UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES
CAMPUS DE ERECHIM
DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

KELWIN KOMKA

CHATBOT INTEGRADO A APLICATIVO DE MENSAGENS E PLATAFORMA DE *HELP DESK*

ERECHIM - RS
2020

KELWIN KOMKA

CHATBOT INTEGRADO A APLICATIVO DE MENSAGENS E PLATAFORMA DE *HELP DESK*

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel, Departamento de Engenharias e Ciência da Computação da Universidade Regional Integrada do Alto Uruguai e das Missões Campus de Erechim.

Orientador: Prof. Me. Marcos André Lucas

ERECHIM - RS
2020

AGRADECIMENTOS

Agradeço a Deus por me prover saúde, força de vontade e perseverança, necessárias para a conclusão deste trabalho. À minha mãe, que sempre compreendeu o que eu buscava fazer na vida, sem poupar esforços para me auxiliar e me encorajar a seguir este caminho. Aos professores, que sempre estavam de prontidão para nos ensinar e auxiliar a superar as dificuldades. Em especial ao meu orientador, prof. Marcos Lucas, que me ajudou com sugestões e informações para produção desta monografia. À meus colegas de trabalho e faculdade que sempre colaboraram também para o meu aprendizado, não só profissional, mas também pessoal. Por fim, à URI Erechim, por prover um estabelecimento e salas equipadas para um melhor aprendizado, além dos diversos cursos que enriqueceram a nossa sabedoria.

RESUMO

Os meios de comunicação expandiram com a utilização comercial da internet e, com esta expansão, diversas necessidades cresceram na mesma escala, sendo uma delas a demanda por um atendimento de suporte rápido e de qualidade. Este tipo de serviço ao cliente deve ser bem planejado, para que não haja a possibilidade de problemas e atrasos. Atualmente, a automação destes serviços pode ser a chave para assegurar que o atendimento seja prestado da melhor forma possível. Este trabalho apresenta um sistema de *chatbot* com integração à plataformas para automatização do serviço de suporte. Esta ferramenta utiliza a linguagem AIML e o modelo *Sequence to Sequence* como agentes inteligentes. Também, integra com um aplicativo de mensagens instantâneas, o Telegram e, com a plataforma de *help desk* JIRA. Seu uso deve automatizar a entrada de informações por parte dos clientes e melhorar a experiência do atendimento *service desk*.

Palavras-chave: Automação. Aprendizado de Máquina. Integração.

ABSTRACT

The means of communication expanded with the commercial use of the internet and, with this expansion, several needs grew on the same scale, one of them being the demand for fast and quality support service. This type of customer service must be well planned, so that there is no possibility of problems and delays. Currently, the automation of these services may be the key to ensure that the service is provided in the best possible way. This work presents a chatbot system with integration to platforms, for automation of the support service. This tool uses the AIML language and the Sequence to Sequence model as intelligent agents. It also integrates with an instant messaging application, Telegram and, a help desk platform, JIRA. Its use should automate the entry of information by customers and improve the customer service experience.

Keywords: Automation. Machine Learning. Integration.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Com a inclusão de empresas na internet, houve a evolução dos meios de comunicação e, com isso, o aumento da demanda por um atendimento rápido e de qualidade. Este tipo de demanda pode acarretar em problemas com, a qualidade do atendimento e organização de informações a cerca de ocorrências feitas por clientes, para empresas que não utilizarem soluções para gerenciar e suprir a necessidade de comunicação com seus clientes. Uma ferramenta que consiga manter e melhorar a qualidade do serviço pode ser a solução para o problema.

Os *chatbots* são eficientes para desempenhar a função de atendimento ao público através de plataformas próprias ou de canais de comunicação já existentes. A empresa Accenture aponta que:

> Em uma empresa de telecomunicações européia, um chatbot foi usado em um programa piloto em um conjunto de consultas comuns de clientes e resolveu 82% das interações sozinho, aumentando para 88% das interações quando combinado com a intervenção ao vivo de um agente humano. (ACCENTURE, 2016, p. 5, tradução nossa)

Isso demonstra que *chatbots* conseguem atender grande parte da demanda de forma independente, além de proporcionarem rapidez no atendimento de diversos clientes.

Como mencionado, além do aumento da demanda do atendimento ao cliente, as formas de contato expandiram. Dessa forma, diversas empresas abrem a possibilidade de contato por redes sociais ou plataformas de comunicação populares. Porém, muitas vezes, por falta de planejamento ou comunicação interna, não há controle e registro de atendimentos realizados por estes canais, possibilitando a perda de informação e atraso no serviço para o cliente.

O sistema prove um *chatbot* com utilização de inteligência artificial, especificamente *machine learning*, para que, dessa forma, a qualidade do atendimento ao cliente seja aprimorada ao longo do tempo.

A ferramenta visa a integração com plataformas de comunicação familiares ao usuário, permitindo o uso de canais de conversa em redes sociais e aplicativos. Para controle destas informações e gerenciamento das solicitações, é disponibilizada a opção de integração à plataformas *help desk*, que gerenciam chamados e projetos de empresas. Desta forma, todo processo de atendimento do cliente até a entrada de chamados será automatizado, gerando qualidade e eficiência ao serviço.

Espera-se que a construção e uso deste sistema, buscando automatizar o processo de atendimento, gere uma melhora expressiva em termos de velocidade, comodidade e organização. A aplicação dos modelos de *machine learning* devem proporcionar esta melhora, enriquecendo a experiência de utilização do *chatbot*.

A estrutura do texto apresenta os tópicos de forma linear, buscando apresentar os conceitos básicos para melhor compreensão dos capítulos posteriores.

O Capítulo 2 apresenta uma ideia geral sobre *chatbots*, apresentado algumas funções e objetivos conhecidos. O processamento de linguagem natural, que utiliza técnicas de tokenização e normalização, junto do uso de *intents* e entidades para auxílio no processamento da mensagem. E a AIML, uma linguagem que possibilita a construção de *chatterbots* utilizando um interpretador e o algoritmo *Graphmaster* para busca de padrões.

No Capítulo 3 é explicado sobre o que é *machine learning*, e qual o seu propósito. Sobre agentes, definidos por serem entidades "racionais", e quais são os tipos existentes. Os tipos de aprendizados, supervisionado, não supervisionado e por reforço são apresentados. Por fim, existem modelos e algoritmos explorados, como o *Actor-critic with experience replay*, *Automatic Neural Network* e *Deep Semantic Similarity Model*, que estão presentes em trabalhos relacionados à área.

O Capítulo 4 informa sobre o modelo utilizado neste trabalho, o *Sequente to Sequence*, que é frequentemente utilizado para tradutores. Há a explicação sobre *Recurrent Neural Networks* e *Long Short Term Memory*, que constituem o modelo. Também está presente o mecanismo de atenção, utilizado para resolver um problema de limitação do modelo.

No Capítulo 5, são apresentados a plataforma de *help desk* JIRA e o aplicativo de mensagens Telegram, pois são ferramentas auxiliares que complementam o uso do *chatbot*.

O Capítulo 6 apresenta os requisitos funcionais e não funcionais do sistema, que definem o escopo do trabalho. A modelagem demonstra o fluxo e funcionamento esperado do sistema, com um caso de uso frequente. são vistas as ferramentas utilizadas para o desenvolvimento do trabalho, sendo as principais o LaTeXpara a documentação e o *Python* para a construção do sistema, e também as bibliotecas necessárias para o funcionamento do *chatbot*, como a TensorFlow e a Program-Y. A descrição do sistema, que explica pontos importantes do código-fonte, como a criação de chamados, o treino e previsão do modelo usado.

No Capítulo 7 é apresentada visualmente a demonstração do uso da ferramenta, abordando alguns casos de uso como, a criação de chamados com seus resultados positivos (chamado criado) e negativos (cancelamento da criação do chamado), conversas com o *bot* e atendentes. Uma visualização do treino realizado, onde é possível ver o progresso através de um terminal. E de forma adicional, alguns comandos disponíveis à usuários cadastrados como atendentes e administradores.

O Capítulo 8 conclui as ideias sobre o trabalho, apresentando os desafios enfrentados, problemas ocorridos e resultados observados durante o desenvolvimento. Por fim, alguns trabalhos futuros são descritos, dentre eles a construção de uma interface gráfica, o aprimoramento do modelo ou uso de outros modelos e, a integração com outras plataformas presentes no mercado.

2 CHATBOTS

Os *chatbots* podem ser definidos como "um programa de computador que processa a entrada em linguagem natural de um usuário e gera respostas inteligentes e relativas que são enviadas de volta ao usuário"(KHAN; DAS, 2018, p. 1, tradução nossa). Estão em crescente uso por empresas para aprimoramento do atendimento ao cliente, pois auxiliam e aceleram este serviço. Como Raj (2019, p. 2) aponta, desde 2016 a busca por *chatbots* na internet cresceu exponencialmente e a produtividade se destaca como o principal motivo da utilização desta tecnologia.

Segundo Castro e Barros (2015, p. 1), os *chatterbots* podem ter diversos tipos de finalidades, como: entretenimento, responder perguntas frequentes, marketing, gerais e suporte ao consumidor. A finalidade de suporte ao consumidor ou cliente, é a designada para este trabalho, pois seu objetivo é auxiliar e suprir a demanda em um setor de suporte, buscando aprender sobre o ambiente e proporcionar o mesmo atendimento que um humano faria.

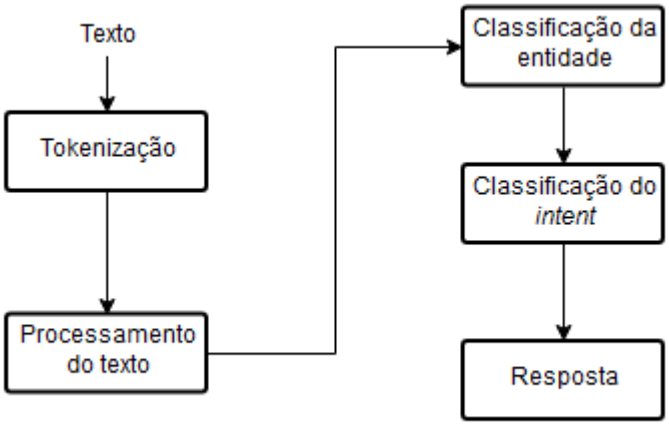
Muitos dos *chatbots* são desenvolvidos utilizando várias técnicas e algoritmos de IA, que auxiliam a compreensão de textos e entradas para o *bot*, dessa forma, aprimoram a acuracidade da resposta gerada. Sendo a NLP (*Natural Language Processing*), ou o Processamento de Linguagem Natural, a técnica utilizada neste trabalho, pois esta irá processar uma mensagem, separando as palavras úteis e visualizando o contexto da frase, assim, criando uma entrada mais adequada para o processamento do *chatterbot*. (WIJAYA; RAHMADDENI; ZOROMI, 2020)

De acordo com Cahn (2017, p. 5, tradução nossa), os *chatbots* possuem métodos de avaliação, para que seja verificado se seu objetivo está mais próximo ou mais distante. Como exemplo, há *chatbots* para recuperação de informação, onde sua eficácia é avaliada por: precisão, acuracidade e velocidade na recuperação. Para *bots* focados em interação com usuários, sua avaliação é feita pelo *feedback* dos usuários sobre sua usabilidade e satisfação. E os focados em conversa, são medidos por sua acuracidade em gerar uma sentença de melhor sentido e que se encaixa no contexto.

2.1 Processamento de Linguagem Natural

O processamento de linguagem natural é utilizado para melhor compreensão do funcionamento de um *chatbot*, pois fragmentam o processamento de mensagens, focando nas partes importantes para que o resultado seja efetivo. A Figura 1 apresenta um fluxograma da execução do *chatbot* sobre uma mensagem enviada, em que, cada passo é uma operação a ser realizada para filtrar e categorizar o texto, direcionando, assim, o sistema à resposta.

Figura 1 – Fluxograma de uma execução do chatbot



Fonte: Adaptado de Manaswi (2018, p. 147)

2.1.1 Tokenização e Processamento do texto

Estas etapas têm o papel de listar e manter apenas as palavras-chave que serão transformadas em ***tokens***, sendo que cada espaço em uma frase delimita este *token*, criando, assim, uma lista de palavras a serem utilizadas para a execução.

Após esta fragmentação, o processamento do texto pode ser categorizado como filtro para vírgulas e pontos de exclamação/interrogação, pois são símbolos desnecessários para a interpretação do contexto e, também, uma seleção para palavras que serão definidas como entidades ou demonstram a intenção do usuário. Manaswi (2018, p. 147) considera os *tokens* descartados como *stop words*, por não terem importância na compreensão do texto pelo sistema.

2.1.2 *Intents*

A intenção ou propósito do usuário é categorizado como um *intent*, que será utilizado para guiar o sistema ao caminho relevante. Khan e Das (2018, p. 1) explica que cada *chatbot* terá uma lista de *intents* específicos para sua área de atuação. No caso deste trabalho, pode-se exemplificar com uma entrada de um cliente a um serviço de suporte, onde o cliente informa ao bot a seguinte mensagem: "Está acontecendo um erro no módulo de cadastro...". Esta frase apresenta a intenção de informar um erro, neste caso o sistema deve direcionar esta conversa ao *intent* chamado *erro_modulo*. Esta técnica é chamada de *text classification* (classificação de texto), onde o programa deve classificar as frases e *tokens* para encontrar o contexto adequado.

2.1.3 Entidades

As entidades são como variáveis para um *intent*, onde estas guardam um *token* como valor. Raj (2019, p. 3) demonstra que podem haver várias para um contexto e são consideradas como metadados do *intent*. Utilizando o exemplo da Subseção 2.1.2, pode-se ter duas entidades

para especificar o problema do cliente:

- ocorrência: erro.
- módulo: cadastro.

Assim, o sistema pode buscar informações e retornar respostas para solução de erros que ocorreram no módulo de cadastro. Quanto mais específica a mensagem, mais entidades podem ser utilizadas, facilitando a busca de informações pertinentes para a resolução do processo.

2.2 AIML

Segundo AIML Foundation (2018, tradução nossa), a "AIML é uma linguagem baseada na estrutura de dados do XML para especificar o conteúdo do chatbot. Um interpretador AIML é um programa capaz de carregar e executar o *bot* e fornecer as respostas em uma sessão de chat com um usuário humano, chamado de cliente.". Desta forma, facilita a criação de estruturas de diálogo. Foi desenvolvida por Richard Wallace no final da década de 90, onde conseguiu diversos prêmios na área da inteligência artificial com seu *bot* A.L.I.C.E., que utilizava da AIML para encontrar padrões e, relacionar uma entrada do usuário com uma resposta de uma base de dados.

Ela é baseada em XML, possuindo uma estrutura semelhante, com *tags* específicas para a criação de diálogos. Nela, um objeto AIML é definido, sendo este o responsável pela modelagem da conversa. (MARIETTO et al., 2020)

Segundo Wijaya, Rahmaddeni e Zoromi (2020, p. 4, tradução nossa) "O elemento AIML especula o conhecimento na forma de estímulo-resposta no documento. A AIML contém uma coleção de padrões e respostas que podem ser usados por chatbots para procurar respostas para cada frase dada". A linguagem auxilia a construção de chatbots, possibilitando que o *chatbot* consiga processar a sentença de entrada com base nos padrões definidos no objeto criado.

2.2.1 *Wildcards*

As *wildcards* são símbolos que podem ser utilizados dentro da AIML para representar intervalos variáveis dentro de uma sentença, proporcionando uma customização da entrada e possibilitando que o interpretador aja de forma mais otimizada, além de permitir armazenar informações da entrada para melhorar a acurácia do *bot*. (Pandorabots, 2020)

Cada símbolo tem uma prioridade e uma função. Abaixo serão listadas as prioridades, do interpretador, em ordem descendente, junto com a função de cada símbolo.

- **$**: Atribui a prioridade máxima em uma sentença, e.g. "Olá, me chamo $", portanto, qualquer frase que coincida, será compatível obrigatóriamente com o padrão.

- **#**: Alta prioridade para zero ou mais palavras, e.g. "# me chamo #", fazendo com que qualquer frase que possua as palavras "me chamo"coincida com o padrão.
- _: Alta prioridade para uma ou mais palavras, e.g. "Qual o seu _ ?", desta forma, a sentença irá coincidir, caso possua uma ou mais palavras após as palavras "Qual o seu".
- **Sentença**: Valor igual à sentença de um padrão declarado, e.g. "Bom dia".
- ^: Baixa prioridade para zero ou mais palavras, e.g. "Como vai ^?", semelhante ao segundo item, porém com uma prioridade menor.
- *: Baixa prioridade para uma ou mais palavras, e.g. "* está pronto?", semelhante ao terceiro item, porém com a menor prioridade.

Na primeira versão da linguagem (1.0), os símbolos # e ^, não estavam presentes. A partir da versão 2.0, estes foram adicionados, assim, facilitando a criação de padrões de sentenças e tratamentos de frases complexas.

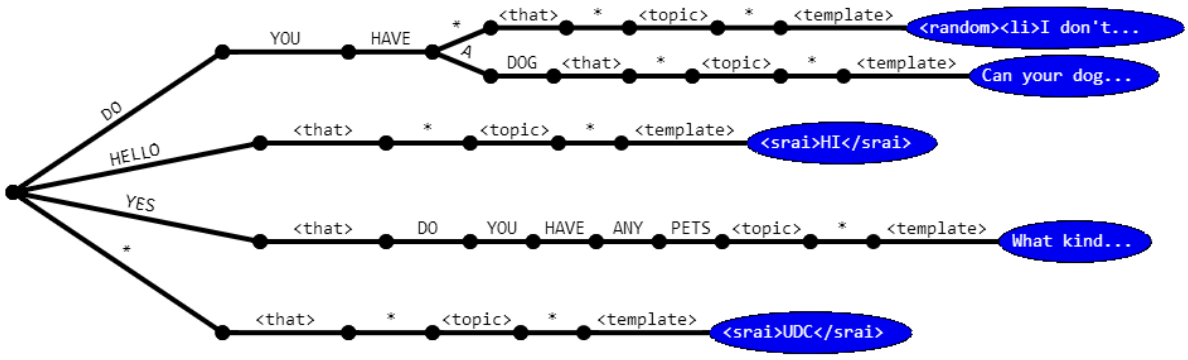
2.2.2 Processamento e reconhecimento de padrões

Marietto et al. (2020, p. 3, tradução nossa) apontam que o processamento pode ser dividido em três etapas:

> "A primeira etapa é a entrada da sentença pelo usuário. Na segunda etapa, o sistema executa ações de processamento de texto ajustando a entrada para um formato pré-estabelecido pelo *designer* e faz a correspondência de padrões entre esta entrada e a base de conhecimento. Finalmente, uma resposta é apresentada ao usuário na terceira etapa."

Todo este processamento é realizado pelo algoritmo *Graphmaster*, utilizado no *bot* ALICE e, que serve como base de diversos interpretadores construídos posteriormente. Wallace (2003, p. 21, tradução nossa) explica que o algoritmo "consiste em uma coleção de nós chamados *Nodemappers*. Esses *Nodemappers* mapeiam as ramificações de cada nó. Os ramos são palavras simples ou *wildcards*."Sendo que, cada ponta destes nós é reconhecida por possuir uma *tag* <*template*>.

Figura 2 – *Graphmaster* simples com cinco categorias



Fonte: AIML Foundation (2018)

Wallace (2003, p. 21) explana que este algoritmo passa por três etapas para realizar o reconhecimento de um padrão, sendo que, caso não seja encontrado um padrão equivalente, o algoritmo vai para a próxima etapa:

1. Verifica se há categorias com *wildcards* de alta prioridade - como demonstrado na Subseção 2.2.1 - e caso existam, percorre os nós filhos que contenham a entrada.
2. Pesquisa pela sentença de entrada e, caso seja encontrada, irá percorrer os nós filhos, removendo o sufixo da entrada.
3. Busca por categorias que possuam *wildcards* de baixa prioridade e, se estas existirem, pesquisa nos nós filhos por padrões que possuam a entrada

Então, caso o algoritmo encontre a *tag* <*template*>, esta é reconhecida como uma resposta válida, assim, a busca é interrompida e a resposta retornada. Em casos onde haja um nó, que possua somente a *wildcard* * como padrão e seja considerado ponta da ramificação, é garantido que o algoritmo retorne uma resposta.

2.2.3 *Tags*

As *tags* definem uma função a ser desempenhada ou declarada, assim, construindo um fluxo para o interpretador seguir e retornar uma resposta. A seguir haverá uma explicação das principais *tags* utilizadas no desenvolvimento deste trabalho. A documentação completa pode ser encontrada no site da AIML Foundation (2018).

2.2.3.1 <*aiml*>

Define o corpo do objeto AIML. No corpo da *tag* há o parâmetro *version* que define qual a versão do AIML utilizada. Este trabalho utiliza a versão 2.0 da linguagem.

Quadro 1 – Definição da *tag* <*aiml*>.

```
<aiml version="2.0">
...
</aiml>
```

Fonte: Autor

2.2.3.2 <*category*>

Unidade básica de conhecimento. Dentro do escopo é obrigatório a existência das *tags* <*pattern*> e <*template*>. Outras tags podem estar declaradas opcionalmente no escopo.

Quadro 2 – Definição da *tag* <*category*>.

```
<category>
    <pattern>OLA ^</pattern>
    <template>
        <srai>OI</srai>
    </template>
</category>
```

Fonte: Autor

2.2.3.3 <*pattern*>

Define um padrão de entrada. Pode conter uma sentença constante ou pode ser variante com a utilização de *wildcards* (e.g. ^, *).

Quadro 3 – Definição da *tag* <*pattern*>.

```
...
<pattern>* CRIAR UM CHAMADO *</pattern>
...
```

Fonte: Autor

2.2.3.4 <*template*>

Define a resposta para um categoria. Segundo a AIML Foundation (2018, tradução nossa) "Em geral, entretanto, a *tag* <*template*> contém o que é, na verdade, um mini programa

de computador, escrito com *tags* AIML, que calculam uma resposta".

Quadro 4 – Definição da *tag* <*template*>.

```
...
<template>Tudo bem, não vou criar um chamado.</template>
...
```

Fonte: Autor

3 *MACHINE LEARNING*

O aprendizado de máquina, conhecido como *machine learning*, de acordo com Wijaya, Rahmaddeni e Zoromi (2020) é "uma sub-área da ciência da computação e inteligência artificial que evoluiu do estudo da teoria de aprendizado computacional e, reconhecimento de padrões". Para isso, existem algoritmos que geram - a partir do conhecimento dos dados - saídas de dados com base nos padrões conhecidos. No caso de um *chatbot*, é necessária uma quantidade grande de dados para que a solução de um problema seja gerada, havendo a possibilidade de construir um processo contínuo de aprendizado para aprimoramento do *bot*. (Smart Sheet, 2019)

3.1 Agente

Dentro do campo da inteligência artificial existem entidades "racionais", conhecidas como **agentes**, definidos por Russell e Norvig (2010, p. 34, tradução nossa) como "qualquer coisa que possa ser vista percebendo seu **ambiente** por meio de **sensores** e agindo sobre esse ambiente por meio de **atuadores**.". Pode-se entender por isso que, o sistema que executa o algoritmo é o agente. Existem cinco tipos de agentes, segundo o artigo de EDUCBA (2020) são eles:

- **agentes de reflexo simples**: que apenas reagem a algum evento e não armazenam informações sobre o estado atual;
- **agentes baseados em modelos**: que são semelhantes aos de reflexo simples, porém armazena os estados em que esteve e, utiliza-os para chegar ao próximo estado;
- **agentes agentes baseados em objetivos**: onde buscam diminuir a distância entre o estado atual e o objetivo desejado, priorizando diminuir esta distância em detrimento da melhor alternativa;
- **agentes utilitários**: onde estes agem de acordo com o estado atual para chegar ao seu objetivo, buscando sempre a melhor alternativa para a solução;
- **agentes que aprendem**: possuindo métodos de aprendizado, havendo a possibilidade de iniciar com poucos dados e ir aprendendo de seu ambiente.

Um agente possuirá um ambiente de tarefas, como Russell e Norvig (2010, p. 40, tradução nossa) explica que "nós tivemos que especificar a medida de desempenho, o ambiente e os atuadores e sensores do agente. Nós agrupamos tudo isso sob o título de **ambiente de tarefas**.". Este ambiente de tarefas terá dados sobre: avaliação do desempenho, ambiente que está inserido, atuadores e sensores.

Tipo de agente	Avaliação de desempenho	Ambiente	Atuadores	Sensores
Chatbot	resposta correta e satisfatória	mensagens, diálogos	sistema, usuário	algoritmo de avaliação, tempo de conversa

Quadro 5 – Ambiente de tarefas do *chatbot*

O Quadro 5, baseado em Russell e Norvig (2010, p. 41), apresenta o ambiente de tarefas do *chatbot* desenvolvido neste trabalho. O tipo de agente é o *chatbot*, pois é o nosso sistema e algoritmo/modelo. A avaliação de desempenho deve verificar e criticar as respostas, buscando sempre a correta e avaliando se está satisfatória para o usuário. O ambiente do agente são as plataformas integradas, que possuem mensagens e diálogos. Os atuadores no ambiente são o sistema e o usuário, que poderão realizar ações. Os sensores são, o algoritmo de avaliação e o tempo de conversa, pois são dois métodos para auxiliar na avaliação de desempenho.

3.2 Tipos de aprendizados

Em *machine learning*, há várias formas de fazer o programa aprender para desempenhar a função esperada. No algoritmo de aprendizado **supervisionado**, a máquina irá treinar com uma base de dados, onde pode-se dividir os dados para treino e para teste, categorizando e interpretando cada objeto (dado) que pode possuir diversos atributos e, a partir deste atributos, predizer valores novos para atributos de objetos não categorizados. (GUPTA, 2020)

O algoritmo **não supervisionado** irá obter os objetos como o algoritmo anterior, porém estes não possuem atributos, pois o algoritmo irá identificar estes atributos e chegará a uma nova lista de objetos categorizados ou a um único resultado. Com esta forma de processamento, é possível fazer a função de *outlier detection*, uma análise de valores para verificar se pertencem ou não e quanto diferem do grupo em específico. Há também o algoritmo semi supervisionado que junta tanto objetos com e sem atributos. Porém, estes últimos, em quantidade maior, com o mesmo objetivo do supervisionado mas permitindo a possibilidade de melhora do modelo por parte da máquina. (WOOD, 2020)

O aprendizado **por reforço** utiliza o ambiente em que a máquina está executando e faz com que ela analise o estado deste ambiente e suas características. Gerando um resultado a partir desta análise, a máquina irá receber uma **recompensa**, um *feedback*, que irá categorizar seu comportamento como certo ou errado, fazendo com que comportamentos corretos sejam mais propensos a ocorrer. Esta forma de tentativa e erro indica para o sistema qual peso de cada caminho que pode ser escolhido para uma situação e, caso o caminho adequado seja escolhido, haverá um reforço para que o mesmo ocorra na próxima iteração. (BURKOV, 2019, p. 3, 4)

3.3 Modelos e algoritmos

Os modelos e algoritmos definem o funcionamento de uma técnica ou categoria da inteligência artificial, onde são direcionados a executar tarefa dentro de um escopo específico, pois há algoritmos que não serão eficientes dependendo do processo executado. Diversas técnicas podem ser aplicadas à resolução de um mesmo problema, porém, deve-se verificar quais os prós e contras de cada implementação.

Nas próximas sub-seções serão apresentados alguns modelos e algoritmos que possibilitam a construção de *chatterbots*, porém estes não foram utilizados no presente trabalho, estando apenas listados para demonstração de possibilidades e para corroborar na explicação do modelo escolhido.

3.3.1 *Actor-critic with experience replay*

O algoritmo ACER é composto por um método de *actor-critic* (ator crítico) e o algoritmo de *experience replay*.

Os métodos *actor-critic*, Mark Lee afirma que:

> São métodos de *Temporal-Difference Learning* que possuem uma estrutura de memória separada para representar explicitamente a política independente da função de valor. A estrutura da política é conhecida como **ator**, porque é utilizada para selecionar ações, e a função de valor estimado é conhecida como **crítica**, porque critica as ações realizadas pelo ator. (LEE, 2005, tradução nossa)

Então, o método acaba separando ambas partes (política e função de valor), fazendo com que uma parte aja e, outra avalie cada ação tomada, respectivamente.

Já o algoritmo de *experience replay*, segundo Fedus et al. (2020) "é o mecanismo de geração de dados fundamental no *deep reinforcement learning* (aprendizagem por reforço profundo) fora da política". O algoritmo consiste em um *buffer* que armazena a última transição recebida pela política de funcionamento. Está categorizado como um método de DQN (*Deep-Q-Networks*).

A mescla de ambos métodos citados, junto à um desenvolvimento aprofundado destas técnicas, cria o algoritmo ACER, sendo este focado em ultrapassar limitações do *Q-learning* como: a natureza determinística de politicas de otimização e a busca por *greedy actions* em grandes *action spaces*. (WANG et al., 2016, p. 1,2, tradução nossa). Weisz (2018, p. 6, tradução nossa) explica em seu artigo, que utiliza o algoritmo em um sistema de diálogo dentro de um *action space* grande.

3.3.2 *Automatic Neural Network*

A rede neural, segundo Abreu (2019, p. 4, tradução nossa), é considerada "uma rede neural artificial, composta por unidades de processamento, ou neurônios. Cada neurônio realiza uma função de transferência, onde se tem uma saída do neurônio, e uma função de ativação".

Estas redes, então, buscam simular o processo de pensamento humano e, a partir de uma entrada, processam os dados através de diversas camadas, onde estão presentes os neurônios.

Massaro, Maritati e Galiano (2018) propõem a utilização de uma ANN (*Automatic Neural Network*) para a construção de um *chatbot* de FAQ, onde, segundo eles, "A criação automática da rede neural foi implementada explorando alguns algoritmos capazes de processar a base de conhecimento do sistema sem a intervenção de atores humanos.".

O *bot* de FAQ utiliza a técnica chamada *one hot encoding* em sua rede neural, esta técnica cria uma matriz balanceada, utilizando as perguntas cadastradas previamente, permitindo criar um dicionário, onde cada palavra possui um índice. A partir disso, o algoritmo irá buscar uma nova entrada na matriz criada, comparando os índices e palavras, onde o resultado será a resposta à pergunta com o maior número de palavras compatíveis. (MASSARO; MARITATI; GALIANO, 2018)

3.3.3 *Deep Semantic Similarity Model*

Reddy (2017, tradução nossa) explica que o modelo "é uma *Deep Neural Network* usada para modelar similaridade semântica entre um par de *strings*. Em termos simples, a semelhança semântica de duas frases é a semelhança com base em seu significado, e o DSSM nos ajuda a capturar isso.". Junto a isso, a documentação da biblioteca CNTK (*Microsoft Cognitive Toolkit*), descreve que:

> O DSSM pode ser usado para desenvolver modelos semânticos latentes que projetam entidades de diferentes tipos em um espaço semântico comum de baixa dimensão para uma variedade de tarefas de aprendizado de máquina, como ranqueamento e classificação. (MICROSOFT, 2017, tradução nossa)

Wijaya, Rahmaddeni e Zoromi (2020) utilizam do DSSM para a criação de um *chatbot* com a função de um sistema de serviço para registro de novos estudantes. Eles explicam que, a rede seletiva possui duas "torres", a primeira sendo para o contexto, a segunda para as respostas e, ambas podem ter qualquer arquitetura desejada. A rede então, compara dois vetores (torres) utilizando a similaridade por cosseno, buscando o de maior similaridade.

No artigo, também há a explicação da utilização de uma *Confusion Matrix*, que segundo Narkhede (2018, tradução nossa) "é uma medida de desempenho para o problema de classificação de aprendizado de máquina em que a saída pode ser duas ou mais classes". Esta medida é usada para avaliar a acurácia, precisão e recuperação do sistema proposto.

4 *SEQUENCE TO SEQUENCE*

O modelo *Sequence to Sequence*, mais conhecido como seq2seq, segundo Manideep (2020, tradução nossa) "consiste em duas RNNs (*Recurrent Neural Networks*) - um codificador e um decodificador.". Alguns exemplos de RNNs são: LSTM (*Long Short Term Memory*) e GRU (*Gated Recurrent Units*), sendo que normalmente a LSTM é utilizada para o modelo.

Shah (2020, tradução nossa) informa que o modelo "prevê uma palavra dada na entrada do usuário e, em seguida, cada uma das próximas palavras é prevista usando a probabilidade de possibilidade dessa palavra ocorrer."

Sojasingarayar (2020, tradução nossa) utiliza o seq2seq em seu artigo, montando o com redes neurais LSTM, para a criação de um *chatbot* com aprendizagem de diálogos. Junto ao modelo, foi implementado um *attention mechanism* (mecanismo de atenção), que segundo o autor "permite que o decodificador olhe seletivamente para a sequência de entrada durante a decodificação. Isso tira a pressão do codificador para codificar todas as informações úteis da entrada.". Isso soluciona a limitação de que toda a informação na sentença de entrada deve ser codificada em um vetor de comprimento fixo.

Este modelo foi escolhido pois é frequentemente utilizado para o treinamento com diálogos, com objetivo em criar tradutores ou *chatterbots*, pois como descrito anteriormente, o modelo faz um processamento sequencial, o que é eficiente no processamento de textos e sentenças, além de suprir a necessidade do trabalho, sem requerer um código e operações muito complexas. (GANEGEDARA, 2020)

4.1 *Recurrent Neural Networks* e *Long Short Term Memory*

Para uma breve explicação do funcionamento de uma RNN (*Recurrent Neural Network*), Mittal (2019, tradução nossa) explica que "A RNN é recorrente por natureza, uma vez que executa a mesma função para cada entrada de dados, enquanto a saída da entrada atual depende do cálculo anterior.". Desta forma, elas se tornam recorrentes por calcular o próximo valor com o valor anterior, utilizando ambas entradas para o aprendizado.

A LSTM (*Long Short Term Memory*), como explicado pela Data Science Academy (2019, p. 51), "é uma arquitetura de rede neural recorrente (RNN) que 'lembra' valores em intervalos arbitrários.". Os autores descrevem que a arquitetura gera bons resultados em processos de classificação, processamento e previsão de séries temporais, onde os intervalos de tempo são desconhecidos. Possui diferentes blocos de memória, conhecidos como células, que armazenam informações, já as manipulações de dados nessas células são realizadas por componentes conhecidos como *gates*.

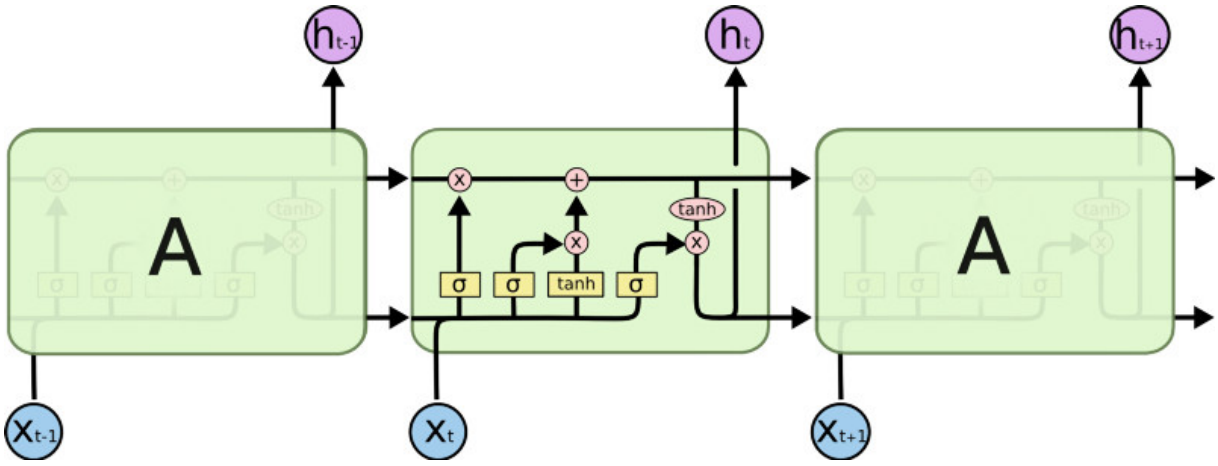
Phi (2018, tradução nossa) relata que "os *gates* são diferentes redes neurais que decidem quais informações são permitidas no estado da célula", e eles podem aprender quais são

informações importantes para decidir se devem guarda-las ou descarta-las.

4.1.1 Modelo

A Figura 3 representa um conjunto de módulos com quatro camadas de uma rede neural. Segundo Pereira (2017, p. 19) "a principal ideia das redes LSTM é criar uma representação do estado da célula, que corresponde a linha horizontal na parte superior do diagrama.".

Figura 3 – Modelo LSTM



Fonte: Ramamoorthy (2016)

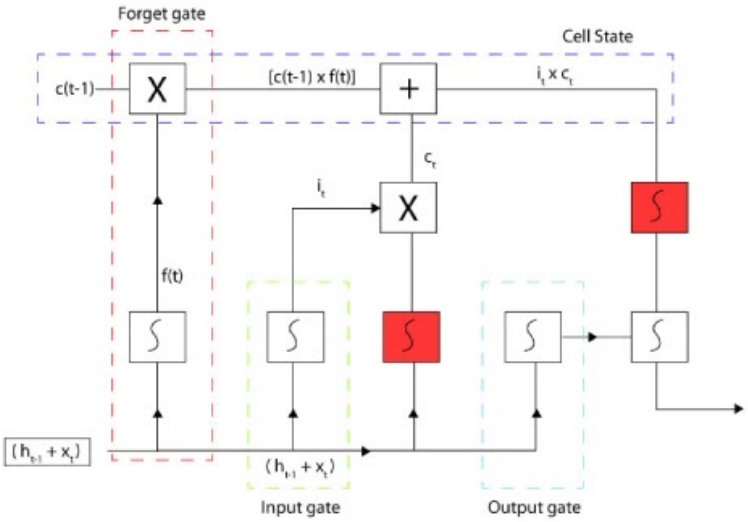
Na Figura 3, segundo Olah (2015), os retângulos amarelos representam as quatro camadas que irão influenciar no estado do fluxo de vetores. As linhas simbolizam uma transferência de vetores, partindo da saída de um componente e sendo direcionados para a entrada de outro. Os componentes rosas são operações realizadas nos vetores (adição, multiplicação, função de ativação Tanh). Linhas divididas significam que o conteúdo será copiado e enviado para diferentes destinos.

A função de ativação Tanh, segundo Sharma (2017), é $f(x) = tanh(x) = 2/(1 + e^{-2x}) - 1$, que basicamente utiliza da seguinte função sigmóide $A = 1/(1 + e^{-x})$, porém gerando intervalos entre -1 e 1.

4.1.2 *Gates*

Balodi (2019) aponta que a LSTM possui quatro *gates*: *forget gate*, *input gate*, *cell gate* e *output gate*. A Figura 4 apresenta visualmente a divisão dos *gates* dentro da arquitetura.

Figura 4 – Representação dos *gates* na LSTM



Fonte: Balodi (2019)

O *forget gate*, como Grandic (2019) explica, é onde toda a informação irrelevante de longo prazo é descartada, pois é utilizado um fator de esquecimento, multiplicando ele com as entradas, onde o resultado determinará se a informação deve ser armazenada ou esquecida.

Segundo Data Science Academy (2019), o *input gate* adiciona informações úteis ao estado da célula. Realiza uma função, semelhante a feita pelo *forget gate*, que filtra os valores a serem lembrados, porém não descarta nenhum valor. Então, um vetor com todos os possíveis valores é criado.

Phi (2018) descreve que o *cell gate*, inicialmente multiplica o estado da célula com o vetor criado pelo *forget gate*, possibilitando que alguns valores sejam descartados de acordo com o cálculo. Após, o vetor gerado pelo *input gate* é adicionado à célula, criando valores relevantes para a rede neural, gerando assim, um novo estado na célula.

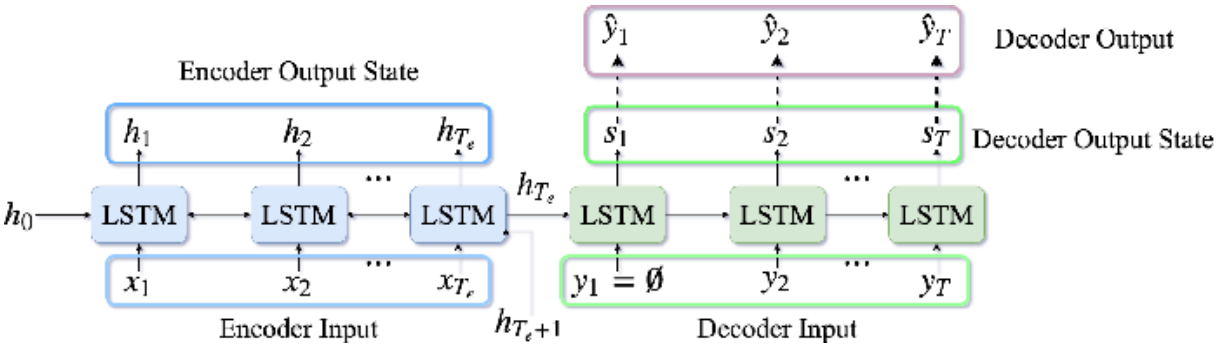
Por fim, como informa Balodi (2019), o *output gate* faz um processamento com a função de ativação sigmóide, comprimindo os dados para um intervalo de 0 a 1, multiplicando-os com os dados no estado atual da célula. Após, a informação da célula também é multiplicada por elemento, através da função de ativação Tanh. Por último, as informações são novamente multiplicadas, agora pelos dados de saída.

4.2 *Encoder* e *Decoder*

O *encoder* e *decoder* são o que formam o modelo seq2seq. Segundo Ramamoorthy (2016, tradução nossa), o *encoder* "lê a sequência de entrada, palavra por palavra, e emite um contexto, que idealmente capturaria a essência da sequência de entrada.". Já o *decoder* "gera a sequência de saída, uma palavra por vez enquanto olha para o contexto e a palavra anterior

durante cada intervalo de tempo".

Figura 5 – Modelo simples de *encoder* e *decoder*



Fonte: Keneshloo et al. (2019)

Para melhor explicação do processo, a Figura 5 representa visualmente o fluxo do modelo, onde a sentença é inserida no *encoder*, representado pelos componentes azuis. O *decoder*, representado pelos componentes verdes, irá receber o estado gerado pelo *encoder* e irá processa-lo em suas redes neurais, que por fim, irá gerar uma resposta para a sentença de entrada. Já o Quadro 5 apresenta um pseudocódigo de como o modelo é treinado e testado, percebendo assim, o processo executado e, consequentemente, colaborando com a explicação anterior. (KENESHLOO et al., 2019)

Quadro 6 – Pseudocódigo Seq2Seq

Algoritmo 1: Treinamento do modelo Seq2Seq

Entrada: Sequência de entrada (X) e sequência de saída (Y).
Saída: Modelo seq2seq treinado.

Treino:
for *lote de sequência de entrada e saída* X *e* Y **do**
    Executa a codificação em X e obtém o último estado do codificador h_{T_e}.
    Executa a decodificação alimentando h_{T_e} ao primeiro decodificador e obtém a sequência de saída amostrada $\hat{Y}$.
    Calcula a perda de acordo com uma equação e atualiza os parâmetros do modelo.
end
Teste:
for *lote de sequência de entrada e saída* X *e* Y **do**
    Usa o modelo treinado e uma equação para gerar a amostra da saída $\hat{Y}$.
    Avalia o modelo usando uma medida de performance.
end

Fonte: Keneshloo et al. (2019, tradução nossa)

4.3 Mecanismo de atenção

O modelo possui, em sua forma original, uma limitação, como GONÇALVES (2018 apud BAHDANAU; CHO; BENGIO, 2016) cita que "a rede neural precisa ser capaz de condensar toda a informação necessária da sentença de entrada em um vetor de dimensões fixas". Tornando inviável o processamento de sequências longas e não padronizadas para o conjunto de dados existente.

Para resolver este problema, Dugar (2019, tradução nossa) explica que "já que o problema era que um único vetor de estado oculto no final do codificador não era suficiente, enviamos tantos vetores de estado ocultos quanto o número de instâncias na sequência de entrada".

Sojasingarayar (2020) explica em seu artigo que, o mecanismo *attention* permite o *decoder* selecionar de forma otimizada a entrada durante seu processamento, tirando uma carga de trabalho grande do *encoder*, que precisava codificar todos os dados úteis até então. Para realizar esta seleção de forma otimizada, é criado um vetor de contexto, que armazena a soma ponderada dos estados ocultos do *encoder*.

Marathe (2020, tradução nossa) explica que o mecanismo de Bahdanau, utilizado neste trabalho, "também é conhecido como atenção aditiva, pois executa uma combinação linear dos estados do codificador e dos estados do decodificador.", onde o mecanismo faz com que todos os estados ocultos sejam enviados ao próximo passo, ao invés de enviar apenas um, assim, removendo a limitação citada previamente.

5 PLATAFORMAS E APLICATIVOS

Este Capítulo apresenta a plataforma de *help desk* JIRA e o aplicativo de mensagens Telegram, ambos utilizados para prover um fluxo, desde a entrada de informações, por parte de clientes, até o gerenciamento destas informações, na forma de chamados.

5.1 Plataformas *Help Desk*

Segundo Teles (2018), "um sistema de *help desk* é um software que centraliza os chamados da sua empresa em um único lugar, o que facilita o gerenciamento das solicitações.". Estes sistemas são amplamente utilizados por empresas de TI, pois além de facilitar o gerenciamento, também aceleram o processo de atendimento, solução e, dependendo do sistema, pode manter o cliente sempre informado das atualizações sobre uma solicitação existente.

5.1.1 JIRA

O JIRA *Service Desk* é o software da empresa Atlassian, que foi lançado em 2013, desenvolvido com a linguagem Java EE (Enterprise Edition). Como Oliveira (2016) aponta, o sistema possui "integração com CRM, abertura de chamado através de ligações, até diversas automações de triagem", demonstrando ser uma ferramenta completa para setores de suporte e desenvolvimento, além de ser conhecida e amplamente utilizada por empresas de TI. O software disponibiliza uma API REST para que outras ferramentas e sistemas possam fazer uma integração. A API proporciona opções para criar, editar e buscar chamados através destas chamadas. (ATLASSIAN, 2020)

As funcionalidades que se destacam na plataforma, são: agrupamento de clientes, pois permite que os clientes cadastrados sejam segmentados em grupos, facilitando o controle das informações das empresas cliente; personalização de email, que auxilia no gerenciamento das notificações por email, onde buscam deixar o cliente em conhecimento de cada passo realizado; permite que uma base de conhecimento seja construída, gerando mais informações disponíveis, de forma clara e rápida, aos clientes e colaboradores. (L3 Software, 2020)

A plataforma também disponibiliza duas formas de solução, uma *cloud* e outra instalada no servidor de preferência do usuário. A versão *cloud* trás benefícios como a facilidade de integração com outras plataformas, além de estar 99.9% disponível, sempre em constante atualização. Porém, quem prefere a versão *server* tem a vantagem de acessar informações diretamente na base de dados, além do espaço em disco ser apenas limitado ao *hardware* e também tem acesso a diversos *add-ons* e customizações. (OLIVEIRA, 2016)

5.2 Aplicativos de mensagens

As aplicações de mensagens instantâneas, os *chats*, vêm sendo usados mais a cada dia por empresas para contato com seus clientes. Segundo TI Inside (2015), uma "pesquisa realizada com dados do primeiro trimestre do ano [2015], mostra que a popularidade dos chats online com respostas instantâneas tem crescido de forma rápida, e cada vez mais consumidores alegam ser este o melhor canal para entrar em contato com a empresa.", sendo que, até o presente momento, a utilização destes canais aumentou consideravelmente.

5.2.1 Telegram

O Telegram é um aplicativo de mensagens *open source* criado em 2013, sendo desenvolvido com foco em proporcionar segurança dos dados dos usuários e ser rápido na troca de informações. O aplicativo vem sendo cada vez mais usado por ser semelhante ao Whatsapp, porém com mais liberdade para os desenvolvedores e sendo mais focado nos objetivos do usuário. (MELO, 2018)

Desde 2015, o Telegram permite desenvolvedores e usuários criarem *bots* na plataforma, que podem receber e responder mensagens e, até ser colocados em grupos. As mensagens e comandos enviados ao *bot* passam pelos servidores do Telegram, lidando com a criptografia e as comunicações com a API. A criação de um *bot* pode ser feita através do *BotFather*, após iniciar a conversa com ele, todas as informações e passos a serem seguidos serão explicados na conversa. (JúNIOR, 2020)

6 SISTEMA

Este Capítulo apresenta a modelagem e a forma que tecnologias citadas anteriormente são empregadas no funcionamento do sistema.

6.1 Motivação

O atendimento ao cliente, por parte de setores de suporte, é fundamental para o bom relacionamento e visibilidade desta empresa. O fator humano neste setor torna-o suscetível à falhas, como: demora no atendimento, desconhecimento de soluções para problemas recorrentes e falha na comunicação. Além de possuir demanda de tempo do setor para treinamento de novos colaboradores e documentação dos chamados.

As formas de comunicação, comercialmente, possibilitam diversos canais para que esta discussão seja feita e, por esse motivo, empresas não automatizam a entrada de chamados, gerando mais trabalho e, possivelmente, perda de informação.

Um sistema que, possa atender as necessidades dos clientes e solucionar dúvidas corriqueiras, utilizando o aprendizado de casos passados e uma base de dados, possibilita uma perda menor de recursos, além de agilizar o processo e auxiliar na entrada de chamados aos sistemas de gerenciamento de chamados. Uma ferramenta como esta, trará benefícios e solucionará casos como os apresentados.

Este trabalho constitui-se em um *chatbots* para atendimento de suporte, que utiliza técnicas de aprendizado sobre suas iterações para aprimoramento do atendimento ao cliente. Integrando com plataformas de comunicação conhecidas para acesso ao cliente e, também, à plataformas de gerenciamento de chamados para registro dos atendimentos.

6.2 Requisitos do sistema

Os requisitos do sistema definem quais funcionalidades devem estar disponíveis aos usuários, descrevendo as capacidades e limitações do aplicativo. A seguir estarão listados os requisitos funcionais e não funcionais do sistema desenvolvido.

6.2.1 Requisitos funcionais

Os requisitos funcionais são constituídos por dois itens principais, disponíveis para os clientes do sistema:

- **Criação de chamados**: permitir e auxiliar a criar chamados na plataforma de *help desk*, buscando prover uma forma intuitiva e mais humana por parte do *bot*.

- **Resolução de problemas previamente detectados**: auxiliar o usuário, informando-o de possíveis soluções sobre o problema relatado, de acordo com experiências e aprendizado passado.

O segundo item é aprimorado através da funcionalidade de conversa com um atendente disponível, pois o diálogo irá gerar mais dados para aprendizado do *bot*, provendo mais conhecimento para consultas futuras.

Outros requisitos funcionais adicionais relevantes, destinados os usuários que utilizarão do sistema para auxilio em seu trabalho, são:

- **Cadastro de senhas**: este cadastro se torna necessário pois, qualquer usuário poderá acessar a conversa com o *bot*, porém apenas usuários que possuam uma senha registrada poderão criar chamados e serão reconhecidos como clientes.
- **Cadastro de atendentes**: esta função estará disponível para administradores, para que possam gerenciar quais usuários podem prestar o atendimento à usuários do sistema e, também provendo acesso ao gerenciamento de senhas de clientes.

6.2.2 Requisitos não funcionais

Os requisitos não funcionais descrevem os items necessários para o funcionamento correto do sistema, desligando-se das funcionalidades providas aos usuários. O sistema atual possui os seguintes requisitos não funcionais:

- O sistema requer que o *Python 3.7* esteja instalado.
- O sistema necessita uma conexão à uma base de dados pré-configurada no banco *PostgreSQL*.
- Para a funcionalidade de aprendizado, é necessário a criação de um serviço agendado para iniciar o processo de treino.
- Um servidor do serviço JIRA deve estar configurado e deve possuir as credenciais suficientes para a conexão por parte do sistema. Também é necessário que um projeto *Service Desk* exista no servidor.
- Para integração com o Telegram é necessária a existência de um *bot* cadastrado.

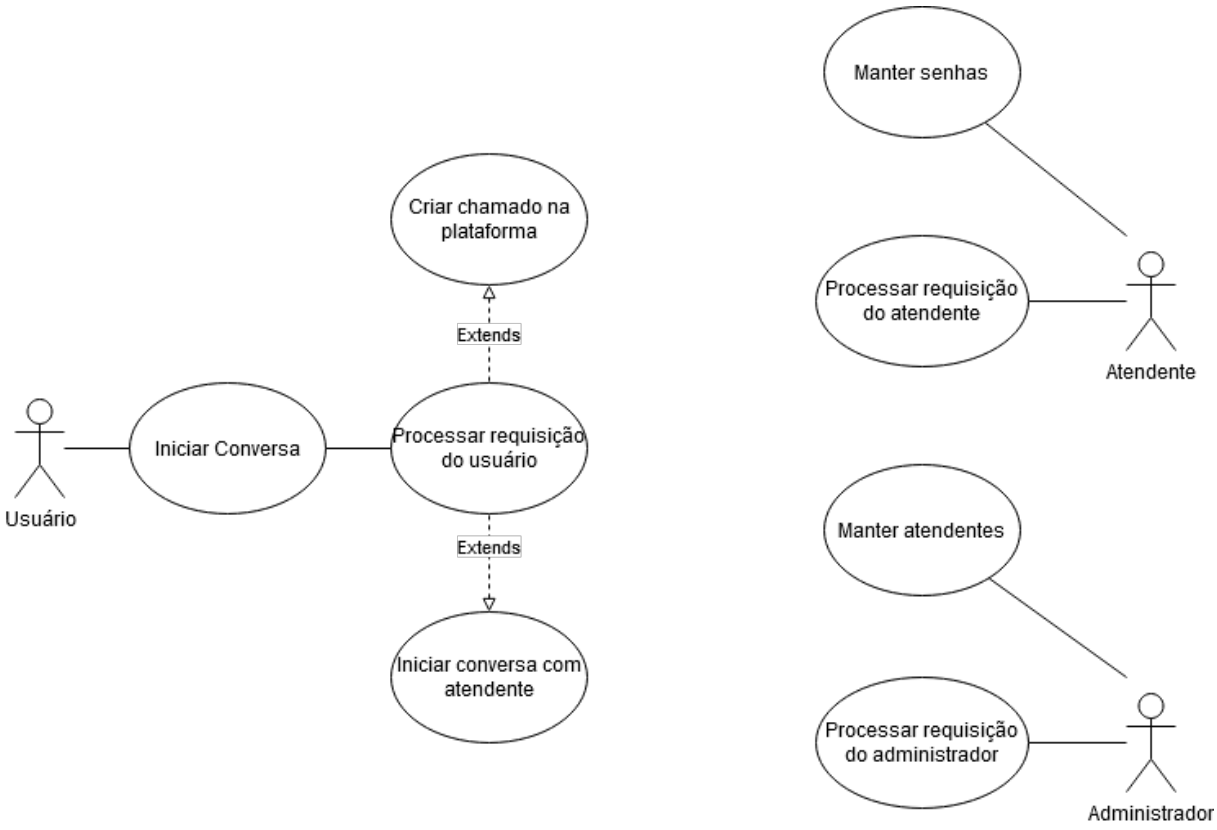
6.3 Modelagem

Esta seção de modelagem apresenta o funcionamento do sistema de forma abstrata e simplificada, contendo o caso de uso, o diagrama de classes, o diagrama de sequência e o diagrama de atividades.

6.3.1 Caso de Uso

A Figura 6 apresenta o caminho que o usuário seguirá durante a execução do sistema. O usuário inicia a conversa com o *chatbot*, onde cada mensagem dispara o evento do *bot* para interpretar a mensagem e responde-la. O sistema continua processando indeterminadamente as mensagens, até que o usuário queira criar um chamado ou conclua a conversa. Ao iniciar uma criação de chamado, o *bot* irá solicitar as informações pertinentes ao caso.

Figura 6 – Caso de uso - Geral



Fonte: Autor

O atendente irá executar comandos que serão processados, possibilitando que inicie uma conversa com um usuário e, que possa gerenciar senhas para clientes, que serão utilizadas para a autorização na criação de chamados. O administrador terá permissão de executar comandos para manter os atendentes registrados, adicionando e removendo existentes.

6.3.2 Descrição de caso de uso

A descrição de caso de uso, representada no Quadro 6, descreve o processo de criação de chamado, que pode ser realizado por parte do usuário.

Quadro 7 – Descrição do caso de uso para criação de chamado

Caso de Uso:	Criação de chamado
Ator(es):	Cliente
Pré-condições:	Chave registrada para criação de chamado.
Pós-condições:	Chamado criado com sucesso.

	Ator		**Sistema**	
1	Iniciar conversa com *bot*.			
		2	*Bot* inicia um chat com usuário.	
3	Informar a intenção de criar um chamado.			
		4	Confirmar intenção de criação de chamado.	R1
5	Confirmar intenção.			A1
		6	Verificar chave.	E1
		7	Perguntar o título do chamado.	
8	Informar título do chamado.			A1
		9	Requisitar descrição acerca do chamado.	
10	Informar descrição da situação no chamado.			A1
		11	Criar chamado na plataforma.	
		12	Retornar informação sobre o chamado criado.	

R1	Verificação da existência de uma chave registrada e vinculada ao usuário.
A1	Informar uma intenção de cancelamento direcionará o bot a confirmar o cancelamento do chamado em processo de criação.
E1	Caso a chave for inválida, o processo de criação é cancelado.

Fonte: Autor

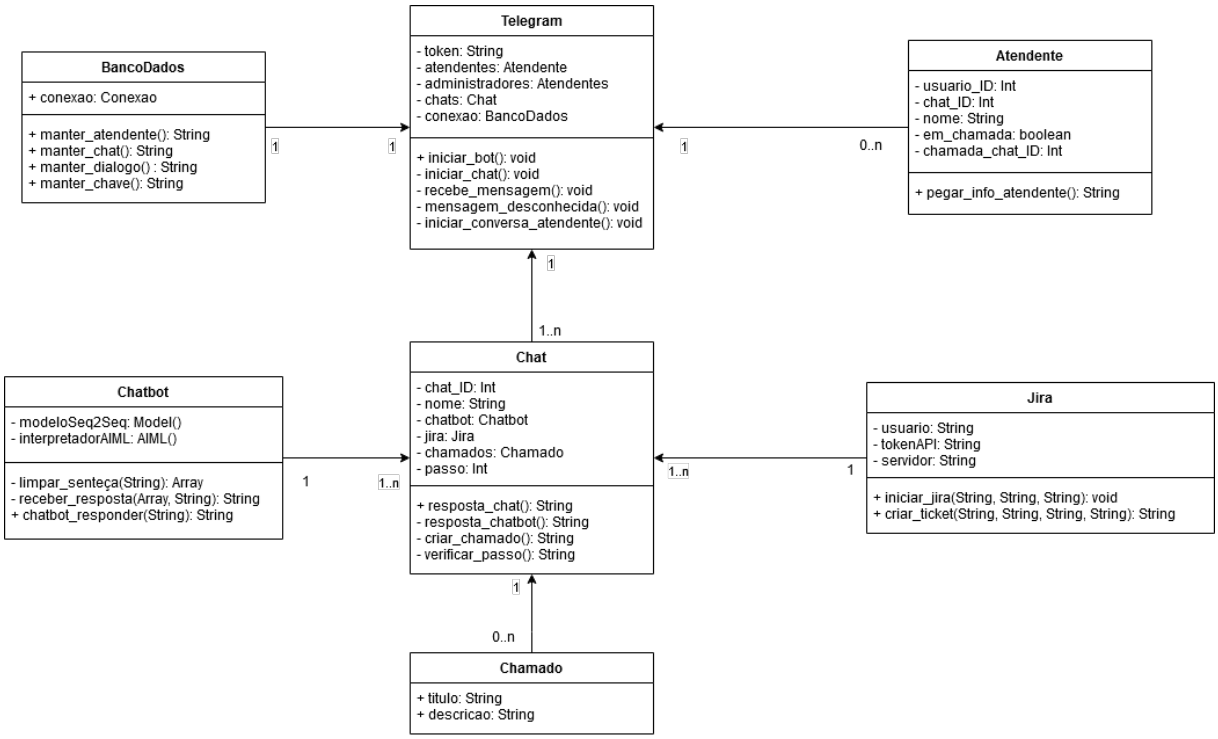
6.3.3 Diagrama de classes

A Figura 7 demonstra as classes utilizadas no sistema. A classe **Telegram** possui as informações para integração com a API da plataforma de mensagens. A classe **Atendente** armazena as informações dos atendentes registrados no sistema. A classe **Chat** faz o processamento das mensagens, retornando a resposta à API. A classe **Chatbot** possui os atributos para a execução dos algorítmos de inteligência artificial. A classe **Jira** possui atributos para conexão com a API da plataforma de *service desk*, utilizados no método *criar_ticket*, que criará um *ticket* na plataforma integrada. A classe **Chamado** armazena as informações do chamado a ser criado por

um cliente. A classe **BancoDados** faz a conexão com o banco de dados PostgreSQL, presente na Subseção 6.4.4 e provê os métodos para executar comandos no banco.

Este diagrama representa as classes de forma abstrata, não seguindo fielmente o nome dos atributos e métodos, com intuito de facilitar a compreensão da estrutura utilizada pelo sistema para sua execução.

Figura 7 – Diagrama de classes do sistema

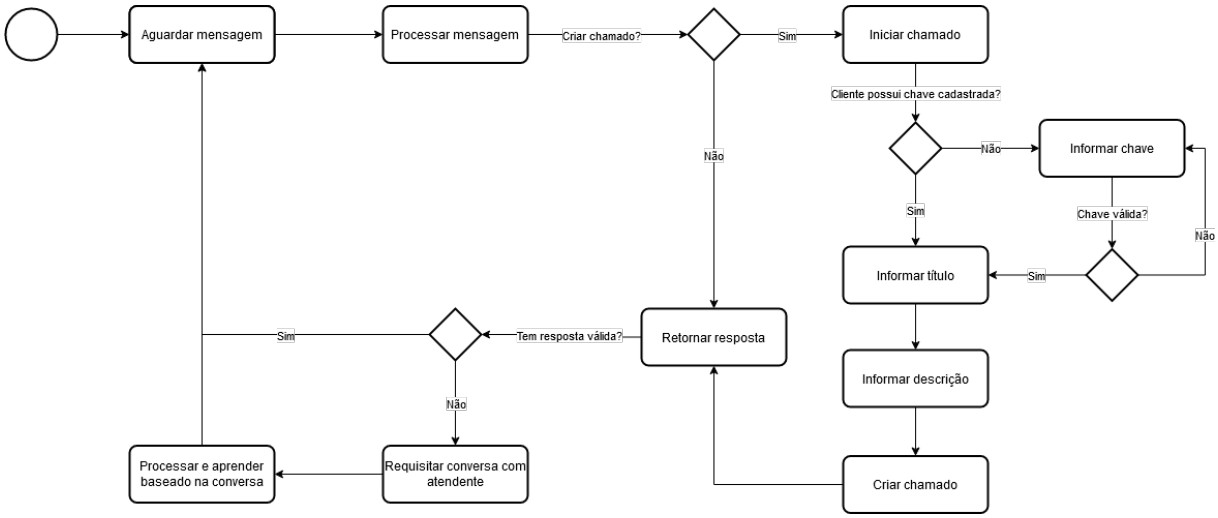


Fonte: Autor

6.3.4 Diagrama de atividade

A Figura 8, apresenta o principal fluxo do sistema, demostrando os caminhos disponíveis para o usuário interagindo com o *bot*. Como apresentado na figura, o *chatbot* irá verificar se possui uma resposta válida e, caso não possua, dará a possibilidade de conversa com um atendente disponível. Caso ocorra uma conversa com algum atendente, o sistema irá registrar a conversa para aprendizado. Após o inicio da conversa, o sistema não tem um ponto final, pois sempre terá o evento de recebimento de mensagem.

Figura 8 – Diagrama de atividade do sistema

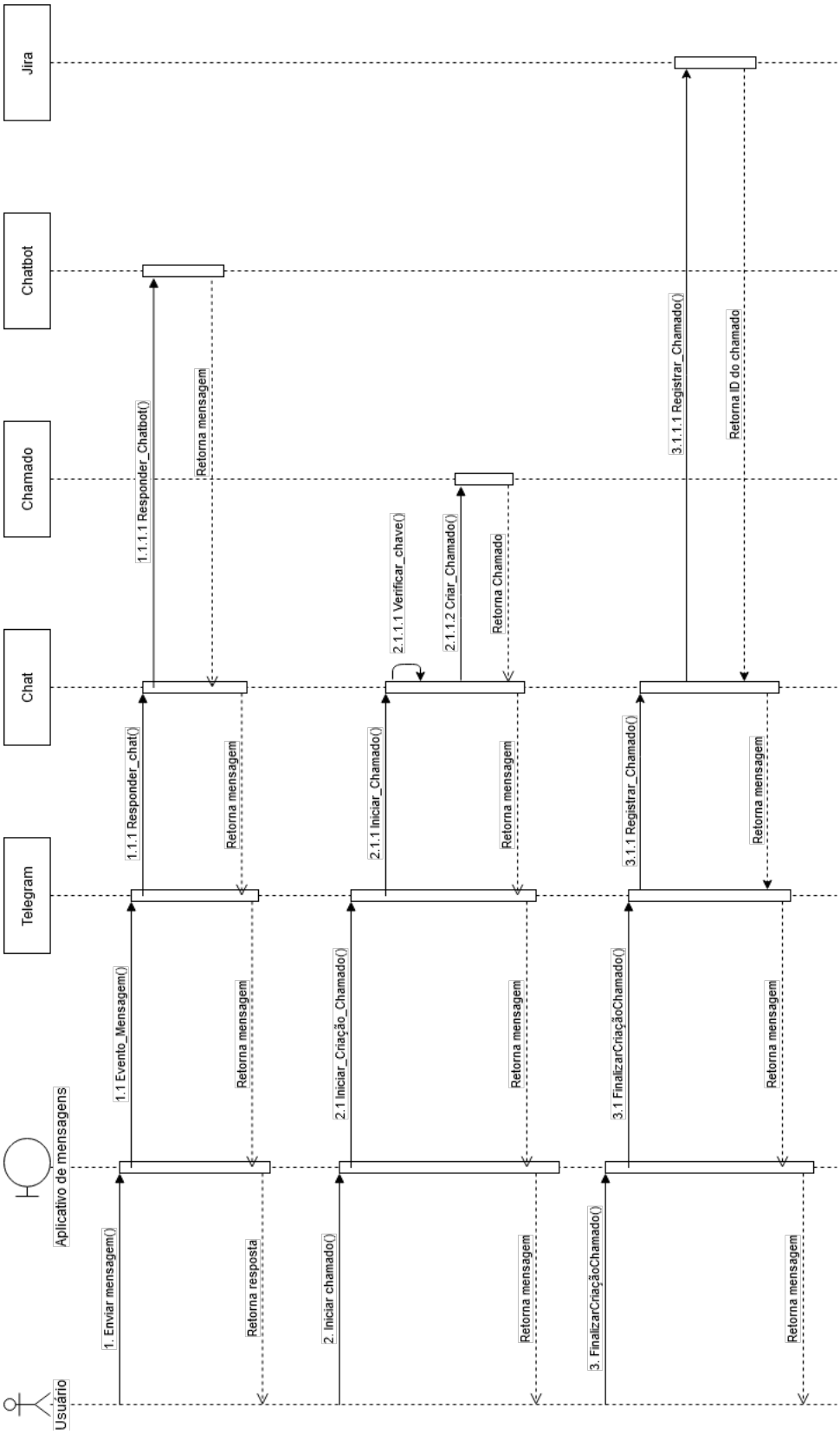


Fonte: Autor

6.3.5 Diagrama de sequência

A Figura 9 apresenta a sequencia de métodos de um fluxo do sistema, onde o usuário, após interação com o *bot*, deseja criar um chamado sobre o assunto discutido. Os objetos **Chat**, **Telegram** e **Chatbot** são utilizados para processar a conversa, sendo o objeto *chat* responsável pelo fluxo e controle principal dos dados. O objeto **Jira** é utilizado apenas quando há alguma chamada para a criação de chamado na plataforma de *service desk*. O objeto **Chamado** possui os campos para armazenar as informações necessárias para a criação de um chamado.

Figura 9 – Diagrama de sequência do sistema

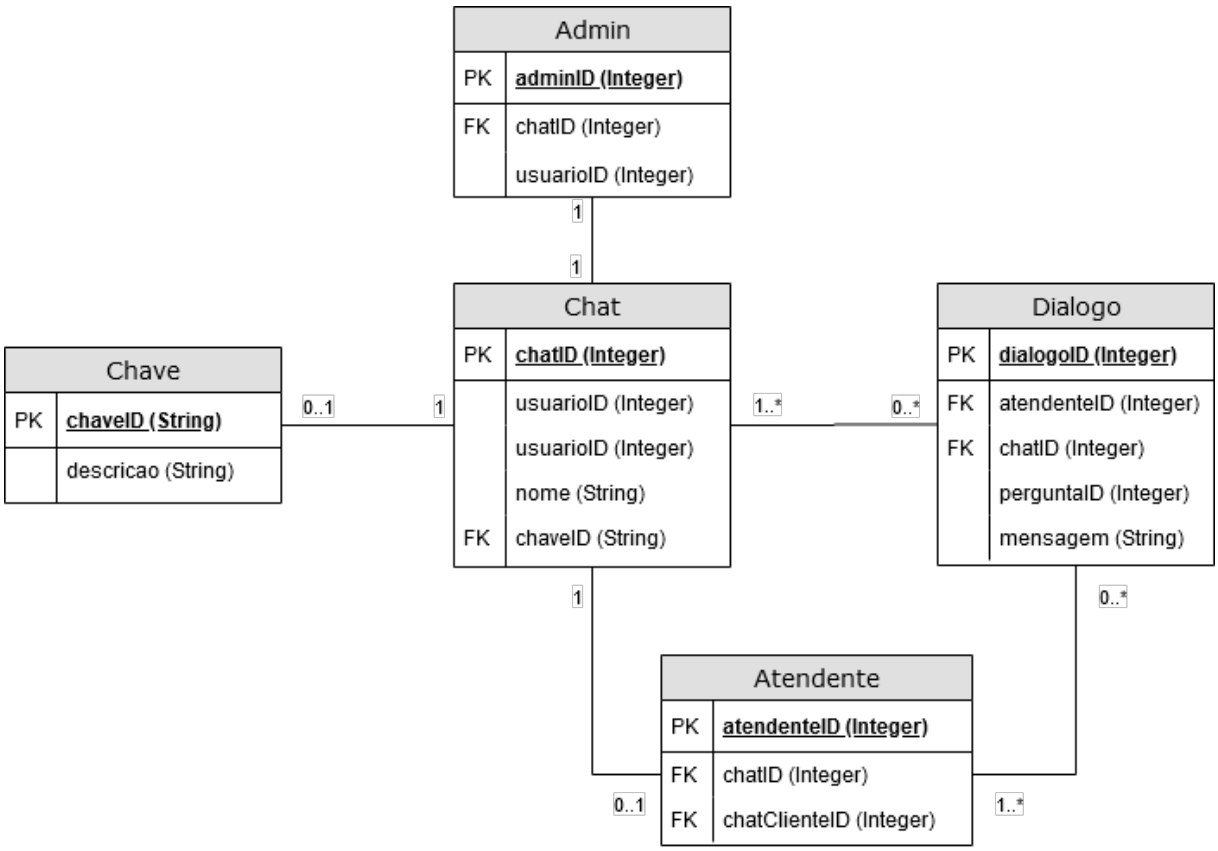


Fonte: Autor

6.3.6 Diagrama Entidade-Relacionamento

Na Figura 10, que representa o diagrama de entidades do banco de dados e o relacionamento entre elas por chaves estrangeiras. A tabela **Chat** armazena cada dialogo que o *bot* processou, onde o valor dos campos *chatID* e *usuarioID* são IDs do *chat*, sendo providos pela plataforma do Telegram. As tabelas **Atendente** e **Admin** armazenam os atendentes e administradores cadastrados no sistema, respectivamente. A tabela **Chave** guarda as chaves cadastradas por atendentes e, que são para habilitar a criação de chamados para clientes registrados. Por fim, a tabela **Dialogo** irá armazenar a conversa entre o atendente e um cliente (tabela **Chat**), para que o agente de aprendizado tenha uma base de conhecimento para treino.

Figura 10 – Diagrama Entidade-Relacionamento do sistema



Fonte: Autor

6.4 Ferramentas

Para o desenvolvimento do trabalho, foi necessária a utilização de algumas ferramentas. Esta seção pretende apresentar as ferramentas, descrevendo brevemente suas características.

6.4.1 *Visual Studio Code*

O *Visual Studio Code*, segundo Lardinois (2015) é um editor de código multi-plataforma leve que possibilita o desenvolvimento de aplicações *web*, programas em *cloud* e aplicativos. O editor aceita uma variedade de linguagens de programação, como *Python* e LaTeX, além de ter uma interface intuitiva e fácil para a instalação e uso destas linguagens.

6.4.2 LaTeX

O **LaTeX**, como descrito no site do The Latex Project (2020, tradução nossa), "é um sistema de preparação de documentos para composição tipográfica de alta qualidade.". Pois o sistema permite a configuração do *layout* do texto, facilitando a padronização do documento e provendo comandos para auxílio na escrita. Os comandos provêm uma forma rápida de inserir e criar estruturas, como figuras, tabelas e trechos de código.

6.4.3 *Python*

A linguagem de programação *Python*, de acordo com a Python Software Foundation (2020b) é uma linguagem interpretada, interativa e orientada à objetos. É considerada de alto nível, pois possui uma grande abstração das técnicas empregadas para a execução de aplicativos em máquinas. Segundo Data Science Academy (2020), é a principal linguagem utilizada para *machine learning* por três possíveis razões: sua simplicidade de escrita e facilidade de aprendizado, a grande quantidade de bibliotecas focadas em IA e, o *Jupyter Notebook*, uma interface *web* que permite a prototipagem rápida e a depuração de código.

6.4.4 PostgreSQL

O PostgreSQL é um banco de dados objeto-relacional de código aberto que utiliza da linguagem SQL e PL/SQL para o desenvolvimento de estruturas e funções. (The PostgreSQL Global Development Group, 2020) É um banco robusto, que pode ser executado em múltiplas plataformas, além de ser bastante utilizado no ambiente *web*. (4Linux, 2017)

6.4.5 *Git*

O *Git* é um sistema de controle de versões, que auxilia no versionamento de códigos-fonte, provendo ferramentas e recursos para criação de versões e fácil manutenção do código versionado. Junto ao sistema, foi utilizado o ***GitHub***, uma plataforma *web* para armazenamento do repositório gerenciado pelo *Git*, proporcionando também a contribuição de diversos programadores na construção e manutenção de códigos-fonte abertos à comunidade. (ESTRELLA, 2019)

6.4.6 *Draw.io*

A plataforma *Draw.io* proporciona um editor gráfico online para desenhos, planilhas e diagramas UML. Esta ferramenta foi utilizada para a criação dos diagramas presentes na Seção 6.3. (FURTADO, 2013)

6.5 Bibliotecas

A utilização de algumas bibliotecas da linguagem *Python* foi necessária, pois estas facilitam o desenvolvimento da aplicação, principalmente para a integração com as plataformas e para a utilização das técnicas de inteligência artificial. Esta seção descreverá de forma rápida as bibliotecas usadas e o que proporcionam.

6.5.1 *Program-Y*

Program-Y é um *framework* para *Python* 3, que processa AIML 2.1, provendo uma ferramenta para criação de *chatbots*. Ele segue a metodologia descrita na Seção 2.2, utilizando o algorítmo *Graphmaster* para a execução da busca de padrões nos arquivos '.aiml'. Também proporciona diversas configurações para a utilização da NLP, deixando o *chatbot* mais eficiente e preciso ao processar sentenças. (STERLING, 2020)

6.5.2 *Tensorflow*

A biblioteca *Tensorflow* oferece diversos métodos e algoritmos para aprendizado de máquina, computação numérica e outras áreas da inteligência artificial. Desenvolvida pela *Google*, se tornou a principal biblioteca para a criação de aplicativos nesta área de IA. Ela possui o modelo *Sequence to Sequence*, descrito no Capítulo 4, portanto, torna-se necessária para a implementação de um agente. (TensorFlow, 2020)

Como a Didática Tech (2020) explica em seu artigo, "primeiro você cria um blueprint do código, ou seja, um esqueleto daquilo que será executado, e depois executa abrindo uma sessão. Na prática, estamos criando um grafo computacional", esclarecendo que deve-se então, montar a estrutura do código, definindo, por exemplo, as camadas de uma rede neural e, por fim, executando a sessão que possui esta estrutura.

6.5.3 *NumPy*

A *NumPy* é uma biblioteca principalmente utilizada para efetuar cálculos em matrizes, pois fornece uma grande quantidade de funções e operações matemáticas. Ela se torna indispensável ao utilizarmos a biblioteca *Tensorflow*, pois constantemente são necessários cálculos para o processamento e aprendizado de agentes. (JR., 2018)

Neto (2020) descreve, sobre a biblioteca *NumPy*, que "seu principal objeto é o vetor n-dimensional, ou *ndarray*. Um vetor n-dimensional também é conhecido pelo nome **tensor**.",

sendo este o tensor usado pela biblioteca citada anteriormente na Subseção 6.5.2. Este tensor obriga que todos os elementos sejam mesmo tipo e, pode tanto, automaticamente definir quantas dimensões terá, quanto permite o desenvolvedor informar esta quantidade de dimensões. Desta forma, dispõem de uma flexibilidade necessária para os diversos modelos utilizados em aprendizados de máquina e aprendizado profundo.

6.5.4 *Telegram Python Bot*

Telegram Python Bot é uma biblioteca que abstrai a integração com a API oficial disponibilizada pelo Telegram, descrita na Subseção 5.2.1. Um objeto provido pela biblioteca gerencia o *bot* criado. Cada chat é processado por uma *thread* separada, permitindo a interação de diversos usuários simultaneamente, além de permitir a armazenagem de dados, em uma sessão, auxiliando no processamento dos dados por parte do *chatbot*. Requer apenas um token gerado pela API para a integração. (Python-Telegram-Bot, 2020)

6.5.5 *Python JIRA*

A biblioteca *Python JIRA* prove objetos para a utilização da API REST da plataforma Jira, apresentada na Subseção 5.1.1. Provê um acesso rápido a plataforma, requerendo apenas algumas informações como: usuário, endereço do servidor, e *token* para integração com a API. (Python Jira, 2020)

6.5.6 *ConfigParser*

A *ConfigParser* dispõem de objetos para a leitura de arquivos de extensão '.ini', normalmente utilizados em sistemas *Microsoft Windows*, para armazenar parâmetros utilizados em uma aplicação. (Python Software Foundation, 2020a)

6.5.7 *Psycopg*

A conexão com o banco *PostgreSQL* foi realizada através da biblioteca *Psycopg*, que é bastante utilizada por desenvolvedores. Implementa completamente as especificações da *Python DB API 2.0*, além de prover a utilização segura das *threads*, pois foi construída visando a utilização de múltiplas *threads* com concorrência entre as ações que influenciam a estrutura e registros do banco de dados. (GREGORIO, 2020)

6.6 Descrição do sistema

Esta seção descreverá o funcionamento das partes integrantes do sistema, sua configuração e processos internos. As reticências presentes nos quadros, representam trechos não expostos do código, pois não são relevantes à questão abordada.

6.6.1 Configurações

A configuração para funcionamento do sistema requer passos préviamente realizados, como:

- Para integração à plataforma *Telegram*, é necessária a criação de um *bot* e a obtenção de seu *Token*.
- Para integração à plataforma *Jira*, é preciso um usuário cadastrado na plataforma, a obtenção de uma chave da API e o endereço do servidor.
- Para conectar ao banco, necessita-se de uma conexão válida, com as credenciais necessárias para ter sucesso.

As informações descritas anteriormente, após obtidas, devem estar presentes no arquivo 'config.ini', de acordo com o Quadro 7.

Quadro 8 – Conteúdo do arquivo config.ini

```
1  [Jira]
2  usuario= <usuário cadastrado na plataforma JIRA>
3  token_api= <chave da API na plataforma>
4  server= <endereço do servidor que possui o serviço da plataforma> (Ex:
       https://helpdesk-chatbot.atlassian.net)
5  project_key= <chave do projeto de Service Desk>
6  tipo_chamado= <tipo de chamado a ser criado pelo bot>
7
8  [Telegram]
9  token= <token do bot para conexão com a API>
10
11 [Database]
12 host=<ip da base de dados>
13 name=<nome da base de dados>
14 user=<usuário no banco>
15 password=<senha do usuário>
```

Fonte: Autor

6.6.2 Integração com Telegram

Como explicado na Seção 6.5, a integração é realizara através da biblioteca *Telegram Python Bot*, provendo objetos para o gerenciamento das conversas.

O Quadro 8 apresenta o escopo da classe Telegram, deixando visível os métodos da biblioteca utilizados.

Quadro 9 – Integração com Telegram

```
1   class Telegram:
2       ...
3       def __init__(self, config):
4           self.__updater = Updater(token=config['Telegram']['token'],
                use_context=True)
5       ...
6       def __start(self, update, context):
7           update.message.reply_text('Olá ' + update.message.from_user.
                first_name + '!')
8
9       def __msg(self, update, context):
10          chat = self.__get_chat_session(update.message.from_user.id)
11          reply = chat.chat_reply(update.message.text)
12          ...
13          update.message.reply_text(reply)
14
15      def __unknown(self, update, context):
16          context.bot.send_message(chat_id=update.effective_chat.id,
                text='Desculpe, não consegui entender.')
17
18      def start_bot(self):
19          start_handler = CommandHandler('start', self.__start)
20          ...
21          msg_handler = MessageHandler(Filters.text & (~Filters.command),
                self.__msg)
22          unknown_handler = MessageHandler(Filters.command, self.
                __unknown)
23
24          dispatcher.add_handler(start_handler)
25          ...
26          dispatcher.add_handler(msg_handler)
27          dispatcher.add_handler(unknown_handler)
28
29          self.__updater.start_polling()
30          self.__updater.idle()
```

Fonte: Autor

O construtor da classe *__init__* (linha 3) instancia o objeto *updater*, que recebe o *token* da API para conexão. Os três métodos declarados (*start*, *msg*, *unknown*) são vinculados ao objeto durante a execução do método *start_bot* (linha 18). O método *start* (linha 6) é executado quando o usuário envia o comando *'/start'* ao *bot* do Telegram. O método *msg* (linha 9) recebe cada sentença enviada ao *bot*, onde chama o método *chat_reply* da classe *Chat*, processando a

mensagem e respondendo-a. Por fim, o método *unknown* (linha 15) trata-se de uma precaução, pois é disparado caso um comando inexistente seja enviado ao *bot*.

6.6.3 Integração com Jira

A integração com a plataforma de *service desk* Jira é simples, pois a biblioteca abstrai grande parte do trabalho. Como visível no Quadro 9, deve-se apenas instanciar um objeto da classe *Jira* e, chamar o método *create_issue* passando um dicionario com as informações requeridas pela API.

Quadro 10 – Integração com Jira

```
1     key = ''
2     jira = None
3     def start_jira(user, token_API, server, project_key):
4         global jira, key
5         jira = JIRA(basic_auth=(user, token_API), options={'server':
              server})
6         key = project_key
7
8     def create_issue(issueType, ticket):
9         global jira, key
10        fields={
11        'project': {'key': key},
12        'issuetype': {"name": issueType},
13        'summary': ticket.summary,
14        'description': ticket.description}
15        issue = jira.create_issue(fields)
16        return issue.key
```

Fonte: Autor

6.6.4 Processamento da Mensagem

O processamento da mensagem enviada pelo usuário poderá ser direcionado para três fluxos, dependendo de como cada um destes fluxos consegue atender à mensagem.

Primeiramente, a mensagem é tratada pela classe *Chat*, representada no Quadro 10. O construtor da classe (linha 2) cria atributos para o ID do usuário, passo do processo de criação de chamado e instancia a classe *Chatbot*. O método *chat_reply* (linha 25) irá receber a mensagem e encaminhará ao processador de AIML, caso não exista uma resposta para a sentença, buscará no modelo treinado a partir das conversas anteriores e, se este não retornar uma resposta, o *bot* questionará o usuário a possibilidade de iniciar uma conversa com um atendente.

O método *check_message* (linha 9) verifica se a resposta corresponde ao contexto da criação de um chamado.

Quadro 11 – Processamento da mensagem

```
1     class Chat:
2         def __init__(self, id, config):
3             self.__user_id = id
4             ...
5             self.__issue_process = 0
6             ...
7             self.__chatbot = ChatBot()
8
9         def __check_message(self, msg, answer):
10            if answer == 'CRIARCHAMADO.':
11            ...
12            if answer == 'CANCELARCHAMADO.':
13                ...
14            if answer == 'RESPOSTASIM.':
15                ...
16            if answer == 'RESPOSTANAO.':
17                ...
18            if self.__issue_process == 2:
19                ...
20            elif self.__issue_process == 3:
21                ...
22                key = create_issue(self.__issue_type, self.__ticket)
23                return self.__chatbot.chatbot_response('CRIADOCHAMADO').
                      replace('#1', key)
24        ...
25        def chat_reply(self, msg):
26            reply = self.__chatbot.chatbot_response(msg)
27            reply = self.__check_message(msg, reply)
28            if reply:
29                reply = reply.encode("cp1252")
30                reply = reply.decode()
31            return reply
```

Fonte: Autor

6.6.5 Processador de AIML

A classe *Chatbot* possui o processador de AIML (*Program-Y*). O Quadro 11 apresenta um trecho do código da classe, onde no construtor é instanciado o processador, que requer diversos arquivos de configuração e regras para executar. O método *chatbot_response* (linha 23)

primeiramente trata a sentença, removendo caracteres especiais e, após, envia ao processador para que este execute e retorne uma resposta.

Quadro 12 – Processador de AIML

```
1     class ChatBot:
2         def __init__(self):
3             arquivos = {'aiml': ['chat/dados/aiml'],
4                         'learnf': ['chat/dados/learnf'],
5                         'properties': 'chat/dados/properties/properties.txt'
                              ,
6                         'defaults': 'chat/dados/properties/defaults.txt',
7                         'sets': ['chat/dados/sets'],
8                         'maps': ['chat/dados/maps'],
9                         'rdfs': ['chat/dados/rdfs'],
10                        'denormals': 'chat/dados/lookups/denormal.txt',
11                        'normals': 'chat/dados/lookups/normal.txt',
12                        'genders': 'chat/dados/lookups/gender.txt',
13                        'persons': 'chat/dados/lookups/person.txt',
14                        'person2s': 'chat/dados/lookups/person2.txt',
15                        'regexes': 'chat/dados/regex/regex-templates.txt',
16                        'spellings': 'chat/dados/spelling/spelling.txt',
17                        'preprocessors': 'chat/dados/processing/
                              preprocessors.conf',
18                        'postprocessors': 'chat/dados/processing/
                              postprocessors.conf',
19                        'postquestionprocessors': 'chat/dados/processing/
                              postquestionprocessors.conf'
20            }
21            self.__bot = EmbeddedDataFileBot(arquivos, logging_filename='
                  chat/config/logging.yaml')
22        ...
23        def chatbot_response(self, msg):
24            try:
25                msg = self.__remove_special_characters(msg)
26                reply = self.__bot.ask_question(msg)
27                if not reply:
28                    reply = ''
29                return reply
30            except Exception as e:
31                return ''
```

Fonte: Autor

6.6.6 Treinamento do modelo

O treinamento do modelo é feito com a biblioteca *TensorFlow*, que consiste na construção do modelo no código e seu uso para treinar os dados.

Quadro 13 – Construção do modelo Seq2Seq

```
1   def seq2seq_model(source_vocab_size,embed_size, rnn_size, keep_prob,
2       target_vocab_size, batch_size, vocabs_to_index):
3       input_data, target_data, input_data_len, target_data_len, lr_rate,
            keep_probs = model_input()
4
5       encoder_embedded = encoder_input(source_vocab_size, embed_size,
            input_data)
6       stacked_cells = lstm(rnn_size, keep_prob)
7       encoder_outputs, encoder_states = encoder_layer(stacked_cells,
            encoder_embedded, input_data_len)
8       dec_cell = lstm(rnn_size*2, keep_prob)
9       dense_layer = tf.layers.Dense(target_vocab_size)
10
11      attention_mechanism = tf.contrib.seq2seq.BahdanauAttention(
12          rnn_size*2,encoder_outputs, memory_sequence_length=
                target_data_len)
13
14      attention_cell = tf.contrib.seq2seq.AttentionWrapper(
15          dec_cell, attention_mechanism, attention_layer_size=rnn_size/2)
16      state = attention_cell.zero_state(dtype=tf.float32, batch_size=
            batch_size)
17      state = state.clone(cell_state=encoder_states)
18
19      outputs_train,outputs_infer = decoder_layer(rnn_size,
            encoder_outputs,
20          target_data_len, dec_cell, encoder_states, target_data,
21          vocabs_to_index, target_vocab_size, embed_size, dense_layer,
22          attention_cell, state, batch_size)
23
24      inference_logits,cost,train_op = opt_loss(outputs_train,
            outputs_infer,
25      target_data_len, target_data, lr_rate)
26
27      return input_data, target_data, input_data_len, target_data_len,
            lr_rate,keep_probs, inference_logits, cost, train_op
```

Fonte: Autor

O Quadro 12 apresenta a construção do modelo Seq2seq, criando-se duas LSTMs (linhas 6 e 8) e adicionando o mecanismo de atenção (linha 11), mencionado na Seção 4.3. Já no Quadro 13, pode-se ver que é gerada uma sessão para treinamento (linha 4), iterando as épocas definidas (linha 6) com os dados pré-processados para que possam ser computados pelo modelo.

Quadro 14 – Treinamento do modelo Seq2Seq

```
1     ...
2     input_data, target_data, input_data_len, target_data_len lr_rate,
         keep_probs, inference_logits, cost, train_op = seq2seq_model(
         question_vocab_size, EMBED_SIZE, RNN_SIZE, KEEP_PROB,
         answer_vocab_size, BATCH_SIZE, vocabs_to_index)
3     ...
4     with tf.Session() as sess:
5         sess.run(tf.global_variables_initializer())
6         for epoch in range(EPOCHS):
7             for bs in tqdm(range(0, round_no, BATCH_SIZE)):
8                 index = min(bs + BATCH_SIZE, round_no)
9
10                batch_x,len_x = pad_sentence(train_data[bs:index],pad_int)
11                batch_y,len_y = pad_sentence(test_data[bs:index],pad_int)
12                batch_x = np.array(batch_x)
13                batch_y = np.array(batch_y)
14                sess.run([inference_logits,cost,train_op],
15                         feed_dict={input_data:batch_x,
16                                    target_data:batch_y,
17                                    input_data_len:len_x,
18                                    target_data_len:len_y,
19                                    lr_rate:LEARNING_RATE,
20                                    keep_probs:KEEP_PROB}
21                        )
22
23            sess.run(inference_logits,
24                   {input_data: [translate_sentence]*BATCH_SIZE,
25                    input_data_len: [len(translate_sentence)]*BATCH_SIZE,
26                    target_data_len: [len(translate_sentence)]*BATCH_SIZE,
27                    keep_probs: KEEP_PROB}
28                   )[0]
29            saver = tf.train.Saver()
30            saver.save(sess,MODEL_DIR+"/"+SAVE_PATH)
```

Fonte: Autor

As variáveis em maiúsculo, presentes nos Quadros 13 e 14, são valores configuráveis para o treino:

- BATCH_SIZE = 80 <Tamanho do lote de dados para treino de uma época (em porcentagem)>
- RNN_SIZE = 512 <Número de camadas de uma LSTM>
- EMBED_SIZE = 512 <Tamanho do mapa de *embedding*>
- LEARNING_RATE = 0.001 <Taxa de aprendizado para cada época>
- KEEP_PROB = 0.75 <Determina a taxa de *dropout*, reduzindo a possibilidade de *overfitting*>
- EPOCHS = 100 <Número de épocas a serem realizadas>
- MODEL_DIR = '<diretório do modelo>'
- SAVE_PATH = '<diretório para salvar o checkpoint>'
- MIN_LEARNING_RATE = 0.0001 <Limite mínimo da taxa de aprendizado>
- LEARNING_RATE_DECAY = 0.9 <Taxa de perda do aprendizado>

As variáveis MIN_LEARNING_RATE, LEARNING_RATE_DECAY RNN_SIZE, EMBED_SIZE, LEARNING_RATE e KEEP_PROB são utilizadas pelo modelo do TensorFlow para aprendizado, a variável BATCH_SIZE é usada para definir a porcentagem dos dados que serão usados para treino. O número de épocas, ou seja, ciclos de treino, é definido pela variável EPOCHS. As variáveis MODEL_DIR e SAVE_PATH server para definir o caminho e nome dos arquivos gerados pelo modelo.

Os códigos fonte dos quadros 12, 13 e 14, junto com os valores das variáveis citadas, são baseados e derivados do artigo de Sojasingarayar (2020, p.14). A autora verificou uma acurácia de 63%, sendo a maior alcançada em testes com o modelo Seq2Seq, ao utilizar os valores em questão.

6.6.7 Processamento com Seq2Seq

O processamento com modelo, como visto no Quadro 14, é feito através de uma sessão interativa que carrega os arquivos do modelo, gerados préviamente pelo processo de treinamento. O método *get_response* (linha 18) processa a sentença e retorna uma resposta.

Quadro 15 – Processamento do modelo Seq2Seq

```
class Seq2Seq:
    def __init__(self, config):
        loaded_graph = tf.Graph()
        self.__sess = tfc.InteractiveSession(graph=loaded_graph)
        save_path = MODEL_DIR + '/' + SAVE_PATH
        loader = tf.train.import_meta_graph(save_path + '.meta')
        loader.restore(sess, save_path)
        self.__input_data = loaded_graph.get_tensor_by_name('input:0')
        self.__logits = loaded_graph.get_tensor_by_name(
            'predictions:0')
        self.__input_data_len = loaded_graph.get_tensor_by_name(
            'input_len:0')
        self.__target_data_len = loaded_graph.get_tensor_by_name(
            'target_len:0')
        self.__keep_prob = loaded_graph.get_tensor_by_name(
            'keep_prob:0')
    ...
    def get_response(self, text):
        model_input = sentence_to_seq(text, vocabs_to_index)
        output = make_pred(self.__sess, self.__input_data, self.
            __input_data_len, self.__target_data_len, self.__keep_prob,
             model_input, BATCH_SIZE, self.__logits, index_to_vocabs)
        return output
```

Fonte: Autor

7 DEMONSTRAÇÃO

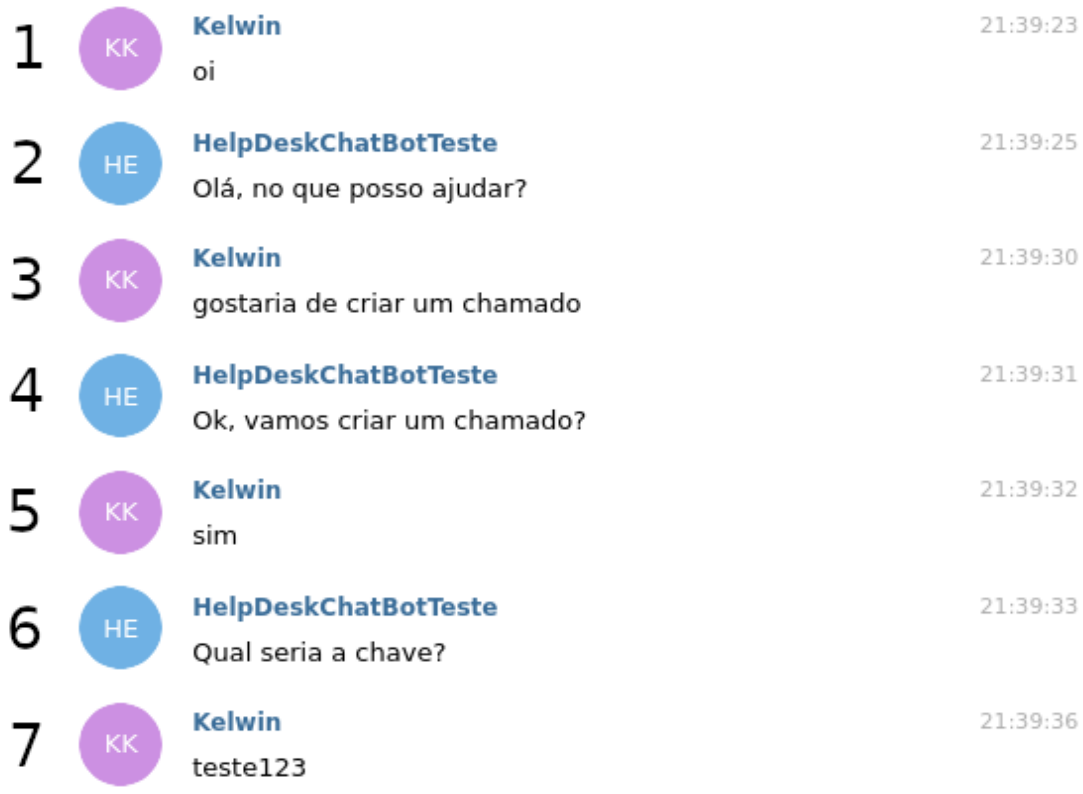
Neste Capítulo será apresentado o funcionamento do sistema, utilizando-se de figuras para melhor representação das funcionalidades descritas no Capítulo 6. Algumas das figuras que apresentam mensagens no aplicativo Telegram (Subseção 5.2.1), possuem um número ao lado de cada sentença para melhor visualização e compreensão da demonstração.

7.1 Criação de chamados

Este processo inicia-se quando o usuário informa a intenção de criar um chamado. A Figura 11 apresenta uma conversa realizada com o *chatbot*. Ao demonstrar a intenção (mensagem 3), o *bot* faz uma pergunta para confirmar se a intenção está correta (mensagem 4), e caso receba uma resposta positiva (mensagem 5), inicia o processo de criação de chamado.

O usuários necessitam de uma chave para ter permissão para criar chamados, na Figura 11 o *chatbot* requisita uma chave válida para dar continuidade ao processo (mensagem 6). Caso o operador possua uma chave gravada, esta etapa é pulada.

Figura 11 – Criação de chamado - Parte 1



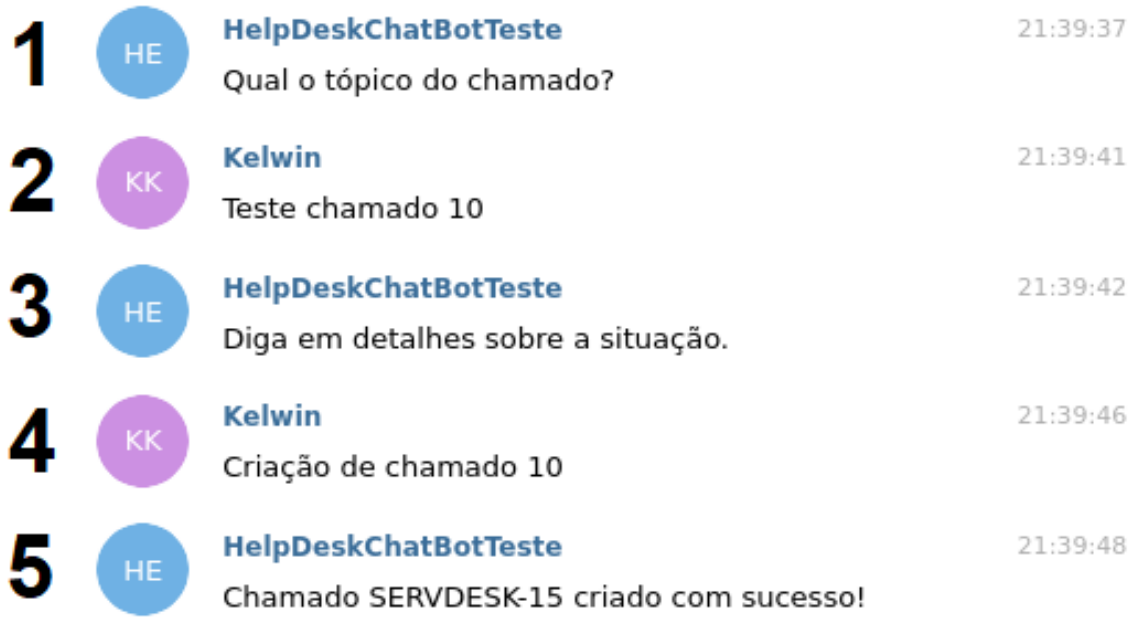
Fonte: Autor

A Figura 12 mostra a continuação do processo, onde o *chatbot* requisita ao usuário

sobre o título e descrição do chamado a ser criado (mensagens 1 e 3). Por fim, o *bot* retorna ao usuário qual é a chave, na plataforma de *service desk*, do chamado criado (mensagem 5).

Todo o processo realizado para criação de chamado passa pelo interpretador de AIML, explicado na Seção 2.2.

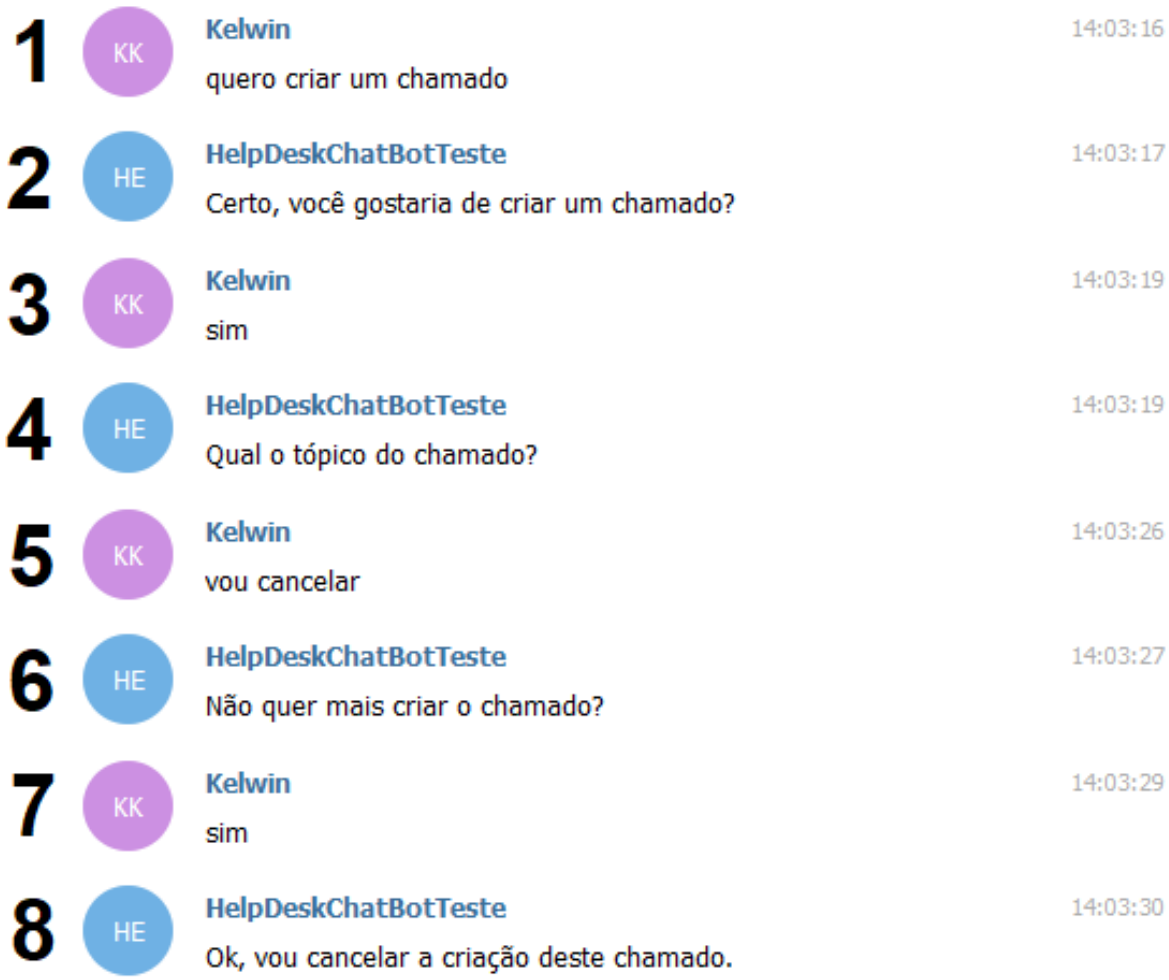
Figura 12 – Criação de chamado - Parte 2



Fonte: Autor

Caso o usuário queira interromper o processo em andamento, o mesmo deve informar a intenção de cancelamento, como visto na Figura 13, assim, o sistema irá requisitar uma confirmação sobre a intenção (mensagem 6), caso a intenção seja positiva, o processo é cancelado e, caso seja negativa, o *chatbot* volta a etapa que estava previamente.

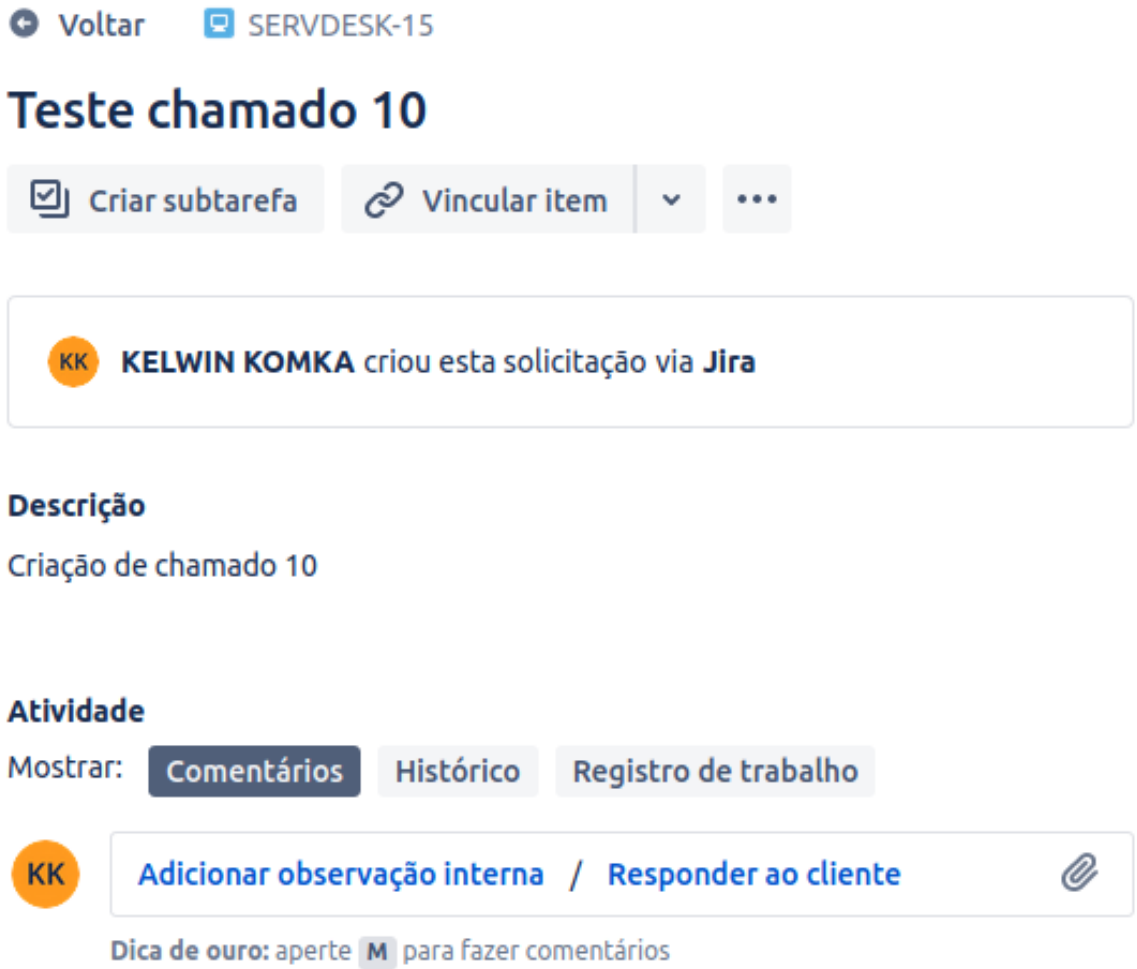
Figura 13 – Chamado cancelado



Fonte: Autor

A Figura 14 apresenta o chamado criado na plataforma JIRA (Subseção 5.1.1), a partir da demonstração apresentada anteriormente, onde pode-se ver que, o título e a descrição condizem com o exemplo das Figuras 11 e 12.

Figura 14 – Chamado criado na plataforma JIRA



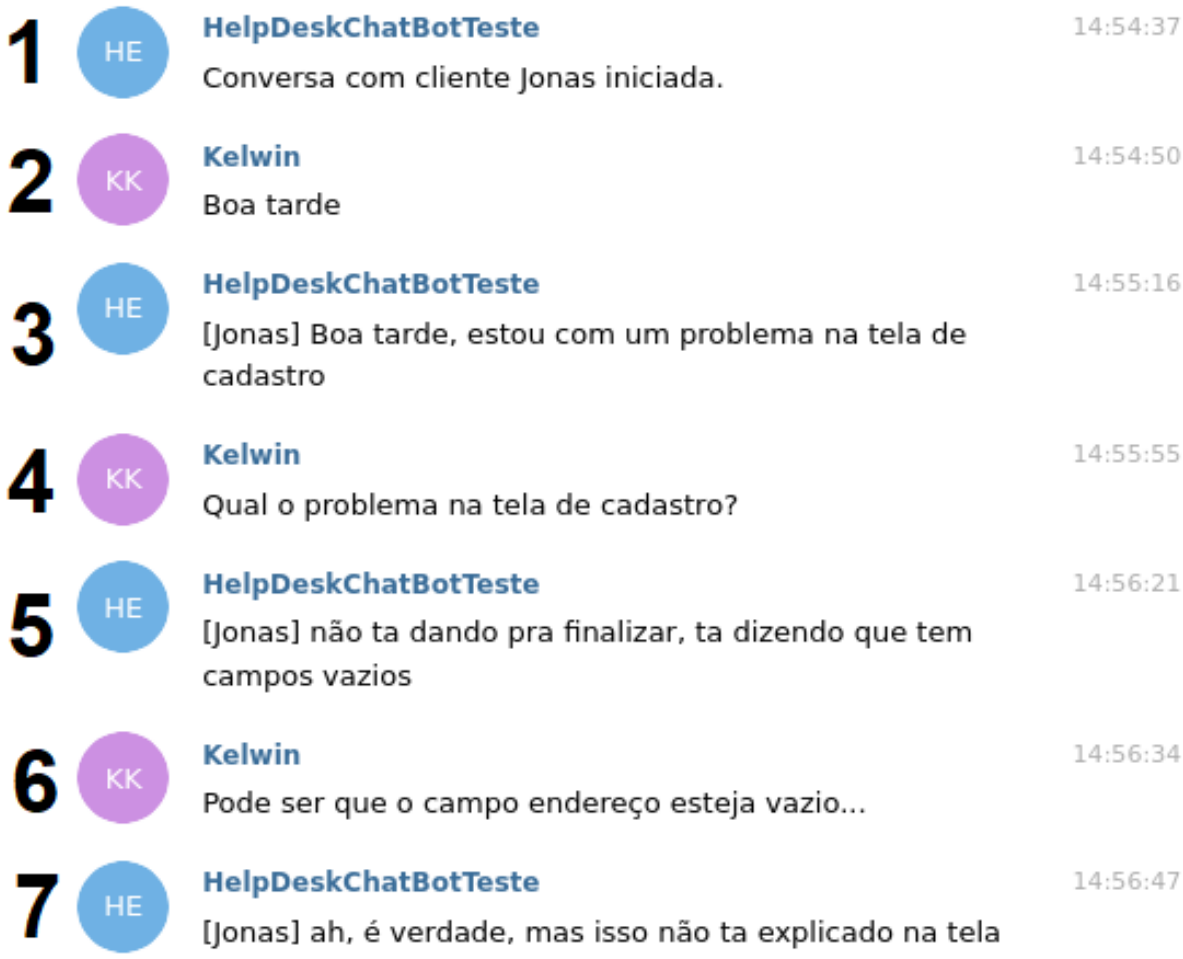
Fonte: Autor

7.2 Conversa com atendente

A Figura 15 mostra a conversa entre um cliente e um atendente através do *chatbot*. As mensagens do cliente tem como prefixo o nome do cliente dentro de colchetes, para melhor compreensão por parte do atendente (mensagens 3, 5 e 7). O inicio deste processo pode ser visto na Seção 7.3.

Durante este processo, tanto o *chat* do cliente, quanto o do atendente, não irão disparar eventos e processamento por parte do interpretador de AIML e o modelo Seq2Seq. O atendente, ao final da conversa, executa o comando */finalizarconversa* para encerrá-la, assim, as funcionalidades desabilitadas serão ativadas novamente.

Figura 15 – Conversa com cliente através do *bot*

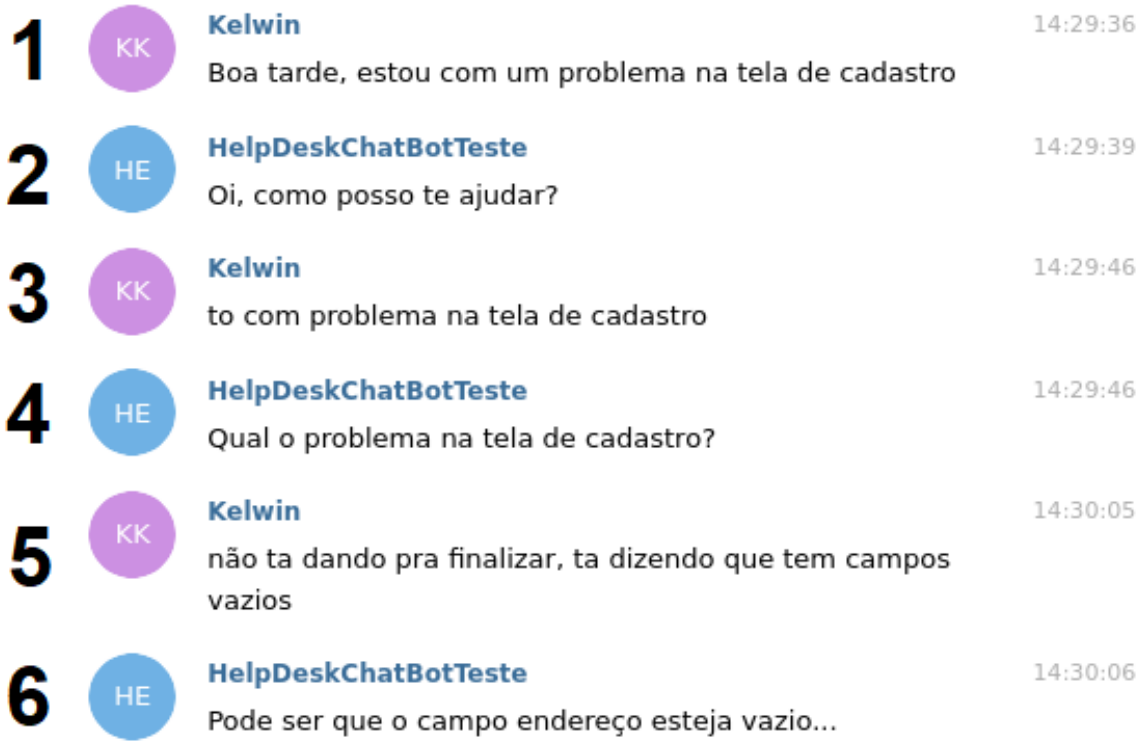


Fonte: Autor

7.3 Serviço de suporte

O serviço de suporte, como é chamada a parte onde o modelo Seq2Seq (Capítulo 4) faz seu processamento. A Figura 16 apresenta um diálogo entre o usuário e o *bot*, onde o *bot* segue a conversa a partir dos dados treinados, provendo soluções semelhantes às informadas anteriormente (mensagens 4 e 6), vistas na Seção 7.2.

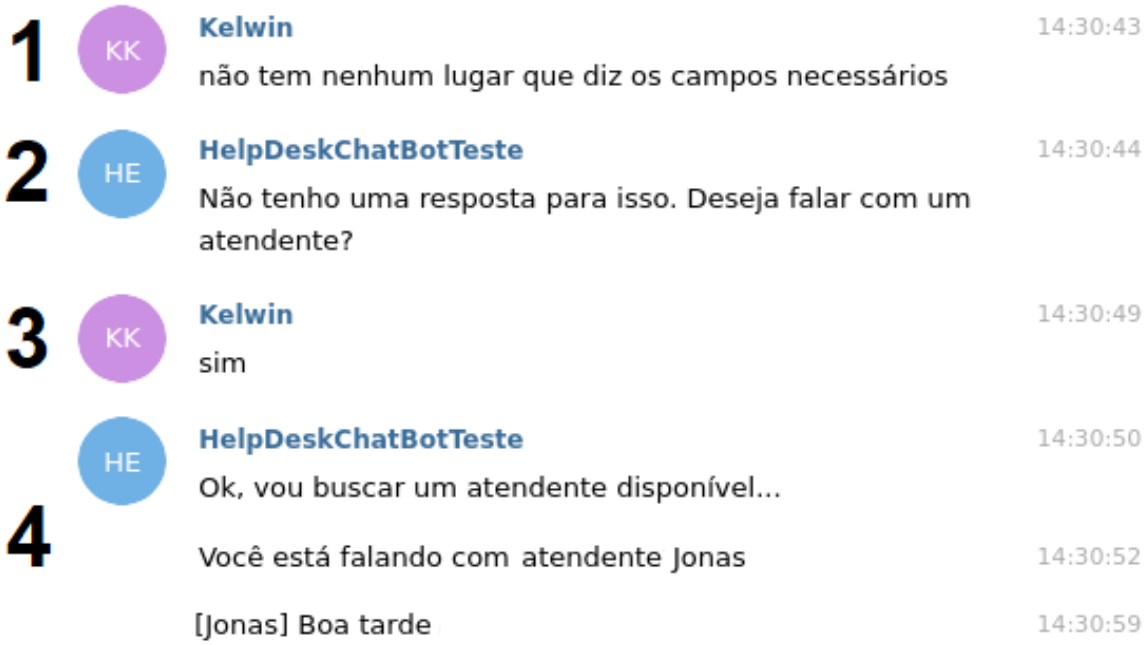
Figura 16 – Conversa com *bot* a partir do modelo treinado



Fonte: Autor

Na Figura 17, pode-se ver o momento em que o modelo Seq2Seq não possui uma resposta válida e precisa o suficiente (mensagem 2), portanto, o sistema possibilita ao usuário iniciar um diálogo com um atendente disponível. Caso o usuário dê uma intenção positiva, o sistema busca e inicia a conversa com um atendente (mensagem 4).

Figura 17 – Direcionamento para diálogo com atendente

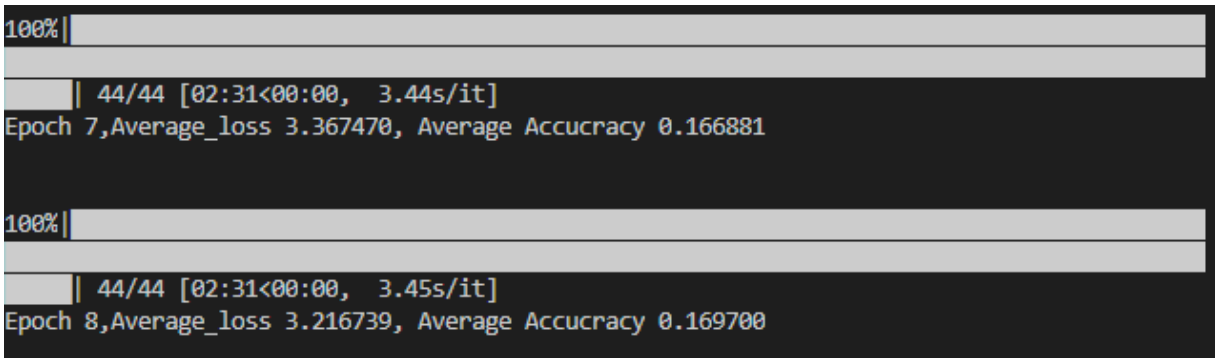


Fonte: Autor

7.4 Treinamento dos dados

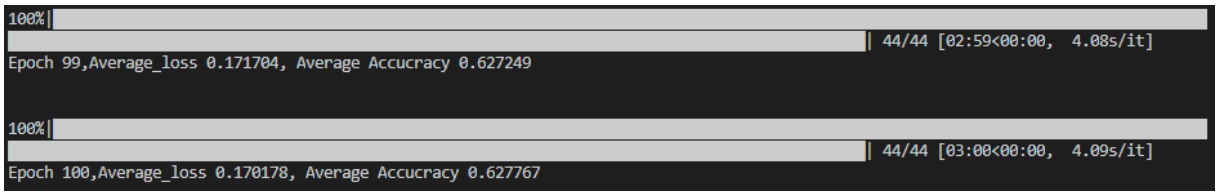
O treinamento dos dados, como explicado na Subseção 6.6.6, é feito de forma separada, pois, dependendo da quantidade de dados e poder de processamento da máquina, o tempo decorrido pode ser longo e, isso poderia afetar o processamento principal da ferramenta. A Figura 18 apresenta o inicio do treino, onde a média deperda ainda é relevante e a média de precisão está baixa.

Figura 18 – Inicio do treino dos dados



Fonte: Autor

Figura 19 – Fim do treino dos dados



Fonte: Autor

Na Figura 19 é possível ver que, após as 100 épocas, as médias de perda e precisão estão melhores, com a perda abaixo de 1% e a precisão em 62%.

7.5 Comandos de atendente e administrador

A plataforma Telegram permite que comandos personalizados sejam criados e, através da biblioteca *Python Telegram Bot*, presente na Subseção 6.5.4, é possível criá-los e atribuí-los ao *bot*. Todos os comandos dever possuir uma barra com prefixo para serem computados pelo *bot*. Os seguintes comandos podem ser utilizados por administradores:

- **registraratendente**: O comando recebe três valores, o ID de usuário, o ID de chat e o nome do usuário, respectivamente. Ao executá-lo, o sistema irá registrar o usuário informado como atendente. Ex: */registraratendente 0000002 000000004 Atendente1*.
- **removeratendente**: Recebe apenas o Id do chat do atendente como valor, caso o usuário esteja registrado como atendente, então este será removido da posição de atendente. Ex: */removeratendente 000000004*.

Os comandos seguintes podem ser utilizados por atendentes registrados:

- **registrarchave**: O comando recebe três dois valores, a chave e uma descrição, respectivamente. Ao ser executado o sistema grava esta chave, permitindo que algum usuário a informe ao criar chamados. Ex: */registrarchave chaveteste1 ChaveTeste*.
- **removerchave**: Irá remover uma chave cadastrada, recebendo sua descrição como parâmetro. Ex: */removerchave ChaveTeste*.
- **finalizarconversa**: Durante uma conversa com algum usuário, o atendente poderá executar este comando para encerrar o diálogo.

Por fim, o seguinte comando pode ser utilizado por qualquer usuário:

- **informacoes**: O comando retorna o ID do usuário e o ID de *chat* com o *bot*.

8 CONCLUSÃO E TRABALHOS FUTUROS

O trabalho promoveu muito conhecimento, especialmente para a inteligência artificial, os conceitos e técnicas para aprendizado de máquina. O aprendizado sobre AIML também se provou de grande auxilio para o desenvolvimento, pois é algo focado em um agente para diálogos. Além dos conhecimentos teóricos, a utilização de algoritmos e bibliotecas conhecidas proporcionou uma nova visão sobre a construção de sistemas e aplicativos para *data science*.

Alguns desafios ao longo do período de desenvolvimento foram encontrados, pois o aprendizado de conceitos de IA se torna complexo e árduo para um indivíduo que tem apenas um conhecimento raso da área e não possui um conhecimento extenso sobre matemática, que se provam necessários para a completa compreensão destes conteúdos. Outro problema enfrentado foi com a compatibilidade de certas bibliotecas com os recursos utilizados, que não correspondiam com os resultados esperados, que por fim, requereu uma busca por outras ferramentas para suprir a necessidade do sistema.

Para os testes e treinamento dos modelos, a falta de recursos na lingua portuguesa dificulta o aperfeiçoamento destes agentes, pois a quantidade de dados disponíveis e de alta qualidade, é baixa e de difícil acesso, portanto, os resultados dos testes não foram excepcionalmente bons, porém, suficientes para concluir-se que a ferramenta consegue cumprir o mínimo esperado, provendo desta forma, um *chatbot* que auxiliará uma equipe de suporte e/ou *service desk*.

Os resultados apontam que a ferramenta possui um potêncial para gerar a automação necessária à um setor de suporte, sendo que, com o aumento na utilização do *chatbot*, a sua acurácia e área de conhecimento aprimorem na mesma velocidade deste uso. Alinhando à evolução das tecnologias e do aprendizado de máquina, a metodologia atual supre a necessidade para que a burocracia e os processos possam ocorrer e, por fim, serem concluídos.

Para trabalhos futuros, a criação de uma interface gráfica para análise dos recursos e configuração do sistema seria de grande ajuda, além do aperfeiçoamento da base inicial de dados. Outro caminho viável é, a busca de um modelo de aprendizado diferente, que possa trazer resultados de forma mais precisa, com um requerimento de uma menor quantia de informações. A possibilidade da integração com outros sistemas também é relevante, podendo-se até criar uma API para integração das outras plataformas à este sistema.

REFERÊNCIAS

4Linux. *PostgreSQL e MySQL – Os bancos de dados mais utilizados no mercado*. 2017. Disponível em: <https://blog.4linux.com.br/postgresql-e-mysql/>. Acesso em: 05 de novembro de 2020. Citado na página 29.

ABREU, S. Automated architecture design for deep neural networks. p. 58, 2 2019. Citado na página 12.

ACCENTURE. Chatbots in customer service. 2016. Citado na página 1.

AIML Foundation. **AIML 2.0 Documentation**. 2018. Disponível em: <http://www.aiml.foundation/doc.html>. Acesso em: 02 de outubro de 2020. Citado 3 vezes nas páginas 5, 7 e 8.

ATLASSIAN. **Jira REST API examples**. 2020. Disponível em: <https://developer.atlassian.com/server/jira/platform/jira-rest-api-examples/>. Acesso em: 25 de abril de 2020. Citado na página 19.

BAHDANAU, D.; CHO, K.; BENGIO, Y. Neural machine translation by jointly learning to align and translate. p. 15, 5 2016. Citado na página 18.

BALODI, T. **How do Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) work in Deep Learning?** 2019. Disponível em: <https://www.analyticssteps.com/blogs/how-do-long-short-term-memorylstm-and-gated-recurrent-unitgru-work-deep-learning>. Acesso em: 02 de novembro de 2020. Citado 2 vezes nas páginas 15 e 16.

BURKOV, A. **The Hundred-Page Machine Learning Book**. [S.l.]: Andriy Burkov, 2019. Citado na página 11.

CAHN, J. Chatbot: Architecture, design, & development. 2017. Citado na página 3.

CASTRO, B. M. de; BARROS, W. A. Chatterbot em aiml: Chatterbot implementado utilizando aiml para manutenção de diálogo sobre futebol. 2015. Citado na página 3.

Data Science Academy. **Deep Learning Book**. 2019. Disponível em: <http://deeplearningbook.com.br/arquitetura-de-redes-neurais-long-short-term-memory/>. Acesso em: 28 de outubro de 2020. Citado 2 vezes nas páginas 14 e 16.

Data Science Academy. **Por Que a Linguagem Python é Tão Popular em Machine Learning e Inteligência Artificial?** 2020. Disponível em: <http://datascienceacademy.com.br/blog/por-que-a-linguagem-python-e-tao-popular-em-machine-learning-e-inteligencia-artificial/>. Acesso em: 05 de novembro de 2020. Citado na página 29.

Didática Tech. **O que é TensorFlow? Para que serve?** 2020. Disponível em: <https://didatica.tech/o-que-e-tensorflow-para-que-serve/>. Acesso em: 21 de novembro de 2020. Citado na página 30.

DUGAR, P. **Attention — Seq2Seq Models**. 2019. Disponível em: <https://towardsdatascience.com/day-1-2-attention-seq2seq-models-65df3f49e263>. Acesso em: 02 de novembro de 2020. Citado na página 18.

EDUCBA. **Intelligent Agents**. 2020. Disponível em: <https://www.educba.com/intelligent-agents/>. Acesso em: 26 de outubro de 2020. Citado na página 10.

ESTRELLA, C. **O Que é GitHub e Para Que Serve**. 2019. Disponível em: <https://www.weblink.com.br/blog/programacao/o-que-e-github/>. Acesso em: 06 de novembro de 2020. Citado na página 29.

FEDUS, W. et al. Revisiting fundamentals of experience replay. 2020. Citado na página 12.

FURTADO, T. **Draw.io é ótimo para criar gráficos e desenhos sem baixar nada**. 2013. Disponível em: <https://www.techtudo.com.br/tudo-sobre/drawio.html>. Acesso em: 06 de novembro de 2020. Citado na página 30.

GANEGEDARA, T. **Is the race over for Seq2Seq models?** 2020. Disponível em: <https://towardsdatascience.com/is-the-race-over-for-seq2seq-models-adef2b24841c>. Acesso em: 14 de novembro de 2020. Citado na página 14.

GONÇALVES, L. **O que é Sequence-to-sequence em deep learning?** 2018. Disponível em: <https://medium.com/luisfredgs/o-que-%C3%A9-sequence-to-sequence-em-deep-learning-9f8857a423ca>. Acesso em: 04 de novembro de 2020. Citado na página 18.

GRANDIC, I. **How LSTM's work**. 2019. Disponível em: <https://medium.com/@izzygrandic/how-lstms-work-263ac4e412ba>. Acesso em: 02 de novembro de 2020. Citado na página 16.

GREGORIO, D. V. F. D. **Psycopg – PostgreSQL database adapter for Python**. 2020. Disponível em: <https://www.psycopg.org/docs/>. Acesso em: 14 de novembro de 2020. Citado na página 31.

GUPTA, M. **ML | Types of Learning – Supervised Learning**. 2020. Disponível em: <https://www.geeksforgeeks.org/ml-types-learning-supervised-learning/>. Acesso em: 26 de outubro de 2020. Citado na página 11.

JR., L. S. **Entendendo a biblioteca NumPy**. 2018. Disponível em: <https://medium.com/ensina-ai/entendendo-a-biblioteca-numpy-4858fde63355>. Acesso em: 06 de novembro de 2020. Citado na página 30.

JúNIOR, M. **Criando chatbots para Telegram**. 2020. Disponível em: <https://take.net/blog/chatbots/chatbot-para-telegram>. Acesso em: 14 de novembro de 2020. Citado na página 20.

KENESHLOO, Y. et al. Deep reinforcement learning forsequence-to-sequence models. p. 22, 4 2019. Citado na página 17.

KHAN, R.; DAS, A. **Build Better Chatbots: A Complete Guide to Getting Started with Chatbots**. 1. ed. Bangalore, Karnataka, India: Apress, 2018. Citado 2 vezes nas páginas 3 e 4.

L3 Software. **Jira Service Desk: software integra equipes e melhora comunicação**. 2020. Disponível em: <https://l3software.com.br/softwares/jira-service-desk-software-integra-equipes-e-melhora-comunicacao/>. Acesso em: 14 de novembro de 2020. Citado na página 19.

LARDINOIS, F. **Microsoft Launches Visual Studio Code, A Free Cross-Platform Code Editor For OS X, Linux And Windows**. 2015. Disponível em: <https://techcrunch.com/2015/04/29/microsoft-shocks-the-world-with-visual-studio-code-a-free-code-editor-for-os-x-linux-and-windows>. Acesso em: 04 de novembro de 2020. Citado na página 29.

LEE, M. **Actor-Critic Methods**. 2005. Disponível em: <http://incompleteideas.net/book/first/ebook/node66.html>. Acesso em: 21 de outubro de 2020. Citado na página 12.

MANASWI, N. K. **Deep Learning with Applications Using Python: Chatbots and Face, Object, and Speech Recognition With TensorFlow and Keras**. 1. ed. Bangalore, Karnataka, India: Apress, 2018. Citado na página 4.

MANIDEEP, V. **Chat Bot using Seq2Seq Model!** 2020. Disponível em: <https://medium.com/@vvsmanideep/interview-chat-bot-using-seq2seq-model-fe9059fffe64>. Acesso em: 26 de outubro de 2020. Citado na página 14.

MARATHE, Y. **Neural Machine Translation using Bahdanau Attention Mechanism**. 2020. Disponível em: <https://medium.com/analytics-vidhya/neural-machine-translation-using-bahdanau-attention-mechanism-d496c9be30c3>. Acesso em: 25 de novembro de 2020. Citado na página 18.

MARIETTO, M. das G. B. et al. Artificial intelligence markup language: A brief tutorial. 2020. Citado 2 vezes nas páginas 5 e 6.

MASSARO, A.; MARITATI, V.; GALIANO, A. Automated self-learning chatbot initially built as a faqs database information retrieval system: Multi-level and intelligent universal virtual front-office implementing neural network. p. 12, 2 2018. Citado na página 13.

MELO, S. **Entenda o que é Telegram e por que ele é um "oceano azul" do marketing digital**. 2018. Disponível em: <https://klickpages.com.br/blog/o-que-e-telegram>. Acesso em: 25 de abril de 2020. Citado na página 20.

MICROSOFT. **CNTK 303: Deep Structured Semantic Modeling with LSTM Networks**. 2017. Disponível em: <https://www.cntk.ai/pythondocs/CNTK_303_Deep_Structured_Semantic_Modeling_with_LSTM_Networks.html>. Acesso em: 21 de outubro de 2020. Citado na página 13.

MITTAL, A. **Understanding RNN and LSTM**. 2019. Disponível em: <https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e>. Acesso em: 28 de outubro de 2020. Citado na página 14.

NARKHEDE, S. **Understanding Confusion Matrix**. 2018. Disponível em: <https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62>. Acesso em: 26 de outubro de 2020. Citado na página 13.

NETO, A. R. P. **Introdução ao Numerical Python (Numpy)**. 2020. Disponível em: <http://www.opl.ufc.br/pt/post/numpy/>. Acesso em: 21 de novembro de 2020. Citado na página 30.

OLAH, C. **Understanding LSTM Networks**. 2015. Disponível em: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. Acesso em: 02 de novembro de 2020. Citado na página 15.

OLIVEIRA, L. **JIRA Service Desk: tudo que você precisa saber**. 2016. Disponível em: <https://blog.diferencialti.com.br/jira-service-desk/>. Acesso em: 25 de abril de 2020. Citado na página 19.

Pandorabots. **Core Concept**. 2020. Disponível em: <https://www.pandorabots.com/docs/>. Acesso em: 10 de outubro de 2020. Citado na página 5.

PEREIRA, M. de M. Aprendizado profundo: redes lstm. p. 40, 3 2017. Citado na página 15.

PHI, M. **Illustrated Guide to LSTM's and GRU's: A step by step explanation**. 2018. Disponível em: <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>. Acesso em: 31 de outubro de 2020. Citado 2 vezes nas páginas 14 e 16.

Python Jira. **Python Jira**. 2020. Disponível em: <https://jira.readthedocs.io/en/master/>. Acesso em: 06 de novembro de 2020. Citado na página 31.

Python Software Foundation. **Configuration file parser**. 2020. Disponível em: <https://docs.python.org/3/library/configparser.html>. Acesso em: 06 de novembro de 2020. Citado na página 31.

Python Software Foundation. **What is Python?** 2020. Disponível em: <https://docs.python.org/3/faq/general.html#what-is-python>. Acesso em: 05 de novembro de 2020. Citado na página 29.

Python-Telegram-Bot. **Introduction to the API**. 2020. Disponível em: <https://github.com/python-telegram-bot/python-telegram-bot/wiki/Introduction-to-the-API>. Acesso em: 06 de novembro de 2020. Citado na página 31.

RAJ, S. **Building Chatbots with Python: Using Natural Language Processing and Machine Learning**. 1. ed. Bangalore, Karnataka, India: Apress, 2019. Citado 2 vezes nas páginas 3 e 4.

RAMAMOORTHY, S. **Practical seq2seq**. 2016. Disponível em: <http://suriyadeepan.github.io/2016-12-31-practical-seq2seq/>. Acesso em: 28 de outubro de 2020. Citado 2 vezes nas páginas 15 e 16.

REDDY, K. P. **DSSM (Deep Semantic Similarity Model) - Building in TensorFlow**. 2017. Disponível em: <https://kishorepv.github.io/DSSM/>. Acesso em: 21 de outubro de 2020. Citado na página 13.

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. [S.l.]: Pearson Education, Inc., 2010. Citado 2 vezes nas páginas 10 e 11.

SHAH, D. **Generative chatbots using the seq2seq model!** 2020. Disponível em: <https://towardsdatascience.com/generative-chatbots-using-the-seq2seq-model-d411c8738ab5>. Acesso em: 26 de outubro de 2020. Citado na página 14.

SHARMA, A. **Understanding Activation Functions in Neural Networks**. 2017. Disponível em: <https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>. Acesso em: 25 de novembro de 2020. Citado na página 15.

Smart Sheet. **Artificial Intelligence Chatbots Are Changing the Way You Do Business and May Impact Your Bottom Line**. 2019. Disponível em: <https://www.smartsheet.com/artificial-intelligence-chatbots>. Acesso em: 02 de maio de 2020. Citado na página 10.

SOJASINGARAYAR, A. Seq2seq ai chatbot with attention mechanism. p. 18, 5 2020. Citado 3 vezes nas páginas 14, 18 e 39.

STERLING, K. **Program-Y**. 2020. Disponível em: <https://github.com/keiffster/program-y/wiki>. Acesso em: 06 de novembro de 2020. Citado na página 30.

TELES, F. **Entenda de vez o que é help desk e sua relação com o setor de TI**. 2018. Disponível em: <https://blog.deskmanager.com.br/o-que-e-help-desk/>. Acesso em: 25 de abril de 2020. Citado na página 19.

TensorFlow. **Por que usar o TensorFlow**. 2020. Disponível em: <https://www.tensorflow.org/about>. Acesso em: 06 de novembro de 2020. Citado na página 30.

The Latex Project. **An introduction to LaTeX**. 2020. Disponível em: <https://www.latex-project.org/about/>. Acesso em: 04 de novembro de 2020. Citado na página 29.

The PostgreSQL Global Development Group. **What is PostgreSQL?** 2020. Disponível em: <https://www.postgresql.org/about/>. Acesso em: 05 de novembro de 2020. Citado na página 29.

TI Inside. **Pesquisa mostra crescimento do uso de chats no atendimento**. 2015. Disponível em: <https://tiinside.com.br/29/06/2015/pesquisa-mostra-crescimento-do-uso-de-chats-no-atendimento>. Acesso em: 25 de abril de 2020. Citado na página 20.

WALLACE, D. R. S. The elements of aiml style. 2003. Citado 2 vezes nas páginas 6 e 7.

WANG, Z. et al. Sample efficient actor-critic with experience replay. p. 1,2,3, 11 2016. Citado na página 12.

WEISZ, P. B. G. Sample efficient deep reinforcement learning for dialogue systems with large action spaces. p. 6, 2 2018. Citado na página 12.

WIJAYA, Y. S.; RAHMADDENI; ZOROMI, F. Chatbot designing information service for new student registration based on aiml and machine learning. 2020. Citado 4 vezes nas páginas 3, 5, 10 e 13.

WOOD, T. **What is Unsupervised Learning?** 2020. Disponível em: <https://deepai.org/machine-learning-glossary-and-terms/unsupervised-learning>. Acesso em: 26 de outubro de 2020. Citado na página 11.

Apêndices

APÊNDICE A – *Tags* Adicionais

A tag <random> permite que o programa selecione aleatóriamente uma resposta de uma lista presente no seu escopo. Cada item é determinado pela *tag* <*li*>, havendo a possibilidade de outras *tags* serem declaradas dentro do item.

```
...
<template>
    <random>
        <li>Olá, no que posso ajudar?</li>
        <li>Oi, como posso te ajudar?</li>
        <li>Olá, o que deseja?</li>
    </random>
</template>
...
```

A *tag* <*srai*> é utilizada para direcionar a categoria atual à outra categoria existente, criando uma recursividade. Normalmente utilizada para normalizar uma frase, ou seja, auxilia no redirecionamento de sentenças gramáticamente erradas, porém pode direcionar diversas entradas a uma resposta padrão.

No trecho abaixo, a primeira categoria recebe qualquer sentença que inicie com a palavra "Olá", já a segunda recebe qualquer sentença que inicie com a palavra "Oi". Dessa forma, ela direciona para a última categoria descrita, assim, a resposta ao usuário será uma das frases dentro da *tag* <*random*>. Pode-se então, como utilizado neste trabalho, criar diversos padrões de sentenças com um contexto de saudação, onde sempre são direcionadas para uma categoria específica, deixando o *chatbot* com aspecto mais humano.

```
<category>
    <pattern>OLA ^</pattern>
    <template>
        <srai>OI</srai>
    </template>
</category>

<category>
    <pattern>OI ^</pattern>
    <template>
```

```
            <srai>OI</srai>
        </template>
    </category>

    <category>
        <pattern>OI</pattern>
        <template>
            <random>
                <li>Olá, no que posso ajudar?</li>
                <li>Oi, como posso te ajudar?</li>
                <li>Olá, o que deseja?</li>
            </random>
        </template>
    </category>
```

Para esclarecimento, abaixo há dois exemplos de um diálogo no contexto de uma saudação, utilizando das categorias acima.

```
Usuário: Olá, me chamo Rodrigo.
Bot: Oi, como posso te ajudar?
```

```
Usuário: Oi, poderia me ajudar?
Bot: Olá, o que deseja?
```

A *tag* <*condition*> define uma condição para executar o valor no seu escopo. No caso abaixo, a condição compara o valor da variável **sentenca** com o valor dos itens. O último item declarado se comporta, similar a outras linguagens de programação, como a palavra reservada *else*, portanto, caso o valor da variável não seja equivalente, a resposta será o conteúdo do último item.

```
<category>
    <pattern>*</pattern>
    <template>
        <think><set name="sentenca"><star/></set></think>
        <condition name="sentenca">
            <li value="SIM">RESPOSTASIM</li>
            <li value="NAO">RESPOSTANAO</li>
```

```
                <li>Não entendi! Repita por favor.</li>
            </condition>
        </template>
    </category>
```

A *tag* <*that*> define um contexto baseado na última sentença retornada pelo programa. Isto auxilia a controlar as sentenças do usuário dentro do contexto da conversa. Por exemplo, perguntas de sim e não podem ter uma categoria direcionada a resposta do usuário, onde o contexto está vinculado a pergunta anterior.

```
<category>
    <pattern>SIM</pattern>
    <that>VOCE ESTA CANSADO?</that>
    <template>Talvez voce devia descançar.
    Vou continuar aqui.</ template>
</category>
```

A *tag* <*think*> é utilizada para definir escopo a ser processado sem que seu conteúdo faça parte da resposta retornada. No exemplo abaixo, a resposta será apenas o texto "CRIAR-CHAMADO", assim, ocultado o texto CHAMADO, que se torna o valor do predicado *topic*, ou neste caso, o contexto.

```
...
<template>
    CRIARCHAMADO
    <think><set name="topic">CHAMADO</set></think>
</template>
...
```

A *tag* <*set*> é utilizada para criar variáveis e predicados atribuindo-os valores. Estas variáveis e predicados são específicos para cada cliente/sessão dentro do *chatterbot*. O caso abaixo atribui o valor CHAMADO ao predicado *topic*.

Predicados são criados pela propriedade *name* no corpo da tag e pode ser utilizado por todo corpo do objeto AIML. Já as variáveis são criadas com a propriedade *var*, que somente pode ser usada dentro do escopo da categoria onde foi criada.

```
...
<set name="topic">CHAMADO</set>
```

```
...
```

A *tag* <*get*> é usada para receber o valor de uma variável existente. Utiliza as mesmas propriedades que a *tag* <*set*>, dessa forma, pode receber o valor de um predicado ou de uma variável.

```
...
<template>
    <get name="topic"/>
</template>
...
```

A *tag* <*star*> pode ser declarada dentro da *tag* <*template*>. Recebe o valor da *wildcard* * existente no padrão da categoria, agindo como uma variável naquele momento.

```
<category>
    <pattern>OI, ME CHAMO *</pattern>
    <template>
        Olá, <star>, tudo bem?
    </template>
</category>
```

Caso exista mais de uma *wildcard* * presente no padrão, a propriedade *index* pode ser adicionada ao corpo da *tag* para identificar qual *wildcard* deve ser usada. As *tags* <*star*> e <*star index="1»* possuem a mesma funcionalidade. Também neste caso, seguindo da esquerda para a direita, cada *wildcard* recebe uma palavra da entrada e somente a última terá todo o resto da sentença.

```
<category>
    <pattern>OI, ME CHAMO * *</pattern>
    <template>
        Olá, Sr(a). <star index="2">, tudo bem? Posso lhe chamar
            apenas de <star>?
    </template>
</category>
```

A *tag* <*topic*> define o escopo de um tópico/contexto, categorias presentes em seu escopo são acessadas somente se o tópico estiver ativo. Dentro da AIML, o valor do tópico

padrão é denominado pela *wildcard* *.

No pedaço de código abaixo, ao receber a sentença CRIARCHAMADO, o interpretador processa a *tag* <*set*>, colocando o valor do tópico como CHAMADO, assim, as categorias dentro da *tag* <*topic*> serão priorizadas a partir da próxima entrada.

Neste exemplo, de um fluxo do *chatbot* deste trabalho, caso o usuário queira cancelar a criação do chamado em execução, o sistema deve enviar ao interpretador a entrada "CHAMADOCANCELADO", assim, o interpretador responde e atribui um valor vazio à variável *topic*, desta forma, voltando ao tópico geral (*).

```
1   <category>
2       <pattern>CRIARCHAMADO</pattern>
3       <template>
4           CRIARCHAMADO
5           <think><set name="topic">CHAMADO</set></think>
6       </template>
7   </category>
8
9   <topic name="CHAMADO">
10      <category>
11          <pattern>^ CANCELAR ^</pattern>
12          <template>
13              <srai>CANCELAR</srai>
14          </template>
15      </category>
16
17      <category>
18          <pattern>CHAMADOCANCELADO</pattern>
19          <template>
20              Ok, vou cancelar a criação deste chamado.
21              <think><set name="topic"></set></think>
22          </template>
23      </category>
24
25      <category>
26          <pattern>CRIARCHAMADO</pattern>
27          <template>
28              <random>
29                  <li>Você gostaria de criar um chamado?
30                  </li>
```

```
31                      <li>Ok, vamos criar um chamado?</li>
32                      <li>Certo, você gostaria de criar um
33                      chamado?</li>
34                  </random>
35              </template>
36          </category>
37      </topic>
```