Centro de Informática
UFPE

Matheus Branco de Siqueira

# Chatbot em Português Para Esclarecimento de Dúvidas Sobre Smartphones


Universidade Federal de Pernambuco
graduacao@cin.ufpe.br
www.cin.ufpe.br/~graduacao

Recife
2021

Matheus Branco de Siqueira

# Chatbot em Português Para Esclarecimento de Dúvidas Sobre Smartphones

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

***Área de Concentração***: *Inteligência Artificial*
***Orientador***: *Luciano de Andrade Barbosa*

Recife
2021

# AGRADECIMENTOS

Agradeço à minha família, especialmente à minha mãe que nunca mediu esfroços para oferecer o melhor para cada um dos seus filhos, nos dar oportunidades e fornecer todo o amor que ela pudesse nos dar. Ela que foi a base para que eu pudesse conquistar tudo até hoje, formou meu caráter e me guiou pelo caminho correto. Obrigado mãe, sua dedicação e amor são inigualáveis!

Aos meus avôs, que já não estão mais entre nós, mas que sempre foram o elo responsável por unir toda a família.

Aos meus irmãos por sempre me guiarem pelo o caminho correto, me apresentar a computação e abrir meus olhos para a realidade.

Agradeço aos meus primos-irmãos, José Carlos e José Augusto, que cresceram comigo, me apoiaram sempre no meu crescimento e hoje fazemos parte do universo da computação.

De uma maneira especial, agradeço à minha namorada e todos da família dela, são minha segunda família! Por todo amor e por me acolherem de maneira tão especial.

Gostaria de agradecer e parabenizar meu orientador, Luciano, pela dedicação e paciência. Um exemplo de profissional e que é indiscutível a qualidade, zelo e gosto no trabalho que exerce.

Pela amizade dos meus amigos, pelos conselhos, noites viradas, risadas, ensinamentos, sentimentos compartilhados e pelos ombros amigos nas horas difíceis. E agradeço em especial a Marcos, quem dividiu a moradia comigo durante 3 anos, superando obstáculos juntos e o responsável por trazer alegria e bom humor para dentro de casa.

Agradeço à UFPE por propocionar uma formação de qualidade e especialmente ao Cin, minha segunda casa por alguns anos. Agradeço a todas as pessoas que fazem parte do centro e, diariamente, zelam por ele ao passo que constroem um centro de excelência.

Muito obrigado a todos!

# RESUMO

Com a grande variedade e oferta de produtos inovadores disponíveis em comércios eletrônicos (*e-commerce*), tornou-se cada vez mais importante o investimento por parte das empresas no setor de atendimento ao cliente. A velocidade com que uma dúvida sobre um determinado produto é respondida influencia diretamente na probabilidade do consumidor realizar a compra do produto. Porém, *e-commerces* funcionam 24 horas por dia, 7 dias por semana, logo, na maioria dos casos, é inviável manter uma equipe de atendimento ao cliente que atenda tal disponibilidade. Neste cenário de atendimento ininterrupto os *chatbots* se destacam. Visamos projetar um assistente em português que seja capaz de responder a dúvidas repetitivas e de fácil resposta através de um canal de comunicação de fácil acesso. Inicialmente foram extraídas informações de *smartphones* de um *e-commerce* e salvos em um repositório local e usado um engenho de busca para indexar as buscas na base de dados. Em seguida, foi treinada uma inteligência artificial com conhecimentos na área de dispositivos móveis e com acesso à base de dados. Finalmente, esse *chatbot* foi conectado ao canal de comunicação o que o torna acessível à maior parte dos usuários. Apesar de carecer de refinamentos, ajustes e mais dados de treinamento, o assistente apresentou resultados satisfatórios e facilidade de acesso aos usuários.

**Palavras-chave:** *Chatbot*, *E-commerce*, Atendimento ao Cliente, Assistente Virtual, Inteligência Artificial.

# ABSTRACT

With the wide variety of innovative products available in e-commerces, investment in customer service by the companies has become increasingly important. The speed which a question about a particular product is answered, directly influences in the probability of the customer to purchase the product. Although e-commerces are open 7x24, so, in most cases, is impracticable to maintain a customer service team with such availability. In these circumstances of uninterrupted service is where the chatbots stand out. We aim to develop an assistant in Portuguese that is able to answer frequent and easy-to-answer questions through an accessible communication channel. Initially, information about smartphones was extracted from an e-commerce and saved to a local repository and a search engine was used to index the searches in the database. Then, we trained an artificial intelligence with knowledge in the field of mobile devices and which has access to the database. Finally, this chatbot was connected to the communication channel which makes it accessible to most users. Despite lacking refinements, adjustments and more training data, the solution presented satisfactory result and a easy to use interface to most users.

**Keywords:** Chatbot, E-commerce, Customer Service, Virtual Assistant, Artificial Intelligence

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ACRÔNIMOS

| | |
|---|---|
| **API** | *Application Programming Interface* |
| **CLI** | *Command Line Interface* |
| **CP** | Características do Produto |
| **DIET** | *Dual Intent Entity Transformer* |
| **IA** | Inteligência Artificial |
| **NLU** | *Natural Language Understanding* |
| **PR** | Perguntas e Respostas dos Clientes |
| **RI** | *Recuperação da Informação* |
| **SRI** | Sistema de Recuperação da Informação |
| **WWW** | *World Wide Web* |

# SUMÁRIO

# 1

# INTRODUÇÃO

Com a grande variedade e oferta de produtos inovadores em comércio eletrônico (*e-commerce*), tornou-se cada vez mais importante o investimento por parte das empresas no setor de atendimento ao cliente. A velocidade com que uma dúvida sobre um determinado produto é respondida influencia diretamente na probabilidade do consumidor realizar a compra do produto. Com isso em mente, as lojas geralmente possuem uma equipe de atendentes para tirar dúvidas sobre seus produtos. Existem dois problemas em contar apenas com atendentes humanos para responder às dúvidas dos consumidores: primeiro, a equipe costuma receber perguntas repetitivas de uma variedade de clientes; segundo, os *e-commerces* estão abertos durante dia e noite, portanto, na maioria dos casos, é inviável manter serviços de atendimento ao cliente 24 horas por dia, 7 dias por semana [1]. Nesse cenário, onde é necessário respostas 24 horas por dia a perguntas majoritariamente repetitivas e de fácil resposta, é onde os assistentes virtuais (*chatbots*) se destacam. *Chatbots* são programas de computadores que conseguem manter uma conversa com um humano usando texto ou Fala em Linguagem Natural (*Natural Language Speech*) [2]. Esses assistentes podem ser usados para reduzir os custos de atendimento ao cliente, uma vez que automatizam o processo de serviços [3].

Atualmente, grandes sites de *e-commerce*, e.g., amazon.com, ebay.com e JD.com, contêm uma ótima fonte de informações sobre um produto, bem como conteúdo gerado por usuários [1]. A Figura 1 mostra a página de um produto da amazon.com.br que contém Características do Produto (CP) e Perguntas e Respostas dos Clientes (PR).

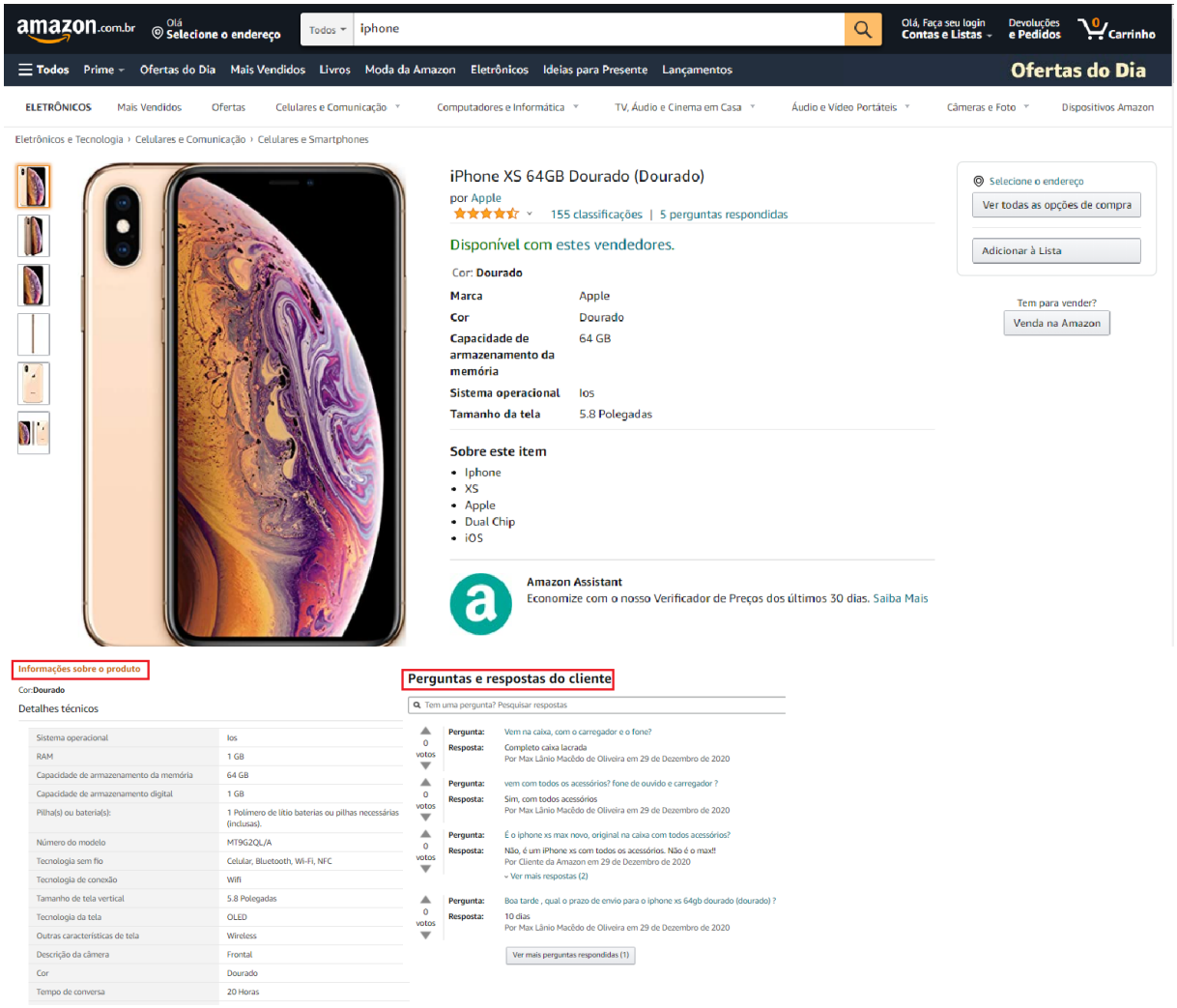



Figura 1: Exemplo de produto na amazon.com.br, contendo as seções com Características do Produto e Perguntas e Respostas dos clientes

Neste trabalho foi implementado um *chatbot* em português para esclarecer dúvidas sobre produtos de uma loja, no nicho de *smartphones*. A escolha desse nicho é atribuída ao fato deste ser um mercado com crescimento acelerado, possuir usuários de várias faixas etárias e conter anúncios de produtos ricos em informações. Para a construção do *chatbot* foi usado o *framework open source Rasa* [4] que permite implementar as funcionalidades básicas de um assistente virtual. Um *crawler* foi desenvolvido para extrair dados sobre produtos - CP e PR - do domínio amazon.com.br e gerar a base de dados com o conhecimento sobre os produtos para o *chatbot*. Para otimizar o tempo de consulta à base de dados, foi configurado e utilizado localmente um servidor do *Elasticsearch* [5].

Esta trabalho está estruturado da seguinte forma: o Capítulo 2 cita algumas das soluções existentes na literatura; no Capítulo 3 são definidos os fundamentos e conceitos usados; o Capítulo 4 explica toda a arquitetura da solução, cada um dos componentes e como ocorre a comunicação entre eles; o Capítulo 5 contém um caso de uso prático da solução; no Capítulo 6 estão expostos os resultados em relação ao desempenho do assistente; por fim, no Capítulo 7 são

discutidos os resultados, contribuições e trabalhos futuros.

# 2

# REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentadas visões gerais de duas soluções da literatura que estão fortemente relacionadas com o propósito deste trabalho.

L. Cui et al. [1], desenvolveu um assistente virtual que pode ser integrado a navegadores *web*, capaz de responder dúvidas e perguntas de um usuário ao navegar na página de um produto anunciado em um e-commerce. A arquitetura do *chatbot* conta com um *crawler* para extração de informações contidas na página do produto, uma base de conhecimentos e uma inteligência artificial dividida em 4 submódulos: (1) *Fact QA*, responsável por responder às perguntas sobre características intrínsecas do produto; (2) *FAQ Search*, utiliza o conhecimento obtido da seção de perguntas e respostas dos clientes, para responder dúvidas mais genéricas; (3) *Opinion Oriented Text QA*, utiliza informações contidas na seção de avaliação do produto; (4) *Chit-chat*, que trata fluxos de conversa que desviam do propósito da solução desenvolvida. Também foi implementado uma *Meta Engine* para orquestrar os submódulos e usar a resposta mais adequada à pergunta do usuário. A solução foi testada em 5 bancos de dados diferentes e apresentou resultados superiores em relação ao primeiro colocado testado nos mesmos bancos de dados.

A. Nursetyo et al. [3], implementou um *chatbot* com foco em e-commerce baseado em Artificial Intelligence Markup Language (AIML) [6] e que usa o *Telegram* [7] como canal de comunicação. A arquitetura da solução contém: (1) uma base de conhecimento que alimenta a inteligência artificial; (2) um módulo para analisar as mensagens e intenções do usuário; (3) um módulo para realizar o casamento de padrões e (4) um componente que implementa a lógica de AIML para processar o conhecimento armazenado e responder às perguntas do usúario. A solução implementou com sucesso um assistente para e-commerce e obteve tempos de respostas médio de 3,4 segundos.

# 3

# FUNDAMENTOS

Neste capítulo são explicados os conceitos, ferramentas e tecnologias usados para a construção da solução proposta. É mostrada uma visão geral de tecnologias que podem ser usadas para implementar *crawlers*, engenhos de busca e assistentes virtuais.

## 3.1 E-COMMERCE

*E-commerce* é uma abreviação de *eletronic commerce*, podendo ser traduzido como comércio eletrônico [8]. Como citado anteriormente, atualmente estes comércios possuem um ótima fonte de informações sobre um produto e uma gama de conteúdo gerado pelos usuários. Todo esse conteúdo pode ser usado para treinar e fornecer uma base de conhecimento para que assistentes conversacionais possa consultá-la, como é explicado ao decorrer deste trabalho.

## 3.2 RECUPERAÇÃO DA INFORMAÇÃO

*Recuperação da Informação* (RI) preocupa-se com a estrutura, análise, organização, armazenamento, busca e disseminação da informacão. Um Sistema de Recuperação da Informação (SRI) é projetado para disponibilizar uma determinada coleção armazenada de itens de informação para uma população de usuários [9]. Um modelo de SRI pode ser dividido em dois módulos: (1) um módulo responsável por extrair os dados de fontes; (2) um módulo responsável por armazenamento e busca de dados.

### 3.2.1 Web Crawler (Scrapy)

O *Scrapy* [10] é um *framework* de código aberto com todos os recursos básicos para a implementação de um *crawler*. Kausar et al [11] define um *web crawler* como um programa/software ou *script* programado para navegar pela *World Wide Web* (WWW) de maneira sistemática e automatizada. A Internet pode ser representada como um grafo direcionado, onde as páginas são os nós e os *hyperlinks* são os vértices, com essa estrutura o *crawler* é capaz de explorar as páginas HTML e salvar em um repositório local. O *Scrapy* é uma ferramenta

extensível e que, além da funcionalidade básica de salvar as páginas, possui a capacidade de extrair e processar os dados de cada página visitada e organizar as informações de diferentes formas de acordo com a configuração. A arquitetura geral do *framework* é ilustrada na Figura 2.

A *Engine* é responsável por controlar todo o fluxo de dados entre os componentes do sistema e acionar eventos quando certas ações ocorrem. As *Spiders* são classes que definem regras de extração, processamento e formatação de dados contidos nas páginas visitadas. O *Downloader* é o módulo responsável por buscar as páginas web e alimentar a *Engine* com elas, que então envia as páginas para as *Spiders*. O *Scheduler* recebe as requisições da *Engine* e enfileira as requisições para devolve-las para a *Engine* futuramente na ordem correta. *Item Pipeline* é o componente responsável por processar as informações após serem extraídas pelas *Spiders*, geralmente realiza tarefas de limpeza, validação e persistência (como armazenar os itens em uma base de dados, por exemplo). *Downloader middlewares* são camadas que permitem definir regras enquanto os dados transitam da *Engine* para o *Downloader* e pelo caminho de volta. *Spider middlewares* são capazes de processar entradas das *Spider* (respostas) e saídas (itens ou requisições).

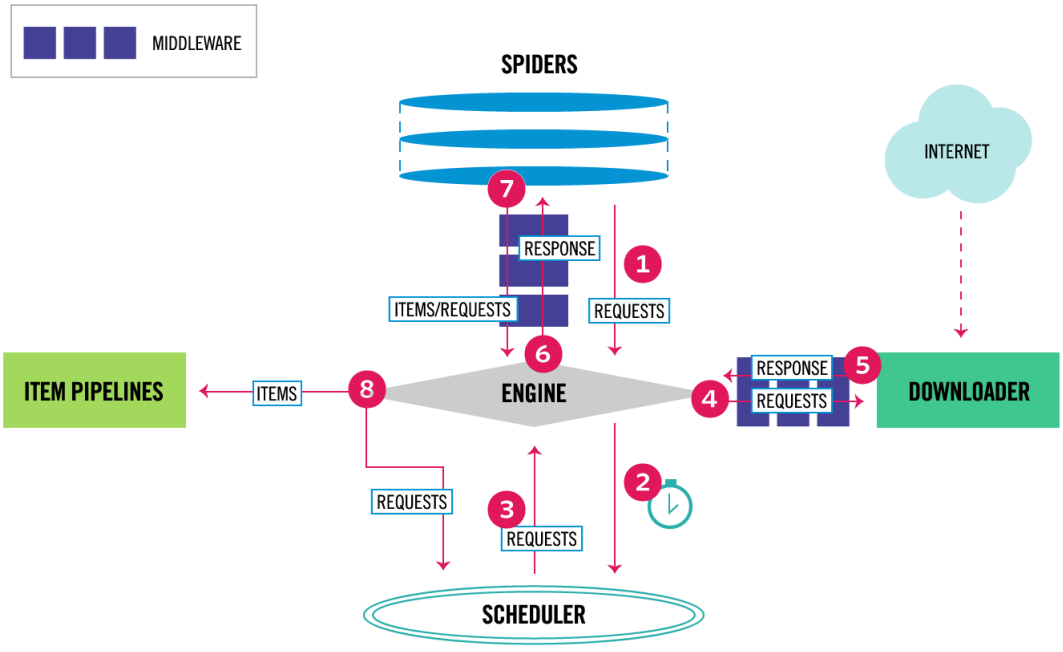



Figura 2: Arquitetura do Scrapy. Disponível em [12]

O fluxo que os dados percorrem é detalhado a seguir:

1. A *Engine* obtém as requisições iniciais através de uma das *Spiders*;
2. A *Engine* agenda as requisições no *Scheduler* e solicita a próxima requisição a ser executada;

3. O *Scheduler* retorna as próximas requisições para a *Engine*;
4. A *Engine* envia as requisições para o *Downloader*, passando os *Middlewares* de download a serem usados;
5. Quando o download é finalizado, o *Downloader* gera a resposta com a página e envia para a *Engine*;
6. A *Engine* recebe a resposta do *Downloader* e envia para a *Spider* para processamento;
7. A *Spider* processa a resposta e retorna os itens extraídos e novas requisições para a *Engine*;
8. A *Engine* envia os itens processado para o *Item Pipelines*, onde os itens passarão por um fluxo configurável de formatação e armazenamento. Em seguida a *Engine* solicita as possíveis próximas requisições da fila;
9. O processo é repetido do passo 1 até que não haja mais requisições no *Scheduler*.

### 3.2.2 Busca (Elasticsearch)

Engenhos de Busca são SRI's capazes de armazenar, organizar e buscar informações de forma a otimizar ao máximo o tempo de consulta. Como definido por L. d. A. BARBOSA [13], os engenhos de busca são responsáveis principalmente por duas tarefas: (1) extrair e armazenar informações da *Web*, e (2) disponibilizá-las para que os usuários possam realizar consultas sobre elas. Barbosa também especifica que essas funções são desempenhadas por dois módulos, respectivamente: o (1) Módulo de Indexação e (2) Módulo de Busca. Esses módulos realizam consultas sobre a base de índices, ou simplemente índice, que contém toda a informação coletada de maneira organizada e consistente. Atualmente os engenhos de busca também são usados para encontrar funcionalidades em sistemas operacionais, como é o caso do *Spotlight* [14] do *Mac OS*. Os engenhos também podem ser usados para organizar informações em um repositório local.

O *Elasticsearch* foi densenvolvido sobre o Apache Lucene [15] e foi escolhido como engenho de busca, especialmente, pelos seguintes motivos: (1) capacidade de realizar buscas próximas de tempo real em um grande volume de dados, (2) ser gratuito e (3) ingestão de dados facilitada. A ferramenta permite armazenar os dados no formato de índices, neste caso os índices são coleções de documentos. Cada documento é armazenado em formato JSON e pode conter valores em formato textual, numérico, geoespacial, booleanos, datas, matrizes de valores ou outros tipos de dados. Adicionalmente, o *Elasticsearch* pode ser usado com o *Kibana* [16], uma ferramenta de visualização de dados que permite gerar vários tipos de gráficos e análises completas sobre os dados armazenados. O *Kibana* também permite, através de sua interface gráfica, gerenciamento dos índices e objetos armazenados no *Elasticsearch*.

## 3.3 CHATBOT (RASA)

*Rasa* é uma plataforma de código aberto (*open source*) que conta com diversos componentes básicos para a implementação de um assistente conversacional. Suporta múltiplos idiomas, garante classificação de intenções, captura de contexto, possibilidade de conexão com diversos canais, Compreensão de Linguagem Natural (*Natural Language Understanding* (NLU)) e conta com uma arquitetura configurável e reúsavel. A arquitetura da plataforma é ilustrada na Figura 3, e os principais componentes dessa arquitetura estão detalhados a seguir.

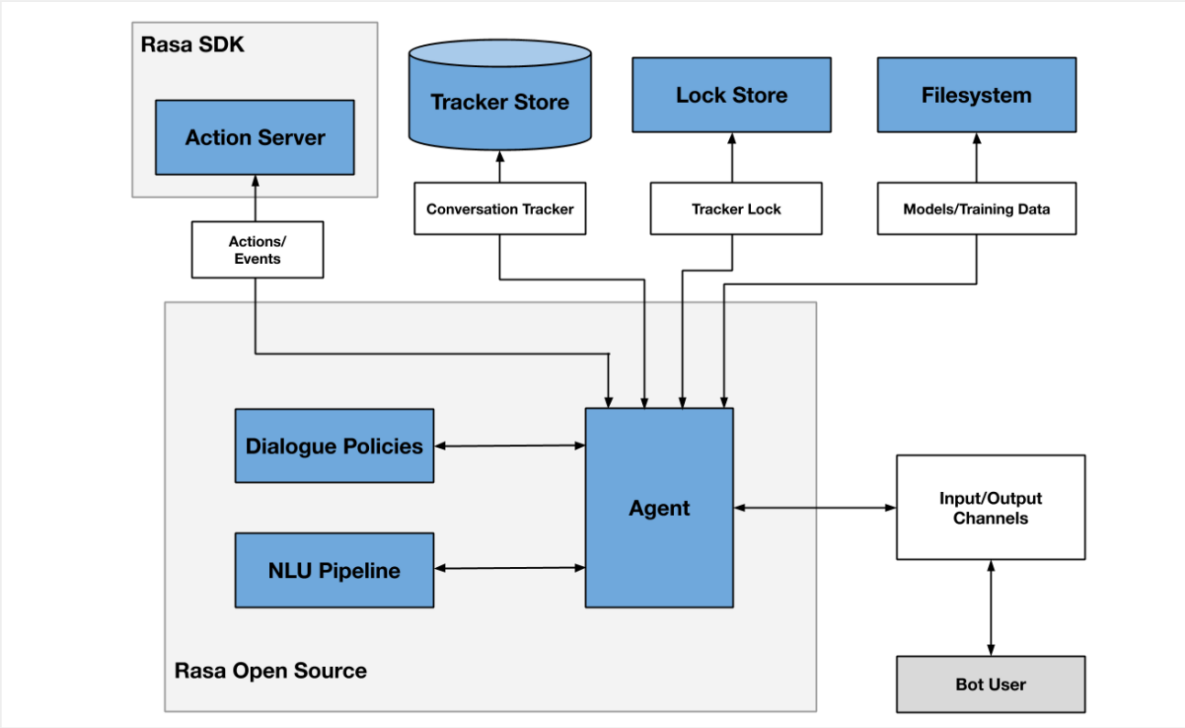



Figura 3: Arquitetura da plataforma *Rasa*. Disponível em [17]

### 3.3.1 *Action Server* (Servidor de Ações)

Este módulo da plataforma instancia um servidor de ações (*Rasa Actions* [18]) capaz de receber requisições REST do servidor do agente (*Rasa server*) e executar qualquer código em Python baseado com a requisição e estado atual da conversa com o usuário. A comunicação entre os servidores ocorre através de *payloads* no formato JSON.

O servidor de ações do *Rasa* permite criar ações personalizadas para atender a demanda da área de atuação do *chatbot*, é possível, por exemplo,realizar requisições a banco de dados, executar um script, realizar chamadas a API's externas e outras funcionalidades.

### 3.3.2 *Tracker Store* (Armazenamento de Rastreador)

Este componente é responsável por manter o histórico das conversas. Por padrão, o histórico é salvo na memória da máquina executando o servidor, então ao reiniciar o servidor do *Rasa* o histórico é perdido. A plataforma oferece suporte para armazenar o histórico de conversas em bancos SQL, *DynamoDB [19], MongoDB [20], Redis [21]* ou outro tipo de armazenamento que pode ser personalizado.

### 3.3.3 *Filesystem* (Sistema de Arquivos)

Encarregado de gerenciar os modelos de aprendizagem de máquina gerados após o treinamento do assistente e realizar a carga do modelo ao iniciar o serviço do servidor do *Rasa*. Este módulo permite carregar arquivos que estão no disco local da máquina, em um servidor ou em um respositório remoto (nuvem).

### 3.3.4 *Dialogue Policies* (Políticas de Diálogo)

As decisões dos próximos passos do *chatbot* são realizadas através de políticas de diálogo [22]. A plataforma do *Rasa* oferece suporte para múltiplas políticas de diálogo, inclusive é possível usar uma combinação de políticas ao treinar o assistente.

### 3.3.5 *NLU Pipeline* (Fluxo NLU)

No *Rasa* cada mensagem enviada pelo usuário é processada por uma série de componentes, isto é chamado de fluxo NLU [23]. É permitido escolher combinações diferentes de componentes de acordo com a necessidade do assistente, área de atuação e regras de negócio.

### 3.3.6 *Input/Output Channels* (Canais de Entrada/Saída)

Os canais de entrada e saída são as formas que o usuário pode entrar em contato com o assistente, são os canais de comunicação ou a interface gráfica. O *Rasa* oferece suporte para vários canais: *Website, Facebook Messeger, Slack, Telegram, Twilio, Google Hangouts Chat*, entre outros. Cada canal oferece componentes que podem ser usados para compor as respostas do *chatbot*, como botões, anexos, animação, dados geoespaciais e vários outros.

### 3.3.7 Treinamento do *Chatbot*

O *Rasa* utiliza YAML [24] como forma unificada e extensível de gerenciar todos os dados de treinamento, incluindo dados de NLU, histórias e regras [25]. Esses dados são divididos em 3 arquivos principais: *nlu.yml, stories.yml* e *rules.yml*. Cada um desses arquivos são explicados adiante, bem como a função dos mesmos no treinamento do assistente.

#### *3.3.7.1 Intenções (nlu.yml)*

Responsável por conter exemplos de palavras ou frases que mapeiam as intenções do usuário. Intenções são palavras ou frases que mapeiam os objetivos do usuário, e.g., caso um usuário digite frases como 'sim', 'claro', 'isso mesmo' ou 'correto', então o assistente deve mapear tais frases para a intenção de afirmar. Por outro lado, se o usuário digita 'não', 'nem', 'não é isso' ou 'de jeito nenhum', a intenção é negar algo.

#### *3.3.7.2 Histórias (stories.yml)*

As histórias (*stories*) [26] são representações de fluxos que a conversa entre o usuário e o assistente pode seguir. Histórias são constituídas basicamente pelos campos:

- *story*: nome da história, arbitrário, não usado no treinamento;
- *metadata*: opcional, contém metadados sobre a história, não usado no treino;
- *steps*: mensagens do usuário e ações que formam o fluxo da história.

#### *3.3.7.3 Regras (rules.yml)*

As regras (rules) [27] seguem uma estrutura semelhante às histórias, e tem uma precedência maior, i.e., sempre que durante o fluxo de uma história o assistente prever um padrão de conversa que corresponde a uma regra especificada, o fluxo atual será interrompido e a regra definida será executada.

### 3.3.8 Domínio

O domínio [28], definido no arquivo *domain.yml*, usa o mesmo formato YAML que os arquivos de dados de treinamento, é responsável por especificar o universo em que o assistente opera, além de configurações para sessões de conversas. Neste arquivo são definidos:

- respostas (*responses*) [29]: são as mensagens que o *chatbot* conhece para se comunicar com o usuário;
- formulários (*forms*) [30]: usados para coletar informações do usuário;
- intenções (*intents*): lista com o nome de todas as intenções do usuário já definidas no arquivo *nlu.yml*;
- entidades (*entities*): lista todas as entidades que podem ser extraídas durante a fluxo da conversa;
- *slots*: variáveis usadas para armazenar contexto e dados fornecidos pelo usuário;

- ações (*actions*): lista com nome das ações definidas no arquivo *actions.py*;
- configuração da sessão (*session_config*): configurações da sessão de conversa atual, como tempo de limite de inatividade e persistência dos valores dos *slots*.

### 3.3.9 Componentes

Conforme explicado na seção 3.3.5, cada mensagem enviada pelo usuário é processada por um conjunto de componentes do *Rasa*. Os detalhes e função de cada um podem ser econtrados na documentação de componentes do *Rasa* [31].

### 3.3.10 Sinônimos

No *Rasa* sinônimos (*synonyms*) mapeiam as entidades extraídas para outro valor definido, diferente do texto literal extraído da mensagem do usuário [32]. Por exemplo, se um usuário referire-se à 'memória RAM', ele pode usar palavras como 'memória' ou 'ram', apesar de diferentes na escrita, todas se referem à mesma entidade e devem ser mapeadas para um único valor textual. O mecanismo de sinônimos do *Rasa* permite realizar este mapeamento.

# 4

# SOLUÇÃO

Neste capítulo são explicados cada um dos componentes da solução, a forma como foram usadas as tecnologias citadas no capítulo 3 e como ocorre o fluxo de comunicação entre os componentes. A Figura 4 ilustra a arquitetura da solução.

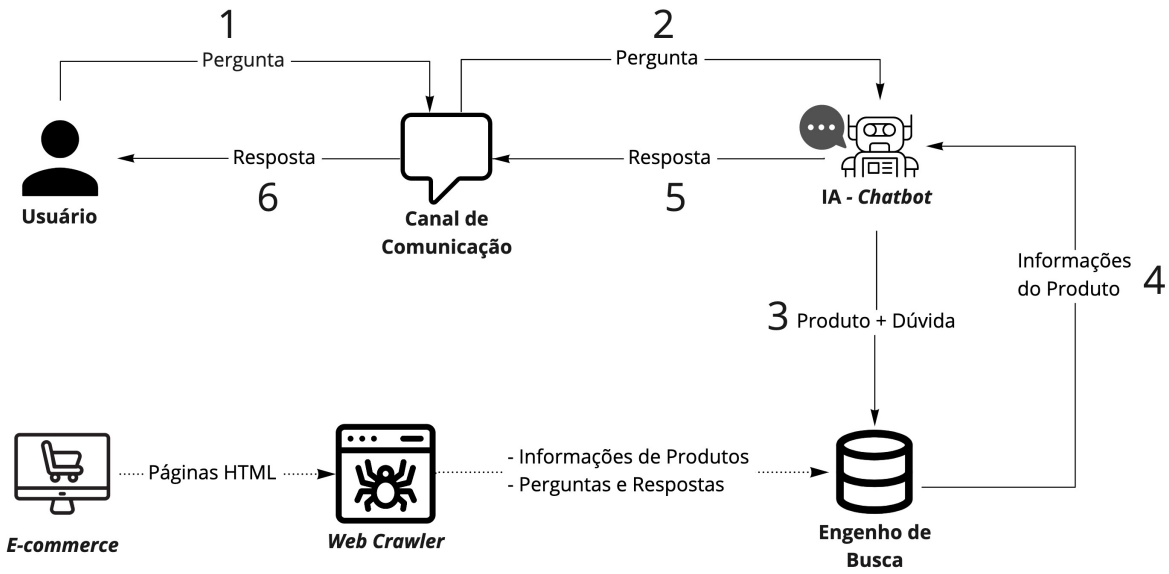



Figura 4: Arquitetura da solução

## 4.1 WEB CRAWLER

Para construir este componente da solução, foi usado o *Scrapy*, detalhado anteriormente na seção 3.2.1. O *Web Crawler* é responsável por navegar pelas páginas do domínio da amazon.com.br, encontrar *smartphones* e extrair CP e PR em cada página visitada. Este módulo também transforma os dados semiestruturados em dados estruturados e salva-os em arquivos do tipo '.ndjson' [33]. O processo de extração dos dados das páginas HTML foi realizado através de duas *Spiders* desenvolvidas:

- *smartphones_spider.py*: responsável pelo processamento de CP, gera o arquivo *products.ndjson*;

- *qa-smartphones.py*: responsável pelo processamento de PR, gera o arquivo *qa.ndjson*.

Para evitar ser bloqueado pelo domínio da amazon.com.br, é necessário que as *Spiders* simulem a requisição de um navegador web. Para isto, foi seguida a solução proposta em [34]. O script *randomAgentMiddleware.py* contém a classe Python *MyUserAgentMiddleware*, esta classe é um *Downloader middleware* do *Scrapy* que seleciona aleatoriamente um *User Agent* de uma lista para realizar cada uma das requisições das *Spiders*. Foi necessário também configurar o *Scrapy* através do arquivo *settings.py* para utilizar a classe *MyUserAgentMiddleware* como *middleware*. A lista contendo os diferentes *User Agents* está no arquivo *settings.py*, no repositório do projeto [35].

### 4.1.1 Estrutura do Arquivo products.ndjson

Este arquivo foi utilizado para armazenar as CP extraídas das páginas visitadas pelo crawler. O arquivo contém várias linhas, cada linha contém um JSON, estruturado conforme a Figura 5, que corresponde às CP encontradas na página HTML visitada pelo crawler. Algumas chaves do JSON podem assumir outros valores ou não estarem presentes em todas as páginas visitadas, como por exemplo a chave 'Tamanho de tela', em alguns produtos o nome do campo dentro da seção de CP pode vir a ser 'Tamanho de tela vertical'. Também é possível que as chaves só existam na descrição de alguns produtos, como é o caso da chave 'Entrada de usuário'.


```
{
    "title": "Smartphone Poco X3 PRO 256gb 8gb RAM – Phantom Black – Preto | Amazon.com.br",
    "product": "Smartphone Poco X3 PRO 256gb 8gb RAM – Phantom Black – Preto",
    "price": "R$ 1.998,90",
    "product_info": {
        "RAM": "8 GB",
        "Capacidade de armazenamento": "256 GB",
        "Pilha(s) ou bateria(s):": "1 Polímero de lítio baterias ou pilhas necessárias (inclusas).",
        "Número do modelo": "M2102J20SG",
        "Tamanho de tela": "6.67 Polegadas",
        "Outras características de tela": "Wireless",
        "Entrada de usuário": "Dial, Microphone, Teclado pequeno, Tela sensível ao toque",
        "Descrição da câmera": "Traseira, Frontal",
        "Formato": "Smartphone",
        "Cor": "Preto",
        "Tempo de conversa": "40 Horas",
        "Tempo de espera com dados": "583 horas",
        "Número de unidades": "1",
        "Modelos compatíveis": "Não se aplica",
        "Peso do produto": "215 g",
        "Dimensões do produto": "16.53 x 7.68 x 0.94 cm; 215 g",
        "Marca": "Xiaomi",
        "Funciona a bateria ou pilha?": "Sim",
        "EAN": "6934177738180, 6934177738333"
    },
    "additional_info": {
        "Dimensões do pacote": "17.4 x 8.6 x 6.2 centímetros",
        "ASIN": "B0919PLV7S",
        "Disponível para compra desde": "23 abril 2021",
        "Avaliações de clientes": "",
        "Ranking dos mais vendidos": "",
        "Descontinuado pelo fabricante": ""
    },
    "url": "https://www.amazon.com.br/gp/product/B0919PLV7S"
}
```


Figura 5: Exemplo de um registro no arquivo products.ndjson

### 4.1.2 Estrutura do Arquivo qa.ndjson

Este arquivo foi utilizado para armazenar as PR extraídas das páginas de perguntas dos clientes visitadas pelo crawler. Cada linha deste arquivo contém um JSON com a estrutura mostrada na Figura 6. Onde as chaves são definidas da seguinte forma:

- 'asin': representa um identificador único para o produto no domínio da amazon;
- 'question': a pergunta feita por algum cliente na seção de PR do produto;
- 'answer': a resposta de um outro usuário ou vendedor para a pergunta;
- 'q_id': identificador único da pergunta no domínio da amazon.

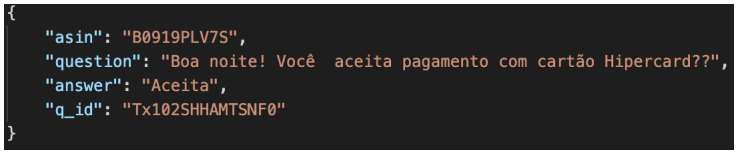



Figura 6: Exemplo de um registro no arquivo qa.ndjson

Pode haver perguntas que ainda não foram respondidas, neste caso o *Crawler* não extrai essas perguntas. Para a solução foi limitado em 50 o número de perguntas extraídas para cada produto.

## 4.2 ENGENHO DE BUSCA

O *Elasticsearch* foi a ferramenta escolhida para realizar o papel de engenho de busca da solução. Após o *Web Crawler* concluir a extração dos dados dos produtos, é realizada a ingestão dos arquivos ('products.ndjson' e 'qa.ndjson') no *Elasticsearch* no formato de dois indíces: *products* e *qa*. Eles contém as informações de CP e PR dos produtos, respectivamente. Sempre que o *chatbot* detectar que a intenção do usuário é consultar informações relacionadas a produtos, esses índices criados serão utilizados para indexar uma busca e retornar as possíveis informações de acordo com um ranking de melhores resultados.

Para dar suporte ao *chatbot* e diminuir a complexidade das consultas ao *Elasticsearch*, foi desenvolvida uma camada de abstração: o módulo *EsHandle*. Responsável por construir as requisições HTTP para consultar os índices do *Elasticsearch* e também formatar a resposta recebida após a requisição. Dessa forma, o assistente pode realizar consultas na base de dados sem preocupar-se com a estrutura de organização dos índices do *Elasticsearch*, usando apenas 2 parâmetros para a consulta: (1) nome do produto e a (2) dúvida do usuário.

## 4.3 IA - *CHATBOT*

A plataforma usada para implementar a Inteligência Artificial (IA) do *chatbot* foi o *Rasa*. As histórias, ou fluxos de conversa, usados para treinamento do modelo foram definidos no arquivo *stories.yml*, a Figura 7 ilustra essas histórias em formato de grafo. Cada caminho do grafo representa um possível fluxo de conversa entre o usuário e o *chatbot*. Os retângulos azuis são possíveis intenções dos usuário que serão preenchidos durante a conversa. Os retângulos brancos, por sua vez, são possíveis respostas que o assistente pode retornar para cada uma das intenções do usuário. Existe a possibilidade de mais de uma intenção ser mapeada para a mesma resposta do *chatbot*, como pode ser notado na resposta *utter_greet*, que contém três possíveis intenções sendo mapeadas para esta resposta.

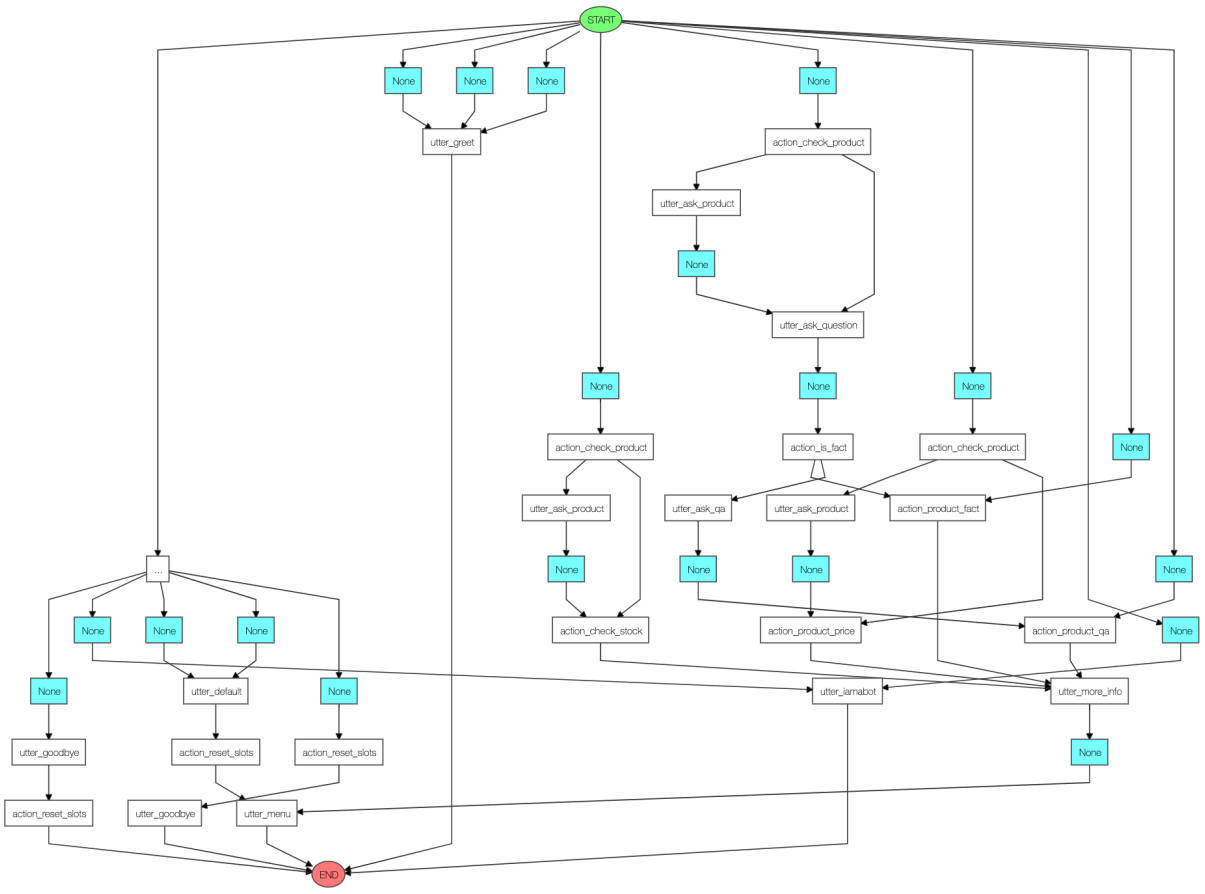



Figura 7: Grafo das histórias usadas no treinamento do *chatbot*

Para comunicação com o servidor do *Elasticsearch* e controle do contexto da conversa, foram desenvolvidas 7 ações no arquivo *actions.py*, arquivo padrão de ações do *Rasa Action*:

- *ActionGetProductFact*: solicita CP para um determinado produto;
- *ActionGetProductPrice*: solicita o preço para um determinado produto;
- *ActionCheckStock*: verifica a disponibilidade de um produto;

- *ActionGetProductQA*: solicita uma informação na seção de PR de um produto;
- *ActionResetSlots*: limpa as variáveis de contexto do *Rasa*;
- *ActionCheckProduct*: identifica se a sessão da conversa atual possui um produto citado como contexto;
- *ActionIsFact*: identifica se a informação extraída da mensagem do usuário é uma CP.

Todas as ações de comunicação com o engenho de busca utilizam o módulo *EsHandle.py* que é instanciado apenas uma vez ao iniciar o *Rasa Actions*. A chamada para cada uma dessas ações é determinada de acordo com o fluxo da conversa entre o assistente conversacional e o usuário. Durante o fluxo, são feitas perguntas para o usuário que permite à inteligência artificial identificar padrões e então extrair marcas e modelos de *smartphones* e também dúvidas relacionadas aos produtos. A marca e modelo do smartphone são salvas no *slot product*, enquanto que dúvidas em relação a CP são salvas no *slot fact* e as perguntas fora do contexto de CP são salvas no *slot question*.

O histórico das sessões de conversa entre o usuário e o assistente são salvos na memória, para o escopo deste trabalho não há interesse em persistir o histórico, então ao final da sessão de conversa, o histórico é excluído. Todos os modelos de NLU treinados foram salvos e carregados localmente.

Em relação às políticas de diálogo do *chatbot*, foram usadas as seguintes políticas: *MemoizationPolicy, TEDPolicy* e *RulePolicy*. *MemoizationPolicy* é usada para memorizar as histórias definidas no arquivo *stories.yml*, essa política é responsável por identificar se a conversa atual corresponde a alguma das representações de histórias usadas para o treinamento. *TEDPolicy* possui uma arquitetura complexa de multi-tarefa, responsável por prever a próxima ação do *bot* e reconhecer entidades no contexto da conversa. Esta política prevê uma sequência de rótulos de entidades através de CRF (Conditional Random Field) [36]. Por último, a *RulePolicy*, realiza predições sobre as regras definidas no arquivo *rules.yml* do conjunto de treinamento, tratando trechos de conversas que devem ter um comportamento fixo.

### 4.3.1 Componentes de NLU

Para que todas as mensagens recebidas sejam processadas de maneira correta, o *Rasa* utiliza o Fluxo de NLU, definido na seção 3.3.5. Uma vez que o assistente proposto neste trabalho tem uma área bem definida, apenas *smartphones*, foi escolhido uma configuração de fluxo NLU que usa apenas os dados de treinamento fornecidos (arquivo *nlu.yml*) e torna as previsões mais assertivas para o domínio em específico, a configuração escolhida está ilustrada na Figura 8. Existem outras configurações de *pipeline* que usam componentes que contém dados prontos de pré-treinamento para determinados idiomas, mas que não tiveram bom desempenho para o propósito desta solução. Mais detalhes dos componentes podem ser encontrados na documentação de componentes do *Rasa* [31].

```yaml
# Configuration for Rasa NLU.
# https://rasa.com/docs/rasa/nlu/components/
language: pt

# # See https://rasa.com/docs/rasa/tuning-your-model for more information.
pipeline:
  - name: WhitespaceTokenizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 100
    constrain_similarities: true
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 100
    constrain_similarities: true
  - name: FallbackClassifier
    threshold: 0.3
    ambiguity_threshold: 0.1
```

Figura 8: Fluxo NLU escolhido para o *chatbot*

#### *4.3.1.1* WhiteSpaceTokenizer

O *WhiteSpaceTokenizer* é responsável por gerar tokens a partir das frases, usando o espaço em branco como separador. Antes de realizar a tokenização por espaço em branco, é realizada uma análise da frase, onde: qualquer caractere que não esteja em 'a-zA-Z0-9_#@&' será substituido por espaço em branco se o caractere satisfaz uma das seguintes condições:

- o caractere é precedido por um espaço em branco: " !texto"→ "texto";
- há um espaço em branco logo após o caractere: "texto! "→ "texto";
- o caractere é está no início da frase: "!texto"→ "texto";
- o caractere está no final da frase: "texto!"→ "texto";

#### *4.3.1.2* RegexFeaturizer

Cria uma representação em formato de vetor a partir mensagem do usuário usando expressões regulares. É o responsável por detectar padrões a partir do conjunto de treinamento e criar uma lista de expressões regulares que possibilita identificar quais mensagens do usuário contém entidades para serem extraídas. No *Rasa*, a extração de entidades a partir de expressões regulares é apenas suportada se for utilizado o *DIETClassifier* ou o *CRFEntityExtractor*.

#### *4.3.1.3* LexicalSyntacticFeaturizer

*LexicalSyntacticFeaturizer* é responsável por criar características léxicas e sintáticas para suportar a extração de entidades. Este componente funciona como um janela que desliza por

cada ums dos tokens gerados a partir da frase do usuário e identifica características do token para permitir extrair entidades. Este componente permite informar se um token está no início ou no final da frase, se está em letras minúsculas ou maiúsculas, se todos os caracteres são dígitos, identificação de sufixos e entre outras características que podem ser usadas para descrever um token.

#### *4.3.1.4* CountVectorsFeaturizer

*CountVectorsFeaturizer* cria uma representação de *bag-of-words* a partir das mensagens do usuário, intenções e respostas. Existem várias opções de configuração deste componente, para esta solução foi utilizada a configuração padrão do *Rasa*.

#### *4.3.1.5* EntitySynonymMapper

Mapeia os sinônimos definidos no conjunto de treinamento, se houver algum. Este componente permite mapear diferentes palavras que possuem o mesmo significado (sinônimos), para uma única representação textual da palavra. Por exemplo, o usuário pode decidir abreviar a palavra "sistema operacional"para "SO", o *EntitySynonymMapper* permite identificar a abreviação da palavra e mapear o valor para "sistema operacional". Dessa forma, a tarefa de extração de entidades é simplificada.

Para melhorar a extração de entidades, uma vez que existem muitas formas de um usuário referir-se a uma mesma CP usando diferentes palavras ou frases. Foram usados o mecanismo de sinônimos do *Rasa*, explicados em 3.3.10. Foram mapeados 8 sinônimos, todos em relação à CP: 'RAM', 'Sistema Operacional', 'Capacidade de Armazenamento', 'Tamanho de tela vertical', 'Cor', 'Peso do produto', 'Dimensões do produto' e 'Marca'. Dessa forma, palavras com mesmo significado, mas com escrita diferentes, são mapeadas para o mesmo valor textual. Isto uniformiza e melhora as buscas nos índices do *Elasticsearch*.

#### *4.3.1.6* ResponseSelector

Prevê a resposta mais adequada a ser retornada para o usuário de acordo com um conjunto de respostas candidatas. O *ResponseSelector* estrutura um modelo de recuperação de resposta (response retrieval), então, insere as entradas do usuário e os rótulos de respostas em um mesmo espaço e segue a mesma arquitetura de rede neural do *DIETClassifier*.

#### *4.3.1.7* FallbackClassifier

Classifica como valor padrão intenções que tiverem previsões com baixo nível de confiança ou com valores ambíguos. Este componente permite definir um limiar mínimo de confiança ao *chatbot* prever as intenções do usuário, i.e., se o *chatbot* prevê a intenção do usuário com uma

confiança abaixo do limiar (*threshold*), então a antiga previsão será desconsiderada e substituída por *nlu_fallback*, uma intenção padrão do *Rasa*.

#### *4.3.1.8 DIETClassifier*

Por último, o *DIETClassifier*, *Dual Intent Entity Transformer* (DIET), é um dos mais importantes para a solução, este classificador é capaz de detectar as intenções e, simultaneamente, extrair entidades das mensagens enviadas pelo usuário. Este classificador utiliza uma arquitetura similar ao *TEDPolicy*. A estrutura do classificador é baseada em um transformador que é compartilhado para as duas tarefas: (1) extração de entidades e (2) classificação de intenções. Uma sequência de rótulos de entidades é prevista através de um CRF [36] realizando uma analogia entre a sequência de entrada dos tokens e a sequência de saída. Para os rótulos de inteção, o transformador tem como saída a expressão completa e o rótulos são inseridos em um único espaço vetorial semântico. É usado a perda de produto escalar (*dot-product loss*) para maximizar as similaridades com o alvo e minimizar as similaridades com amostras negativas [37].

O classificador permite escolher se deve ser realizada apenas a extração de entidades ou apenas a classificação de intenções. Nesta solução foi usada a configuração padrão que permite realizar as duas tarefas simultaneamente. O *Rasa* permite configurar um total de 43 parâmetros do *DIETClassifier*, entre eles podem ser configurados o número de épocas de treinamento, o modelo de confiança, o tamanho do *batch*, taxa de aprendizagem do otimizador, *drop rate* (para regularização), escala de regularização, entre outros parâmetros. Para este trabalho o classificador foi treinado por 100 épocas.

## 4.4 BASE DE DADOS DOS PRODUTOS

Para armezenar as informações de CP e PR, foram criados no servidor do *Elasticsearch* os índices *products* e *qa*. Para analisar os dados foi usado o *Kibana* [16] uma ferramenta de visualização e análise de dados para o *Elasticsearch*. As Figuras 9 e 10 ilustram os índices vistos pela interface gráfica do *Kibana*.

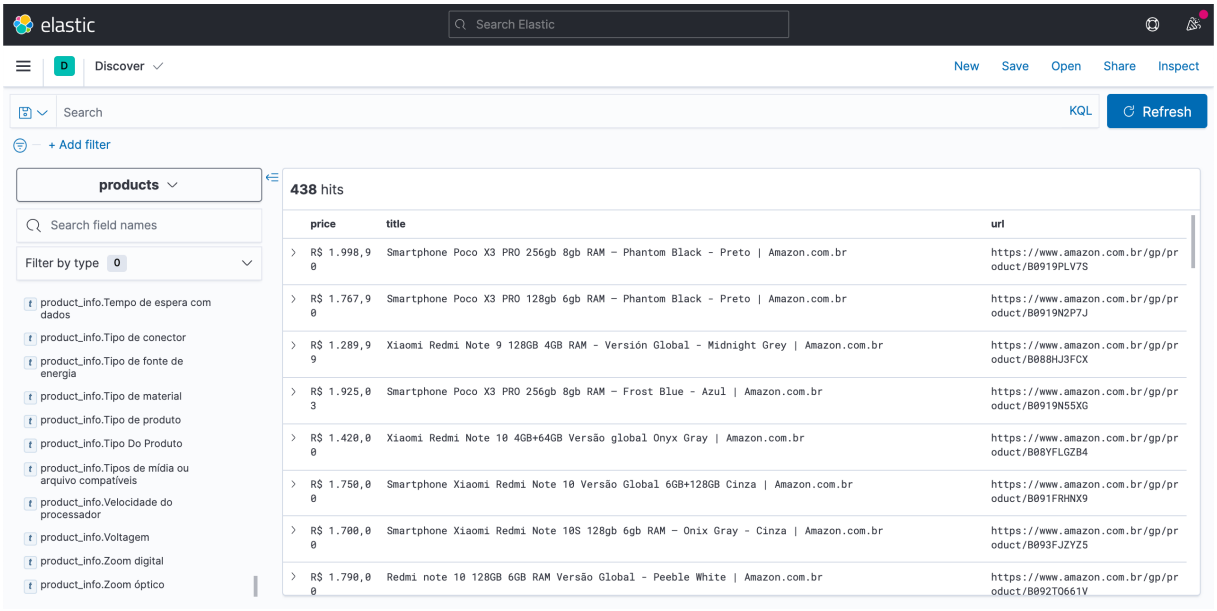



Figura 9: Índice de produtos visto através do *Kibana* mostrando apenas as colunas preço, título e url


Figura 10: Índice de qa visto através do *Kibana* mostrando apenas as colunas pergunta, resposta e ASIN

Pode-se observar que o índice *products* contém 438 produtos e o índice *qa* contém 14.580 perguntas, está é a base que será usada pelo *chatbot* para consultar informações sobre os *smartphones*. Por motivos de escopo do projeto, foi escolhido limitar o tamanho da base em cerca de 450 produtos. Existiram erros durante o processo de extração de informações nas páginas dos produtos, e.g., serviço indisponível e oscilações na rede, que impossibilitaram extrair todos os 450 produtos. Usando o *Kibana* também é possível realizar a construção de visualizações. A Figura 11 mostra o número de produtos inseridos no *Elasticsearch* agrupados por marca do *smartphone* em duas formas de visualização diferentes. Nota-se que o número de smartphones

das marcas Xiaomi, Samsung e Apple, juntos, correspondem a cerca de 65% do total da base de dados dos produtos. De acordo com esta pesquisa de mercado [38], a base de dados corresponde a uma amostra representativa do mercado de *smartphones* no Brasil, que reforça que estas três marcas dominam o mercado brasileiro.

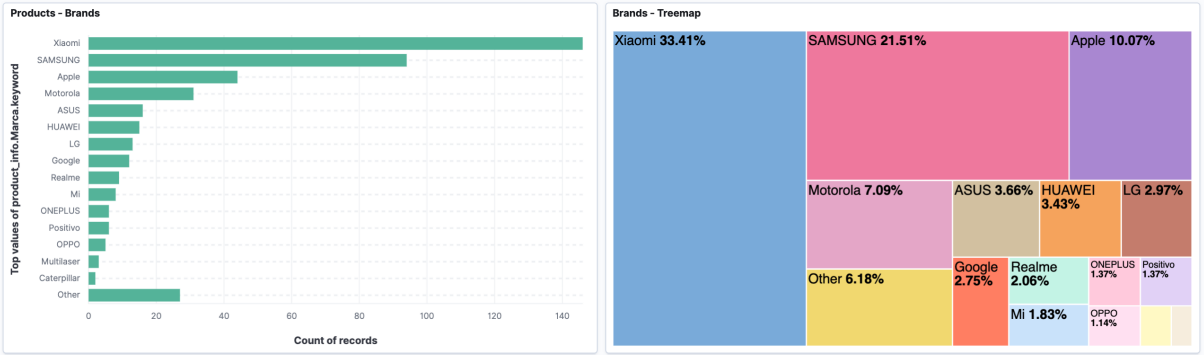



Figura 11: Número de *smartphones* de acordo com a marca

## 4.5 DADOS DE TREINAMETO DO *CHATBOT*

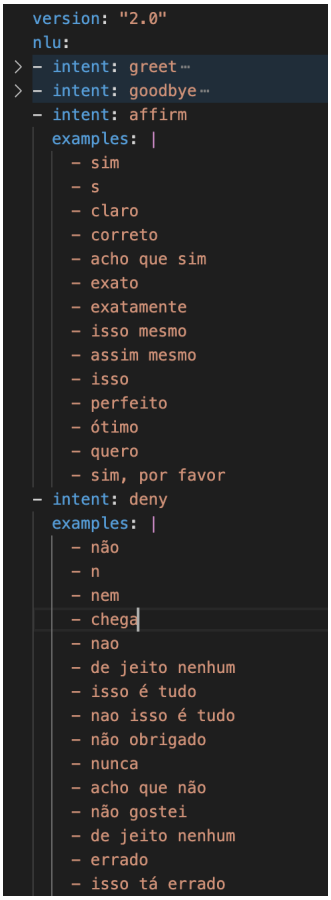
Os arquivos usados para treinar o *chatbot* são:

1. *nlu.yml*: contendo as intenções do usuário, foram mapeadas 16 possíveis intenções, com um total de 309 exemplos de frases ou palavras que são mapeadas para estas intenções;

2. *stories.yml*: contendo 14 histórias, para que o chatbot consiga lidar com diferentes caminhos e contexto da conversa;

3. *rules.yml*: com apenas 4 regras, seguindo a recomendação do documentação do *Rasa* para que o número de regras seja o menor possível e que as mesmas sejam curtas;

As Figuras 12, 13 e 14 ilustram a estrutura do conteúdo dos arquivos da lista. O conteúdo completo dos arquivos pode ser acessado no repositório do projeto [35].



```yaml
version: "2.0"
nlu:
- intent: greet
- intent: goodbye
- intent: affirm
  examples: |
    - sim
    - s
    - claro
    - correto
    - acho que sim
    - exato
    - exatamente
    - isso mesmo
    - assim mesmo
    - isso
    - perfeito
    - ótimo
    - quero
    - sim, por favor
- intent: deny
  examples: |
    - não
    - n
    - nem
    - chega
    - nao
    - de jeito nenhum
    - isso é tudo
    - nao isso é tudo
    - não obrigado
    - nunca
    - acho que não
    - não gostei
    - de jeito nenhum
    - errado
    - isso tá errado
```

Figura 12: Conteúdo parcial e estrutura do arquivo *nlu.yml*

```yaml
version: "2.0"

stories:
- story: user greets and start conversation
  steps:
  - intent: greet
  - action: utter_greet

- story: User ask for help
  steps:
  - intent: ask_help
  - action: utter_greet

- story: User wants to check if a smartphone is in stock
  steps:
  - intent: check_stock
  - action: utter_ask_product
  - intent: inform_product
  - action: action_check_stock
  - action: utter_more_info
  - intent: affirm
  - action: utter_menu

- story: User wants to check if a smartphone is in stock - product setted
  steps:
  - intent: check_stock
  - slot_was_set:
    - product: produto
  - action: action_check_stock
  - action: utter_more_info
  - intent: affirm
  - action: utter_menu

- story: User wants to know factual information
  steps:
  - intent: get_product_fact
  - action: utter_ask_product
```

Figura 13: Conteúdo parcial e estrutura do arquivo *stories.yml*

```yaml
version: "2.0"

rules:

- rule: Say goodbye anytime the user says goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye

- rule: Say 'I am a bot' anytime the user challenges
  steps:
  - intent: bot_challenge
  - action: utter_iamabot

- rule: out of scope
  steps:
  - or:
    - intent: nlu_fallback
    - intent: out_of_scope
  - action: utter_default
  - action: action_reset_slots
  - action: utter_menu

- rule: User deny bot action
  steps:
  - intent: deny
  - action: action_reset_slots
  - action: utter_restart
  - action: utter_menu
```

Figura 14: Conteúdo e estrutura do arquivo *rules.yml*

## 4.6 DOMÍNIO DA SOLUÇÃO

Como citado na seção 3.3.8 o domínio é responsável por definir o universo de operação do *chatbot*. O arquivo *domain.yml* da solução contém: (1) uma lista com todas as intenções do usuário já definidas no arquivo *nlu.yml*, (2) uma outra lista com todas as respostas (*responses*) que o *chatbot* é capaz de dar e (3) uma lista com o nome das ações definidas em *actions.py*. Adicionalmente, para a solução, foram definidas entidades e *slots*, com os propósitos descritos a seguir.

### 4.6.1 Entidades da Solução

Na solução foram definidas 3 entidades:

- *product*: representa um *smartphone* reconhecido através de marca ou modelo durante a sessão de conversa, é o nome do produto que o usuário deseja informações;
- *fact*: uma dúvida relacionada a CP identificada durante a conversa;
- *question*: uma pergunta genérica do usuário, que não se enquadra como CP e que, possivelmente, pode ser respondida consultando a seção de PR do produto.

### 4.6.2 *Slots* da Solução

Foram definidos 5 *slots* para a solução:

- *product*: armazena, durante a sessão, o valor da entidade *product*;
- *fact*: armazena, durante a sessão, o valor da entidade *fact*;
- *question*: armezena, durante a sessão o valor da entidade *question*;
- *is_fact*: booleano indicando se o valor armazendo em *fact* é uma CP ou não;
- *prod_context*: booleano para indicar se o usuário, durante a sessão atual, informou o nome de algum *smartphone* ao assistente.

## 4.7 CANAL DE COMUNICAÇÃO

Como canal de comunicação da solução foi escolhido o *Telegram*, um mensageiro open source, que possui uma interface gráfica intuitiva e uma *Application Programming Interface* (API) - *botfather* [39] - que possibilita a criação de um novo *bot* de maneira simples. A conexão entre o *Telegram* e o servidor do *Rasa* foi feita seguindo a documentação de integração entre o *Telegram* e o *Rasa* [40]. Para realizar a conexão entre o mensageiro e o servidor do *Rasa* executando localmente na máquina na porta 5005, foi necessário tornar a porta publicamente disponível para a internet. Para isto, foi usado o *localhost run* [41], uma aplicação de linha de comando que tem como objetivo permitir o acesso via internet a qualquer porta de uma máquina. Neste caso a porta 5005, padrão do servidor *Rasa*, foi mapeada para a porta 80, padrão para transferência HTTP, através do comando:

```
$ ssh -R 80:localhost:5005 localhost.run
```

# 5

# CASO DE USO

Neste capítulo é demonstrado um caso de uso prático da solução. Para que o *chatbot* esteja funcional, é necessário iniciar o *localhost run* através do script *start-server.sh* e, em seguida, iniciar os servidores do *Elasticsearch, Rasa Actions e Rasa*, com o script *start-services.sh* do repositório do projeto [35]. Qualquer usuário do *Telegram* pode encontrar o *chatbot* criado, basta acessar o aplicativo e pesquisar pelo usuário @Celitobot. A Figura 15 ilustra um caso de uso da solução.


Figura 15: Exemplo de caso de uso da solução

O assistente foi programado para, ao receber um mensagem com intenção de cumprimentar, mostrar um menu para guiar o usuário, este menu contempla três opções:

- Pesquisar smartphone no estoque: verifica se o produto requisitado está na base de conhecimento do *chatbot*;
- Dúvidas sobre um produto: entra no fluxo de resposta de dúvidas sobre produtos;
- Consultar preço: pesquisa de preço na base de conhecimento.

O fluxo de resposta de dúvidas permite que o usuário escolha entre 8 principais características de um *smartphone* que normalmente estão disponíveis na seção de CP de cada página de produto, estas características são:

1. Sistema operacional;
2. Memória RAM;
3. Tamanho de tela;
4. Capacidade de armazenamento;
5. Cor;
6. Peso do produto;
7. Fabricante/Marca;
8. Dimensões do produto.

Adicionalmente, caso a dúvida do usuário não seja referente a nenuma dessas opções, também é possivel a escolha da opção 'Outra', para que o usuário escreva sua dúvida sobre o produto e então o assistente irá consultar a seção de PR do produto para responder da melhor forma à pergunta feita pelo usuário.

# 6

# RESULTADOS

Neste capítulo são apresentados os resultados da avaliação de desempenho do modelo treinado e também explicados os dados usados durante treinamento e teste.

Para analisar e testar as histórias e modelo NLU construídos na solução, foi usado o comando 'rasa test' da *Command Line Interface* (CLI) do *Rasa* que avalia o modelo treinado. Porém, antes de utilizar o comando, foi necessário criar histórias-testes (test-stories) que são capazes de simular a interação de um usuário com o *chatbot*. Estas histórias foram criadas baseadas em possíveis comportamentos do usuário, vale ressaltar que as histórias não cobrem todos os possíveis fluxos de conversa, mas que foram desenvolvidas com o objetivo de generalizar ao máximo as interações entre um usuário real e o assistente. As histórias-testes estão disponíveis no repositório do projeto [35] e foram definidas nos arquivos:

- *test_happy_stories.yml*: contém testes em que o usuário realiza as ações conforme um fluxo esperado e lógico;
- *test_unhappy_stories.yml*: contém testes em que o usuário decide realizar perguntas fora do contexto da sessão de conversa atual.

Com as histórias-testes moldadas, foi usada a técnica de validação cruzada (cross-validation) [42] com 5 *folds* para avaliar o modelo de NLU treinado, através do comando de teste do *Rasa*:

```
$ rasa test --cross-validation
```

Os conjuntos de treinamento e testes, foram divididos em 80%-20%. O conjunto completo de dados de treinamento possui 309 frases de exemplo, os conjuntos de treino e teste divididos durante o *cross-validation* possuem 247 e 62 amostras, respectivamente. Com isto foi possível avaliar as histórias utilizando as histórias-testes para simular as mensagens do usuário.

O comando de testes do *Rasa* tem como resultado matrizes de confusão, histogramas e relatórios em relação à predição de histórias, intenções do usuário e extração de entidades. Para a análise, as funcionalidades dos classificadores foram divididas em 3 tarefas: (1) Previsão de Histórias, (2) Classificação de Intenções e (3) Extração de Entidades. Previsão de Histórias é a capacidade que o *chatbot* tem em prever os próximo passos do fluxo de conversa de acordo as

mensagens e intenções do usuário. Classificação de Intenções é a habilidade do assistente em classificar cada uma das frases ou palavras do usuário em uma das intenções definidas no arquivo *nlu.yml*. Já Extração de Entidades é capacidade do *chatbot* em identificar e extrair entidades contidas nas frases do usuário.

As métricas utilizadas para medir o desempenho de acordo com cada tarefa foram: precisão (*precision*), revocação (*recall*), f1-score e acurácia (*accuracy*). Para a avaliação do modelo de NLU (tarefas de Classificação de Intenções e Extração de Entidades) a base foi dividida entre treino e teste usando a estratégia de *cross-validation*. Os resultados obtidos estão listados na Tabela 1. Também foi gerado o gráfico de erro do modelo de NLU, ilustrado na Figura 16.

Tabela 1: Resultados de acordo com a tarefa exercida pelos classificadores

| **Tarefa** | **Base** | **Precisão (%)** | **Revocação (%)** | **F1-score (%)** | **Acurácia (%)** |
|---|---|---|---|---|---|
| Previsão de Histórias | Teste | 85,06 | 63,47 | 72,70 | 50,00 |
| Classificação de Intenções | Treino | 100,00 | - | 100,00 | 100,00 |
| | Teste | 74,35 | 74,51 | 71,73 | 74,51 |
| Extração de Entidades | Treino | 99,99 | - | 99,99 | 99,99 |
| | Teste | 82,32 | 85,35 | 83,67 | 85,00 |

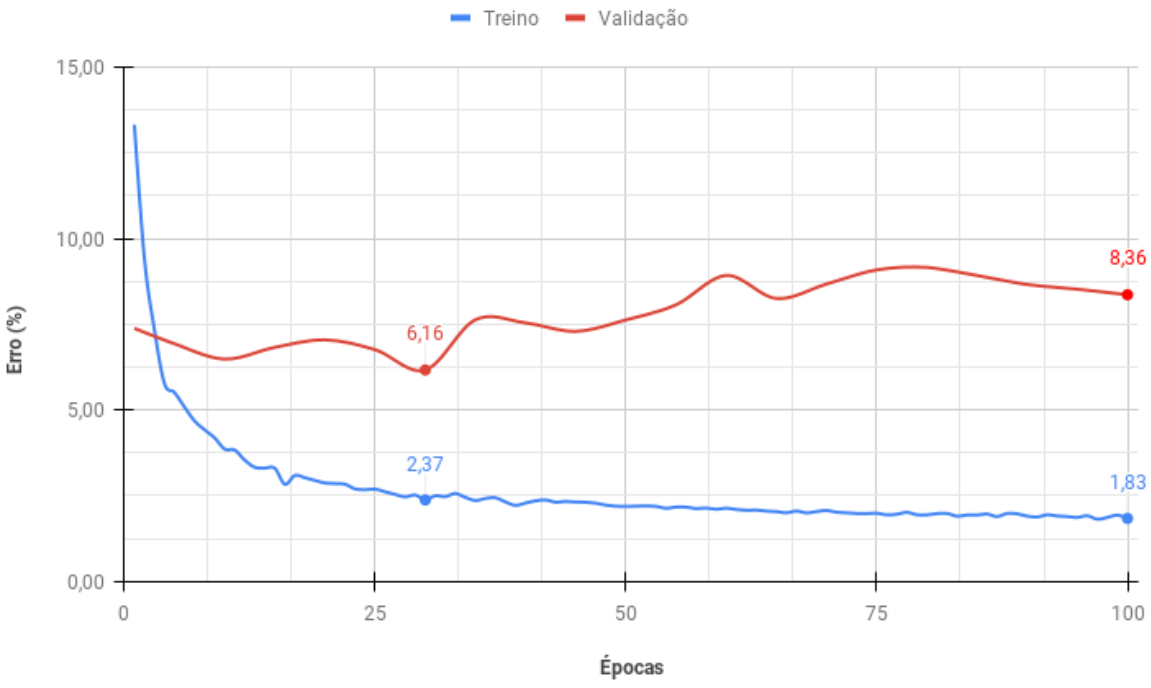



Figura 16: Erro percentual treino x validação

Em relação ao modelo de NLU, através da comparação das métricas na base de treino e de teste, foi observado que o modelo resultou em uma situação de *overfitting*. O gráfico de

erro confirma esta situação, uma vez que, ao aumentar o número de épocas, o erro do conjunto de validação também cresce. Isto é, o modelo adequou-se totalmente aos dados de treinamento mas não foi capaz de prever generalizações de forma satisfatória. A variância do modelo está elevada, ao passo que, o viés está abaixo do esperado. Isto explica as métricas na base de treino possuírem uma diferença significativa quando comparadas às métricas da base de testes.

A Figura 17 ilustra a matriz de confusão para a Previsão de Histórias e ações que devem ser realizadas pelo *chatbot*. Pode-se notar que os resultados estão concentrados na diagonal da matriz, a inteligência artificial consegue prever o fluxo da conversa de maneira satisfatória, acertando as previsões na maioria das vezes. O valor de acurácia em (50%) para Previsão de Histórias na Tabela 1, deve-se ao fato que o *Rasa* é criterioso, e contabiliza apenas como acerto se todos os passos da história forem previstos corretamente. Caso uma entidade seja extraída errada ou um passo previsto incorretamente, o teste da história é marcado como falha.

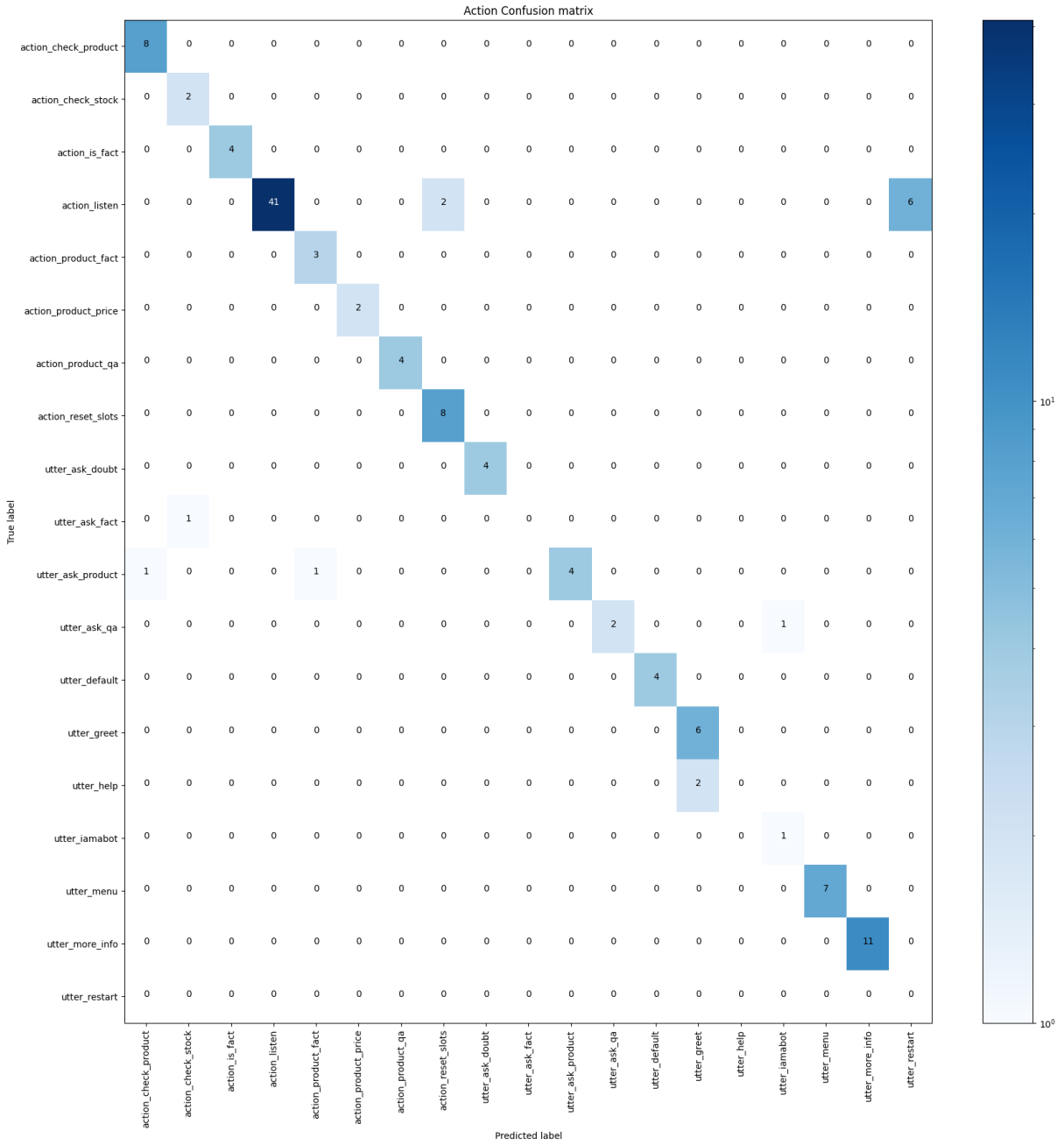



Figura 17: Matriz de confusão da tarefa de Previsão de Histórias

As Figuras 18 e 19 ilustram a matriz de confusão e distribuição de confiança da previsão na Classificação de Intenções, respectivamente. Com exceção das inteções de verificação de stoque (*check_stock*), informar pergunta (*inform_question*) e fora de escopo (*out_of_scope*), o classificador concentrou os resultados na diagonal da matriz, como é o esperado. Quanto à distribuição de confiança, pode-se notar que o classificador também foi capaz de prever, a maior parte das intenções de forma correta com um nível de confiança acima de 0,87. Porém, por falta de refinamento e ajustes no modelo de NLU, foram previstas cerca de 60 intenções de forma incorreta e com um nível de confiança acima de 0,90.

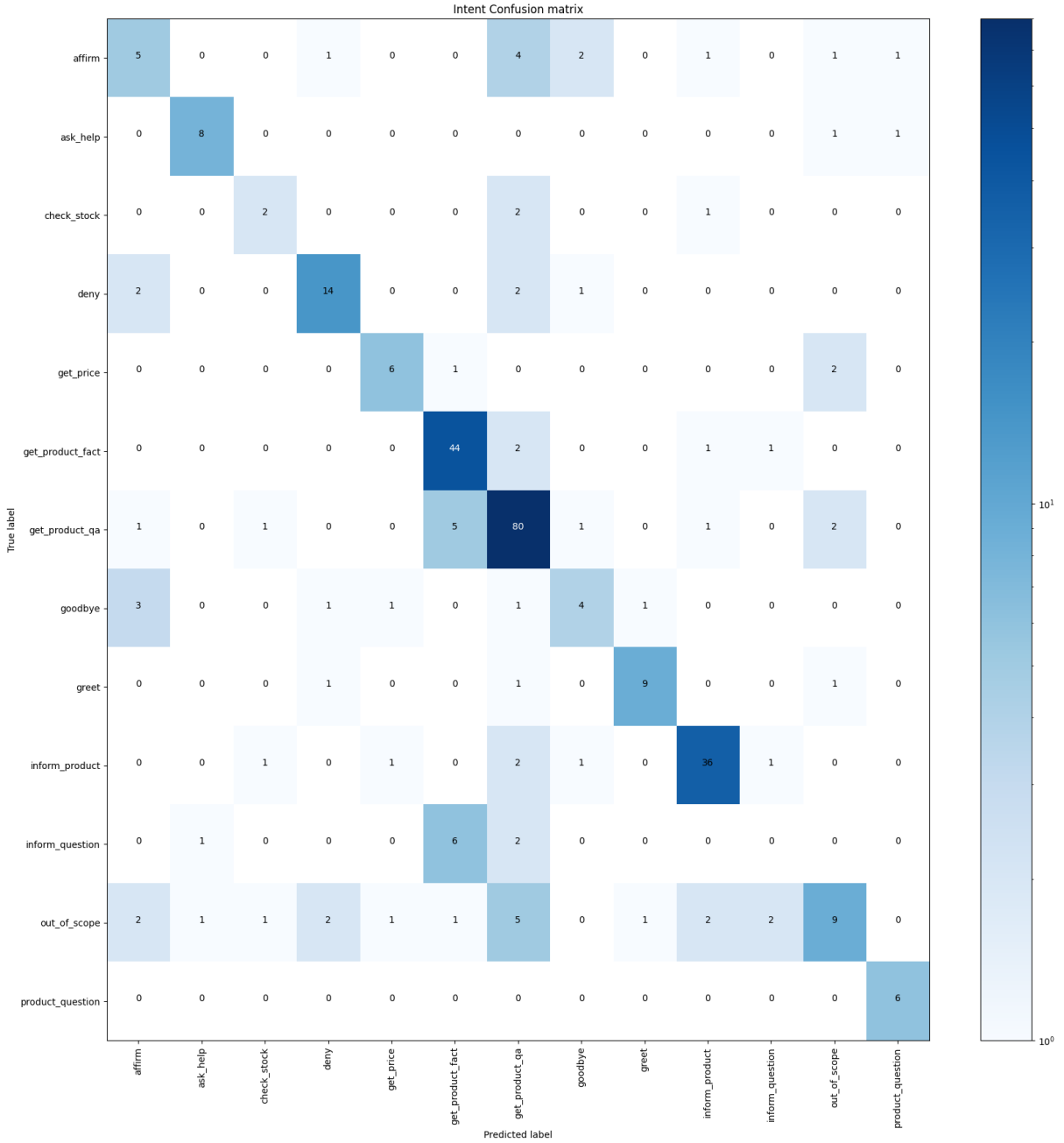



Figura 18: Matriz de confusão da tarefa de Classificação de Intenções

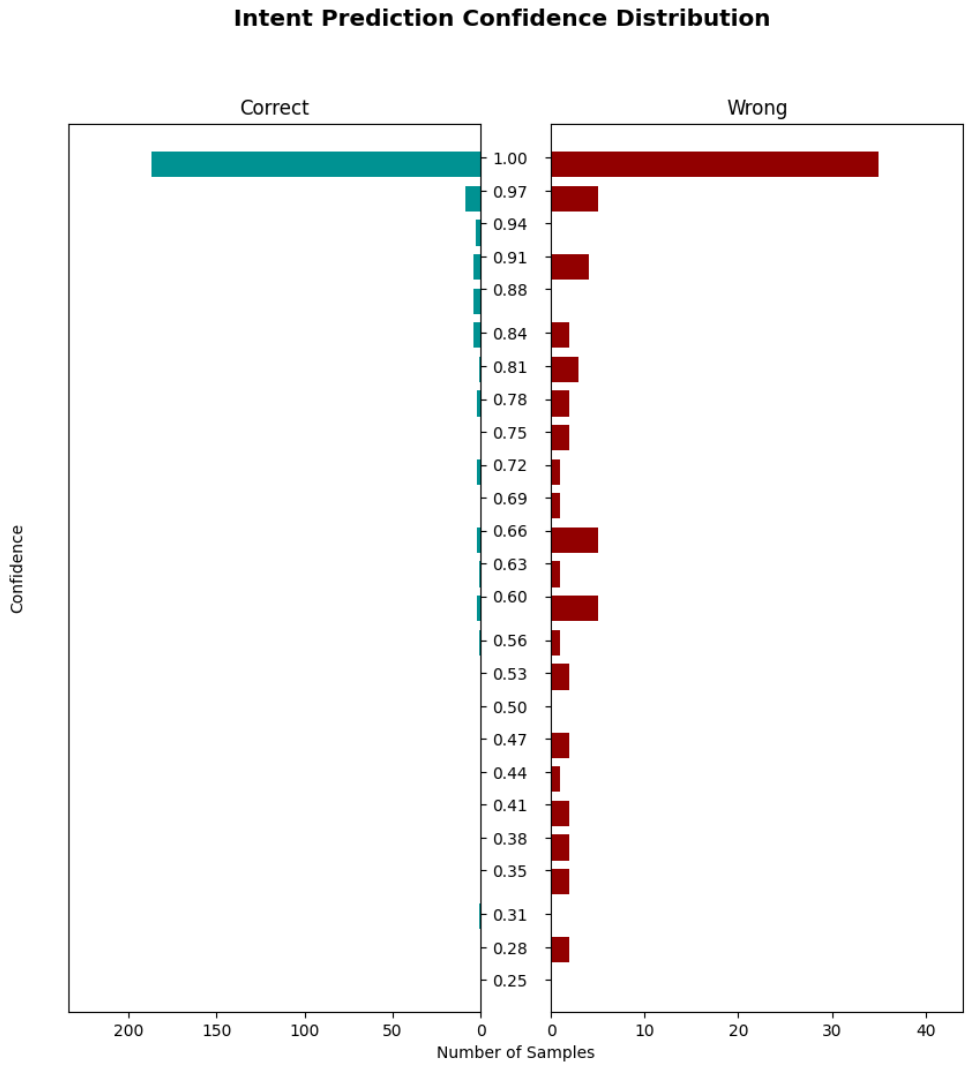



Figura 19: Distribuição de confiança da Classificação de Intenções

As Figuras 20 e 21 ilustram a matriz de confusão e distribuição de confiança na Extração de Entidades. Foi observado de acordo com a matriz de confusão que as entidades foram, em sua maioria, extraídas e associadas corretamente. Os principais erros do modelo foram: (1) confundir frases ou palavras comuns (*no_entity*) com uma entidade do tipo pergunta (*question*) e (2) confundir entidades do tipo pergunta com entidades do tipo produto (*product*). Quanto à distribuição de confiança, as entidades extraídas são, majoritariamente, corretas e possuem um nível de confiança acima de 0,81. Cerca de 42 entidades foram extraídas e associadas de maneira incorreta, com um nível de confiança acima de 0,92. Abaixo de uma confiança de 0,84, as entidades extraídas incorretamente crescem e superam os casos de extrações corretas. Estes problemas podem ser mitigados adicionando-se mais exemplos de entidades do tipo *product* e *question*, por exemplo.

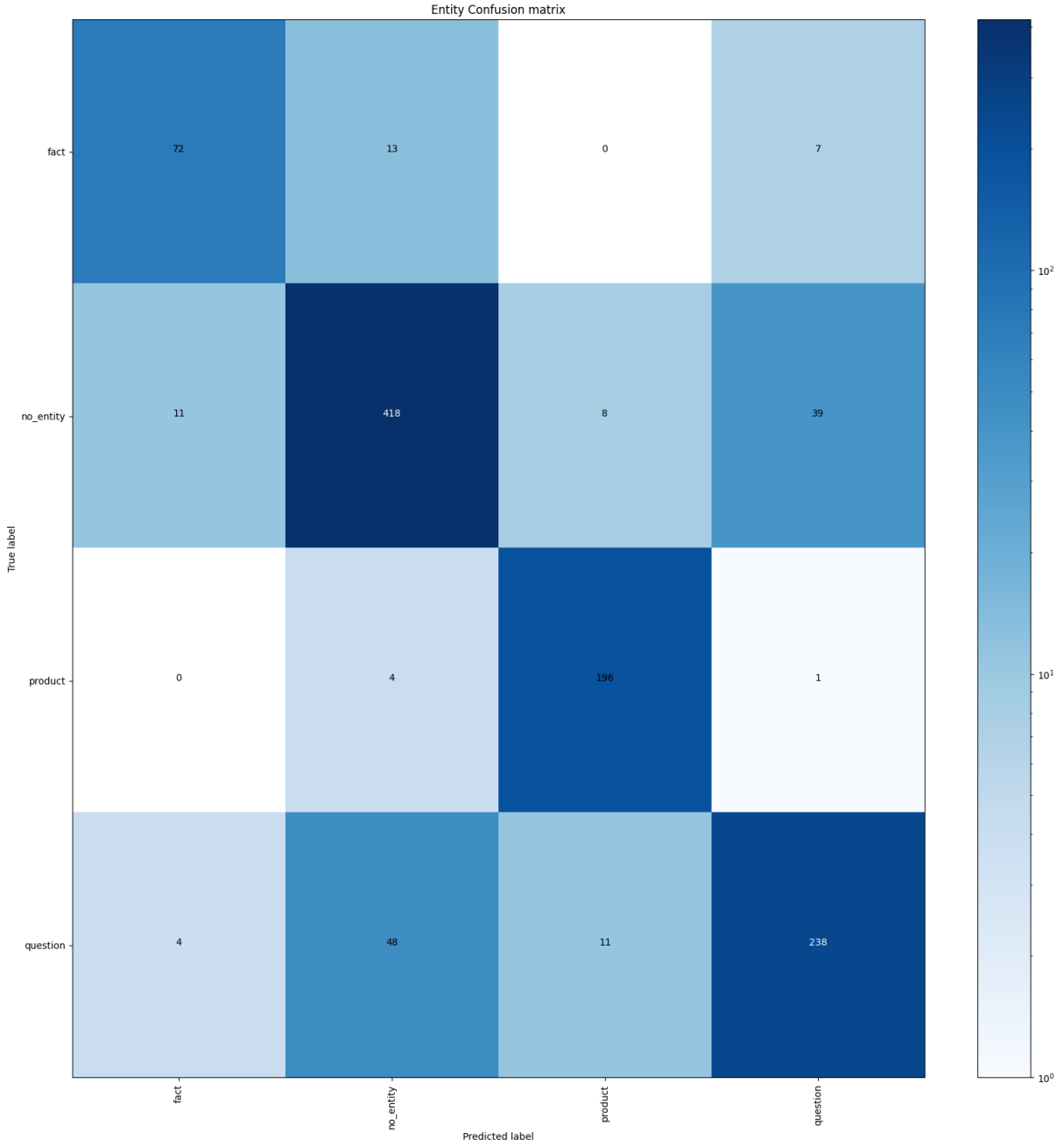



Figura 20: Matriz de confusão da tarefa de Extração de Entidades

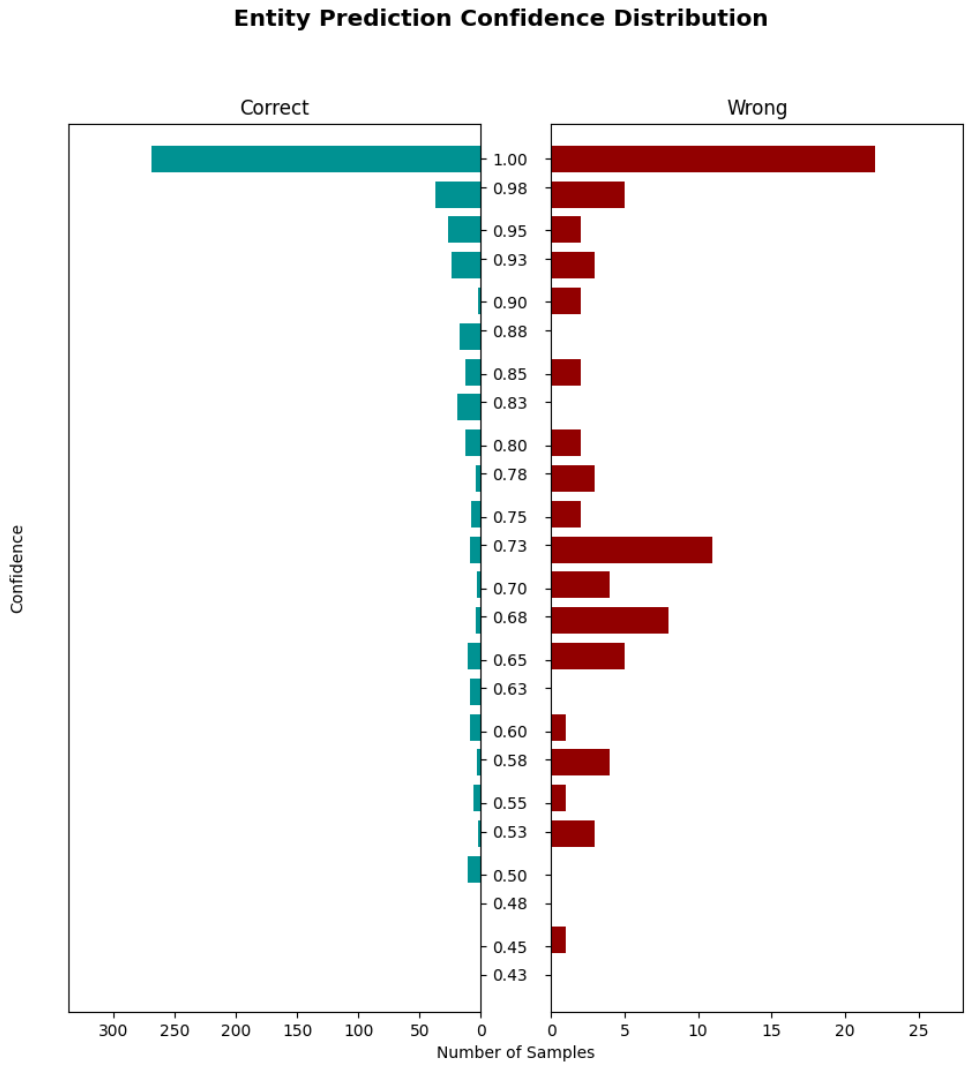



Figura 21: Distribuição de confiança de Extração de Entidades

# 7

# CONCLUSÃO

## 7.1 DISCUSSÃO

De acordo com os resultados obtidos, foi notado que o assistente necessita de um conjunto maior de treinamento, para que possa mapear mais fielmente as intenções do usuário e identificar os próximos passos nos fluxos de conversa. Também é perceptível que, apesar da base de conhecimento ser uma amostra representativa do mercado brasileiro de *smartphones*, é necessário um maior número de produtos para que a solução possa responder a dúvidas relacionadas a uma variedade maior de produtos. Como o modelo está em situação de *overfitting* ele não foi capaz de generalizar bem os casos de teste. Pode-se aplicar algumas técnicas conhecidas para mitigar o problema, como: simplificação do modelo, adicionar mais dados ao conjunto de treinamento, *early stopping* ou regularização.

Em relação à usabilidade, foram realizados testes com usuários que relataram facilidade e praticidade ao usar a solução, devida à interface intuitiva *Telegram* e pelos menus adotados na solução com objetivo de diminuir o atrito entre o usuário e assistente. Por usar o *Elasticsearch* como engenho de busca e os documentos salvos utilizarem uma arquitetura simplificada, o tempo de resposta das mensagens também foi satisfatório, próximo a tempo real. Mesmo que esporadicamente o assistente falhe em responder a pergunta do usuário, ele foi capaz de responder perguntas simples e repetitivas, atingindo o objetivo principal da solução.

A solução proposta por A. Nursetyo et al. [3], obteve tempos de respostas médio de 3,4 segundo. Devido à velocidade de busca do *Elasticsearch*, a solução apresentada neste trabalho obteve tempos os tempos de respostas menores que 1 segundo, próximos a tempo real. Em relação às funcionalidades apresentadas neste trabalho, pode-se contar com menus interativos que não estão presentes no *SuperAgent*, solução proposta por L. Cui et al. [1]. Porém, o *SuperAgent* conta com uma maior variedade de fontes de informações e não depende do *Telegram* como canal de comunicação, uma vez que o canal de comunicação do *SuperAgent* pode ser integrado no próprio navegador *web* do usuário.

## 7.2 CONTRIBUIÇÕES

Esperamos que o conhecimento obtido através da análise deste trabalho possa ser utilizado como guia para implementações futuras de *chatbots* e oferecer suporte para uso das tecnologias utilizadas neste trabalho.

## 7.3 TRABALHOS FUTUROS

Existem melhorias que podem ser realizadas neste trabalho:

- adicionar mais dados ao conjunto de treino do assistente;
- adicionar mais produtos à base de conhecimento;
- tratamento da situação de *overfitting* do modelo;
- melhorar o desempenho *Web Crawler* utilizado para extrair dados do *e-commerce*;
- limpeza dos dados extraídos;
- otimizar os índices do *Elasticsearch*.

# BIBLIOGRAFIA


[1] L. Cui, S. Huang, F. Wei, C. Tan, C. Duan e M. Zhou, "Superagent: A customer service chatbot for e-commerce websites," em *Proceedings of ACL 2017, System Demonstrations*, 2017, pp. 97–102.

[2] S. A. Abdul-Kader e J. Woods, "Survey on chatbot design techniques in speech conversation systems," *International Journal of Advanced Computer Science and Applications*, v. 6, n. 7, 2015.

[3] A. Nursetyo, E. R. Subhiyakto et al., "Smart chatbot system for E-commerce assitance based on AIML," em *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, IEEE, 2018, pp. 641–645.

[4] (2021). "Rasa - Open source conversational AI," endereço: `https://rasa.com/` (acesso em 05/08/2021).

[5] (2021). "O que é Elasticsearch?" Endereço: `https://www.elastic.co/pt/what-is/elasticsearch` (acesso em 05/08/2021).

[6] A. Pradana, G. O. Sing e Y. Kumar, "SamBot-intelligent conversational bot for interactive marketing with consumer-centric approach," *International Journal of Computer Information Systems and Industrial Management Applications*, v. 6, n. 2014, pp. 265–275, 2017.

[7] (2021). "Telegram Messenger," endereço: `https://telegram.org/?setln=pt-br` (acesso em 05/08/2021).

[8] H. G. de Mendonça, "E-commerce," *Revista Inovação, Projetos e Tecnologias*, v. 4, n. 2, pp. 240–251, 2016.

[9] G. Salton e D. Harman, "Information retrieval," em *Encyclopedia of computer science*, 2003, pp. 858–863.

[10] (2021). "Scrapy - Scrapy | A Fast and Powerful Scraping and Web Crawling," endereço: `https://scrapy.org/` (acesso em 16/02/2021).

[11] M. A. Kausar, V. Dhaka e S. K. Singh, "Web crawler: a review," *International Journal of Computer Applications*, v. 63, n. 2, 2013.

[12] (2021). "*Scrapy Architecture*," endereço: `https://docs.scrapy.org/en/latest/topics/architecture.html` (acesso em 12/08/2021).

[13] L. d. A. BARBOSA, "Uma proposta para a atualização da base de dados em engenhos de busca utilizando classificadores," diss. de mestr., Universidade Federal de Pernambuco, 2003.

[14] J. Williams, *Spotlight*. Springer, 2007.

[15] (2021). “Apache Lucene,” endereço: https://lucene.apache.org/core/ (acesso em 12/08/2021).

[16] (2021). “O que é Kibana?” Endereço: https://www.elastic.co/pt/what-is/kibana (acesso em 06/08/2021).

[17] (2021). “Visão geral da arquitetura do *Rasa*,” endereço: https://rasa.com/docs/rasa/arch-overview (acesso em 06/08/2021).

[18] (2021). “Servidor de ações do *Rasa*,” endereço: https://rasa.com/docs/action-server (acesso em 06/08/2021).

[19] (2021). “*DynamoDB*,” endereço: https://aws.amazon.com/dynamodb/ (acesso em 07/08/2021).

[20] (2021). “*MongoDB*,” endereço: https://www.mongodb.com/ (acesso em 07/08/2021).

[21] (2021). “*Redis*,” endereço: https://redis.io/ (acesso em 07/08/2021).

[22] (2021). “Políticas de diálogo do *Rasa*,” endereço: https://rasa.com/docs/rasa/policies (acesso em 07/08/2021).

[23] (2021). “Fluxo NLU escolhido para a solução,” endereço: https://rasa.com/docs/rasa/tuning-your-model (acesso em 07/08/2021).

[24] (2021). “YAML Ain’t Markup Language (YAML™) Version 1.2,” endereço: https://yaml.org/spec/1.2/spec.html (acesso em 09/08/2021).

[25] (2021). “*Rasa - Training Data Format*,” endereço: https://rasa.com/docs/rasa/training-data-format (acesso em 09/08/2021).

[26] (2021). “*Rasa Stories*,” endereço: https://rasa.com/docs/rasa/stories (acesso em 09/08/2021).

[27] (2021). “*Rasa Rules*,” endereço: https://rasa.com/docs/rasa/rules (acesso em 09/08/2021).

[28] (2021). “*Rasa Domain*,” endereço: https://rasa.com/docs/rasa/domain (acesso em 09/08/2021).

[29] (2021). “*Rasa Responses*,” endereço: https://rasa.com/docs/rasa/responses (acesso em 10/08/2021).

[30] (2021). “*Rasa Forms*,” endereço: https://rasa.com/docs/rasa/forms (acesso em 10/08/2021).

[31] (2021). “*Rasa Components*,” endereço: https://rasa.com/docs/rasa/components (acesso em 10/08/2021).

[32] (2021). “Rasa Synonyms,” endereço: https://rasa.com/docs/rasa/nlu-training-data#synonyms (acesso em 16/08/2021).

[33] (2021). “ndjson,” endereço: http://ndjson.org (acesso em 15/08/2021).

[34] (2021). “Programmer Sought - scrapy crawling http returns 503 Service Unavailablec error,” endereço: https://www.programmersought.com/article/72643410189/ (acesso em 15/08/2021).

[35] (2021). “Repositório do projeto,” endereço: https://github.com/mbs8/chatbot-smartphones (acesso em 07/08/2021).

[36] H. M. Wallach, “Conditional random fields: An introduction,” *Technical Reports (CIS)*, p. 22, 2004.

[37] (2021). “Rasa DIETClassifier,” endereço: https://rasa.com/docs/rasa/components/#dietclassifier (acesso em 26/08/2021).

[38] (2021). “Marcas de celulares: qual é a melhor? - DeUmZoom,” endereço: https://www.zoom.com.br/celular/deumzoom/qual-a-melhor-marca-de-celular (acesso em 07/08/2021).

[39] (2021). “Bots: An introduction for developers,” endereço: https://core.telegram.org/bots (acesso em 07/08/2021).

[40] (2021). “Rasa Open Source Documentation - Telegram,” endereço: https://rasa.com/docs/rasa/connectors/telegram (acesso em 07/08/2021).

[41] (2021). “Localhost.run,” endereço: https://localhost.run/ (acesso em 07/08/2021).

[42] M. W. Browne, “Cross-validation methods,” *Journal of mathematical psychology*, v. 44, n. 1, pp. 108–132, 2000.