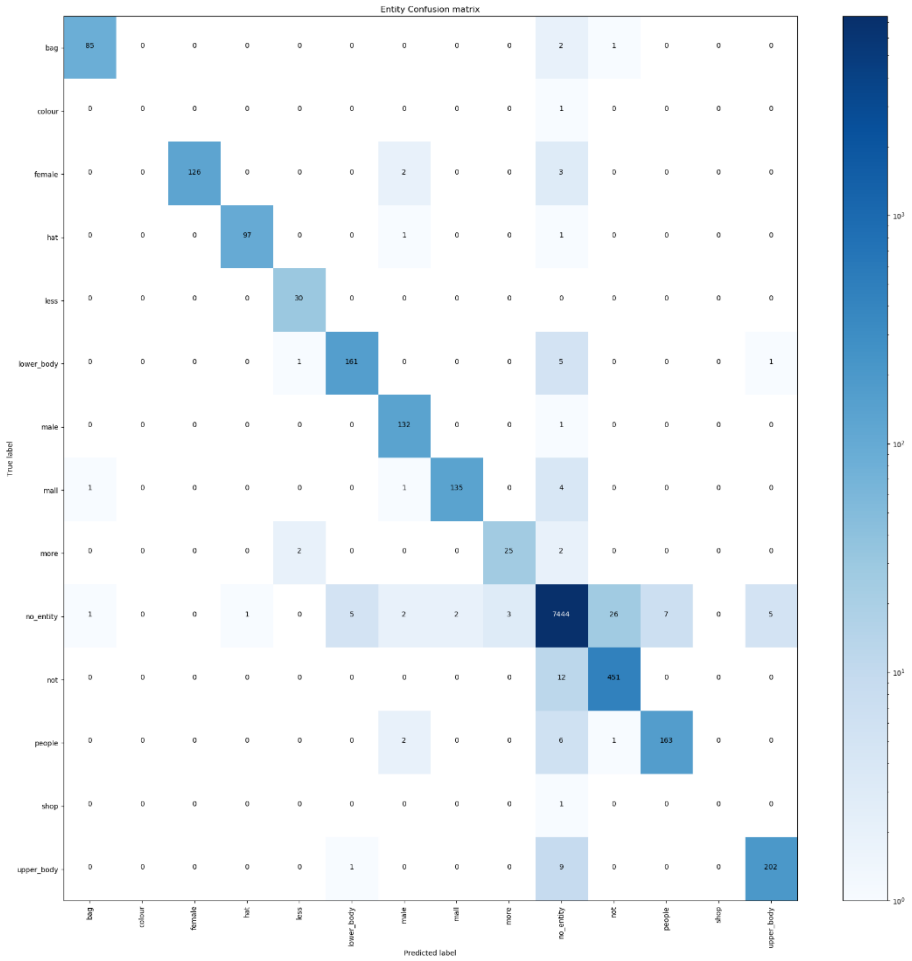
Dai risultati sopra riportati, si può notare che:

- Il sistema incontra alcune difficoltà nella classificazione degli intenti *ask_count* ed *ask_location.* In particolare, in determinate situazioni, tende a confondere tali intenti con l'intento *inform.* Questa difficoltà deriva dalla somiglianza strutturale tra alcuni esempi associati all'intento *inform* e quelli appartenenti agli intenti *ask_count* e *ask_location*, il che porta a riconoscimenti erronei.

  Per capire meglio il problema, si riporta un semplice esempio. La frase *"I'm looking for lads with black jumper"* è associata all'intento *ask_location*. Dal momento che sono presenti alcuni esempi associati all'intento *inform* aventi una forma molto simile alla frase appena riportata (ad esempio *"I'm looking for white upper cloth"*, *"looking for black shirt"*, *"looking for red"*), il sistema classifica erroneamente la frase suddetta come un intento *inform*.

- Inoltre, si osserva che, in determinate circostanze, alcuni intenti, come *deny*, *greet* e *inform*, vengono erroneamente associati all'intento *unknown*. Ciò si verifica poiché l'intento *unknown* incorpora due opzioni: la risposta dell'utente quando non ha una preferenza specifica e la risposta quando non conosce il valore dell'attributo richiesto. Gli esempi correlati a questo intento mancano di una struttura chiaramente definita, il che comporta, in alcuni casi, una non corretta identificazione da parte del sistema.

Segue la matrice di confusione delle entità.



In generale, si può osservare che:

- In diversi casi, alcune entità sono state erroneamente etichettate come *no_entity*. In altre parole, le entità presenti non sono state rilevate. Questa problematica è probabilmente associata alla limitata presenza di esempi per specifiche entità. A causa di ciò, il sistema incontra difficoltà nel riconoscere correttamente tali entità.
- In aggiunta, emergono diverse situazioni in cui si verifica una confusione tra *no_entity* e *not*. Questo fenomeno potrebbe derivare dall'utilizzo di pre-trained word embeddings: la presenza di parole estremamente simili a quelle etichettate come *not* ha indotto il sistema a interpretarle come entità *not*, anche se nella realtà non lo sono.

Lo stesso ragionamento, in realtà, si applica anche ad altre entità: è evidente dalla matrice riportata che si verificano frequentemente casi in cui una *no_entity* viene erroneamente associata ad un'entità (osservando la riga corrispondente a *no_entity*, si nota che una considerevole parte degli errori è concentrata in questa zona).

Si noti che i risultati sopra mostrati fanno riferimento alla versione finale del dataset realizzato. In particolare, sono state affrontate diverse fasi in cui, a partire dal dataset disponibile, sono stati generati i risultati e osservati gli errori risultanti. Sulla base di questi ultimi, si è proseguito con l'introdurre nuovi esempi e l'apportare varie modifiche al fine di ottenere prestazioni progressivamente migliori. Ulteriori informazioni circa i risultati di tali test (es: precision, recall, ecc.) sono contenuti nella cartella *results*.

7.2 Risultati dei test finali

I risultati dei test finali sono riassunti nella tabella seguente:

Test	# successi / # ripetizioni
Test 1	30/30
Test 2	30/30
Test 3	30/30

Test 4	26/30
Test 5	8/10
Test 6	8/10
Test 7	10/10
Test 8	9/10
Test 9	9/10
Test 10	7/10
Test 11	9/10

I risultati indicano che le fasi di engagement e terminazione della conversazione funzionano sempre correttamente. Gli errori compiuti nei test successivi al terzo sono in larga parte imputabili alla *voice detection* ed alla *speech recognition*. Talvolta, il VAD non riesce a rilevare la voce dell'utente; in altre occasioni l'API di Google Speech Recognition non riesce a capire il contenuto dell'audio inviatogli oppure restituisce una trascrizione in parte errata.
Complessivamente, i risultati ottenuti testimoniano il buon lavoro svolto.