SOLUÇÃO CHATBOT NO AMBIENTE ACADÊMICO DA UFRJ

Eduardo Fernando dos Santos Araujo

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Flávio Luis de Mello

Rio de Janeiro

Março de 2020

SOLUÇÃO CHATBOT NO AMBIENTE ACADÊMICO DA UFRJ

Eduardo Fernando dos Santos Araujo

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO.

Examinado por:

Autor:

Eduardo Fernando dos Santos Araujo

Orientador:

Prof. Flávio Luis de Mello, D.Sc.

Examinador:

Prof. Heraldo Luis Silveira de Almeida, D.Sc.

Examinador:

Prof. Diego Leonel Cadette Dutra, D.Sc.

Rio de Janeiro – RJ, Brasil

Março de 2020

Declaração de Autoria e de Direitos

Eu, *Eduardo Fernando dos Santos Araujo* CPF *141.034.167-42*, autor da monografia *SOLUÇÃO CHATBOT NO AMBIENTE ACADÊMICO DA UFRJ*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.

Eduardo F. dos Santos Araujo

Eduardo Fernando dos Santos Araujo

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica - Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro - RJ CEP 21949-900

AGRADECIMENTO

Foi uma longa caminhada até aqui e eu logicamente não cheguei aqui sozinho. Primeiramente quero agradecer a própria UFRJ e sua política de cotas e bolsas, que me permitiu não somente entrar na universidade mas também concluir o meu curso de graduação. Os gastos com a faculdade são imensos, mesmo numa universidade pública, ainda mais para quem mora em outra cidade e tem despesas altas com o transporte. Depois que se entra na faculdade um dos grandes desafios é conseguir permanecer até o final, e as bolsas de auxílio nos primeiros anos de faculdade foram de grande ajuda.

Agradeço aos meus professores que, cada um da sua forma, acrescentaram e ajudaram a formar o profissional que eu sou hoje. Um agradecimento em especial ao Flávio Mello, que tem me acompanhado e orientado com muita paciência e sabedoria nessa última etapa da minha trajetória na UFRJ.

Agradeço também aos meus amigos da faculdade, que como dizem: “Ninguém se forma sozinho”, e não foi diferente, aprendi e fui salvo por eles muitas vezes, além de me impedirem de surtar durante o processo e fazer toda caminhada muito mais divertida. Um salve em especial para os meus amigos Pedro e Henrique.

Por fim, agradeço a minha família, por tudo. Por todo o suporte, físico, mental e financeiro. Muita coisa aconteceu nesses 7 anos de UFRJ e sempre tive o amparo da minha família, e provavelmente por isso não desisti. Eu venho de uma família muito humilde, porém, nunca me faltou nada, nunca me faltou carinho, apoio, incentivo aos meus estudos, e sei que essa conquista é tão importante para mim quanto para todos eles também.

RESUMO

Um *chatbot* é um programa de computador projetado para simular uma conversa inteligente. Sua aplicação é possível em diversas áreas e sua procura tem se tornado crescente no mundo atual. Para a criação desse tipo de assistente virtual muitas plataformas e *frameworks* de código aberto estão disponíveis. Neste trabalho foi realizado um estudo sobre o funcionamento dos *chatbots*, algumas ferramentas para a sua construção e uma efetiva implementação para o uso em um aplicativo de mensagens, o *Telegram*. Por fim, esta aplicação tem como objetivo sanar dúvidas e responder perguntas dentro do contexto universitário da Escola Politécnica da Universidade Federal do Rio de Janeiro. O serviço foi implementado e depurado, restando ser colocado em produção.

Palavras-chave: chatbot, bot, aplicação, Telegram, inteligência artificial, processamento de linguagem natural

ABSTRACT

A chatbot is a computer program designed to simulate intelligent conversation. Its application is possible in several areas and its demand has been growing in today's world. For the creation of this type of virtual assistant many open source platforms and frameworks are available. In this work, a study was carried out on the functioning of chatbots, some tools for their construction and an effective implementation for use in a messaging application, Telegram. Finally, this application aims to clarify doubts and answer questions within the university context of the Escola Politécnica of the Universidade Federal do Rio de Janeiro. The service was implemented and debugged, it remains to be put into production.

Keywords: chatbot, bot, application, Telegram, artificial intelligence, natural language processing

Sumário

Introdução **11**
1.1 Tema 11
1.2 Delimitação 11
1.3 Justificativa 11
1.4 Objetivos 12
1.5 Metodologia 12
1.6 Descrição 13

Fundamentação Teórica **14**
2.1 NLP 14
2.1.1 Natural Language Understanding (NLU) 15
2.1.2 Natural Language Generation (NLG) 16
2.2 Chatbot 17
2.3 Plataformas e Frameworks para construção de Chatbots 20
2.3.1 Bot Libre 20
2.3.2 IBM Watson 21
2.3.3 Chatterbot 21
2.3.4 Rasa 22
2.3.4.1 Rasa NLU 23
2.3.4.1.1 Pipeline 23
2.3.4.1.2 Componentes 27
2.3.4.2 Rasa Core 32
2.3.4.2.1 Histórias 33
2.3.4.2.2 Domínio 34
2.3.4.3 Rasa X 39
2.3.4.3.1 Controle de Versão 40
2.3.4.3.2 Módulo Talk to your bot 41
2.3.4.3.3 Módulo Conversations 42
2.3.4.3.4 Módulo Models 43
2.3.4.3.5 Módulo Training 43

3 - Proposta de Solução **45**
3.1 Problema: substituir o robô atual (Bot Libre) por um novo 45
3.2 Implementação 47
3.2.1 Preparação do ambiente e criação do projeto com o Rasa 47
3.2.2 Migração entre plataformas 48
3.2.4 Configurações do assistente Rasa 50
3.2.4.1 Definição do pipeline 51
3.2.4.2 Políticas de Treinamento 53

3.3 Deploy no servidor 54
3.3.1 Instalação do Rasa X 54
3.3.2 Criação de domínio e configuração do DNS 61
3.3.3 Instalação do Certificado SSL 64
3.3.4 Integração com o Telegram 65

4 - Considerações finais 68
4.1 Conclusão 68
4.2 Trabalhos futuros 72
4.2.1 Migrar chatbot e servidor 72
4.2.2 Acompanhar atualizações do framework 73
4.2.3 Ampliação dos dados de treinamento 74

Referências Bibliográficas 75

LISTA DE ABREVIATURAS E SIGLAS

AIML	Artificial Intelligence Markup Language
API	Application Programming Interface
CSV	Comma Separated Values
CUI	Character User Interface
FAQ	Frequently Asked Questions
GCP	Google Cloud Platform
GUI	Graphical User Interface
IA	Inteligência Artificial
JSON	JavaScript Object Notation
NLP	Neuro-linguistic Programming
NLU	Natural-language Understanding
PLN	Processamento de Linguagem Natural
SSL	Secure Sockets Layer
SVM	Support Vector Machines
TCC	Trabalho de Conclusão de Curso
UFRJ	Universidade Federal do Rio de Janeiro

Capítulo 1

Introdução

1.1 Tema

O projeto baseia-se na construção de um *chatbot* para auxiliar os alunos quanto a possíveis questões referentes ao ambiente acadêmico, em especial ao da Escola Politécnica da Universidade Federal do Rio de Janeiro (UFRJ). Trata-se de um assistente virtual que é executado no ecossistema do aplicativo de troca de mensagens *Telegram*, capaz de fornecer informações ao seus usuários de forma rápida e dinâmica.

1.2 Delimitação

Apesar da universidade receber alunos de diferentes regiões do mundo, falantes de diferentes línguas, o *bot* contemplará apenas perguntas e respostas em português. Foram utilizadas apenas as versões gratuitas dos serviços empregados, o que, portanto, pode apresentar algumas restrições ao uso pleno das plataformas usadas como objeto de estudo deste trabalho. Por se tratar de novas tecnologias e por haver uma comunidade muito ativa neste ramo, essas ferramentas encontram-se em contínuo desenvolvimento. Utilizaremos nesta pesquisa as versões dos frameworks desenvolvidas até o mês de fevereiro de 2020.

1.3 Justificativa

A necessidade de interação usuário-sistema por meio da comunicação em linguagem natural usando interfaces computacionais vem aumentando, sendo alvo de cada vez mais estudos e pesquisas [1]. Atualmente, o desenvolvimento da Inteligência Artificial (IA) permite a criação de *bots* de conversação, conhecidos como *chatbots* ou *chatterbots* (*chat* = conversa, *bot* = robô). Os termos se referem a programas de computador que usam técnicas de IA, com o propósito de simular a habilidade de conversação de um agente computacional com um ser humano.

A utilização de *chatbots* vem sendo muito comum em diversas áreas como: educação, saúde, turismo, psicologia, empresarial e outras [2]. Estes são capazes de responder perguntas

sobre determinados domínios de conhecimento e com isso facilitar o usuário em alguma determinada ação que ele queira realizar, ou mesmo fazer com que ele evite de ter de pesquisar em vários locais para encontrar uma dada informação. Dessa forma, foi investigada uma maneira de modelar um *bot* que seja capaz de ajudar alunos, professores e coordenadores da Escola Politécnica da UFRJ.

É desejável, portanto, um entendimento a cerca de ferramentas e plataformas que possibilitem a construção desse tipo de aplicação de IA que está sendo utilizada em diversos meios e que sua necessidade e procura é crescente. Este estudo propõe o entendimento do que são, como funcionam e como criar esses *chatbots*. Desde plataformas e *frameworks* atuais que permitam a criação de *chatbots*, até uma efetiva implementação, em que haja uma capacidade alta de personalização e manutenção de seu funcionamento.

1.4 Objetivos

O objetivo geral deste projeto é a construção de um *chatbot* capaz de trazer facilidade aos estudantes no momento de sanar suas dúvidas quanto a rotina universitária, atividades extracurriculares, projeto final, dentre outros temas do ramo acadêmico da Escola Politécnica da UFRJ, diminuindo o tempo gasto na busca de informações nos diversos grupos, portais e websites institucionais, com uma alta taxa de acerto. Deste modo, possui como objetivos específicos: (1) pesquisar e analisar plataformas para a criação de *chatbots*; (2) desenvolver um assistente virtual para uso em aplicativos de troca de mensagens; (3) possibilitar a personalização do *bot* e atingir uma boa taxa de acertos às perguntas dos usuários.

1.5 Metodologia

No intuito de se alcançar os objetivos definidos anteriormente, foi decidido que a metodologia de pesquisa fosse subdividida em 3 momentos. Na primeira etapa, foi realizada uma revisão bibliográfica sobre o Processamento de Linguagem Natural e sua aplicação em *chatbots*, em conjunto com uma análise de ferramentas e frameworks que pudessem ser utilizados na construção desses assistentes virtuais. Em seguida na etapa 2, realizou-se efetivamente a implementação do *bot* e sua integração com a aplicação *Telegram*. Por fim, na etapa 3 há a conclusão da pesquisa e análise dos próximos passos.

1.6 Descrição

Nos capítulos seguintes são relatados os processos de estudo, pesquisa e implementação do *bot*. No capítulo 2 há uma breve descrição da fundamentação teórica na qual este trabalho foi embasado, desde como é feito o processamento de linguagem natural, do que é propriamente um *chatbot* e de alguns *frameworks* presentes no mercado. A escolha da solução e a implementação propriamente dita é apresentada no capítulo 3, onde se é comentado sobre os recursos da nova solução e a arquitetura utilizada. Por fim, no capítulo 4 há a conclusão do trabalho e os possíveis próximos passos a serem desenvolvidos.

Capítulo 2

Fundamentação Teórica

2.1 NLP

Processamento de Linguagem Natural (do inglês *Natural Language Processing* - NLP) é uma área de pesquisa e aplicação que explora como os computadores podem ser usados para entender e manipular texto ou fala em linguagem natural para fazer coisas úteis [3]. Pesquisadores que utilizam a NLP buscam reunir conhecimento sobre como os seres humanos entendem e usam uma linguagem para que ferramentas e técnicas apropriadas possam ser desenvolvidas para a construção de sistemas que entendem e manipulam linguagens naturais na execução de suas tarefas.

O objetivo da NLP é o de realizar um processamento de linguagem semelhante a um humano, a fim de projetar e construir software que analise, entenda e gere diálogos similares aos que os humanos reproduzem naturalmente. A finalidade é que o usuário, ao abordar seu computador, tenha a sensação de estar se dirigindo a outra pessoa.

Nas aplicações da NLP, as linguagens naturais são analisadas em diferentes níveis. Esses níveis existem por causa das regularidades e das propriedades exibidas por uma linguagem [4]. Esses são:

i. Fonologia: realiza a interpretação dos sons da fala. É a parte da linguística que se refere ao arranjo sistemático do som.

ii. Morfologia: lida com a estrutura das palavras na linguagem, que são compostas de morfemas - as menores unidades de significado, que refere-se à natureza que compõe as palavras. Como, por exemplo, os morfemas de prefixo, sufixo e radical.

iii. Lexical: contribui para o entendimento em nível de palavras. Palavras que têm apenas um sentido ou significado possível podem ser substituídas por uma representação semântica desse significado.

iv. Sintática: como as sequências são estruturadas. Requer uma gramática e um analisador, que verifica as palavras em uma frase, a fim de descobrir a estrutura gramatical da sentença. A saída desse nível de processamento é uma representação da sentença que revela as relações de dependência estrutural entre as palavras.

v. Semântica: relacionada ao significado das palavras. Tem foco nas interações entre significados ao nível da palavra na sentença.

vi. Pragmática: preocupa-se com o uso da linguagem em situações e utilização do conteúdo do texto para compreensão. Algumas aplicações de NLP podem utilizar bases de conhecimento e módulos de inferência que exigem muito conhecimento de mundo, incluindo a compreensão de intenções, planos e metas.

vii. Discurso: o nível de discurso da NLP trabalha com unidades de texto maiores que uma sentença. Enfatiza nas propriedades do texto como um todo, que transmitem significado ao estabelecer conexões entre as frases.

O NLP é geralmente feito em duas etapas, *Natural Language Understanding* (NLU) e *Natural Language Generation* (NLG), que envolve, respectivamente, as tarefas de entender e de gerar o texto. Abaixo uma representação do relacionamento entre esses campos de estudo.

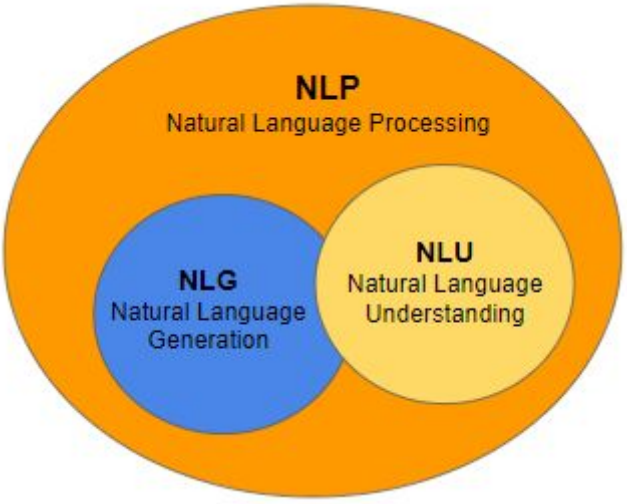

Figura 2.1: Diagrama NLP com as subseções NLG e NLU

2.1.1 Natural Language Understanding (NLU)

Um dos maiores desafios de sistemas conversacionais é o de entender o texto produzido por um usuário humano. As razões pelas quais isso é difícil se estendem por amplas razões, mas no centro delas está a infinita variabilidade da linguagem natural. Existem vários caminhos diferentes que um usuário pode seguir para expressar um único conceito. Diante dessa variabilidade, a capacidade de um humano de extrair significado é algo que talvez nunca seja completamente duplicado.

O *Natural Language Understanding* (NLU) tem como objetivo lidar com esses insumos não necessariamente estruturados que são governados por regras flexíveis, e convertê-los em uma forma estruturada que uma máquina pode entender e agir sobre estes [5]. O NLU busca o entendimento do significado do que o usuário ou a entrada que é dada quer dizer.

No contexto de um *chatbot*, o objetivo básico do NLU é determinar a conexão entre o texto recebido de um usuário e a resposta apropriada do sistema. Essa resposta pode ser para fornecer uma resposta simples a uma pergunta, ou envolver uma ação iniciada pelo usuário ou armazenar as informações fornecidas pelo usuário em resposta a uma pergunta. A maioria das abordagens ao NLU nos kits de ferramentas do *chatbot*, portanto, divide o NLU em duas subtarefas: classificação de intenção e extração de entidades.

Classificar a intenção do texto de um usuário é parte de um turno de diálogo iniciado pelo mesmo: o usuário expressou uma solicitação, ou fez uma pergunta, e o *chatbot* precisa classificar essa solicitação ou questão como pertencente a uma das solicitações que ele sabe como responder ou rejeitá-la, como um evento "sem correspondência". Isso determinará quais das ações ou respostas o *chatbot* irá fornecer.

Extrair entidades do texto do usuário é tipicamente parte de um turno de diálogo que foi (pelo menos implicitamente) iniciado pelo sistema. O usuário está fornecendo informações em resposta a uma pergunta. O trabalho da extração é selecionar as informações do texto e possivelmente normalizar essas informações quando, por exemplo, houver várias maneiras de expressar as mesmas informações.

Outro termo para essa tarefa de extrair entidades da tarefa do usuário para concluir as informações necessárias para um determinado caso de uso é o preenchimento de *slots*. Esse tipo de artifício pode ser interessante para que se evite repetições de perguntas para diferentes fluxos de conversa, ao reter uma informação que pode ser reutilizada em outros momentos.

2.1.2 Natural Language Generation (NLG)

Geração de Linguagem Natural (do inglês *Natural Language Generation* - NLG) é o processo de produzir frases, sentenças e parágrafos que são significativos a partir de uma representação interna. É uma etapa do Processamento de Linguagem Natural e acontece em três fases: identificar os objetivos, planejar como os objetivos podem ser alcançados,

avaliando a situação e as fontes comunicativas disponíveis, e efetivando os planos na forma de um texto. Os componentes do NLG são os seguintes [4]:

Gerador - para gerar um texto, precisamos ter um gerador ou um programa que exiba as respostas da aplicação, em frases fluentes relevantes para a situação.

Componentes e Níveis de Representação - O processo de geração de linguagem envolve as seguintes tarefas interligadas:

- Seleção de conteúdo: as informações devem ser selecionadas e incluídas na configuração. Dependendo de como essa informação é analisada em unidades representacionais, partes das unidades podem ter que ser removidas, enquanto outras podem ser adicionadas por padrão.
- Organização textual: a informação deve ser organizada textualmente de acordo com a gramática, deve ser ordenada sequencialmente e em termos de relações lingüísticas.
- Recursos lingüísticos: para apoiar a realização da informação, os recursos lingüísticos devem ser escolhidos. O que engloba a escolhas de palavras específicas, expressões idiomáticas, construções sintáticas e etc.
- Realização: os recursos selecionados e organizados devem ser transformados em um texto real ou uma saída de voz.
- Aplicação: armazena o histórico, estrutura o conteúdo que é potencialmente relevante e implanta uma representação do que realmente sabe.

2.2 Chatbot

Um *chatbot* pode ser entendido como um programa de computador autônomo que interage com usuários ou sistemas on-line, em tempo real, na forma de conversas, muitas vezes lúdicas e informais [6]. Dependendo da natureza das conversas, os *chatbots* são divididos em domínios abertos e fechados [7]. *Chatbots* de domínio aberto podem participar de uma conversa de forma livre com um usuário, não tendo nenhuma meta específica definida. Enquanto que *chatbots* de domínio fechado são criados para ajudar um usuário a atingir uma meta específica.

Por meio de perguntas e respostas, *chatbots* podem ajudar na interação humana com os computadores e possuem a capacidade de examinar e influenciar o comportamento do

usuário [8]. O *chatbot* é um programa de computador que imita a conversação inteligente, ao fornecer respostas adequadas a palavras-chave ou frases extraídas da fala, de modo a manter uma conversação contínua. A entrada para este programa é um texto em linguagem natural e a aplicação deve fornecer uma resposta que seja a resposta mais inteligente para a sentença de entrada. Esse processo é repetido à medida que a conversa continua, sendo a resposta dada através de texto ou fala.

Diferentemente de aplicações *desktop*, dispositivos móveis e Web, os *chatbots* não fornecem uma interface gráfica de usuário (GUI). Ao invés disso, se comunicam por meio da interface de usuário de conversação (CUI) ou pela integração com aplicativos de mensagens existentes [7]. O que traz como vantagens: a redução do custo e do tempo de desenvolvimento; a não necessidade do usuário baixar aplicativos específicos ou aprender a interagir com uma nova interface, ele continuaria usando o seu aplicativo de mensagens que já utiliza para outras conversas; a possibilidade de identificação do usuário, sem pedir que ele se registre e a facilidade de se adicionar novas funcionalidades, sem a necessidade de criação de uma aplicação separada por serviço.

Existem duas maneiras fundamentais que um *chatbot* pode se comunicar com usuários: por meio de elementos da interface do usuário, como botões, ou utilizando linguagem natural [7]. Os elementos de interface limitam a entrada do usuário a um número de ações predefinidas, enquanto a linguagem natural não tem restrições sobre possíveis entradas. Ao sugerir possíveis ações na forma de uma lista de botões ou similares, as possíveis interações tornam-se óbvias e mais simples para o usuário. Mesmo limitando a interação a certas ações, a característica principal de um *chatbot* permanece; a interação ainda é estruturada na forma de uma conversa, apenas que a escolha da entrada é restrita. Em determinados cenários, há muitas entradas de usuário possíveis para caber em uma lista fixa. Nestes casos, a linguagem natural é um método de entrada mais apropriado.

Enquanto que na interface com botões há um conjunto de entradas previsíveis, a entrada personalizada precisa ser analisada para se extrair informações. Lidar de forma coerente com uma interface de linguagem natural não restrita torna-se um grande desafio, uma vez que a entrada do usuário não é de forma alguma limitada a um único tópico, todos os tipos de reações inesperadas do usuário devem ser levadas em conta. É possível também se combinar ambos os métodos e aproveitar os benefícios de ambos, pois haverá uma diretriz

clara para interações e, ao mesmo tempo, o usuário permanece livre para expressar qualquer entrada personalizada possível.

Chatbots também são referidos como assistentes virtuais. Uma forma rudimentar de software de inteligência artificial que pode imitar a conversa humana. É uma das maneiras simples de transportar dados de um computador sem ter que pensar em palavras-chave adequadas para procurar em uma pesquisa ou navegar em várias páginas da Web para coletar informações [9]. O *chatbot* é uma ótima ferramenta para interação rápida com o usuário. Eles nos ajudam fornecendo entretenimento, economizando tempo e respondendo às perguntas com respostas difíceis de encontrar. Os *chatbots* podem ser analisados, melhorados e utilizados em vários campos, como educação, negócios, conversas on-line e etc.

Os *chatbots* podem potencialmente ser usados em várias situações instrucionais. [10] explica que *chatbots* podem desempenhar um papel útil para fins educacionais, porque eles são um mecanismo interativo em comparação aos sistemas tradicionais de *e-learning*. Os alunos podem interagir continuamente com o *bot* ao fazer perguntas relacionadas a um campo específico. Pode ser usado como uma ferramenta para aprender ou estudar um novo idioma; uma ferramenta para acessar um sistema de informação, uma ferramenta para visualizar o conteúdo de um corpus; e uma ferramenta para dar respostas a perguntas em um domínio específico.

A implementação de um projeto de um *chatbot* em um ambiente universitário pode ser particularmente útil para estudantes que buscam informações sobre admissão e seu currículo de curso [11]. Embora a maioria das informações esteja disponível na Web, os alunos geralmente gostam de interagir pessoalmente com o orientador. Um robô de bate-papo poderia ser projetado para fornecer aconselhamento acadêmico e simular esse tipo de interação. Nesse tipo de projeto, inclui-se as responsabilidades de ler as entradas do usuário e depois responder à consulta, tentando sempre manter a conversa dentro de um respectivo domínio, neste caso, relacionado ao ambiente acadêmico. Um FAQ *bot* é iniciado com um extenso brainstorming, no registro do máximo de perguntas frequentes e respostas possíveis, e a precisão do sistema tende a melhorar com a ampliação dessa base de conhecimento.

Assim como há um processo na criação de uma logo, os *chatbots* também precisam passar por uma etapa de projeção. Trata-se de estabelecer como se dará o comportamento do robô diante das interações que o público iniciará com eles. Como, por exemplo, pensar no que ele responderá se determinada pergunta for feita. E para isso, é necessário se conhecer

bem o público alvo e entender os possíveis diálogos que o público apresentará [12]. Afinal, todas as informações darão base para a criação de uma persona, onde esta servirá de exemplo para descobrir as mais variadas situações que o *chatbot* será exposto e, consequentemente, terá que interagir. Na realidade, essa etapa deverá ser constituída de duas partes:

- Criação da persona que interagirá com o *chatbot*: esse será o perfil do potencial usuário, ou seja, preverá situações onde o *chatbot* será exposto.
- Criação da persona que formará a personalidade do *chatbot*: esse será o perfil do assistente, ou seja, como a inteligência artificial reagirá diante das interações dos usuários. Nesse caso, deverá ser construída essa personalidade, como se fosse um humano que trabalhasse neste ramo, um funcionário perfeito, que tem o comportamento desenhado com base nos preceitos da identidade da organização .

Avaliando e compatibilizando as duas personas acima e seus comportamentos, se deve trabalhar na construção de um fluxo de comunicação, onde se encaixará "falas" de acordo com a estratégia do negócio.

2.3 Plataformas e Frameworks para construção de Chatbots

Há, atualmente, várias ferramentas, plataformas e frameworks para a criação de *chatbots* disponíveis no mercado. Elas variam em alguns fatores, como por exemplo, preço, funcionalidades, algoritmos, facilidade de extensão e integração com outros serviços e utilização de interface gráfica ou somente de código para o desenvolvimento do *bot*. Abaixo discutiremos sobre algumas das ferramentas avaliadas durante o desenvolvimento deste projeto.

2.3.1 Bot Libre

A plataforma *Bot Libre* [13] é um projeto de código aberto desenvolvido pela BotLibre.org. Permite a criação de *bots* de bate-papo de inteligência artificial para uso pessoal, comercial ou educacional. Fornece a possibilidade de integração a diferentes serviços e plataformas, como e-mail, SMS, *Twitter*, *Facebook*, *Skype*, *Telegram*, *Alexa*, *Google Home*, *Kik* e *WeChat*. Há a possibilidade ainda de criar um *bot* e incorporá-lo ao próprio site e criar aplicativos para Android e IOS. O *Bot Libre* permite que qualquer pessoa crie seu próprio *bot* de bate-papo gratuitamente, incluindo hospedagem gratuita, mesmo para *bots* comerciais.

Possui um plano *free*, com possibilidade de criação de até 10 *bots* e 500 chamadas de api diárias. Entretanto, para a criação do *bot* fornece uma interface não tão moderna, com poucos recursos, pouco personalizável e pouco intuitiva. Nos testes realizados, o *bot*, com a configuração *default*, somente era capaz de identificar uma pergunta que fosse muito próxima a pergunta cadastrada, o que resultava taxas de acerto não tão altas.

2.3.2 IBM Watson

O *chatbot* Watson da IBM [14] possui uma interface bastante simples, trata-se de uma plataforma online, em que skills de programação não são primordiais, onde é possível ir da criação ao deploy na nuvem em alguns cliques. O *Watson Assistant* traz consigo toda a tecnologia da IBM para reconhecimento de linguagem natural, com a possibilidade de fácil integração com outros canais e plataformas, como *Facebook*, *Slack* ou *Wordpress*. Está apto a trabalhar com diferentes idiomas, além de possuir uma estrutura de conversação pronta para ser utilizada em diferentes tipos de serviços, como de o atendimento ao cliente, bancário e de comércio eletrônico.

O Watson é uma ferramenta poderosa para a criação de robôs, preparada para diferentes tipos de aplicações e de fácil manuseio. Porém, ela não é 100% gratuita. Sua versão free é limitada a 10,000 chamadas de API por mês, o que seria pouco para os fins deste projeto.

2.3.3 Chatterbot

O *ChatterBot* [15] é uma biblioteca em Python, criada por Gunther Cox, que possibilita a geração de respostas com base em uma coleção de conversas previamente conhecidas, em qualquer idioma. O *ChatterBot* utiliza uma seleção de algoritmos de aprendizado de máquina para produzir diferentes tipos de respostas e, inclusive, aprender ao interagir com seres humanos e outras fontes de dados informativos.

Possui pré-criadas classes, que permitem a conexão com diferentes tipos de bancos de dados; alguns filtros, funções de pré-processamento e adaptadores lógicos, que determinam a lógica de como o *ChatterBot* trata os dados da entrada e seleciona uma resposta. Havendo a possibilidade de criação de novos adaptadores e customização de toda a forma como o *bot* processa a mensagem recebida do usuário.

Nota-se que é uma biblioteca que possui bons recursos para um *bot* de diálogo. Traz uma versatilidade às funcionalidades do *bot*, por ser programável. Entretanto, a documentação não é muito extensa e informativa, aparentemente não existem muitas opções prontas, e há uma certa dificuldade de se diferenciar perguntas e respostas. Por *default*, essas informações são inseridas juntas no treinamento, e, por conta disso, nos testes realizados, o *bot* acabava apresentando um comportamento um pouco diferente do esperado, ao utilizar como respostas frases que deveriam ser consideradas apenas para perguntas.

2.3.4 Rasa

O *Rasa* [16] é um framework em Python de IA conversacional de código aberto. O robô criado com o Rasa fornece suporte a qualquer linguagem e fácil integração a diversas plataformas de chat, como *Facebook*, *Slack* e *Telegram*, além de uma documentação bem informativa e detalhada.

O *Rasa* possui dois componentes principais - *Rasa NLU* e *Rasa Core*. O *Rasa NLU* é responsável pelo entendimento da linguagem natural, que identifica as intenções e extrai as entidades da frase de entrada . Enquanto que o trabalho do *Rasa Core* é essencialmente gerar a mensagem de resposta para o *chatbot*. Este obtém a saída do *Rasa NLU* (intenção e entidades), utiliza as informações do arquivo de configuração, do arquivo de domínio e das histórias de exemplo, e aplica os modelos de Aprendizado de Máquina para gerar uma resposta.

Outra funcionalidade bastante útil é o modo de aprendizagem interativa. Nele o programador fornece *feedback* ao seu *bot* enquanto fala com ele. Trata-se de uma maneira poderosa de explorar o que o *bot* pode fazer e uma maneira fácil de corrigir qualquer erro que ele cometa. É possível literalmente ensinar o *bot* a como reagir a cada situação.

O *Rasa* foi entendido como a melhor opção para o desenvolvimento deste projeto. Em virtude de ser um *framework* gratuito, possuir boa documentação, uma organização onde as informações e funcionalidades conseguem ser bem distinguidas, uma alta capacidade de customização, e efetiva participação no aprendizado do *bot*, fizeram com que esta fosse a escolha mais apropriada.

Recentemente, lançaram ainda o *Rasa X*, baseado na necessidade de levar aos seus usuários mais do que algoritmos, os criadores do *framework* desenvolveram uma ferramenta

poderosa para coletar e anotar dados de treinamento [17]. Trata-se de uma ferramenta para visualizar e filtrar conversas entre humanos e seu assistente *Rasa*, para transformar essas conversas em dados de treinamento, para gerenciar e versionar modelos e para facilitar o acesso de usuários de teste a seus assistentes.

Maior detalhamento sobre o funcionamento da biblioteca, assim como de seus componentes, que serão comentadas a seguir, podem ser encontradas no Blog Oficial [18] e na documentação [19].

2.3.4.1 Rasa NLU

O *Rasa NLU* é utilizado para extrair significado da entrada de texto. Recebe informações do usuário na forma de linguagem humana não estruturada e extrai dados estruturados na forma de intenções e entidades.

Intenções podem ser entendidas como rótulos para cada entrada do usuário que representam o objetivo ou o significado de suas mensagens. Por exemplo, um usuário que dá como entrada “Oi”, pode possuir uma intenção de saudação, porque o objetivo dessa entrada é cumprimentar.

Entidades são partes de uma informação que o assistente pode precisar em certo contexto. Por exemplo, na entrada “Oi! Meu nome é Eduardo.”, é dito um nome. O assistente deve ser capaz de fracionar o nome em uma entidade e lembrar disso durante a conversa, para fazer com que a interação seja natural.

O treinamento de um modelo NLU nos dados de treinamento permite que o modelo faça previsões sobre as intenções e entidades em novas mensagens do usuário, mesmo quando a mensagem não corresponde a nenhum dos exemplos que o modelo já viu antes. Para designar o assistente a entender as intenções e extrair as entidades que são definidas nos arquivos de dados de treinamento é necessário se construir um modelo NLU, que é criado a partir de um *pipeline* de treinamento, também conhecido como *pipeline* de processamento.

2.3.4.1.1 Pipeline

Um *pipeline* de processamento pode ser entendido como uma sequência de etapas de processamento usadas para extrair recursos de textos específicos e treinar determinados componentes que permitem ao modelo aprender os padrões subjacentes dos exemplos fornecidos.

No *Rasa*, há a possibilidade de criar um *pipeline* personalizado ou utilizar um dos pré-configurados. Existem dois *pipelines* pré-configurados a escolha, um deles é chamado *pretrained_embeddings_spacy*. Esse *pipeline* usa uma biblioteca chamada *Spacy*, que carrega modelos de linguagem pré-treinados para representar cada palavra em uma frase. Trata-se de uma representação vetorial das palavras, o que significa que cada palavra em uma mensagem do usuário é convertida em um vetor numérico denso.

Os vetores capturam o aspecto semântico e sintático das palavras, o que significa que palavras semelhantes devem ser representadas por vetores semelhantes. Os aspectos semânticos e sintáticos conhecidos das palavras aumentarão o desempenho dos modelos, mesmo que haja uma amostra de dados de treinamento muito pequena. Além disso, já que o treinamento não começa completamente do zero, o treinamento dos modelos costuma ser rápido, com tempos de iteração curtos.

Há algumas deficiências nesse tipo de configuração. Primeiro, as boas combinações de palavras não estão disponíveis para todos os idiomas, porque geralmente são treinadas em conjuntos de dados disponíveis ao público, em sua maioria em inglês, o que pode ser um fator limitante. Outro ponto é que as combinações de palavras geralmente são treinadas a partir de conjuntos de dados de treinamento bastante genéricos, por exemplo, artigos da Wikipedia e similares, o que significa que eles não cobrem palavras específicas de um domínio como acrônimos, siglas e etc.

Em situações em que seja necessário solucionar esses problemas, é recomendado usar o segundo tipo de *pipeline* pré-configurado, chamado de *supervised_embeddings*. A principal diferença desse *pipeline* para o anterior é que, em vez de usar um vocabulário pronto, este *pipeline* aprende tudo do zero, usando os exemplos que forem fornecidos no arquivo de dados de treinamento, da forma totalmente personalizada. Como não estaria usando nenhum conhecimento pré-aprendido, naturalmente, é preciso de mais exemplos de treinamento para que os modelos realmente aprendam e comecem a generalizar as entradas do usuário.

Para treinar um bom modelo com o *pipeline supervised_embeddings* serão necessários mais exemplos de treinamento do que em um *pipeline pretrained_embeddings_spacy.* Não há um número estrito para a quantidade de dados de treinamento; uma quantidade recomendada de exemplos para o uso do *pipeline supervised_embeddings* é de mil exemplos rotulados ou mais.

Na prática, uma configuração de *pipeline* de processamento é definida em um arquivo config.yml do projeto. Este arquivo é criado automaticamente no momento de criação de um novo projeto, ao executar o comando `rasa init` no *Rasa Command Line Interface* (*Rasa CLI*). Ao abrir a pasta do projeto inicial há um arquivo de exemplo de configuração, que consiste no indicador de idioma que corresponde ao idioma no qual está sendo construído o assistente e o nome do pipeline.

Para treinar um modelo NLU usando o *pipeline supervised_embeddings*, basta defini-lo no arquivo config.yml e executar o comando `rasa train nlu` no *Rasa CLI*. Este comando treinará o modelo junto a seus dados de treinamento e o salvará em um diretório chamado *models*.

Para testar o modelo recém-treinado executando o comando `rasa shell nlu` no *Rasa CLI*, que carrega o modelo NLU treinado mais recentemente e permite que seja testado seu desempenho, conversando com o assistente na linha de comando. Enquanto estiver no modo de teste é possível testar o modelo com várias entradas e ver como o modelo se comporta, por exemplo, verificando como a saída do modelo se parece com uma mensagem “Ola”. A saída do modelo consiste na intenção mais provável que foi prevista para esta mensagem de entrada e, ao lado, a confiança. Neste exemplo, obteve como retorno o JSON {“name: saudacao”, “confidence: 0.8844845294952393”}. Isso significa que o modelo tem 88% de certeza de que "Ola" é uma saudação. Se houver outras intenções, mostrará também uma lista de *intent_rankings*. Esses resultados mostram a classificação de intenções para todas as outras intenções definidas nos dados de treinamento.

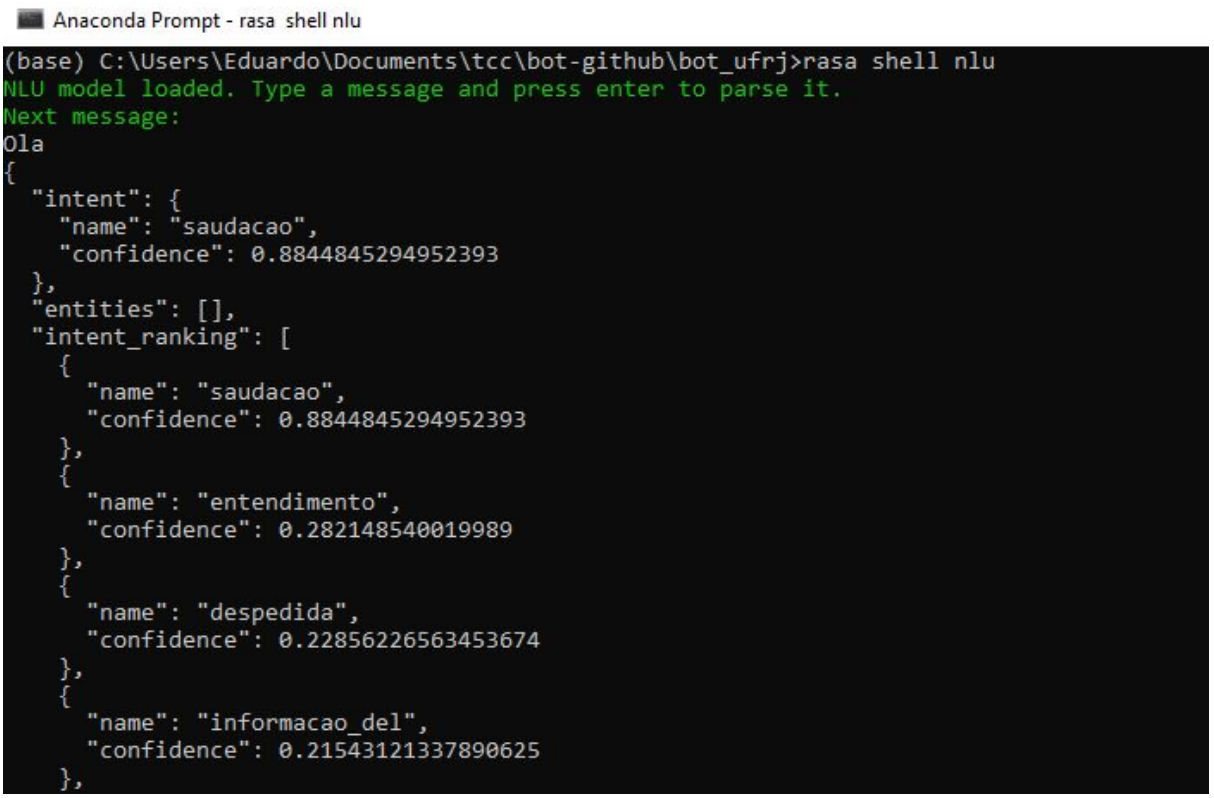

Figura 2.2: Análise das intenções previstas e suas respectivas confianças NLU no Rasa CLI

A medida que o projeto cresce, é natural a necessidade de alterar aspectos específicos do modelo, e que seja preciso explorar o que compõe o *pipeline*. Os *pipelines* pré-configurados são na realidade um atalho para uma lista de diferentes componentes responsáveis por etapas específicas do processamento, como podemos visualizar abaixo:

Figura 2.3: Comparando componentes para dois pipelines pré-configurados. Fonte: [20].

Para que seja possível a criação e configuração de *pipelines* personalizados, que melhor se adequem ao projeto, as definições dos componentes precisam estar claras. A seguir, a fim de melhor compreender porque os modelos estão se comportando de determinada maneira e como projetar os dados de treinamento para obter o melhor desempenho, entenderemos o que cada componente é, o que faz e porque é importante no processamento dos dados.

2.3.4.1.2 Componentes

As mensagens recebidas são processadas por uma sequência de componentes. Esses componentes são executados um após o outro no supracitado *pipeline* de processamento, que permite personalizar o modelo e ajustá-lo ao conjunto de dados. Existem componentes para extração de entidade, para classificação de intenção, seleção de resposta, pré-processamento e outros. É permitida também a criação de novos componentes, customizados de acordo com as necessidades do projeto .

Cada componente processa a entrada e cria uma saída. A saída pode ser usada por qualquer componente que vem depois desse componente no *pipeline*. Existem componentes que produzem apenas informações usadas por outros componentes no *pipeline* e outros que produzem atributos de saída que serão retornados após o término do processamento.

Anteriormente, foi comentando sobre os dois *pipelines* pré-configurados, *supervised_embeddings* e *pretrained_embedding_spacy*. Comparando, esses dois pipelines consistem em um conjunto de componentes responsáveis por tarefas específicas. De recurso de processamento de texto e extração de características, até definir o atual modelo que foi usado para classificar as intenções e extrair as entidades. Vamos analisar os componentes e entender melhor, pois uma vez que se é entendido o que eles fazem e o porque são necessários torna-se mais fácil de se criar *pipelines* customizados e fazer as devidas mudanças nos que já existem.

A. SpacyNLP

O *pipeline pretrained_embedding_spacy* começa com um competente chamado *SpacyNLP* que é responsável por carregar modelos de linguagem *Spacy* e ter isso pronto para os próximos passos do processo. Esse componente só é importante na utilização de modelos de linguagem *Spacy* e não é necessário para outras configurações de *pipeline*.

B. Tokenizer

Usando combinações de palavras pré estabelecidas ou aprendendo-as do zero, o *pipeline* deverá definir como as mensagens de usuários são processadas. É um passo importante para que os modelos aprendam os padrões de significado, os exemplos que estão sendo utilizados no treino devem ser divididos em menores unidades.

No *Rasa*, a frase é tratada usando um *tokenizer* de palavras, um componente que obtém uma entrada de usuário, de uma simples não estruturada linguagem humana, e a transforma em pedaços pequenos, *tokens*. Um *token*, por exemplo, pode ser uma palavra. O componente utilizado por padrão no *pipeline supervised_embeddings* é o *WhitespaceTokenizer*. O espaço em branco em uma frase vai criar um *token* para cada sequência de caracter. Este é situado para processar a maioria das linguagens.

Existem alguns *tokens* que podem ser escolhidos dependendo da abordagem utilizada. Caso o assistente esteja sendo construído em uma linguagem que requer um processo de *tokenizer* mais específico, é possível escolher outro *token* disponível ou criar um personalizado. Por exemplo, para processar a linguagem chinesa pode ser utilizado o *tokenizer* chamado *Jieba*, que foi construído especificamente para processar esta linguagem. Se a escolha for utilizar *Spacy* em modelos de linguagem prontos, pode ser utilizado o *SpacyTokenizer* que vem com regras de organização para estes modelos.

Uma vez que *tokenization* é um dos primeiros passos do processo aplicado na entrada dos usuários, no processo de organização do *pipeline*, este deve estar entre um dos primeiros componentes. Os tokens criados pelo *tokenizer* são geralmente usados por outros componentes subsequentes no *pipeline*, como por exemplo, um classificador de intenção ou um extrator de entidade, entre outros.

Para extrair as entidades o modelo necessita de componentes para reconhecimento de nomes. Assim como todo os outros componentes, o Rasa fornece uma gama de opções a escolha.

C. CRFEntityExtractor

O componente mais robusto é chamado *CRFEntityExtractor* que define um modelo, chamado *Conditional Random Field*, que aprende a identificar quais palavras em uma sentença são entidades e qual tipo de entidade elas são, observando as sequências de *tokens*. Escolhe uma palavra alvo e verifica palavras no entorno, extraindo características de texto de

todas essas palavras ao redor. Um componente CRF produz a saída que é diretamente adicionada no conjunto de saída do NLU model, as palavras em uma sentença que forem entidades e quais são suas etiquetas, quão confiante o modelo foi de fazer essas previsões e qual modelo foi usado.

Se houver uma pequena quantidade de dados de treinamento e estiver sendo utilizado modelos de linguagem *Spacy*, usando combinações de palavras pré-treinadas, para melhorar a performance pode ser utilizado o *SpacyEntityExtractor*, que pode aproveitar parte da marcação de fala e outros recursos para localizar as entidades nos exemplos de treinamento.

D. DucklingHttpExtractor

Algumas entidades podem seguir padrões bastante específicos, isso se aplica para entidades como data, números, códigos postais, entre outros. Há a opção de habilitar os modelos para extrair tal entidade usando componentes como o *DucklingHttpExtractor*. Uma biblioteca designada especificamente para extrair informações como datas, números, códigos postais, números de telefone, e-mails e outras informações sem treinar um modelo de extração de entidade do zero.

E. RegexFeaturizer

Para casos avançados de extração de entidades onde modelos baseados em *Machine Learning* não são o suficientes, é possível incrementar a extração de entidade do *CRFEntityExtractor* usando expressões regulares ou tabelas de pesquisa.

Expressões regulares combinam padrões de escrita difíceis, por exemplo, como pode ser feito o reconhecimento de códigos postais de 8 dígitos. Tabelas de pesquisa são usadas quando as entidades forem pré-definidas como um conjunto de valores, por exemplo, caso um país seja uma entidade, este pode ter 195 valores pré-definidos.

Para usar expressões regulares e tabelas de pesquisa é necessário adicionar o componente *RegexFeaturizer* antes do componente *CRFEntityExtractor* no pipeline. Expressões regulares e a tabela de pesquisa são adicionadas aos recursos do *CRFEntityExtractor*, que passa a marcar as palavras em que suas combinações resultem no conteúdo esperado pelas expressões regulares ou da tabela de pesquisa.

F. Featurizer

Para ativar os modelos para classificarem as intenções inicialmente são necessários dois componentes principais, um *featurizer* e um real modelo de classificação. O *featurizer* é utilizado para criar novas *features* (atributos ou variáveis de entrada) a partir dos *tokens* que podem ser usados pelo modelo de classificação para aprender padrões subjacentes e fazer as previsões.

Se não for utilizado um *pipeline pretrained_embeddings_spacy*, então um componente chamado *CountVectorsFeaturizer* é uma boa escolha para *featurizer* como componente. Este *featurizer* cria uma representação de *bag-of-words* da mensagem de um usuário usando o *CountVectorizer* do *Sklearn*. O modelo *bag-of-words* desconsidera a ordem das palavras em um corpo de texto e, em vez disso, concentra-se no número de vezes que as palavras aparecem. Assim, o *CountVectorizer* conta quanta vezes palavras distintas aparecem nas mensagens e as utiliza como entrada para o modelo de classificação de intenção.

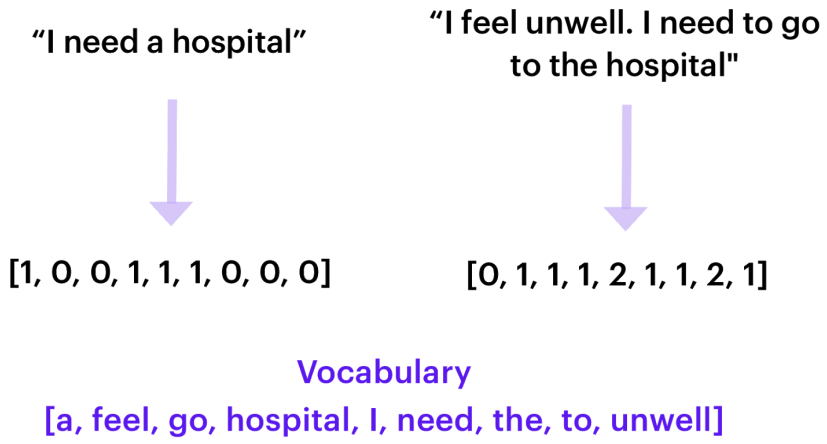

Figura 2.4: Exemplo de representação de frases com *bag-of-words*. Fonte: [20].

Em algumas situações o modelo vai funcionar melhor se em vez de contar palavras usar *n-grams*. Um *n-grams* é uma sequência de n itens em dados de texto, em que n representa as unidades linguísticas usadas para dividir os dados, por exemplo, por caracteres, sílabas ou palavras. Usar *n-grams* pode fazer com que o modelo seja mais robusto contra erros de digitação mas ao mesmo tempo incrementa o tempo de treinamento porque o vetor de *features* fica maior. Este fator pode ser especificado customizando o parâmetro *analyser* na configuração do componente, possuindo as opções 'word', 'char' ou 'char_wb', o último

cria caracteres *n-grams* somente a partir do texto dentro dos limites da palavra. Para este tipo de *featurizer*, um componente chamado *EmbeddingIntentClassifier* é a melhor opção para a classificação das intenções.

Se houver o uso de combinações de palavras pré-treinadas, o *SpacyFeaturizer* é o componente *featurizer* recomendado. Ele retorna vetores de palavras *SpaCy* para cada *token* que é passado ao *SklearnIntentClassifier* para a classificação de intenção.

G. Classificador de intenção

As *features* criadas pelo *CountVectorizer* alimentará o modelo de classificação de intenção e os resultados serão produzidos. O *EmbeddingIntentClassifier* funciona alimentado por entradas de mensagens do usuário e rótulos de intenção dos dados de treinamento em duas redes neurais separadas. As similaridades de cosseno entre o processamento das mensagens de entrada e os rótulos de intenção incorporados são calculadas. É feita a maximização das semelhanças com o rótulo de destino e a minimização entre as semelhanças com os incorretos. Os resultados são previsões de intenção que são expressas na saída final do modelo NLU.

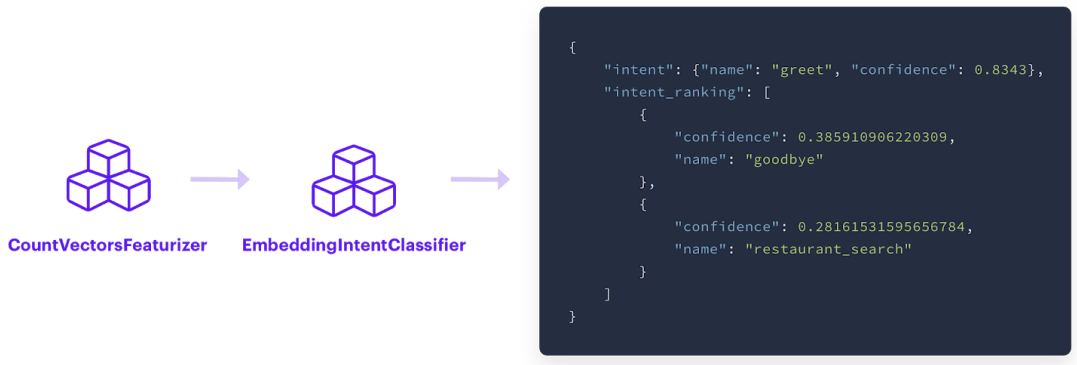

Figura 2.5: Representação do processamento feito com o EmbeddingIntentClassifier. Fonte: [20].

Ao usar combinações de palavras pré-treinadas, a recomendação é a de utilizar o componente *SklearnIntentClassifier* para classificação de intenção. Este componente usa os recursos extraídos pelo *SpacyFeaturizer*, bem como combinações de palavras pré-treinadas para treinar um modelo chamado SVM (*Support Vector Machine*). O modelo SVM prevê a intenção da entrada do usuário com base nos recursos de texto observados. A saída é um

objeto que mostra a principal intenção classificada e uma matriz listando as classificações de outras possíveis intenções.

Como os componentes de classificação de intenção aprendem a partir das features extraídas do *featurizer*, as *features* devem sempre vir antes do classificador como componente na configuração *pipeline*. Ao mesmo tempo, uma vez que as *features* usam os símbolos produzidos pelo *tokenizer*, um *tokenizer* deve vir antes do *featurizer*. Os componentes de classificação produzem os resultados que são diretamente adicionados às saídas do modelo NLU. Isso significa que o componente de classificação de intenção pode ser especificado no final do *pipeline* ou em qualquer lugar, contanto que os componentes responsáveis pelos processos de *featurizer* e *tokenizer* venham primeiro.

Construir processos customizados de *pipelines* e escolher componentes é um processo que geralmente leva a alguns passos para frente e para trás quando se constrói modelos. O que pode necessitar de atenção e serem necessárias as realizações de mudanças a medida que o sistema crescer. O treinamento também é uma parte crucial quando se quer construir bons modelos. Quanto mais treinamento puder ter e mais puder ser aprendido, melhor o modelo poderá ser, o que tem uma influência direta sobre quais componentes devem ser incluídos no *pipeline*.

2.3.4.2 Rasa Core

Foi discutido primeiramente como a ferramenta consegue habilitar o assistente para entender entradas de usuários, por meio de modelos NLU. Outro fator importante está relacionado a gestão de diálogo e como habilitar o assistente a responder e dirigir a conversa, mantendo o contexto.

Atualmente, a abordagem mais popular para construir uma conversa com assistente na indústria é usando um conjunto de regras ou um uma máquina de estados. Essa abordagem consegue lidar bem com o caminho feliz, situações onde os usuários perfeitamente seguem a conversa predefinida por um assistente. Entretanto, as coisas ficam confusas uma vez que usuários desviam do caminho. Acontece que é impossível escrever regras para qualquer diálogo possível. E uma vez que os usuários ficam à deriva do caminho feliz, o que cedo ou mais tarde vai acontecer com usuários reais, ocorrem quebras na conversa, o que vai resultar em bastante desapontamento na experiência do usuário. Além disso, assistentes baseados em

regras são difíceis de trabalhar, pois quando ficam grandes por causa do longo conjuntos de regras, fica difícil de gerenciar e manter.

Rasa tomou diferentes abordagens no gerenciamento de diálogo. Antes de criar regras e impor elas aos usuários, se ensina a máquina a aprender padrões de conversa de um dado conjunto de exemplos de conversa e prevê como o assistente deve responder em específicas situações baseado no contexto da história da conversa e em outros detalhes. O componente responsável pelo gerenciamento de diálogo no *Rasa* é chamado *Rasa Core*.

O *Rasa Core* usa o Aprendizado de Máquina para captar padrões de conversação em conversas de exemplo. Com base nesses padrões, o assistente pode generalizar, permitindo que o modelo preveja a próxima melhor ação a ser realizada em uma conversa. Isso permite que o modelo forneça uma resposta apropriada, mesmo quando a conversa não corresponder exatamente a nenhum dos exemplos de treinamento vistos antes.

É importante ressaltar que isso significa que não é preciso programar todas as conversas possíveis para o assistente com antecedência. É possível fornecer dados de treinamento contendo alguns exemplos de conversa quando criar o assistente pela primeira vez e, em seguida, reunir novos dados de conversa diretamente de interações reais do usuário. Usando o Aprendizado de Máquina, o assistente pode melhorar com o tempo, com base no que os usuários estão realmente dizendo.

2.3.4.2.1 Histórias

No gerenciamento de diálogo os dados de treinamento são chamados de histórias, exemplos de conversas entre usuários e assistentes escritas em um determinado formato. Esse formato utiliza a entrada de usuário, expressa como a intenção correspondente enquanto que as respostas dos assistentes são expressadas como nomes de ações.

Abaixo, um exemplo de história, que começa com dupla hashtag que marca o nome da história. Não é obrigatório escrever os nomes das histórias, entretanto, recomenda-se providenciar nomes de histórias descritivas, alguma coisa que descreva um exemplo do que a conversa é. O final de uma história é marcado com um novo nome, e então uma nova história começa com uma hashtag dupla.

```
## greet + location/price + cuisine + num people    <!-- name of the story - just for debugging -->
* greet
   - action_ask_howcanhelp
* inform{"location": "rome", "price": "cheap"}  <!-- user utterance, in format intent{entities} -->
   - action_on_it
   - action_ask_cuisine
* inform{"cuisine": "spanish"}
   - action_ask_numpeople        <!-- action that the bot should execute -->
* inform{"people": "six"}
   - action_ack_dosearch
```

Figura 2.6: Exemplo de um diálogo no formato de história Rasa. Fonte [21].

As entradas dos usuário são expressas como intenções rotuladas igualmente definidas nos dados de treinamento e dentro das chaves a entidade extraída deve ser definida provisionando o nome e o valor da entidade. As histórias de treinamento não têm que incluir a real mensagem do usuário. Com isso, torna-se possível alavancar os resultados do modelo NLU, por meio de intenções e entidades generalizadas.

As respostas do assistente são expressas como nomes de ações. São dispostas como linhas começando com um hífen, contendo o nome da ação. Há dois tipos de ação que podem ser usadas com o *Rasa*: expressões e ações customizadas. Expressões são cadeias de texto codificadas que representam o que o assistente irá dizer ao usuário. Dentro de uma história, uma expressão é rotulada com o prefixo ‘utter_’. Ações personalizadas executam código personalizado, como busca de dados de uma API. Nas histórias, ações personalizadas são rotuladas com o prefixo ‘action_’

2.3.4.2.2 Domínio

Um domínio é uma parte essencial para começar a construir um modelo de gerenciamento de conversa com o *Rasa*. Nele são definidos os universos em que o assistente vai operar: quais intenções e entidades vão estar disponíveis para aprender, quais ações customizadas e expressões o assistente deve responder e quais informações o assistente deve lembrar durante a conversa. Um domínio é geralmente especificado em um arquivo onde temos 3 principais seções: Intents, Actions e Templates. Abaixo um exemplo automaticamente criado num projeto inicial.

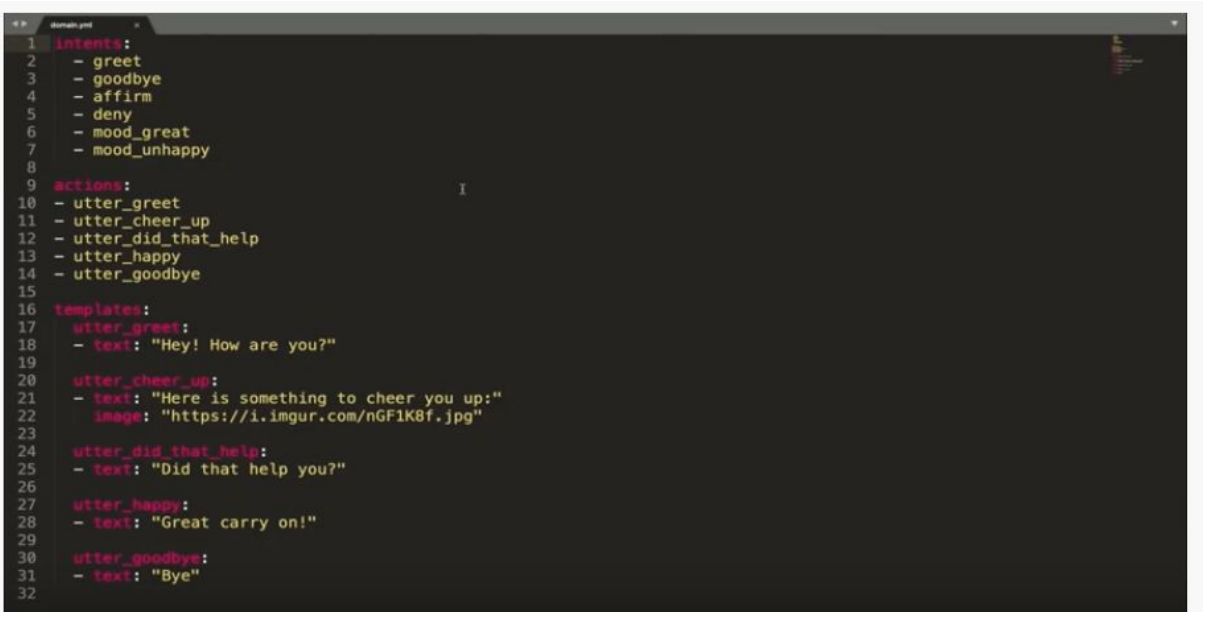

Figura 2.7: Arquivo domain.yml de um projeto inicial criado pelo Rasa

A seção chamada *Intents* define uma lista de intenções que o assistente é capaz de entender. Estes detalhes, devem vir do modelo NLU, logo, nessa seção devem ser fornecidos todos os rótulos de todas as intenções que existem no modelo. As entidades também podem ter influência em como o assistente vai decidir como responder as entradas dos usuários, por isso que as entidades também devem ser incluídas no arquivo de domínio, caso existam. Para fazer isso é preciso criar uma sessão chamada *Entitys* e uma lista de todas as entidades que existem no modelo que foi treinado para extrair.

A seção chamada *Actions* deve conter uma lista de todas as expressões e ações customizadas que o assistente deve usar para responder as entradas dos usuários. Isto deve ser preenchido com as actions utilizadas no arquivo de histórias.

A terceira seção no domínio é chamada *Templates*. Nela serão definidas as respostas do assistente, utilizada para responder com específicas respostas quando determinadas expressões são previstas. Pode haver mais de um *template* para cada expressão. Isso pode ir além de uma simples mensagem, podem incluir objetos como botões, imagens customizadas, etc.

Responder os *templates* diretamente no arquivo do domínio é o jeito mais fácil de definir qual mensagem um assistente vai mandar de volta ao usuário quando ele fizer alguma declaração, entretanto, existe outro jeito de conseguir o mesmo resultado criando ações customizadas. Ações customizadas são respostas do assistente que incluem algum código

customizado, esse código pode ser um simples texto de resposta de volta para o usuário, algum tipo de processo, fazer uma chamada a alguma API ou conectar na base de dados, de um modo geral, algum detalhe importante ou atípico que o assistente precisa lidar. Ações customizadas são definidas em um arquivo actions.py, assim como o nome da extensão sugere, será necessário escrever algum código em Python para a sua criação. Todas as ações customizadas devem ter nomes incluídos no arquivo de domínio para poderem ser utilizadas.

Os arquivos de domínios de todos os dados de treinamento e o de histórias são estritamente conectados. Não tem nenhuma regra específica para cada um e sobre qual deve ser criado primeiro, mas em geral, mudanças em um arquivo vai resultar em alguma mudança em outras partes de outros arquivos. No desenvolvimento de assistentes AI com *Rasa* são normalmente usadas algumas iterações e a expectativa é que esses arquivos sejam constantemente alterados.

A. Slots

Também podem ser incluídos nos arquivos de configuração e podem ser bem úteis para o gerenciamento de diálogos. Os *slots* podem ser usados para disponibilizar ao assistente como lembrar de importantes detalhes e usar eles em um certo contexto onde deve se dirigir a conversação. Como, por exemplo, conteúdos de identidades extraídas do modelo NLU, ou mesmo se a informação estiver fora, como em resultados de extrações dos dados de bases externas.

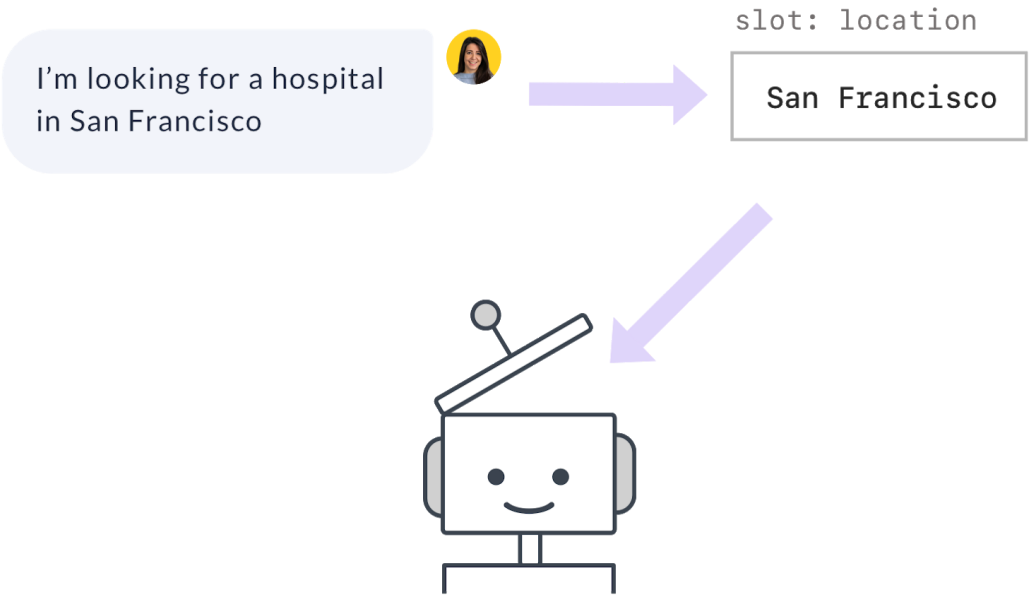

Figura 2.8: Representação do armazenamento da informação em slot. Fonte [22].

Os tipos de *slots* possuem uma influência direta em como o gerenciador de diálogos faz previsões. Em algumas situações apenas a presença ou ausência dos slots vai importar, mas em outros tipos de slots o conteúdo vai importar também. A seguir comentaremos os tipos de slots disponíveis no *Rasa*:

- *Text Slot*: os textos serão criados quando o rasa visualizar *slots* específicos. O valor real deste *slot* não faz nenhuma diferença, apenas é importante saber se o dado foi fornecido ou não. *Slots* com tipo texto são úteis quando o diálogo deve alternar entre turnos, dependendo se os usuários forneceram ou não detalhes específicos. Exemplo: se o *slot location* não tiver sido definido, um assistente deve solicitar esses detalhes; caso contrário, avance e leve a conversa adiante.
- *Boolean Slots*: é usado quando está se lidando com detalhes, onde há duas possibilidades, ser verdadeiro ou falso. Aqui não é só a presença ou ausência que vai importar, mas sim o conteúdo. O modelo de gerenciamento vai checar se esse *slot* é verdadeiro e vai utilizar essa informação quando estiver fazendo as previsões para a próxima ação.
- *Categorical Slots*: para este tipo, a presença e também o conteúdo vão ser levados em conta. Os *slots* categórico são usados quando um pedaço da informação puder assumir de 1 a N possíveis valores. Por exemplo, baixo, médio e grande. O gerenciador de diálogos vai checar o conteúdo do *slot* e levar as previsões até a próxima ação.
- *Float Slots*: usado para armazenar informações contínuas, como números de ponto flutuante. Aqui de novo a presença e o conteúdo irão importar, com os parâmetros de *min_value* e *max_value* é possível definir os valores mínimo e máximo possíveis para o *slot*. Tudo o que estiver acima de *max_value* será definido como *max_value*, enquanto tudo que estiver abaixo de *min_value* será definido como *min_value*.
- *List Slots*: se o modelo NLU extrair mais de um valor para uma entidade, convém armazenar todos os valores fornecidos. Um *slot* com a lista de tipos foi projetado para armazenar detalhes com vários valores.
- *Unfeaturized Slots*: algumas vezes pode ser útil usar os *slots* para histórias com algumas informações em que o assistente use-as apenas em específicas ações customizadas. Usar *Unfeaturized Slots* significa que o conteúdo do slot não vai ter influência em como o modelo do gerenciamento de diálogos vai realizar as previsões.

B. Políticas

A política no *Rasa* é um componente responsável por fazer a decisão de como assistente vai responder a próxima mensagem. Políticas de treinamento podem ser muito úteis em modelos de conversas mais sofisticados, uma vez que são capazes de prever a próxima ação baseada em diversos de detalhes, no histórico da conversa e no contexto.

No geral, as configurações de política consistem nos nomes das políticas e no conjunto de parâmetros que são usados para treinar os modelos que podem ser configurados pelo desenvolvedor. Abaixo, alguns exemplos de políticas que podem ser utilizadas:

- *MemoizationPolicy*: essa política é uma das mais simples que estão disponíveis os históricos e são treinadas dependendo do parâmetro max_history. Isto tenta combinar fragmentos do histórico atual com o histórico previsto nos dados de treinamento procurando previsões nos arquivos para as próximas ações. Esta política apenas memoriza as conversas dos seus dados de treinamento. Ela prevê a próxima ação com confiança "1.0" se essa conversa exata existir nos dados de treinamento, caso contrário, prevê "None" com confiança "0.0".
- *MappingPolicy*: essa política permite adicionar alguns trabalhos lógicos para o assistente, em que uma intenção específica deve ser sempre seguida de uma ação específica.
- *KerasPolicy*: essa política é usada para o uso de redes neurais, implementada por meio de uma biblioteca de *Deep Learning*, chamada *Keras*. É usada para prever as próximas ações, para isso considera vários fatores ao fazer uma previsão, incluindo: a última ação; as intenções e entidades extraídas pelo modelo NLU; os *slots* que foram definidos; turnos conversacionais anteriores. A configuração e arquitetura padrão do modelo é baseado no LSTM (Long Short Term Memory), um tipo de rede neural recorrente (RNN), que também pode ser alterado.
- *FallbackPolicy*: essa política lida com o gerenciamento de conversas com inteligência artificial, em situações onde é impossível prever as situações que os usuários vão perguntar, por isso, o assistente não é designado para realmente fazer alguma coisa quando os usuários perguntarem. Essa política permite que sejam respondidas frases como "Desculpa, não compreendi", ou "Eu não entendi", ou qualquer outra a escolha, que o assistente poderia dizer nesse tipo de situação.

- *TwoStageFallbackPolicy*: trabalha de um jeito similar a anterior, sendo que ao invés de executar imediatamente a ação de fallback, esta política pede ao usuário para verificar a intenção prevista. Se o usuário verificar que a intenção estava correta, a história continua. Se o usuário informar ao assistente que a intenção prevista não era o que queria dizer, o assistente solicitará que o usuário reformule a mensagem.
- *FormPolicy*: é usada em situações em que o assistente precisa coletar informações específicas do usuário antes de executar uma ação. Utilizada muito comumente para o preenchimento de formulários.
- *Transformer Embedding Dialogue Policy* (TEDP): utiliza o *Transformer*, um modelo de *Deep Learning* que produz resultados precisos em uma variedade de conjunto de dados e também lida bem com entradas inesperadas do usuário.

2.3.4.2.3 Aprendizagem interativa

No modo de aprendizado interativo, é fornecido o feedback do administrador ao seu *bot* enquanto fala com ele. Essa é uma efetiva maneira de explorar o que *bot* pode fazer e a forma mais fácil de corrigir os erros que ele cometa. Uma vantagem do diálogo baseado em Aprendizado de Máquina é que, quando o *bot* ainda não sabe fazer algo, ele poderá ser ensinado.

No modo interativo, o *Rasa* solicitará que seja confirmada todas as previsões feitas pelo NLU e pelo *Core* antes de continuar. O histórico do bate-papo e os valores das intenções previstas são impressas na tela, onde deve conter todas as informações necessárias para decidir qual será a próxima ação correta. Ao final, é permitido ao administrador do *bot* salvar as novas anotações, dados de treinamento e histórias criadas durante a aprendizagem interativa, adicionando de forma automática os dados obtidos aos arquivos com os dados de treinamento iniciais.

2.3.4.3 Rasa X

No desenvolvimento de assistentes de conversas com AI, um dos maiores desafios é ter um bom dado de treinamento. Tudo começa com o desenvolvedor manual que gera alguns exemplos de conversas para iniciar o processo de desenvolvimento. Porém, é muito difícil ter uma gama de variedade nos exemplos de treinamento NLU.

Quando isso chegar na generalização de dados de treinamento, humanos subconscientemente tendem a ter padrões específicos. Conversas hipotéticas não refletem realmente o mundo real, isso pode diferenciar drasticamente os usuários reais e como esses usuários querem conversar com seus assistentes. É por isso que usuários reais devem ser envolvidos no processo de desenvolvimento o mais cedo possível, porque as conversas que eles têm com seu assistente é o melhor treinamento de dados possível para alcançar a aprovação do assistente.

Ter alguns *feedbacks* de usuários reais vai ajudar a levar o assistente para o outro nível. Melhorar o modelo para um dado do mundo real adicionando novas habilidades ou introduzindo algumas mudanças para as que já existem. Um jeito de alcançar tudo isso é introduzir *Rasa X* no desenvolvimento.

O *Rasa X* é uma ferramenta que permite manipular assistentes construídos com o *Rasa*. Isso permite que o assistente seja compartilhado com usuários reais, colete conversas que eles têm com o assistente, as reveja e decida a arquitetura que deva ser utilizada. Uma vez que o *Rasa X* se tornou uma ferramenta para alcançar assistentes existentes, o único pré-requisito para usar o *Rasa X* é que haja um assistente construído com o *Rasa*.

Após a abertura no Web Browser da página do Rasa X é disponibilizada a *feature* de controle de versão e há as seções "Talk to your bot", "Conversations", "Models" e "Training".

2.3.4.3.1 Controle de Versão

Controladores de versão permitem o desenvolvimento de versões dos dados e dos códigos a fim de armazenamento, revisão de mudanças antes de irem para produção e execução de testes integrados com as mudanças propostas.

O *Rasa X* permite a conexão do assistente à plataformas que disponibilizam um controlador de versão, como o *GitHub* ou qualquer outro servidor remoto. Para isso, é necessário ter o assistente no servidor remoto com todos os dados e configurações de arquivos, assim como se tem em um computador local. Uma vez que o assistente está conectado ao *GitHub* pode-se habilitar o controle de versão no Rasa X.

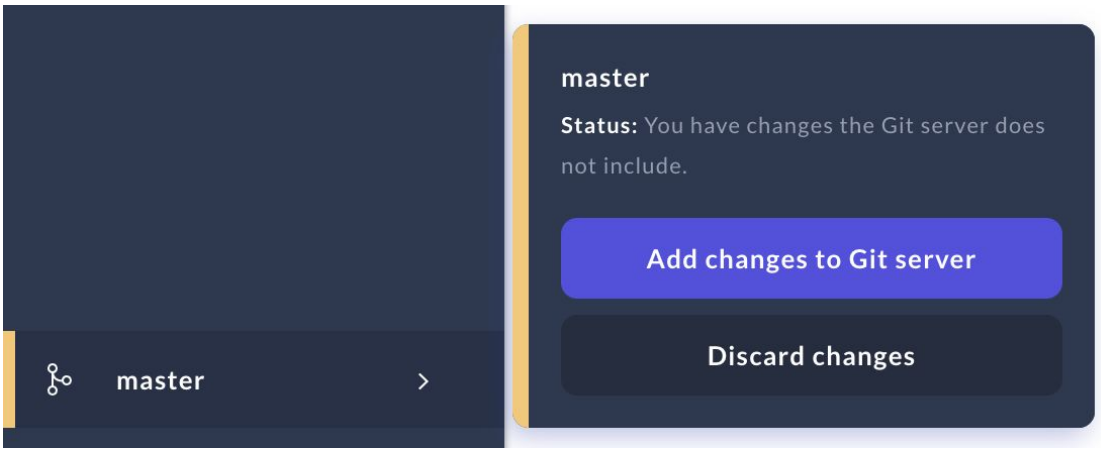

Figura 2.9: Controle de versão integrado ao Rasa X

Uma vez que houver uma mudança e houver uma nova anotação nos dados de treinamento, se forem alteradas as configurações do modelo ou qualquer outro aspecto, o processo de controle de versão integrada ficará laranja, sugerindo que novas mudanças foram feitas e não foram integradas ao servidor. Para incluir essas mudanças, é possível fazer o upload destas na *branch master*, ou criar uma nova *branch*. Uma vez que isso for feito, deverá ser encontrada uma solicitação de requerimento no repositório do assistente que sugere mudanças.

2.3.4.3.2 Módulo Talk to your bot

Esta seção possui dois modos: *Talk* e *Interactive Learning*. O modo *Talk* refere-se propriamente a um modo de conversa com a versão atual do *bot*, em que há, a cada interação, a visualização na interface de como o *bot* identifica a intenção da frase de entrada e que ação ele executa em seguida como resposta. Apresenta também a história apresentada e os slots que tenham sido preenchidos.

Já o modo *Interactive Learning*, é um modo de ensinamento ao assistente virtual. Nele é possível validar cada passo da conversação, como cada frase deverá ser identificada, qual deverá ser a próxima ação do *bot* e qual deverá ser a história para um determinado fluxo de conversação. Ao final, todas informações deste treinamento poderão ser salvas nos arquivos de configuração do *bot*.

2.3.4.3.3 Módulo Conversations

Depois que a interface do usuário do *Rasa X* é iniciada, é possível visualizar as conversas realizadas na guia *Conversations*. Nesta guia, encontram-se as conversas do *bot* com quaisquer usuários e há a possibilidade de marcar partes da conversas para futura visualização e anotar corretamente a intenção de uma determinada frase. A aba *Conversations* também permite filtrar conversas por duração, por intenções e ações, o que pode ser útil caso haja um grande número de conversas.

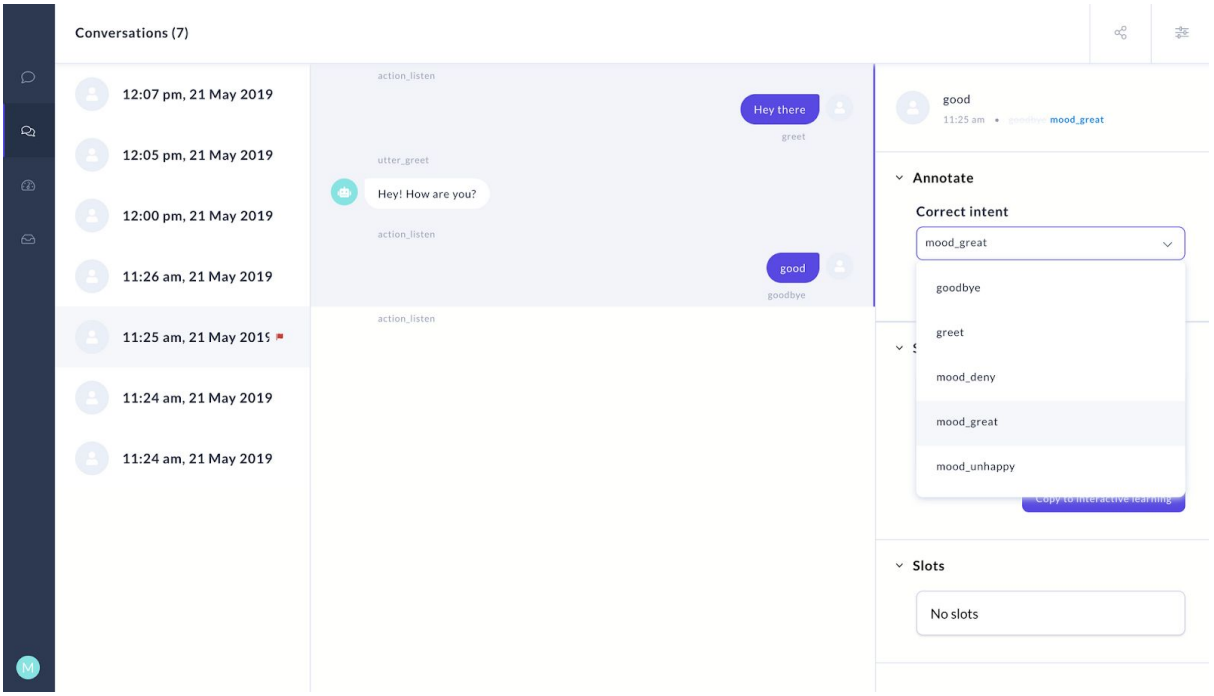

Figura 2.10: Aba conversations do Rasa X. Fonte [23].

A sugestão do blog da *Rasa* [23] é que o *bot*, após a geração de uma versão básica, seja entregue a usuários reais o mais rápido possível, por ser uma das únicas formas do assistente ser capaz de aprender diferentes padrões de conversação. Ao clicar no botão *Share your assistant with Guest Testers*, o *bot* poderá ser testado por usuários externo.

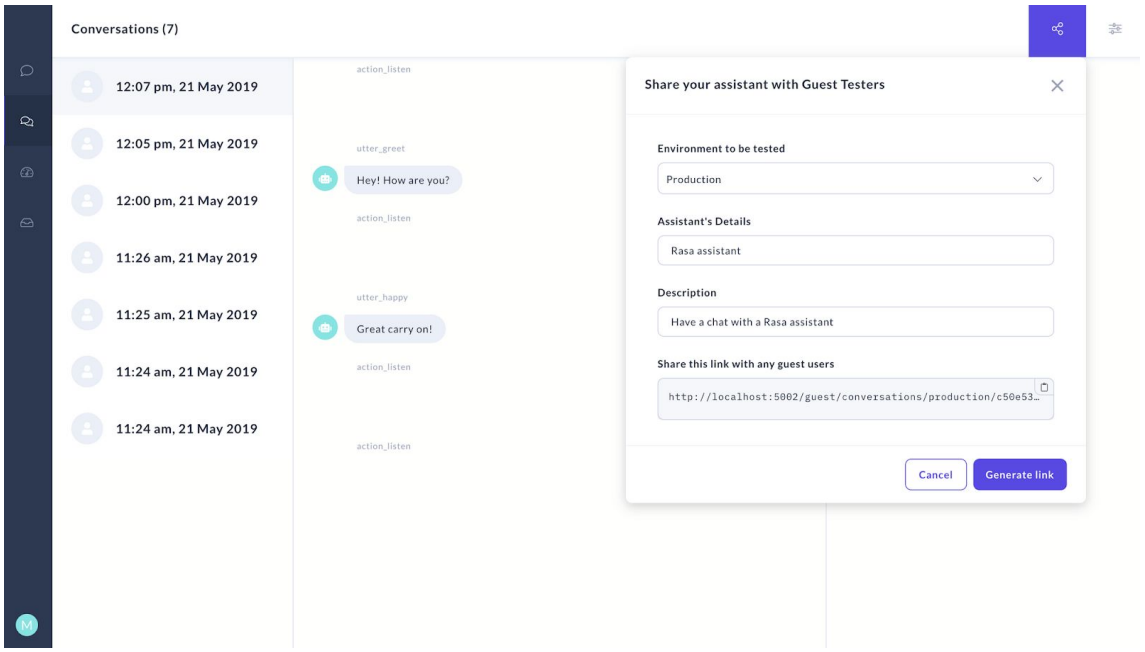

Figura 2.11: Janela de compartilhar o assistente com usuários testadores. Fonte [23].

Ao clicar em *Generate Link* será gerada uma url, em que os testadores do assistente poderão conversar com ele imediatamente, sem instalar ou configurar nada, e as conversas serão exibidas na guia *Conversations*. Lá, como dito no início da sessão, pode-se anotar os dados incorretos, criar mais histórias de treinamento e usá-los para melhorar continuamente o assistente ao longo do tempo.

2.3.4.3.4 Módulo Models

Após a realização das configurações e a inserção dos dados de treinamento, com um click no botão *Train* é possível gerar um novo modelo. Ao finalizar o treinamento, o modelo será exibido na aba *Models*. Nesta aba é exibido o nome de cada modelo, sua data de criação e seu status. A nova versão do *bot* gerada pode começar a ser utilizada após o click nos três pontos à direita do modelo, e o tornando ativo.

2.3.4.3.5 Módulo Training

Nesta seção é feita toda a configuração e treinamento do *bot*. É subdivida em outras 5 subseções:

- *NLU training* - É onde é feita a adição e a anotação de de novos dados a serem recebidos pelo *bot*. Realiza-se a classificação da intenção do conjunto de dados de treinamento referentes ao que pode ser dito ao *bot*.

- *Responses* - Aqui são configurados os *templates* e dados de resposta que serão reproduzidos quando determinada ação do *bot* for ativada.
- *Stories* - Para a visualização, criação e comparação de histórias, trata-se de caminhos que o *bot* irá seguir durante a conversa com o usuário. Pode ser definida a ação a ser executada a partir da identificação de uma dada intenção de entrada e possíveis desdobramentos e fluxos.
- *Configuration* - Define as regras de treinamento do modelo.
- *Domain* - Define o universo em que o assistente trabalha. Especifica as intenções, entidades, *slots*, *templates* e ações que o *bot* deve conhecer.

Capítulo 3

3 - Proposta de Solução

3.1 Problema: substituir o robô atual (Bot Libre) por um novo

Atualmente a Escola Politécnica da Universidade Federal do Rio de Janeiro (UFRJ) e, o Departamento de Engenharia Eletrônica e de Computação (DEL) dispõe de um assistente virtual presente no *Telegram*, o *R.U.Rbot*, que auxilia os alunos dos seus cursos de graduação a tirarem suas dúvidas acerca de assuntos relacionados à faculdade. Inicialmente, possuía uma ênfase em responder questões sobre temas no contexto de estudantes que já estavam finalizando a graduação, principalmente acerca do Trabalho de Conclusão de Curso (TCC), onde os alunos, em geral, possuem muitas dúvidas. Porém, perguntas sobre outros temas, no contexto universitário, têm surgido com o tempo, e a base de dados do *bot* vem sendo expandida para melhor atender às novas demandas.

O *chatbot* foi desenvolvido utilizando a plataforma *Bot Libre* [13], já citada anteriormente, em que o *bot* é criado a partir da interface gráfica do site. Com poucos cliques, torna-se possível adicionar perguntas, respostas e integrá-lo a uma plataforma, como o *Telegram*.

A interação entre criador e *bot* é feita em alto nível, o que facilita o processo de criação e, ao mesmo tempo, faz com que as estruturas sejam um pouco rígidas, o que resulta numa baixa capacidade de customização de todo o fluxo de informações e da forma que o *bot* realiza o aprendizado. Essa falta de controle na forma em que o *bot* configurado, armazena e processa os dados que recebe tem resultado num desempenho não tão satisfatório para os fins que este se destina.

Durante a análise do *chatterbot*, notou-se que o assistente, dentro do domínio de informações que foi fornecido a este, possuía um desempenho interessante durante as conversas. Apto, em muito dos casos, a exibir a resposta que deveria ser mostrada, dado que esteja bem próximo do que foi originalmente cadastrado. Assim como muitas vezes identificar assuntos totalmente não relacionados, ou que não possui de fato uma resposta e

exibir uma mensagem genérica de não entendimento ou indicar ao usuário que verifique os erros ortográficos e refaça a frase.

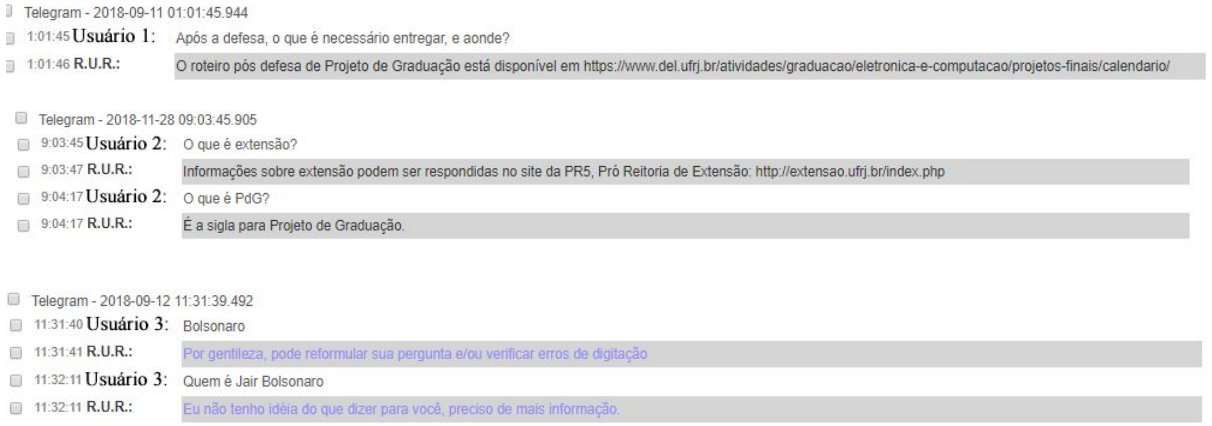

Figura 3.1: Exemplos de conversas bem sucedidas do bot gerado no Bot Libre

Entretanto, comete alguns deslizes em algumas situações. Como por exemplo, referenciando respostas à perguntas que não necessariamente estão relacionadas; ou "não entendendo" algo semelhante ao que foi cadastrado; ou fornecendo uma resposta errada a uma pergunta em que a resposta foi cadastrada à uma pergunta semelhante.

Figura 3.2: Exemplos de conversas mal sucedidas do bot gerado no Bot Libre

Partindo da ideia de melhorar sua taxa de acerto, em busca de uma maior capacidade de personalização, customização e ampliação de suas funcionalidades, foi iniciada essa

pesquisa. Após pesquisar algumas ferramentas de criação de *chatbot*, etapa descrita na seção anterior, escolhemos o framework escrito em Python, o *Rasa*.

3.2 Implementação

A implementação é voltada para a criação de um *chatterbot* FAQ. Que como já comentado, refere-se a um assistente de perguntas e respostas direcionado para um determinado domínio de conhecimento, neste caso, para o ambiente universitário da Escola Politécnica da UFRJ.

Quando pensamos em personas, temos como público alvo universitários, jovens em sua maioria na faixa de 18-30 anos, que têm um determinado linguajar e expressões idiomáticas, dentro do contexto da faculdade, com siglas específicas do ambiente universitário. Em geral, alunos que podem estar no início ou no final da graduação, ou mesmo pós-graduandos.

Do outro lado, temos um robô pensado como um assistente, com personalidade não tão formal mas também não muito informal, que tem que ser capaz de dialogar com esse público e ao mesmo tempo conseguir retornar informações institucionais de forma correta e sucinta, como um coordenador de curso de graduação. O assistente deve estar apto a responder questões sobre a graduação, principalmente quanto ao Trabalho de Conclusão de Curso (TCC), ou ao menos indicar algum tipo de referência ou profissional qualificado que disponha das informações. Assuntos externos, ou extremamente desconexos não devem ser de entendimento do *bot*, o objetivo é que este atenda bem a demanda para qual ele foi criado e não à temas diversos ou seja um *bot* de conversação.

3.2.1 Preparação do ambiente e criação do projeto com o Rasa

Primeiramente, foi realizada a instalação do *Rasa* numa máquina local, com sistema operacional Windows 10. A instalação do *Rasa* tem como pré-requisitos o gerenciador de pacotes *Pip* e a presença do Python nas versões 3.6 ou 3.7 [24]. Foi instalado, portanto, o *Anaconda Distribution* para Python 3.7, que é um instalador *open souce* que reúne uma série de utilitários, como o próprio Python 3.7, ferramentas utilizadas por desenvolvedores Python, como *Jupyter Notebook*, a *IDE Spyder*, além de famosas bibliotecas no ramo da Ciência de Dados, como *NumPy*, *Pandas* e *Scikit-learn* [25].

Após a instalação do *Anaconda*, no *Anaconda Prompt*, um *prompt* onde é possível a execução de comandos em Python e comandos do Anaconda, foi executado o comando `pip3 install rasa` para a instalação do Rasa. Em seguida, foi feita a criação de um projeto Rasa inicial, com o comando `rasa init --no-prompt`. O comando cria todos os arquivos que um projeto Rasa precisa e treina um *bot* simples com alguns dados de amostra. Sem a flag de `--no-prompt` serão feitas algumas perguntas sobre como é desejado que o projeto seja configurado, como por exemplo, o diretório destino de extração dos arquivos.

Após a execução do comando, os seguintes arquivos são criados:

__init__.py	um arquivo vazio que ajuda o python a encontrar as ações
actions.py	código para suas ações personalizadas
config.yml	configuração dos modelos NLU e Core
credentials.yml	detalhes para conectar-se a outros serviços
data/nlu.md	os dados de treinamento NLU
data/stories.md	as histórias
domain.yml	o domínio do assistente
endpoints.yml	para definição de endpoints, URLs onde o serviço possa ser acessado por outras aplicações clientes
models/<timestamp>.tar.gz	o modelo inicial

3.2.2 Migração entre plataformas

O passo seguinte para criar o novo *bot* foi coletar os os dados que o assistente antigo já possuía, que estava hospedado na plataforma *Bot Libre*. O *BotLibre* fornece dentro das configurações do *bot* uma seção de *Training* e *Chat Logs [26]*, onde é possível exportar um

arquivo em formato AIML (*Artificial Intelligence Markup Language*), com as perguntas e respostas cadastradas.

```
<?xml version="1.0" encoding="UTF-8"?>
<aiml version="1.1">
<category>
    <pattern>#GREETING</pattern>
    <template>Olá, em que posso ajudar?</template>
</category>
<category>
    <pattern>#DEFAULT</pattern>
    <template>
        <random>
            <li>Desculpe, mas não compreendi.</li>
            <li>Desculpe, eu não entendi, talvez você pudesse refazer a sua pergunta.</li>
            <li>Eu não tenho idéia do que dizer para você, preciso de mais informação.</li>
            <li>Por gentileza, pode reformular sua pergunta e/ou verificar erros de digitação</li>
        </random>
    </template>
</category>
<category>
    <pattern>SE VOCÊ JÁ DEFENDEU SEU PDG VEJA OS PRÓXIMOS PASSOS EM
    HTTPS://WWW.DEL.UFRJ.BR/ATIVIDADES/GRADUACAO/ELETRONICA-E-COMPUTACAO/PROJETOS-FINAIS/PROCEDIMENTOS-APOS-DEFESA-DE-P
    DG/VIEW</pattern>
    <template>Os procedimentos que antes da defesa de PdG estão em
    https://www.del.ufrj.br/atividades/graduacao/eletronica-e-computacao/projetos-finais/norma/</template>
</category>
<category>
    <pattern>É PRECISO TER CONCLUÍDO TODOS OS CRÉDITOS PARA APRESENTAR O PROJETO FINAL</pattern>
    <template>Não é preciso concluir todos os créditos para apresentar o PdG.</template>
</category>
<category>
    <pattern>O QUE EU PRECISO FAZER PARA ANTES DE COMEÇAR O PROJETO FINAL</pattern>
    <template>Os procedimentos que antes da defesa de PdG estão em
    https://www.del.ufrj.br/atividades/graduacao/eletronica-e-computacao/projetos-finais/norma/</template>
</category>
```

Figura 3.3: Arquivo com as antigas perguntas e respostas cadastradas do antigo assistente

Com esses dados foi possível alimentar, de forma inicial, o *dataset* do robô criado pelo *Rasa*. Entretanto, os arquivos utilizados pelo *Bot Libre* e o *Rasa* são diferentes, na questão do formato em que as informações devem ser dispostas no arquivo e da forma em si como as configurações são realizadas. Havia não só a necessidade de cadastrar possíveis perguntas e respostas no formato em que o *Rasa* identificasse, como também era demandada a criação do nome de cada intenção referente a cada pergunta, assim como a criação das histórias com as ações que o *bot* deverá executar em cada fluxo. Assim, houve a transformação dos dados exportados do *Bot Libre*, para que fossem adaptados aos requisitos do *Rasa*, de modo que ao final foram cadastrados 58 histórias, 51 intenções, 53 ações e 59 templates de respostas.

domain.yml

```yaml
intents:
  - saudacao
  - despedida
  - despedida_precoce
  - agradecimento
  - insulto:
      triggers: utter_insulto
  - entendimento
  - apresentacao
  - extensao
  - data_colacao_grau
  - informacao_colacao
  - aprender_libras
  - roteiro_projeto_final
  - pos_defesa_projeto_final
  - numero_avaliadores_banca
  - creditos_disciplina
  - quem_pode_orientar
  - projetos_graduacao_existentes
  - bandejao
  - contato_coordenadores
  - rcs
  - como_conseguir_orientador
  - fazer_projeto_graduacao
  - conhecimento_limitado
  - ementa_disciplinas
  - anos_estudo_engenharia
  - informacao_del
  - antes_defesa_projeto_final
  - pre_requisito_projeto_final
  - formato_projeto_final
  - eventos
```

stories.md

```
## saudacao
* saudacao
    - utter_saudacao
    - action_restart

## saudacao + despedida
* saudacao
    - utter_saudacao
* despedida
    - utter_despedida_precoce
    - action_restart

## despedida
* despedida
    - utter_despedida
    - action_restart

## apresentacao
* apresentacao
    - utter_apresentacao
    - action_restart

## agradecimento
* agradecimento
    - utter_agradecimento
    - action_restart

## entendimento
* entendimento
    - utter_entendimento
    - action_restart
```

Figura 3.4: Arquivos com as configurações do domínio e as histórias

Atualmente, a documentação do Rasa [27] fornece a possibilidade personalizar como o Rasa importa os dados de treinamento. Onde é possível, por exemplo, criar um script em Python que consiga carregar dados de treinamento, estando estes em outros formatos, ou mesmo obter dados de treinamento de diferentes origens.

3.2.4 Configurações do assistente Rasa

Diante de uma variada gama de possíveis configurações, uma etapa que demandou bastante testes foi a definição do arquivo de configurações. No final, mesmo com o entendimento do funcionamento dos componentes do *pipeline* e das políticas de treinamento, tornou-se necessário ir modificando aos poucos as combinações de componentes, políticas e hiperparâmetros para obtenção da configuração atual, de modo que as necessidades do projeto fossem atendidas.

3.2.4.1 Definição do pipeline

Inicialmente, foi pensado na escolha de um dos *pipelines* pré-configurados. O *pipeline supervised_embeddings* seria o mais adequado, uma vez que seriam construídos vetores de palavras personalizados para o domínio do projeto. O outro *pipeline*, o *pretrained_embeddings_spacy*, faz uso de modelos pré-treinados na obtenção da resposta final, o que poderia utilizar muitas palavras que não fazem parte do contexto de um FAQ acadêmico da UFRJ, atrapalhando a previsão do modelo.

Contudo, como já comentado, um pipeline pré-configurado é composto por vários componentes, que não necessariamente são utilizados. Assim, em virtude de otimizar o processo de treinamento e obtenção de respostas do *bot*, foi iniciado um processo de verificar o que poderia ser removido/adicionado dessa listagem de componentes.

```
pipeline:
- name: "WhitespaceTokenizer"
- name: "RegexFeaturizer"
- name: "CRFEntityExtractor"
- name: "EntitySynonymMapper"
- name: "CountVectorsFeaturizer"
- name: "CountVectorsFeaturizer"
  analyzer: "char_wb"
  min_ngram: 1
  max_ngram: 4
- name: "EmbeddingIntentClassifier"
```

Figura 3.5: Componentes que compõem o supervised_embeddings_pipeline. Fonte [28].

O primeiro componente, o *WhitespaceTokenizer*, tem como responsabilidade a criação de um *token* para cada sequência de caracteres separados por espaços em branco. Este é essencial para que os modelos consigam decompor as frases em partes menores e aprendam os padrões de significado das palavras. Este componente é *case sensitive* (diferencia letras minúsculas de maiúsculas) por padrão, e nesta implementação, entende-se que as palavras não precisam ser diferenciadas dessa forma. Portanto, além de manter esse componente, foi adicionado um atributo, *case_sensitive: false*, indicando que o processamento não diferencie letras maiúsculas de minúsculas.

Os componentes de *RegexFeaturizer*, *CRFEntityExtractor* e *EntitySynonymMapper* não foram utilizados. Até o momento não foi feito nenhum uso de regex para detecção de algum padrão, ou de entidades que precisassem ser extraídas das frases para alguma finalidade, muito menos de sinônimos de entidades. Isso pode ser pensado e adicionado no futuro, mas no momento esses componentes não teriam utilidade.

O próximo componente da lista é o *CountVectorFeaturizer*, que é o *featurizer* responsável por criar novas *features* a partir dos *tokens*, para que possam ser utilizados pelo modelo de classificação, de modo a aprender padrões subjacentes e fazer as previsões. O *pipeline* usa duas instâncias de *CountVectorsFeaturizer*. O primeiro cria *features* com base em palavras. O segundo cria *features* com base em caracteres *n-grams*. O segundo *featurizer* tende a ser mais poderoso, mas é recomendado pela documentação a manutenção dos dois *featurizers* para tornar a *featurization* mais robusta.

Por fim, o *EmbeddingIntentClassifier*, que utiliza as *features* extraídas pelo *CountVectorsFeaturizer* para produzir previsões de intenção. Este classificador faz uso de redes neurais em suas predições e há uma série de configurações que podem ser alteradas no classificador, como por exemplo, o número de neurônios nas camadas ocultas e o número de épocas. Foram mantidas as configurações *default*, com exceção do parâmetro *random_seed*, setado para 10, com o objetivo de obter resultados de treinamento reproduzíveis para as mesmas entradas.

```
pipeline:
    - name: "WhitespaceTokenizer"
      case_sensitive: false
    - name: "CountVectorsFeaturizer"
    - name: "CountVectorsFeaturizer"
      analyzer: "char_wb"
      min_ngram: 1
      max_ngram: 4
    - name: "EmbeddingIntentClassifier"
      random_seed: 10
```

Figura 3.6: Configuração final do pipeline do arquivo config.yml

3.2.4.2 Políticas de Treinamento

As políticas, como já comentado, ajudam a definir qual ação tomar a cada passo da conversa. Foram utilizadas as políticas *KerasPolicy*, *MappingPolicy* e *TwoStageFallbackPolicy*.

A *KerasPolicy* apresentou um desempenho mais próximo do esperado durantes os testes (quando comparada a *MemoizationPolicy* e a *TEDP)*, que possui uma rede neural implementada utilizando *Keras* para selecionar a próxima ação. Outra política utilizada foi a *MappingPolicy*, que permite mapear diretamente intenções para ações, onde a resposta a uma determinada intenção será dada independente do contexto da história. Por exemplo, no caso de detecção de insultos, o *bot* vai parar a história da conversa e responder sempre de uma determinada maneira.

Por fim, há a política *TwoStageFallbackPolicy*, que lida com a baixa confiança da NLU em dois estágios, tentando desambiguar a entrada do usuário. O objetivo aqui é evitar o fornecimento de informações incorretas e não deixar de dar uma resposta que possivelmente esteja cadastrada mas que há uma certa incerteza se ela seria apropriada.

Desse modo, foi definido que caso o nível de confiança do classificador NLU seja abaixo de 0.85, será dito ao usuário que o *bot* não entendeu muito bem o que ele gostaria de dizer, o fornece até 2 opções dentre as intenções mais bem ranqueadas (substituindo no nome da intenção o '_' por um espaço em branco) e uma última opção indicando que ele queria dizer alguma outra coisa que não estava na lista de opções.

Se o usuário escolher uma das 2 opções sugeridas, a história continua como se a intenção fosse classificada com alta confiança desde o início. Caso a última opção seja selecionada, o *bot* vai pedir para que a pergunta seja reformulada e caso o nível de confiança seja abaixo de 0.85, novamente as opções aparecerão da mesma forma. Se for escolhido pela segunda vez que o usuário deseja perguntar algo que não está nas opções, o *bot* vai responder que não entendeu e/ou que não tem aquele tipo de informação registrada até então.

```yaml
policies:
  - name: KerasPolicy
  - name: MappingPolicy
  - name: TwoStageFallbackPolicy
    nlu_threshold: 0.85
    fallback_nlu_action_name: "utter_default"
    fallback_core_action_name: "action_default_ask_affirmation"
    deny_suggestion_intent_name: "out_of_scope"
```

Figura 3.7: Configuração final das políticas no arquivo config.yml

Explicando de forma resumida a configuração da política, se uma previsão de NLU tiver um baixo índice de confiança, o usuário deverá confirmar ou não a classificação da intenção. A ação padrão chamada nesse caso é a *action_default_ask_affirmation*. Caso o usuário não encontre a classificação desejada, ele vai clicar na opção que indica uma outra intenção não listada, que vai disparar uma intenção de negação (*out_of_scope*). Essa intenção de negação é o gatilho para que a ação de reformular frase seja chamada, a *action_default_ask_rephrase*. Após o usuário responder, caso o nível de confiança NLU da intenção esteja novamente baixo, a ação *action_default_ask_affirmation* é chamada novamente. Por fim, caso haja mais uma negação, a ação *utter_default* é chamada, que dispara uma mensagem padrão de não entendimento por parte do *bot*.

O Rasa fornece as implementações padrão de *action_default_ask_affirmation* e *action_default_ask_rephrase*. A implementação padrão da *action_default_ask_rephrase* executa a resposta dada por *utter_ask_rephrase*, por isso, essa resposta foi sobrescrita no arquivo de domínio. Já a *action_default_ask_affirmation* foi sobrescrita no arquivo de ações customizadas, action.py, onde foi codificada, na linguagem Python, toda a lógica de funcionamento da política.

3.3 Deploy no servidor

Após ter realizado o *set up* inicial do projeto, o mesmo foi hospedado no *Github*. Com isso, foi iniciado o processo de deploy do Rasa X no servidor.

3.3.1 Instalação do Rasa X

Como já comentado, o *Rasa X* é uma ferramenta que permite o manuseio de assistentes construídos com o *Rasa*. Para o deploy do *Rasa X* no servidor foi utilizado o passo

a passo contido na documentação, que utiliza o *Docker*. O *Docker* é uma plataforma de virtualização que faz uso de containers para facilitar a criação, implementação e execução de aplicações. Os containers permitem o empacotamento de uma aplicação com todas as suas partes necessárias, o que possibilita a execução da mesma em qualquer outra máquina, uma vez que a imagem gerada inclui todas as dependências necessárias para executar o código.

O *deploy* que foi realizado utilizando o *Docker-Compose*. O *Compose* é uma ferramenta para definir e executar aplicativos *Docker* de vários containers [29]. Com o *Compose*, se é utilizado um arquivo YAML para configurar os serviços da aplicação. Em seguida, com um único comando, é criado e iniciado todos os serviços definidos na configuração.

A execução do *Rasa X,* via *Docker*, lança vários serviços integrados entre si para o funcionamento da plataforma:

Nome do Serviço	Descrição
rasa-x	*UI Rasa X* e *API HTTP*
rasa-production	Serviço *Rasa* executando um modelo treinado, usado para analisar mensagens de intenção e prever ações em conversas com usuários pelo canal de entrada ou interface do usuário do *Rasa X*
rasa-worker	Serviço *Rasa* usado para tarefas em segundo plano, como modelos de treinamento
db	Serviço de banco de dados *PostgreSQL*
duckling	Serviço de *duckling* usado para extração de entidade quando *DucklingHTTPExtractor* é incluído no pipeline do modelo
rabbit	O intermediário de mensagens usado para transmitir eventos de conversação entre rasa-production, rasa-x e o db

redis	Serviço agindo como uma camada de persistência para os *locks* de conversa. Um mecanismo de bloqueio para garantir que as mensagens recebidas para um determinado ID de conversa sejam processadas na ordem correta e bloqueie as conversas enquanto as mensagens são processadas ativamente
nginx	*Proxy* reverso usado para redirecionar solicitações para os diferentes serviços
app	Servidor de ação personalizado

O recomendado é o uso de sistema operacional *Linux* para a configuração do *Rasa X*, mas pode-se usar outro sistema operacional se houver o suporte para *Docker*. O *Rasa X* foi instalado em um uma instância do *Google Cloud Platform* (GCP). No GCP, após efetuado o login no console, foi criada uma nova Máquina Virtual (do inglês *Virtual Machine* - VM) no caminho *Compute Engine> VM Instance*.

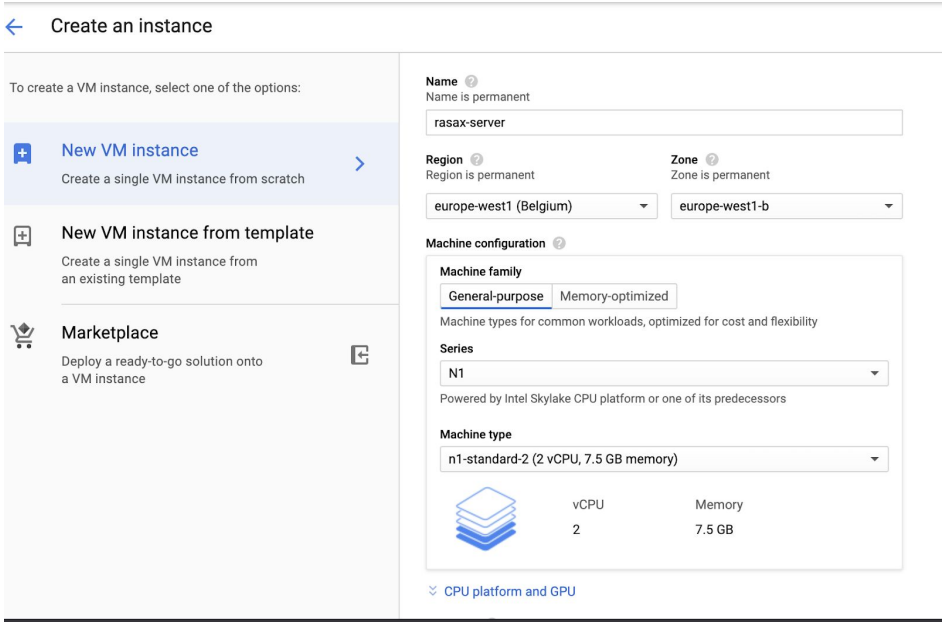

Figura 3.7: Configuração da Máquina Virtual no Google Cloud Platform para a instalação do Rasa X

Primeiramente, foi configurada a instalação da máquina virtual, definindo aspectos como nome, região que será localizada, tipo de máquina e CPU que será usado. O próximo passo, foi especificar o sistema de inicialização. No GCP há uma variação de diferentes distribuições *Linux*. Foi utilizada a versão *Ubuntu 16.04*, o tamanho do disco setado para 100 *gigabytes* e, nas configurações do Firewall, foram marcados os *boxes* para permitir o tráfego HTTP e HTTPS.

Figura 3.8: Seleção do Sistema Operacional e espaço em disco para Boot

Uma vez que a máquina foi configurada e criada, iniciou-se o processo de instalação do *Rasa X*. Foi baixado e instalado o *Rasa X* usando os comandos conforme imagem abaixo. O “0.25.1” refere-se a versão do Rasa X a ser instalada.

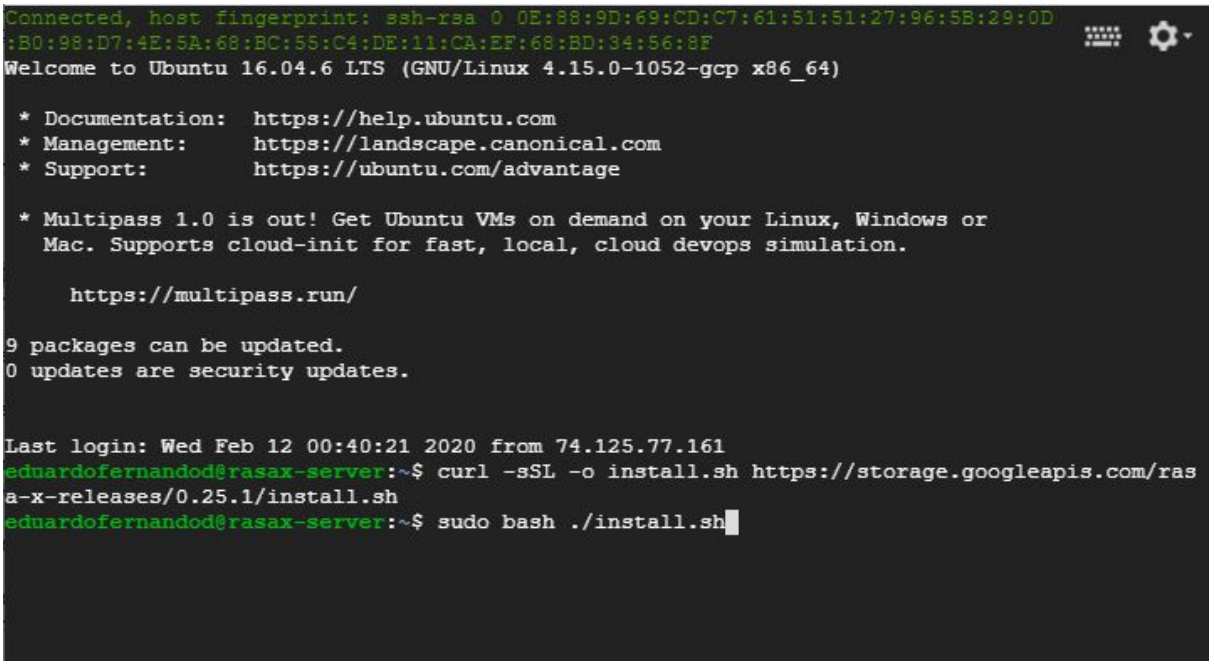

Figura 3.9: Instalação do Rasa X através do Prompt de Comando

O processo de instalação geralmente pode levar alguns minutos e iniciará após o aceite dos termos e condições para utilizar o *Rasa X*. Os arquivos vão ser instalados, por padrão, no diretório "etc/rasa/" no servidor, o que também pode ser customizado. Esses arquivos consistem em tudo que o *Rasa X* precisa para fornecer informações aos assistentes. Por exemplo, possui arquivos como: credenciais, o diretório de modelos, um arquivo de banco de dados (que será usado para armazenar as conversas entre usuários e o assistente), logs de contêiner e um arquivo do *docker-compose*.

No diretório onde estão os arquivos instalados, bastou executar o comando `sudo docker-compose up -d` para que os containers fossem baixados e o *Rasa X* fosse executado. Uma vez que os containers estejam em funcionamento, é necessário definir a senha do usuário administrador com o seguinte comando `sudo python rasa_x_commands.py create --update admin me <passsword>`.

Com essas configurações, o *Rasa X* fica pronto para ser utilizado, com capacidade de conexão com os assistentes e implementação de melhorias. Para abrir o *Rasa X*, conectar os assistentes e trabalhar nas melhorias deve-se abrir o servidor usando o endereço de IP.

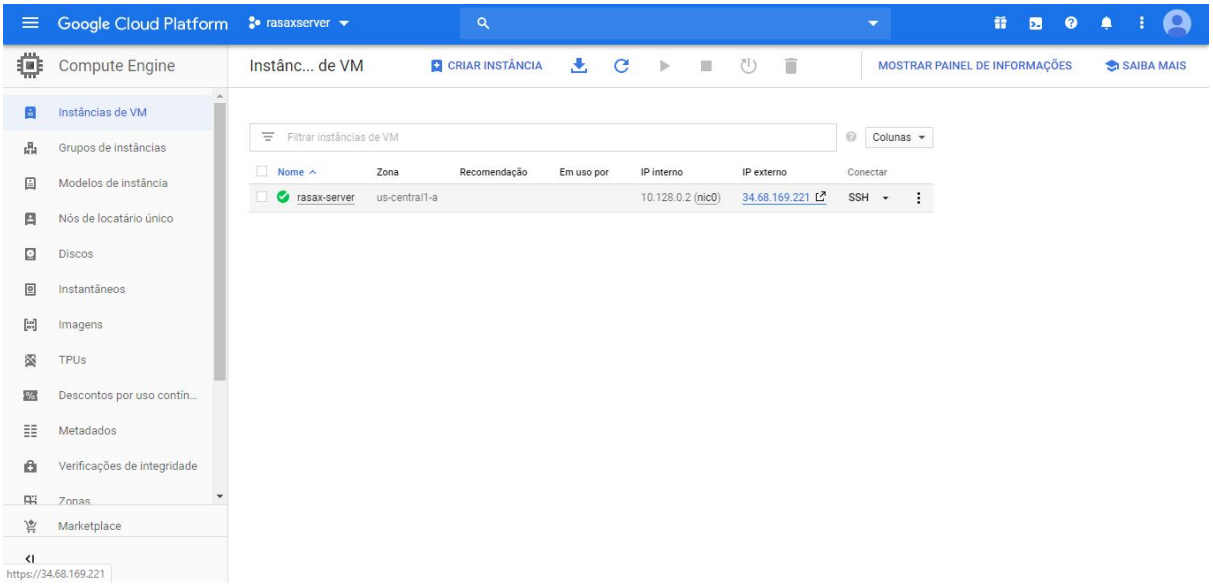

Figura 3.10: Instância da VM criada e seu endereço de IP

Como já comentado, o *Rasa X* possui um controle de versionamento integrado que permite conectar e controlar o assistente que está hospedado no *GitHub* ou qualquer outro servidor remoto. Este controle de versionamento faz com que o *Rasa X* torne muito mais fácil a conexão de assistente existente ao servidor, assim como o trabalho nas melhorias. Os requerimentos para utilizar essa funcionalidade é possuir um projeto *Rasa* com uma estrutura espelhada na estrutura *default* de um projeto criado pelo comando `rasa init` no *Rasa CLI*

Figura 3.11: Estrutura de um projeto padrão Rasa

Uma vez organizado é necessário autenticar o servidor. Primeiro, gerar uma SSL key para o repositório remoto usando o seguinte comando: `ssh-keygen -t rsa -b 4096 -f git-deploy-key`. O comando irá gerar uma chave pública e uma privada. A partir disso, é necessário copiar a chave guardada no arquivo git-deploy-key.pub (que corresponde a uma chave pública) e colar isso na sessões de chaves de *deploy* do projeto no *GitHub*.

Após isso é preciso criar um arquivo repository.json com seguintes parâmetros:

- repository_url: é uma SSH URL para o repositório remoto. Isso pode ser encontrado na página do repositório no *Github*;
- ssh_key é a chave privada que foi gerada (no arquivo git-deploy-key) ;
- target_branch: é o parâmetro que vai apontar o servidor *Rasa X* para essa *branch*, que foi definido como sendo a própria *master*.

Deve-se atualizar todos os detalhes e salvar o arquivo. Uma vez feito, para realizar a autenticação com o Rasa X deve utilizado o seguinte comando:

```
curl --request POST \
     --url https://<Rasa X server
host>/api/projects/default/git_repositories?api_token=<your
api token> \
     --header 'content-type: application/json' \
     --data-binary @repository.json
```

Para executar esse comando será necessário fornecer o Host URL do servidor (que neste caso vai ser o servidor IP) e o *token* da API. O jeito mais fácil de encontrar o token é navegando na página de modelos *Rasa X* e clicando no botão *Upload Model*. Se a autenticação tiver sido bem sucedida, Rasa X vai dispor da versão atual dos arquivos que está no repositório de assistente. Deve-se checar isso abrindo o UI do *Rasa X*, no canto inferior esquerdo, em que o ícone de integração *GitHub* indica que a instância do *Rasa X* está conectada à master do projeto do assistente no *GitHub*.

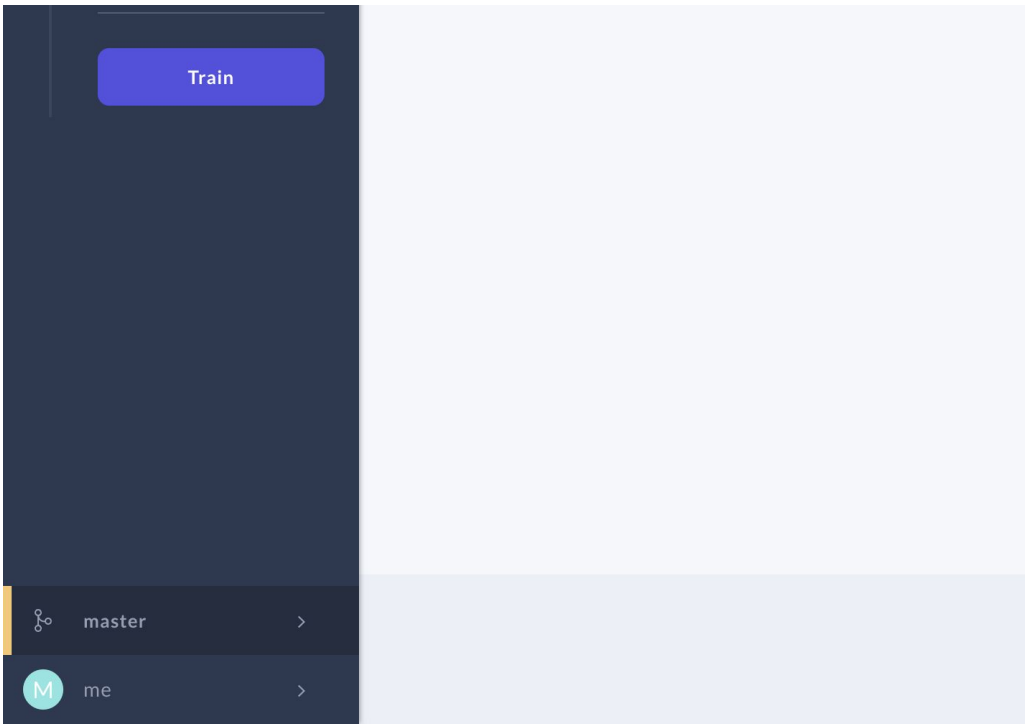

Figura 3.12. UI do Rasa X demonstrando que a integração com o repositório foi bem sucedida

Como relatado na seção anterior, foi necessária a criação de uma ação customizada para a implementação da política *TwoFallbackPolicy*. Essa ação precisa ser executada no servidor. Portanto, no diretório '/etc/rasa/actions' foi substituído o código do arquivo actions.py, pelo código do arquivo de mesmo nome que está presente no projeto no *Github*.

Para poder implantar o assistente a amplo público de usuários conectando-o ao canais externos, existem algumas pendências que precisam ser configuradas em relação ao servidor do *bot*, a partir do momento em que servidor *Rasa X* foi hospedado e possui um endereço IP. É necessário definir o nome de domínio, configurar DNS e obter um certificado SSL para que o servidor possa comunicar com alguns aplicativos da Web, como o *Telegram*.

3.3.2 Criação de domínio e configuração do DNS

Para a criação de domínio foi utilizado o site *Freenom*, que possibilita a criação de domínios gratuitos por um período de 12 meses. Após os 12 meses é possível renovar o registro novamente de forma gratuita, em que só é necessário o pagamento se for reservado por mais de 12 meses na etapa do pagamento [30]. Assim, foi registrado o domínio botufrj.tk.

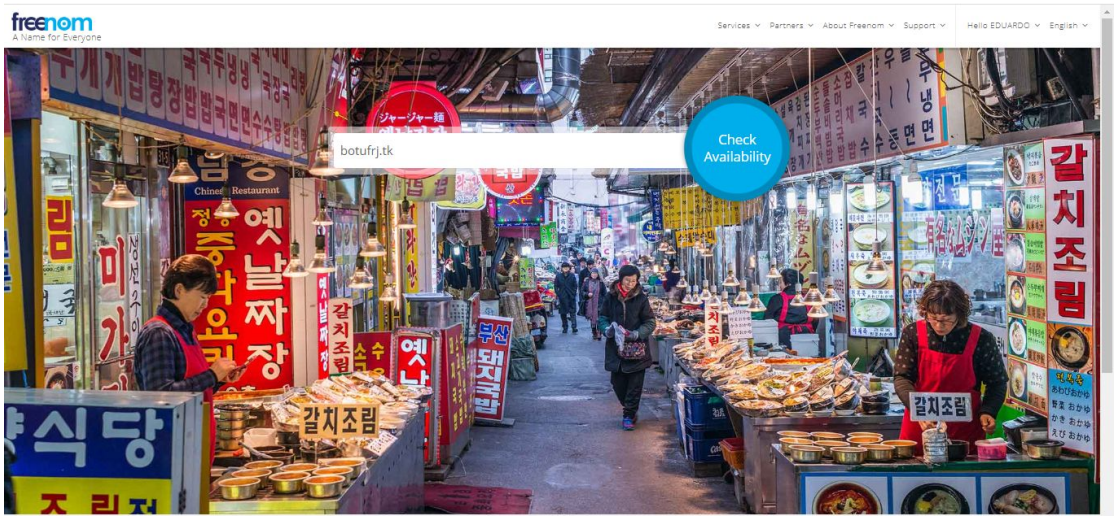

Figura 3.13: Site de registro de domínios gratuitos Freenom

Depois de ter um nome de domínio é necessário configurar o DNS para que o endereço IP do servidor do assistente seja vinculado ao domínio. Para isso, no site *Freenom*, na área do cliente e na lista de domínios registrados, é preciso localizar o domínio criado e clicar em *Manage Domain*.

Domain	Registration Date	Expiry date	Status	Type	
botufrj.tk	2020-02-07	2021-02-07	ACTIVE	Free	Manage Domain

Figura 3.14: Domínio criado no site Freenom

No menu *Management Tools*, foi clicado em *Nameservers* para visualizar as configurações do domínio para apontar para o projeto do *Google Cloud Platform*.

No *Google Cloud Platform*, é necessário acessar no menu lateral do *Cloud Console* a opção Serviços de rede > *Cloud DNS* e lá criar uma nova Zona de DNS. Uma Zona de DNS é o local dentro do projeto onde é cadastrado o domínio registrado e são apontados os endereços de IP das máquinas que responderão às requisições.

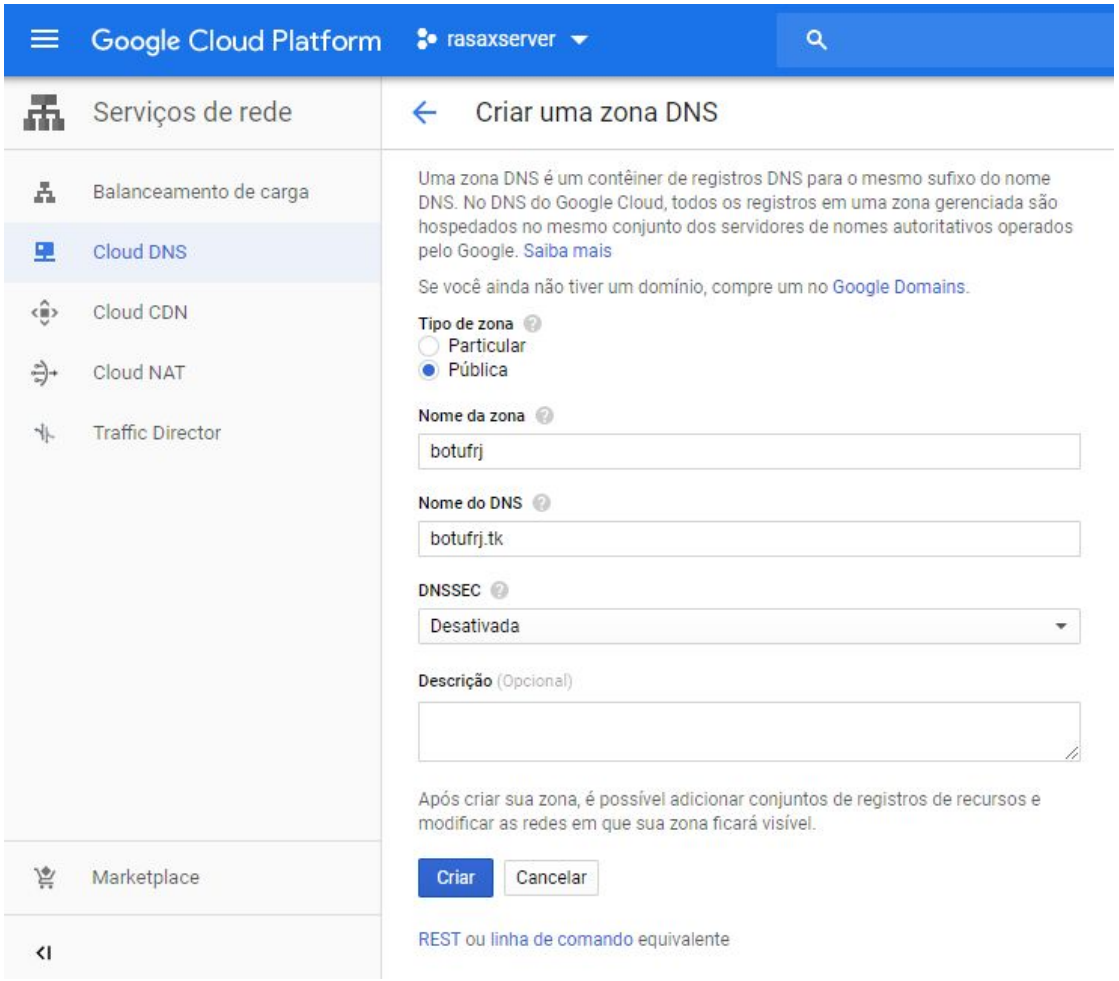

Figura 3.15: Janela de criação de zona DNS do Google Cloud Platform

No momento do cadastro da zona é definido um nome qualquer e no nome do DNS deve ser atribuído o domínio da zona, que neste projeto foi o botufrj.tk. Com a zona criada, basta clicar no nome dela para entrar em sua tela de detalhes e, logo de início, o *Google Cloud Platform* cria dois registros. Um do tipo NS, que é utilizada para configurar a opção de *Nameservers* do *Freenom*, e outro do tipo SOA ou *Start Of Authority*, que indica o responsável por respostas autoritárias a um domínio, ou seja, o responsável pelo domínio. Também indica outras informações úteis como número serial da zona, replicação, etc.

Por fim, foi copiado cada endereço de NS do *Google Cloud Plataform* para a tela de configuração de *Nameservers* do *Freenom*. Após alguns minutos, é possível verificar se tudo foi configurado com sucesso. Ao apontar o domínio no navegador, deve ser possível visualizar a página de login do *Rasa X* para então se configurar o SSL.

3.3.3 Instalação do Certificado SSL

Um certificado SSL cria uma conexão segura entre o navegador e o servidor. Verifica se o nome do *host* solicitado corresponde ao nome registrado no certificado. É possível se adquirir um certificado por meio de uma autoridade de certificação paga ou usar uma opção gratuita, como o *Let's Encrypt*. A *Let's Encrypt* é uma autoridade certificadora gratuita, automatizada e aberta ao público [31].

Foi utilizado o *Certbot*: uma ferramenta gratuita e de código aberto que facilita a instalação de um certificado SSL do *Let's Encrypt*. Inicialmente, deve se parar os containers do docker, que pode ser feito ao executar o comando `docker-compose down`. Após a instalação do *Certbot* (que pode ser feito, no linux Ubuntu, pelo comando `sudo apt-get certbot`), este pode ser executado pelo comando `sudo certbot certonly`.

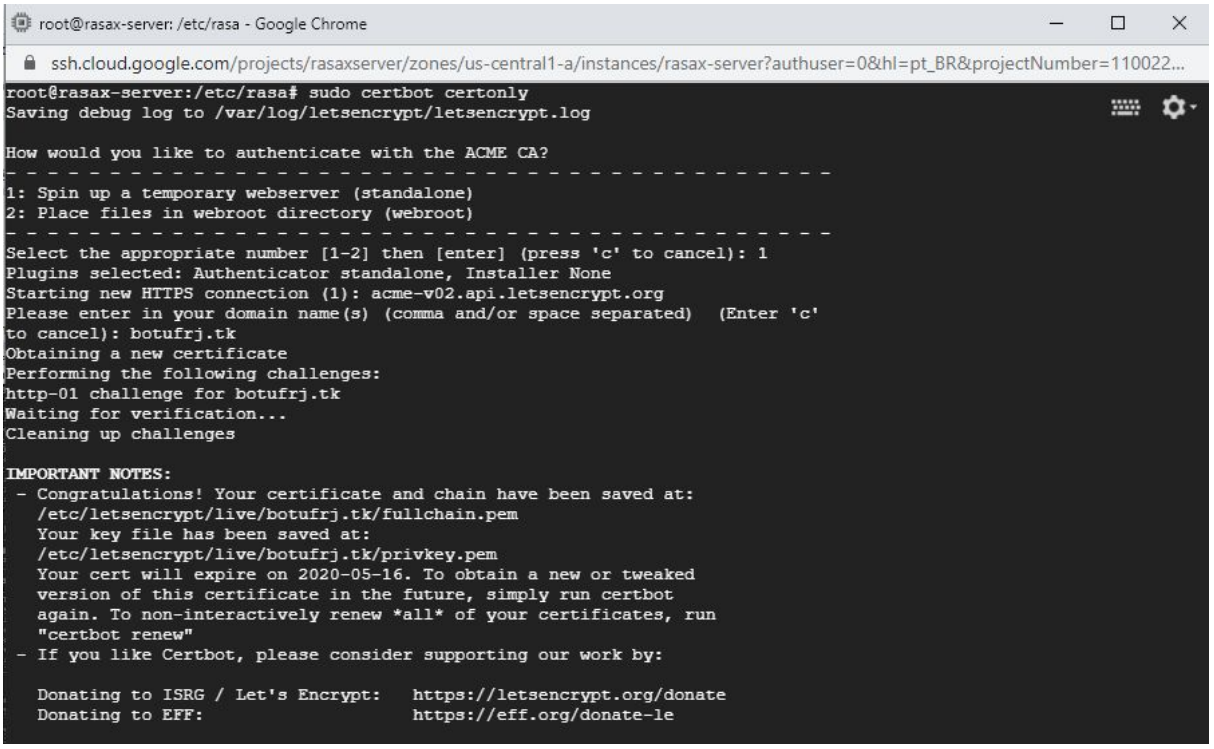

Figura 3.16: Autenticação com o Certbot

Uma vez terminado o processo de autenticação e de aceite dos termos de condições, será exibida uma mensagem que indica que o processo de certificação foi concluído com êxito. Por padrão arquivos de certificado serão armazenados em um diretório de Let's Encrypt, neste caso, os arquivos privkey.pem e fullchaiml.pem no diretório /etc/letsencrypt/live/botufrj.tk . Para torná-los acessíveis ao container *Docker* deve-se mover

os arquivos de certificado para o diretório *Rasa* (para o caso padrão '/etc/rasa/certs'). Certificados criptografados geralmente expiram após 90 dias, mas é possível renová-lo usando o comando `certbot renew`, que pode também ser automatizado usando uma tarefa *Cron*. Com isso, o *Rasa X* fica pronto para conectar o assistente a canais externos e fazer acessível a usuários reais, permitindo amplo público de usuários a conversar com o *bot*.

3.3.4 Integração com o Telegram

O *Telegram* é um aplicativo de mensagens gratuito que pode ser usado em uma variedade de diferentes dispositivos, como tablets, smartphones ou laptops [32]. Suporta aplicações de terceiros como *chatbots*.

Os processos de conexão de um assistente do *Rasa* a diferentes plataformas são semelhantes, neste projeto o foco foi a integração com o *Telegram*. O primeiro passo, foi a configuração da conta do assistente no *Telegram* através do chamado *BotFather*, o gerenciador de *bots* do *Telegram*.

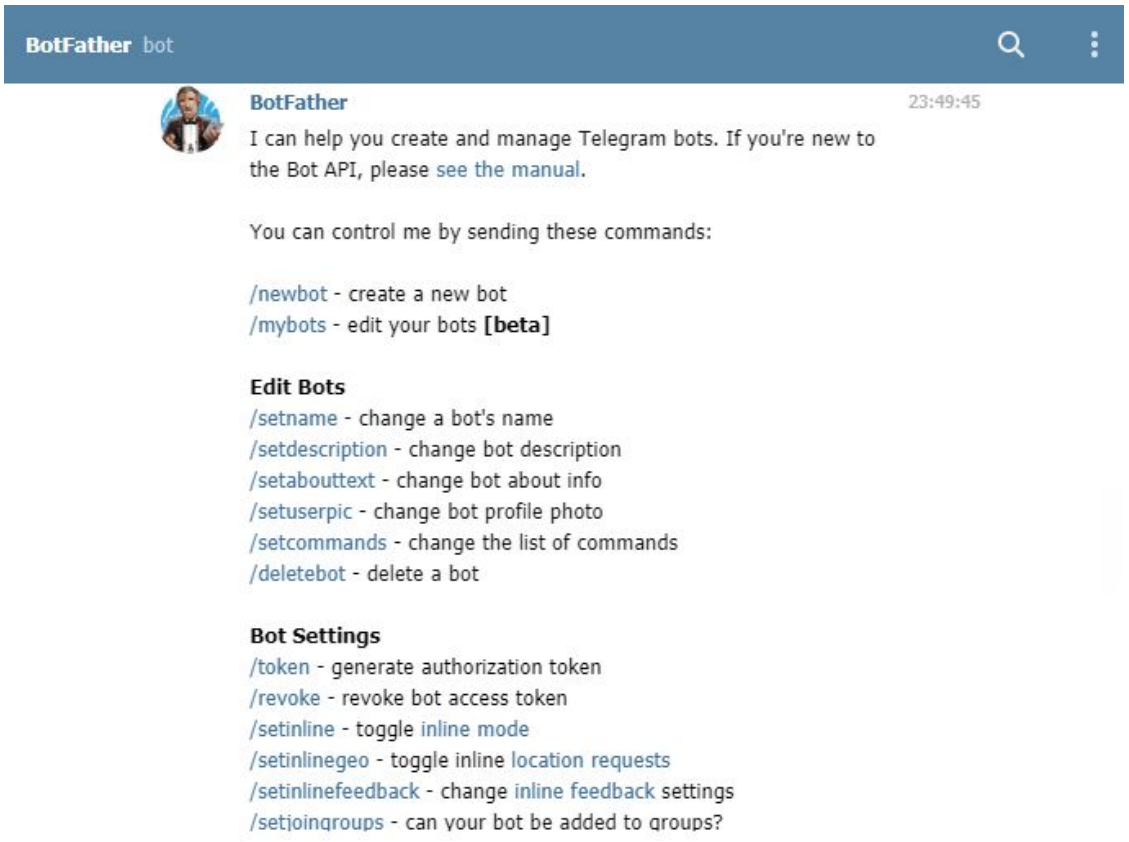

Figura 3.17 - Robô BotFather no aplicativo Telegram

Para criar um novo *bot* nesta plataforma digita-se `/new bot` e será solicitada a criação de um nome de usuário para o assistente. Quando realizado, o *BotFather* irá configurar o assistente no *Telegram* e compartilhar suas credenciais, o nome de usuário e *token* de acesso, que serão usados para conectar o *bot* ao assistente do *Telegram*.

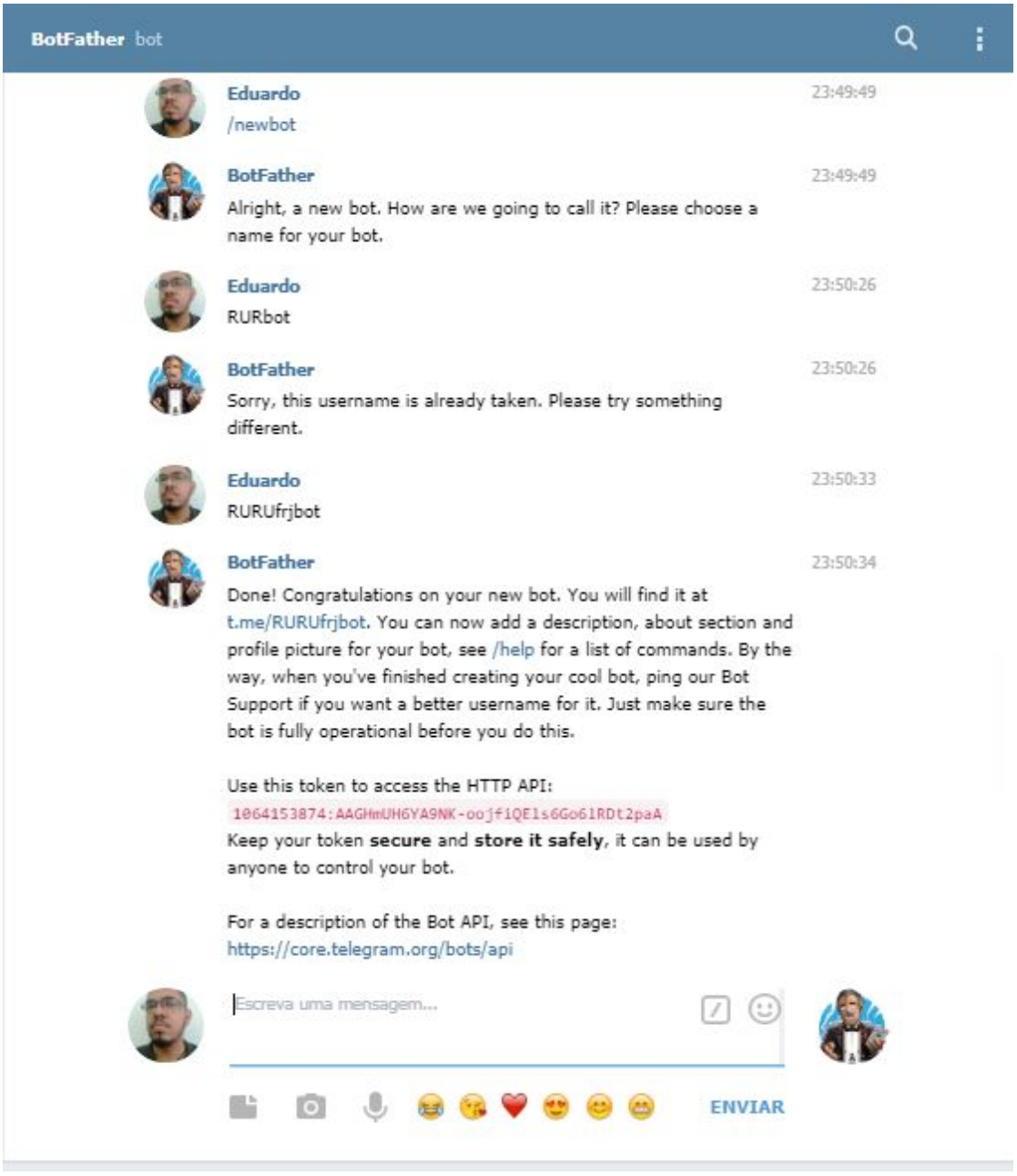

Figura 3.18: Criação de novo bot no Telegram, através do BotFather

O assistente *Rasa* deve conectá-lo ao canal externo e para fazer isso é necessário voltar ao servidor e editar o arquivo chamado ‘credentials.yml’, fornecendo os detalhes de autenticação sugeridos pela documentação do *Rasa* [33].

Para conectar o assistente ao *Telegram*, é necessário fornecer os seguintes parâmetros:

- access token: é o token fornecido pelo *BotFather* na criação do *bot*;
- verify: é o nome do usuário do *bot*;
- webhook_url: é a url onde está sendo executado o webhook no servidor.

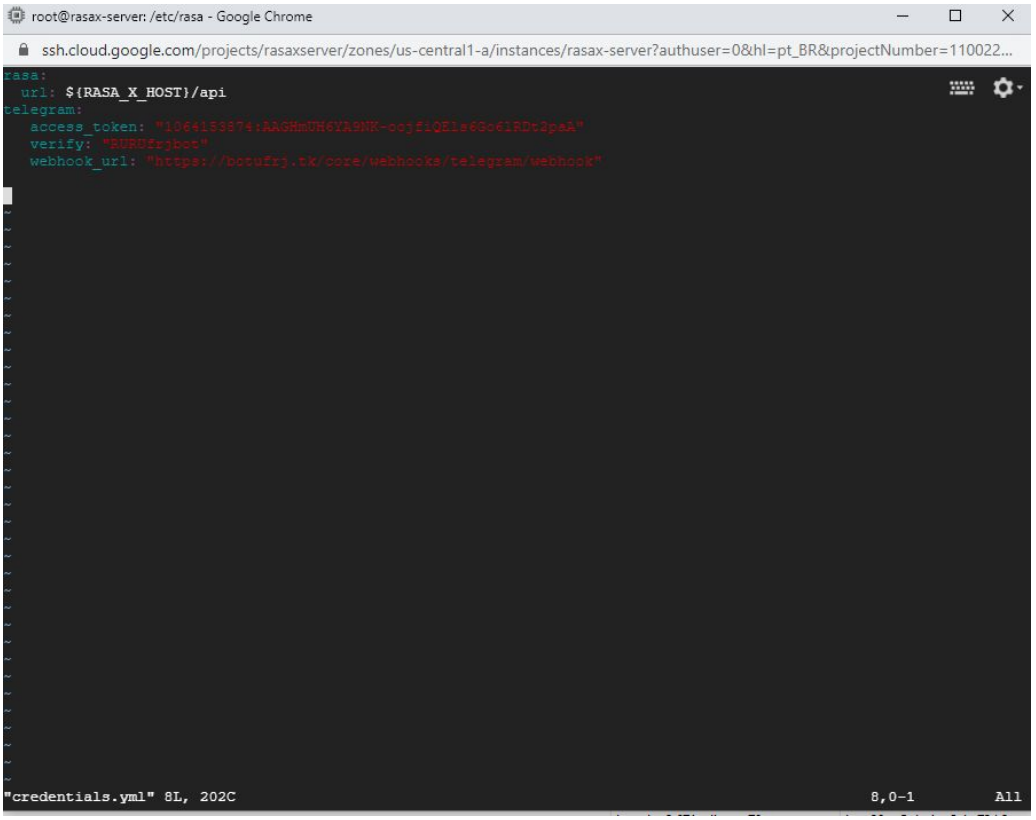

Figura 3.20: Informações necessárias para conectar o assistente ao Telegram editadas no credentials.yml

Feito isso, as alterações foram salvas e é necessário que se reinicie o container do docker. Assim, o assistente foi conectado ao Telegram e todas as conversas que os usuários mantêm com o assistente no Telegram são visíveis na guia Conversations do Rasa X.

Capítulo 4

4 - Considerações finais

4.1 Conclusão

Este trabalho buscou explorar ferramentas para a construção de *chatbots* e o desenvolvimento efetivo de um *bot* de diálogo para o *Telegram*.

Na seção 1.4 foi definida como uma das metas a análise e pesquisa por diferentes plataformas para a construção de *chatbots*, para em seguida utilizar a mais apropriada para construção de um *bot* para as finalidades deste projeto. A pesquisa foi feita e houve o entendimento do *framework Rasa* ser a escolha ideal para o projeto, por fornecer uma gama de funcionalidades já implementadas, capacidade de personalização e boa documentação. Com o lançamento ainda do *Rasa X*, tornou-se viável a utilização de um conjunto de ferramentas, por meio de uma plataforma Web, que possibilita ir revisando e fazendo anotações nas conversas realizadas, melhorar continuamente o assistente e ainda ter tudo isso integrado a um controlador de versão.

O desenvolvimento do projeto foi iniciado em março de 2019, onde uma estrutura inicial foi construída. Por volta de maio do mesmo ano, o *Rasa* teve uma grande mudança estrutural com o lançamento da sua versão 1.0, inclusive uma grande mudança na documentação, com novas funcionalidades e algumas configurações que passaram a ter outros nomes nessa nova versão. O que fez com que o projeto necessitasse ser atualizado, com receio da dificuldade de manutenção no futuro. Na mesma época surgiu o *Rasa X*, que possibilitaria a manutenção do robô por meio de uma interface gráfica, sem necessitar mexer no servidor a todo o momento para fazer uma simples alteração. Que também foi entendida como uma nova funcionalidade muito poderosa que deveria ser acoplada ao projeto.

No entanto, essas alterações foram muito custosas em termos de desenvolvimento. Sem contar que, como são novas versões, não havia muita discussão na Web sobre possíveis problemas que poderiam surgir. O *Rasa X* está, inclusive, até o presente momento (março de 2020), em versão de testes (v0.25.1). Logo, este ainda possui uma série de falhas que estão sendo corrigidas e atualizações a todo o tempo, o que também se tornou um empecilho

durante a implementação. Hoje, por exemplo, o *Rasa X* não exibe as mensagens de erro na interface, o que provavelmente no futuro, numa versão mais estável, isso seja corrigido. É possível visualizar os erros no log do servidor, mas isso ainda não é o ideal.

As melhorias que foram sendo feitas na suíte *Rasa* se tornaram importantes para o sucesso dessa implementação. Contudo, poderia ser obtido um resultado mais interessante e imediato se o *framework* estivesse numa versão mais estável no momento em que esse projeto foi iniciado.

Mesmo com alguns contratempos, ao final, foi obtido um bom resultado. Foi construído um novo *chatbot* que consegue ser muito mais customizável do que o anterior, não há somente o cadastro de uma pergunta e de uma respectiva resposta, mas há agora o reconhecimento da intenção da frase do usuário para que então seja decidida qual a próxima ação do *bot*. Com o *Rasa* há a possibilidade de desenho de possíveis caminhos que a conversa possa seguir através das histórias. Onde, por exemplo, pôde ser configurado que uma frase com intenção de saudação seja respondida com uma saudação, e que uma frase com intenção de despedida seja respondida com uma frase de despedida, porém quando uma frase de saudação seja seguida de uma frase de despedida, o *bot* execute uma outra ação, perguntando porque o usuário já está indo embora.

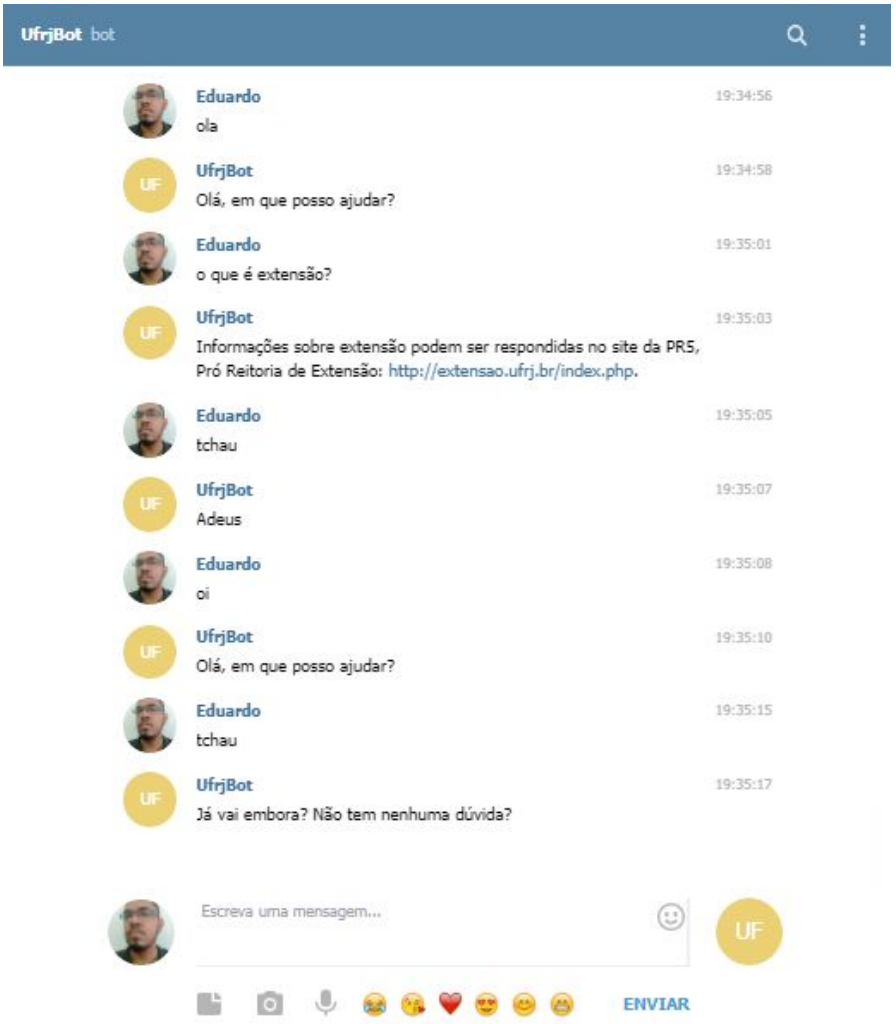

Figura 4.1: Exemplo do uso de histórias no chatbot

Também se tornou viável a utilização de ações customizadas, em que é possível via código, na linguagem Python, programar como o robô pode formar a sua resposta ou mesmo chamar alguma API externa. É possível definir e personalizar políticas para a obtenção das respostas, em que a partir da política de *TwoFallbackPolicy*, por exemplo, foi possível fazer com que quando o assistente não consiga realizar uma previsão precisa, ele não deixe de dar uma resposta adequada, nem mesmo forneça uma resposta incorreta, além de fomentar um diálogo com o usuário.

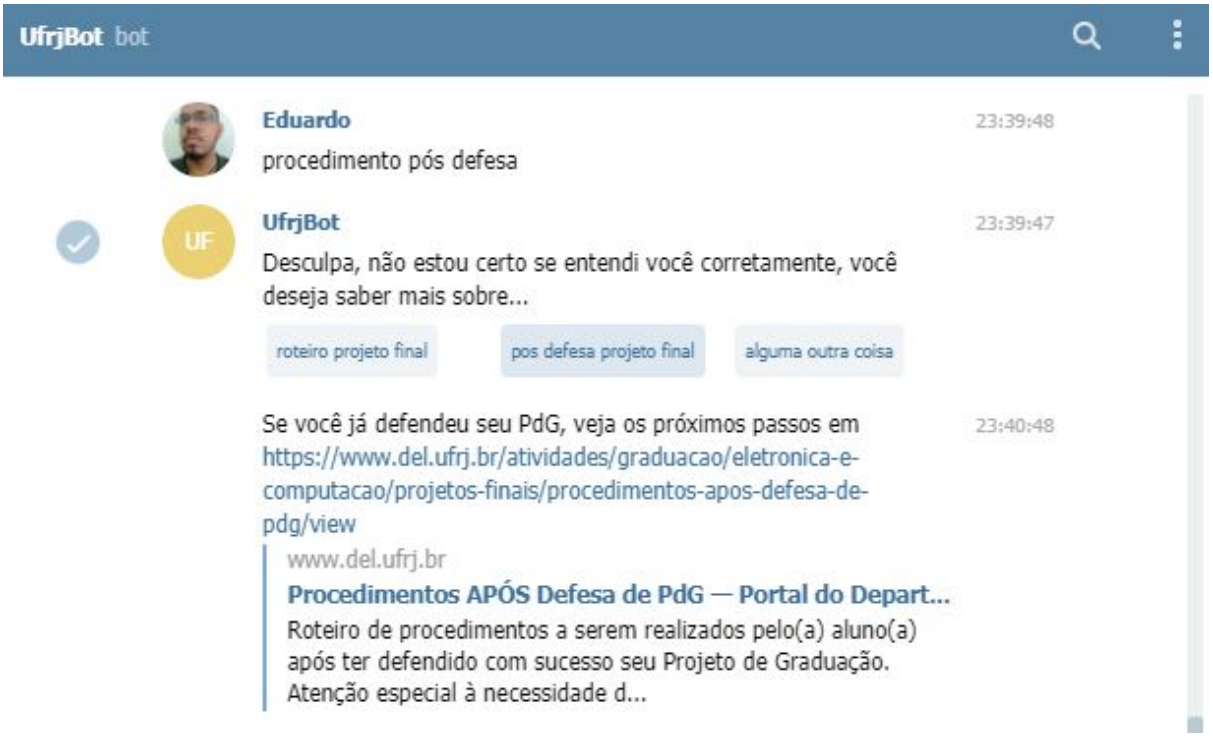

Figura 4.2: Exemplo de conversa com a ação do TwoFallbakPolicy, após o usuário clicar na opção ‘pos defesa projeto final’

Há ainda um ambiente online em que se pode atualizar continuamente o assistente e ensiná-lo como agir em determinadas circunstâncias. Com o Rasa X se tornou mais facilitada a realização tanto de ajustes em configurações e nos dados de treinamento, quanto em revisar conversas anteriores, verificar como o robô foi entendendo as frases e que ações foram desencadeadas, analisar as histórias e fazer anotações, adicionando novas regras ou corrigindo previsões que não estavam corretas.

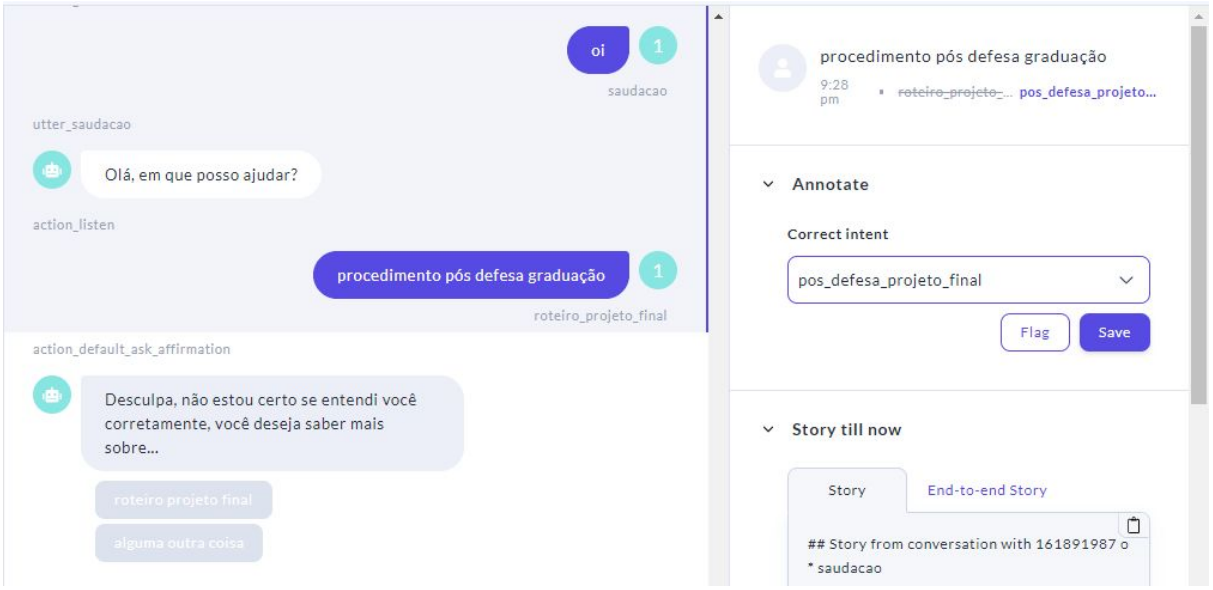

Figura 4.3: Correção da intenção prevista na conversa por meio do Rasa X

Há ainda uma série de recursos que ainda não foram utilizados, como a extração de entidades numa frase de entrada, ou o armazenamento de informações por meio do uso de *slots*, ou o uso de formulários durante a conversa, ou mesmo a criação de histórias mais complexas, com diferentes caminhos a serem seguidos, dentre outros. Ainda não foi pensado em como ou se todos esses recursos podem ser utilizados neste projeto, até por que esses recursos não estavam disponíveis anteriormente. Dado que novas possibilidades de customização estejam disponíveis, novas ideias podem surgir e serem implementadas com o passar do tempo.

4.2 Trabalhos futuros

4.2.1 Migrar chatbot e servidor

Nesse projeto foi criado um novo *bot* no *Telegram*, a partir de dados de um inicial já existente, hospedado e construído na plataforma *Bot Libre*. A ideia é que o *bot* criado neste trabalho passe a ser o novo *chatbot* em funcionamento num futuro próximo, e para isso seria necessária a substituição das credenciais no servidor do *Rasa X*, apontando para o *chatbot* anterior, modificando para o respectivo nome e o *token* do novo *bot*.

Além disso, o *Rasa X* está atualmente hospedado no *Google Cloud Platform*, uma ferramenta paga, que está sendo utilizada em sua avaliação gratuita, que vale por 12 meses

com crédito de US$ 300,00 [34]. Pelo custo da máquina virtual para os testes (US$55,00 / mês), o robô se manteria no ar apenas por 5 meses. Então a migração do servidor e do domínio para as dependências da UFRJ seria o ideal.

4.2.2 Acompanhar atualizações do framework

O *framework* vem sendo atualizado constantemente. Como comentado, o *Rasa X*, por exemplo, ainda está em versão beta. Logo, acompanhar as atualizações, pelo menos até uma versão estável, seria útil não apenas para que possíveis erros existentes sejam corrigidos, mas também em obter melhorias nas implementações que já existem e ter disponíveis possíveis novas funcionalidades.

Parte dos recursos usados neste projeto foram adicionados nos últimos meses, como por exemplo, a integração do *Rasa X* com o *Git*, que surge em sua versão 0.23.0 [35]. O controle de versão integrado executa uma sincronização bidirecional para verificar as diferenças entre os dados de treinamento no *Rasa X* e o repositório remoto no servidor *Git*. É um avanço interessante na questão de armazenamento adequado das configurações, com a possibilidade de reverter as alterações e fornecendo transparência à maneira como as atualizações influenciam no comportamento do modelo. Permite que as equipes construam assistentes de IA de forma colaborativa, realizando alterações em uma ou mais *branches* de desenvolvimento. Isso ainda pode facilitar o *Rasa X* na integração com fluxos de trabalho de entrega contínua. Em que o *Rasa X* pode se tornar o primeiro ponto de contato em um pipeline de implantação automatizado, usando ferramentas como *Jenkins*, *CircleCI* e *Travis*.

As novas atualizações que se seguem podem ser simples, desde o lançamento de um novo componente, uma nova política, correções de bugs, até uma atualização de grande impacto como a anterior. A equipe de desenvolvedores do *Rasa* tem esse viés de cada vez fornecer mais ferramentas que facilitam a construção de um *bot*, que seja possível revisar conversas reais e convertê-las em dados de treinamento, além de elevar o desenvolvimento de um assistente virtual a um outro nível. Utilizar *Docker* para o *deploy* no servidor, facilitar a integração com controle de versão, dentre outras características, fazem com que o *Rasa X* funcione como as ferramentas e junto aos fluxos de trabalho já existentes dos desenvolvedores, em que estes podem aplicar as mesmas práticas recomendadas e ferramentas que eles já utilizam em seus projetos de software no dia-a-dia também para

gerenciar os seus *chatbots* e suas atualizações de dados de treinamento. Como resultado, as equipes de software podem versionar e melhorar os assistentes usando processos escaláveis e repetíveis.

4.2.3 Ampliação dos dados de treinamento

Por fim, é importante manter as informações do *chatbot* atualizadas e acompanhar a forma como o assistente está interagindo com os usuários. Desde a verificação se as conversas seguem as histórias como o planejado, se o *bot* está conseguindo reconhecer as intenções das frases como deveria e o tempo que ele está levando para fornecer as respostas, até a percepção de perguntas que sejam frequentes e que o *bot* ainda não saiba como responder.

A ideia é realmente compartilhar o assistente com testadores reais, coletar as conversas que eles tem com o assistente e usar isso para fazer melhorias. De modo que ao final, talvez precisem ser feitas algumas mudanças, maiores ou menores, como correções das intenções previstas ou mesmo arquiteturais, na forma em que as previsões são feitas e as respostas são fornecidas. Realizar esse processo continuamente vai ajudar a localizar pontos de falha e a fazer as melhorias necessárias para assegurar que novos usuários tenham uma boa experiência quando conversarem com o assistente.

Referências Bibliográficas

[1] Maciel, Herison (2019). Ferramentas e criação de chatbot – Maciel o robô acadêmico. Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Russas, Curso de Engenharia de Software, Russas, p. 15-16.

[2] WANTROBA, E. J.; RATUSZNEI, J.; SOUZA, L. de; VENSKE, S. M. G. S. Um exemplo de uso do padrão xml na definição de uma linguagem especializada para a inteligência artificial. Publicatio UEPG: Ciências Exatas e da Terra, Agrárias e Engenharias. v. 14, n. 03, 2008.

[3] Preeti dan BrahmaleenKaurSidhu, "Natural language processing," Int.J.Computer Technology & Applications, vol. 4, no. 5, hal. 751–758, 2013.

[4] Diksha Khurana, Aditya Koli , Kiran Khatter and Sukhdev Singh."Natural Language Processing: State of The Art, Current Trends and Challenges". CoRR abs/1708.05148, 2017.

[5] ASPECT SOFTWARE, 2007. "Customer Service Chatbots and Natural Language", disponível em: <https://www.aspect.com/globalassets/microsite/nlu-lab/images/Customer-Service-Chatbots-and-Natural-Language-WP.pdf >, acesso em: 10 nov. 2019.

[6] Vogel, J. "Chatbots: Development and Applications". Tese (Graduação em International Media and Computing) - HTW Berlin, University of Applied Sciences. Berlin, p. 3-20, 2017.

[7] Mislēvičs, A., Grundspeņķis, J., Rollande, R. A Systematic Approach to Implementing Chatbots in Organizations – RTU Leo Showcase. Em: Joint Proceedings of the BIR 2018 Short Papers, Workshops and Doctoral Consortium co-located with 17th International Conference Perspectives in Business Informatics Research (BIR 2018). CEUR Workshop Proceedings. Vol.2218, Sweden, Stockholm, 24-26 Setembro, 2018. Aachen: RWTH, 2018, pp.356-365. ISSN 1613-0073.

[8] Abdul-Kader, SA and Woods, JC (2015) 'Survey on Chatbot Design Techniques in Speech Conversation Systems.' International Journal of Advanced Computer Science and Applications, 6 (7). ISSN 2156-5570

[9] Dahiya, M. "A tool of conversation: chatbot". International Journal of Computer Sciences and Engineering 5(5), p. 158-161, 2017.

[10] Kiptonui, Bii. (2013). Chatbot technology: A possible means of unlocking student potential to learn how to learn. Educational Research. Vol 4. 218-221.

[11] Rahman, Johan. ”Implementation of ALICE chatbot as domain specific knowledge bot for BRAC U (FAQ bot),” Thesis Paper, BRAC University.

[12] “Dicas Matadoras para Criar Chatbots Bons de Conversa”, disponível em: <https://www.tiagotessmann.com.br/dicas-matadoras-para-criar-chatbots-bons-de-conversa/> , acesso em: 7 fev. 2020.

[13] "Bot Libre", disponível em: <https://www.botlibre.com>, acesso em: 15 nov. 2019.

[14] "IBM Watson", disponível em: <https://www.ibm.com/watson/br-pt/>, acesso em: 16 nov. 2019.

[15] "About ChatterBot", disponível em: <https://chatterbot.readthedocs.io>, acesso em: 16 nov. 2019.

[16] "Getting Started with Rasa", disponível em: <http://rasa.com/docs/getting-started/>, acesso em: 20 nov. 2019.

[17] "Algorithms alone won’t solve conversational AI — Introducing Rasa X", disponível em:<https://medium.com/rasa-blog/algorithms-alone-wont-solve-conversational-ai-introducing-rasa-x-b2767d1964de>, acesso em: 22 nov. 2019.

[18] “Rasa Blog”, disponível em: <https://blog.rasa.com/>, acesso em: 2 fev. 2020.

[19] “Rasa Docs”, disponível em: <https://rasa.com/docs/>, acesso em: 2 fev. 2020.

[20] “The Rasa Masterclass Handbook: Episode 4”, disponível em <https://blog.rasa.com/the-rasa-masterclass-handbook-episode-4/>, acesso em 18 fev, 2020.

[21] “Stories”, disponível em <https://rasa.com/docs/rasa/core/stories/>, acesso em 20 Jan, 2020.

[22] “The Rasa Masterclass Handbook: Episode 6”, disponível em <https://blog.rasa.com/the-rasa-masterclass-handbook-episode-6-2/>, acesso em 21 Jan, 2020.

[23] “Rasa X: Getting started as a current Rasa user”, disponível em <https://blog.rasa.com/rasa-x-getting-started-as-a-current-rasa-user/>, acesso em 10 Dez, 2019.

[24] “Installation”, disponível em <https://rasa.com/docs/rasa/user-guide/installation/>, acesso em 18 fev, 2020.

[25] “Anaconda Distribution”, disponível em <https://www.anaconda.com/distribution/>, acesso em 18 fev, 2020.

[26] “Training & Chat Logs”, disponível em <https://www.botlibre.com/manual-chatlogs.jsp>, acesso em 15 Fev, 2020.

[27] “Training Data Importers”, disponível em <https://rasa.com/docs/rasa/api/training-data-importers/>, acesso em 18 fev, 2020.

[28] “Choosing a Pipeline”, disponível em <https://rasa.com/docs/rasa/nlu/choosing-a-pipeline/>, acesso em 12 Dez, 2019.

[29] “Overview of Docker Compose”, disponível em <https://docs.docker.com/compose/>, acesso em: 10 fev, 2020.

[30] “Domínios gratuitos e pagos”, disponível em <https://www.freenom.com/pt/freeandpaiddomains.html>, acesso em 21 Dez, 2019.

[31] “Sobre a Let's Encrypt”, disponível em <https://letsencrypt.org/pt-br/about/>, acesso em: 14 fev, 2020.

[32] “Telegram FAQ”, disponível em <https://telegram.org/faq>, acesso em: 14 fev, 2020.

[33] “Telegram”, disponível em <https://rasa.com/docs/rasa/user-guide/connectors/telegram/>, acesso em: 14 fev, 2020.

[34] “Nível gratuito do Google Cloud”, disponível em <https://cloud.google.com/free/docs/gcp-free-tier?hl=pt-br>, acesso em 18 Dez, 2019.

[35] “Integrated Version Control: Linking Rasa X with Git-based Development Workflows”, disponível em <https://blog.rasa.com/integrated-version-control-linking-rasa-x-with-git-based-development-workflows-untitled/>, acesso em 22 Jan, 2020.