

## WildBytes - puntata 3. Memento

Ciao Youtubers e bentrovati su WildBytes! Nella puntata di oggi scopriremo come migliorare il codice e creare una vera e propria chat, creeremo una History per salvare le nostre chat e vedremo se è possibile usare le risposte precedenti per vedere se possiamo sviluppare una sorta di "memoria".

Così come abbiamo gestito la conservazione dei dati relativi ai **tokens** e al **budget**, faremo lo stesso con la History. Per farlo, definiremo una variabile chiamata "**chat History**" e la inizializzeremo come una lista vuota:

```
chatHistory = []
```

Nella funzione `getAnswer`, posso aggiungere un `append` per la domanda che farò e successivamente un altro `append` per la risposta che riceverò.

```
def getAnswer(self, question):
    """
    Ottiene una risposta da openAI. Ogni risposta costa 0.002 dollari.
    :param question:
    :return:
    """
    messages = [
        {"role": "user", "content": f"{question}"},
    ]
    self.chatHistory.append({"role": "user", "content": f"{question}"})
    response = openai.ChatCompletion.create(
        model=self.model,
        messages=messages,
        max_tokens=50, # limita la lunghezza della risposta
        temperature=0, # 0 = risposta più probabile, 1 = risposta più
creativa
    )
    answer = response['choices'][0]['message']['content']
    self.promptTokens += int(response['usage']['prompt_tokens'])
    self.completionTokens += int(response['usage']['completion_tokens'])
    self.totalTokens += int(response['usage']['total_tokens'])
    self.chatHistory.append({"role": "assistant", "content": f"{answer}"})
    return f"Risposta: {answer}\n{self.remainingTokens()}"
```

Come abbiamo fatto per i nostri dati, a questo punto possiamo creare una variabile `jsonHistory` alla quale passeremo il valore di ritorno di `json.dumps`. Una delle opzioni più utili di `dumps` è quella di salvare i file in modo che siano più **human readable** utilizzando le tabulazioni. Un buon valore di partenza per l'indentazione è 4.

```
def serialize(self):
    """
    Serializza i dati in un file json.
```

```
:return: i dati in formato json
"""
data = {
    "promptTokens": self.promptTokens,
    "completionTokens": self.completionTokens,
    "totalTokens": self.totalTokens,
    "budget": self.budget,
}
jsonData = json.dumps(data, indent=4)
# salva i valori dei tokens
self.save("data.json", str(jsonData))
# salva la chat history
jSonHistory = json.dumps(self.chatHistory, indent=4)
self.save("history.json", str(jSonHistory))
return data
```

Quando facciamo una domanda verrà quindi salvata insieme alla risposta sul file `history.json`.

**Attenzione!** È fondamentale assicurarsi che le informazioni sensibili non vengano immagazzinate in modo inappropriato, esposte o utilizzate per scopi non autorizzati. Quindi, è importante conservare il contenuto delle vostre chat in modo sicuro. Se non sapete come gestirle, è meglio cancellarle.

Prima di fare la domanda però volevo creare un loop per simulare una interazione tipo quella della chat. Posso farlo attraverso una funzione chiamata `startChat` che implementa un ciclo `while` basato sulla variabile `isWannaExit`. Fino a quando questa variabile rimane `False`, la chat mi chiederà di inserire una domanda.

```
def startChat(self):
    while self.isWannaExit is False:
        question = input("Domanda: ")
        if question.lower() == "#exit":
            self.isWannaExit = True
            break
        print(self.getAnswer(question))
```

In questo modo, all'interno del metodo `__init__`, possiamo chiamare `startChat()` per avviare il loop. Per interrompere la chat, posso semplicemente inserire `#exit`.

Dopo aver provato ad uscire dal programma e aver verificato che funziona, posso procedere con la prima domanda.

Domanda: Ciao chat sono K!

Risposta: Ciao K! Come posso aiutarti oggi?

Ti rimangono \$0.0499. Puoi ancora utilizzare 24973 tokens.

Domanda: Ciao chat come mi chiamo?

Risposta: Mi dispiace, ma non posso sapere il tuo nome a meno che tu non me lo dica.

Ti rimangono \$0.0498. Puoi ancora utilizzare 24907 tokens.

Domanda:

Facendo la stessa cosa sulla chat ufficiale di OpenAI:

Ciao chat sono K!

Ciao K! Come posso aiutarti oggi?

come mi chiamo?

Ti chiami K, come hai scritto nel tuo primo messaggio: "Ciao chat sono K!"

Come posso assisterti ulteriormente, K?

Andiamo quindi a verificare se la nostra "History" è stata salvata nel file:

```
[
  {
    "role": "user",
    "content": "Ciao chat sono K!"
  },
  {
    "role": "assistant",
    "content": "Ciao K! Come posso aiutarti oggi?"
  },
  {
    "role": "user",
    "content": "Ciao chat come mi chiamo?"
  },
  {
    "role": "assistant",
    "content": "Mi dispiace, ma non posso sapere il tuo nome a meno che tu non me lo dica."
  }
]
```

Come avrete notato, la chat di OpenAI ha la capacità di conservare le domande e le risposte precedenti all'interno di una sessione. Questa funzionalità è essenziale per garantire conversazioni fluide e costruire un contesto adeguato. Ma come possiamo implementare questa "memoria"?

La nostra variabile `chatHistory` funge da registro di tutte le domande e risposte scambiate durante una sessione. Ogni volta che formuliamo una domanda o otteniamo una risposta, la inseriamo nella nostra `chatHistory`.

Se desiderate che la chat tenga a mente una precedente interazione, dovrete inviare l'intera conversazione come prompt. Ciò significa che non si invierà solo l'ultima domanda alla funzione `getAnswer`, ma l'intera `chatHistory`.

Passare l'intera conversazione ogni volta tuttavia, potrebbe non essere la soluzione ideale. Questo perché potrebbe incrementare il numero di tokens utilizzati, aumentando così il costo della richiesta e raggiungendo potenzialmente il limite massimo di parole consentite nella domanda. I modelli di linguaggio, come il GPT-3.5 Turbo, hanno dei limiti sul numero di tokens che possono gestire in un'unica sessione. Ad esempio, GPT-3.5 Turbo ha due versioni: una con un limite di 8.000 tokens e l'altra con un limite di 16.000 tokens.

Un'analogia interessante potrebbe essere tratta dal film "Memento" di Christopher Nolan, del 2000. Il protagonista, Leonard Shelby, a causa di un disturbo di memoria a breve termine, si fa tatuare tutto ciò che non deve assolutamente dimenticare.

Dobbiamo trasporre questo concetto in termini informatici e per farlo potremmo, ad esempio, creare una funzione, come quella che abbiamo usato per uscire dalla chat: (`#exit`), `#memorize` che dovrebbe servire a "tatuare" nella memoria della chat informazioni essenziali. Potremmo, ad esempio, creare una stringa chiamata `memory` con il valore "MEMORIES: " e successivamente definire una funzione chiamata `addMemories` o `tattooMemories`.

```
def addMemories(self, value):  
    """  
    Aggiunge un ricordo alla memoria.  
    :param value:  
    :return:  
    """  
    self.memory += f"{value}, "
```

Nella funzione `getAnswer`, a questo punto potremmo integrare la memoria che desideriamo conservare per GPT.

```
messages = [  
    {"role": "user", "content": f"{self.memory}, Question:  
{question}"},  
]
```

```
def startChat(self):  
    while self.isWannaExit is False:  
        question = input("Domanda: ")  
        if question.lower() == "#exit":  
            self.isWannaExit = True  
            break  
        elif question.lower() == "#memorize":  
            self.addMemories(input("Cosa vuoi ricordare? "))  
            continue
```

Domanda: #memorize  
Cosa vuoi ricordare? mi chiamo K!

Domanda: come mi chiamo?

Risposta: Ti chiami K!

Ti rimangono \$0.0496. Puoi ancora utilizzare 24813 tokens.

Domanda: #exit

Così facendo, avrete la possibilità di creare, come con le calcolatrici, delle variabili che potrete evocare ogni volta che ne avrete bisogno. È ovviamente possibile implementare una serie di comandi aggiuntivi, come salvare o caricare la memoria da un file, visualizzare le informazioni memorizzate o eliminarle se non più necessarie. Tuttavia, per ora, proseguiamo con la nostra chat, considerando che non abbiamo ancora implementato la funzionalità di apertura del file `chatHistory` all'avvio del programma.

```
def __init__(self):
    openai.api_key = secretKeys.openAi
    self.initVariables()
    self.initHistory()
    self.startChat()

def initVariables(self):
    # inizializza le variabili che contengono i tokens
    data = self.open("data.json")
    self.deserializeData(data)

def initHistory(self):
    # inizializza la chat history
    data = self.open("history.json")
    self.deserializeHistory(data)
```

Nel metodo `__init__`, stiamo inizializzando alcune variabili e la nostra chat. Per mantenere tutto organizzato e leggibile, dobbiamo suddividere i compiti in diverse funzioni: una per inizializzare le variabili (`initVariables`) e una per inizializzare la storia della chat (`initHistory`).

Questa suddivisione rende il nostro metodo `__init__` più pulito e le funzioni individuali più comprensibili anche ad una prima lettura.

Continuiamo ora scrivendo la funzione per deserializzare la chat history.

```
def deserializeHistory(self, data):
    """
    Deserializza la chat history da un file json.
    :param data:
    :return:
    """
    jsonData = json.loads(data)
    self.chatHistory = jsonData
    return jsonData
```

Una delle cose che potremmo voler controllare durante la nostra sessione di chat è il budget. Se esauriamo il nostro credito, sarebbe utile avere un controllo che ci avvisa e ci chiede se vogliamo ricaricare il nostro account o uscire dalla chat. Ecco come potremmo implementare tale funzionalità:

```
def startChat(self):
    while self.isWannaExit is False:
        question = input("Domanda: ")
        if question.lower() == "#exit":
            self.isWannaExit = True
            break
        elif question.lower() == "#memorize":
            self.addMemories(input("Cosa vuoi ricordare? "))
            continue
        if self.budget <= 0.002:
            print("Non hai abbastanza soldi per continuare a chattare.")
            self.isWannaExit = self.askToAddMoreBudget()
        print(self.getAnswer(question))
```

Se il nostro budget scende al di sotto di una certa cifra, la chat ci avviserà che stiamo esaurendo i fondi. Dobbiamo quindi implementare la funzione `askToAddMoreBudget` per chiedere all'utente se desidera ricaricare il proprio credito.

```
def askToAddMoreBudget(self):
    """
    Chiede all'utente se vuole aumentare il budget.
    :return:
    """
    answer = input("Vuoi aumentare il budget? [Y/n]")
    if answer.lower() != "y":
        exit(0)
    self.coinUp()
    return True
```

Nel caso in cui l'utente desideri continuare, possiamo fornire un'opzione per aggiungere fondi al budget attraverso la funzione `coinUp`.

```
def coinUp(self):
    """
    Aggiunge 5 centesimi di dollaro al budget.
    """
    newBudget = input("Inserisci il nuovo budget: ")
    self.budget += float(newBudget)
    self.promptTokens = 0
    self.completionTokens = 0
    self.totalTokens = 0
    self.serialize()
```

Ora, per verificare se il nostro controllo del budget funziona come previsto, possiamo modificare il valore del "budget" nel nostro file `data.json` impostandolo su 0.002 dollari. Avviando il programma, dovremmo ottenere una risposta simile alla seguente:

Domanda:  
Non hai abbastanza soldi per continuare a chattare.  
Vuoi aumentare il budget? [Y/n]n

Durante questo test, possiamo anche notare che attualmente non c'è un meccanismo che gestisca l'eventualità in cui l'utente inserisca una domanda vuota. Potremmo considerare l'aggiunta di un controllo per tale situazione.

```
def startChat(self):
    while self.isWannaExit is False:
        question = input("Domanda: ").strip() # Utilizza strip() per
        # rimuovere spazi bianchi all'inizio e alla fine
        if not question: # Controlla se la domanda è vuota
            print("Per favore, inserisci una domanda valida.")
            continue
        if question.lower() == "#exit":
            self.isWannaExit = True
            break
        elif question.lower() == "#memorize":
            self.addMemories(input("Cosa vuoi ricordare? "))
            continue
        if self.budget <= 0.002:
            print("Non hai abbastanza soldi per continuare a chattare.")
            self.isWannaExit = self.askToAddMoreBudget()
        print(self.getAnswer(question))
```

Anche in questo caso, faccio un test per vedere se funziona e ottengo il mio risultato.

Domanda:  
Per favore, inserisci una domanda valida.  
Domanda: asdasd  
Non hai abbastanza soldi per continuare a chattare.  
Vuoi aumentare il budget? [Y/n]n

Process finished with exit code 0

Siamo giunti al termine di questa puntata, e oltre a ricordarvi è completamente gratuito, o come si dice, Open Source con licenza MIT vi ricordo che nella descrizione di ogni puntata, potete trovare il link alla mia pagina gitHub dove trovare il codice che utilizzeremo durante il video, e il pdf con il riassunto dei punti chiave oltre ai link per gli eventuali approfondimenti. Ma prima di salutarvi vi ricordo di mettere un like e iscrivermi al canale se non lo avete fatto e come sempre fate domande, tante domande... ma soprattutto sperimentate gente, sperimentate...

GoodByte Alla Prossima!