Cognitive Robotics

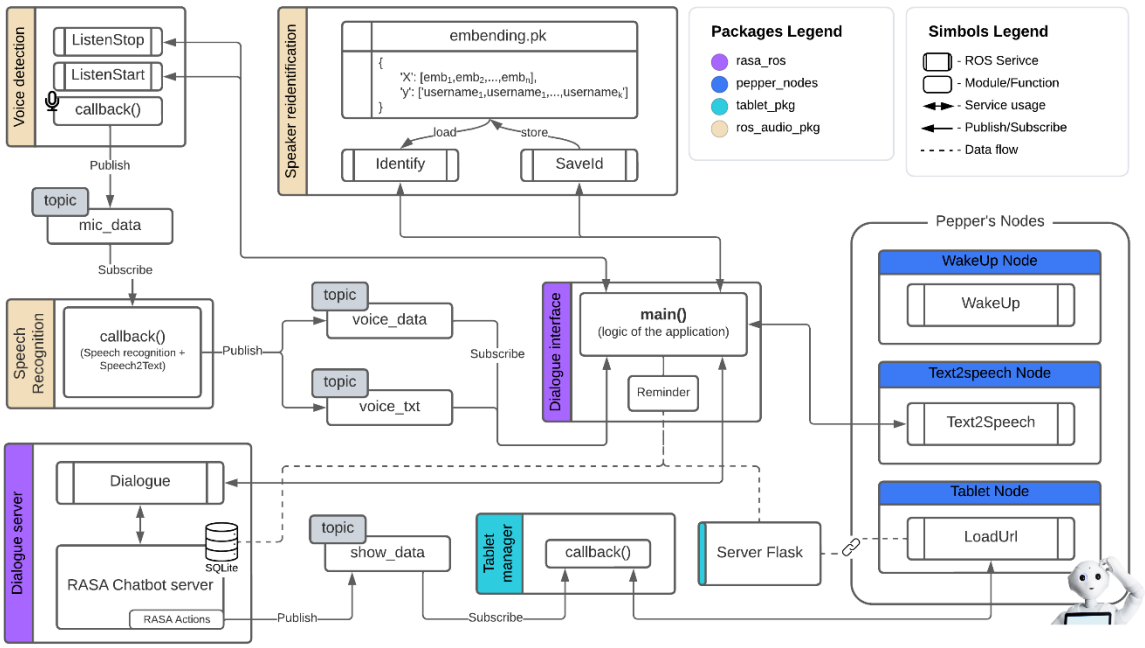
Progetto finale 2022/2023 - To Do List con Pepper – Gruppo 06

Amato Mario	0622701670	m.amato72@studenti.unisa.it
Avitabile Margherita	0622701825	m.avitabile6@studenti.unisa.it
Battipaglia Lucia	0622701758	l.battipaglia6@studenti.unisa.it
Sonnessa Francesco	0622701672	f.sonnessa@studenti.unisa.it

Sommario

1. WP1 & WP2 – Architettura

In questo capitolo descriviamo l'architettura della nostra applicazione introducendo le sue singole parti.



Architettura finale

1.1. Pacchetto rasa_ros

Nel pacchetto *rasa_ros* sono implementati i nodi *Dialogue Server* e *Dialogue Interface*, rispettivamente per l'integrazione del chatbot RASA con l'ambiente ROS e per la gestione dell'intera logica dell'applicazione.

1.1.1. Dialogue Server Node

Segue la spiegazione in dettaglio del servizio *Dialog* e del *RASA Chatbot Server*.

Dialogue service. Implementa un servizio ROS con il quale interagire con il chatbot RASA. Tale servizio prende in ingresso una stringa, come una domanda o una semplice frase, da sottoporre al chatbot e restituisce la risposta del chatbot come stringa.

RASA Chatbot server. Esegue il chatbot vero e proprio implementato per la midterm. Le parti evidenziate nell'immagine dell'architettura sono il database SQLite, in cui vengono salvate le attività e le categorie per ciascun utente, e le RASA Actions.

Le funzionalità richieste per il chatbot sono:

1. Gestione di multiutenti
2. Inserire una nuova attività in una categoria
3. Rimuovere un'attività in una categoria
4. Rimuovere una categoria e tutte le attività in essa contenute
5. Visualizzare le categorie disponibili
6. Visualizzare tutte le attività all'interno di una categoria
7. Modificare un'attività all'interno di una categoria
8. Impostare una deadline e una notifica (*reminder*) per un'attività

1.1.2. Dialogue Interface Node

Contiene il *main()* ovvero il core del sistema. Integra tutti i moduli qui descritti e implementata l'intera logica dell'applicazione. Scandisce le seguenti fasi:

1. **Re-identificazione –** il sistema attende che l'utente inizi la conversazione. Identificata la voce dell'interlocutore, viene chiesto al modulo di re-identificazione di riconoscere l'utente invocando il servizio *Identify*.

   Se il modulo di re-identificazione non riesce a riconoscere l'interlocutore, (risposta *None* da parte di *Identify*) il sistema chiede all'utente di presentarsi fornendo un nome con cui vuole essere identificato. Ottenuto il nome, questo viene passato al modulo di re-identificazione in modo che possa salvare l'associazione $(embedding_i, username_i)$ tramite il servizio *SaveId*. Si procede allo step 2.

   Se invece il modulo di re-identificazione riconosce l'utente, il servizio *Identify* ne restituirà l'identificativo e si può procedere allo step 2.

2. **Sessione di conversazione –** dopo aver identificato l'utente, può iniziare la conversazione vera e propria. In questa fase il sistema acquisisce la voce dell'utente, la traduce in testo e lo invia al chatbot. La stringa di risposta dal chatbot viene riprodotta da Pepper tramite il servizio *Text2Speech*, se è presente, altrimenti dal computer.

   La sessione di conversazione termina dopo il saluto da parte dell'utente (es. "bye"), quindi il sistema si resetta per poter accogliere un nuovo utente e torna allo step 1.

Reminder. Implementa la funzionalità di allert. Dopo l'identificazione dell'utente e durante la sessione di conversazione, controlla se ci sono attività in scadenza per l'utente corrente. Se presenti, restituisce solo quelle attività per le quali è previsto il reminder, al fine di poter notificare l'utente. Quanto prima della deadline ricevere il reminder è un parametro del sistema.

1.2. Pacchetto ros_audio_pkg

In questo pacchetto sono raccolti tutti i nodi e i moduli necessari a processare l'audio acquisito dal microfono.

1.2.1. Voice detection node

Consente l'acquisizione dell'audio tramite microfono. Una funzione di *callback* pubblica la traccia audio sul topic *mic_data* una volta acquisita.

Il nodo fornisce anche i servizi *ListenStop* e *ListenStart*, rispettivamente per la disattivazione e attivazione del microfono.

1.2.2. Speech Recognition node

Implementa una funzione di *Speech Recognition* pubblicando sul topic *voice_data* solo quelle tracce audio in cui è presente del parlato. Una volta acquisita una traccia audio che contiene del parlato pubblica su *voice_txt* la sua trasposizione in testo (*Speech2Text*).

1.2.3. Speaker reidentification node

Consente di identificare un interlocutore tramite la voce. Tale funzionalità è gestista tramite due servizi.

Identify. Acquisita una traccia audio in cui è stato identificato del parlato, restituisce l'identità di un utente (se nota) con il suo embedding, *None* altrimenti.

SaveId. Consente di associare ad un embedding una identità (label). Ogni associazione $(embedding, username)$ viene salvata in un file pickle così da mantenere lo stato dell'embedding space. Per ogni utente garantisce almeno `MAX_EMBEDDING` campioni per massimizzarne l'identificazione.

Il funzionamento completo della procedura di re-identificazione viene descritto nel capitolo 3 (WP4).

1.3. Pacchetto table_pkg

Composto dei seguenti moduli.

Tablet manager node. Quando l'utente chiede di visualizzare le attività o le categorie, l'intento si propaga dalle RASA actions attraverso il topic *show_data* fino al nodo Tablet manager. Quest'ultimo funge da ponte per consentire la visualizzazione sul tablet di Pepper tramite le apposite API. Nello specifico consente di visualizzare una pagina web in host su un server Flask sulla rete locale.

Server Flask. Un semplice Web Server in esecuzione sulla rete locale usato per visualizzare le attività e le categorie per l'utente corrente su una pagina HTML.

1.4. Pacchetto pepper_nodes

Dentro questo pacchetto sono presenti diversi nodi già implementati per utilizzare alcune delle funzionalità di Pepper. Il pacchetto originario è stato implementato dal DIEM, fornitoci per la realizzazione del progetto e disponibile su GitHub.

Delle funzionalità implementate, quelle utilizzate nella nostra applicazione sono:

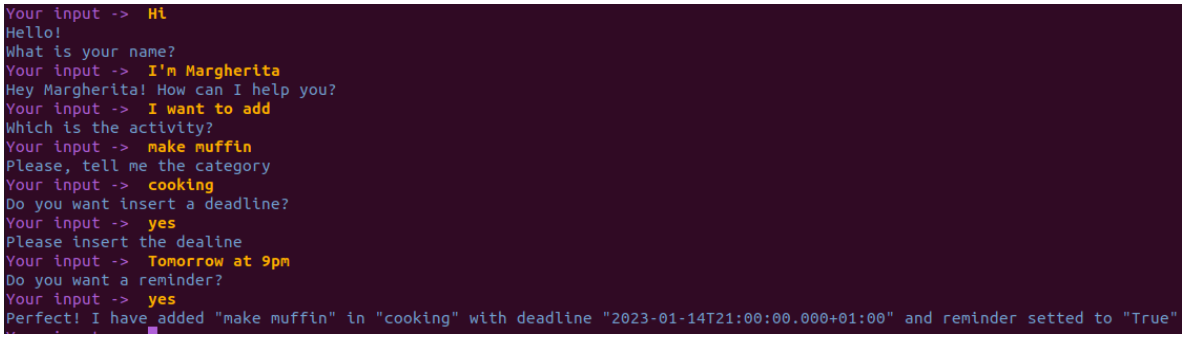
1. **WakeUp –** per il risveglio e la messa in opera del robot
2. **Text2Speech –** per l'utilizzo della voce sintetica di Pepper
3. **TabletNode –** per visualizzare sul browser del tablet delle pagine web.

2. WP3 – Rasa integration

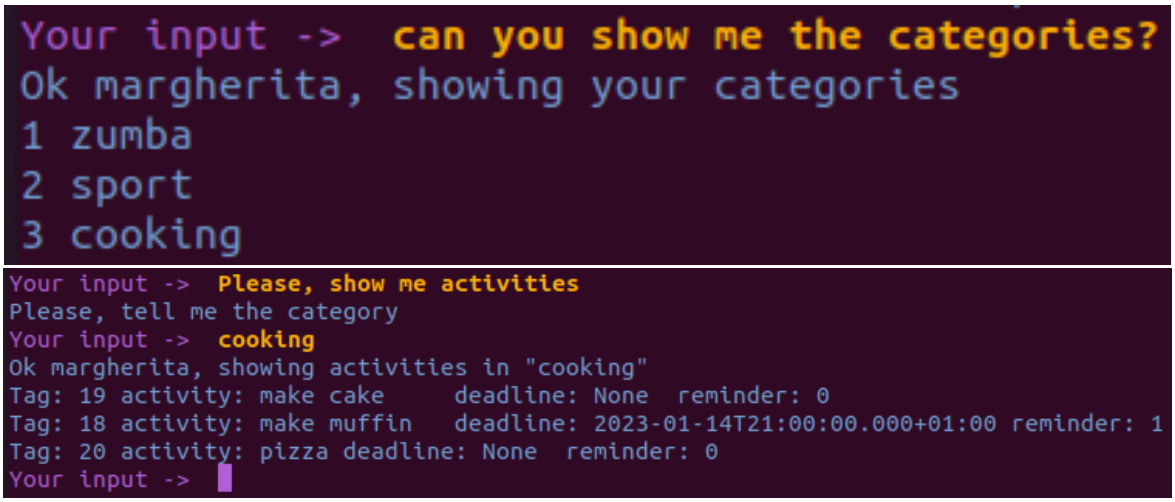
2.1. Esempi di lavoro

In questo paragrafo si riportano dei semplici esempi di funzionamento del chatbot.

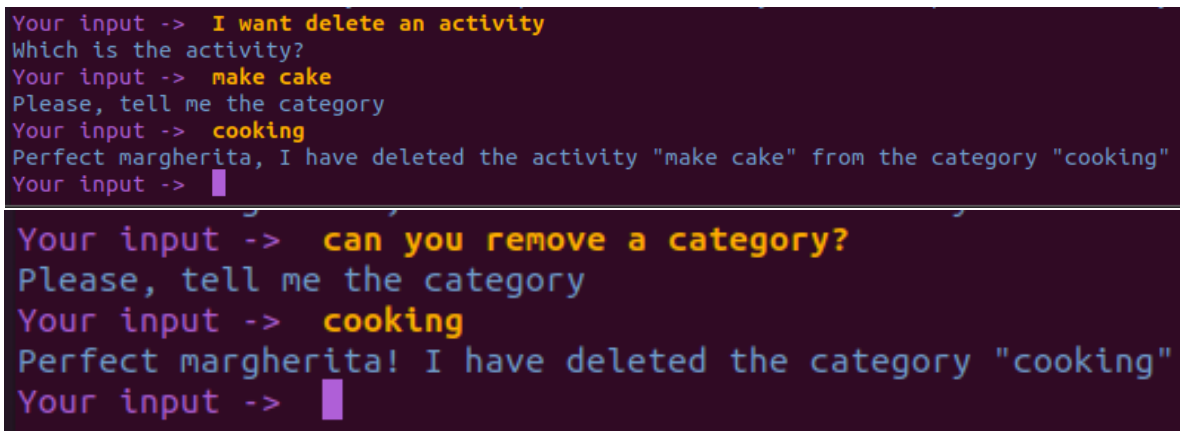
Action insert



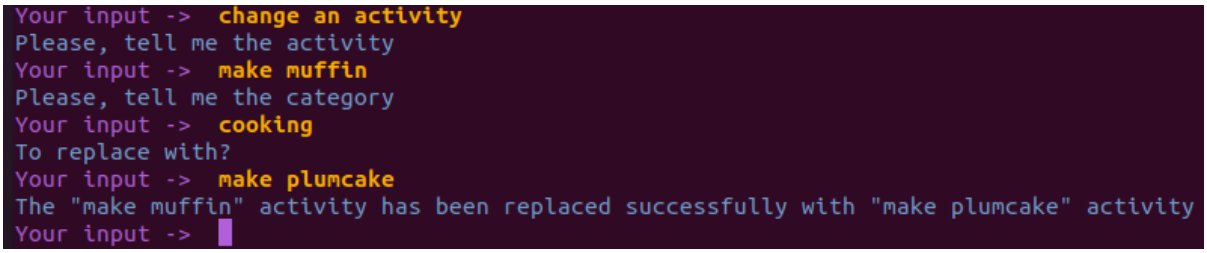
Action show



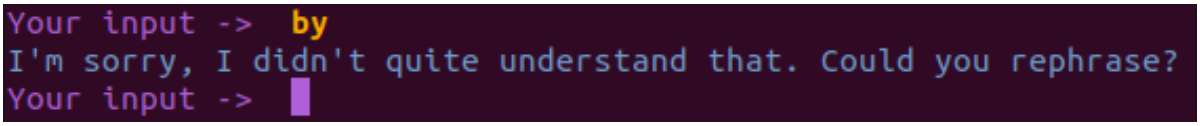
Action delete



Action update



Un'importante caratteristica del bot è che, se non ottiene una sufficiente confidenza nel rilevamento degli intenti, chiede all'utente di riformulare la frase. Questo evita errori, false interpretazioni e l'esecuzione di azioni non desiderate.



Questo è un caso lampante, ma può essere realizzato anche in casi d'uso più complessi.

2.2. Explanation of the design choices

In questo capitolo andiamo ad approfondire il dialogue management, in particolare analizziamo i file più significativi per l'implementazione del chatbot, discutiamo nel dettaglio la pipeline NLU, le custom action e come abbiamo implementato alcune funzionalità opzionali per il bot, come la gestione dei multiutenti.

Una scelta progettuale è stata quella di utilizzare un database che salvasse in un'unica tabella le todo list degli utenti. In questo modo, grazie a delle semplici query, riusciamo ad accedere ai campi di interesse e svolgere le funzionalità. Per la gestione del database utilizziamo SQLite.

tag	user	category	activity	deadline	reminder
10	roberto	shopping list	buy banana	2022-11-12T00:00:00.000+0…	1
11	francesco	shopping list	by pizza	NULL	0
12	lucia	reminder	solve exercise	2022-12-16T00:00:00.000+0…	0

Dall'immagine, riusciamo a visualizzare i campi della tabella che contiene le todo list dei vari utenti.

Domain File. Di seguito elenchiamo gli intenti definiti per il chatbot:

- **Greet -** gestisce i saluti iniziali della conversazione dell'utente
- **Goodbye** - gestisce i saluti di fine conversazione da parte dell'utente
- **Thankyou** - gestisce le conversazioni di ringraziamento
- **Identification** - gestisce l'identificazione da parte dell'utente
- **Update** - l'utente desidera eseguire l'azione di modifica
- **Affirm -** l'intenzione dell'utente è quella di affermare
- **Deny** - l'intenzione dell'utente è quella di negare
- **Insert** - l'utente desidera eseguire l'operazione di inserimento
- **show_categories** - l'utente desidera visualizzare le sue categorie
- **show_activities -** l'utente desidera visualizzare le sue attività in una categoria
- **remove_category** - l'utente desidera rimuovere una categoria
- **remove_activity** - l'utente desidera rimuovere un'attivata in una categoria
- **inform** - tale intento identifica le informazioni di interesse da fornire al *form* RASA
- **bot_challenge** - gestisce le chiacchiere da parte dell'utente con il bot

Infine, elenchiamo le entità definite per il chatbot:

- **activity**
- **category**
- **time**
- **username**

2.2.1. NLU File

Gli esempi di addestramento sono raggruppati per intento nel file *nlu.yml*. La strutturazione delle frasi di esempio ha subito qualche modifica, insieme ai rispettivi intenti, rispetto alla consegna della prova midterm.

Infatti, prima di integrare il chatbot con il modulo di Speech2Text, il sistema utilizzava esempi di addestramento che includevano frasi lunghe, all'interno delle quali erano specificate più entità. Ad esempio, per l'intento di insert, la formulazione era la seguente: "*I want insert [go out with friends](activity) in [reminder](category) for [tomorrow](time)*".

Alla luce delle prove fatte con il modulo di Speech2Text, si è preferito ristrutturare la conversazione in frasi più brevi così da migliorarne l'interpretabilità.

2.2.2. Config File

Pipeline. Di seguito riportiamo la pipeline scelta.

```yaml
pipeline:
  - name: SpacyNLP
    model: "en_core_web_md"
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
  - name: RegexFeaturizer
  - name: CRFEntityExtractor
  - name: EntitySynonymMapper
  - name: SklearnIntentClassifier
    kernels: ["linear","poly","rbf"]
  - name: ResponseSelector
    epochs: 100
    model_confidence: "linear_norm"
    constrain_similarities: true
  - name: "DucklingEntityExtractor"
    url: "http://localhost:8000"
    dimensions: "time"
    timezone: "Europe/Berlin"
    - name: FallbackClassifier
    threshold: 0.3
```

Si tratta di una pre-trained pipeline basata sul modello SpacyNLP pre-addestrato per la lingua inglese (*en_core_web_md*) e completata dai seguenti componenti:

1. **SpacyTokenizer** - per dividere il testo dell'input dell'utente in token
2. **SpacyFeaturizer** - per creare caratteristiche dal testo dell'input della conversazione, che possono essere utilizzate per identificare l'intenzione dell'utente e le entità
3. **RegexFeaturizer** - che utilizza espressioni regolari per estrarre informazioni dal testo in input
4. **CRFEntityExtractor** - che utilizza un algoritmo di riconoscimento per identificare le entità specifiche nel testo dell'input dell'utente
5. **EntitySynonymMapper** - utilizzato per mappare i sinonimi di un'entità specifica ad un'entità standard
6. **SklearnIntentClassifier** – ulteriore modulo per la classificazione delle intenzioni
7. **ResponseSelector** - utilizzato per selezionare la risposta più appropriata per una determinata intenzione dell'utente
8. **DucklingEntityExtractor** - estrattore di entità che utilizza il servizio Duckling di Facebook per estrarre informazioni di tipo temporale e numerico dal testo in input.
9. **FallbackClassifier** - per la gestione di casi in cui il modello non è in grado di identificare con certezza l'intenzione o l'entità, a causa della confidenza troppo bassa.
   Con il parametro *threshold* specifichiamo la soglia di confidenza minima per cui il modello deve considerare una risposta come fallback.

Questi componenti vengono eseguiti in ordine sequenziale, con l'output di un componente che diventa l'input per il successivo. In questo modo, la pipeline consente di elaborare l'input dell'utente in modo efficiente e preciso, fornendo informazioni utilizzabili per il modello di conversazione.

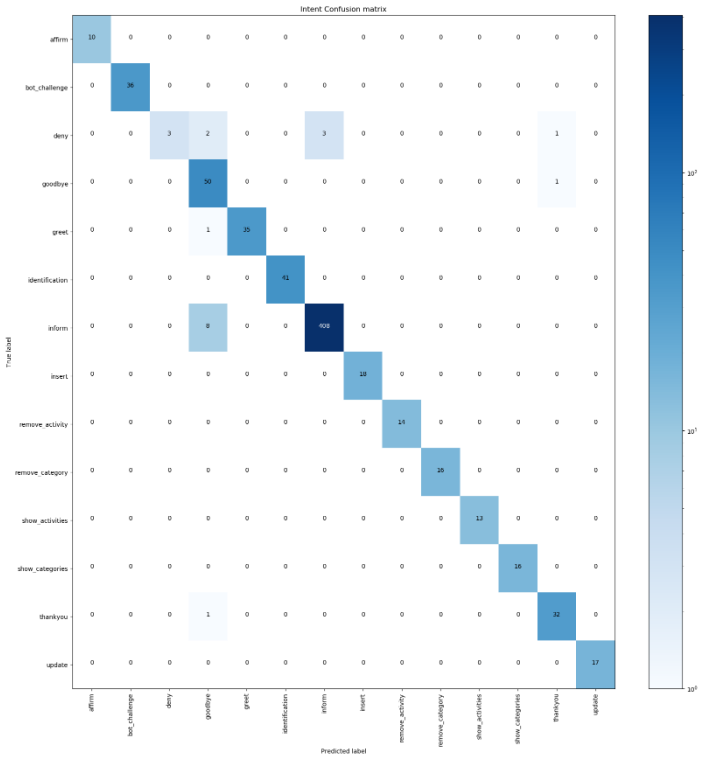
Policies. L'evolversi della conversazione è gestito da diversi meccanismi di policy. Le tre politiche utilizzate sono:

1. **RulePolicy –** la conversazione evolve in base alla *rules*, definite nel file *rules.yml*
2. **MemoizationPolicy –** utilizza le *stories* per predire l'evolversi della conversazione se non ci sono corrispondenze con le *rules* definite.
3. **TEDPolicy –** utilizza l'apprendimento automatico per predire la migliore azione successiva, se non ci sono corrispondenze con quanto scritto nelle *rules* e nelle *stories*.

Queste tre policy, ad ogni turno, prevedono l'azione successiva con un certo livello di confidenza, secondo una gerarchia basata sulle priorità.

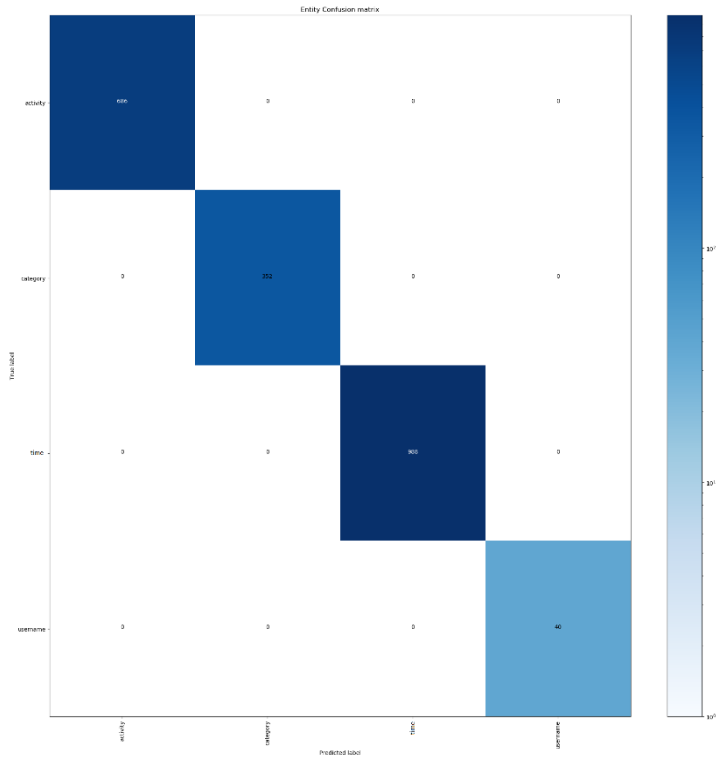
Performance. Le matrici sotto riportate hanno lo scopo di evidenziare le performances ottenute dal sistema con la pipeline scelta.

Segue la matrice di confusione degli intenti.



Gli intenti su cui il sistema ha più difficoltà sono *deny*, *inform* e *thank you*.

Infine, la matrice di confusione delle entità.



Notiamo come essendo stati forniti più esempi di *activity* queste vengono meglio classificate rispetto a *category* e *username*. L'utilizzo dell'estrattore *duckling* consente di ottenere un elevata accuratezza sull'entità *time*.

2.2.3. Actions

Il sistema di dialogo deve permettere all'utente di aggiungere, rimuovere, aggiornare e mostrare attività nella todo list dell'utente. Per questo motivo, nel file *actions.py* sono state definite le seguenti sei custom actions:

1. **ActionInsert** – per l'inserimento di un'attività e tutti i suoi dettagli nel database;
2. **ActionRemove** – per la rimozione di una attività o di una categoria dal database;
3. **ActionShow** – per mostrare le attività in una categoria o le categorie per l'utente corrente;
4. **ActionUpdate** – per modificare il nome di un'attività nel database;
5. **ActionCustomReset** – per resettare tutti gli slots, tranne "username" (utile ai fini Dialog Management);
6. **ActionStoreActivity** – per salvare l'activity identificata in uno slot temporaneo (utile ai fini Dialog Management)

Apposite regole e storie, in combinazione a form, guidano l'utente nel completare gli slot necessari al corretto funzionamento delle azioni.

2.2.4. Manage multiple users

Alla base delle gestione multiutente c'è la scelta di utilizzare un database SQLite per la memorizzazione delle todo list associate a ciascun utente.

Per garantire l'accesso alla todo list personale abbiamo strutturato la conversazione in modo tale che nella fase iniziale l'utente debba necessariamente identificarsi. Al seguito dell'identificazione, il chatbot manterrà in un apposito slot l'identificativo dell'utente con il quale accedere alla propria todo list. Al termine della conversazione, lo slot sarà resettato preparando il sistema ad accogliere un nuovo utente.

L'identificativo dell'utente viene passato direttamente al chatbot se questo viene correttamente re-identificato dal modulo di re-identification, altrimenti viene fornito dall'utente sotto richiesta del sistema.

2.2.5. Set deadline and reminder

In fase di inserimento di una nuova attività, viene chiesto all'utente se vuole aggiungere una deadline ed un reminder che gli notifichi la scadenza di quella deadline. Queste informazioni vengono mantenute in un apposito slot e salvate all'interno del database.

La logica del remainder è la seguente: il modulo Reminder accede alle informazioni salvate all'interno del database e tramite semplici query verifica quali attività sono in scadenza per l'utente specificato consentendone la notifica da parte del Dialogue Interface.

2.3. ROS integration

Come specificato nella sezione 1.1.1, per l'integrazione Rasa-Ros è stato implementato un servizio ROS che prende in input una stringa da sottoporre al chatbot e restituisce la risposta del chatbot sotto forma di stringa. Questo meccanismo è realizzato tramite richieste HTTP da parte del servizio ROS con il server RASA.

3. WP4 – Re-Identification

In questo capitolo discutiamo nel dettaglio del modulo di re-identification. Esso sfrutta principalmente due servizi Identify Service e SaveId Service.

3.1. Identify Service

Tutti i processi di re-identificazione lavorano utilizzando l'embedding audio. Tale embedding è ottenuto chiamando il servizio Identify, il quale, preso in input una forma d'onda restituisce in output il vettore di embedding e l'id della persona associata.

3.2. SaveId Service

Questo servizio prende in input l'id della persona che sta parlando, il segnale audio della sua voce o il suo vettore di embedding. Questi parametri verranno salvati sia nell'embedding space attuale che in un file, per rendere i dati persistenti.

3.3. Speaker Identification

Per la re-identificazione viene utilizzato uno spazio di embedding costruito salvando le identità delle persone che hanno interagito con il sistema e i loro rispettivi vettori di embedding.

La procedura di re-identificazione è divisa nelle seguenti fasi:

- **Estrazione delle feature:** preso un segnale audio in input viene pre-elaborato per estrarne le feature;
- **Fase di predizione:** le feature estratte al passo precedente vengono passate alla DNN *DeepSpeaker* per creare un nuovo embedding che verrà comparato con tutti gli embedding salvati utilizzando la metrica *cosine similarity*.
  Se la probabilità di appartenere all'identità con minore distanza coseno è maggiore di una soglia prefissata, l'identità verrà restituita. In caso contrario, verrà restituito *None*.

3.4. Scelte implementative

Per garantire un corretto processo di identificazione è stato supposto che l'interlocutore del chatbot sia sempre lo stesso e che un suo possibile cambio avvenga solo dopo la ricezione dell'intento *goodbye*, indice di fine conversazione.

Inoltre, al fine di garantire una maggiore efficienza, lo spazio di embedding dell'interlocutore viene arricchito solo all'inizio della conversazione. È stata fatta questa scelta siccome è stato notato che continuando ad aggiornare l'embedding durante la conversazione il carico di lavoro per il chatbot diventava eccessivo, con il conseguente calo di prestazioni e aumento di errori.

Infine, abbiamo deciso di inserire una soglia massima pari a sette campioni per utente in quanto, in seguito agli esperimenti condotti, si è rivelato un buon compromesso per ottenere una buona discriminazione delle identità. Raggiunta tale soglia non verranno più salvati campioni per quell'utente.

3.5. Pepper speaking issues

Per il corretto funzionamento del dispositivo si deve assicurare un ambiente silenzioso. Inoltre, essendo prevista l'interazione verbale con Pepper, se il microfono resta attivo acquisirà anche l'audio prodotto da Pepper portando la conversazione in un loop o a un blocco del sistema, nel peggiore dei casi.

Per ovviare a questo problema sono stati inseriti due servizi all'interno del nodo di Voice Detection (ListenStart e ListenStop) che disattivano il microfono quando il robot sta parlando e lo riattivano quando l'utente deve interagire.

4. WP5 & WP6 – Test

I primi test svolti si sono concentrati nel verificare il corretto funzionamento di singole funzionalità del progetto. Solo dopo, in fase di integrazione, abbiamo testato il sistema nel complesso.

4.1. Unit Test

4.1.1. Re-identification node

I test condotti sul nodo di re-identification sono di due tipi:

1. Test sull'implementazione dei servizi Predict e SaveId (per verificarne il corretto funzionamento)
2. Test specifici per l'identificazione degli utenti

Al di là dei test di implementazione, i test del secondo tipo si sono svolti nella seguente modalità:

1. **Setup -** Raccolta di `MAX_EMBEDDING` campioni per due diversi utenti
2. **Predict -** a turni di tre tentativi, ogni utente ha sottoposto un suo campione audio al modulo segnando le predizioni corrette ottenute.

	Turno 1	Turno 2	Turno 3
Utente 1	3/3	2/3	2/3
Utente 2	3/3	2/3	2/3

Dai test effettuati evinciamo che:

- un numero di sette campioni per utente sembra essere sufficiente per discriminare le diverse identità;
- ci sono differenze di performance (in termini di identificazione corretta) dal luogo in cui i campioni per utente sono acquisiti e dal luogo in cui il sistema viene messo in opera. Per tanto, raccomandiamo di riacquisire i campioni utilizzati per la re-identificazione qualora l'ambiente della messa in opera del sistema e quello dell'acquisizione dei campioni non dovessero coincidere.

I test sono stati condotti tramite l'esecuzione di un nodo ROS implementato dal file *reidentification_test.py* in *ros_audio_pkg*.

4.1.2. Speech2Text

Per testare questo modulo abbiamo lanciato il file *speech2text.launch* all'interno del pacchetto *ros_audio_pkg* con il quale ci assicuriamo che venga eseguita una corretta calibrazione dell'ambiente, un corretto avvio della registrazione e che il funzionamento del modello di riconoscimento vocale di Google avvenga correttamente.

4.1.3. Pepper nodes

Dopo aver avviato i nodi di Pepper, nel pacchetto *pepper_nodes*, tramite *pepper_bringup.launch*, per testare il corretto funzionamento dei servizi implementati, abbiamo utilizzato il comando:

```
rosservice call [nome servizio] [parametri]
```

4.2. Integration test

4.2.1. Ros and Rasa

Alla consegna della midterm ci è stata fornita l'implementazione di un nodo ROS con il quale poter inviare da terminale i messaggi che volevamo sottoporre al chatbot in esecuzione sul server RASA.

Per completare l'integrazione, abbiamo realizzato un file *.launch* con il quale avviare tutti i servizi necessari al chatbot per funzionare (rasa server, rasa actions, server duckling) compresi i nodi per l'integrazione forniti.

Con questo setup abbiamo potuto verificare la corretta integrazione tra RASA e ROS semplicemente conversando con il chatbot tramite ROS node.

4.2.2. Chatting with Pepper

Di seguito i test che siamo riusciti ad effettuare su Pepper.

Siamo riusciti ad integrare con successo il modulo di Text2Speech di Pepper con il Dialog Interface. Abbiamo quindi intrapreso una conversazione dando voce al chatbot tramite Pepper e a sottoporgli dei comandi tramite microfono esterno. La conversazione ha toccato le funzionalità di inserimento, rimozione e aggiornamento richieste per il chatbot. Tuttavia, non siamo riusciti a tesate una conversazione che incorporasse più di due o tre comandi a causa di problemi con il microfono esterno.

Siamo anche riusciti ad utilizzare con successo il servizio *LoadUrl* per visualizzare le pagine web da noi realizzate. Quindi, in seguito alla richiesta di visualizzazione, il tablet ha correttamente mostrato una tabella contenente attività o categorie per un utente.

I test effettuati sul modulo di re-identificazione con il microfono esterno fornitoci sono stati soggetti a eccessivo “rumore” e non sufficienti in numero da poter dichiarare il corretto funzionamento.

4.3. Final test (without Pepper)

Con questi test abbiamo provato l'intera logica dell'applicazione simulando i servizi di Speech2Text e del tablet forniti da Pepper rispettivamente con un sintetizzatore vocale e con il browser del computer.

L'esecuzione di questi test prevede l'impostazione della variabile `PEPPER` all'interno del file *config.py* a `False`. Seguono i test effettuati.

TEST 1.a. In questo test verifichiamo che la re-identificazione di un utente A avvenga correttamente.

TEST 1.b. In questo test verifichiamo che la re-identificazione di due utenti diversi avvenga correttamente. In particolare, effettuiamo la re-identificazione con un utente A e avviamo una nuova conversazione re-identificando un altro utente B.

TEST 2. Questo test ha come obiettivo quello di verificare che l'integrazione di tutti i moduli sia avvenuta correttamente e che il sistema rispetti la logica richiesta.

Il test è strutturato come segue:

1. Re-identificazione di un utente A
2. Aggiunta di un attività in una categoria C1
3. Aggiunta di un'altra attività nella categoria C1
4. Aggiunta di una attività in una categoria C2
5. Visualizzazione delle categorie per l'utente A
6. Visualizzazione delle attività nella categoria C1
7. Aggiornamento di una attività nella categoria C1
8. Rimozione di una attività nella categoria C1
9. Rimozione della categoria C2
10. Saluto

Al termine di questo test abbiamo potuto verificare:

- Il corretto funzionamento del chatbot e delle sue funzionalità (inserimento, rimozione, visualizzazione e aggiornamento)
- La corretta integrazione tra chatbot e l'enviroment ROS
- La corretta integrazione tra i moduli di Voice Detection, Speech2Text e Re-identification con la logica implementata in Dialog Interface.

TEST 3. Con questo test abbiamo verificato la funzionalità di *reminder* e che la sua integrazione con la logica del sistema sia avvenuta correttamente.

Ricordiamo che la funzionalità in esame consente di notificare l'utente quando si è a ridosso di una deadline impostata per una attività.

Nel test l'utente A inserisce una nuova attività impostando una deadline e la funzionalità di notifica (*reminder*) in una categoria a sua scelta. Il test termina con successo quando, arrivati vicini alla deadline, il sistema notifica correttamente l'attività in scadenza.