

UNIVERSITÀ DEGLI STUDI DI PARMA

Corso di Laurea in Ingegneria dei Sistemi Informativi

Tesi di Laurea di primo livello

Realizzazione di un assistente vocale per il triage ospedaliero

Triage semi-automatizzato per la sicurezza e la tempestività



Relatori

prof.ssa Mordonini Monica
prof. Tomaiuolo Michele

Laureando

Simone Montali
matricola 288144

09 luglio 2020

A tutti i professori che nel passato sono riusciti a mantenere viva la fiamma della mia curiosità, nonostante mille venti vi soffiassero sopra.

prof. Serventi, prof. Vescovi, prof. Melegari, prof. Mori, prof. Franchi, prof. Forni, prof. De Martino...

A computer would deserve to be called intelligent if it could deceive a human into
believing that it was human.

Alan Turing

Ringraziamenti

È anzitutto dovuto un enorme ringraziamento ai relatori di questa tesi, la prof.ssa **Mondonini**, il prof. **Tomaiuolo**, ed il prof. **Angiani**. Dal partorire l'idea, alla scelta degli strumenti più adatti, al suggerimento di nuove feature, questi professori hanno supportato questo viaggio in ogni modo possibile, dandomi la possibilità di confrontarmi con temi che, un giorno, spero diventino il mio lavoro. Riconosco di essere una persona a volte testarda e ostinata: questo non mi è mai stato fatto pesare ed è anzi stato assecondato. Un sentito grazie va anche a **Gianfranco Lombardo**, dottorando del laboratorio Sowide, che mi ha dato ottimi spunti per migliorare il funzionamento del progetto, senza i quali non esisterebbe il capitolo 5 di questa tesi.

Vorrei ringraziare **mio fratello Nico**, da anni la mia *stella polare* nel mondo dell'informatica, cui sottopongo idee a qualsiasi ora del giorno e della notte e che ha sempre un consiglio, un'opinione, una soluzione che la svoltano.

Grazie a **mia madre Rossella**, che è sempre stata comprensiva, mi ha supportato anche quando le mie scelte non le piacevano, ha dovuto per anni ascoltare i discorsi di tre noiosi informatici ad ogni pasto. Grazie a **mio padre Fabrizio**, che mi ha insegnato a non accontentarmi mai, di nulla. Che mi ha cresciuto con il mantra del "se fai qualcosa, fallo come si deve". Che, mentre frequentavo le scuole elementari, mi regalò "Il mio primo libro di BASIC". Spero di diventare un decimo dell'uomo che sei tu.

Infine, un grande ringraziamento ai miei compagni di corso **Caterina, Daniele, Guido, Mattia, Pietro, Riccardo**. Avete reso questi tre anni un po' più facili, soprattutto per uno come me che a lezione proprio non ci sa stare.

Sommario

Questa tesi si pone l'ambizioso obiettivo di poter **automatizzare le procedure di triage** nei pronto soccorso italiani ed esteri, tramite la creazione di un assistente vocale con cui i pazienti possano interagire per risolvere le proprie necessità mediche. Il COVID19 ha reso necessarie procedure di contenimento sempre più stringenti, che difficilmente verranno abbandonate in futuro. Per questo, **ridurre l'interazione** tra il personale sanitario ed i pazienti potrebbe essere un grande passo in avanti per la sicurezza di tutti. Con triage, termine nato in ambito di emergenze mediche oggi utilizzato anche negli ospedali, si intende la classificazione dei pazienti in base all'**urgenza delle loro patologie** ed alla **probabilità di sopravvivenza** che hanno. Questa procedura è oggi svolta da infermieri, ma potrebbe in parte essere affidata ad un computer. Spesso, però, gli utilizzatori dei servizi medici **non sanno utilizzarne uno**: la necessità di un'interazione più semplice è palese. Per questo, un assistente vocale potrebbe risolvere questa problematica: l'interazione vocale è semplice e chiara per tutti.

Per lo sviluppo di questo strumento si è fatto uso del software **Mycroft**, un assistente vocale open source, con skills modulari espandibili tramite Python. Sono state definite due nuove skills, che ascoltano il paziente e svolgono la procedura di classificazione: cercano di comprendere a quale **macrocategoria** appartengono i sintomi, domandano al paziente informazioni riguardo a **come si sente**, come *dolore, febbre, capacità di camminare*, ed infine approfondiscono l'**anagrafica**, con nome, età, etc...

Vista la difficoltà del periodo, si è configurata la necessità di effettuare **diagnostica alla ricerca di sintomi del COVID19**. Per questo, quando il paziente lamenta patologie compatibili con il suddetto, viene attivata un'automazione che tenta di capire, con una determinata accuratezza, quanto il paziente potrebbe esserne affetto.

Infine, i pazienti potrebbero anche desiderare **informazioni di carattere medico**, senza doversi per forza far ricoverare: è stata quindi implementata la possibilità di consultare le informazioni disponibili sull'Enciclopedia Medica del Ministero della Salute. Per ottenere le suddette, è stato creato uno **scraper** che consulta il sito e le salva o invia tramite un'API REST.

Questo insieme di strumenti ha **necessità hardware** molto permissive: è possibile installarlo su un Raspberry Pi, su un vecchio computer o anche su un dispositivo Android. Questo rende possibile l'upgrade a strumentazioni simili anche per le realtà più in difficoltà, dentro e fuori dall'Italia.

L'installazione di un bot simile a questo negli ospedali ne migliorerebbe la sicurezza, l'efficienza, i costi e **la possibilità di salvare vite**.

Indice

Elenco delle figure	VIII
1 Introduzione	1
2 Stato dell'arte	2
2.1 Machine Learning	2
2.1.1 Reti neurali	4
2.2 Assistenti vocali	6
2.2.1 Assistenti vocali in ambito medico	7
2.3 Triage	9
2.3.1 Come il COVID19 ha modificato le procedure di triage	9
3 Mycroft	11
3.1 Open source	11
3.1.1 Cosa si intende con open source?	11
3.1.2 Perché open source?	12
3.2 Introduzione a Mycroft	12
3.2.1 Stack di funzionamento	12
3.2.2 Interfaccia grafica	14
4 Realizzazione di skills	15
4.1 Struttura dei file della skill	15
4.2 Funzionamento della skill	16
4.2.1 Metodi relativi alle sintomatologie	17
4.2.2 Decorators utilizzati	18
4.2.3 Helper methods	20
4.2.4 Oggetto <code>med_record</code>	21
4.3 Esempio di conversazione	22
5 Fallback skills e NLP	25
5.1 Classificazione del testo	25
5.2 Dataset utilizzato	25
5.2.1 Traduzione del dataset	26
5.3 Addestramento della rete neurale	27
5.3.1 <code>fastai</code>	27
5.3.2 Preparazione dei dati	27

5.3.3	Addestramento della rete	29
5.4	Realizzazione di fallback skills	31
5.4.1	Registrazione della skill	31
5.5	Esempio di conversazione	32
6	Richieste di informazioni da parte degli utenti	34
6.1	Enciclopedia medica del Ministero della Salute	34
6.1.1	Utilizzo dell'encyclopedia	34
6.2	Esempio di conversazione	38
6.3	Aggiornamento dei dati	39
7	Applicabilità e requisiti tecnici	40
7.1	Requisiti tecnici	40
7.1.1	Text To Speech e Speech To Text	40
7.1.2	Analisi degli intent	40
7.1.3	fastai	40
7.1.4	Interfaccia grafica	41
7.2	Compatibilità, dispositivi ottimali e costi	41
8	Conclusioni	43
Riferimenti bibliografici		45

Elenco delle figure

2.1	Email spam/non-spam	3
2.2	Categorizzazione del dataset	4
2.3	Modello lineare	5
2.4	Modello a 2 strati, ancora lineare	5
2.5	Rete neurale	6
2.6	Percentuali di utenti che hanno utilizzato/vorrebbero utilizzare assistenti vocali in ambito medico	7
2.7	Interesse dei consumatori negli assistenti vocali in ambito medico, per fasce di età	8
2.8	Schema del protocollo CESIRA	9
3.1	Schema di funzionamento di Precise	13
4.1	Alcune schermate della GUI durante la diagnostica del COVID19	24
5.1	Reti neurali standard, chiamate più volte in base alla dimensione dell'input	28
5.2	Recurrent Neural Network, aperta	28
5.3	Recurrent Neural Network	29
5.4	Confusion Matrix	30
5.5	Tabella delle accuracies	30
6.1	Enciclopedia della Salute	35
7.1	Raspberry Pi con display originale	41
7.2	Mycroft Mark II	42

Capitolo 1

Introduzione

In una società sempre più digitalizzata e dinamica, spesso viene a crearsi un netto distacco tra i settori capaci di **evolvere assieme alla tecnologia**, e quelli che, per un motivo o per l'altro, rimangono ancorati a procedure e metodologie tradizionali. Il settore medico, in continuo rinnovamento sul lato scientifico, è, anche in Italia, affidato ad **infrastrutture informatiche progettate tempo fa**. Questo, soprattutto per motivi di stabilità e affidabilità: gli errori, qui, non sono ammissibili.

Per questo motivo, spesso non si notano le evidenti possibilità di miglioramenti che la ricerca informatica potrebbe apportare agli ospedali, agli ambulatori, agli studi. L'obiettivo di questa tesi è mettere in luce una delle possibili modalità con cui l'informatica potrebbe, in un futuro prossimo, migliorare la praticità ma soprattutto la sicurezza degli ambienti ospedalieri.

La procedura di triage, ossia il processo di selezione dei pazienti richiedenti cure, è oggi affidata totalmente ad infermieri. Questa scelta è dovuta, oltre ad un evidente bisogno di poter osservare il paziente, alla necessità dell'**immediatezza di un contatto verbale** con il personale sanitario.

Perciò, la sfida nella realizzazione della tesi è soprattutto legata ad ottenere un'interazione che sia la **più umana possibile**. Il bot deve quindi accogliere il paziente, comprenderne le problematiche ed i sintomi, e farlo sentire compreso. Non si escluderà del tutto un apporto umano: gli infermieri sono addestrati per poter osservare e comprendere il richiedente cura, e l'apporto dell'osservazione diretta del paziente è ancora troppo importante per escluderla. È senz'altro possibile, però, affidare la prima parte di **profilazione dell'utente** a procedure automatizzate.

Tutto il codice di questa tesi è open source con licenza GPL-3.0, ed è reperibile su GitHub nelle seguenti repository:

- Questo documento: [montali/tesi](#)
- Skill di Mycroft per diagnosi e informazioni (capitolo 4) [montali/hospital-triage-skill](#)
- Skill fallback di Mycroft per il riconoscimento di sintomatologie tramite NLP (capitolo 5) [montali/hospital-fallback-skill](#)
- Scraper dell'Enciclopedia Medica del Ministero della Salute (capitolo 6) [montali/medical-infos-api](#)

Capitolo 2

Stato dell'arte

Illustriamo ora lo stato dell'arte delle varie tecniche e tecnologie che coinvolgono il bot.

2.1 Machine Learning

Il machine learning (in seguito, ML) è una disciplina che si occupa di rispondere, sostanzialmente, a due domande: *Come si può costruire un sistema informatico che migliora con l'esperienza?* e *Quali sono le leggi teoriche fondamentali che governano ogni sistema di apprendimento, sia esso implementato nei computer, negli umani, nelle organizzazioni?*. Il ML copre una serie di compiti diversi, dalla classificazione di email spam, al riconoscimento facciale, al controllo di robot [1]. Ogni problema di ML può essere riassunto come il miglioramento di una determinata performance (data, ad esempio, dall'errore sulle predizioni di un algoritmo di classificazione) durante l'esecuzione di un'operazione. Seguendo l'esempio delle email spam, un algoritmo di ML potrebbe apprendere, tramite un *training set* di email già etichettate come spam/non-spam, a categorizzarne altre. Portando il problema in termini matematici, vogliamo, dato il grafico della figura 2.1,

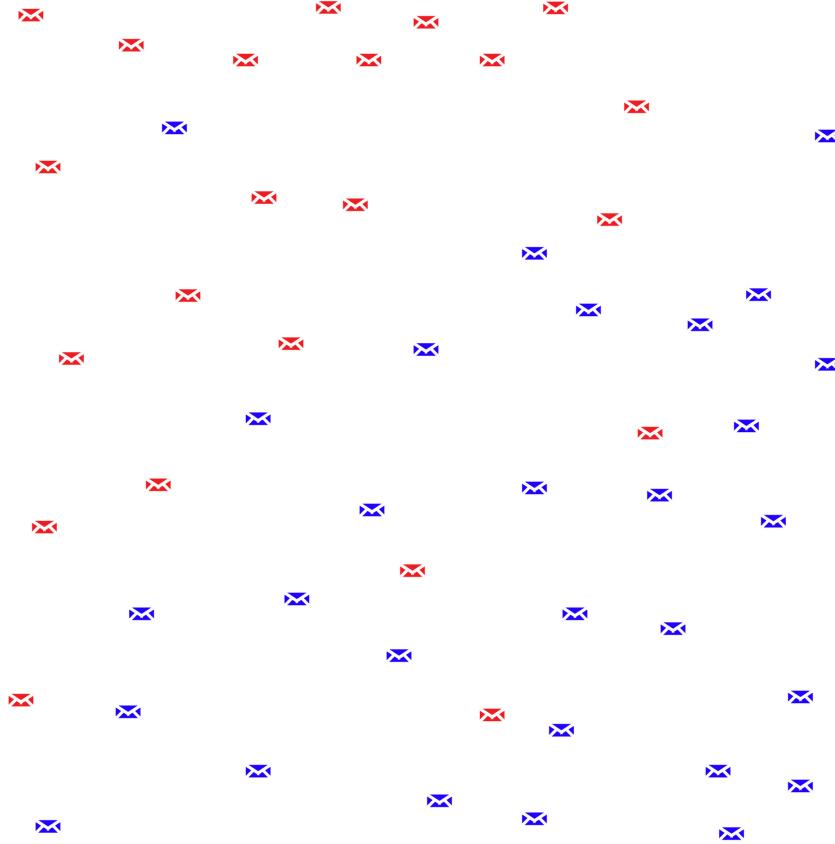


Figura 2.1: Email spam/non-spam

in cui i simboli rossi sono email spam, e quelli blu no, trovare una funzione che ci permetta di evidenziare le email da eliminare. Ciò che faremo sarà quindi definire un insieme di **features**, consistenti, ad esempio, nel contenuto testuale della mail (vettorizzato in termini numerici), l’orario di arrivo, la lunghezza in parole, etc... Oltre a ciò, definiremo una **label**, ossia la caratteristica che vogliamo predire: sarà, per esempio, 1 se l'email è spam, 0 altrimenti. Ora, il problema consiste in una semplice regressione lineare: date le features x_0, x_1, x_2 e la label y , vogliamo trovare dei pesi w_0, w_1, w_2 tali che

$$w_0x_0 + w_1x_1 + w_2x_2$$

approssimi al meglio y . Per fare ciò, procediamo per iterazioni: partiremo con dei pesi arbitrari, calcolando y per ogni esempio (già etichettato) fornito, e ne calcoleremo l’errore rispetto alla y reale. Potremo quindi calcolare il gradiente (ossia la somma delle derivate parziali sui rispettivi pesi) della funzione che lega l’errore ai pesi w_i , ottenendone un’informazione fondamentale: indicherà infatti la **direzione nella quale la funzione errore è decrescente**. Ci potremo quindi spostare, di una quantità pari al *learning rate* (uno degli iperparametri definiti arbitrariamente), nella direzione di diminuzione dell’errore. Scegliere

un corretto *learning rate* è una parte fondamentale dello sviluppo dell'algoritmo di ML: se è troppo piccolo, la soluzione richiederà troppo tempo, se troppo grande, la soluzione non sarà abbastanza precisa. Dopo un sufficiente numero di esecuzioni dell'algoritmo, il nostro grafico avrà ora l'aspetto in figura 2.2.

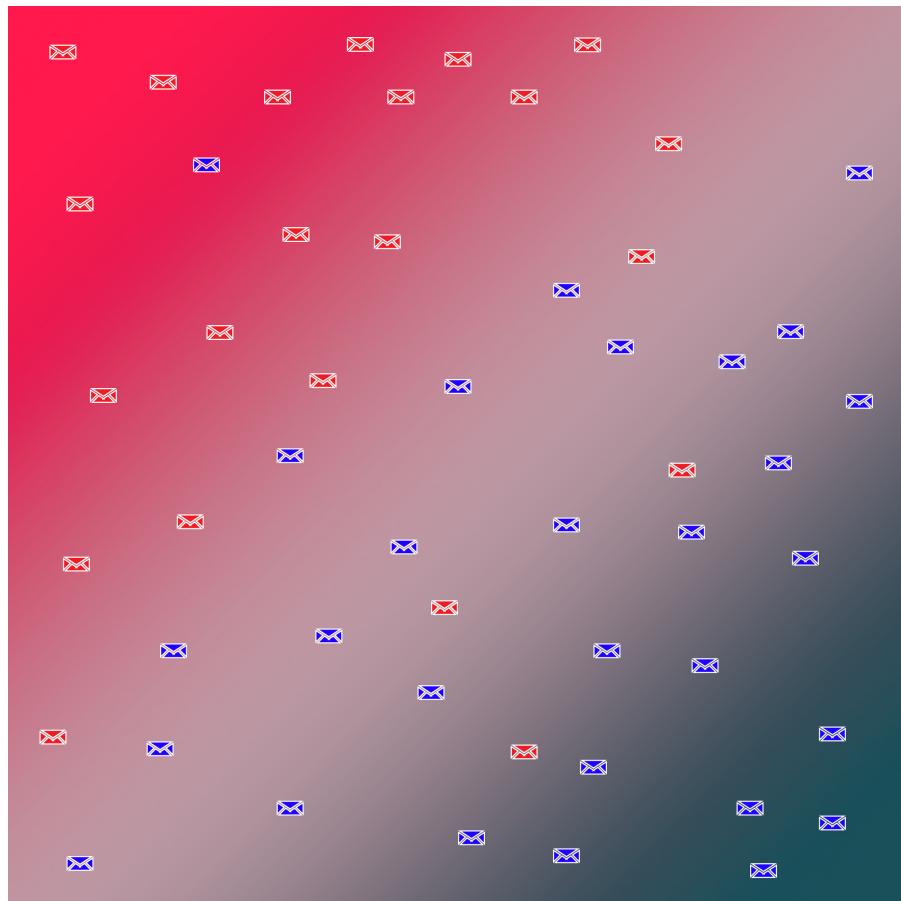


Figura 2.2: Categorizzazione del dataset

Saremo quindi in grado di determinare, con una determinata *accuracy* (in questo caso data dall'opacità del colore di sfondo), l'appartenenza di un esempio ad una classe o all'altra.

2.1.1 Reti neurali

Per illustrare il concetto di rete neurale, produciamo anzitutto un grafico del modello lineare appena definito 2.3.

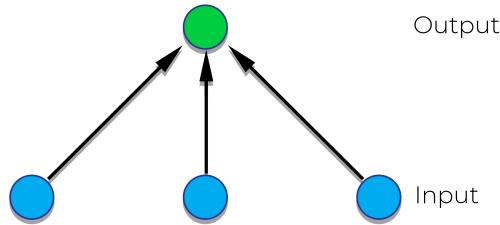


Figura 2.3: Modello lineare

Ogni cerchio blu rappresenta una feature, ed il cerchio verde rappresenta la somma pesata degli input. Vorremmo che questo modello fosse in grado di approssimare anche problemi non-lineari. Potremmo quindi decidere di inserire un nuovo strato di valori intermedi, ottenendo il modello in figura 2.4:

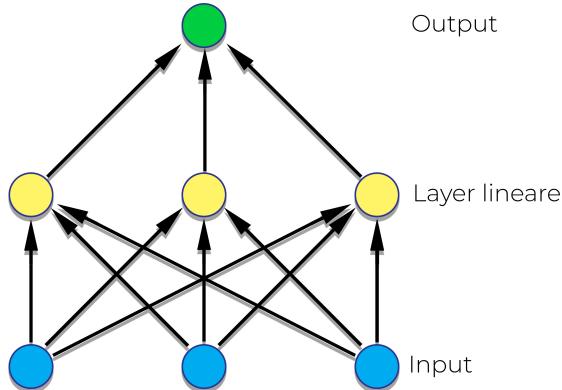


Figura 2.4: Modello a 2 strati, ancora lineare

Questo modello, però, non è ancora esente da difetti: infatti, essendo questo ancora un modello lineare, è possibile semplificarlo, ottenendo il primo grafico. È utile, allora, inserire una non-linearità all’interno del modello, che chiameremo *funzione di attivazione*: ora avremo, oltre ai layer lineari, una funzione tra di loro non lineare. Grazie a essa, l’aggiunta di nuovi strati è diventata significativa: non sarebbe più possibile semplificare il sistema ad uno lineare.

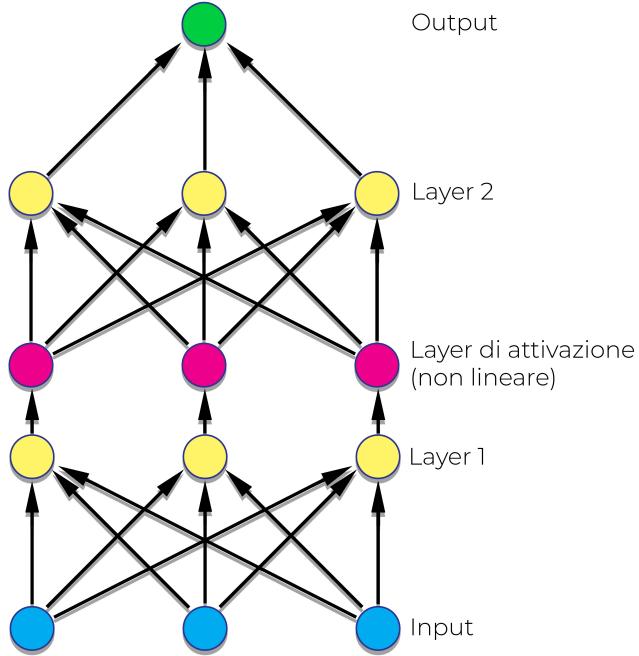


Figura 2.5: Rete neurale

Tra le funzioni di attivazione, citiamo la funzione sigmoidea $F(x) = \frac{1}{1+e^{-x}}$, o la *Rectified Linear Unit* (ReLU) $F(x) = \max(0, x)$. Non esiste una funzione di attivazione più corretta delle altre: ogni problema avrà una configurazione dei layer e delle loro attivazioni diversa. Trovare la configurazione corretta della rete neurale è la sfida più grande nel lavoro di *data analysis*.

Avremo ora quindi un modello capace di ottenere degli output date le input features. Ma come possiamo addestrarlo per ottenere dei risultati validi? Una soluzione ricorrente è quella della **retropropagazione dell’errore**, o backpropagation. Durante questo processo, la rete neurale calcola il gradiente dell’errore (ottenuto da una funzione di *loss*) sull’output, rispettivamente ai vari pesi degli strati. Questo permette alla rete neurale di individuare quali nodi hanno più effetto sulla generazione dell’errore, e di conseguenza, regolarne i pesi per ridurre l’errore. Fatto ciò sullo strato più vicino all’output, si tratta solo di ripetere il procedimento "a salire" per ogni layer della rete neurale.

2.2 Assistenti vocali

Gli assistenti vocali sono software in grado di interpretare dialoghi umani e rispondere attraverso voci sintetizzate. Alcuni esempi famosi sono *Apple Siri*, *Amazon Alexa*, *Google Assistant*. Gli utenti possono effettuare domande, controllare dispositivi di domotica, e svolgere un numero di operazioni in continua espansione[2]. Il software rimane costantemente in attesa della pronuncia di una *wake word* da parte dell’utente; una volta riconosciuta questa parola, si mette in ascolto di comandi. La richiesta viene quindi tradotta in testo tramite un motore di *Speech To Text* ed interpretata dal software. Quest’ultima è senz’altro la parte

più complessa: il compito di interpretazione di linguaggio naturale è reso complesso dalle svariate sfaccettature che ogni diversa lingua presenta. Interpretata la richiesta, l'assistente procede con le operazioni collegate, e prepara una risposta testuale da dare all'utente. Questa risposta verrà poi sintetizzata in audio da un motore di *Text To Speech*. I moderni dispositivi di assistenza vocale come Google Home o Amazon Echo si appoggiano a server in cloud per l'interpretazione delle richieste: questo permette di ridurne notevolmente le dimensioni ed i requisiti tecnici. Lo sviluppo, negli ultimi anni, di nuove tecniche di ML, unite ad un grande incremento delle potenze di calcolo e alla disponibilità di vastissimi *dataset linguistici*, hanno permesso grandi miglioramenti nella tecnologia, ma soprattutto l'applicazione ad ambiti, fino a qualche anno fa, impensabili.

2.2.1 Assistenti vocali in ambito medico

Negli anni sono emerse alcune possibili applicazioni delle tecnologie per assistenti vocali in ambito medico. La startup **Saykara** ha creato un software che permette di analizzare le conversazioni medico-paziente, allo scopo di creare schede cliniche accurate senza dover impegnare il medico in lunghe procedure di *data entry*. Un report [3] di Voicebot, organizzazione per la diffusione degli assistenti vocali, afferma che il 52% degli intervistati sarebbe intenzionato ad utilizzare assistenti vocali in ambito medico nel futuro. Solo il 7.5% (1 su 13) l'ha già fatto.

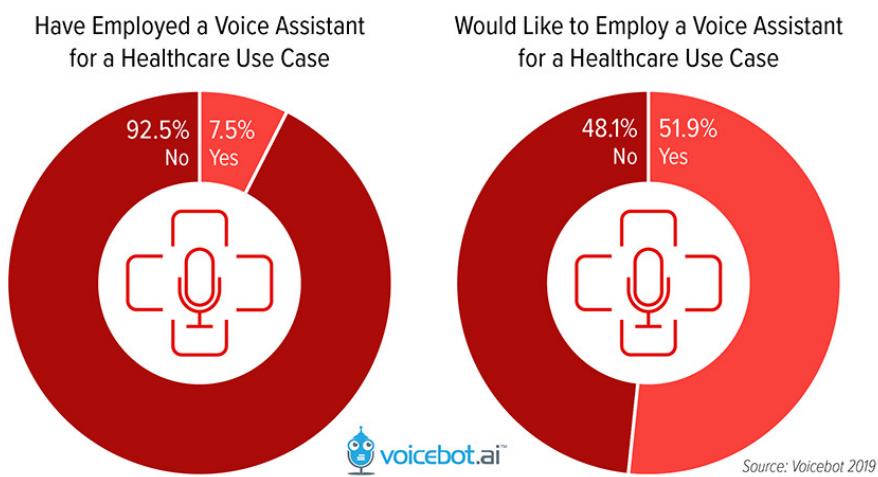


Figura 2.6: Percentuali di utenti che hanno utilizzato/vorrebbero utilizzare assistenti vocali in ambito medico

È interessante notare come anche le fasce di età più avanzate siano, seppur con percentuali più basse, interessate a provare simili applicazioni della tecnologia.

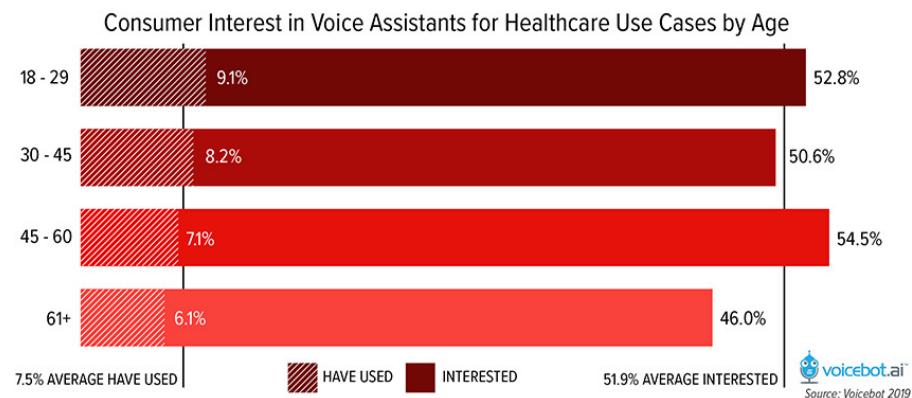


Figura 2.7: Interesse dei consumatori negli assistenti vocali in ambito medico, per fasce di età

Un altro esempio interessante di applicazione della tecnologia in ambito medico è *Dragon Medical Virtual Assistant* rilasciato da **Nuance**, una suite di strumenti vocali capaci di creare cartelle cliniche, eseguire diagnostica sul paziente, fornire risultati in diretta di ricerche mediche, supportare le operazioni di radiologia.

2.3 Triage

Il **triage** è, per definizione, il sistema utilizzato per selezionare i soggetti coinvolti in infortuni, gravi o leggeri che siano, secondo classi di emergenza, in base alla gravità delle lesioni riportate o del loro quadro clinico. Il termine nacque in ambiti di guerra (se ne cominciò a parlare durante le *guerre napoleoniche*), durante i quali si classificavano i soldati feriti in base alla probabilità di sopravvivenza. Il procedimento cominciò poi, nel corso degli anni, ad essere utilizzato in ambito ospedaliero, con l’applicazione di metodi più *scientifici* come il modello *START*, basato su un algoritmo che prevede la verifica di sintomi visibili, frequenza respiratoria, pulsazioni, risposta ai comandi. Sebbene inizialmente le procedure di triage fossero sfruttate solo in ambito di maxi-emergenze, oggi tutti i pronto soccorso italiani ne fanno uso in qualche modo. Non esiste però uno standard di codici utilizzati in ogni ospedale: le normative cambiano tra un pronto soccorso e l’altro. Distinguiamo però, generalmente, 4 tipi di codice:

- Codice **rosso**: paziente critico con cedimento di una o più funzioni vitali. *Accesso immediato*.
- Codice **giallo**: potenziale rischio di vita o compromissione di una o più funzioni vitali. *Accesso entro 15 minuti*.
- Codice **verde**: valutazione medica differibile nel tempo senza alcun rischio evolutivo. *Accesso entro 60-90 minuti*.
- Codice **bianco**: paziente senza emergenza, con problematiche risolvibili a domicilio o dal medico di famiglia.

Il triage infermieristico produce quindi una **scheda di triage**, che ha valore giuridico, contenente anagrafica, data/ora di accettazione, motivo, codice colore. Durante le emergenze è comune utilizzare il protocollo **CESIRA** (Coscienza, Emorragia, Shock, Insufficienza respiratoria, Rotture, Altro).

2.3.1 Come il COVID19 ha modificato le procedure di triage

La pandemia globale di COVID19 ha modificato i processi di triage in ospedale: lo scopo è limitare al massimo le possibilità di contagio tra i pazienti, rendendo il pronto soccorso più sicuro tanto per i pazienti, quanto per il personale sanitario. Alcuni ospedali in Inghilterra hanno addirittura iniziato a praticare triage remoto: il paziente comunica le sintomatologie online



Figura 2.8: Schema del protocollo CESIRA

e, solo se necessario, può recarsi in ospedale per la visita medica [4]. In Italia, ogni ospedale ha provveduto a definire delle procedure di **pre-triage** atte ad evitare il contatto tra pazienti affetti da COVID19 e non. Generalmente, si tratta di percorsi obbligati in cui i pazienti con sintomi compatibili con COVID19 (febbre, tosse, insufficienza respiratoria) vengono separati in un’*area di decontaminazione* e visitati senza accedere agli ambienti del pronto soccorso comune. Nell’ospedale Maggiore di Parma sono state previste, durante i momenti di picco dell’epidemia, due tende di pre-triage esterne all’ospedale stesso.

Capitolo 3

Mycroft

There's an entire community of developers looking to access this technology, but so far, it's been the purview of a few large companies. The technology is walled-off, proprietary, and secretive.

*Joshua Montgomery
CEO di Mycroft AI, Inc.*

Nel panorama degli assistenti vocali disponibili al pubblico, le grandi aziende catturano la maggior parte delle attenzioni dei consumatori. I nomi di Amazon Alexa, Google Home, Microsoft Cortana sono familiari a tanti. Meno familiare, però, è lo scenario degli stessi strumenti in ambito di assistenti vocali. Piattaforme come *Jarvis*, *Linto*, *Open Assistant* e **Mycroft** sono sconosciute ai più. Quest'ultimo, negli ultimi anni, ha catturato l'attenzione di molti esperti di settore per la sua stabilità, espansibilità, popolarità.

3.1 Open source

3.1.1 Cosa si intende con open source?

Il termine **open source** si riferisce a qualcosa di modificabile e condivisibile dalle persone, avente quindi un **design pubblicamente accessibile**. Il termine nacque nell'ambito dello sviluppo software, ma oggi rappresenta, piuttosto, un'etica. Il software open source è software il cui codice sorgente è ispezionabile, modificabile, e migliorabile da chiunque voglia farlo. Con *codice sorgente* intendiamo il codice che definisce il comportamento del programma, ossia il prodotto del lavoro di un programmatore.

Molti dei software che oggi utilizziamo sono invece **closed source** (o *proprietario*), ossia software il cui codice sorgente non è accessibile agli utenti, e sul quale l'organizzazione che crea il software ha *pieni poteri*. Alcuni esempi di software proprietario possono essere Windows, Photoshop, Safari. Solitamente, durante l'installazione di uno di questi software,

l’utente accetta dei termini e condizioni che lo vincolano a non fare nulla di non autorizzato dall’azienda creatrice del software.

3.1.2 Perché open source?

Le motivazioni per preferire software open source rispetto a quelli proprietari sono svariate. Sebbene i software proprietari spesso abbiano funzionalità più avanzate e, spesso, meglio sviluppate, hanno gravi mancanze in progetti come questo. Il software open source, a scapito della possibilità di minori funzionalità (nonostante ciò non sia sempre vero, basti pensare alla potenza del kernel Linux), dà ad un utente esperto tante possibilità:

- Possibilità di **controllo** sul software: la modifica e il miglioramento del software è permessa e ben vista. La comunità open source lavora con coesione per produrre software sempre migliore.
- Possibilità di conoscere e studiare i **meccanismi interni**: l’utilizzo di software per fini scientifici richiede la piena conoscenza di come un software funziona. Inoltre, la disponibilità del codice a tutti permette agli studenti e agli interessati di scoprire le logiche che ne permettono il corretto funzionamento.
- Grandi garanzie di **sicurezza**: se tutti possono accedere al codice, la presenza di bug di sicurezza e problematiche di privacy viene rilevata molto più velocemente di come accade nel software proprietario.
- Certeze sulla **privacy**: la società in cui viviamo è sempre più basata sulla vendita di dati personali acquisiti tramite software. L’open source, spesso e volentieri, si oppone a tale tendenza: essendo software della comunità e non di un’azienda che punta all’ottenimento di capitale, nessuno ha interesse nel *monetizzare gli utenti*.
- Supporto della **comunità**: lo sviluppo di software open source è molto supportato dalla comunità informatica. In caso di problematiche o necessità, ci sarà sempre un utente più esperto disposto ad aiutare.
- Correttezza **etica**: software come quello oggetto di questa tesi hanno l’obiettivo di salvare vite. È giusto che tutti possano accedervi, migliorarlo e conoscerlo. La ricerca scientifica dovrebbe essere il più possibile **libera**.

3.2 Introduzione a Mycroft

Mycroft nacque tramite crowdfunding nel 2015 e catturò subito molto interesse da parte della comunità informatica. L’obiettivo era chiaro: opporsi alla sola presenza di software proprietario nel campo degli assistenti vocali. Ricevette presto l’appoggio di molte organizzazioni come la Canonical (promotrice di Ubuntu) e la Mozilla Foundation. La repository su Github di Mycroft conta oggi *4300 stars* in continua crescita.

3.2.1 Stack di funzionamento

Come spiegato nel capitolo 2.2, un assistente vocale si compone principalmente di:

Rilevamento della wake word

L'assistente ha bisogno di rilevare una parola per attivarsi. Gli esempi più famosi sono *Hey, Google* o *Alexa*. Mycroft permette di personalizzare la propria wake word, che all'installazione è semplicemente *Hey, Mycroft*. Il progetto utilizza **Precise**, un wake word listener basato su reti neurali addestrate su esempi audio. Questo componente, totalmente open source, è basato su *pattern sonori*, non sul testo. La caratteristica ne riduce la dipendenza da accenti e linguaggi diversi. Precise offre la possibilità di addestrare il listener su un proprio dataset. Il funzionamento è basato su una singola rete neurale ricorrente, più precisamente una **Gated Recurrent Unit** o GRU.

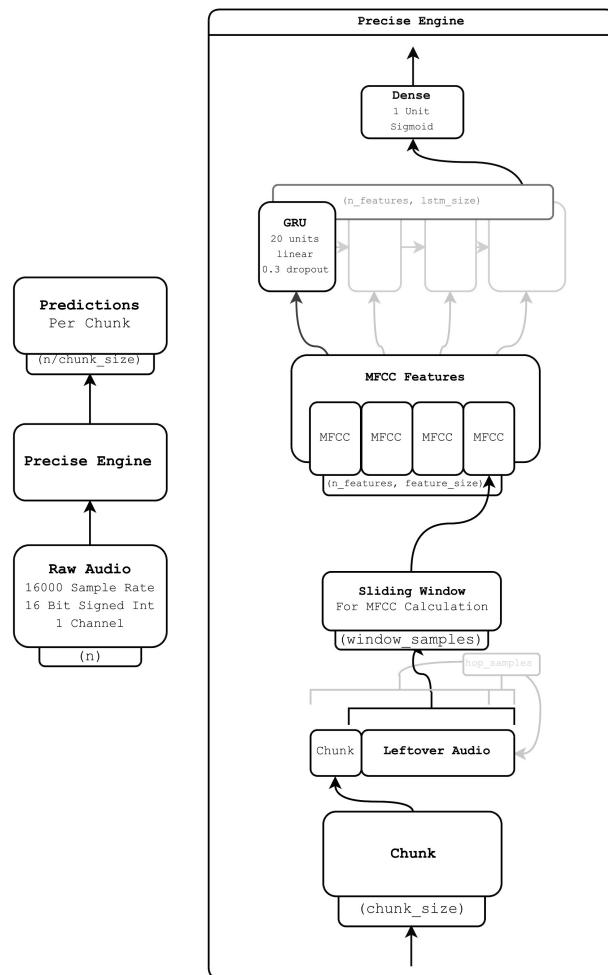


Figura 3.1: Schema di funzionamento di Precise

Speech To Text

Mycroft sfrutta il motore STT di Google per effettuare questo processo. Per aggiungere uno strato di privacy a queste richieste, esse vengono fatte passare attraverso i server Mycroft: Google non può ricollegare una richiesta all'utente che l'ha fatta. In questo modo, tutte le query effettuate a Google contengono solamente l'audio e hanno lo stesso mittente. In futuro sarà possibile sfruttare il dataset di **Mozilla DeepSpeech**, che non è, ad oggi, ancora sufficientemente affidabile.

Interpretazione degli intent

Nell'ambito della speech recognition, un intent è l'operazione che l'utente *intende* svolgere. Un utente può richiedere un'operazione in modi molto diversi. Lo scopo dell'intent parser è proprio quello di riuscire a superare questo scoglio, estraendo dalla richiesta scritta gli elementi chiave. Per esempio, se avessimo una richiesta del tipo "Hey Mycroft, domani pioverà a Parma?", l'intent parser dovrebbe capire:

- L'utente desidera conoscere il **tempo atmosferico**.
- Il tempo atmosferico deve essere cercato per **Parma**.
- La data di interesse è **domani**.

Mycroft rende disponibili due software per la rilevazione di intenti: Padatious e Adapt. Mentre il secondo è basato sul riconoscimento di parole chiave, ed è quindi molto suscettibile alle richieste, il primo è composto da una rete neurale addestrata su frasi intere. Nel corso del progetto sfrutteremo **Padatious**. Questo motore permette di avere creazione semplice degli intent, quantità di dati relativamente bassa, estrazione delle *entities* semplice, addestramento rapido.

Text To Speech

Il componente di Text To Speech ha il compito di generare una traccia audio partendo dal testo di risposta codificato nella skill. Mycroft rende disponibile il progetto **Mimic**, basato sul software FLITE dell'università Carnegie Mellon. Purtroppo quest'ultimo non è disponibile in italiano, quindi nel corso del progetto utilizzeremo il motore TTS di Google, ottimizzato per l'italiano. Come sempre, le richieste passano attraverso i server Mycroft per ottenere un buon livello di privacy. I testi di risposta sono codificati nei file vocabolario delle skills.

3.2.2 Interfaccia grafica

I dispositivi più avanzati di Mycroft come il *Mark II* forniscono la possibilità di aggiungere interazioni visive con gli utenti. L'interfaccia è gestita da Mycroft-GUI, un componente aggiuntivo del bot che sfrutta i KDE Plasmaoids, oggetti dell'interfaccia grafica KDE Plasma, simili a widget. La tecnologia è basata sul linguaggio QML, che permette una totale libertà nella creazione di interfacce, ma anche dei template standard semplici e rapidi da implementare. QML fa parte dello stack di tecnologie di Qt, una libreria multipiattaforma per lo sviluppo di programmi con interfaccia grafica.

Capitolo 4

Realizzazione di skills

Mycroft permette di espandere le abilità del bot in maniera semplice: è infatti costruito modularmente, ed è in ogni momento possibile espandere o ridurre i moduli attivi [5]. Questi moduli sono detti **skills**. Ogni skill è responsabile di una funzionalità: avremo ad esempio la skill del meteo, la skill che racconta barzellette o la skill che si occupa della domotica. Alla ricezione di una richiesta, Mycroft verifica gli intent e le keywords di ogni skill, attivando quella giusta. La creazione di nuove skills è possibile grazie alla classe Python `MycroftSkill`, tramite la cui espansione possiamo definire i comportamenti della skill. All'installazione di Mycroft verrà reso disponibile il **Mycroft Skills Kit**, una command line interface che permette di eseguire le operazioni fondamentali collegate alle skills Mycroft. Tramite il comando `mycroft-msk create` viene avviata una CLI che, dopo aver richiesto al programmatore alcune informazioni, genera il *boilerplate code* necessario alla realizzazione della skill. Tra le informazioni richieste avremo nome, frasi d'esempio, dialoghi di risposta, descrizioni, etc... Il template viene inoltre fornito di `git`, un sistema di versioning che permette di tenere traccia delle modifiche al codice e semplificare l'interazione tra programmatori.

4.1 Struttura dei file della skill

La struttura delle skills è ben organizzata, e vi sono diverse tipologie di file. Oltre a licenza e README, abbiamo tre componenti fondamentali per il funzionamento.

Directory locale Questa directory contiene i file dipendenti dalla lingua. Le sottocartelle seguono i tag linguistici IETF, quindi l'italiano sarà rappresentato da `it-it` e l'inglese (americano) da `en-us`. All'interno di queste, possiamo incontrare quattro tipologie di file:

- **dialog**: questi file contengono i dialoghi di risposta del bot. Sono presenti più frasi simili, in modo da avere interazioni più "umane" con il bot, che sceglie una frase casuale tra queste. Ad esempio, `get_age.dialog` conterrà le frasi *"Mi potresti dire quanti anni hai?", "Per favore, dimmi la tua età", "Qual è la tua età?"...*

- **intent**: questi file contengono le frasi utilizzate per addestrare la rete neurale di Padatious. Ovviamente, più esempi vengono fatti, meglio la rete funzionerà. Per esempio, il file `symptoms.bleeding.intent` conterrà frasi come *"Ho un'emorragia"*, *"Sto sanguinando molto"*, *"Ho una grande perdita di sangue"*...
- **voc**: questi file contengono keywords utilizzate in Adapt e nelle domande a risposta chiusa. Siccome il progetto si basa fortemente su Padatious, troviamo solo i file vocabolario per le domande sì/no ed il riconoscimento di numeri. Ad esempio, il file `yes.voc` conterrà *"Sì"*, *"Certo"*, *"Esatto"*...
- **entity**: questi file contengono entità da estrarre negli intent. Ad esempio, l'intent *fratture* estrae l'arto coinvolto nella frattura. Il file `limb.entity` conterrà quindi termini come *"gamba"*, *"pollice"*, *"spalla"*...

settingsmeta.yaml Questo file contiene la definizione di impostazioni modificabili dall'utente sul sito di gestione di Mycroft. Ad esempio, l'utente potrebbe voler impostare i comportamenti di default, il nome del file esportato dal bot, una chiave API per l'integrazione con altri tool.

__init__.py Questo è il file principale della skill. Contiene il codice che definisce i comportamenti della skill. Il funzionamento è basato sull'espansione della classe `MycroftSkill`, che contiene i decorator ed i metodi necessari alla gestione della skill. Il prossimo capitolo si occuperà di definire la realizzazione vera e propria.

4.2 Funzionamento della skill

Tutta la realizzazione della skill è articolata intorno alla classe `MycroftSkill`.

"""Mycroft skill that does a pre-triage on hospital patients.

The skill tries to ask the patient its symptoms, its personal data, and more. Then, it assigns a color code, stating a priority for medical interventions.

"""

```
from mycroft import MycroftSkill, intent_file_handler
import json

class HospitalTriage(MycroftSkill):
    """Main skill class for the triage.

    This is the main skill class (extending MycroftSkill),
    which contains all the operations we need to perform the
    triage.

    Attributes:
        med_record: a dict containing all the patient data.
    """

```

```
def __init__(self):
    MycroftSkill.__init__(self)
    self.med_record = {}
```

4.2.1 Metodi relativi alle sintomatologie

La skill è stata strutturata come segue: ogni tipologia di sintomi equivale ad un intent, ed è quindi gestita da un metodo specifico. In base alla tipologia di sintomo, l'interazione viene poi gestita in modalità diverse: verrà assegnato al paziente un codice di triage, viene confermata al paziente la corretta comprensione del sintomo, viene mostrata sulla GUI l'informazione recepita.Terminate le operazioni specifiche del sintomo, le restanti procedure vengono gestite da decorators: la richiesta dell'anagrafica, dei sintomi minori, della febbre vengono eseguite indipendentemente dal problema riportato dal paziente. Un esempio di metodo relativo alle sintomatologie può essere il seguente:

```
# BREATH
@intent_file_handler('symptoms.breath.intent')
@symptom_handler
@covid_symptom
def handle_breathing(self, message):
    """This function handles a "breathing fatigue" symptom.

    Breathing fatigue is recognized as a red code, and is a
    COVID-compatible symptom.

Args:
    message: the message object returned from Mycroft

GUI: show open mouth emoji
"""
    self.gui.show_text("[emoji rimossa per motivi di stampa]")
    self.med_record["main_symptom"] = "breathing"
    self.med_record["code"] = "red"
    self.speak_dialog('symptoms.breath')
```

Il metodo salva nell'oggetto `med_record` (*scheda clinica*) il sintomo principale, ossia problemi di respirazione, ed il codice relativo, in questo caso il rosso (accesso immediato alle cure). Poi, tramite il metodo `speak_dialog` di `MycroftSkill`, pronuncia uno a caso tra i dialoghi contenuti nel file `symptoms.breath.dialog`, come

- "Capisco, qualche problema di respirazione."
- "Ok, segno nella scheda problemi di respirazione."
- "Capito: problemi di respirazione."

Da qui in poi, le operazioni sono affidate ai metodi chiamati dai decorators.

4.2.2 Decorators utilizzati

Che cos’è un decorator?

Anzitutto, occorre definire il comportamento di un decorator. Esso permette di assegnare responsabilità addizionali ad un oggetto, dinamicamente. Fornisce quindi un’alternativa flessibile al *subclassing*. Definire un decorator in Python significa definire dei comportamenti da eseguire prima e dopo l’esecuzione del metodo vero e proprio.

Applicazione del concetto al programma

Alcune operazioni sono, all’interno del progetto, ricorrenti. Ad esempio, la richiesta di anagrafica o di sintomi minori deve essere fatta in tutti e soli i metodi relativi alla gestione di sintomi. Alcuni di questi (quelli compatibili con il COVID19, come la tosse), avranno bisogno inoltre di approfondimenti per valutare la possibilità che il paziente sia affetto da COVID19, facendogli domande specifiche. Sono stati quindi definiti due decorators aggiuntivi:

symptom_handler Questo metodo, dopo aver gestito le operazioni specifiche di un sintomo, prosegue con le procedure generali, come la richiesta dell’anagrafica, di una valutazione del dolore, di altri sintomi, e, per ultima, l’esportazione della cartella clinica.

```
def symptom_handler(handler):
    """Decorates a symptom with the needed operations.

    This function is used as a decorator for symptoms, adding
    operations like personal data asking, age, other symptoms...

    Returns:
        The decorator function
    """
    def ask_about_symptoms(*args, **kwargs):
        returned = handler(*args, **kwargs)
        args[0].med_record["symptom_declaration"] = args[1].data["utterance"]
        # I'm using args[0] here instead of self, but it works the same
        args[0].request_age()
        args[0].request_other_symptoms()
        args[0].evaluate_pain()
        args[0].request_name()
        args[0].export_med_record()
        return returned
    return ask_about_symptoms
```

covid_symptom La situazione di emergenza che il mondo sta vivendo al momento della stesura di questa tesi ha richiesto particolare attenzione verso le sintomatologie compatibili con COVID19: distanziare i pazienti che potrebbero essere portatori di virus al più presto è la **sfida più grande del momento**. Per questo, il decorator, aggiunto ai soli sintomi

compatibili con la malattia, permette di interrogare il paziente riguardo alcune caratteristiche tipiche della malattia, come il fiato corto, la difficoltà a percepire i sapori, la febbre alta.

```
def covid_symptom(handler):
    """Decorates a COVID-compatible symptom.

    This function is used as a decorator in the COVID-compatible
    symptoms. It proceeds to ask the COVID-related questions
    to the patient.

    Returns:
        The decorator function
    """
    def check_if_covid(*args, **kwargs):
        returned = handler(*args, **kwargs)
        args[0].ask_covid_questions()
        return returned
    return check_if_covid
```

Il metodo `ask_covid_questions` procede ad intervistare il paziente con il modello di diagnosi più usato al momento. Ogni domanda posta al paziente ha un suo *score* caratteristico, che moltiplica un indice relativo al paziente in base alla gravità ed alla connessione con la malattia. Le caratteristiche indagate sono:

- **Febbre**: moltiplicatore 2
- **Mal di gola**: moltiplicatore 1.3
- **Raffreddore**: moltiplicatore 1.3
- **Fatica respiratoria**: moltiplicatore 1.6
- **Tosse**: moltiplicatore 1.6
- **Contatti con infetti**: moltiplicatore 2
- **Mancanza di gusto**: moltiplicatore 1.7

Ogni paziente sospetto ha inizialmente un `covid_score` di 1, che viene moltiplicato per i vari indici se il paziente risponde in modo affermativo. Se lo score supera una soglia predeterminata (al momento fissata a 15), è sospetto COVID19 e verrà destinato ad un'area di triage apposita.

```
def ask_covid_questions(self):
    """Checks for COVID symptoms.

    When triggered by a COVID-compatible symptom,
    this function evaluates the patient symptoms to
    try to guess if he/she has COVID19.

    GUI: show face mask emoji
```

```

"""
self.speak_dialog('gotta_check_covid')
covid_score = 1
# Let's check if the patient knows the temperature. Skip if he already declared it.
if not "fever" in self.med_record:
    self.check_fever()
if "fever" in self.med_record:
    if self.med_record["fever"] > 37.5:
        covid_score = covid_score * 2
# Let's define an array of tuples, each containing the yes/no question
# string and its COVID index multiplier
yesno_questions = [("has_sore_throat", 1.3), ("has_cold", 1.3),
                    ("has_breathing_difficulties", 1.6), ("has_cough", 1.6),
                    ("has_had_contacts", 2), ("misses_taste", 1.7)]
self.speak_dialog('will_ask_yesno')
# Check if he/she has COVID-compatible symptoms
for question in yesno_questions:
    self.med_record[question[0]] = self.ask_yesno(question[0])
    if self.med_record[question[0]] == 'yes':
        covid_score = covid_score * question[1]
    self.log.info(covid_score)

self.med_record["covid_score"] = covid_score
if covid_score > 15:
    self.speak_dialog('probably_has_covid')
else:
    self.speak_dialog('doesnt_have_covid')
self.log.info(self.med_record)

```

4.2.3 Helper methods

Sono, oltre ai metodi caratteristici della classe, presenti alcuni *helper methods* utili ad operazioni interne del software. Di seguito elenchiamo i più importanti:

extract_temperature Questo metodo, data una stringa in ingresso rappresentante la temperatura corporea che può essere in vari formati (ad esempio, il paziente potrebbe pronunciare *"Trentasette e mezzo"*, *"Trentasette virgola cinque"*, *"Trentasette punto cinque"*, *"Trentasette e cinque"* indicando sempre la stessa temperatura), la estrae in formato decimale.

```
def extract_temperature(utterance):
    """Extracts the patient temperature from the utterance.
```

This is needed because of the various ways of Mycroft interpreting floating point numbers. Some examples:

- 38 e 1
- 38/1
- 38.1
- 38,1
- 38 1

```
>Returns:  
    The floating point value of the temperature, or  
    None if it is impossible to extract.  
"""  
# Beware: the ' e ' has to be before the simple space!  
possible_separators = ['/','.',' ',',',' e ',' ']  
try:  
    for separator in possible_separators:  
        if separator in utterance:  
            temperature_strings = utterance.split(separator)  
            if temperature_strings[1] == "mezzo":  
                temperature_strings[1] = "5"  
            temperature = int(  
                temperature_strings[0])+float(temperature_strings[1])*0.1  
            return temperature  
    return None  
except TypeError:  
    return None
```

age_validator Questo metodo, fornito al metodo `get_response`, definisce se un'età è valida. In caso negativo, viene chiesta nuovamente.

number_validator Questo metodo verifica se l'input vocale è numerico, distinguendo anche il caso in cui un *sei* venga interpretato come *verbo essere* e non 6.

4.2.4 Oggetto `med_record`

Tutti i dati della procedura di triage confluiscono in un oggetto `med_record`, in cui vengono salvati per l'inserimento nel database ospedaliero. Questo oggetto verrà poi esportato in formato JSON (JavaScript Object Notation) con la possibilità di essere inviato ad un'API REST del sistema esistente.

```
def export_med_record(self):  
    """Exports the data to JSON.  
  
    This function is called at the end of the interaction  
    to export the fetched data from the patient. It then  
    assigns a desk to the patient based on his/her severeness.  
  
    GUI: show hands emoji  
    """  
    with open("med_record.json", "w") as med_record_file:  
        med_record_file.write(json.dumps(self.med_record))  
    self.speak_dialog('thanks_and_bye', {"desk": self.med_record["code"]})  
    self.med_record = {}
```

Questo formato standard permette di essere integrato semplicemente con infrastrutture preesistenti. Per esempio, un paziente affetto da COVID19 potrebbe avere la seguente scheda clinica:

```
{  
    "can_talk": "yes",  
    "main_symptom": "breathing",  
    "code": "red",  
    "fever": 38.7,  
    "has_sore_throat": "yes",  
    "has_cold": "no",  
    "has_breathing_difficulties": "yes",  
    "has_cough": "yes",  
    "has_had_contacts": "yes",  
    "misses_taste": "yes",  
    "covid_score": 22.6304,  
    "symptom_declaration": "ho il fiatone",  
    "age": 21,  
    "other_symptoms": "ho un po' di mal di testa",  
    "pain_index": "8",  
    "full_name": "marco rossi"  
}
```

4.3 Esempio di conversazione

Riportiamo ora un esempio di conversazione realmente avvenuto. La conversazione è ascoltabile nei video disponibili sulla repo GitHub[6].

- **Utente:** *Puoi aiutarmi?*
- **Bot:** *Prima di tutto, è lei ad avere bisogno?*
- **Utente:** *Esatto*
- **Bot:** *Qual è il sintomo principale?*
- **Utente:** *Ho la febbre*
- **Bot:** *Hai la febbre, quindi. Ti sei misurato di recente?*
- **Utente:** *Sì*
- **Bot:** *Per favore, dimmi a quanto hai la febbre*
- **Utente:** *Trentasette e otto*
- **Bot:** *Mi serve qualche informazione aggiuntiva per la diagnosi. Ora ti farò alcune domande a cui rispondere sì o no. Hai male alla gola?*
- **Utente:** *Sì*
- **Bot:** *Soffri di raffreddore?*
- **Utente:** *Sì*
- **Bot:** *Noti un affaticamento quando respiri?*

- **Utente:** *No*
- **Bot:** *Hai la tosse?*
- **Utente:** *Sì*
- **Bot:** *Sei stato a stretto contatto con persone affette dal nuovo Coronavirus?*
- **Utente:** *Sì*
- **Bot:** *Fai fatica a sentire i sapori e gli odori?*
- **Utente:** *Sì*
- **Bot:** *Stiamo chiamando un medico. È possibile che tu sia affetto da COVID19. Mi potresti dire quanti anni hai?*
- **Utente:** *Ventidue*
- **Bot:** *Se hai altri sintomi da segnalare, dimmeli ora*
- **Utente:** *Nient'altro*
- **Bot:** *Come definirebbe il suo dolore da 1 a 10?*
- **Utente:** *Sette*
- **Bot:** *Potresti dirmi il tuo nome completo?*
- **Utente:** *Simone Montali*
- **Bot:** *Il tuo nome è Simone Montali, giusto?*
- **Utente:** *Esatto*
- **Bot:** *Grazie per la pazienza, recati al banco YELLOW*

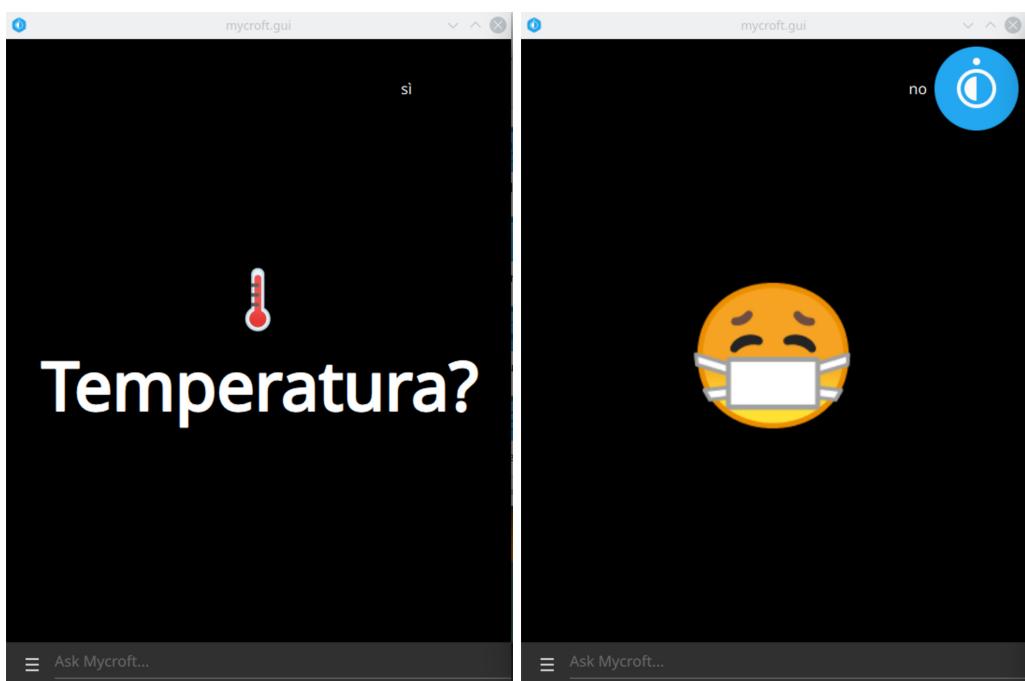


Figura 4.1: Alcune schermate della GUI durante la diagnostica del COVID19

Capitolo 5

Fallback skills e NLP

Sorge, dopo il lavoro svolto fino ad ora, un dubbio: cosa accadrebbe se la richiesta del paziente non fosse tra quelle previste durante la realizzazione della skill? Le necessità mediche delle persone sono, purtroppo, ben più vaste di quanto si possa pensare di programmare. Viene quindi utile un approccio più "statistico", applicabile grazie alle più recenti scoperte nell'ambito dell'intelligenza artificiale.

5.1 Classificazione del testo

Esiste, nell'intelligenza artificiale, una branca che si occupa di **Natural Language Processing**, ossia processamento del linguaggio naturale. Vorremo quindi trovare un *sistema* che approssimi al meglio la corrispondenza tra un input testuale ed un output che indichi una specifica classe. Il processo sarà quindi diviso in più parti: per primo, dovremo trovare un dataset testuale già categorizzato, o categorizzarne manualmente uno. Fatto ciò, dovremo trovare un modo di poter addestrare il computer a riconoscere i *pattern* che legano un determinato testo alla sua classe d'appartenenza.

5.2 Dataset utilizzato

La presenza in rete di siti web come *Kaggle* semplifica nettamente la ricerca di un dataset adatto alle necessità del data analyst. Purtroppo le moderne normative sulla privacy impongono la totale riservatezza dei dati medici delle persone. Esistono però, fortunatamente, dei dataset creati ad-hoc per la risoluzione di problematiche come quella qui presentata. Il dataset sfruttato per questa tesi^[7], creato dalla **Appen**, azienda australiana specializzata in *training data*, contiene audio, trascrizioni e intents di pazienti richiedenti aiuto. Tutto il dataset è in lingua inglese. Siccome la parte di Speech To Text viene gestita da Mycroft, utilizzeremo soltanto il dataset di trascrizioni ed intents. Ad esempio, alcune entries del dataset, potrebbero essere:

phrase:	My son had his lip pierced and it is swollen and the skin inside on his lip is grey and looks infected.
prompt:	Infected wound

phrase:	I used to be out of breath after going up a dozen of stairs, but now I struggle to breath even when I sit down.
prompt:	Hard to breath

5.2.1 Traduzione del dataset

Siccome il bot deve funzionare anche in italiano, risulta evidente la necessità di una traduzione. Potremmo pensare di utilizzare le API di Google Translate, ma la traduzione di tutti i record del dataset (circa 8000), avrebbe un prezzo non indifferente. Si potrebbe quindi risolvere automatizzando l'utilizzo dell'interfaccia web tramite uno strumento come *Selenium*. Fortunatamente, esistono già librerie Python per fare esattamente questo. Tra queste, `googletrans` è la più supportata dalla comunità. Questa permette di definire un oggetto `Translator()` che sfruttiamo per la traduzione:

```
for index, row in data.iterrows():
    phrase = translator.translate(row["phrase"], dest="it").text
    prompt = translator.translate(row["prompt"], dest="it").text
    if translator.detect(phrase).lang == "it" and translator.detect(prompt).lang == "it":
        translated_data.at[i, "phrase"] = phrase
        translated_data.at[i, "prompt"] = prompt
    i = i+1
```

Quando Google Translate non riconosce il significato di una frase, non la traduce e la lascia come l'originale. Notiamo che l'oggetto `translator` permette, oltre alla traduzione, di rilevare la lingua di una stringa. Questa caratteristica è d'aiuto per la realizzazione di un piccolo *workaround* alle mancate traduzioni: se la funzione `detect` non rileva la lingua italiana sia nel testo che nella classe di appartenenza, la riga viene saltata. Questo riduce il numero di entries da circa 8000 a poco più di 2000. Gli esempi prima riportati sono ora:

phrase:	Mio figlio ha avuto il labbro trafitto ed è gonfio e la pelle all'interno del suo labbro è grigia e sembra infetta.
prompt:	Ferita infetta

Notiamo qui che un piercing è diventato un *labbro trafitto*, ma il senso generale è comunque ben chiaro.

phrase:	Ero senza fiato dopo aver salito una dozzina di scale, ma ora faccio fatica a respirare anche quando mi siedo.
prompt:	Difficile da respirare

Anche qui, notiamo un piccolo problema di traduzione: le classi come "*breathing difficulties*" vengono tradotte in modo letterale. Essendo però le classi in numero limitato, possiamo effettuare un *find and replace*, passando, ad esempio, a *Difficoltà respiratorie*.

5.3 Addestramento della rete neurale

Avendo ora un dataset in italiano, possiamo procedere con l’addestramento di una rete neurale che classifichi i testi. Per fare ciò, si è deciso di sfruttare la libreria **fastai**.

5.3.1 fastai

La libreria **fastai** semplifica il training e la creazione di reti neurali tramite le più recenti scoperte nell’ambito del deep learning. Tra le varie interfacce disponibili nella libreria, siamo interessati alla sezione riguardante il testo. Qui, è presente la classe `text.learner`, il cui scopo principale è addestrare un modello tramite il metodo `fit()`. Questa libreria permette inoltre di esportare suddetto modello, allo scopo di integrarlo in applicazioni senza dover effettuare il training ad ogni utilizzo. Esattamente quello che fa al nostro caso: possiamo addestrare la rete neurale una volta sola, ed utilizzare i risultati dell’addestramento a tempo indeterminato nell’assistente vocale, sostituendoli solo quando migliorati.

5.3.2 Preparazione dei dati

Una volta ottenuto il file CSV tradotto, effettuiamo l’importazione su Google Colab e li inseriamo in un dataframe per verificare l’assenza di NaN. Fatto ciò, possiamo creare una `TextList` di *fastai*. Questo oggetto è un’`ItemList` specializzata nel testo, ossia una struttura di memoria che raggruppa gli input per il modello.

```
path = Path('/content/')
data_clas = (TextList.from_csv(path, 'translations.csv',
                               cols='phrase')
              .random_split_by_pct(.2)
              .label_from_df(cols='prompt')
              .databunch(bs=42))
```

Fatto ciò, creiamo l’oggetto `text_classifier_learner`. Questo è basato su una Recurrent Neural Network. Scegliamo, come architettura, **AWD-LSTM**, che si è dimostrata vincente nella risoluzione di problemi di classificazione del testo simili a questo.

AWD-LSTM

Le Recurrent Neural Networks risolvono un problema fondamentale: se avessimo un input di dimensioni variabili, come potremmo creare un modello funzionale? Si potrebbe pensare di utilizzare più reti neurali classiche, ma non esisterebbe interazione tra gli input.

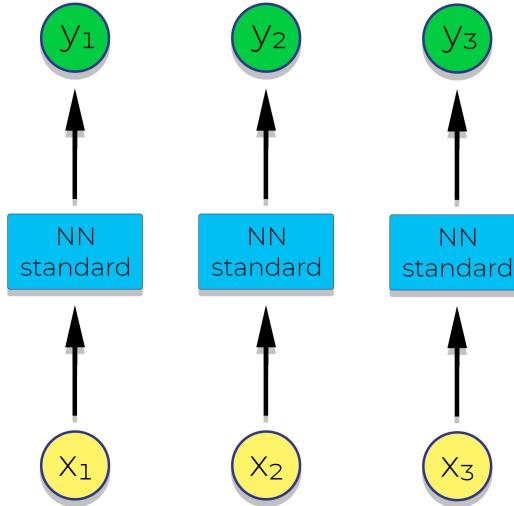


Figura 5.1: Reti neurali standard, chiamate più volte in base alla dimensione dell'input

La soluzione portata dalle RNN è invece utilizzare come input di una rete neurale l'output della precedente. Questa caratteristica porta un vantaggio fondamentale: la rete ha ora *memoria*, e l'output non dipende più solo dall'input singolo, ma da quelli precedenti.

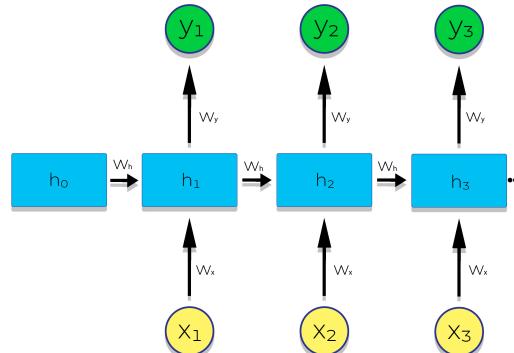


Figura 5.2: Recurrent Neural Network, aperta

Considerando il peso W_h sempre uguale, si ottiene lo schema consueto di RNN.

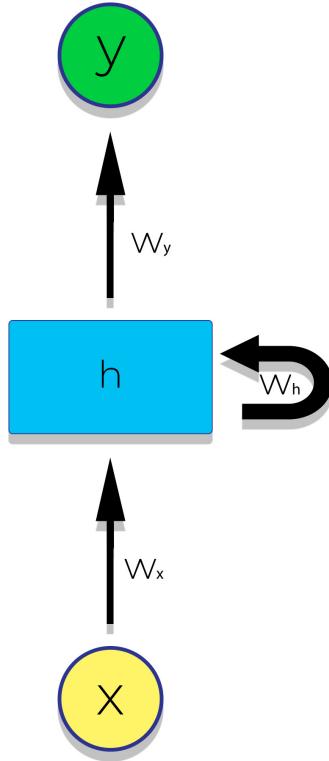


Figura 5.3: Recurrent Neural Network

LSTM sta per *Long Short Term Memory*. Questa tipologia di reti neurali estende il concetto di *memoria a breve termine* delle RNN. A volte, la capacità di queste ultime di avere una *memoria* è molto ridotta: se il gap tra informazioni utili è stretto, la memoria risulta sufficiente, altrimenti no. Le reti LSTM, introdotte da Hochreiter e Schmidhuber nel 1997^[8], risolvono esattamente questo problema. Questo risultato viene raggiunto modificando la struttura interna delle singole reti neurali, più in particolare del modulo *ripetente*, ossia quello che gestisce il passaggio della *memoria* da una cella all'altra. Questa struttura, molto semplice nelle RNN standard (si tratta solamente di una *tanh*), diventa molto più complessa: la cella decide ora quali parti della memoria conservare, e quali sostituire con quelle derivate dal nuovo input. Le strutture di reti LSTM sono svariate, ma solitamente si basano su tre *regolatori*, detti *gates*: l'*input gate*, l'*output gate* ed il *forget gate*. AWD-LSTM (*ASGD Weight-Dropped LSTM*) è una tipologia di reti neurali LSTM che fa utilizzo di *DropConnect* (una particolare tipologia di Dropout che azzera i pesi al posto delle attivazioni) e ASGD (*Average Stochastic Gradient Descent*), una versione dell'algoritmo di SGD che tiene conto di più interazioni piuttosto che di una sola.

5.3.3 Addestramento della rete

Addestriamo quindi la rete, e generiamo una confusion matrix, che permette di verificare quanto errate siano le classificazioni, e quali errori sono più ricorrenti.

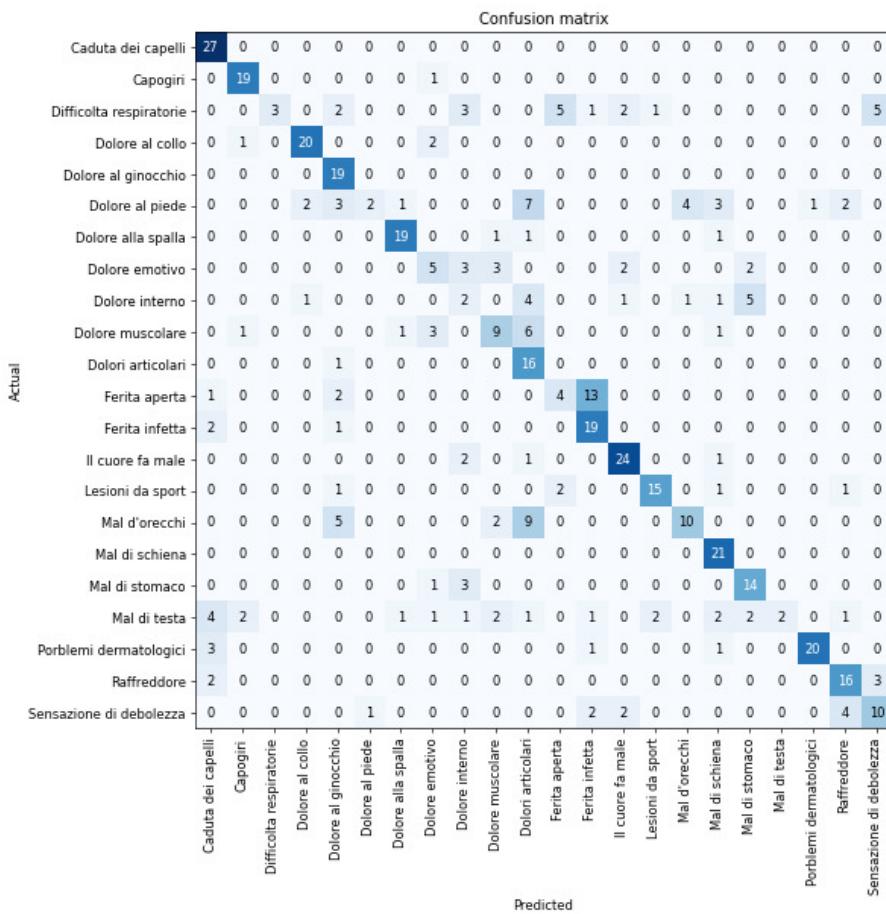


Figura 5.4: Confusion Matrix

Notiamo il formarsi di una diagonale nella tabella: questo è sintomo di un **modello funzionante**. Possiamo quindi ora esportare il modello per l'utilizzo futuro.

```
learn.export('/content/exported_model')
```

Possiamo anche studiare l'accuracy nel corso delle varie epoche di training.

epoch	train_loss	valid_loss	accuracy	time
0	3.261045	3.144287	0.038462	00:33
1	3.216858	3.139450	0.059829	00:34
2	3.184696	3.387876	0.049145	00:31
3	3.161506	3.082859	0.044872	00:32
4	3.141986	3.090852	0.051282	00:32

Figura 5.5: Tabella delle accuracies

Proviamo ora a predire la classe di una frase esempio, come "Sono caduto correndo":

```
int(learn.predict("sono caduto correndo")[0])
```

il risultato ottenuto è 15, ossia la classe "Lesioni da sport".

5.4 Realizzazione di fallback skills

Come precedentemente spiegato, Mycroft funziona riconoscendo determinati *intent* nelle richieste dell'utente. Questi sono stati da noi definiti nella skill del capitolo precedente. Abbiamo però specificato nell'introduzione che, a volte, i casi che abbiamo codificato non sono sufficienti:

1. Svenimenti
2. Emorragie
3. Shock
4. Difficoltà respiratorie
5. Fratture
6. Febbre
7. Ustioni
8. Dolori addominali

Se si trattasse di una categoria esterna a queste, il paziente **non potrebbe venire aiutato**. Esiste però la possibilità di definire delle *fallback skills*, ossia delle skill chiamate dall'assistente vocale quando la frase non viene riconosciuta dalle skill classiche. Queste hanno una priorità, e vengono chiamate nell'ordine definito da questa. Se una fallback skill riesce a risolvere la richiesta, non viene inoltrata alle altre. Vogliamo quindi ora creare un'automazione che sfrutta il modello ricavato dal natural language processing di cui sopra.

5.4.1 Registrazione della skill

La skill estende la classe `FallbackSkill`. Durante l'*init* dell'oggetto, registriamo la fallback con una priorità di 10:

```
self.register_fallback(self.handle_fallback, 10)
```

Fatto ciò, possiamo caricare il modello di classificazione del testo precedentemente generato, assicurandoci che la libreria `fastai` sia installata.

```
# Load the classifier model
self.learner = load_learner('models', 'exported_model')
# Load the classifier classes from JSON
with open('classes.json') as classes:
    self.classes = json.load(classes)
```

Carichiamo inoltre il file `classes.json`, contenente le corrispondenze tra l'output numerico del modello e le classi testuali, oltre alle informazioni riguardanti la compatibilità del sintomo con il COVID19, l'immagine da mostrare nella GUI, e la gravità del sintomo:

```
{  
    "name": "Dolore al collo",  
    "emoji": "[rimosso per motivi di stampa]",  
    "code": "yellow",  
    "covid": false  
}
```

Fatto ciò, resta da definire il comportamento del bot alla chiamata della fallback, operazione che svolgeremo con il metodo `handle_fallback`. Idealmente, questo dovrà predire la classe del sintomo, chiederne conferma al paziente, e aggiungere le informazioni alla scheda di triage.

```
@symptom_handler  
def handle_fallback(self, message):  
    utterance = message.data.get("utterance")  
    symptom = self.classes[int(self.learner.predict(utterance)[0])]  
    self.gui.show_text(symptom["emoji"])  
    did_i_get_that = self.ask_yesno(  
        'symptoms.fallback', {"symptom": symptom["name"]})  
    if did_i_get_that == "no":  
        self.speak_dialog('sorry')  
    else:  
        if symptom["covid"]:  
            self.ask_covid_questions()  
        self.med_record["main_symptom"] = symptom["name"]  
        self.med_record["code"] = symptom["code"]
```

Ora il nostro bot, in caso di mancato riconoscimento del sintomo, potrà comunque indirizzare il paziente in modo automatizzato.

5.5 Esempio di conversazione

Riportiamo ora un esempio di conversazione realmente avvenuto. La conversazione è ascoltabile nei video disponibili sulla repo GitHub[6].

- **Utente:** *Ho bisogno di aiuto*
- **Bot:** *Lei è il paziente, non un accompagnatore, corretto?*
- **Utente:** *Sì*
- **Bot:** *Qual è il sintomo principale?*
- **Utente:** *Mi è uscita la spalla mentre giocavo a golf*
- **Bot:** *Trattasi quindi di dolore alla spalla, dico bene?*

- **Utente:** *Esatto*
- **Bot:** *Mi potresti dire quanti anni hai?*
- **Utente:** *Ventidue*
- **Bot:** *Se c'è qualche altro sintomo che devo sapere, dimmelo ora*
- **Utente:** *Nient'altro*
- **Bot:** *Come definirebbe il suo dolore da 1 a 10?*
- **Utente:** *Sette*
- **Bot:** *Potresti dirmi il tuo nome completo?*
- **Utente:** *Mario Rossi*
- **Bot:** *Il tuo nome è Mario Rossi, giusto?*
- **Utente:** *Esatto*
- **Bot:** *Grazie per la pazienza, recati al banco YELLOW*

Capitolo 6

Richieste di informazioni da parte degli utenti

I casi d'uso di uno strumento come questo assistente vocale potrebbero però **non fermarsi al mero indirizzamento di pazienti**. Infatti, potremmo plausibilmente aspettarci che gli utenti desiderino anche **ottenere informazioni di carattere medico** senza doversi per forza rivolgere ad un dottore.

6.1 Enciclopedia medica del Ministero della Salute

Pare ovvio sottolineare che le fonti delle informazioni riferite ai pazienti debbano essere il **più affidabili possibile**. Affidarsi a siti web come Wikipedia, per quanto comodo possa essere, è rischioso: le informazioni possono essere modificate da chiunque, anche e soprattutto persone poco qualificate. Esiste però, in rete, una risorsa sconosciuta ai più, ma senz'altro interessante per i nostri scopi: l'**Enciclopedia Medica del Ministero della Salute**, accessibile sul sito del suddetto.

6.1.1 Utilizzo dell'enciclopedia

Si potrebbe inizialmente pensare di trascrivere le informazioni dal sito: questo non è però certamente un approccio efficiente. Vorremmo allora automatizzare l'interazione con il sito web, in modo da ottenere i dati in modo autonomo. Viene, in nostro soccorso, il framework **Selenium**.

Selenium

Selenium è un framework nato con lo scopo di testare applicazioni web. È utilizzabile tramite un IDE proprietario, disponibile come estensione per Mozilla Firefox e Google Chrome, ma anche tramite scripts che interagiscono con **Selenium WebDriver**. Quest'ultimo accetta comandi in *Selenese* o tramite una *Client API*, e li trasmette ad un browser che ne raccoglie i risultati. È possibile definire questi scripts di testing in linguaggi diversi dal Selenese: dalla versione 2.0, Selenium WebDriver è totalmente implementato e supportato



Figura 6.1: Enciclopedia della Salute

in Python, Ruby, Java e C#. Durante il progetto abbiamo utilizzato la versione Python, installabile tramite PIP:

```
pip3 install -U selenium
```

Dopo l'installazione di Selenium, è ovviamente necessario anche un driver per interfacciarsi con il browser. Quello per Firefox è ad esempio *GeckoDriver*, quello per Chrome *ChromeDriver*. Ora, definire script per l'automazione di applicazioni web è semplice. Creando un oggetto tramite `webdriver.Firefox()`, avremo la possibilità di inviare comandi al browser ed interagire con la pagina web. Per fare ciò, ci basta selezionare un elemento tramite la sua classe CSS, il suo ID, un XPath: `find_element_by_class_name("nav-tabs")`. Fatto ciò, basta, ad esempio, chiamare il metodo `click()` sull'elemento. È anche possibile eseguire codice JavaScript: sul sito del Ministero della Salute, ad esempio, per disattivare il popup contenente le normative sui cookies sarà sufficiente eseguire `execute_script("setPrivacy();")`.

Approccio API

Se i dati sul sito web cambiassero spesso, sarebbe molto utile definire un'API in modo da poter scaricare i dati al momento della richiesta. Tramite la libreria `http.server` è possibile definire una classe API (che estenda `BaseHTTPRequestHandler`), che ascolti una porta in attesa di richieste ed invii le risposte. Ad esempio, definendo un metodo `do_GET`, esso verrà chiamato ogni qualvolta il server riceva una richiesta di tipo GET sulla porta designata.

Definendo un oggetto DataFetcher che si occupi di gestire Selenium e l’interazione con la pagina, il metodo di risposta alla GET potrebbe essere così composto:

```
def do_GET(self):
    parameters = parse_qs(urlparse(self.path).query)
    response = DataFetcher(parameters["query"][0])
    if response is not None:
        self.send_response(200)
    else:
        self.send_response(404)
    self.send_header('Content-type', 'application/json')
    self.send_header('Access-Control-Allow-Origin', '*')
    self.end_headers()
    if response is not None:
        self.wfile.write(json.dumps(response.get_query()).encode())
```

Nella classe DataFetcher, dopo aver lanciato il WebDriver nell’`__init__`, basterà cercare il termine della query, e scorrere le tabs della pagina dell’enciclopedia. Essa è infatti divisa in sezioni come *"Come si trasmette"*, *"Sintomi"*, *"Descrizione"*. Inviando quindi una richiesta tramite *HTTPie*:

```
$ http "localhost:8000/?query=celiachia"

HTTP/1.0 200 OK
Access-Control-Allow-Origin: *
Content-type: application/json
Date: Fri, 19 Jun 2020 09:49:30 GMT
Server: BaseHTTP/0.6 Python/3.7.6

{
    "Cause": "La celiachia è una condizione multifattoriale,
              [testo accorciato per motivi di stampa]",
    "Complicanze": "I soggetti affetti da celiachia non trattata
                    presentano un rischio maggiore di sviluppare complicanze,
                    [testo accorciato per motivi di stampa]",
    "Descrizione": "La celiachia, o malattia celiachia, è una malattia
                    permanente su base infiammatoria dell'intestino tenue caratterizzata
                    dalla distruzione della mucosa di questo tratto intestinale.
                    [testo accorciato per motivi di stampa]",
    "Diagnosi": "Nei soggetti ad alto rischio di celiachia, per familiarità,
                 sintomi o per la presenza di una malattia frequentemente associata,
                 [testo accorciato per motivi di stampa]",
    "Sintomi e segni": "I sintomi e segni della malattia sono estremamente
                       variabili per sede ed intensità.[testo accorciato per motivi di stampa]",
    "Terapia": "L'unica terapia attualmente disponibile per i soggetti celiaci
               è la completa e permanente [testo accorciato per motivi di stampa]"
}
```

Approccio periodico

Eseguire richieste all'API durante l'interazione con l'assistente vocale può generare **fastidiosi ritardi**. Siccome le informazioni sul sito web di cui sopra **non vengono aggiornate spesso**, può essere conveniente aggiornarle una volta al giorno e tenerle salvate in locale. Questo elimina del tutto i ritardi nel recupero delle informazioni, e ci permette un'integrazione **molto più efficace** in Mycroft: sapendo quali termini medici sono presenti nell'enciclopedia e quali no, il bot saprà già quali riconoscere e quali, invece, restituiranno un errore. Sfruttando come sempre Selenium, possiamo definire uno script utilizzando, in parte, il codice definito precedentemente. Basterà infatti chiamare la funzione `get_query()` ciclicamente, una volta per termine disponibile nell'enciclopedia, ed accorpare tutti i risultati in un *dictionary*, da salvare poi in JSON.

```
def get_all(self):
    self.driver.get(MINISTERO_SALUTE_ENCYCLOPEDIA)
    fetched_data = {}
    aree = self.driver.find_elements_by_class_name("aree")
    for i in range(0, len(aree)):
        # This is kinda esoteric: the references on the DOM get changed on every refresh.
        # So, we have to re-get them every time.
        aree = self.driver.find_elements_by_class_name("aree")
        aree[i].click()
        data = self.get_query()
        if data is not None:
            fetched_data[self.driver.title] = data
    self.driver.get(MINISTERO_SALUTE_ENCYCLOPEDIA)
    return fetched_data
```

Una volta terminata l'esecuzione, ci basterà esportare i dati. Per fare ciò, utilizziamo due file: uno contenente le **informazioni**, uno contenente le **patologie conosciute**, da passare a Mycroft per il training di Padatious:

```
data = df.get_all()
with open("informations.json", "w") as info_file:
    json.dump(data, info_file)
with open("locale/it-it/disease.entity", "w+") as entity_file:
    for name in data:
        entity_file.write(name+"\n")
```

Fatto ciò, basterà definire un intent di Mycroft con frasi come "*Dammi informazioni (sul/sulla/del/della/dei/sui) {disease}*", dove `{disease}` indica un riferimento al file appena definito, contenente le malattie conosciute. È importante sottolineare come queste non siano strettamente richieste: un paziente potrebbe usare delle terminologie diverse. Per questo, vorremmo che il bot cercasse la malattia con **la miglior corrispondenza**, piuttosto che cercarla letteralmente. In questo modo, per esempio, "Tumore ai polmoni" sarebbe comunque ricollegabile a "Tumore del polmone". Una volta riconosciuta la malattia, Mycroft dovrà verificare **quali tipologie di informazioni possiede su di essa**, e chiedere all'utente di scegliere, domandando, ad esempio, "*Cosa ti piacerebbe sapere? Scegli tra Sintomi, Cura, Terapia*".

```

# Let's find the most similar disease to what we've got
key, result = dictionary_searcher(
    message.data.get('disease'), self.informations)
# Tell the user the best match, and if it was wrong, say sorry
did_i_get_that = self.ask_yesno(
    "info.check_results", {"disease": key})
if did_i_get_that == "no":
    self.speak_dialog('sorry')
    return
# Let's create an array containing the choices of infos
self.possibilities = result
choices = ""
for choice in self.possibilities:
    choices = choices + choice + ", "
response = self.get_response(dialog='info.possibilities', data={
    "possibilities": choices})
info_key, info = dictionary_searcher(response, result)
self.speak_dialog(
    'info.speak', {"infos": info})

```

Il bot è ora in grado di tenere conversazioni con i pazienti al riguardo di malattie, patologie, cure:

- *"Cosa sai dirmi sulla celiachia?"*
- *"Ho trovato risultati per celiachia, è ciò che cercavi?"*
- *"Esatto!"*
- *"Cosa ti piacerebbe sapere? Scegli tra cause, complicanze, descrizione, diagnosi, sintomi e segni, terapia."*
- *"Dimmi le cause."*
- *"Ecco ciò che so: la celiachia è una condizione multifattoriale, [testo accorciato per motivi di stampa]"*

6.2 Esempio di conversazione

Riportiamo ora un esempio di conversazione realmente avvenuto. La conversazione è ascoltabile nei video disponibili sulla repo GitHub[6].

- **Utente:** *Cosa sai dirmi sul tumore al fegato?*
- **Bot:** *Il miglior risultato è Tumore del Fegato, è ciò che cercavi?*
- **Utente:** *Esatto*
- **Bot:** *Cosa ti piacerebbe sapere? Scegli tra Descrizione, Segni e sintomi, Cause, Diagnosi, Terapia, Prevenzione*

- **Utente:** *Prevenzione*
- **Bot:** *Questo è ciò che so: le uniche misure per ridurre le probabilità di ammalarsi di cancro al fegato sono l'adozione di stili di vita e misure di igiene che contrastino i fattori di rischio per questa forma tumorale*

6.3 Aggiornamento dei dati

Potrebbe configurarsi la necessità di verificare periodicamente il sito del Ministero della Salute in cerca di correzioni, modifiche, novità. Siccome la piattaforma è eseguita in ambienti Linux, uno straordinario strumento viene in nostro soccorso: **cron**. cron è un job scheduler per sistemi operativi UNIX. La sua struttura semplice permette di definire operazioni periodiche da svolgere in determinati istanti di tempo, all'interno di file detti **crontab** (*cron table*). Ogni linea del crontab indica un'operazione da svolgere e gli istanti di tempo a cui farlo:

```
[min] [ore] [giorno] [mese] [giornodellasettimana] <comando da eseguire>
```

Un asterisco sta ad indicare *a tutti/e i/le ore/minuti/giorni....* È anche possibile definire frazioni di intervalli di tempo: **/2* verrà eseguito ad esempio una volta su due. Se quindi volessimo eseguire il fetching dei dati ogni notte, alle 02:00, basterebbe inserire nel crontab la seguente linea:

```
0 2 * * * cd hospital-triage-skill & python3 info_updater.py
```

I più attenti potrebbero chiedersi perché non lanciare direttamente `python3 hospital-triage-skill/info_updater.py`. Questo comando, apparentemente uguale al precedente, cela una grande differenza: non modifica la *working directory* della linea di comando, che rimane la cartella home dell'utente. Essendo però i file salvati secondo un percorso relativo nel corso dello script Python, apparirebbero direttamente nella cartella home piuttosto che in quella del progetto.

Capitolo 7

Applicabilità e requisiti tecnici

Resta ora da analizzare la fattibilità del progetto nel mondo reale. Questo assistente vocale potrebbe **migliorare ampiamente il workflow** di molte cliniche ospedaliere, anche all'estero. Infatti, eseguire la traduzione in un'altra lingua è un lavoro svolgibile da chiunque, trattandosi della mera traduzione di stringhe. Si potrebbe, addirittura, pensare di integrare il modulo Python `googletrans` citato in 5.2.1, rendendo il progetto poliglotta.

7.1 Requisiti tecnici

Analizziamo i vari componenti dell'assistente vocale per comprenderne i requisiti tecnici:

7.1.1 Text To Speech e Speech To Text

Queste operazioni vengono svolte in cloud tramite le API di Google, e richiedono quindi **soltamente una connessione ad internet**. Ovviamente, si configura la necessità di periferiche audio per la cattura e la riproduzione di audio. Un **microfono universale** ed un **set di altoparlanti** possono sopperire a tale necessità.

7.1.2 Analisi degli intent

L'assistente vocale procede all'analisi degli intent tramite Padatious e Adapt. Mentre il secondo si basa sul riconoscimento di hotwords, e non ha quindi grandi necessità in termini di calcolo, il primo è composto da una rete neurale che viene addestrata ad ogni avvio del bot. Questa è probabilmente l'**operazione che richiede più potenza** di calcolo. Tuttavia, essendo svolta al solo avvio del bot, basterebbe pensare di avviare il bot **qualche minuto prima** dell'effettiva necessità di utilizzo per risolvere questa problematica.

7.1.3 fastai

Essendo il training del modello svolto a priori, non avremo grandi necessità di potenza di calcolo: l'operazione di *inference* è **molto più leggera** dell'addestramento. È suggeribile quindi eseguire quest'ultimo su un computer più potente o direttamente in cloud, con

strumenti come Google Colab, per poi **esportare il modello** nel formato di fastai tramite il comando `learn.export('/content/exported_model')`.

7.1.4 Interfaccia grafica

Il software **Mycroft GUI** fa un pesante utilizzo del framework Qt, più specificatamente dei KDE Plasmoids, componenti grafici (simili a widgets) che caratterizzano l'ambiente desktop KDE Plasma. Per questo, risulta ovvio che il dispositivo su cui viene eseguito l'assistente vocale debba essere compatibile con il suddetto. Il team di Mycroft suggerisce, come sistema operativo, **KDE neon**, consistente in una versione di Ubuntu LTS con installato l'ambiente KDE Plasma. Questo nome garantisce un'ottima stabilità, semplicità d'uso, compatibilità.

7.2 Compatibilità, dispositivi ottimali e costi

La sezione precedente ha evidenziato come i requisiti tecnici dell'assistente vocale siano compatibili con una **vastissima gamma di dispositivi**, anche economici. Questo potrebbe **sgravare un'ulteriore spesa** ad un sistema ospedaliero vittima, da anni, di tagli e budget stringenti. Il software potrebbe infatti essere installato su un vecchio computer, o su un Raspberry Pi.



Figura 7.1: Raspberry Pi con display originale

Creando un setup con quest'ultimo, altoparlanti, microfono ed un display da 7", la spesa sarebbe di circa **cento euro alla data odierna**. Questo costo è senz'altro più basso di

tanti dispositivi enterprise proposti ogni giorno agli ospedali, oltre a garantire **tutela dei dati, apertura delle logiche** di funzionamento (open source), **semplicità** di utilizzo. Un altro dispositivo molto interessante per questo progetto è il **Mycroft Mark II**, sviluppato direttamente dai creatori dell'assistente vocale, integrante tecnologie di cancellazione del rumore, un display per l'interfaccia grafica, altoparlanti e kernel Linux, con possibilità di accesso SSH per la configurazione e la codifica di nuove skills. Quest'ultimo costerà, al suo lancio, poco più di cento euro, e rappresenta l'alternativa perfetta ad un sistema autocostruito, probabilmente troppo sperimentale per un ospedale. La **privacy** è in ogni caso **garantita** dal team.



Figura 7.2: Mycroft Mark II

The Mycroft Mark II smart speaker is an open solution for individuals and companies who want to deploy voice technology, but don't want to be in orbit around Silicon Valley. Our technology can be run on premises and provides a great voice experience without sacrificing privacy.

Capitolo 8

Conclusioni

La pandemia globale sviluppatasi nel corso dell’anno 2020 ha messo in luce come gli ospedali, negli ultimi anni, siano stati man mano dimenticati dalla politica e dai budget. Il più recente rapporto sullo stato del SSN, *Annuario Statistico del Servizio Sanitario Nazionale*, è stato pubblicato a settembre 2019 e contiene dati riferiti al 2017. Tra il 1998 e il 2017, il numero degli istituti di cura è **diminuito di circa 400 unità su 1400**. Il numero di posti letto, passato **da 311 mila a 191 mila**. La spesa pubblica per la sanità, è però costantemente aumentata, passando da poco meno di 60 miliardi di euro, a più di 112. [9] Pare ovvia la necessità di una riforma del servizio sanitario pubblico, e la tecnologia potrebbe, in questa, assumere un ruolo chiave. L’utilizzo di strumenti informatici allo stato dell’arte in ambito ospedaliero ridurrebbe le spese, migliorerebbe le performance, e permetterebbe di curare più pazienti, in modo migliore. Spesso la popolazione guarda verso la tecnologia con malafidanza: per questo, la necessità è rendere questo passaggio il più impercettibile ed *umano* possibile. Strumenti come gli assistenti vocali, le app mobili, la realtà aumentata possono contribuire ad appiattire il *digital divide*, rendendo i servizi fruibili anche alle fasce d’età più restie a questi avanzamenti della società, spesso le più frequenti utilizzatrici dei servizi sanitari. La barriera abbattuta non sarebbe solo quella generazionale, ma **anche quella linguistica**: strumenti come questo permetterebbero anche a chi non conosce bene l’italiano di interagire nella propria lingua madre e riuscire a capire e farsi capire. La ri-dotta complessità di questo progetto è la prova che spesso le migliori necessarie sono ben più vicine di quanto sembrino. Il lavoro è *diviso* in due parti: una basata sulle reti neurali di **Padatious**, allenate con alcune frasi per sintomo, ed una basata su un classificatore testuale, allenato con un dataset apposito. Ognuna delle due ha, senz’altro, possibilità di evoluzione: il miglior risultato si otterrebbe accorpandole ed affidandosi totalmente ad un classificatore esterno a Mycroft. Questo richiederebbe però un dataset di training **molto più completo e numeroso**, e per questo la decisione di affidarsi a **due livelli** di skills si è rivelata la più affidabile. Ciò non toglie che, con lo sviluppo di nuovi dataset nel futuro, sarà un giorno possibile definire un progetto come questo ancora più semplicemente. Inoltre, la possibilità di installare questo assistente vocale su dispositivi a **basso costo** piuttosto che appositi totem high-end, rende abbordabile il passaggio ad un pre-triage automatizzato a quasi tutti gli ospedali e cliniche italiani. Tutto il codice scritto per la tesi è **open source con licenza GPL-3.0**. Il codice stesso del documento *LATEX* di questa tesi è open source. Questo permette l’analisi, da parte di chiunque, del codice per il riscontro di

bugs e problemi. Ne permette l’espansione, l’utilizzo gratuito, la sperimentazione da parte dei programmatore e degli ospedali. Infine, è corretto e necessario:

la libertà, delle persone come del software, è **l’unica via per il progresso.**

Bibliografia

- [1] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997, pp. 81–88. ISBN: 9780071154673. URL: <https://books.google.it/books?id=EoYBngEACAAJ>.
- [2] Matthew Hoy. «Alexa, Siri, Cortana, and More: An Introduction to Voice Assistants». In: *Medical Reference Services Quarterly* 37 (gen. 2018), pp. 81–88. DOI: [10.1080/02763869.2018.1404391](https://doi.org/10.1080/02763869.2018.1404391).
- [3] Voicebot - Orbita. «Voice Assistant Consumer Adoption in Healthcare». In: (ott. 2019). URL: https://voicebot.ai/wp-content/uploads/2019/10/voice_assistant_consumer_adoption_in_healthcare_report_voicebot.pdf.
- [4] Piers Ford. «Digital triage changes the game for UK GP practices in the age of COVID-19». In: (mag. 2020). URL: <https://www.mobihealthnews.com/news/europe/digital-triage-changes-game-uk-gp-practices-age-covid-19>.
- [5] Mycroft AI. *Mycroft AI Documentation*. 2019. URL: <https://mycroft-ai.gitbook.io/docs/>.
- [6] Simone Montali. *Video dell'interazione con il bot*. 2020. URL: <https://github.com/montali/tesi/tree/master/Resources/videos>.
- [7] Paul Mooney - Appen. *Medical Speech, Transcription, and Intent*. 2019. URL: <https://appen.com/datasets/audio-recording-and-transcription-for-medical-scenarios/>.
- [8] Sepp Hochreiter e Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780.
- [9] Agenzia Giornalistica Italia. «La riduzione di ospedali e posti letto negli ultimi 10 anni». In: (mar. 2020). URL: <https://www.agi.it/fact-checking/news/2020-03-14/coronavirus-rianimazione-posti-letto-7500889/>.

,