Documentazione

Progetto Brigid AI Chatbot

Catello Martone, Davide Viola, Gabriella Fede, Pasquale Anatriello

7 febbraio 2025

Prof. Fabio Palomba, Corso di Fondamenti di Intelligenza Artificiale
Università degli Studi di Salerno, A.A. 2024/2025

Indice

1 Introduzione

1.1 L'obiettivo della chatbot Brigid

La chatbot sviluppata ha l'obiettivo di fornire un supporto interattivo e intelligente agli utenti, con particolare attenzione all'assistenza psicologica e al dialogo empatico. Grazie all'integrazione di modelli NLP avanzati, come BERT per la comprensione del linguaggio naturale e un classificatore basato su reti neurali, il sistema è in grado di riconoscere gli intenti degli utenti e fornire risposte pertinenti. Inoltre, l'uso di un modello generativo come Gemini consente di gestire input complessi o non previsti, garantendo maggiore flessibilità nella conversazione. La chatbot è progettata per offrire interazioni fluide e contestualizzate, migliorando l'esperienza utente attraverso l'adattamento dinamico delle risposte. L'obiettivo finale è creare un assistente virtuale affidabile, in grado di comprendere e rispondere in modo accurato, favorendo un'interazione naturale ed efficace.

1.2 Scelta nome *Brigid*

Il nome Brigid è stato scelto per la chatbot in riferimento alla figura mitologica e simbolica di Brigid, una divinità celtica associata alla saggezza, alla guarigione e alla poesia. Questa scelta riflette l'intento della chatbot di offrire non solo assistenza e supporto agli utenti, ma anche un dialogo empatico e costruttivo. Brigid è storicamente venerata come un simbolo di protezione e conoscenza, caratteristiche che si allineano con il ruolo della chatbot nel fornire risposte intelligenti e un'interazione rassicurante. Il nome vuole evocare un senso di fiducia e affidabilità, sottolineando la capacità del sistema di comprendere e rispondere alle esigenze dell'utente in modo sensibile ed efficace.

1.3 Metodologia CRISP-DM e Applicazione nel Progetto

Il CRISP-DM (Cross Industry Standard Process for Data Mining) è un modello di sviluppo standardizzato utilizzato nell'ambito dell'analisi dei dati e dell'intelligenza artificiale. Si compone di sei fasi iterative: comprensione del business, comprensione dei dati, preparazione dei dati, modellazione, valutazione e deployment. Nel contesto del nostro progetto, questa metodologia è stata applicata per garantire uno sviluppo strutturato ed efficace della chatbot Brigid.

- **Comprensione del business**: Abbiamo definito gli obiettivi principali della chatbot, ossia fornire assistenza e supporto agli utenti attraverso un'interazione intelligente e contestualizzata.
- **Comprensione dei dati**: È stata analizzata la tipologia di input attesi dagli utenti e strutturato un dataset di intenti (`intents.json`), utile per il riconoscimento delle richieste.
- **Preparazione dei dati**: Sono stati implementati processi di preprocessing NLP per migliorare la qualità dell'input, tra cui tokenizzazione con BERT e pulizia del testo.
- **Modellazione**: È stato sviluppato un modello di classificazione basato su reti neurali con Keras, addestrato sugli intenti, e affiancato da un modello generativo (Gemini) per la gestione di input non previsti.

- **Valutazione**: Il sistema è stato testato su frasi reali e sintetiche, con un'analisi della confidence e metriche di accuratezza per garantire prestazioni affidabili.
- **Deployment**: La chatbot è stato implementato in una modalità di interazione basata esclusivamente su terminale, permettendo agli utenti di avviare una conversazione direttamente tramite riga di comando. Questa scelta semplifica l'accessibilità e facilita i test, consentendo miglioramenti futuri per un'eventuale integrazione in interfacce più avanzate.

L'approccio CRISP-DM ha permesso di sviluppare Brigid in modo metodico, migliorando progressivamente il modello sulla base delle esigenze reali degli utenti e mantenendo una struttura flessibile per future estensioni.

2 Comprensione del Business

La fase di comprensione del business all'interno della metodologia CRISP-DM è fondamentale per definire gli obiettivi del progetto e garantire che il sistema sviluppato soddisfi le esigenze degli utenti finali. In questa sezione vengono analizzati il contesto, gli obiettivi e i requisiti della chatbot Brigid, con un focus sull'impatto che avrà nell'ambito di utilizzo previsto.

2.1 Contesto e Motivazioni

L'idea alla base dello sviluppo di Brigid nasce dalla necessità di creare un assistente virtuale capace di supportare gli utenti attraverso un'interazione naturale e intelligente. La chatbot è stato progettato per offrire risposte pertinenti, basandosi su un modello di classificazione degli intenti e sull'uso di tecnologie avanzate di elaborazione del linguaggio naturale (NLP).

2.2 Obiettivi della Chatbot

Gli obiettivi principali della chatbot Brigid sono:

- Fornire assistenza e supporto agli utenti tramite una conversazione fluida e intuitiva.
- Comprendere il linguaggio naturale utilizzando modelli avanzati come BERT per l'analisi semantica dei messaggi.
- Predire gli intenti dell'utente con un modello di classificazione basato su una rete neurale, addestrato su un dataset di intenti (`intents.json`).
- Gestire input imprevisti grazie all'integrazione con un modello generativo (Gemini), migliorando la capacità di risposta in situazioni non coperte dal dataset.
- Mantenere una struttura modulare e flessibile, con possibilità di espansione futura in interfacce più avanzate.

2.3 Requisiti del Progetto

Per garantire il raggiungimento degli obiettivi sopra elencati, sono stati definiti i seguenti requisiti tecnici e funzionali:

2.3.1 Requisiti Funzionali

Codice	Descrizione
RF_1	La chatbot deve essere in grado di riconoscere e classificare correttamente gli intenti dell'utente.
RF_2	Deve fornire risposte pertinenti in base al contesto della conversazione.
RF_3	Deve gestire input non previsti e rispondere in modo adeguato utilizzando un fallback intelligente (Gemini).
RF_4	L'interazione deve avvenire tramite un'interfaccia a riga di comando (CLI).

2.3.2 Requisiti Tecnici

Codice	Descrizione
RT_1	Utilizzo di BERT per la generazione di embedding delle frasi dell'utente.
RT_2	Implementazione di una rete neurale feedforward con Keras per la classificazione degli intenti.
RT_3	Integrazione con Google Gemini per risposte generative in caso di input fuori dal dominio del dataset.
RT_4	Struttura del codice organizzata in moduli (`pretrain.py`, `chatbot_terminal.py`).

2.4 Benefici Attesi

L'implementazione di Brigid porta diversi vantaggi:

- **Maggiore accessibilità**: grazie alla possibilità di avviare la chatbot direttamente dal terminale senza necessità di un'interfaccia grafica.
- **Migliore comprensione delle richieste dell'utente**: grazie all'integrazione con modelli NLP avanzati.
- **Espandibilità e flessibilità**: con la possibilità di migliorare il dataset, affinare il modello e integrare nuove funzionalità in futuro.

Questa fase di comprensione del business ha guidato tutte le decisioni successive, permettendo di sviluppare la chatbot in modo mirato e orientato agli obiettivi definiti.

3 Comprensione dei dati

La fase di comprensione dei dati all'interno della metodologia CRISP-DM è essenziale per analizzare e strutturare le informazioni su cui il modello si basa. In questa sezione viene descritto il dataset utilizzato per addestrare la chatbot Brigid, il formato dei dati, le categorie di intenti e le loro caratteristiche.

3.1 Struttura del Dataset

Il dataset utilizzato per l'addestramento della chatbot è contenuto nel file `intents.json` e segue un formato standard per chatbot basati su intenti. Ogni intento è definito da:

- Un **tag univoco** che identifica la categoria dell'intento.
- Un insieme di **pattern** (frasi di esempio) che rappresentano possibili input dell'utente per quell'intento.
- Una lista di **risposte predefinite** che la chatbot può fornire se viene rilevato un determinato intento.

3.1.1 Esempio di Struttura di un Intento

```
{
    "intents": [
    {
        "tag": "greeting",
        "patterns": ["Hello", "Hi there!", "Hey", "Good
            morning"],
        "responses": ["Hello! How can I help you?", "Hi!
            What can I do for you?"]
    },
    {
        "tag": "goodbye",
        "patterns": ["Bye", "See you later", "Goodbye"],
        "responses": ["Goodbye! Have a great day.", "See
            you soon!"]
    }
    ]
}
```

3.2 Tipologie di Intenti

Il dataset contiene una serie di intenti suddivisi in diverse categorie, tra cui:

- **Saluti e chiusura** (es. `greeting`, `goodbye`)
- **Richieste informative** (es. `time`, `weather`, `help`)
- **Supporto psicologico** (es. `sadness`, `anxiety`, `stress_relief`)

- **Conversazioni generiche** (es. `smalltalk`, `chitchat`)

L'obiettivo è garantire che la chatbot sia in grado di riconoscere correttamente le intenzioni dell'utente e rispondere in modo pertinente.

3.3 Analisi della Distribuzione dei Dati

Per assicurare un buon addestramento del modello, è stata effettuata un'analisi della distribuzione degli intenti all'interno del dataset:

- **Bilanciamento**: Un dataset equilibrato evita che il modello sia troppo incline a predire alcuni intenti rispetto ad altri.
- **Varietà dei pattern**: Ogni intento deve avere un numero sufficiente di frasi di esempio per garantire una generalizzazione efficace.
- **Gestione di sinonimi e variazioni linguistiche**: Frasi diverse con lo stesso significato devono essere incluse per migliorare la capacità del modello di riconoscere input variati.

Se il dataset risultasse sbilanciato, potrebbero essere necessarie tecniche di *data augmentation*, come la generazione di frasi alternative tramite modelli generativi (es. `GPT`) o l'uso di sinonimi.

3.4 Tipologia di Dati Utilizzati

I dati presenti nel dataset sono **dati testuali non strutturati**, che vengono elaborati e trasformati in vettori numerici tramite `BERT`. Le fasi di elaborazione comprendono:

- **Tokenizzazione e conversione in embedding**: Il testo viene trasformato in rappresentazioni numeriche (vettori).
- **Assegnazione delle etichette**: Ogni frase viene associata al suo intento corrispondente.
- **Training del modello**: Il modello viene addestrato sui dati preprocessati per imparare a classificare gli intenti.

3.5 Problematiche del Dataset Iniziale

Il dataset iniziale, recuperato da Kaggle, conteneva 75 intenti, un numero limitato per coprire una vasta gamma di interazioni utente. Questo ha presentato diverse problematiche che avrebbero potuto compromettere le prestazioni della chatbot Brigid:

- **Copertura insufficiente degli intenti**: Il numero ridotto di intenti limitava la capacità della chatbot di rispondere efficacemente a un'ampia varietà di domande.
- **Scarso numero di pattern per intento**: Ogni intento aveva pochi esempi di frasi (*pattern*), il che poteva portare il modello a non riconoscere correttamente input leggermente diversi da quelli presenti nel dataset.

- **Rischio di overfitting**: Un dataset con pochi esempi per ogni classe può portare il modello ad adattarsi troppo ai dati di training, risultando poco flessibile su input reali.
- **Mancanza di diversità nelle risposte**: Il numero limitato di risposte per ogni intento poteva rendere le interazioni con la chatbot ripetitive e meno naturali.

Per risolvere queste problematiche, è stato necessario espandere e migliorare il dataset attraverso l'applicazione di tecniche di *data augmentation*.

3.6 Applicazione della Data Augmentation e Benefici

Per migliorare la qualità del dataset e garantire un addestramento più efficace della chatbot, è stato applicato un processo di **data augmentation**, ossia la generazione artificiale di nuovi dati per arricchire il dataset esistente.

3.6.1 Espansione del Dataset

- Il numero di intenti è stato ampliato da **75 a 223**, aumentando la varietà delle interazioni possibili.
- Ogni intento è stato arricchito con una media di **50 pattern di input**, aumentando la capacità del modello di riconoscere input con diverse formulazioni.
- Ogni intento è stato dotato di **10 risposte fisse**, garantendo maggiore diversità nelle risposte generate.

3.6.2 Ottimizzazione dei Pattern

Per migliorare la classificazione degli intenti, i **pattern di input** sono stati **ridotti in lunghezza**, evitando eccessiva variabilità e garantendo maggiore uniformità. Questa ottimizzazione ha reso il modello più preciso nel riconoscere le richieste degli utenti, riducendo **ambiguità** tra intenti simili. Inoltre, la rimozione di formulazioni ridondanti ha reso l'**addestramento più efficiente**, limitando il rischio di **sovrapposizione tra classi** e migliorando la capacità di generalizzazione della chatbot.

3.6.3 Generazione di Nuovi Pattern con ChatGPT

Per **arricchire il dataset** e migliorare la capacità della chatbot di gestire input variabili, è stato impiegato **ChatGPT** per la generazione di nuovi **pattern naturali e realistici**. Questo approccio ha ampliato la **diversità delle espressioni linguistiche**, rendendo le interazioni più fluide e l'esperienza utente più naturale. L'automazione ha inoltre consentito un'espansione **rapida ed efficace** del dataset, migliorando la copertura delle possibili varianti linguistiche utilizzate dagli utenti.

3.6.4 Benefici della Data Augmentation

L'applicazione della *data augmentation* ha portato diversi vantaggi:

- **Migliore generalizzazione** del modello, riducendo il rischio di overfitting.

- **Aumento della robustezza** nella comprensione di input variati.
- **Miglior bilanciamento del dataset**, evitando intenti con pochi esempi.
- **Maggiore adattabilità** a input reali, grazie alla varietà dei pattern generati.

Grazie a queste ottimizzazioni, la chatbot Brigid è ora in grado di comprendere e rispondere con maggiore accuratezza e naturalezza agli input degli utenti.

4 Preparazione dei Dati

La fase di preparazione dei dati è fondamentale affinché il chatbot Brigid possa elaborare correttamente gli input testuali e fornire risposte adeguate. Nel nostro progetto, la preparazione dei dati è suddivisa in due fasi principali:

1. Preparazione dei dati di input per il pretrain del modello → Elaborazione del dataset (`intents.json`) per addestrare il modello di classificazione.
2. Preparazione dei dati in input da sottomettere al modello nel terminale → Elaborazione in tempo reale delle frasi fornite dagli utenti per generare risposte pertinenti.

4.1 Preparazione dei Dati per il Pretrain del Modello

Nel file `pretrain.py`, i dati vengono elaborati e trasformati in un formato compatibile con il modello di classificazione. Questo processo è fondamentale affinché la rete neurale possa apprendere in modo efficace e riconoscere gli intenti dell'utente.

4.1.1 Estrazione e Pulizia dei Dati

Il dataset viene caricato da `intents.json` e suddiviso in patterns (frasi di esempio) e tag (categorie di intenti). Ecco il processo seguito:

```
    # Lettura del dataset
    with open('../data/intents.json', encoding='utf-8') as
        file:
    intents = json.load(file)
```

Creazione delle liste per i dati di addestramento:

```
    classes = []
    documents = []

    for intent in intents['intents']:
    for pattern in intent['patterns']:
    documents.append((pattern, intent['tag']))
    if intent['tag'] not in classes:
    classes.append(intent['tag'])

    classes = sorted(set(classes))
```

Viene creata una lista `documents` che associa ogni frase (pattern) al suo intento (tag). La lista `classes` raccoglie tutti gli intenti unici presenti nel dataset.

4.1.2 Tokenizzazione e Generazione degli Embedding

Per permettere al modello di comprendere il significato semantico delle frasi, le frasi di input vengono convertite in vettori numerici (embedding) utilizzando BERT.

Caricamento del tokenizer e del modello BERT:

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
bert_model = BertModel.from_pretrained('bert-base-uncased')
```

Conversione delle frasi in embedding numerici:

```
def get_bert_embedding(sentence):
inputs = tokenizer(sentence, return_tensors="pt", padding=True, truncation=True, max_length=50)
with torch.no_grad():
outputs = bert_model(**inputs)
return outputs.last_hidden_state.mean(dim=1).squeeze().numpy()
```

La funzione `get_bert_embedding()` prende una frase, la tokenizza con BERT e ne estrae l'embedding numerico. L'output è un vettore di 768 dimensioni che rappresenta semanticamente la frase.

Creazione del dataset di addestramento:

```
train_x = []
train_y = []

for pattern, tag in documents:
embedding = get_bert_embedding(pattern)
train_x.append(embedding)

output_row = [0] * len(classes)
output_row[classes.index(tag)] = 1
train_y.append(output_row)

train_x = np.array(train_x)
train_y = np.array(train_y)
```

`train_x` contiene gli embedding BERT delle frasi di esempio. `train_y` contiene le etichette in formato one-hot encoding, necessarie per addestrare il modello di classificazione.

4.1.3 Addestramento del Modello

Una volta elaborati i dati, viene costruito un modello di rete neurale per la classificazione degli intenti.

Definizione del modello:

```
model = Sequential()
model.add(Dense(128, input_shape=(train_x.shape[1],), activation='relu'))
```

```
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(len(train_y[0]), activation='softmax')
        )
```

Rete neurale fully connected con due livelli nascosti e funzioni di attivazione ReLU. Dropout per evitare overfitting. Softmax nell'output per la classificazione degli intenti.

Compilazione e addestramento del modello:

```
    model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
    )
    hist = model.fit(train_x, train_y, epochs=100,
        batch_size=8, verbose=1)
```

Ottimizzazione con Adam e funzione di perdita `categorical_crossentropy`. Addestramento per 100 epoche con batch di 8 esempi per volta.

Salvataggio del modello e delle classi:

```
    model.save('../models/chatbot_model.keras')
    pickle.dump(classes, open('../models/classes.pkl', 'wb
        '))
```

Il modello addestrato viene salvato per essere usato dal chatbot in fase di inferenza. Le classi degli intenti vengono salvate per recuperarle durante la predizione.

4.2 Preparazione dei Dati in Input nel Terminale

Nel file `chatbot_terminal.py`, viene gestita la preparazione degli input ricevuti in tempo reale dagli utenti. L'obiettivo è trasformare ogni messaggio in una forma compatibile con il modello di classificazione.

4.2.1 Pulizia del Testo Utente

Prima di essere analizzato, l'input utente viene preprocessato per migliorare la comprensione da parte del modello.

Passaggi eseguiti nel preprocessing:

```
    def preprocess_sentence(sentence):
    sentence = re.sub(r"(.)\1{2,}", r"\1", sentence)  #
        Riduzione ripetizioni di caratteri
    corrected_sentence = str(TextBlob(sentence).correct())
         # Correzione ortografica
    expanded_sentence = contractions.fix(
        corrected_sentence)  # Espansione delle contrazioni
```

Riduzione di caratteri ripetuti (es. "hellooo" → "hello"). Correzione ortografica con `TextBlob`. Espansione delle contrazioni (es. "I'm" → "I am") per evitare ambiguità.

4.2.2 Tokenizzazione e Generazione degli Embedding in Tempo Reale

L'input preprocessato viene poi trasformato in embedding numerici usando BERT, esattamente come fatto in fase di addestramento.

Tokenizzazione e generazione dell'embedding:

```
    inputs = tokenizer(expanded_sentence, return_tensors="pt", padding=True, truncation=True, max_length=50)
    with torch.no_grad():
    outputs = bert_model(**inputs)
    embedding = outputs.last_hidden_state.mean(dim=1).squeeze().numpy()
    return np.expand_dims(embedding, axis=0)
```

Il testo viene tokenizzato e convertito in embedding BERT. L'embedding (768 dimensioni) viene passato al modello di classificazione per la predizione dell'intento.

Predizione dell'intento:

```
    def predict_intent(sentence):
    bag = preprocess_sentence(sentence)
    res = model.predict(bag)[0]
    max_prob = np.max(res)
    predicted_class = np.argmax(res)
    return predicted_class, max_prob
```

Il modello predice l'intento più probabile con il relativo confidence score.

La fase di preparazione dei dati garantisce che il chatbot Brigid possa interpretare correttamente sia i dati di addestramento che gli input in tempo reale.

Nel pretrain, il dataset viene elaborato e trasformato in embedding per addestrare la rete neurale. Nel terminale, l'input dell'utente viene normalizzato, tokenizzato e convertito in embedding per ottenere una predizione accurata dell'intento. Questa preparazione ottimizzata permette a Brigid di offrire risposte pertinenti e di adattarsi a diversi tipi di input utente.

5 Modellazione

La fase di modellazione nel processo CRISP-DM riguarda la selezione, l'implementazione e l'addestramento del modello di machine learning che permette al chatbot Brigid di riconoscere gli intenti dell'utente e fornire risposte appropriate.

Nel progetto, il chatbot utilizza una combinazione di tecniche di Natural Language Processing (NLP) e Deep Learning per l'elaborazione del linguaggio naturale. Il modello di classificazione sfrutta gli embedding BERT per rappresentare le frasi e una rete neurale fully connected per associare ogni input utente a un intento specifico. Inoltre, viene utilizzato un modello generativo (Gemini) per gestire input non previsti.

5.1 Embedding e Rappresentazione del Testo

Un embedding è una rappresentazione numerica di un testo che cattura le relazioni semantiche tra le parole. Gli embedding permettono al modello di comprendere il significato delle frasi anziché analizzarle come semplici sequenze di caratteri.

Nel chatbot Brigid, gli embedding sono generati utilizzando BERT, un modello avanzato di NLP che produce vettori numerici a 768 dimensioni, rappresentando ogni frase in uno spazio multidimensionale. Questo approccio consente al chatbot di riconoscere somiglianze tra frasi con significati simili, anche se scritte in modi diversi.

Esempio di embedding numerico per una frase:

```
Input: "Hello, how are you?"
Embedding generato: [0.12, -0.34, 0.56, ..., 0.98]
```

Questa rappresentazione vettoriale viene utilizzata come input per la rete neurale del chatbot.

5.2 Architettura del Modello di Classificazione

Il modello di classificazione degli intenti è una rete neurale fully connected realizzata con Keras. Il suo compito è analizzare gli embedding generati da BERT e assegnare un intento all'input dell'utente.

5.2.1 Struttura del modello

Il modello è costituito da tre livelli principali:

- **Input Layer**: accetta gli embedding di BERT (dimensione 768)
- **Hidden Layers**: due livelli densi con attivazione ReLU per apprendere caratteristiche rilevanti
- **Dropout Layers**: riducono il rischio di overfitting
- **Output Layer**: utilizza la funzione di attivazione Softmax per restituire la probabilità di appartenenza a ciascun intento

Il codice che definisce il modello è il seguente:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

model = Sequential([
Dense(128, input_shape=(768,), activation='relu'),
Dropout(0.5),
Dense(64, activation='relu'),
Dropout(0.5),
Dense(len(classes), activation='softmax')
])
```

Dopo la definizione dell'architettura, il modello viene compilato e addestrato con i dati ottenuti dalla fase di preparazione:

```
from keras.optimizers import Adam

model.compile(
```

```
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
    )

    model.fit(train_x, train_y, epochs=100, batch_size=8,
       verbose=1)
```

Viene utilizzato l'ottimizzatore Adam, che accelera la convergenza del modello. La funzione di perdita è categorical crossentropy, adatta alla classificazione multi-classe. La metrica di valutazione è l'accuratezza. Il modello viene addestrato per 100 epoche con un batch di 8 esempi alla volta. Al termine dell'addestramento, il modello viene salvato per essere utilizzato in fase di inferenza:

```
    model.save('../models/chatbot_model.keras')
    import pickle
    pickle.dump(classes, open('../models/classes.pkl', 'wb
       '))
```

5.3 Predizione degli Intenti e Generazione delle Risposte

Una volta addestrato, il modello viene utilizzato nel file `chatbot_terminal.py` per analizzare gli input dell'utente e generare una risposta appropriata.

5.3.1 Preprocessing dell'Input Utente

Prima di essere elaborato, l'input dell'utente viene sottoposto a un processo di pulizia e trasformato in embedding BERT.

```
    def preprocess_sentence(sentence):
    inputs = tokenizer(sentence, return_tensors="pt",
       padding=True, truncation=True, max_length=50)
    with torch.no_grad():
    outputs = bert_model(**inputs)
    embedding = outputs.last_hidden_state.mean(dim=1).
       squeeze().numpy()
    return np.expand_dims(embedding, axis=0)
```

Il testo viene tokenizzato con il BERT tokenizer. L'output è un vettore di 768 dimensioni, compatibile con il modello di classificazione.

5.3.2 Predizione dell'Intento

L'embedding generato viene fornito al modello per ottenere una classificazione dell'input utente.

```
    def predict_intent(sentence):
    bag = preprocess_sentence(sentence)
    res = model.predict(bag)[0]
    max_prob = np.max(res)
```

```
    predicted_class = np.argmax(res)
    return predicted_class, max_prob
```

Il modello calcola la probabilità di ogni classe di intenti e seleziona quella con valore massimo. Se la probabilità massima supera una soglia di confidence, viene restituita una risposta predefinita.

```
    if confidence > 0.75:
    intent_tag = classes[predicted_class]
    for intent in intents['intents']:
    if intent['tag'] == intent_tag:
    response = random.choice(intent['responses'])
```

Se invece il modello non è sicuro della predizione, il chatbot passa a Gemini per generare una risposta.

5.4 Utilizzo del Modello Generativo Gemini

Quando il modello di classificazione non è in grado di identificare con certezza l'intento, viene utilizzato Gemini, un modello generativo di intelligenza artificiale.

5.4.1 Generazione di una risposta con Gemini

```
    def get_gemini_response(input_text):
    model = genai.GenerativeModel("gemini-1.5-flash")
    convo = model.start_chat(history=chat_history["
       contents"])
    response = convo.send_message(input_text,
       max_output_tokens=50)
    return response.text
```

Gemini analizza l'input e genera una risposta basata sul contesto della conversazione. Il numero massimo di token generati è limitato a 50, per garantire risposte concise.

5.4.2 Selezione del Modello di Risposta

```
    def get_response_from_pipeline(input_text):
    predicted_class, confidence = predict_intent(
       input_text)

    if confidence > 0.75:
    return get_predefined_response(predicted_class)
    else:
    return get_gemini_response(input_text)
```

Se il classificatore ha confidence alta, viene selezionata una risposta predefinita. Se la confidence è bassa, viene utilizzato Gemini per generare una risposta dinamica.

La fase di modellazione ha portato alla creazione di un chatbot basato su:

- BERT per la generazione degli embedding
- Una rete neurale con Keras per la classificazione degli intenti
- Un fallback generativo con Gemini per input fuori dominio

Questa combinazione di tecnologie permette a Brigid di comprendere e rispondere in modo efficace, garantendo un'interazione naturale con gli utenti.

6 Testing del Modello

6.1 Introduzione

La fase di testing è fondamentale per verificare l'efficacia del chatbot Brigid e la sua capacità di classificare correttamente gli intenti degli utenti. Il nostro obiettivo è valutare le prestazioni del modello di classificazione basato su BERT e analizzare il comportamento del sistema in diversi scenari di utilizzo.

Per eseguire il testing, sono stati implementati tre test principali:

- **Baseline Test**: verifica l'accuratezza del modello utilizzando frasi presenti nel dataset di addestramento.
- **Generalization Test**: analizza la capacità del modello di classificare correttamente frasi nuove non presenti nel dataset.
- **Threshold Test**: valuta il livello di confidenza delle predizioni per verificare se il modello gestisce correttamente l'incertezza.

I risultati dei test sono stati salvati in un file `testing_results.txt` per una facile analisi e monitoraggio delle prestazioni.

6.2 Baseline Test

Il **Baseline Test** ha l'obiettivo di verificare se il modello ha appreso correttamente gli intenti presenti nel dataset di addestramento. Per evitare un testing eccessivamente lungo, vengono selezionati **tre pattern casuali per ogni intento**.

6.2.1 Metodo di esecuzione

- Si estrae una selezione casuale di tre frasi per ogni intento dal file `intents.json`.
- Si passa ogni frase al modello per ottenere la previsione dell'intento.
- Si confronta l'intento predetto con quello atteso e si registra l'esito.

Questo test ci permette di valutare la capacità del modello di riconoscere frasi viste durante l'addestramento, fornendo una misura dell'accuratezza di base.

6.3 Generalization Test

Il **Generalization Test** serve a valutare la capacità del modello di classificare frasi simili a quelle nel dataset, ma formulate in modi diversi.

6.3.1 Metodo di esecuzione

- Sono state create manualmente alcune frasi di test che non compaiono direttamente nel dataset.
- Ogni frase viene passata al modello per ottenere la previsione dell'intento.
- Il risultato viene confrontato con l'intento atteso.

Questo test è importante perché un chatbot deve essere in grado di generalizzare e non limitarsi a riconoscere solo le frasi apprese durante l'addestramento.

6.4 Threshold Test

Il **Threshold Test** verifica se il valore di confidenza delle predizioni del modello è coerente con il livello di sicurezza richiesto.

6.4.1 Metodo di esecuzione

- Si utilizzano le frasi definite nel Generalization Test.
- Per ogni frase, viene calcolata la confidenza della predizione.
- Se la confidenza è inferiore alla soglia impostata (0.75), la predizione viene considerata incerta.

Questo test aiuta a identificare situazioni in cui il modello potrebbe fare errori e assicura che, in caso di bassa confidenza, il chatbot possa eventualmente ricorrere a un fallback come Gemini.

6.5 Risultati e Considerazioni

L'analisi dei risultati dei test ha permesso di ottenere una visione chiara delle prestazioni del chatbot.

- **Il Baseline Test** ha mostrato un'alta accuratezza, confermando che il modello è in grado di riconoscere correttamente gli intenti appresi durante l'addestramento.
- **Il Generalization Test** ha evidenziato alcune difficoltà nel riconoscere frasi formulate in modi molto diversi da quelli nel dataset, suggerendo possibili miglioramenti nel dataset di training.
- **Il Threshold Test** ha mostrato che in alcuni casi il valore di confidenza era troppo basso per decisioni sicure, suggerendo la necessità di una gestione più attenta della soglia di sicurezza.

6.6 Possibili Miglioramenti

Dai test effettuati, emergono alcune possibili aree di miglioramento:

1. **Espansione del dataset**: l'inclusione di un maggior numero di varianti per ogni intento potrebbe migliorare la capacità di generalizzazione del modello.
2. **Fine-tuning di BERT**: un ulteriore addestramento del modello su un dataset più ricco potrebbe migliorare l'accuratezza sulle frasi nuove.
3. **Gestione adattiva della soglia di confidenza**: potrebbe essere utile adattare dinamicamente la soglia di confidenza in base al tipo di intento.

Questi miglioramenti possono essere implementati in future versioni del chatbot per aumentarne l'efficacia e la robustezza.

7 Deployment del Progetto

7.1 Introduzione

Il deployment del progetto riguarda la configurazione e l'esecuzione del chatbot **Brigid** in un ambiente locale utilizzando **PyCharm**. Questa fase è essenziale per garantire che il chatbot sia pronto per l'uso e possa essere eseguito correttamente su un sistema esterno. In questa sezione verranno fornite le istruzioni dettagliate per impostare l'ambiente di sviluppo e avviare il chatbot in **PyCharm**.

7.2 Configurazione dell'Ambiente in PyCharm

Per eseguire il chatbot su **PyCharm**, è necessario configurare correttamente un ambiente virtuale e installare tutte le dipendenze richieste.

7.2.1 Passaggi per la Configurazione

1. **Aprire il Progetto in PyCharm**
   - Avviare **PyCharm** e selezionare "Open".
   - Scegliere la cartella del progetto e confermare l'apertura.
2. **Creare un Ambiente Virtuale (Virtual Environment)**
   - Andare su **File** → **Settings** → **Project: [Nome Progetto]** → **Python Interpreter**.
   - Cliccare sull'icona dell'"ingranaggio" e selezionare **Add Interpreter** → **Virtualenv Environment**.
   - Scegliere la directory `.venv` all'interno della cartella del progetto oppure crearne una nuova.
   - Selezionare la versione di **Python** compatibile (es. Python 3.10 o 3.11).
   - Confermare e attendere la creazione dell'ambiente.

3. **Installare le Dipendenze del Progetto**

   - Aprire il **Terminale** di PyCharm e attivare l'ambiente virtuale:
     Su Windows:

     ```
     .venv\Scripts\activate
     ```

     Su macOS/Linux:

     ```
     source .venv/bin/activate
     ```

   - Installare le dipendenze necessarie:

     ```
     pip install -r requirements.txt
     ```

   - Se il file `requirements.txt` non è presente, può essere creato eseguendo:

     ```
     pip freeze > requirements.txt
     ```

7.3 Avvio del Chatbot

Dopo aver configurato l'ambiente virtuale e installato le dipendenze, è possibile eseguire il chatbot da PyCharm.

7.3.1 Avvio del Modello

1. Aprire il file `chatbot_terminal.py` in PyCharm.
2. Assicurarsi che l'interprete Python selezionato sia quello dell'ambiente virtuale.
3. Cliccare sul pulsante **Run** oppure eseguire il file dal terminale con:

   ```
   python chatbot_terminal.py
   ```

Se tutto è stato configurato correttamente, il chatbot si avvierà e sarà pronto per ricevere input nel terminale.

7.4 Possibili Problemi e Soluzioni

Durante il deployment, potrebbero verificarsi alcuni problemi. Ecco alcune soluzioni comuni:

7.4.1 Errore: Modulo non trovato (`ModuleNotFoundError`)

- Assicurarsi che tutte le dipendenze siano installate con `pip install -r requirements.txt`.
- Controllare che l'interprete Python selezionato sia quello dell'ambiente virtuale.

7.4.2 Errore: File `.keras` non trovato

- Verificare che il modello addestrato sia stato salvato correttamente nella cartella `models/`.
- Se il file manca, rieseguire `pretrain.py` per generare nuovamente il modello.

7.4.3 Errore: API Key mancante per Gemini

- Se il chatbot utilizza Gemini per generare risposte, assicurarsi di avere una chiave API valida e configurata correttamente.

7.5 Considerazioni Finali

Il deployment su PyCharm permette di testare e sviluppare il chatbot in un ambiente locale controllato. Una volta verificato il corretto funzionamento, il progetto potrà essere successivamente distribuito su altri ambienti, come server cloud o piattaforme web, per una maggiore accessibilità e scalabilità.

8 Glossario

Termini	Descrizione
Diversify	Progetto di Ingegneria del Software che prevede la realizzazione di un sito web per combattere le discriminazioni.
Gemini	Modello di linguaggio multimodale sviluppato da Google DeepMind.
PEAS	Performance Measure, Environment, Actuators, Sensors.
Bias	Errore sistematico causato da dati di addestramento sbilanciati o parziali, che porta il modello a produrre risultati distorti o discriminatori. Può influenzare l'equità, l'accuratezza e l'affidabilità delle previsioni.
NLP	Natural Language Processing.
Kaggle	Piattaforma online per data science e machine learning che offre competizioni, dataset pubblici, e strumenti collaborativi per sviluppare modelli e analizzare dati.
ChatGPT	Modello di linguaggio basato sull'architettura GPT (Generative Pre-trained Transformer), progettato per comprendere e generare testo in linguaggio naturale.
Tag	Etichetta che identifica un argomento specifico.
Pattern	Modelli o frasi che il bot cerca di riconoscere, spesso usando espressioni regolari o parole chiave.
Responses	Risposte predefinite che il bot restituisce quando un pattern corrisponde a un tag.
Intent	Gruppo formato da un tag, patterns e responses.
Dataset	Raccolta di dati organizzati in una struttura definita.
Numpy	Libreria Python fondamentale per il calcolo scientifico.
JSON	Formato di scambio dati leggero e leggibile. Allo stesso momento, è una libreria Python che permette la manipolazione di questi ultimi.
Pickle	Libreria Python che permette di serializzare (salvare) e deserializzare (caricare) oggetti Python in un formato binario.
Transformers	Libreria sviluppata da Hugging Face che semplifica l'uso di modelli avanzati di deep learning.
PyTorch	Libreria di machine learning, sviluppata da Facebook, usata per l'addrestramento di reti neurali.
TensorFlow/Keras	Libreria di machine learning open-source sviluppata da Google, utilizzata per costruire e addestrare modelli di deep learning.
Pretrain	Processo di addestramento iniziale di un modello di machine learning su un ampio dataset generico.

Termini	Descrizione
BERT	Bidirectional Encoder Representations from Transformers. Modello di deep learning per il trattamento del linguaggio naturale sviluppato da Google.
Embedding	Tecnica di rappresentazione numerica di parole, frasi o altre unità di dati in uno spazio continuo.