UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Engenharia Elétrica

JULIO OLIVEIRA DE ANDRADE ARAUJO

AUTOMAÇÃO ATRAVÉS DE SOFTWARE CONVERSACIONAL: UM ESTUDO DE CASO EM DOMÓTICA

UBERLÂNDIA (MG)
2020

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Engenharia Elétrica

AUTOMAÇÃO ATRAVÉS DE SOFTWARE CONVERSACIONAL: UM ESTUDO DE CASO EM DOMÓTICA

Trabalho de Conclusão de Curso de Engenharia de Controle e Automação da Universidade Federal de Uberlândia - UFU - Campus Santa Mônica, como requisito para a obtenção do título de Graduação em Engenharia de Controle e Automação.
Orientador: Prof. Dr. Marcelo Barros.

UBERLÂNDIA (MG)
2020

Araújo, Julio

Automação através de software conversacional: um estudo de caso em domótica/ **Julio Oliveira de Andrade Araújo**. – **UBERLÂNDIA**, **2020**- 28 p.: il. (algumas color.); 30 cm.

Orientador: Prof. Dr. Marcelo Barros

Trabalho de Conclusão de Curso – Universidade Federal de Uberlândia – UFU Faculdade de Engenharia Elétrica. **2020**.
Inclui bibliografia.

1.Software conversacional 1.2. Chatbot. Orientador. Prof. Dr. Marcelo Barros

Universidade Federal de Uberlândia Faculdade de Engenharia Elétrica. Engenharia de Controle e Automação.

Dedico este trabalho aos meus pais,
pelo apoio e pela compreensão.

Agradecimentos

Agradeço a minha família por compreender os obstáculos percorridos até aqui, bem como sua compreensão e seu apoio nas decisões difíceis.

Aos meus amigos, que sempre me ofereceram ajuda e me deram dicas para buscar a melhor opção no desenvolvimento deste trabalho.

Ao professor Dr. Marcelo Barros, por todo o suporte ao longo dessa caminhada, sempre disposto a ouvir e a aconselhar.

Meus sinceros agradecimentos a todas essas pessoas.

“Por vezes sentimos que aquilo que fazemos não é senão uma gota no oceano. Mas sem ela o oceano seria menor” (Madre Teresa de Calcutá).

Resumo

Este trabalho apresenta o estudo e implementação de uma solução personalizável de assistente virtual, ou *Chatbot*, contemplando a sua integração a uma conversa privada no aplicativo de mensagens Telegram além da sua disponibilização em um serviço em nuvem, a fim de que o usuário possa solicitar informações e executar tarefas em um *hardware* que simula uma residência. Assim, o usuário, desde que tenha acesso a internet, controlará os elementos da sua casa a distância.

O desenvolvimento deste trabalho é divido em 4 partes: Criação e treinamento do modelo de *Machine Learning* (*Chatbot*), disponibilização do modelo treinado em um serviço de computação em nuvem da Microsoft Azure, construção de um sistema embarcado e criação de um *bot* no Telegram. Ambas as 4 partes trabalham em sincronia e comunicando entre si.

Por fim, com o intuito de testar o desempenho do produto desenvolvido neste trabalho, são apresentados resultados adquiridos da sua implementação em situações do cotidiano.

O projeto foi desenvolvido levando em consideração critérios de segurança da informação e utilizando *softwares* e módulos *open source*, para a obtenção de um produto com bom custo-benefício.

Palavras-chave: *Chatbot*, *Machine Learning*, *Computação em nuvem, Sistemas Embarcados*.

Abstract

This work presents the study and implementation of a customizable virtual assistant solution, or Chatbot, contemplating its integration with a private conversation in the Telegram messaging application in addition to its availability in a cloud service, so that the user can request information and perform tasks on hardware that simulates a home. Thus, the user, as long as he has access to the internet, will control the elements of his home remotely.

The development of this work is divided into 4 parts: Creation and training of the Machine Learning model (Chatbot), availability of the model trained in a Microsoft Azure cloud computing service, construction of an embedded system and creation of a bot in Telegram. Both 4 parts work in sync and communicate with each other.

Finally, in order to test the performance of the product developed in this work, results obtained from its implementation in everyday situations are presented.

The project was developed taking into account information security criteria and using open source software and modules, to obtain a cost-effective product.

Keywords: Chatbot, Machine Learning, Cloud Computing, Embedded Systems.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Interchange
CPU	Central Process Unit
GB	Gigabyte
GPIO	General Purpose Input/Output
HDMI	High-Definition Multimedia Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
Hz	Hertz - Unidade de medida de frequência
IA	Inteligência Artificial
JSON	JavaScript Object Notation
LED	Light Emitting Diode
OS	Operating System
p	Pixel
PWM	Pulse Width Modulation
RAM	Random Access Memory
URL	Uniform Resource Locator
USB	Universal Serial Bus
TCP	Transmission Control Protocol

Sumário

1. Introdução

Inteligência Artificial é um campo da Computação que estuda a síntese e a análise de agentes computacionais que agem de forma inteligente (POOLE; MACKWORTH, 2010). Poole e Mackworth (2010) definem os objetivos centrais de pesquisa da Inteligência Artificial como sendo: analisar agentes naturais e agentes artificiais; formular e testar hipóteses sobre o que é necessário para a criação de agentes inteligentes; e projetar, estudar e fazer experimentos com sistemas computacionais que executam tarefas que requerem inteligência.

Segundo Manyika (2018), o futuro da automação está diretamente interligado à inteligência artificial, já observando diversos(as) movimentos e ações que demonstram essa tendência. Hoje, a partir da inteligência artificial, a máquina tem condições para tomar decisões autônomas durante o processo produtivo, diferentemente de um robô, que trabalha somente a partir de parâmetros estabelecidos. Com esse recurso, os equipamentos, por meio de sensores e de dados, autoprograma-se de acordo com as mudanças das condições ambientais ou da ocorrência de defeitos e de problemas.

Este trabalho propõe a criação de um *Chatbot* para auxiliar o controle e supervisão de uma residência.

Levando-se em consideração que as assistentes virtuais simulam o comportamento humano de forma autônoma, será verificado se é possível a criação de um sistema que responda as principais perguntas de supervisão e acionamento dos elementos de uma residência. As perguntas devem ser respondidas de modo que quem esteja utilizando o sistema consiga a resposta desejada e tenha a sensação de que estas respostas sejam oriundas de um operador humano.

1.1 Justificativa

Diante do crescimento da inteligência artificial e de novas tecnologias, em busca de melhorar e de facilitar os processos, tanto na indústria quanto em atividades simples do dia a dia, o intuito do presente trabalho é desenvolver um *software* de inteligência artificial, em que, através de uma conversa contextualizada

no aplicativo Telegram, o usuário consiga ter um diálogo amigável com um “robô”, que interage como um ser humano e executa ações de controle em uma residência.

Os sistemas de conversação artificial estão se tornando uma parte indispensável do ecossistema humano. Exemplos bem conhecidos de IA de conversação incluem o Siri da *Apple*, o Alexa da *Amazon*, a Cortana da *Microsoft*, a Bia do Bradesco e a Ana do Itaú.

O *software* conversacional, ou *Chatbot*, é um algoritmo de *machine learning* treinado para interagir dinamicamente com o usuário e efetuar as ações solicitadas por ele.

Assim, com a elaboração do protótipo, o usuário conseguirá acessar dados sobre a sua residência em tempo real e realizar ações nela, mesmo não estando em sua casa.

1.2 Objetivo

O objetivo deste trabalho é apresentar um *software*, moderno e de baixo custo, que ajude a resolver necessidades oriundas de uma residência. Utilizando uma *Raspberry Pi* 3B+, uma *protoboard*, 2 LEDs, 2 resistores, um botão e um *software* de *machine learning open source* serão explicados toda a estrutura e o desenvolvimento de um sistema, cujo usuário consiga controlar componentes de sua casa, através de uma conversa amigável e contextualizada no Telegram.

2. Referencial Teórico

O desenvolvimento do projeto é dividido em 3 partes: Desenvolvimento do software conversacional, disponibilização do projeto em forma de serviço com o objetivo de deixá-lo acessível a qualquer momento e o sistema embarcado que será responsável por controlar os componentes e simular uma residência.

O objetivo desse capítulo é apresentar e fundamentar os conceitos que foram utilizados no desenvolvimento do projeto proposto.

2.1 Machine Learning

Para desenvolver o *chatbot* foi utilizado a ferramenta *Rasa Stack* (que será abordada no próximo capítulo). Essa ferramenta abstrai em uma camada mais externa conceitos e fundamentos de *Machine Learning* para que desenvolvedores possam criar suas assistentes virtuais de uma forma mais otimizada. As técnicas e conceitos escolhidas para o projeto serão abordados a seguir.

A aprendizagem é uma característica própria dos seres humanos. Graças a isso, enquanto executam tarefas semelhantes, adquirem a capacidade de melhorar seu desempenho. Essa habilidade, quando aplicada a sistemas computacionais, é chamada de aprendizado de máquina (KONAR, 1999, p 788).

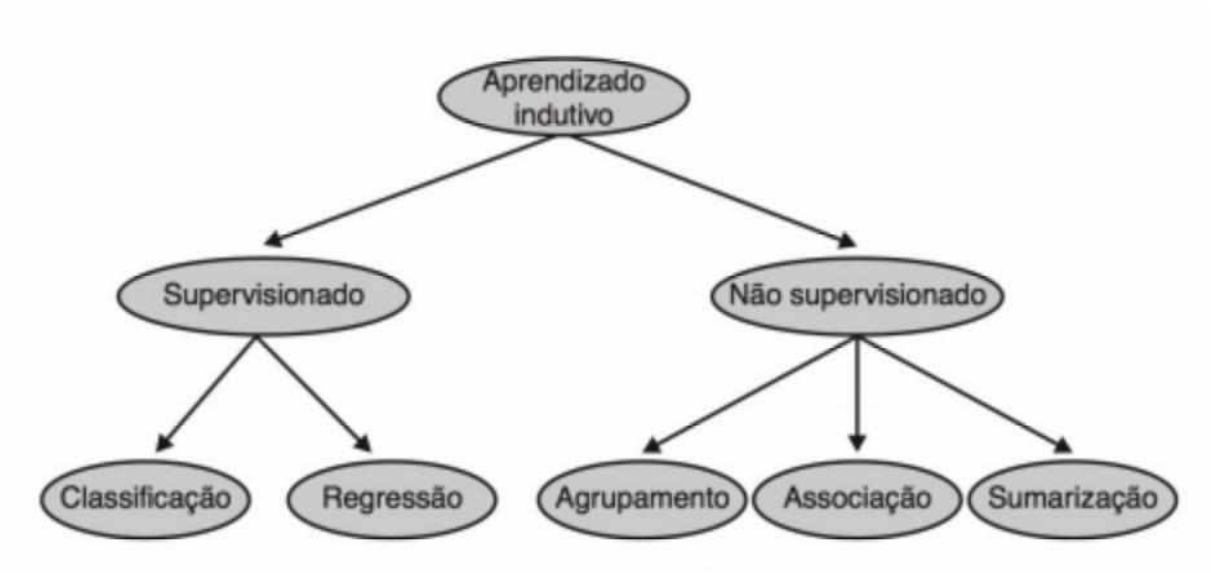
Sistemas de *Machine Learning* ou aprendizado de máquina são sistemas quem aprendem a partir dos dados e que pretende tomar decisão com o mínimo de intervenção humana, uma opção muito interessante e que na última década se espalhou rapidamente por todas as áreas de conhecimento. *Machine Learning* é usado em pesquisas na *web*, filtro de spam de e-mails, sistemas de recomendação, anúncios, detecção de fraude, classificação de imagens e muitas outras aplicações.

Para entender como o aprendizado de máquina gera conhecimento e aprende padrões e dados, precisamos entender sobre a hierarquia do aprendizado de sistema, que seria o processo de indução, a forma de inferir a lógica para obter conclusões genéricas sobre um conjunto particular de exemplos. A indução é o recurso mais utilizado pelo cérebro humano para derivar conhecimento novo. (MONARD; BARANAUSK, 2003) Para algum conceito ser aprendido na indução, gera-se várias hipóteses de conhecimento nos exemplos analisados, e essas hipóteses geradas podem ou não ser algo verdadeiro. (CARVALHO et al., 2011) O

processo de indução é algo que se aplicado com o número baixo de dados, as hipóteses obtidas podem ser de pouco valor. Por isso, o aprendizado de máquina precisa ter grandes volumes de amostras para conseguir aprender e retirar algo relevante para o problema que está sendo resolvido. E as variáveis nesse conjunto de dados, tragam valor e gerem o maior número de hipóteses para o algoritmo aprender, (CARVALHO et al., 2011).

Existem diferentes algoritmos de *Machine Learning* que conforme o problema tem uma melhor eficiência e resultados e que podem ser selecionados a partir de alguns critérios definidos. Um dos critérios, seria se o problema terá que adotar qual paradigma de aprendizado, o preditivo ou o descritivo. Conforme a imagem a seguir, mostra a hierarquia de aprendizado, no topo o processo de aprendizado indutivo, em seguida para o supervisionado que seriam as tarefas preditivas (classificação e regressão) e os não supervisionados que seriam as descritivas (agrupamento, associação, sumarização), esses são os mais conhecidos e mais utilizados. (CARVALHO et al., 2011).

Figura 01 – Hierarquia aprendizado de máquina



Fonte: CARVALHO et al (2011)

2.1.1 Algoritmos

Conforme dito, existem diversos algoritmos de *Machine Learning* e abordagens, para cada situação uma das metodologias vai se encaixar melhor e trará melhores resultados. No caso do projeto proposto foram aplicados aprendizados de sistemas supervisionados e regressão logística.

2.1.1.1 Supervisionados

O algoritmo preditivo ou supervisionado é uma função que, a partir de um conjunto de dados já rotulados e conhecidos, constrói um modelo que consegue estimar rótulos de novos dados. O rótulo é classe do dado ou atributo de saída. Se o domínio deste conjunto de dados com valores nominais é considerado um classificador e caso seja um conjunto infinito e ordenado de valores, tem um problema de regressão. Sendo através de um dado exemplo sem rótulo, consegue identificar uma das possíveis classes ou valor real para dado. (CARVALHO et al, 2011).

São treinados com uma ou mais entradas chamadas de atributos de entrada na qual a saída chamada por classe desejada é conhecida. Por exemplo, pode ser o histórico de conversas a respeito de um determinado assunto e a classificação da entidade que aquela mensagem se propõe.

O algoritmo supervisionado busca através do espaço hipóteses possíveis (classes) por aquele que terá melhor desempenho para o dado, isso acontece também para conjuntos que não fazem parte do conjunto de treinamento. Com um conjunto de testes de exemplos que são distintos do conjunto de treinamento ou com novos exemplos com diferentes valores, podemos medir a precisão de uma hipótese de um modelo gerado. Dizemos que uma hipótese generaliza bem se prevê corretamente o valor de saída para novos exemplos. (NORVIG; RUSSEL, 2013).

2.1.1.2 Árvore de decisão

Uma árvore de decisão representa uma função que toma como entrada um vetor de valores de atributos e retorna uma "decisão" – um valor de saída único. (RUSSEL e NORVIG, 2014) É um algoritmo para construção em cima de dados, tem como base a estratégia de ser guloso de dividir para conquistar e sempre testar o atributo mais importante em primeiro lugar, que vai ter maior diferença na classificação.

O algoritmo da decisão leva em consideração a estratégia de dividir um problema complexo em problemas mais simples, aos quais é atuado com a mesma estratégia recursivamente. Uma árvore de decisão é um grafo acíclico, que não tem ligação com o próprio nó e podem seguir para um próximo nó. Esta árvore é composta por nós, existem o nó de divisão e o nó de folha. O nó de divisão tem dois ou mais sucessores, contendo um teste condicional nos valores de um atributo para realizar a divisão e dizer qual será o próximo nó dado certo valor (CARVALHO et al., 2011).

A indução de decisão da árvore tenta deixa-la consistente com os exemplos e seja a menor possível árvore e seja pouco profunda (NORVIG; RUSSEL, 2013) É utilizada heurística nos algoritmos localmente que olham um passo para frente, uma vez que o algoritmo toma a decisão, ele nunca volta atrás, podendo gerar algo ruim globalmente.

2.1.1.1.2 Regressão Logística

Regressão logística é um modelo de regressão que a variável dependente é binária ou categórica.

A Regressão Logística mede a relação entre a variável dependente binária e uma ou mais variáveis independentes estimando probabilidades utilizando uma função logística, que é a Distribuição Acumulada Logística. Na interpretação da variável dependente deste método, o erro da Regressão Logística assume-se uma distribuição logística padrão.

O modelo pode ser visto como um caso intermediário a estes dois casos vistos acima. Ou seja, o modelo busca suavemente restringir sua saída para um

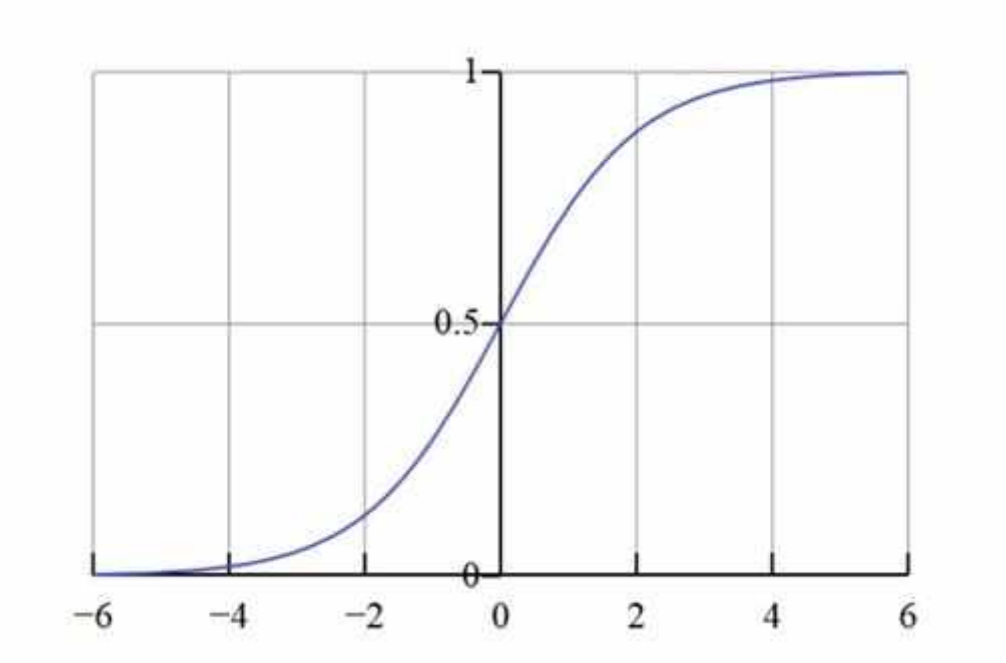
intervalo de probabilidade entre [0,1]. Este modelo é chamado de modelo de Regressão Logística e pode ser representado da seguinte forma:

$$h(x) = \theta(w^T x)$$

Onde teta é chamado de função logística

$$\emptyset(s) = \frac{e^s}{1 + e^s}$$

Figura 02 – Função Logística no intervalo de s



Fonte: COPPIN, B (2017)

2.2 Computação em Nuvem

Para que o *chatbot* esteja disponível a qualquer momento foi utilizado uma plataforma de computação em nuvem chamada Microsoft Azure. Os conceitos e fundamentos para compreensão dessa tecnologia são abordados a seguir.

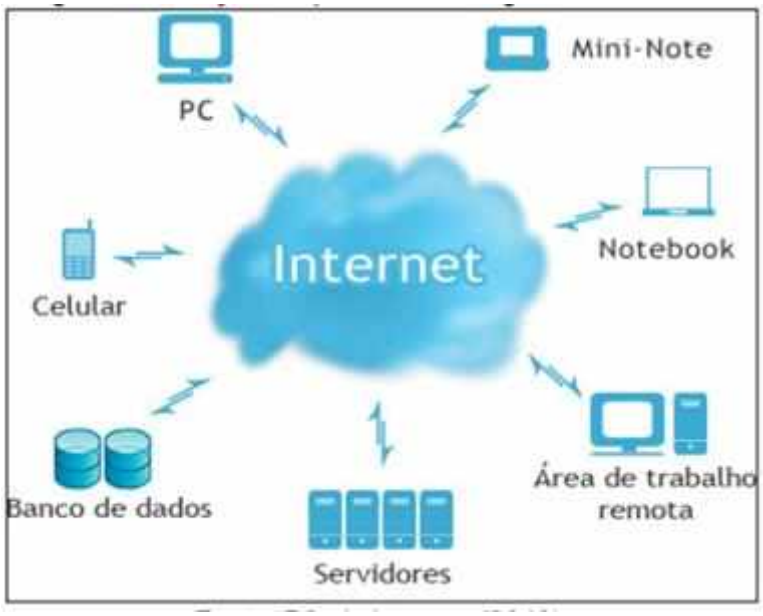
A Computação em nuvem ou *Cloud Computing* (Computação em Nuvem) é um conceito da tecnologia da informação que descreve a organização e

disponibilização de serviços que estão distribuídos por uma vasta rede de servidores (a Nuvem), através de um modelo de utilização sob demanda e com o pagamento baseado no uso destes serviços (Silva, 2010).

De acordo com Taurion (2009, p.7), o termo Computação em Nuvem surgiu em 2006 em uma palestra de Eric Schmidt, da Google, sobre como sua empresa gerenciava seus *Datacenters*.

A Computação em Nuvem é uma possibilidade que o usuário tem de acessar arquivos e executar diferentes tarefas sem a necessidade de instalar aplicativos na sua área de trabalho (ou Desktop) ou em outros tipos de dispositivos. A figura a seguir ilustra o acesso à Nuvem através de diferentes tipos de dispositivo.

Figura 03 – Serviço e dispositivos interligados na “Nuvem”



Fonte: B2ml sistemas (2013)

De acordo com Silva (2010, p.3), a palavra Nuvem sugere uma ideia de ambiente desconhecido, do qual podemos ver somente seu início e fim, onde toda a infraestrutura e recursos computacionais ficam “escondidos”. O usuário acessa apenas a uma interface padrão através da qual é disponibilizado todo o conjunto de aplicações variadas e serviços.

Para tornar este modelo possível, é necessário reunir todas as aplicações e dados dos usuários em grandes centros de armazenamento, denominado *Datacenters*.

Os servidores que hospedam dados e aplicativos ficam localizados em *Datacenters* de empresas de qualquer parte do mundo. Uma vez reunidos, a infraestrutura e as aplicações dos usuários são distribuídas na forma de serviços disponibilizados por meio da internet.

Figura 04 – Datacenter da Google

Fonte: Google (2013)

2.2.1 Características essenciais

a) **Autoatendimento sob demanda:** Segundo NIST (2011), o usuário pode adquirir qualquer recurso computacional, como tempo de processamento no servidor ou armazenamento na rede na medida que necessite e sem precisar de interação humana com os provedores de cada serviço.

b) **Amplo acesso a rede:** Os recursos estão disponíveis através da rede e são acessados por meio de interfaces padronizadas, e mecanismos que promovam o padrão utilizado por dispositivos como smartphones, tablets, notebooks, desktops ou outros dispositivos. A interface de

acesso à Nuvem não obriga os usuários a mudar suas condições e ambientes de trabalho, como por exemplo, o sistema operacional.

c) **Agrupamento por recursos:** Os recursos computacionais como o armazenamento, processamento e memória são agrupados e atribuídos dinamicamente de acordo com a demanda dos consumidores. Os consumidores geralmente não têm controle ou conhecimento da localização dos recursos disponibilizados, mas podem ser capazes de especificá-los em um nível maior de abstração como, por exemplo, o país, estado.

d) **Serviço medido:** É possível controlar e otimizar a utilização dos recursos, proporcionando a medição de uso, como por exemplo, o controle da capacidade de armazenamento ou o tempo de utilização do processamento. Podemos monitorar e controlar o uso de recursos, garantindo a transparência para o provedor e o usuário do serviço utilizado.

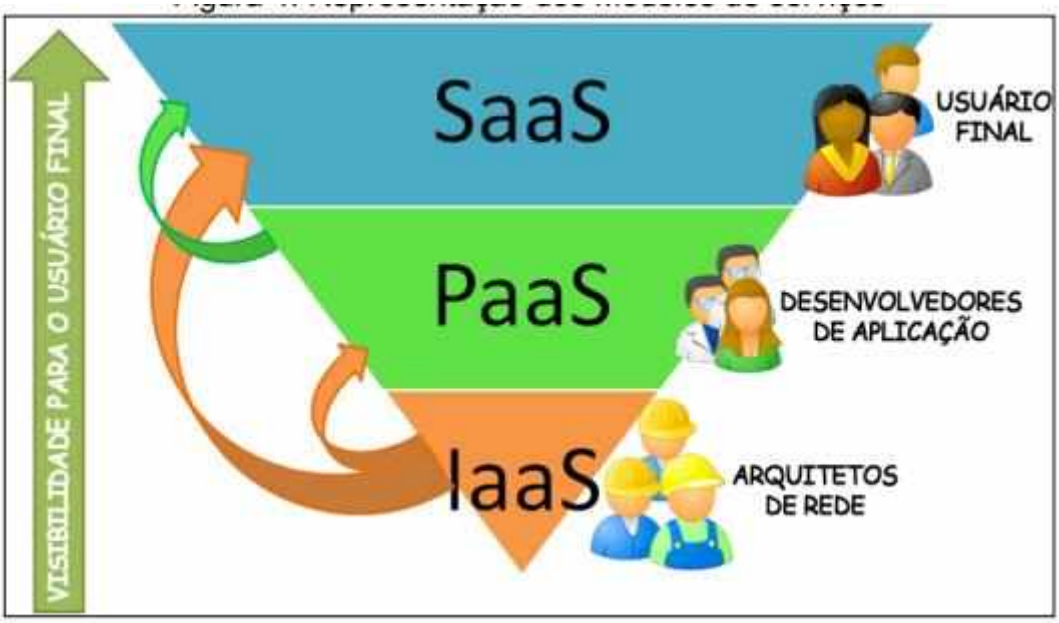
2.2.2 Modelos de serviços na nuvem

A Computação em Nuvem distribui os recursos na forma de serviços. Assim, podemos classificar a Computação em Nuvem em três modelos:

i. Infraestrutura como Serviço (IaaS);
ii. Plataforma como Serviço (PaaS);
iii. Software como Serviço (SaaS)

Os serviços podem ser representados como uma pirâmide invertida, conforme a figura a seguir.

Figura 05: Representação dos modelos de serviços



Fonte: Vitor Meriat (2013)

A infraestrutura como serviço (IaaS) é a ponta da pirâmide, onde o cliente pode ser o arquiteto de rede ou um usuário que necessite um serviço de *hardware*, ou seja, montar uma máquina virtual ou utilizar o serviço de armazenamento. Já no meio da pirâmide consta a plataforma como serviço (PaaS), o qual fornece os recursos de infraestrutura e servidor para os desenvolvedores de aplicação criarem e testarem seus aplicativos. E na base da pirâmide invertida o *software* como serviço (SaaS), que é composto pelos serviços IaaS e PaaS. Porém o usuário final só utiliza os *softwares* prontos fornecidos pelo provedor, portanto não possui acesso a configuração dos outros modelos de serviços.

- **Infraestrutura como Serviço (IaaS):** O principal objetivo do IaaS é tornar mais fácil e acessível o fornecimento de recursos, tais como servidores, rede, armazenamento e outros recursos de computação fundamentais para construir um ambiente sob demanda, ou seja, um ambiente que o usuário possa configurar à medida que necessite. O IaaS oferece infraestrutura de computação, num ambiente de virtualização, onde o usuário cria uma máquina virtual. Daniels (2009) define virtualização como um ambiente ou camada de abstração entre o *hardware* de um sistema computacional e o *software* sendo executado sob esse *hardware*. Nesse ambiente de virtualização é

criada uma máquina virtual que emula um ambiente computacional, ou seja, ao invés de ser uma máquina real, isto é, um computador real, feito de *hardware* e executando um sistema operacional específico, uma máquina virtual é um computador fictício criado por um programa de emulação. Por exemplo, um desktop com sistema operacional Linux, possui uma máquina virtual com outro tipo de sistema operacional o Windows XP, conforme a imagem a seguir

Figura 06: Máquina virtual acessando o Windows XP dentro do Linux

Fonte: Procedural (2013)

- **Plataforma como Serviço (PaaS):** No modelo PaaS o provedor disponibiliza uma plataforma computacional onde o usuário pode criar e testar suas aplicações criadas por linguagens e ferramentas suportadas pelo provedor da Nuvem. Isso facilita o desenvolvimento de aplicações sem o custo e a complexidade de compra e gestão de *hardware* e ferramentas de desenvolvimento de *software*. O desenvolvedor não precisa instalar e configurar servidores de acordo com os pré-requisitos do *software* para criar seu aplicativo, somente precisa utilizar a plataforma PaaS.
- **Software como Serviço (SaaS):** O modelo SaaS é projetado para o usuário final, trata-se de uma forma de trabalho onde o *software* é

oferecido como serviço, ou seja, é uma forma de distribuir e comercializar o *software*. O fornecedor se responsabiliza por toda a estrutura necessária para a utilização, ou seja, servidores, segurança da informação e conectividade, e o cliente utiliza o *software* via Internet, pagando um valor de acordo com seu uso, sem ter que se preocupar com custos de licença e atualizações. O pagamento pelo uso pode se dar por meio de um valor periódico ou pela quantidade de uso. Além disso, alguns serviços podem ser gratuitos. Esses programas podem ser acessados através de diversos dispositivos (notebooks, tablets, smartfones, desktops entre outros), através do uso de uma interface como, por exemplo, a de um navegador. Muitos destes *softwares* são usados diariamente sem que boa parte dos usuários saibam que estão utilizando um serviço da Nuvem, como, por exemplo, o Google Drive que é um serviço de armazenamento que integra o Google Docs, um pacote de aplicativos que permite processamento de texto, apresentações e formulários, assim como importação e exportação de arquivos do 29 tipo .doc, .xls, .html, entre outros. É possível fazer o compartilhamento dos arquivos com um grupo específico de usuários.

2.3 Raspberry Pi 3B+

Segundo Douglas Ciriaco (2015), a *Raspberry Pi* é um computador de baixo custo do tamanho de um cartão de crédito, desenvolvida no Reino Unido pela fundação *Raspberry Pi*. A função básica do *gadget* é oferecer uma alternativa barata, prática e acessível, para que pessoas de várias idades possam explorar todas as capacidades da computação.

Figura 07 – *Raspberry Pi* 3 B+

Fonte: Adaptado de *Raspberry Products.*

Segundo a documentação da *Raspberry Pi*, o modelo 3B+ conta com um processador Broadcom BCM2837B0, chip de 64 bits com quatro núcleos Cortex-A53, 1.4GHz de *clock*, 1GB de memória RAM, adaptador Wifi integrado, 4 portas USB 2.0 e 40 pinos GPIO.

Na figura abaixo estão dispostos as interfaces e os recursos disponíveis no modelo “B”, utilizado neste projeto. Suas especificações são apresentadas a seguir.

Figura 08 – Componentes *Raspberry*

Fonte: Adaptado de *Raspberry Products.*

1. DSI vídeo: Possui uma saída de vídeo com interface serial, nesse conector é possível instalar um monitor.

2. GPIO: Portas *input*/*output*: Proporcionam uma maneira fácil de conectar os componentes de *hardware* desenvolvidos. A GPIO é composta por 26 pinos distribuídos entre *Ground*, *in*/*out*, sendo todos digitais. Possui tensão de 3.3 Volts e tensão de 5 Volts, PWM, RX, TX, MOSI, MISO, SCK, SDA e SCI. A distribuição desses pinos está descrita a seguir.

Figura 09 – Distribuição GPIO

Fonte: Adaptado de *Raspberry Products.*

1. CPU, GPU e RAM: O *Raspberry* opera com um processador ARM e velocidade base de 700MHz, também conta com um processador que decodifica e reproduz vídeo de até 1080p em alta definição e uma memória RAM de 1GB.

2. RCA vídeo: Saída de vídeo analógica.

3. Áudio estéreo: Permite a montagem de uma central multimídia, nesse conector pode-se efetuar a ligação da caixa de som.

4. LED status: Indica o status de energia.

5. 4 USBs: Possui duas entradas USB, que podem ser utilizadas para a integração com diversos dispositivos. Os mais usuais são *mouse*, teclado ou um USB *Switch*.

6. RJ45- Ethernet: Dispõe de uma interface *onboard* para ligação da placa em rede.

7. Entrada para câmera

8. Controle USB e Internet: Microchip responsável por executar as funções de controle e de acesso à *Web* e conexões USB.

9. Saída HDMI

10. Regulador de tensão: Devido às saídas de 3.3 Volts, foi implementado o regulador de tensão, que reduz os 5 Volts de tensão de entrada.

11. Alimentação elétrica *micro* USB. A entrada deve ser de 5 Volts (VDC) com 5% de tolerância e uma corrente mínima de 700mA.

A fundação *Raspberry* desenvolveu uma distribuição otimizada e *open source* baseada em Linux embarcado chamada Raspbian. É um sistema operacional composto de centenas de pacotes de *software*, com desenvolvimento ativo e constante de atualizações, com ênfase na melhoria da estabilidade e do desempenho.

3. Metodologia

Neste capítulo será abordado todo o desenvolvimento do projeto proposto, desde a montagem do circuito eletrônico até o desenvolvimento dos *softwares* e treinamento do modelo. Serão explorados com detalhes os componentes a serem utilizados e os parâmetros a serem estudados.

3.1 Rasa Stack

O *framework Rasa* ou *Rasa Stack* possui um conjunto de ferramentas de aprendizado de máquina, para que desenvolvedores possam criar *Chatbots*

contextuais, diferentes daqueles baseados em regras predefinidas. Esse *framework* é composto de dois módulos, que são independentes e podem ser usados separadamente: o módulo principal (*Rasa Core*) e o módulo de processamento de linguagem natural (chamado de *Rasa Natural Langue e Understanding* ou *Rasa NLU*).

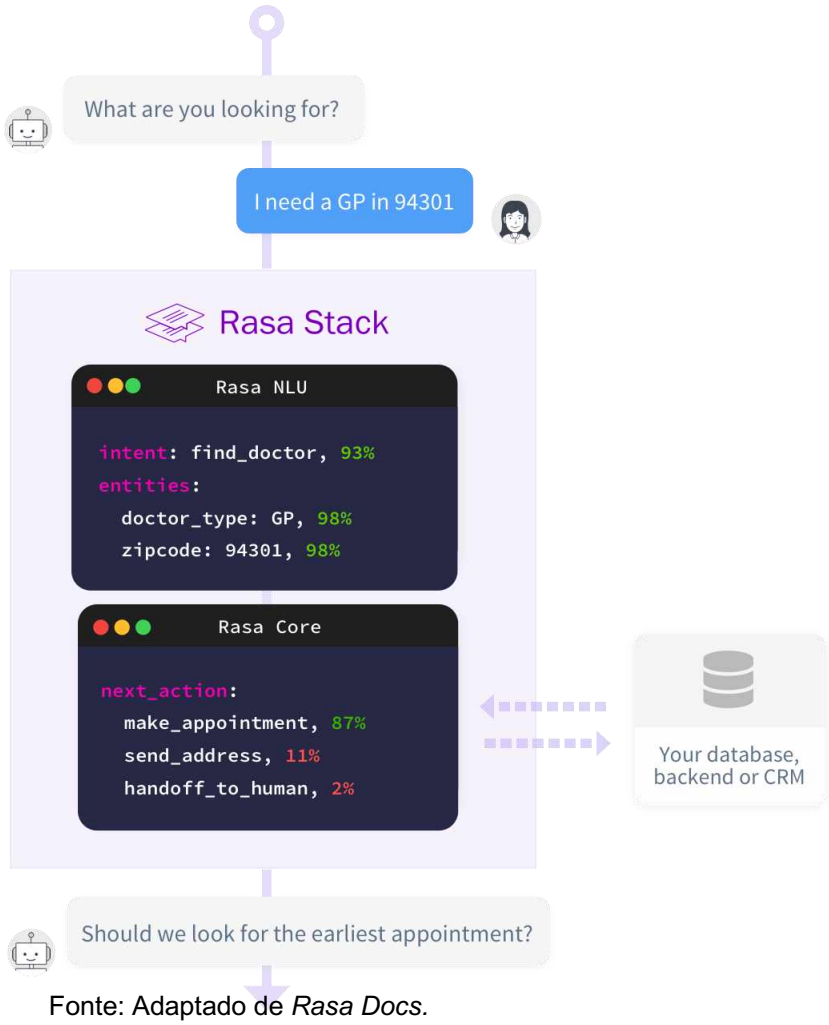
Os principais conceitos no desenvolvimento de um *Chatbot* são:

- **Entidades**: são informações específicas de um domínio, extraídas de uma expressão. Têm como objetivo entender a intenção, além de ajudar a identificar parâmetros necessários para tomar ações específicas (KAR R.; HALDAR, 2016).

- **Intenções**: as intenções são cruciais em uma aplicação de *Chatbot*, representando o que os usuários estão buscando realizar ou saber de uma mensagem (KAR R.; HALDAR, 2016).

- **Diálogo**: o diálogo utiliza as intenções, as entidades e o contexto da aplicação para retornar uma resposta na entrada do usuário.

Figura 10 – *Framework* Rasa



Fonte: Adaptado de *Rasa Docs.*

3.1.1 Rasa NLU

É uma ferramenta para o entendimento de linguagem natural *open source*, capaz de classificar as intenções das conversas com usuários e extrair entidades em *chatbots*. Sendo assim, é possível extrair dados estruturados de uma conversa em linguagem natural. O módulo NLU (*Natural Language Understand*) pode ser usado separadamente em qualquer projeto de *chatbot* inteligente, visto que independe de qualquer infraestrutura proprietária ou de requisições a API's externas, pois todos os dados de treinamento podem ser armazenados localmente.

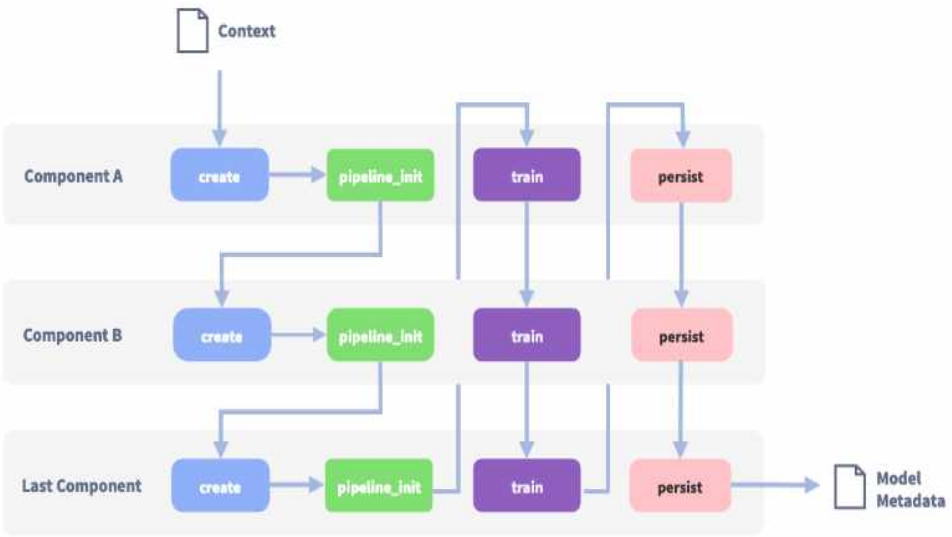
Para fazer esse processo de classificação e de extração, o *Rasa* NLU utiliza de um *pipeline* que define o modelo NLU, descrevendo como características

específicas do texto são extraídas dos exemplos de treinamento, quais classificadores de intenções ou de extrações de entidades são utilizados para fazer as predições, bem como quais informações adicionais são extraídas das entradas do usuário. O *pipeline* não define apenas quais componentes serão considerados do NLU, mas também qual a sequência em que serão treinados e executados. Cada um de seus componentes processa um *input* e cria um *output*, que será utilizado como *input* de outro componente desse *pipeline*. Os componentes utilizados neste projeto serão detalhados mais a frente.

Nesse momento que os conceitos vistos sobre *Machine Learning* são aplicados, pois o Rasa NLU abstrai em uma camada mais externa as técnicas pré-selecionadas. Ao construir o modelo de treinamento foi indicado a classe correspondente (ou intenção) e a partir desses dados que são enriquecidos através das entidades o modelo de regressão logística é treinando e assim ele consegue estatisticamente indicar a intenção de uma nova entrada.

Com o Rasa NLU, foram realizados a preparação dos dados de treinamento para o *chatbot*, a criação dos arquivos de configuração, a seleção de um *pipeline* e o treinamento do modelo.

Figura 11 – Ciclo do NLU



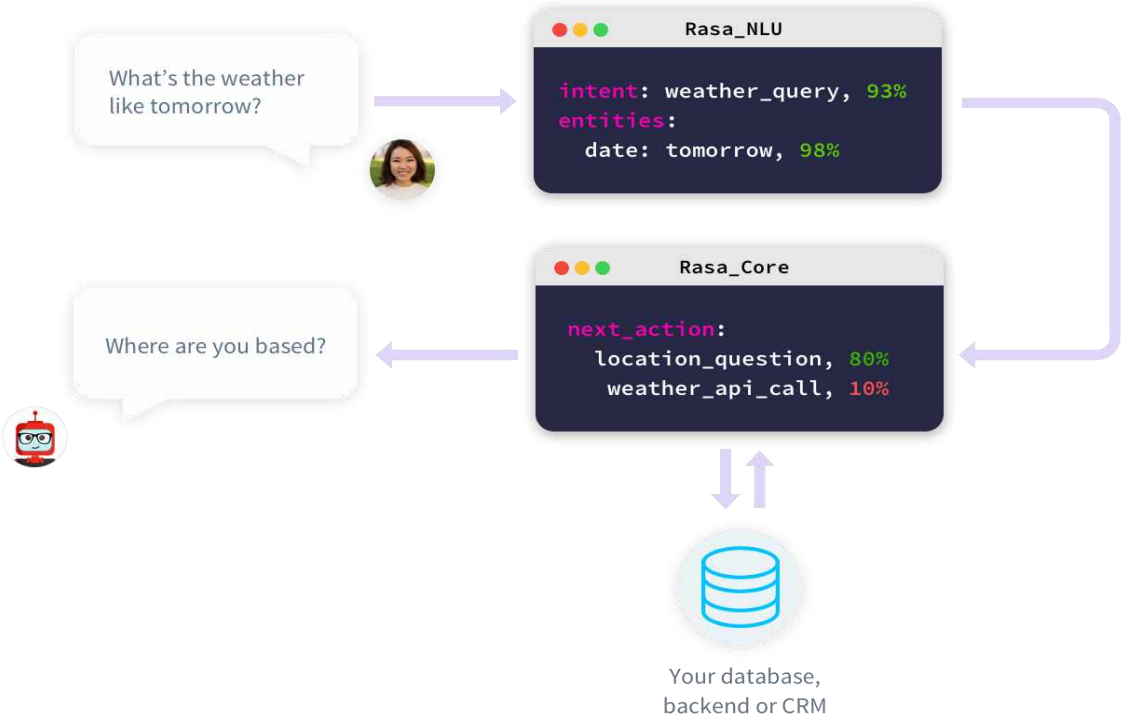
Fonte: Adaptado de *Rasa Docs.*

3.1.2 Rasa Core

O *Rasa Core* (núcleo) é uma ferramenta que usa aprendizado de máquina para inferir possíveis ações a serem executadas pelo *chatbot*. O aprendizado de máquina é feito com o constante *feedback* do usuário e, a partir disso, são calculadas as probabilidades de execução de cada ação. Essa abordagem é conhecida como aprendizado iterativo, pois o *Core* analisa as mensagens por meio de gerência de fluxo de conversas, de modo a deixar o diálogo mais fluido e sem perder o contexto.

Além disso, é responsável por treinar o modelo de gerenciamento de diálogo para preparar as respostas dadas ao usuário. Em vez de incluir várias condições na árvore de decisão e passar horas depurando a mesma, no caso de uma grande aplicação, é melhor ensinar o modelo a criar respostas que podem ser padrões ou ações que podem fazer consultas a bancos de dados e requisições *http*, assim como variar a resposta de acordo com os dados tratados nessas funções.

Figura 12 – Fluxo Rasa



Fonte: Adaptado de *Rasa Docs.*

3.2 Microsoft Azure

O uso da tecnologia de *cloud computing* mudou, na mesma velocidade com que vem se aperfeiçoando nas últimas duas décadas. O que era visto apenas como uma forma de armazenar arquivos remotamente, transformou-se no principal modelo de produtividade. No centro dessas mudanças estão os grandes serviços que permitem a inclusão de estratégias de nuvem, como o *Microsoft Azure*.

Com o crescimento da demanda por computação na nuvem e a sofisticação dessa tecnologia na última década, não demorou muito para que uma das maiores empresas do mundo entrasse para o mercado e oferecesse serviços de computação na nuvem. Apresentado em 2008, o *Windows Azure* foi lançado em 2010 e renomeado para Microsoft Azure em 2014.

Hoje, o serviço concentra toda a plataforma de nuvem da Microsoft – desde a infraestrutura de *cloud computing*, para hospedar sistemas, até ferramentas e recursos que expandem a capacidade produtiva de negócios em todos os setores.

A partir do entendimento do funcionamento do *Microsoft Azure*, destacaremos os principais serviços disponíveis na plataforma:

- **Máquinas virtuais:** São sistemas operacionais que funcionam como computadores completos, mas com todos seus recursos de armazenamento e de computação originados da nuvem.

- **Aplicações em nuvem:** Assim como é possível simular um computador inteiro dentro da nuvem, o Microsoft Azure também pode ser usado como plataforma de *softwares* virtualizados. Assim, é possível acessar ferramentas produtivas importantes de qualquer lugar do mundo.

- **Armazenamento e Backup**

- **Docker**: São pacotes de códigos e de ferramentas isolados do sistema em que estão funcionando. São a solução mais popular hoje para dar portabilidade e flexibilidade ao desenvolvimento de produtos digitais.

A Microsoft Azure tem um bom custo-benefício, seu preço é flexível de acordo com o(s) serviço(s) consumido(s), sendo calculado conforme o número de vezes que o(s) serviço(s) é/são utilizado(s).

3.3 Telegram

O Telegram, tal como outros canais populares como WhatsApp e Viber, é um serviço de mensagens instantâneas. Ele é baseado em nuvem e está disponível em uma grande variedade de dispositivos.

Além de ter parte de seu código-fonte aberto e uma API com vastas possibilidades, o Telegram possui um grande suporte para a criação de *bots* ricos em recursos. Desde junho de 2015, desenvolvedores mundo afora podem criá-los na plataforma.

Os *chatbots* no Telegram consistem em contas especiais que não requerem um número de telefone para serem configurados. Também não é necessária nenhuma autorização prévia para publicar um *bot*, mas seu funcionamento deve estar de acordo com os termos de uso do serviço.

Os usuários podem interagir com ele de duas maneiras:

- Enviando mensagens e comandos, a partir de uma conversa privada com o *bot*, ou o adicionando em grupos.

- Enviando requisições diretamente do campo de texto, digitando o @nomeDoBot e uma *query* (um comando predefinido que o seu *bot* suporte).

Mensagens, comandos e requisições enviados pelos usuários passam pelos servidores do Telegram, que lidam com a criptografia e a comunicação com a *API* do serviço.

3.4 Flask

Lançado em 2010 por Armin Ronacher, o flask é um *microframework* Python, destinado principalmente a pequenas aplicações com requisitos mais simples. Isso significa que ele não precisa de várias dependências para funcionar. É uma tecnologia que rapidamente foi adotada por desenvolvedores do mundo todo, pois

tem uma forma minimalista de utilizar o Python, sem que seja necessário escrever muito código, além de ser muito bem organizado.

Para entendermos o Flask, primeiro precisamos entender como funciona a Web. Quando digitamos um endereço no navegador, por exemplo, ele é enviado (a partir do HTTP) para um servidor, por meio de uma requisição (*request*). Essa requisição é processada no servidor e enviada de volta ao navegador em forma de uma resposta (*response*), também por meio do HTTP. No projeto desenvolvido, todo o processamento foi feito utilizando Flask.

Ele é baseado em 3 pilares:

- **WerkZeug**: Uma biblioteca para o desenvolvimento de programas de *software* WSGI (*Web Server Gateway Interface*), que é a especificação universal de como deve ser a interface entre um *app* Python e um *web server*. Ela possui a implementação básica desse padrão para interceptar *requests* e lidar com *responses*, controle de *cache*, *cookies*, *status* HTTP, roteamento de *urls* e também com uma poderosa ferramenta de *debug*.

- **Jinja2**: É um *template engine*. Vem com a implementação da maioria dos elementos necessários para a construção de um *template* HTML.

- ***Good Intentions***: Liberdade de estruturar sua aplicação da maneira que quiser, além de contar com diversas extensões que podem ser utilizadas para facilitar o desenvolvimento.

3.5 Ngrok

O ngrok é um *proxy* reverso que cria um “túnel” seguro, a partir de um *endpoint* público para uma web *service*, sendo executado localmente. Em outras palavras, ele é utilizado para expor uma porta em seu sistema, por exemplo, a porta em que sua aplicação está rodando, a uma URL pública.

Quando você inicia o ngrok, ele exibe uma interface do usuário em seu terminal com a URL pública de seu túnel, assim como outras informações de status e de métricas sobre as conexões.

Figura 13 – Exemplo *ngrok*

Fonte: Adaptado de *Ngrok docs*

3.6 Montagem do circuito eletrônico

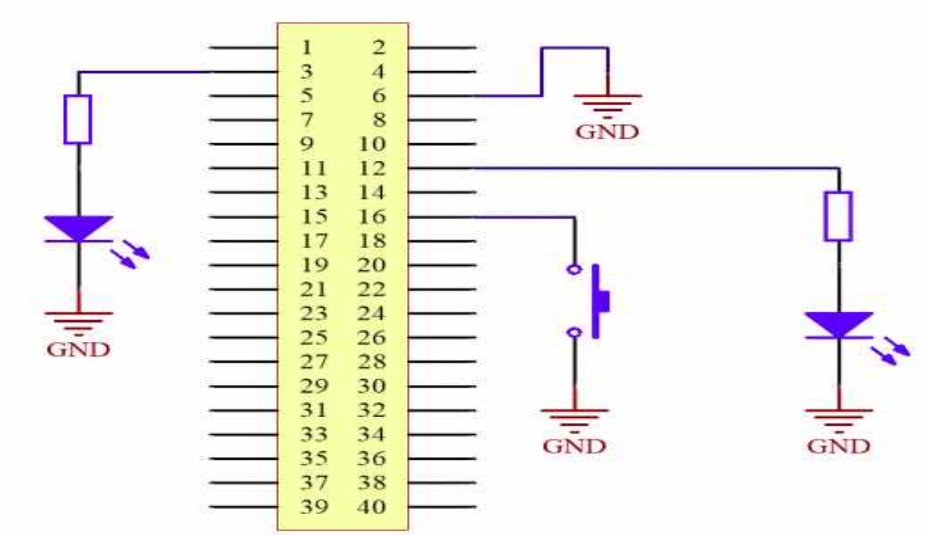
O projeto proposto foi montado utilizando de poucos componentes e ocupando pouco espaço físico.

Para a montagem do circuito foram utilizados:

- 2 LEDs (3mm ou 5mm, um vermelho e outro amarelo);
- 2 resistores de 1k;
- 1 *Push-button* (tipo normalmente aberto);
- 1 *protoboard*
- Fios para interligação da *Raspberry* com a *protoboard*;

A figura abaixo mostra o esquemático do circuito que será acionado. Os dois LEDs são acesos ao se colocar nível alto nos pinos em que estão conectados, e o *push-button* envia um sinal de nível baixo quando conectado (o processador da *Raspberry* pode ser configurado para colocar um resistor de *pull up* nos pinos de I/O).

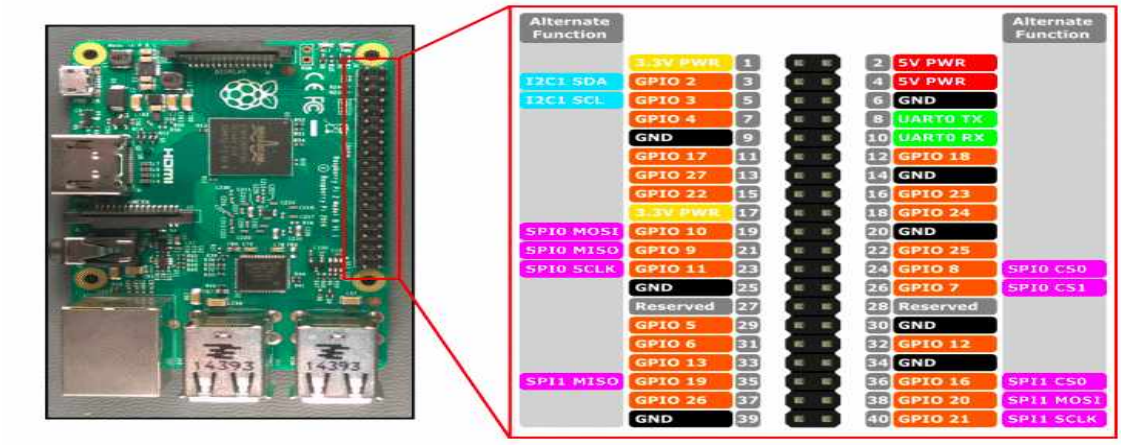
Figura 14 – Diagrama esquemático



Fonte: Karve et al. (2018).

A figura a seguir mostra o nome dos pinos do conector da *Raspberry Pi* 3B+, bem como sua numeração.

Figura 15 – Distribuição pinos *Raspberry Pi* 3B+



Fonte: Raspberry Docs

O primeiro LED foi conectado ao GPIO2 da *Raspberry Pi*, pino 3. Essa GPIO não foi escolhida por algum motivo particular, e qualquer outro GPIO poderia ter sido escolhido, bastando alterar o código para se adequar a essa mudança.

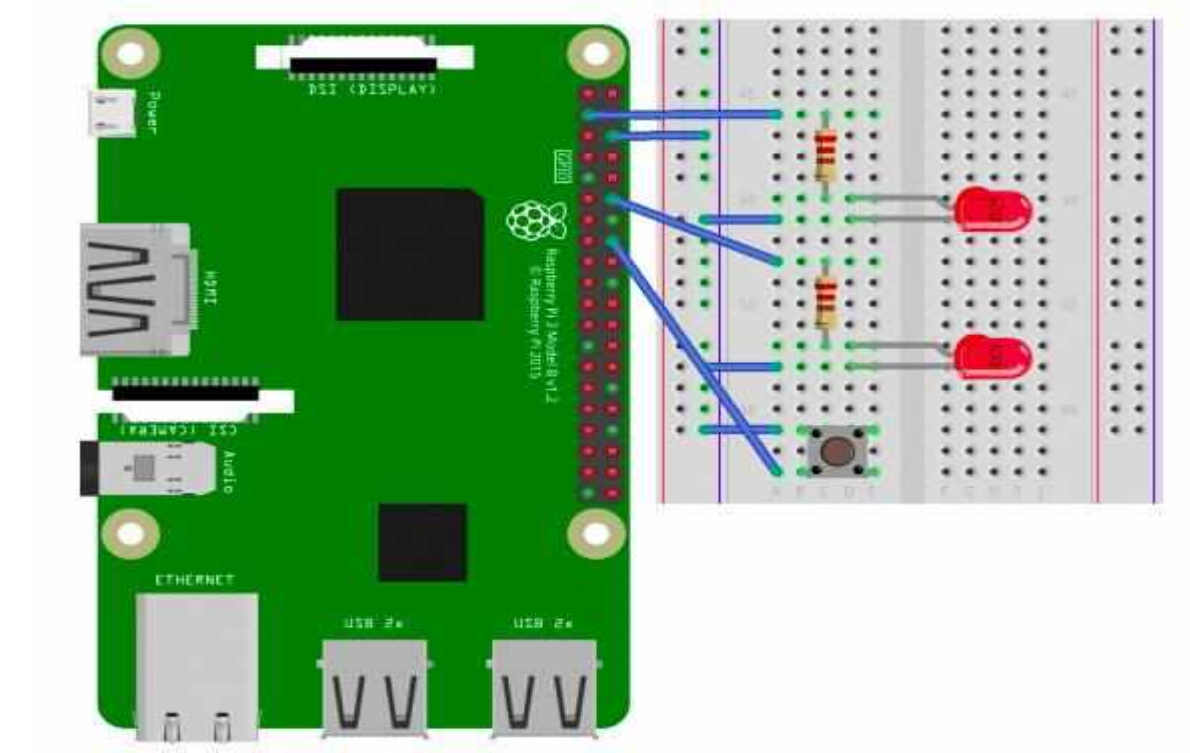
O segundo LED foi conectado ao GPIO18, pino 12. O processador da *Raspberry Pi* possui vários pinos com capacidade de operar em modo PWM (por

hardware), porém o único pino presente no conector é o GPIO18. Assim, ele deve ser utilizado caso se utilize a função de PWM por *hardware*.

O *push-button* foi conectado ao GPIO23, pino 16. Para tornar o circuito simples, o código que configura e manipula os GPIOs habilita o resistor de *pull up* interno nesse pino. Esse GPIO também não foi escolhido por motivo particular, de modo que qualquer outro poderia ser utilizado.

A figura abaixo mostra a forma como os componentes foram montados e conectados na *protoboard*.

Figura 16 – Montagem dos componentes



Fonte: Adaptado de *Raspberry Products*

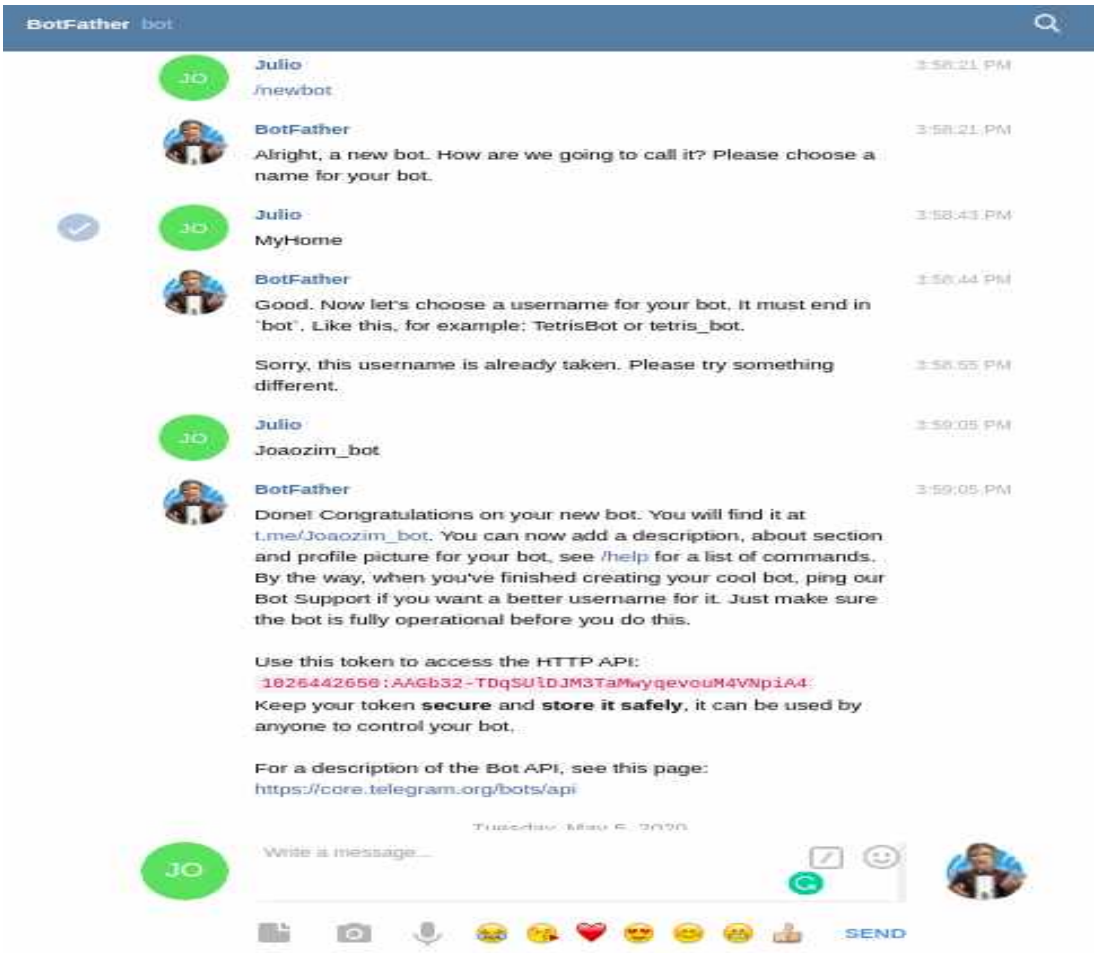
3.7 Telegram Bot

O Telegram oferece um suporte a criação de *bots* tornando a sua criação um processo bem intuitivo e visual. Para isso, basta conversar com outro *bot*, chamado *BotFather*, seguir os passos instruídos por ele e então você receberá um *token* de autorização, por meio do qual é possível realizar chamadas HTTP na API do Telegram.

O *BotFather* funciona basicamente por comandos e uma lista deles pode ser vista ao se digitar "/" na caixa de texto, em que o *app* irá sugerir comandos disponíveis, cujos nomes são bem explicativos.

A criação do *bot* utilizado neste projeto está demonstrada na figura a seguir.

Figura 17 – Criação do *bot*



Fonte: autoria própria (2020).

Após essa conversa simples, o *bot MyHome* foi criado e um *token* foi gerado para acessar a API do Telegram.

3.8 Desenvolvimento do software na Raspberry Pi

3.8.1 Script de controle dos LEDs

Para o controle dos LEDs que compõem o projeto foi desenvolvido um *script* que cria rotas e *views* a partir do Flask para ligar ou desligá-los.

No primeiro momento, as bibliotecas e os pacotes são importados.

Figura 18 – Importação bibliotecas para controle dos LEDs

```
1  import RPi.GPIO as GPIO # Import Raspberry Pi GPIO library
2
3  import time, datetime # Creates a date from a timestamp
4
5  from flask import Flask # Import flask
6
7  from flask import request # Import request
8
9  from flask import abort # Import abort
10
11 from flask import jsonify # Import jsonify
12
13 import time # Import Time
```

Fonte: autoria própria (2020).

Acima é importada a biblioteca RPi.GPIO para acessar as portas GPIO e a classe *flask* do pacote *Flask*.

Figura 19 – Inicialização dos pacotes

```
17  app = Flask('_name_') # Creates a Flask instance
18
19  GPIO.setwarnings(False) # Ignore warning for now
20
21  GPIO.setmode(GPIO.BOARD) # Use physical pin numbering
22
23  GPIO.setup(5, GPIO.OUT) # Define pin 5 as output
24
25  GPIO.setup(40, GPIO.OUT) # Define pin 40 as output
26
27  pwm = GPIO.PWM(40, 100) # Configure PWM in pin 40 with 100Hz
28
29  pwm.start(100) # Start DC = 100%
```

Fonte: autoria própria (2020).

Nesse fragmento do código, foi instanciado um objeto da classe *Flask*, utilizado para configurar a aplicação e para executá-la com o servidor do próprio *Flask*. Além disso, foram configurados o modo de acesso ao GPIO para BOARD (posição física dos pinos), os pinos 5 e 40 como *output* e o 40 como PWM, iniciando com um *Duty Cycle* de 100%.

Com o *Flask* e os pinos configurados, a próxima etapa foi criar as rotas e as *views* para o *chatbot* conseguir se comunicar com a *Raspberry Pi* e executar as ações solicitadas.

Figura 20 – Criação da rota que liga o LED

```
31  @app.route('/on', methods=['GET']) # Creates route /on
32
33  # View for route /on
34  def on():
35
36          GPIO.output(5, True)
37          return jsonify({'on':'done'}), 201
```

Fonte: autoria própria (2020).

@app.route é um decorador responsável por interpretar a rota que acessamos, de modo que, assim que acessada a URL /on, a função que está embaixo é executada. A função é responsável por acender o LED que se encontra no pino 5 e retornar um JSON.

São criadas outras duas rotas: uma para apagar o LED e outra para controlar o PWM. Ambas seguem o mesmo modelo em que são configuradas as rotas e definidas as funções que serão executadas.

Figura 21 – Criação da rota que controla o PWM

```
47  @app.route('/pwm', methods=['GET']) # Creates route /pwm
48
49  # View for route /pwm
50  def pwm():
51
52          dc = 100
53          while(dc != 0):
54                  pwm.ChangeDutyCycle(dc)
55                  time.sleep(0.05)
56                  dc = dc - 1
57          return jsonify({'pwm':'off'})
```

Fonte: autoria própria (2020).

Acima são criadas a rota para o PWM e a função que vai controlar seu funcionamento, iniciando o valor com o *Duty Cycle* igual a 100 e reduzindo de um em um até chegar a 0.

Com isso, temos os dois LEDs configurados e a *Raspberry Pi* pronta para receber as requisições do *chatbot* e apagar ou desligá-los.

3.8.2 Script de controle do botão

Para o funcionamento do botão, foi feita uma integração direta entre a *Raspberry Pi* e o Telegram, para isso, foi utilizada a biblioteca *telepot*, que realiza a comunicação da placa com o API do Telegram, sem que seja necessário esse processo passar pelo *ChatBot*. Para usar o *telepot*, alguns parâmetros foram configurados para realizar a comunicação de uma forma segura.

Figura 22 – Inicialização do *telepot*

```
5  import telepot # Communicate with Telegram API
6
7  from telepot.loop import MessageLoop
8
9  chat_id = 0
```

Fonte: autoria própria (2020).

O trecho acima demonstra como são feitas a importação da biblioteca *telepot* e a criação da variável *chat_id*, que depois será referenciada ao *id* do Telegram.

Figura 23 – Função para receber a mensagem

```
33  # Get the chat_id
34  def action(msg):
35
36      global  chat_id
37
38      chat_id = msg['chat']['id']
```

Fonte: autoria própria (2020).

Nessa parte é feita a função que irá receber a mensagem do Telegram, e, a partir dela, será extraído o parâmetro *id* da conversa que será atribuído à variável *chat_id*.

Figura 24 – Resposta ao usuário

```
20  GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
21
22  GPIO.add_event_detect(button,GPIO.RISING,callback=buttor
23
24  # Send a message when button is click
25  def button_callback(channel):
26
27      global chat_id
28
29      message_button = "Sua campanhia ta tocando"
30
31      telegram_bot.sendMessage (chat_id, message_button)
```

Fonte: autoria própria (2020).

Acima é configurada a variável *button* como entrada e criado um evento para o *rising* do botão, chamando a função *button_callback* de *callback*. A função de *callback* retorna uma mensagem padrão ao Telegram, utilizando o *chat_id* para dispará-la.

3.9 Desenvolvimento do software do Chatbot

Esta seção descreve os principais passos para a construção e a execução do *chatbot* usando o *framework Rasa*[2]*.* Neste trabalho foi utilizada como referência a versão 1.1.8 do Rasa, a mais recente no período de desenvolvimento do mesmo.

3.9.1 Instalação e execução

Para instalar e executar o *chatbot*, siga as instruções:

1. Baixe e instale o *Docker-Compose*[3] em sua máquina.

Código 1 – Instalação do *Docker* e *docker-compose*

```
1  docker -v && docker-compose -v
```

Fonte: autoria própria (2020).

2. Clone o projeto *home-automation*, executando:

Código 2 – Clonando o *git* do projeto

```
git clone https://github.com/juliooaa/myhome-bot.git
```

Fonte: autoria própria (2020).

[2] Para obter mais informações, acesse a documentação oficial do *Rasa* em: https://rasa.com/docs/
[3] Para mais informações, acesse: https://docs.docker.com/compose/install

3. Vá até o diretório da pasta clonada, abra um novo terminal como *root* e execute o seguinte comando:

Código 3 – Subindo o *container*

```
1  docker-compose up
```

Fonte: autoria própria (2020).

A execução desse comando irá baixar automaticamente todas as dependências e instalar o projeto.

3.9.2 Rasa NLU

Nesta subseção será analisado cada arquivo considerado importante do *Rasa* NLU para desenvolver o *chatbot*. Serão tomados como exemplos os arquivos reais do projeto fruto deste trabalho.

3.9.2.1 Arquivo de intents

O primeiro arquivo a se considerar no projeto de uma *chatbot* usando *Rasa* é o arquivo de *intents*. Nele contém os exemplos de treinamento com as possíveis intenções do usuário que dizem como o *Rasa* NLU deve entender as mensagens dele.

Nesse arquivo, as linhas que começam com '##' seguido do nome '*intent*' definem os nomes das intenções, que são grupos de mensagens com o mesmo significado. Assim, o trabalho do *Rasa* será prever a intenção correta, usando um modelo probabilístico, quando os usuários enviarem mensagens novas. O código abaixo mostra exemplos de *intents* do arquivo citado.

Código 4 – Exemplos de *intents*

```
54  ## intent:goodbye
55  - flw
56  - adeus
57  - até mais tarde
58  - falou
59  - tchau
60  - bye
61  - adios
62
63  ## intent:affirm
64  - sim
65  - claro
66  - blz
67  - beleza
68  - ótimo
69  - fechou
70  - vlw
71
72  ## intent:request_status_led
73  - O led está acesa ou apagada ?
74  - Como ta o led ?
75  - O led ai ta acesa ou apagada ?
76  - q jeito q ta o led ?
```

Fonte: autoria própria (2020).

Com esse exemplo, sempre que o usuário entrar com uma dessas palavras ou frases, o *chatbot* identificará qual sua intenção de acordo com o que foi estabelecido.

3.9.2.2 Arquivo config.yml

O arquivo config.yml é o arquivo de configuração que define os componentes que o *Rasa* NLU usará parar treinar o modelo.

Código 5 – Arquivo de configuração do NLU

```
1  language: pt
2
3  pipeline:
4    - name: WhitespaceTokenizer
5    - name: CRFEntityExtractor
6    - name: EntitySynonymMapper
7    - name: CountVectorsFeaturizer
8      token_pattern: (?u)\b\w+\b
9    - name: EmbeddingIntentClassifier
10   - name: DucklingHTTPExtractor
11     url: http://localhost:8000
12     dimensions:
13       - number
```

Fonte: autoria própria (2020).

A chave *language* traz a configuração do idioma no qual o modelo será treinado.

No *Rasa* NLU, as mensagens recebidas são processadas por uma sequência de componentes, executados um após o outro, em um *pipeline*. Há componentes para extração de entidade, classificação de intenção, seleção de resposta, pré-processamento e outros.

O *pipeline* especifica como o modelo NLU deve ser construído e a escolha de um *pipeline* NLU permite personalizar o modelo e ajustá-lo ao conjunto de dados.

Neste projeto usamos o *supervised_embeddings*. A vantagem desse *pipeline* é que seus vetores de palavras serão personalizados para o seu domínio. Por exemplo, em inglês, a palavra 'saldo' está intimamente relacionada à 'simetria', mas muito diferente da palavra 'dinheiro'. Em um domínio bancário, 'saldo' e 'caixa' estão intimamente relacionados e o modelo deve capturar isso. Esse *pipeline* não usa um modelo específico de idioma, portanto, funcionará com qualquer idioma que seja possível *tokenizar* (no espaço em branco ou usando um *tokenizer* personalizado). Abaixo detalhamos a funcionalidade de cada componente do *supervised_embeddings*, utilizado neste trabalho:

- ***WhitespaceTokenizer*:** cria um *token* para cada sequência de caracteres separados por espaços em branco;

- ***CRFEntityExtractor:*** esse componente implementa campos aleatórios condicionados para fazer o reconhecimento de entidades nomeadas.

- ***EntitySynomymMapper:*** mapeia os valores da entidade sinônima para o mesmo valor.

- ***CountVectorsFeaturizer:*** cria uma representação em conjunto de palavras dos recursos que serão extraídos.

- ***EmbeddingIntentClassifier:*** incorpora entradas do usuário e rótulos de intenção no mesmo espaço.

- ***DucklingHTTPExtractor:*** permite extrair entidades comuns.

3.9.3 Rasa core

Nesta subseção será analisado cada arquivo considerado importante do *Rasa Core* para desenvolver o *chatbot*.

3.9.3.1 Arquivo config.yml

No arquivo config.yml também estão as configurações dos componentes do *Rasa Core* que o modelo de gerenciamento de diálogo usará.

Código 6 – Arquivo de configuração

```
3  language: pt
4  pipeline: supervised_embeddings
5
6  # Configuration for Rasa Core.
7  # https://rasa.com/docs/rasa/core/policies/
8  policies:
9    - name: MemoizationPolicy
10   - name: KerasPolicy
11   - name: MappingPolicy
12   - name: "FallbackPolicy"
13     nlu_threshold: 0.8
14     core_threshold: 0.8
15     fallback_action_name: "utter_default"
```

Fonte: autoria própria (2020).

A chave *policies* define as políticas que o módulo *Core* do *Rasa* usará. A seguir, são detalhadas todas essas políticas que foram utilizadas no desenvolvimento do *chatbot:*

- ***MemoizationPolicy*****:** apenas memoriza as conversas nos seus dados de treinamento. Ele prevê a próxima ação com confiança 1.0, se essa conversa exata existir nos dados de treinamento, caso contrário, prevê com confiança 0.0.

- ***KerasPolicy:*** usa uma rede neural implementada no *Keras* para selecionar a próxima ação.

- ***MappingPolicy:*** é usado para mapear intenções diretamente para ações. Uma intenção só pode ser mapeada para, no máximo, uma ação. O *bot* executará a ação mapeada assim que receber uma mensagem da intenção de acionamento.

- ***FallbackPolicy:*** invoca uma ação de *fallback*, se pelo menos um dos seguintes ocorrer: 1. O reconhecimento de intenção possui uma confiança abaixo de *nlu_threshold*. 2. Nenhuma das políticas de diálogo prevê uma ação com confiança superior ao *core_threshold.*

3.9.3.2 Arquivo stories.md

Aqui será tratado a respeito do arquivo de *stories*, através do qual será possível ensinar o *chatbot* a responder às mensagens dos usuários. A comunidade *Rasa* chama isso de gerenciamento de diálogos e é tratada pelo módulo *Core*. O Código 7, abaixo, apresenta exemplos de *stories* deste projeto.

Código 7 – Arquivo de *stories*

```
## case happy path :
* greet
  - utter_greet
* request_status_led
  - action_response_led
* request_resolve_led_on
  - action_resolve_led_on
* request_status_pwm
  - action_response_pwm
* request_resolve_pwm_on
  - action_resolve_pwm_on
* goodbye
  - utter_goodbye
```

```
## case happy path 2:
* greet
  - utter_greet
* request_status_led
  - action_response_led
* request_resolve_led_off
  - action_resolve_led_off
* request_status_pwm
  - action_response_pwm
* request_resolve_pwm_off
  - action_resolve_pwm_off
* goodbye
  - utter_goodbye
```

Fonte: autoria própria (2020).

Os modelos básicos aprendem com dados reais de conversação na forma de “histórias” de treinamento. Uma história é um fluxo de conversa real entre um usuário e um assistente. No código 7, as linhas com *intents* refletem o que o assistente está esperando como entrada do usuário e qual ação deve ter para responder, caso aconteça o que foi esperado por ele. Nesse exemplo, para que se inicie a conversa, o assistente já espera que o usuário o cumprimente, por esse motivo a *storie* se inicia com a *intent greet*. Desse modo, quando o usuário diz ‘olá’ ou ‘oi’, o assistente entende que a sua intenção é cumprimentar, então responde de volta com um ‘olá’ ou ‘em que posso ajudar?’, o que vai depender de como foi programado para responder a uma intenção de cumprimento. Quanto mais fluxos de conversas como essas, mais o *Rasa Core* vai aprender a identificar as *intents* corretas e+ consequentemente executar as ações, o que melhorará cada vez mais seus diálogos com os usuários.

3.9.3.3 Integração com o Telegram

Para fazer a integração do Telegram com o *chatbot*, é necessária a configuração de três parâmetros: *access_token*, *verify* e *webjook_url*.

O *access_token* e o *verify* são gerados pelo *botfather* ao criar o *bot* e o *webhook_url* é a *url* gerada pelo *ngrok* para se fazer o *webhook* entre as aplicações.

Essa configuração é feita no arquivo *config.yml* e abaixo está o código utilizado neste projeto.

Código 8 – Configuração Telegram

```
24  telegram:
25    access_token: "1026442650:AAGb32-TDqSUlDJM3TaMwyqevouM4VNpiA4"
26    verify: "Joaozim_bot"
27    webhook_url: "https://9c83e9f9.ngrok.io/webhooks/telegram/webhook"
```

Fonte: autoria própria (2020).

3.9.3.4 Arquivo actions.py

O arquivo actions.py é onde são estruturadas as ações de respostas ao usuário de uma forma dinâmica.

Código 9 – Arquivo de ações

```
18  class ActionLedOn(Action):
19
20      def name(self) -> Text:
21          return "action_resolve_led_on"
22
23      def run(self, dispatcher, tracker, domain):
24
25          status = tracker.get_slot('led')
26          if status == False:
27              dispatcher.utter_message("led aceso!")
28              r = requests.post('https://1ead8749.ngrok.io/on')
29              SlotSet(led: True)
30          else:
31              dispatcher.utter_message("O led já está aceso!")
32
33          return[]
```

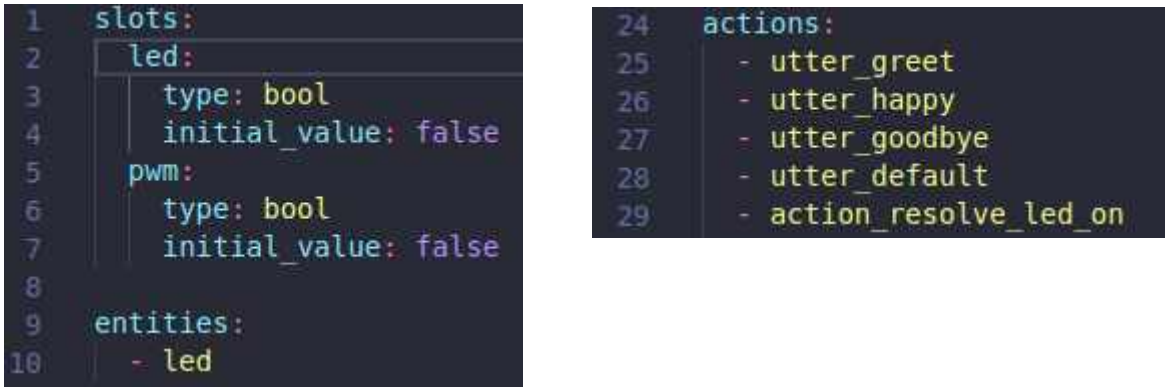
Fonte: autoria própria (2020).

A classe definida no código 9, chamada *ActionLedOn*, faz a tratativa da solicitação da intenção de acender o *led*. Caso o *slot* do *led* esteja *false* (*led* desligado), ela dispara uma mensagem de sucesso, faz um *POST request* à API da *Raspberry* e altera o valor do *slot* para *false*. Caso o *slot* esteja *true* (*led* já está ligado), ele dispara uma mensagem de que o *led* já se encontra aceso.

3.9.3.5 Arquivo domain.yml

Nesse arquivo, são declarados todos(as) os slots, entidades, intenções e ações do *bot*. O *Rasa* utiliza esse arquivo como fonte de declaração de todas as variáveis e ações necessárias para seu funcionamento.

Código 10 – Arquivo de declaração



```
slots:
  led:
    type: bool
    initial_value: false
  pwm:
    type: bool
    initial_value: false

entities:
  - led
```

```
actions:
  - utter_greet
  - utter_happy
  - utter_goodbye
  - utter_default
  - action_resolve_led_on
```

Fonte: autoria própria (2020).

Acima temos alguns exemplos das declarações realizadas no domain.yml. Percebe-se que essas declarações possuem uma estrutura de declaração bem estruturada e devem ser respeitadas.

3.9.4 Deploy na Microsoft Azure

Após criar e testar o *bot* localmente, o próximo passo é implementá-lo no *Microsoft Azure*. Para isso foi criada uma conta na plataforma e feito o *login* através do *cli* do *Azure*.

Após o *login* na plataforma, foi criado um serviço de aplicativo de *bot* que define a fase de implantação do *chatbot*. Nesse serviço, foi usado um modelo ARM, um novo plano de serviço e um novo grupo de recursos.

Para criar o serviço de aplicativo do *bot* que define a fase de implantação é necessário configurar alguns parâmetros de implantação, fornecidos como uma lista de pares chave=valor. São eles:

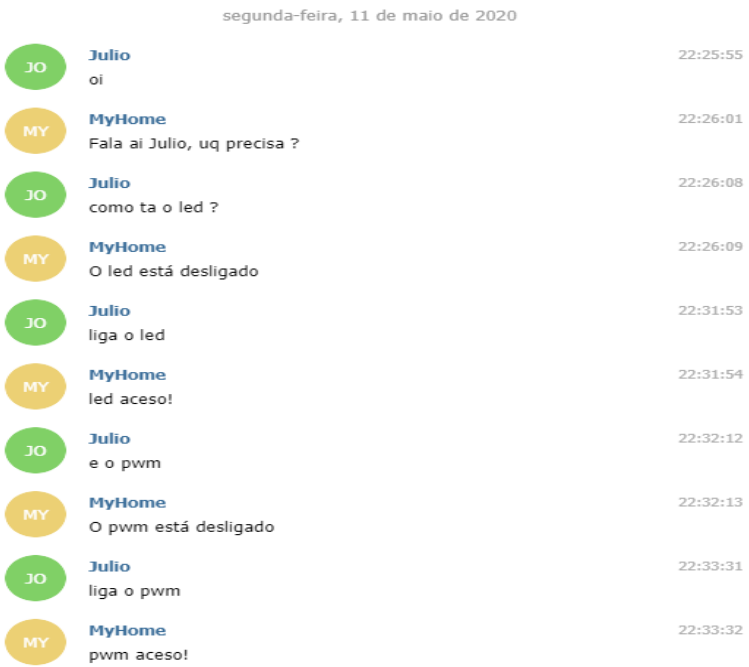
- ***AppId*** – O valor de *id* do aplicativo gerado durante a criação do serviço.
- ***AppSecret*** – Senha do *cli* da *Azure.*
- ***BotId*** – Um nome para o recurso de Registro de Canais do *bot* a ser criado. Deve ser globalmente exclusivo. É usado como a *id* do *bot* imutável.

Por fim, utilizando a própria interface do *Azure*, foram arrastados os arquivos do projeto, configurados o *Docker Compose* e o arquivo YAML, que define todos os serviços a ser implementados, e iniciada a aplicação em nuvem.

4. Resultados e discussões

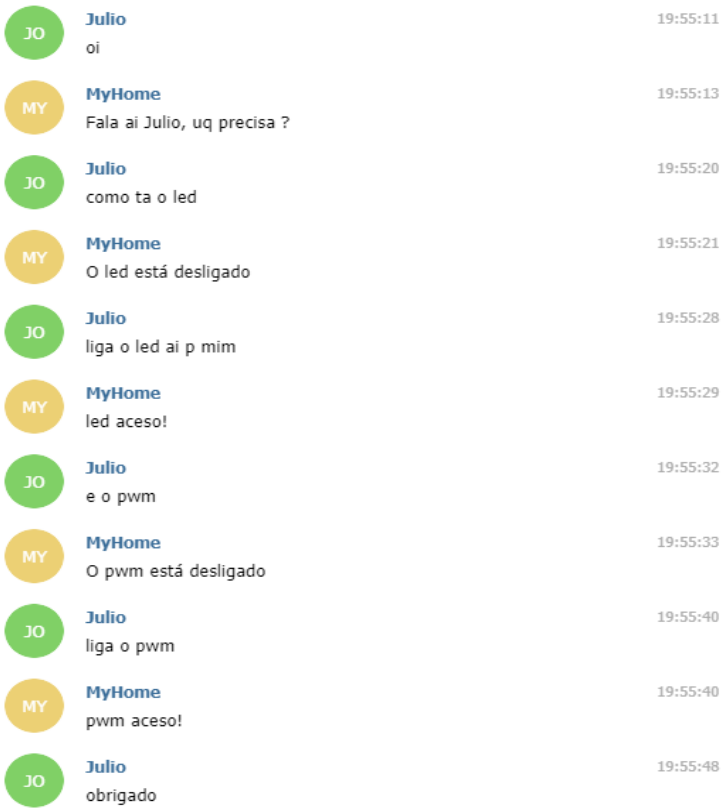
As figuras abaixo mostram conversas com o *chatbot MyHome*, desenvolvido neste trabalho. Notamos que, durante todo o diálogo, ele consegue entender o contexto, as abreviações, as gírias e até as mudanças na forma lógica do diálogo, além de se manter sempre entendido de como estão os estados dos *LEDs* que compõe o projeto, proporcionando, assim, experiências mais amigáveis que os *chatbots* que usam apenas regras pré-treinadas. Também é possível notar que, na transição de uma conversa para outra, ele sempre se mantém no contexto, isso é, no domínio de automação residencial e, mesmo que o usuário lhe faça perguntas fora do contexto, o *chatbot* não permite perder o contexto conversacional.

Figura 25 – Conversa *MyHome*



Fonte: autoria própria (2020).

Figura 26 – Conversa *MyHome*



Fonte: autoria própria (2020).

5. Conclusão e Trabalhos Futuros

Com o avanço dos estudos na área de Inteligência Artificial, os *chatbots* estão cada vez mais presentes e popularizados, sendo em forma de serviço de

atendimento ao cliente, em forma de comunicação e *marketing*, ou até em formas mais avançadas, como a realização de transações financeiras.

O presente trabalho visou o desenvolvimento do *chatbot MyHome* para tratar sobre questões de domínio de automação residencial. Uma das principais vantagens na utilização de *chatbots* baseados em inteligência artificial é que podem reduzir em mais de 50% os gastos de empresas com material humano, visto que mantêm uma conversa simples, intuitiva e contextualizada. Além do mais, o curto tempo de resposta e a capacidade de interagir com milhões de usuários simultaneamente fazem dos *chatbots* uma ferramenta de grande utilidade comercial.

A implementação do *chatbot MyHome* foi realizada por meio das tecnologias de código aberto *Rasa*, *Docker-compose*, *GitHub*, *Ngrok* e *Flask*. Sendo assim, este estudo verificou que é possível optar por uma alternativa de código aberto e grátis, obtendo eficiência no desenvolvimento em contrapartida às soluções proprietárias. Após o desenvolvimento do *chatbot*, pode-se concluir, a partir da primeira tarefa de verificação, que ele conseguiu responder a todas as perguntas realizadas, como demonstrado na seção “Resultados e discussões”.

Ficou comprovado, nos testes de conversão, que o *chatbot MyHome* atingiu os objetivos propostos, uma vez que conseguiu manter o diálogo sem sair do contexto e responder a todas as perguntas realizadas nos testes. Também vimos que ele abordou todos os requisitos funcionais e as regras de negócio definidos neste estudo.

Por último, para trabalhos futuros, faz-se necessário ampliar a base de conhecimento do *bot*, isso é, desenvolver mais *intents*, *stories* e o *template* do domínio. Além disso, também é de suma importância ampliar a base de dados e incorporar nele algum tipo de algoritmo de recomendação.

6. Referências bibliográficas

AL-ZUBAIDE, H.; ISSA, A. A. *Ontbot*: Ontology based chatbot. 2016. Acessado em: 27 de março de 2020.

ALLEN, J. *Natural language understanding*. The Benjamin Cummings Pub. Co., 1995. 654p. Acessado em: 12 de maio de 2020.

CARVALHO, André et al. Inteligência Artificial: Uma abordagem de Aprendizado de Máquina. Rio de Janeiro; Ltc, 2011. Acessado em 09 de novembro de 2020.

AMEUR, R.; HEUDIN, J.-C.. Interactive intelligent agent architecture. In: *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology* (WI-IATW 2006), pp. 331–334. IEEE Computer Society, Washington, 2006. Acessado em: 01 de junho de 2020.

COPPIN, B. *Inteligêncial artificial*. Gen LTC, 2017. Acessado em: 15 de março de 2020.

FRANCONI, E. *Description logics for natural language processing*. 2001. Acessado em: 13 de maio de 2020.

GÓMEZ, A.; ROPERO, J.; LEÓN, C. *A fuzzy logic system for classifying the contents of a database and searching consultations in natural language*. 2006. Acessado em: 28 de maio de 2020.

JACOB, R. J. *User interface*. John Wiley and Sons ltd, 2003. Acessado em: 06 de junho de 2020.

JUSTO, A. V. et al. *Exploring ontologies to improve the empathy of interactive bots*. 2018. Acessado em: 30 de março de 2020.

KAR R.; HALDAR, R. *Applying chatbots to the internet of things*: Opportunities and architectural elements. 2016. Acessado em: 13 de junho de 2020.

POOLE, D. L.; MACKWORTH, A. K. Artificial Intelligence: foundations of computational agents. [S.I]: Cambridge University Press, 2010. Acessado em: 15 de junho de 2020.

PROCEDURAL. Desmistificando a diferença de Emulador e a Maquina Virtual. Disponível em: . Acesso em: 16 jun. 2020.

MANYIKA, J; SNEADER, K. *AI, automation, and the future of work*: ten things to solve for, McKinsey Global Institute, 2018. Acessado em: 25 de maio de 2020.

MERIAT, V. Modelos de Serviço na Nuvem: IaaS, PaaS e Saas. Disponível em: .Acesso em: 12 set. 2020.

NORVIG, Peter; RUSSEL, Stuart. Inteligência artificial. Rio de Janeiro, 2013. Acessada em 09 de novembro de 2020.

TAURION, C. Cloud Computing Computação em Nuvem: Transformando o mundo da tecnologia da informação. Rio de Janeiro: Brasport, 2009. 204p

VELTE, T; VELTE, A; ELSENPETER, R. Cloud Computing: Computação em Nuvem: uma abordagem prática. Rio de Janeiro: Alta Books, 2011.

WITTEN, I. H.; FRANK, E. *Data mining*: Practical machine learning tools and techniques with java implementations (the Morgan Kaufmann series in data management systems). Morgan Kaufmann, 1999. Acessado em: 25 de junho de 2020.

XU, A. et al. A new chatbot for customer service on social media. In: *ACM. Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. [S.l.], pp. 3506–3510, 2017. Acessado em: 27 de junho de 2020.