Aprendizado com Base na Contribuição dos Usuários Aplicado a chatbots

Rodrigo Rangel Vargas dos Santos

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC
DEPARTAMENTO DE INFORMÁTICA
Curso de Graduação em Engenharia da Computação

Orientador: Prof. Hélio Côrtes Vieira Lopes

Rio de Janeiro
Dezembro de 2021

Rodrigo Rangel Vargas dos Santos

Aprendizado com Base na Contribuição dos Usuários Aplicado a chatbots

Relatório de Projeto Final, apresentado ao programa Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do Bacharel em Engenharia de Computação.

Orientador: Prof. Hélio Côrtes Vieira Lopes Orientador

Departamento de Informática — PUC–Rio

Rio de Janeiro
Dezembro de 2021

i. Resumo:

Neste trabalho, foi desenvolvido um chatbot em língua portuguesa na plataforma de conversação do Telegram usando uma rede neural desenvolvida com a ferramenta Pytorch, que aplica o modelo seq-2-seq como mecanismo de processamento de linguagem natural. Outro artefato deste trabalho é o uso de um banco de dados para armazenar as conversas com os usuários a fim de realizar um retreinamento da rede e ampliar seu conhecimento.

ii. Palavras-chave:

Rede Neural. chatbot. Processamento de Linguagem Natural.

iii. Abstract

In this project a chatbot that uses portuguese was developed in the messaging platform Telegram using a neural network, developed with Pytorch, that applies the seq-2-seq model as a Natural Language Processing tool. Another product of this work is the use of a database to store the conversations with the users in order to perform a new cycle of training and expand the network's knowledge base.

iv. Keywords

Neural Network. chatbot. Natural Language Processing

Sumário

1. Introdução

> Alan Turing in 1950 proposed the Turing Test (“Can machines think?”), and it was at that time that the idea of a chatbot was popularized. The first known chatbot was Eliza, developed in 1966, whose purpose was to act as a psychotherapist returning the user utterances in a question form. It used simple pattern matching and a template-based response mechanism. Its conversational ability was not good, but it was enough to confuse people at a time when they were not used to interacting with computers and give them the impetus to start developing other chatbots. An improvement over ELIZA was a chatbot with a personality named PARRY developed in 1972. In 1995, the chatbot ALICE was developed which won the Loebner Prize, an annual Turing Test, in years 2000, 2001, and 2004. It was the first computer to gain the rank of the “most human computer”. ALICE relies on a simple pattern-matching algorithm with the underlying intelligence based on the Artificial Intelligence Markup Language (AIML), which makes it possible for developers to define the building blocks of the chatbot knowledge. chatbots, like SmarterChild in 2001, were developed and became available through messenger applications. The next step was the creation of virtual personal assistants like Apple Siri, Microsoft Cortana, Amazon Alexa, Google Assistant and IBM Watson. [1, p. 374]

Existe uma crescente demanda para que serviços utilizados diariamente pela sociedade tenham respostas cada vez mais rápidas e precisas. A tecnologia é uma aliada nesse sentido. Em particular, existe interesse e investimentos significativos em interfaces de usuário para atendimento ao cliente. De acordo com Scopus [2], apud [1], o interesse por chatbots vem crescendo, sendo possível observar um aumento acelerado a partir de 2016. (Figura 1)

Esse cenário tem levado muitas empresas a desenvolverem sistemas de conversação automáticos, ou chatbots. Um chatbot pode ser descrito de maneira simples como um programa projetado para promover uma comunicação inteligente baseada em correspondência de padrões. Os objetivos principais desses recursos são suprir a demanda, oferecer eficiência e, ao mesmo tempo, ampliar e otimizar custos com o atendimento ao cliente.

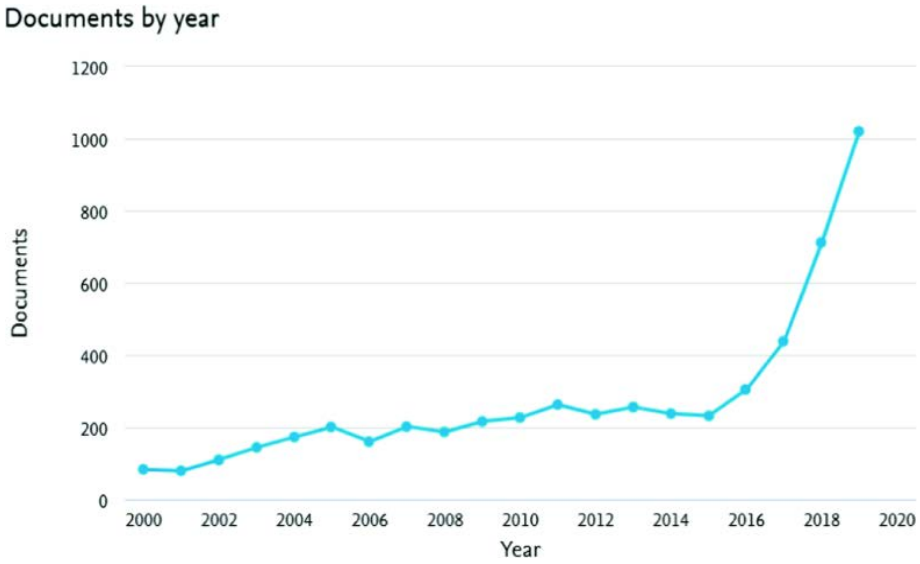

FIGURA 1: Resultado das buscas em Scopus por ano com os termos “chatbot”, “conversation agent” ou “conversational interface” como palavras chave de 2000 a 2019. [1, p.374]

No entanto, frequentemente, os chatbots não possuem uma resposta satisfatória para uma dada pergunta do usuário. O desafio que se apresenta é aprimorar os sistemas e as bases de dados. O aprendizado é o que amplia a capacidade e a eficiência do atendimento de chatbots. Nesse contexto, pesquisadores levantaram a ideia de inteligência coletiva [3], que envolve a humanização da inteligência artificial. Visa-se assim a melhorar a inteligência artificial de um programa usando o conhecimento coletivo. No caso do chatbot, isso se aplicaria na expansão e aprimoramento da base de conhecimento do bot, através de sugestões dos próprios usuários.

Uma área da inteligência artificial é o processamento de linguagem natural (NLP), que explora a manipulação de texto ou de fala em linguagem natural. O conhecimento da compreensão e do uso da linguagem humana é reunido para desenvolver técnicas que irão fazer com que os computadores entendam e manipulem expressões naturais para realizar as tarefas desejadas. A maioria das técnicas de NLP é baseada em aprendizado de máquina [1] [3].

Neste trabalho, propõe-se o desenvolvimento de um chatbot em língua portuguesa baseado no NLP caracterizado por um sistema de aprendizado capaz de utilizar o feedback dos usuários para aprimorar sua base de conhecimento. Esse chatbot será construído de modo que, quando a sua rede neural não conseguir identificar uma resposta, solicite ao usuário possíveis respostas

satisfatórias. Esse processo se alimenta da própria comunicação para ampliar o banco de dados e possibilitar novos treinamentos da rede. Com isso, a base de conhecimento do chatbot é ampliada de forma que, na próxima vez que um usuário fizer a mesma pergunta, o chatbot esteja apto a responder.

Visando ao objetivo geral de elaboração de um chatbot com as características indicadas, desenvolveu-se:

(i) investigação e seleção de ferramentas fundamentais para desenvolvimento de um chatbot, de forma a selecionar a mais adequada para o projeto;

(ii) elaboração de um protótipo para o chatbot pretendido, bem como acoplamento desse protótipo com o sistema de conversação e o banco de dados;

(iii) realização de testes com o protótipo para a definição dos parâmetros estruturais da rede neural;

(iv) treinamento da rede neural usando os parâmetros definidos e a base de dados escolhida;

(v) avaliação do aprendizado da rede neural e desempenho após retreinamento.

Na próxima seção, descreveremos brevemente o contexto no qual o trabalho se apresenta, apontando possibilidades e limitações tecnológicas que sustentam a proposta. Na terceira seção são apresentados exemplos de outros trabalhos com propostas similares. A quarta seção detalha a sequência de atividades realizadas no desenvolvimento do trabalho: estudos preliminares, estudos conceituais e de tecnologia, testes e protótipos para aprendizado e desenvolvimento e método. A quinta seção descreve as estruturas e aspectos técnicos que compõem o projeto. A sexta seção relata de forma avaliativa a implementação do projeto. Por fim, são apresentadas considerações finais.

2. Contexto

Na interação com um cliente, se o assistente virtual não conhece uma resposta para uma pergunta do usuário, é comum que ele responda com uma mensagem que não esclarece a questão apresentada, que não é adequada por alguma razão ou que simplesmente não faz sentido. Por exemplo, um bot alerta de forma repetida que há um erro no envio da mensagem do usuário, mesmo já tendo respondido-a satisfatoriamente (Figura 2A). Outro exemplo é o caso do bot não estar devidamente preparado para possíveis perguntas básicas, respondendo sempre com uma mensagem padrão, que não ajuda o usuário (Figura 2B).

No entanto, é possível que um chatbot, apesar de não ter uma resposta programada para uma dada pergunta do usuário, consiga redirecionar o usuário para um atendente humano (Figura 3A) ou sugerir outras opções que possam ajudá-lo (Figura 3B). Nesses casos, o chatbot deixa de ser o meio de interação com o cliente, transferindo a tarefa.

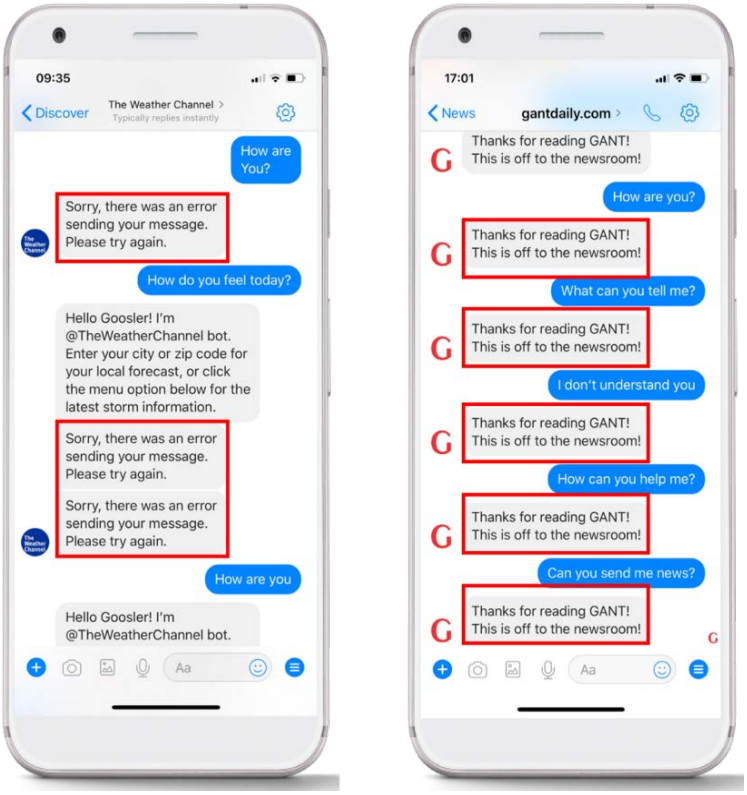

Figura 2A Figura 2B

FIGURA 2: Exemplo de respostas de chatbots [4]

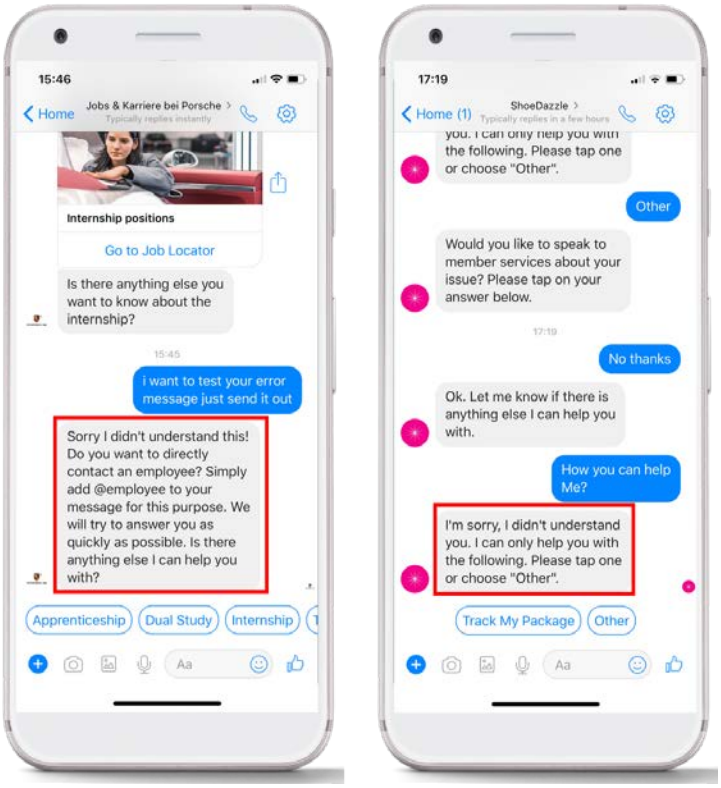

Figura 3A Figura 3B

Figura 3 - Exemplo de respostas de chatbots [4]

Projetar e desenvolver um chatbot é uma tarefa complexa, que envolve o uso de técnicas e conhecimentos variados que vão desde escolher os algoritmos, plataformas e ferramentas para criá-lo até considerar aspectos específicos da sua finalidade e dos seus usuários. De acordo com Adamopoulou & Moussiades,

> os requisitos para projetar um chatbot incluem uma representação precisa do conhecimento, uma estratégia de geração de respostas e um conjunto de respostas neutras predefinidas para responder quando a expressão do usuário não é compreendida [38]. O primeiro passo no projeto de qualquer sistema é dividi-lo em partes constituintes de acordo com um padrão, de forma que uma abordagem de desenvolvimento modular possa ser seguida [1, p.379-380].

A Figura 4 apresenta uma arquitetura geral de um chatbot.

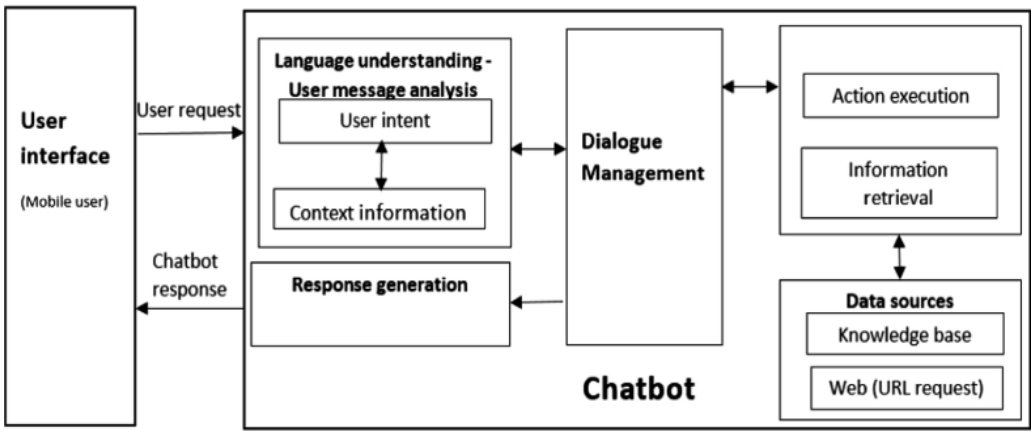

FIGURA 4: Arquitetura geral de um chatbot.
Fonte:[1, p.380]

São vários os conceitos e técnicas fundamentais relacionados ao desenvolvimento de um chatbot, Abdul-kader & Woods [5, p.73], ancorados em [6] e [7], entendem que, para desenvolver um chatbot, existem diversas técnicas e conhecimentos que são comumente utilizados, dentres os quais valem destacar:

> ***Parsing***: essa técnica inclui a análise e manipulação do texto de entrada usando funções de NLP; por exemplo, árvores em Python NLTK.
> ***Chat Script***: é a técnica que ajuda quando nenhum match ocorre no AIML. Ela foca na melhor sintaxe para construir uma resposta padrão coerente. Ela fornece uma série de funcionalidades como por exemplo conceito de variáveis, fatos, e operadores lógicos *and/or.*
> **SQL e banco de dados relacional**: é a técnica usada recentemente em design de chatbots para permitir que o chatbot lembre-se de conversas anteriores.

Para além dos aspectos técnicos, reconhecendo o papel protagonista do usuário, são diversos os desafios que se apresentam visando à qualidade e à eficiência de um chatbot. Por exemplo, é necessário considerar o contexto a que se destina, de saúde, de comércio, de educação etc. Além disso, é preciso considerar a relação do usuário com o chatbot [8] [9].

A complexidade da linguagem natural de comunicação certamente é um dos maiores desafios da implementação de um chatbot. Para compreender as mensagens é necessário conhecer a linguagem natural do usuário, no idioma correspondente e em diversas formas de entrada (texto, voz, por exemplo). A

intervenção neste cenário envolve fundamentalmente formas de ampliar a base de dados. É necessário estabelecer mecanismos de aprendizado que ampliem a base de conhecimento do chatbot para que seja possível aprimorar a eficiência dessa ferramenta e, consequentemente, melhorar a experiência dos usuários com o assistente virtual. Em particular, se considerada a língua portuguesa, o desafio é grande. Maeda & Moraes destacam que:

> Apesar dos diversos frameworks existentes hoje para o desenvolvimento de chatbots, ainda há muito esforço manual envolvido na construção desses agentes. É importante ressaltar que esse esforço aumenta ainda mais quando se trata da Língua Portuguesa. Poucos frameworks dão suporte ao Português e, quando o disponibilizam, são insuficientes para o processamento automático dessa língua. Em função dessas dificuldades e limitações, novas abordagens têm surgido. A Google, por exemplo, em Vinyals and Le, [11], propõe o uso de uma rede neural sequence to sequence [12] para definir a base de conhecimento de um agente conversacional. Naquele trabalho, o agente "aprende a responder" a partir de um corpus de diálogos. [10, p.11-12]

Reconhecendo a pouca disponibilidade de *datasets* outra língua que não seja o Inglês, em particular a língua portuguesa, este trabalho tem o objetivo de contribuir para a reflexão sobre a questão.

3. Trabalhos Relacionados

De forma a enfatizar a relevância e a demanda que existe para projetos como este, apresentaremos alguns trabalhos que utilizam tecnologias e conceitos análogos aos que foram fundamentais para o desenvolvimento dessa solução.

Maeda & Moraes [10] propuseram o desenvolvimento de um chatbot em língua portuguesa baseado no trabalho de Vinyals and Le [11] pela Google. Eles usaram datasets contendo legendas de filmes e também mensagens de um aplicativo de celular.

No seu trabalho, eles concluíram que não tiveram um nível de coerência semântica equivalente ao apresentado no trabalho da Google, possivelmente por conta da heurística usada para definir os turnos dos interlocutores. Isso se deu por conta da ausência de corpora de diálogo para língua portuguesa com as características necessárias para esse estudo.

Apesar disso, no caso do dataset de bate-papo para celular, eles puderam perceber traços de personalidade dependendo do interlocutor que estava interagindo com o agente, indicando que a rede esteja aprendendo o padrão de escrita (forma e vocabulário usual) dos interlocutores cujas entradas estão no dataset.

Outro exemplo de um projeto similar é o MILABOT, desenvolvido por Serban et al. [13]. O seu trabalho consiste na criação de um chatbot usando vários modelos de rede neural, incluindo modelo bag-of-words e sequence-to-sequence. Eles aplicaram um método de aprendizado por reforço em dados coletados de interações reais com usuários para criar um sistema que escolhe a melhor resposta dada dentre os diferentes modelos de rede neural usados.

Foram realizados testes A/B com os usuários nos quais eles concluíram que seu robô teve resultados melhores que seus concorrentes e, dado a natureza da sua arquitetura de aprendizado de máquina, é esperado que o sistema melhore ao obter mais dados para os seus modelos.

Um terceiro caso é o trabalho de Li et al. [14], que construiu um modelo que simula diálogos entre dois agentes virtuais, usando métodos de gradiente para recompensar sequências que apresentam três propriedades conversacionais úteis: informatividade, coerência e facilidade de resposta.

Eles avaliaram seu modelo com relação a diversidade e tamanho das respostas, além de utilizar jurados humanos. Eles concluíram que o algoritmo proposto gera respostas mais interativas e consegue promover uma simulação de diálogo no qual a conversação é sustentada por mais tempo.

4. Atividades Realizadas

4.1. Estudos Preliminares

O autor traz experiência anterior com o desenvolvimento de chatbots com base na linguagem AIML, devido a participação em projetos de pesquisa prévios. Durante essas pesquisas, o autor também se familiarizou com a plataforma de aprendizado de máquina Tensorflow. Contudo, o uso dessa plataforma foi básico e não chegou a ser aprofundado. Assim, o desenvolvimento do trabalho aqui apresentado oferece uma oportunidade de ampliação e aprofundamento de estudo.

Considerando que o trabalho tem como base a construção de uma rede neural, foi feita uma pesquisa preliminar sobre as possíveis ferramentas, e a ferramenta escolhida foi o Pytorch. Segundo Simmons & Holiday [15], essa ferramenta tem um tempo de treinamento menor do que o Tensorflow, e, por ser mais simples, é mais fácil implementar um protótipo de rede neural usando o Pytorch.

A eficiência do Pytorch também é defendida por Paszke et al. [16, p.1], “It (Pytorch) provides an imperative and Pythonic programming style that supports code as a model, makes debugging easy and is consistent with other popular scientific computing libraries, while remaining efficient and supporting hardware accelerators such as GPUs.”

4.2. Estudos conceituais e de tecnologia

O Pytorch se configura como um pilar fundamental para a elaboração deste projeto. Assim, exploramos a sua documentação oficial a fim de entender como essa ferramenta aplica os principais conceitos de aprendizado de máquina e redes neurais.

Após uma análise sobre as diferentes formas de implementação de rede neural, de forma a encontrar o que melhor se aplica à proposta de desenvolver um chatbot, optamos por seguir o exemplo apresentado na documentação oficial, desenvolvido por Inkawhich [17]. Esse exemplo implementa um modelo *seq-2-seq*, proposto inicialmente por Sutsvekever et al [12]. Como apontam Maeda & Moraes [10], esses autores mostraram que uma arquitetura utilizando redes Long short-term memory (LSTM) é capaz de traduzir textos em francês para inglês com desempenho comparável ao estado da arte na área de tradução automática.

O modelo de Inkawhich [17] usa o mecanismo de atenção de Luong et al. [18], que atribui uma pontuação de importância para cada palavra na sequência de entrada, de forma a obter um melhor resultado ao final do treinamento da rede. Inkawhich [17] também faz um treinamento conjunto das camadas de encoding e decoding usando pequenos lotes do dataset de entrada. Por fim, ele implementa o módulo de decoding com busca gulosa, que simplesmente seleciona, dentre as possíveis respostas apresentadas pela rede neural, aquela com melhor resultado.

Sobre a plataforma de conversa na qual o projeto será implementado, o autor possui experiência prévia com o desenvolvimento de bots para as plataformas do Telegram, Google Assistant e Facebook Messenger. Entre essas, foi escolhido o Telegram, por possuir uma API mais simples e não exigir uma avaliação prévia da aplicação para torná-la pública.

Para realizar o armazenamento dos dados, foi escolhido o banco de dados relacional Postgresql, pelo autor ter experiência prévia com essa ferramenta e ter fácil integração com Python. Conforme o volume de dados aumente, um banco de dados não relacional, como MongoDB por exemplo, seria mais adequado, por possuir melhor escalabilidade quando é necessário um alto número de requisições. Segundo Gupta et al. *“Bancos de dados não relacionais surgiram como uma alternativa importante a bancos de dados relacionais e nós os escolhemos de acordo com recursos como escalabilidade, disponibilidade e tolerância a falhas..”* [19, p.293]

4.3. Testes e Protótipos para Aprendizado e Demonstração

Em um primeiro momento, foi utilizado o tutorial de Inkawhich [17] como referência para a construção de um protótipo da rede neural. O modelo inicial foi testado visando a verificar a efetividade de integração com a API que faz a conexão com o Telegram, interface de comunicação com o usuário.

Diante da oferta escassa de bases de dados em língua portuguesa, utilizamos a base *Cornell Movie--Dialogs Corpus* [20], cujo idioma é inglês, para treinar o modelo da rede neural do chatbot. Esse dataset é baseado em um trabalho que tem o apoio da *National Science Foundation* e, segundo o autor, possui as seguintes características.

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies
- in total 304,713 utterances
- movie metadata included:
  - genres
  - release year
  - IMDB rating
  - number of IMDB votes
  - IMDB rating
- character metadata included:
  - gender (for 3,774 characters)
  - position on movie credits (3,321 characters)

Entendemos que essas características oferecem um bom ponto de partida para calibrar as configurações básicas da rede neural, bem como testar o funcionamento da arquitetura do projeto como um todo, incluindo a conexão com o cliente do Telegram e com o banco de dados. Foi então assim concluída a etapa de criação de um protótipo capaz de treinar uma rede neural com as configurações padrão estabelecidas por Inkawhich [17] e então usar essa rede para responder a um usuário no Telegram.

Além de estabelecer comunicação, o protótipo desenvolvido foi programado para armazenar as mensagens no banco de dados.

FIGURA 5: Exemplo de conversa com um protótipo inicial

4.4. Método

Com um protótipo em funcionamento, o próximo passo foi realizar uma sequência de testes para investigar a melhor configuração para a rede neural, de forma que se tenha a menor perda possível durante o treinamento

Primeiramente, foram feitos testes visando ao modelo de atenção considerando os métodos como definidos por Luong et al. [18]: dot, general e concat. Foram realizadas 1000 interações. Os resultados desses testes no protótipo podem ser vistos na Figura 6.

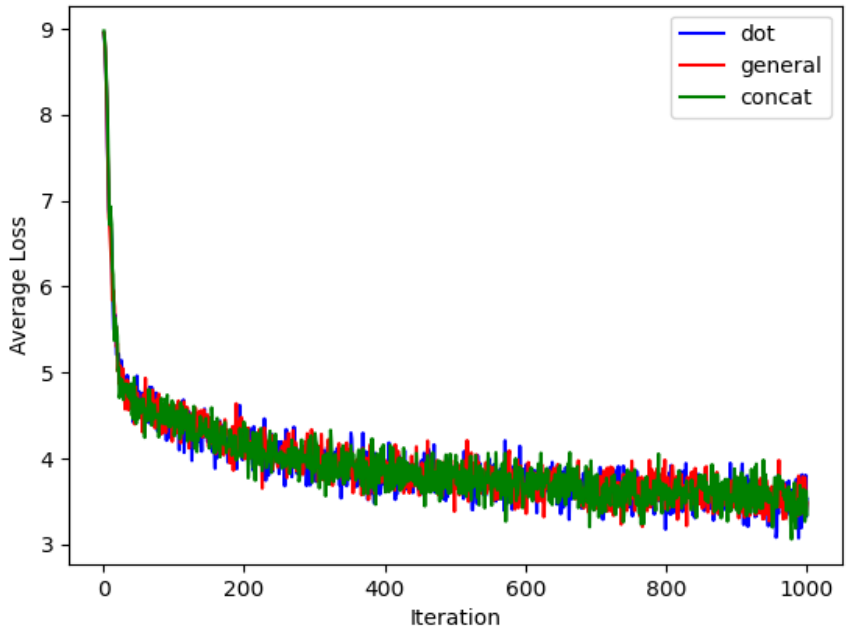

FIGURA 6: Gráfico ilustrando as curvas de perda média da rede ao longo do treinamento usando os métodos dot, general e concat.

De acordo com os resultados, observou-se que não há uma diferença significativa de perda média entre os três tipos. Dessa forma, o tipo escolhido foi *dot*, pois como foi estabelecido por Luong et al. [X], esse tipo funciona melhor para o modelo de atenção global, que é o caso desse projeto.

O passo seguinte foi realizar testes visando a determinar os melhores valores para a taxa de aprendizado da rede. Os valores escolhidos foram 0.001, 0.0001, e 0.0001, e o teste teve 5000 iterações. Os resultados podem ser vistos na Figura 7. Esse processo de testagem evidenciou que uma taxa de aprendizado de 0.0001 é a que tem o melhor resultado quando se trata da perda média, portanto essa foi a taxa escolhida para o treinamento final da rede.

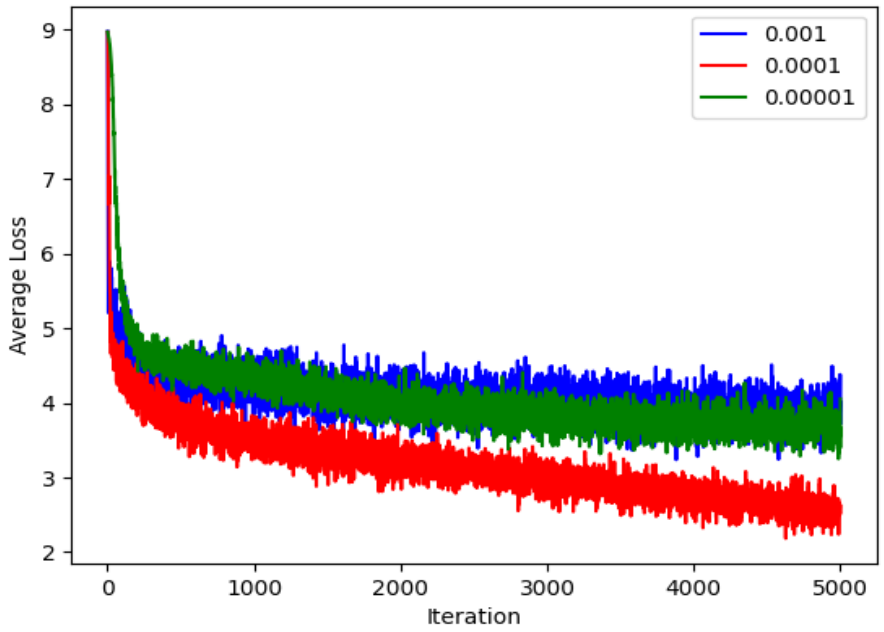

FIGURA 7: Gráfico ilustrando as curvas de perda média ao longo do treinamento usando taxa de aprendizado 0.001, 0.0001 e 0.0000.

Após a testagem com atenção à aprendizagem, foram realizados testes para determinar o melhor valor para o número de nós das camadas ocultas. Os valores testados foram 500, 250 e 100 nós. Foram realizadas 1000 interações. Os resultados podem ser vistos na Figura [9]. Nesse caso, podemos ver que o treinamento com 500 nós na camada escondida foi o que obteve o melhor resultado e, portanto, esse foi o valor escolhido para a rede.

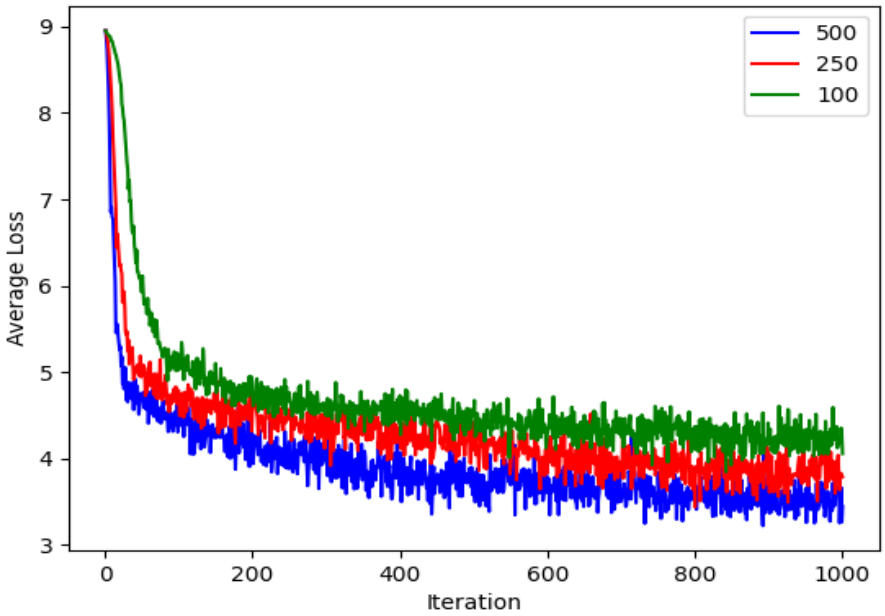

FIGURA 8: gráfico ilustrando as curvas de perda média ao longo do treinamento usando 500, 250 e 100 nós nas camadas ocultas

Por fim, o último teste realizado avalia o fator de aprendizado da camada *Decoder*, com os possíveis valores sendo 5.0, 2.5 e 1.0. Os resultados podem ser vistos na Figura [10]. Nesse teste, é possível ver que o fator de 5.0 teve um resultado ligeiramente melhor que os demais. Dessa forma, esse foi o fator escolhido para o treinamento da rede.

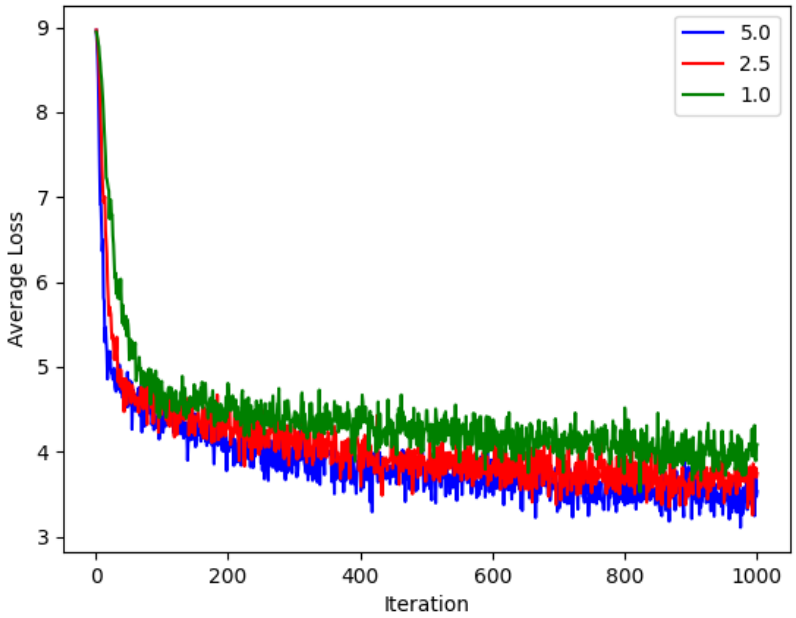

FIGURA 9: Gráfico ilustrando as curvas de perda média ao longo do treinamento usando 5.0, 2.5 e 1.0 como fator de aprendizado do *Decoder*

5. Projeto e Especificação do Sistema

5.1. API

A API tem como componente principal o cliente do Telegram, que faz a conexão com o bot para receber e enviar mensagens pelo chat. A biblioteca utilizada para fazer essa interface foi a *python-telegram-bot*, uma biblioteca *open source* que fornece uma interface com a aplicação do Telegram através do Python. Ao inicializar a API, é carregado o modelo de rede neural e, toda vez que for enviada uma mensagem para o bot, ele busca na rede a melhor resposta e a retorna para o usuário. Caso a rede não consiga encontrar uma resposta, o robô solicita ao usuário que indique uma resposta adequada. Abaixo segue o trecho do código correspondente.

```
def send_message(self, update, context):
    """
    método para obter a resposta do modelo de rede neural e
responder ao usuário
    Args:
        update: última atualização da conversa
        context: contexto da conversa
    """

    if self.sessions[update.effective_chat.id].correct_answer:
        self.handle_message(update, '')
        answer = "Ok, vou lembrar disso para um próxima vez!"
        self.sessions[update.effective_chat.id].correct_answer =
False
    else:
        try:
            # obtendo resposta da rede neural
            answer =
format_answer(self.ml_client.evaluate(update.message.text))
            # método para salvar a nova mensagem no banco de
dados
            self.handle_message(update, answer)
        except KeyError:
            # caso a rede neural não consiga responder
            self.handle_message(update, '')
            answer = "Desculpe, não consegui entender, pode me
informar o que eu deveria respoder?"
            self.sessions[update.effective_chat.id].correct_answ
er = True

    context.bot.send_message(chat_id=update.effective_chat.id,
text=answer)
```

Todas as mensagens enviadas pelos usuários são armazenadas no banco de dados para poderem ser utilizadas futuramente em novos treinamentos da rede. Dessa forma, o robô constrói seu próprio dataset ao longo do tempo, aprimorando sua base de conhecimento. Abaixo segue o trecho do código correspondente.

```
def handle_message(self, update, answer):
    """
    método para salvar a nova conversa no banco de dados e
atualizar a conversa e a sessão
    Args:
        update: última atualização da conversa
        answer: resposta à mensagem à ser armazenada
    """

    user_id = update.message.chat_id

    # verificando se já existe uma sessão para a quele usuário
    if user_id not in self.sessions:
        user_info = update.message.chat

        # criando uma nova conversa
        new_conversation = Conversation(count_conversations()+1)

        # criando um novo usuário
        new_user = User(
            user_id,
            user_info.username,
            user_info.first_name,
            user_info.last_name
        )

        # criando uma nova sessão
        new_session = Session(user_id, new_conversation,
new_user, None)

        # criando uma nvoa mensagem
        new_session.new_message(update.message.text, user_id,
datetime.now())

        self.sessions[user_id] = new_session

    else:
        # criando uma nova mensagem
        self.sessions[user_id].new_message(update.message.text,
user_id, datetime.now())

    if answer != '':
        self.sessions[user_id].new_message(answer, self.bot_id,
datetime.now())
```

A API também é responsável por manter o estado das sessões com os usuários, guardando, para cada usuário, a conversa atual e a última mensagem enviada na sessão. Essas informações também são armazenadas no banco de dados, e são carregadas quando a API é inicializada, obtendo dessa forma uma persistência dos dados das sessões.

As mensagens são armazenadas no banco de forma que seja possível separá-las em conversas, na mesma ordem em que foram enviadas. Assim, quando o programa executar um novo treinamento, ele consegue coletar e estruturar os dados no formato necessário para o modelo da rede neural.

5.2. Modelo de Rede Neural

O modelo de rede neural usado no projeto foi baseado no exemplo do tutorial de Inkawhich [17] e é composto dos seguintes módulos:

i. Encoder

O módulo *Encoder* estende a classe base do Pytorch e implementa o método que prepara a entrada para ser enviada para o módulo *Decoder*. Quando criado, ele recebe como parâmetro o número de camadas ocultas, o número de nós das camadas ocultas, um objeto de *Embedding*, usado para indexar as palavras de entrada, e um fator de dropout que é usado para evitar overfitting durante a transição entre as camadas ocultas. Os dois primeiros parâmetros podem ser ajustados para melhorar a acurácia geral da rede. Abaixo segue o trecho do código correspondente.

```
class EncoderRNN(nn.Module):
    """
    classe que define o modelo de encoder para a rede neural
    """
    def __init__(self, hidden_size, embedding, n_layers=1,
dropout=0):
        """
        método de inicialização do encoder
        Args:
            hidden_size: tamanho da camada oculta
            embedding: embedding do modelo para redução de
dimensão
```

```
            n_layers: número de camadas
            dropout: fator de dropout para reduzir overfitting
        """
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # Initialize GRU; the input_size and hidden_size params
are both set to 'hidden_size'
        #   because our input size is a word embedding with
number of features == hidden_size
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers,
                          dropout=(0 if n_layers == 1 else
dropout), bidirectional=True)
```

Durante a execução, esse módulo vai receber uma sequência de índices referentes às palavras de entrada, e transforma essa sequência em um vetor de contexto com tamanho fixo, para então enviar para o módulo *Decoder.* Abaixo segue o trecho do código correspondente.

```
def forward(self, input_seq, input_lengths, hidden=None):
    """
    método para enondar a sequência de entrada e preparar para a
rede
    Args:
        input_seq: sequência de indíces das palavras de entrada
        input_lengths: tamanho das palavras de entrada
        hidden: última camada oculta

    Returns: resultado da entrada codificada e o novo estado da
camada oculta
    """
    # Convert word indexes to embeddings
    embedded = self.embedding(input_seq)
    # Pack padded batch of sequences for RNN module
    packed = nn.utils.rnn.pack_padded_sequence(embedded,
input_lengths)
    # Forward pass through GRU
    outputs, hidden = self.gru(packed, hidden)
    # Unpack padding
    outputs, _ = nn.utils.rnn.pad_packed_sequence(outputs)
    # Sum bidirectional GRU outputs
    outputs = outputs[:, :, :self.hidden_size] + outputs[:, :,
self.hidden_size:]
    # Return output and final hidden state
    return outputs, hidden
```

ii. Decoder

O módulo *Decoder* estende a classe base do Pytorch e implementa a camada da rede que, para cada palavra, retorna uma predição de qual seria a próxima palavra na sequência. Quando criado, ele recebe, além dos mesmos parâmetros que o módulo *Encoder*, o tamanho da saída e o tipo do modelo de atenção usado. Abaixo segue o trecho do código correspondente.

```
class LuongAttnDecoderRNN(nn.Module):
    """
    classe que define o decoder de Luong baseado no método de
modelos de atenção
    """
    def __init__(self, attn_model, embedding, hidden_size,
output_size, n_layers=1, dropout=0.1):
        """
        Args:
            attn_model: tipo do modelo de atenção
            embedding: embeding do modelo para redução de
dimensão
            hidden_size: tamanho da caamda oculta
            output_size: tamanho da saída
            n_layers: número de camadas
            dropout: fator de dropout para reduzir overfitting
        """
        super(LuongAttnDecoderRNN, self).__init__()

        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers,
dropout=(0 if n_layers == 1 else dropout))
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)
```

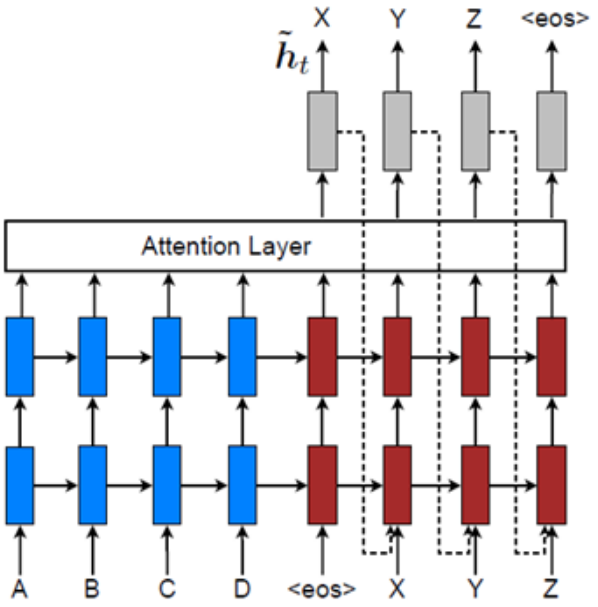

FIGURA 10: Ilustração da arquitetura de uma rede neural que possui uma camada de atenção como definido por Luong et al. [18]

Como já dito, o método do modelo de atenção é usado para gerar pesos para cada estado durante o treinamento, ajudando o *Decoder* a escolher a melhor resposta. Nesse projeto, o modelo de atenção usado é o Global, e ele pode calcular os pesos de 3 formas diferentes, como definido por Luong et al. [18]

a. Dot: Produto entre a probabilidade encontrada pelo *Decoder* e a recebida pelo *Encoder.*

```
def dot_score(hidden, encoder_output):
    """
    método para calcular os pesos de atenção de acordo com o
metodo 'dot'
    Args:
        hidden: última camada oculta
        encoder_output: sequência de entrada

    Returns: resultado dos pesos de atenção
    """
    return torch.sum(hidden * encoder_output, dim=2)
```

b. General: Produto entre a probabilidade encontrada pelo *Decoder* e a probabilidade ponderada do *Encoder*.

```
def general_score(self, hidden, encoder_output):
    """
    método para calcular os pesos de atenção de acordo com o
metodo 'general'
    Args:
        hidden: tamanho da camada oculta
        encoder_output: sequência de entrada

    Returns: resultado dos pesos de atenção
    """
    energy = self.attn(encoder_output)
    return torch.sum(hidden * energy, dim=2)
```

c. Concat: Resultado da rede neural após processar a concatenação entre as duas probabilidades.

```
def concat_score(self, hidden, encoder_output):
    """
    método para calcular os pesos de atenção de acordo com o
metodo 'concat'
    Args:
        hidden: última camada oculta
        encoder_output: sequência de entrada

    Returns: resultado dos pesos de atenção
    """
    energy =
self.attn(torch.cat((hidden.expand(encoder_output.size(0), -1,
-1), encoder_output), 2)).tanh()
    return torch.sum(self.v * energy, dim=2)
```

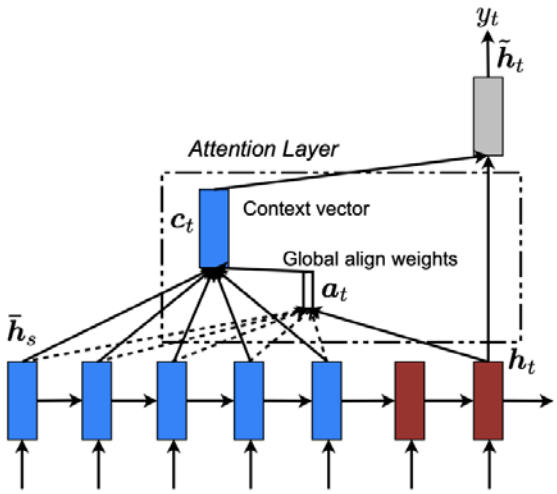

FIGURA 11: Ilustração do cálculo da pontuação através da camada de atenção, como definido por Luong et al. [18]

Após processar a entrada, o *Decoder* usa o método escolhido, no caso desse projeto o dot, para calcular as pontuações de cada uma das possíveis respostas para aquela palavra, retornando um vetor com essas pontuações junto com o estado da última camada escondida.

iii. Search Decoder

```
class GreedySearchDecoder(nn.Module):
    """
    classe que define o modelo de busca gulosa para determinar a
melhor
    resposta da rede
    """
    def __init__(self, encoder, decoder, device):
        """
        método de inicialização do GreedyDecoder
        Args:
            encoder: modelo de encoder usado na rede
            decoder: modelo de decoder usado na rede
            device: tipo de hardware usado (CPU ou GPU)
        """
        super(GreedySearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.device = device
```

O código acima descreve a estrutura do módulo *Search Decoder,* que estende a classe base do Pytorch e é responsável por coordenar os módulos *Encoder* e *Decoder* após o treinamento. Quando uma entrada é enviada para o modelo para ser avaliada, o *Search Decoder* passa para o *Encoder* a sequência de entrada e então faz da sua última camada oculta a primeira entrada da camada oculta do *Decoder*.

Em seguida, ele inicializa a entrada do *Decoder*, indicando o início de uma frase, e os tensores que irão armazenar os tokens da saída e suas pontuações. Então, para cada palavra da sequência de entrada, ele chama o *Decoder* e seleciona a resposta com melhor pontuação. Essa resposta então é usada como próxima entrada do *Decoder* e esse processo se repete. Durante a execução, as respostas escolhidas e suas pontuações são inseridas nos tensores para que, no final, eles estejam preenchidos com a resposta final e possam ser retornados para o módulo *Model*. Abaixo segue o trecho do código correspondente.

```
def forward(self, input_seq, input_length, max_length):
    """
    método para calcular a melhor resposta para cada palavra da
sequência de entrada
    Args:
        input_seq: sequência de entrada
        input_length: tamanho da sequência de entrada
        max_length: tamanho máximo da sequẽncia de entrada

    Returns: uma lista com as palavras que compõem a melhor
resposta da rede, e uma lista com suas potuações
    """
    # Forward input through encoder ml_model
    encoder_outputs, encoder_hidden = self.encoder(input_seq,
input_length)
    # Prepare encoder's final hidden layer to be first hidden
input to the decoder
    decoder_hidden = encoder_hidden[:self.decoder.n_layers]
    # Initialize decoder input with SOS_token
    decoder_input = torch.ones(1, 1, device=self.device,
dtype=torch.long) * SOS_token
    # Initialize tensors to append decoded words to
    all_tokens = torch.zeros([0], device=self.device,
dtype=torch.long)
    all_scores = torch.zeros([0], device=self.device)
    # Iteratively decode one word token at a time
    for _ in range(max_length):
        # Forward pass through decoder
        decoder_output, decoder_hidden =
self.decoder(decoder_input, decoder_hidden, encoder_outputs)
        # Obtain most likely word token and its softmax score
        decoder_scores, decoder_input =
torch.max(decoder_output, dim=1)
        # Record token and score
        all_tokens = torch.cat((all_tokens, decoder_input),
dim=0)
        all_scores = torch.cat((all_scores, decoder_scores),
dim=0)
        # Prepare current token to be next decoder input (add a
dimension)
        decoder_input = torch.unsqueeze(decoder_input, 0)
    # Return collections of word tokens and scores
    return all_tokens, all_scores
```

iv. Model

Model é o módulo principal do modelo de rede neural. Ele é responsável por treinar as camadas de *Encoder* e *Decoder*, bem como chamar o módulo *Search Decoder* quando receber uma entrada para poder retornar a resposta da rede para a API.

Quando inicializado, ele cria os objetos de *Encoder*, *Decoder* e *Search Decoder*, usando os parâmetros tipo do modelo de atenção, números de camadas ocultas, quantidade de nós da camada oculta, taxa de dropout e tamanho do lote. Abaixo segue o trecho do código correspondente.

```
self.encoder = EncoderRNN(
   self.hidden_size,
   self.embedding,
   self.encoder_n_layers,
   self.dropout
)
self.decoder = LuongAttnDecoderRNN(
   self.attn_model,
   self.embedding,
   self.hidden_size,
   self.processor.vocabulary.num_words,
   self.decoder_n_layers,
   self.dropout
)
self.searchDecoder = GreedySearchDecoder(
   self.encoder,
   self.decoder,
   self.device
)
```

Ao ser executado o treinamento, ele usa os parâmetros taxa de aprendizado, fator de aprendizado do *Decoder*, taxa de aprendizado forçado, número de iterações e gradiente de corte para preparar as camadas de *Encoder* e *Decoder,* que compõem a rede neural em si. Durante o treinamento, são salvos *checkpoints* da rede a cada 500 iterações, para evitar retreinamentos a cada vez que o modelo é inicializado. Abaixo segue o trecho do código correspondente.

```
for iteration in range(start_iteration, self.n_iteration + 1):
   training_batch = training_batches[iteration - 1]
   # Extract fields from batch
   input_variable, lengths, target_variable, mask,
max_target_len = training_batch

   # Run a training iteration with batch
   loss = self._train(
       input_variable,
       lengths,
       target_variable,
       mask,
       max_target_len,
   )
   print_loss += loss
```

Com a rede treinada, quando o modelo recebe uma entrada da API, ele a envia para o módulo *Search Decoder*, que usa a rede para determinar a melhor resposta e então retornar para a API.

v. Data Processor

O módulo *Data Processor* é responsável por coletar os dados do banco de dados e formatá-los de forma que eles possam ser usados para o treinamento da rede. Para isso, ele gera um arquivo com os pares de sentenças a partir das conversas e mensagens existentes no banco. Cada par consiste em uma frase e a resposta a essa frase dentro de uma mesma conversa. Abaixo segue o trecho do código correspondente.

```
def format_text(self, output_file, start_date=None,
end_date=None):
    """
    método para escrever os dados coletados do banco de dados em
um arquivo a ser usado pela rede
    Args:
        output_file: caminho do arquivo de saída
        start_date: data inicial para extração
        end_date: data final para extração
    """

    print("\nWriting newly formatted file...")
    if start_date:
        with open(output_file, 'a', encoding='utf-8') as output:
            writer = csv.writer(output,
delimiter=self.delimiter, lineterminator='\n')
            for pair in self.extract_sentence_pairs(start_date,
end_date):
                writer.writerow(pair)
    else:
        with open(output_file, 'w', encoding='utf-8') as output:
            writer = csv.writer(output,
delimiter=self.delimiter, lineterminator='\n')
            for pair in self.extract_sentence_pairs():
                writer.writerow(pair)

    print("\nDone!")
```

Com isso, quando a rede for ler esse arquivo para o treinamento, o módulo *Data Processor* antes faz um tratamento nesses pares para obter um melhor resultado. Ele remove aqueles que possuem palavras muito raras e frases muito

curtas ou muito longas, de acordo com os parâmetros que são passados na sua inicialização.

Depois disso, os conteúdos dos textos são normalizados, removendo caracteres não alfanuméricos e espaços desnecessários, e então usados para construir o vocabulário da rede neural, que relaciona cada palavra da rede com um índice para ser usado na hora de enviar a entrada para o módulo *Encoder*. Abaixo segue o trecho do código correspondente.

```
def normalize_string(s):
    """
    método para normalização de string
    Args:
        s: string original

    Returns: string com todos os caracteres minúsculos e apenas
letras e números
    """
    s = unicode_to_ascii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z0-9.!?]+", r" ", s)
    s = re.sub(r"\s+", r" ", s).strip()
    return s

def add_word(self, word):
    """
    método para adiconar palavra ao vocabulário
    Args:
        word: palavra a ser adicionada
    """

    # adiciona a palavra caso não exista
    if word not in self.word2index:
        self.word2index[word] = self.num_words
        self.word2count[word] = 1
        self.index2word[self.num_words] = word
        self.num_words += 1
    else:
        # caso já exista, apenas aumenta a contagem da palavra
        self.word2count[word] += 1
```

O *Data processor* também é responsável por dividir os dados de entrada para o treinamento em lotes, de acordo com o tamanho do lote definido no modelo e enviar esses dados em partes para a rede. Essa divisão é feita para reduzir o tempo e otimizar o uso de memória durante o treinamento.

5.3. Cliente do Banco de dados

O cliente do banco de dados é inicializado quando o programa começa a rodar, e é responsável por fornecer uma conexão com o Postgres através da biblioteca Psycopg2 para os outros módulos. Os dados de conexão com o banco são passados para o cliente através de variáveis de ambiente definidas na raiz do projeto. Esse módulo também possui um método para criar as tabelas no banco de dados em um primeiro momento no qual elas não existem.

Também existem métodos auxiliares para fazer a contagem de sessões, conversas e mensagens que existem no banco a cada momento.

5.4. Modelo Entidade Relacionamento

As classes foram definidas de forma a possibilitar a construção do dataset no formato necessário para o treinamento da rede neural a partir dos dados coletados. Abaixo segue o modelo de entidades que compõem o banco de dados, bem como uma breve descrição de cada um dos campos.

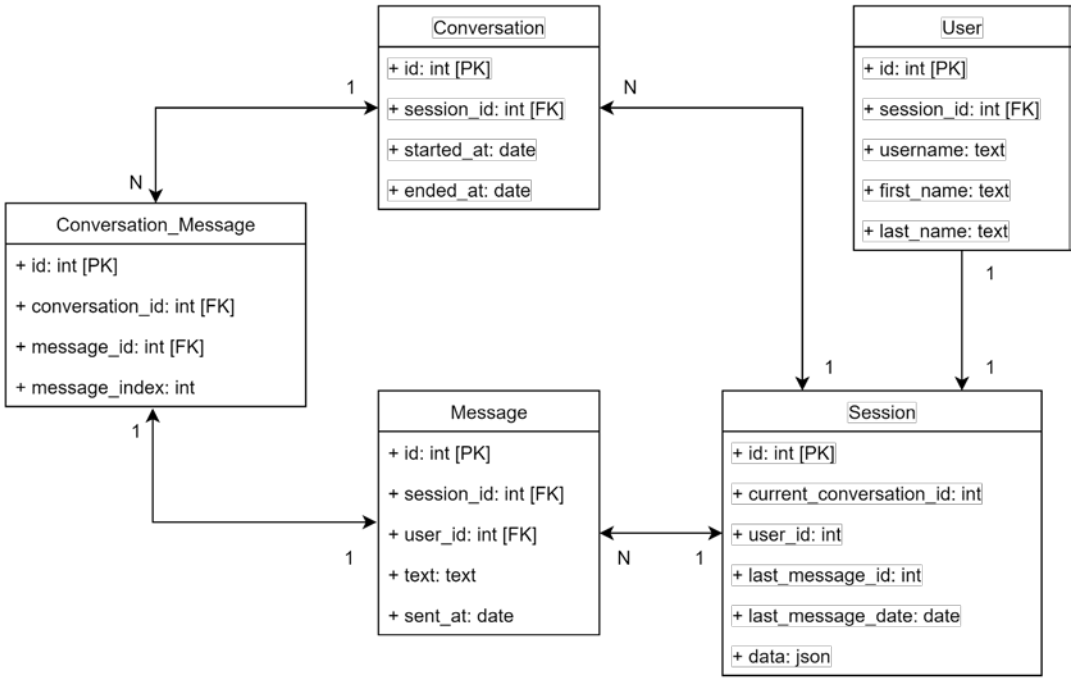

FIGURA 12: Modelo de entidades do banco de dados

Descrição das entidades apresentadas na Figura 12:

A. SESSION: descreve uma sessão única entre o usuário e o bot
   a. Id: id da sessão
   b. Current_conversation_id: id da atual conversa daquela sessão
   c. User_id; id do usuário associado a sessão
   d. Last_message_id: id da última mensagem enviada naquela sessão
   e. Last_message_date: data de envio da última mensagem
   f. Data: dados do estado atual da sessão serializados no formato JSON

B. CONVERSATION: descreve uma conversa entre o usuário e o bot
   a. Id: id da conversa
   b. Session_id: id da sessão à qual a conversa está associada
   c. Started_at: data e horário de início da conversa
   d. Ended _ad: data e horário de término da conversa

C. MESSAGE: descreve uma mensagem enviada em uma conversa
   a. Id: id da conversa
   b. Session_id: id da sessão à qual a mensagem está associada
   c. User_id: id do usuário que enviou a mensagem
   d. Text: conteúdo da mensagem
   e. Sent_at: data e horário de envio da mensagem

D. CONVERSATION_MESSAGE: descreve a relação entre conversa e mensagem
   a. Id: id da relação
   b. Conversation_id: id da conversa associada
   c. Message_id: id da mensagem associada
   d. Message_index: índice da mensagem na conversa de acordo com a ordem de envio

E. USER: descreve um usuário
   a. Id: id do usuário (obtido do Telegram)
   b. Session_id: id da sessão associada ao usuário
   c. Username: nome de usuário
   d. First_name: primeiro nome do usuário (como está no Telegram)
   e. Last_name: sobrenome do usuário (como está no Telegram)

5.5. Imagem no Docker

Para rodar o projeto, foi usado o Docker, uma ferramenta usada para criar aplicações em ambientes isolados denominados containers. Com isso é possível empacotar o programa para poder ser executado em qualquer máquina que possua o Docker.

Assim, foi desenvolvido um Dockerfile, representado no código abaixo, contendo as definições do container desse projeto, que pode ser executado usando o docker-compose. O arquivo docker-compose presente no projeto com as especificações abaixo também executa, junto com a aplicação principal, um container com um banco de dados postgresql, que passa a ser usado pelo programa.

```
version: '3.1'

services:
 db:
   image: postgres
   restart: always
   env_file:
     - .env
   volumes:
     - ./postgres-data:/var/lib/postgresql/data
   ports:
     - "5432:5432"
 app:
   build: .
   environment:
     POSTGRES_HOST: db
   Env_file:
     - .env
   depends_on:
     - db
   volumes:
     - ./data:/data
   restart: on-failure
   ports:
     - "8081:8081"
```

6. Implementação e Avaliação

6.1. Dificuldades Encontradas

Ao longo do desenvolvimento do projeto, foram encontrados diversos desafios, dentre os quais valem destacar a dificuldade de encontrar um dataset com o tamanho e a língua desejada e a capacidade do hardware no qual foi treinada a rede neural.

O dataset usado para treinamento de Danescu-Niculescu-Mizil [20] foi adequado para a realização dos testes dos parâmetros da rede neural e para garantir que o fluxo do programa, incluindo a conexão com o banco de dados e com o Telegram, estava funcionando corretamente. No entanto, é desejável que o chatbot converse em língua portuguesa, mas tivemos dificuldade em encontrar um dataset que não estivesse em inglês. Após pesquisa, encontramos o dataset do projeto *chatterbot-corpus* [21], que inclui alguns conjuntos de conversas como introduções, elogios e conhecimentos gerais em língua portuguesa.

Apesar desse dataset oferecer material em português, ele tem uma limitação de tamanho, contendo apenas 346 pares de mensagens, significativamente menor do que o de Danescu-Niculescu-Mizil [20], que contém 11335 pares. Mesmo assim, como parte desse projeto envolve o armazenamento de dados para geração de um dataset mais completo, decidimos por fazer o treinamento da rede com o conjunto de dados do *chatterbot-corpus* [21].

O computador no qual os testes foram feitos e a rede foi desenvolvida possui as seguintes especificações:

CPU: Intel Core i7-10510U
GPU: NVIDIA GeForce MX250 (10W)
RAM: 16GB

Essas especificações são suficientes para gerar um protótipo inicial e validar a viabilidade do projeto, porém, uma GPU mais potente seria ideal se quisermos melhorar o tempo de treinamento e acurácia da rede, principalmente se os datasets usados tiverem volumes maiores.

6.2. Criação da Rede Neural

Com os resultados dos testes, o modelo da rede neural foi criado com os seguintes parâmetros:

Tipo do modelo de atenção: dot
Número de camadas ocultas do *Encoder*: 2
Número de camadas ocultas do *Decoder:* 2
Número de nós da camada oculta: 500
Taxa de aprendizado: 0.0001
Fator de aprendizado do *Decoder*: 5.0
Taxa de aprendizado forçado: 1.0
Taxa de dropout: 0.1
Gradiente de corte: 50.0
Tamanho do lote: 64

O treinamento foi feito com 1000 iterações e teve como resultado a seguinte curva de perda:

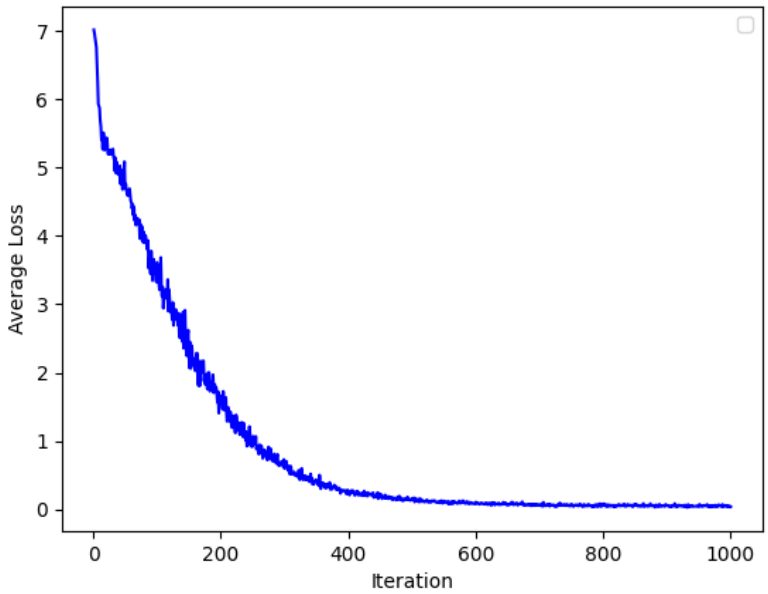

FIGURA 13: Gráfico ilustrando as curvas de perda média ao longo do treinamento

É possível ver que a rede conseguiu atingir uma perda baixa durante o treinamento e aprender com a base do projeto *chatterbot-corpus* [21], como ilustrado na Figura 14.

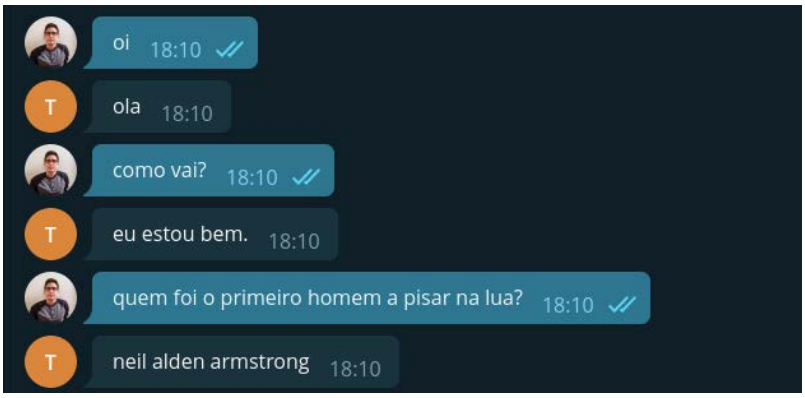

FIGURA 14: Exemplo de uma conversa com a rede treinada com a base em língua portuguesa

6.3. Retreinamento da Rede

Após confirmar que o treinamento inicial havia funcionado, o próximo passo foi avaliar a capacidade da rede de aprender com as mensagens dos usuários através de um novo treinamento. Para isso, primeiro foi identificada uma mensagem que o robô não sabia responder, para então sugerir a ele a resposta correta, como ilustrado na Figura 15.

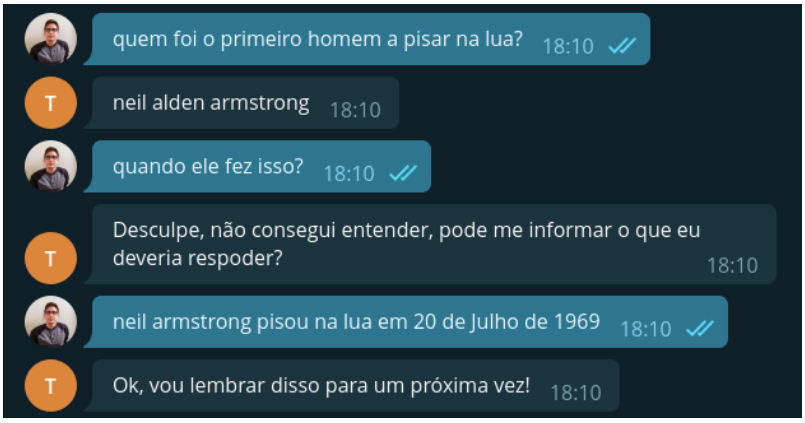

FIGURA 15: Exemplo do chatbot solicitando ao usuário uma resposta adequada para uma pergunta que não soube responder

Com isso, essa nova resposta é armazenada no banco de dados e, quando for executado um novo treinamento, o chatbot passará a saber responder essa pergunta, como ilustrado na Figura 16.

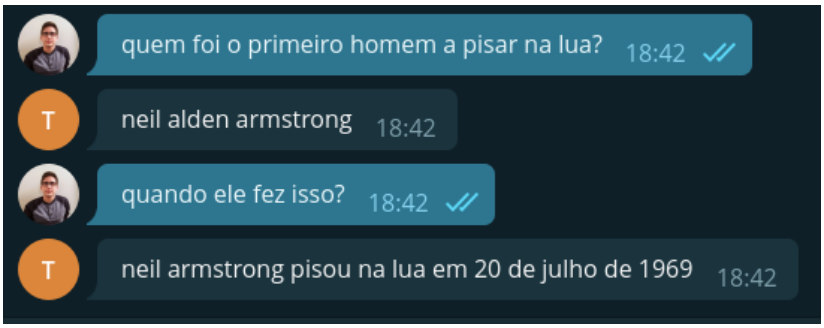

FIGURA 16: exemplo do chatbot respondendo uma pergunta que não soube responder préviamente após um novo treinamento

7. Considerações Finais

Entendemos que o projeto demonstrou a possibilidade de construir um chatbot que, acoplado a uma rede neural desenvolvida com o Pytorch, consegue aproveitar efetivamente os inputs dos usuários para aprender e desenvolver uma base de conhecimentos mais ampla através de novas séries de treinamento. Essa rede foi implementada usando um método de pontuação para assegurar que as respostas dadas aos usuários são as melhores, e o seu modelo foi construído usando os melhores parâmetros encontrados de acordo com os testes realizados ao longo do trabalho. O projeto também permitiu validar a possibilidade de integrar um modelo de rede neural com um robô na aplicação do Telegram e com um banco de dados para a geração de uma base de dados própria.

No entanto, houve desafios ao longo do desenvolvimento, sendo o mais relevante a dificuldade de acesso a uma base de dados em língua portuguesa. Apesar de estar em português, o dataset que foi usado para o treinamento da rede era ainda muito pequeno para o propósito, o que limita o aprendizado da rede durante o treinamento.

Entendemos que um possível próximo passo para dar sequência a este trabalho poderia ser realizar novos testes para otimizar os parâmetros estruturais do modelo da rede neural, visando a melhorar a eficiência do chatbot. Outra possibilidade, mais complexa, seria explorar o recurso em chatbots específicos, por exemplo, dirigido à área de saúde ou relativo ao comércio. Nesse caso, seria necessário ter uma base de dados inicial particular, que tenha como elementos expressões e termos correspondentes.

8. Referências Bibliográficas

[1] Adamopoulou, E., & Moussiades, L. (2020, June). An overview of chatbot technology. In IFIP International Conference on Artificial Intelligence Applications and Innovations (pp. 373-383). Springer. Cham.

[2] Scopus - Document search. https://www.scopus.com/search/form.uri?displ ay= basic.

[3] Verhulst, S.G. Where and when AI and CI meet: exploring the intersection of artificial and collective intelligence towards the goal of innovating how we govern. AI & Soc 33, 293–297 (2018).

[4] Skodowski, Michelle. https://chatbotsmagazine.com/how-to-write-user-friendly-error-messages-41e66a77a026. Publicado em 16/04/2018. Acessado em 19/02/2022.

[5] Abdul-Kader, S.A. and Woods, J.C., 2015. Survey on chatbot design techniques in speech conversation systems. International Journal of Advanced Computer Science and Applications, 6(7).

[6] K. Meffert, "Supporting design patterns with annotations." pp. 8 pp.- 445, 2006.

[7] D. Mladenić, and L. Bradeško, “A survey of chabot system through a Loebner prize competition,” 2012.

[8] Følstad, A., & Skjuve, M. (2019, August). chatbots for customer service: user experience and motivation. In Proceedings of the 1st international conference on conversational user interfaces (pp. 1-9).

[9] Følstad, A., Nordheim, C. B., & Bjørkli, C. A. (2018, October). What makes users trust a chatbot for customer service? An exploratory interview study. In *International conference on internet science* (pp. 194-208). Springer, Cham.

[10] Maeda, A., & Moraes, S. (2017, October). chatbot baseado em deep learning: um estudo para língua portuguesa. In *Symposium on Knowledge Discovery, Mining and Learning, 5th*.

[11] Vinyals, Oriol & Le, Quoc. (2015). A Neural Conversational Model. ICML Deep Learning Workshop, 2015.

[12] Sutskever, I., Vynyaks, O., and Le, Q. V. Sequence to sequence learning with neural networks. In Advances in Neural Information Processing Systems 27, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., pp. 3104–3112, 2014.

[13] Serban, I.V., Sankar, C., Germain, M., Zhang, S., Lin, Z., Subramanian, S., Kim, T., Pieper, M., Chandar, S., Ke, N.R. and Rajeshwar, S., 2017. A deep reinforcement learning chatbot. arXiv preprint arXiv:1709.02349.

[14] Li, J. (2016). Deep Reinforcement Learning for Dialogue Generation. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (pp. 1192–1202). Association for Computational Linguistics.

[15] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32, pp.8026-8037.

[16] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32, pp.8026-8037.

[17] Inkawhich, Matthew. Tutorial de um chatbot básico desenvolvido com Pytorch https://pytorch.org/tutorials/beginner/chatbot_tutorial.html

[18] Luong, C. (2015). Effective Approaches to Attention-based Neural Machine Translation. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (pp. 1412–1421). Association for Computational Linguistics.

[19] Gupta, A., Tyagi, S., Panwar, N., Sachdeva, S. and Saxena, U., 2017, October. NoSQL databases: Critical analysis and comparison. In 2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN) (pp. 293-299). IEEE.

[20] Danescu-Niculescu-Mizil, Cristian. Dataset disponível em https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

[21] ChatterBot Language Training Corpus - A multilingual dialog corpus. Código Fonte disponível em: https://github.com/gunthercox/ chatterbot-corpus