Universidade do Minho
Escola de Ciências

Carlos António Senra Pereira

Modelação e Prototipagem de *ChatBots*

outubro 2018

Modelação e Prototipagem de *ChatBots*

Carlos Pereira

UMinho | 2018

Universidade do Minho
Escola de Ciências

Carlos António Senra Pereira

Modelação e Prototipagem de *ChatBots*

Relatório de Estágio
Mestrado em Matemática e Computação

Trabalho efetuado sob a orientação do
Professor Doutor Luís Pinto
Professor Doutor José João Almeira

outubro 2018

DECLARAÇÃO

Nome: Carlos António Senra Pereira

Endereço eletrónico: carlos.senra.pereira@gmail.com
Telefone: +351 932 873 658
Número do Bilhete de Identidade: 14848987

Título do Relatório de Estágio
Modelação e Prototipagem de *ChatBots*

Orientadores:
Professor Doutor Luís Filipe Ribeiro Pinto
Professor Doutor José João Antunes Guimarães Dias de Almeida

Ano de conclusão: 2018

Designação do Mestrado:
Matemática e Computação

Universidade do Minho, 31 / 10 / 2018

Assinatura: Carlos António Senra Pereira

Agradecimentos

Aos Professores Luís Pinto e José João Dias pelo desafio, orientação, incentivo, e disponibilidade durante a realização deste relatório.

À empresa *Accenture* pela oportunidade para a realização do Estágio Curricular.

À minha avó Celeste que sempre foi uma fonte de inspiração e um exemplo a seguir.

Ao meus pais e irmãos que sempre me apoiarem e ajudaram.

À minha namorada que esteve sempre ao meu lado, ajudando e apoiando nos momentos difíceis.

Resumo

Genericamente, *ChatBots* são programas que interagem com utilizadores humanos através de linguagens naturais. Os *ChatBots* podem ser criados com objetivos muito diversos, como por exemplo manter uma conversa "inteligente" com um humano, ou prestar serviços em algum domínio concreto, como seja atender chamadas num call-center, reencaminhado-as para um operador. Dada a heterogeneidade dos ChatBots, é importante identificar os seus princípios gerais de organização e de funcionamento. Neste trabalho foi desenvolvida uma proposta original de modelação de ChatBots, que procura identificar estes princípios gerais.

Na modelação que desenvolvemos, os ChatBots são organizados em três componentes principais: a interface com o utilizador, o núcleo e o estado do *ChatBot*. O núcleo é a peça central do funcionamento do *Chatbot*, pois ele é responsável por processar as interações recebidas do utilizador, gerando reações em resposta. O núcleo contém um conjunto de regras que associam funções de reação a padrões linguísticos que, juntamente com o estado do *ChatBot*, determinam a escolha da reação a uma dada interação do utilizador. Para a modelação de padrões linguísticos, desenvolvemos aquilo que designamos por expressões regulares linguísticas (*ERL*). As *ERL* baseiam-se em expressões regulares, envolvendo etiquetas gramaticais, e incluem um mecanismo para extração das palavras-chave de um padrão linguístico, e deram origem a uma *Domain Specific Language*.

A modelação que desenvolvemos permite a criação de um motor geral para a construção de *ChatBots*. Para uma prova de conceito, foi criado o *Diabrete*: um motor geral, *open-source*, escrito em Python, versão 3, com a base de dados em MySQL, que permite a criação de *ChatBots* que seguem a modelação desenvolvida neste trabalho. Na implementação do *Diabrete* recorremos a algumas ferramentas *open-source*, para levar a cabo as tarefas da análise sintática das frases dos utilizadores (biblioteca FreeLing) e para a construção de um classificador baseado em técnicas de *machine learning* para a escolha da reação a apresentar a uma dada interação do utilizador (biblioteca NLTK).

Palvras-chave: *ChatBot*, Modelação, Padrões Linguísticos, *ERL*, Regras, Gerador, *Domain Specific Language*, *open-source*

Abstract

ChatBots are programs that interact with human users through natural languages. ChatBots can be created for very different purposes, such as maintaining an "inteligent"conversation with a human, or providing services in a specific domain, such as answering calls in a call-center, and forward them to an operator. Given the heterogeneity of ChatBots, it becomes important to identify their general principles of organization and operation. In this work, we identify some of these general principles, and develop a new proposal for the modeling of ChatBots.

In the developed model, ChatBots are organized into three main components: the user interface, the core of the *ChatBot*, and the state of the *ChatBot*. The core is the centerpiece of Chatbot's operation, as it is responsible for processing the interactions received from the user, generating reactions in response. The core contains a set of rules that associates reaction functions with linguistic patterns that, together with the state of the *ChatBot*, determine the choice of reaction to a given user interaction. For modeling linguistic patterns, we develop what we call regular linguistic expressions (ERL). ERLs are based on regular expressions involving grammatical tags, include a mechanism for extracting the keywords from a linguistic standard, and have given rise to a Domain Specific Language.

The model that we developed allows the design of a general generator for the construction of ChatBots. For a proof of concept, the *Diabrete* was created. *Diabrete* is a general, open-source generator, written in Python, version 3, with the database in MySQL, which allows the construction of ChatBots that follow the modeling developed in this work. In the implementation of Diabrete, we used some open-source tools to perform the tasks of the user-generated sentences (library FreeLing) and to construct a classifier based on machine learning techniques for the choice of reaction to be presented to a given user interaction (library NLTK).

Keywords: *ChatBot*, Modeling, Linguistic Patterns, ERL, Rules, Generator, Domain Specific Language, open-source

Índice

Lista de Figuras

1 Introdução

Em 1950, Turing publicou *Computing Machinery and Intelligence* [7], onde sugeriu um teste para determinar se um programa consegue simular uma conversa real com um ser humano. Este teste, com o nome *The Imitation Game* (*O Jogo da Imitação*), baseia-se no seguinte:

Seja J o júri humano, A e B os dois participantes. Um dos participantes A ou B é um humano e o outro é uma máquina.
Caso J não consiga distinguir A e B de uma máquina, então a máquina consegue simular um diálogo humano.

Quando o júri da conversa chega à conclusão de que não consegue distinguir o participante da máquina, então diz-se que o programa passou o teste de Turing, por conseguir dialogar com uma pessoa, fazendo-se passar por um humano.

No entanto, em 1966, Weizenbaum com o *ELIZA*, concluiu que tal teste de Turing não é correto. Weizenbaum programou o *ELIZA* com o objetivo de simular uma conversa com um humano, ao responder de forma a colocar outras questões ou frases de terminação, para que este prosseguisse com o diálogo sem perceber que era uma máquina. O método que Weizenbaum implementou foi bastante simples, mas muito eficaz e importante para $ChatBots$, que são programas de computador capazes de manter uma conversa com um humano [5].

Ao reconhecer algumas palavras-chave numa frase, Weizenbaum mapeou tais palavras para um conjunto de respostas pré-definidas, que davam a sensação ao utilizador de continuar com o diálogo, apesar de o programa não estar a satisfazer algum pedido ou a responder com sentido de passar informação às questões colocadas pelo utilizador.

Os princípios base no qual o *ELIZA* assenta são: a identificação de palavras-chave; encontrar o contexto da frase; a escolha das transformações apropriadas ao pedido; a geração de respostas quando não são encontradas palavras-chave; e dar a capacidade de introduzir scripts para ações pré-definidas pelo utilizador, ou ações que não estejam programadas no *ELIZA*.

Apesar de definidos em 1966, estes princípios revelaram-se ser essenciais na construção de um $ChatBot$ moderno. No entanto, outros princípios e técnicas também surgiram mais recentemente, como por exemplo a definição de uma base

de dados de conhecimento e o registo do diálogo com os utilizadores [5]. O registo de sessões passadas entre utilizadores e o *ChatBot* traz benefícios, no sentido em que o *ChatBot* pode escolher uma reação conforme os padrões de diálogo do utilizador e de outras reações escolhidas para outros utilizadores onde o contexto da pergunta é o mesmo.

Na indústria, cada vez mais os *ChatBots* são utilizados como ferramentas para ajudar em tarefas simples, como por exemplo a atender chamadas telefónicas e reencaminhá-las para o operador correto. Este é mais um motivo a reforçar a crescente necessidade de entender os princípios e desenvolver técnicas adequadas para a construção de *ChatBots*.

Neste trabalho, o objetivo principal foi, assim, identificar os princípios gerais de organização e funcionamento dos ChatBots, tendo em vista o desenvolvimento de técnicas que possam auxiliar a criação de ChatBots. O primeiro passo deste trabalho foi o desenvolvimento de um modelo abstrato para ChatBots, que permitisse abarcar muitas das principais características de ChatBots concretos, que serviram de inspiração ao desenho do nosso modelo. Subsequentemente, desenvolvemos uma ferramenta para a prototipagem de Chatbots, que segue o nosso modelo abstrato.

A nossa modelação passa por criar um conjunto de regras que contenham padrões linguísticos a reconhecer pelo *ChatBot*, aos quais são associadas funções de reação. Estes padrões linguísticos permitem a introdução de domínios de pesquisa e alguns sistemas de reescrita, que na implementação darão origem a uma *Domain Specific Language*.

Além das regras, o nosso modelo abstrato é também composto por funções essenciais ao *ChatBot*, responsáveis pelas várias tarefas envolvidas em todo o processamento dos inputs recebidos dos utilizadores. Destacamos a este respeito três funções, chamadas *responde*, *verifica* e *analisa*.

Destas, a função *responde* é a função principal. Esta função começa por invocar a função *analisa*, responsável pela análise sintática dos inputs recebidos, de modo a deles produzir informação num formato compatível com o nosso modelo. A implementação desta função foi feita com recurso à biblioteca externa FreeLing, que se revelou muito apropriada para este efeito.

Após a análise sintática dos inputs, é necessário retirar as palavras-chave e encontrar um contexto adequado para o input. Para isso, foi desenvolvido um modelo de padrões linguísticos que permitirá retirar, sobre um certo domínio, as

palavras essenciais e um contexto mínimo. Esta tarefa é desempenhada pela função *verifica*.

Após a verificação das concordâncias existentes entre um input e os padrões pré-definidos no *ChatBot*, toda esta informação e a informação existente na base de conhecimento do *ChatBot* (que armazena os diálogos anteriores com os utilizadores), é usada para apoiar a decisão da escolha da reação que o *ChatBot* deve produzir ao input. Na nossa implementação, este processo de decisão é realizado através de um classificador baseado em técnicas de *machine learning*.

O resto deste relatório está divido em 3 partes. No Capítulo 2 apresentámos a modelação de *ChatBots*, que desenvolvemos neste trabalho. No Capítulo 3 apresentámos a implementação do motor gerador de *ChatBots* que desenvolvemos e exemplificamos a sua utilização. Por fim, no Capítulo 4 apresentámos as conclusões deste trabalho.

O trabalho relatado neste documento resultou de um estágio curricular realizado na empresa Accenture. O tema inicial previsto para o estágio passava por um estudo mais genérico, versando o uso de técnicas de machine learning no contexto do processamento de liguagens naturais. Contudo, dado o interesse da empresa Accenture na criação de mais conhecimento no domínio de ChatBots, foi acordado que o tema do estágio deveria ser mais focado e centrado na questão do desenvolvimento de ChatBots.

2 Modelação de *ChatBots*

Um *Bot*, diminutivo de *Robot* informático, é um programa que funciona como um agente de tarefas, simulando atividade humana. Existem vários tipos de *Bot*, como por exemplo *Crawlers* e *ChatBots*. Os *Crawlers* percorrem páginas *web* para criar ligações e criar motores de pesquisa, já os *ChatBots* são programas que tentam simular humanamente uma conversa com outro humano, por exemplo, respondendo a questões e pedidos efetuados [5].

No entanto, ao contrário do que se pensava nos anos 60, os *ChatBots* são muito heterogéneos, no sentido que podem realizar muito mais do que uma simples conversa. Podem ser úteis como aplicações que realizam tarefas, dão informações, e muito mais. Ou seja, ao invés de *ChatBots* serem apenas aplicações conversacionais, podem ser vistos como ferramentas para ajudar os utilizadores em tarefas simples ou complexas. Assim, e desde esse período, surgiram novos *ChatBots* desenhados com novas arquiteturas, por exemplo, MegaHAL (Hutchens, 1996), ELIZABETH (Abu Shawar and Atwell, 2002) e ALICE (2007).

Para construir um *ChatBot* são necessários três componentes fundamentais: a interface com o utilizador, o núcleo do *ChatBot* e o estado do *ChatBot*. A interface é o elemento que irá permitir apresentar informação e receber informação do utilizador. O núcleo é o elemento que contém todas as regras e funções que mapeiam padrões que podem associar a pergunta a respostas e decidem qual o comportamento a ter pelo *ChatBot*, em cada momento. Já o estado indicará a base de dados de conhecimento do *ChatBot*, armazena todos as regras e informações pertinentes para o funcionamento adequado do *ChatBot*.

A estrutura geral do modelo de *ChatBots* que construímos neste trabalho é descrita no diagrama da Figura 1.

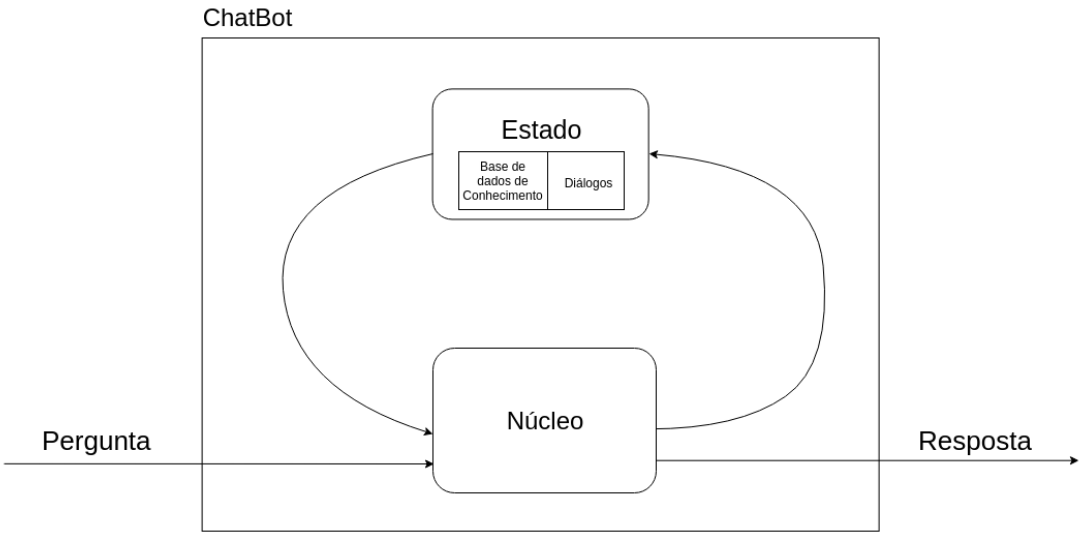

Figura 1: Estrutura geral de um *ChatBot*

Na Figura 1, o retângulo maior representa o *ChatBot*. Este organizado em duas componentes: o estado e o núcleo. A primeira componente, o estado, é constituída pela base de dados do conhecimento e pelos diálogos dos utilizadores. A base de dados de conhecimento tem como objetivo armazenar as palavras-chave e ações tomadas durante o funcionamento do *ChatBot*. Com o armazenamento dos diálogos, o objetivo de criar um histórico sobre cada utilizador para que se tenha um conhecimento sobre cada um e para poderem sugerir reações futuras do *ChatBot* a perguntas de utilizadores. Já a segunda componente, o núcleo, alberga todo o processamento dos pedidos, desde a análise das perguntas recebidas à escolha das reações do *ChatBot*.

Neste trabalho, a componente interface irá permanecer muito simples, podendo a interface ser pensada como um terminal que espera um *input* por parte do utilizador e escreve a resposta gerada pela reação do *ChatBot*. Assim a interface não será nada mais nada menos do que uma plataforma que permite o utilizador escrever perguntas, ou seja dar uma string, e que permita também o *ChatBot* escrever os resultados, ou seja devolver uma string. Na Secção 2.1 será explicado em detalhe a abordagem seguida na modelação do núcleo de um *ChatBot*. Para tornar mais leve a leitura deste documento abreviar-se-á muitas vezes a palavra *ChatBot* por *Bot*.

Na descrição do nosso modelo, tipicamente usaremos a convenção de que: - uma interação, ou seja um input do lado do utilizador, será designada por *pergunta*; - uma interação do lado do *Bot* será designado por *Resposta*. No Capítulo 3,

usaremos também vulgarmente a palavra *pedido* como sinónimo de *pergunta* e *reação* como sinónimo de *resposta*.

2.1 Núcleo

O núcleo ocupa grande parte do funcionamento do *Bot*, e por um motivo natural: ele é responsável por processar as perguntas do utilizador e gerar uma resposta. A Figura 2 apresenta uma descrição diagramática da modelação do núcleo de um *Bot*.

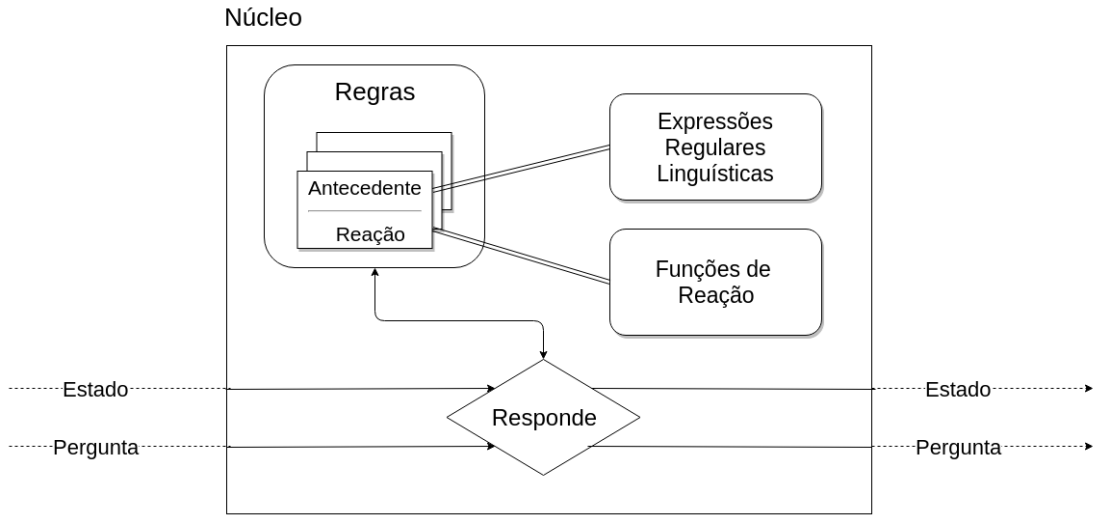

Figura 2: Núcleo de um *Bot*

Seguindo o diagrama da Figura 2, o núcleo é constituído por duas componentes: as regras e a função *responde*.

Uma regra do núcleo é composta por um par (*antecedente*, *reação*). O antecedente é uma *expressão regular linguística* que tem o propósito de modelar padrões, inserindo assim um domínio de pesquisa e aplicação do *Bot*. Já a reação é uma função definida para devolver um resultado, conforme o padrão linguístico ao qual está associada. Um pormenor é de que os antecedentes podem ser repetidos para que várias reações alternativas possam ser usadas perante um mesmo padrão linguístico.

De forma mais precisa, as regras do núcleo de um *Bot* são dadas no nosso modelo pela seguinte gramática:

$$Regras ::= Regra^*$$
$$Regra ::= (ERL,\ FR)$$

onde, ERL significa *Expressão Regular Linguística* e FR *Função de Reação* que descrevemos de seguida.

As *Expressões Regulares Linguísticas* (ERL) são padrões linguísticos construídos a partir de expressões regulares, sobre palavras base e sobre etiquetas gramaticais, que têm por objetivo permitir que o *Bot* possa reconhecer o contexto de uma pergunta e, a partir daí, extrair as palavras-chave da pergunta. Na Secção 2.2 é apresentada a modelação detalhada das *Expressões Regulares Linguísticas*.

As *Funções de Reação* (FR) são as funções executadas para devolver uma resposta ao utilizador. Estas funções podem usar apenas informação interna ao *Bot* ou podem recorrer à chamada de funções externas ao *Bot*, como por exemplo o acesso a outras bases de dados de conhecimento. A assinatura destas funções é a seguinte: $fr : Tokens \mapsto Resposta$. Na Secção 3.5 é exemplificada a implementação de algumas *Funções de Reação*.

Genericamente, seguimos as seguintes convenções nas descrições das funções que se seguem:

- palavras iniciadas com letra maiúscula representam tipos, como por exemplo, ERL, PoS, FR;
- palavras com letras minúsculas, correspondentes as nome desse tipo representam elementos desse tipo;
- palavras reservadas na descrição das funções são escritas a negrito.

Tendo definido o que são as regras do núcleo, é agora necessário definir a componente do núcleo que interagirá com o utilizador e usará as regras do *Bot*. Esta função, de nome *responde*, como visto no diagrama da Figura 2, irá aceitar uma pergunta e devolver uma resposta, atualizando o estado do *Bot*. Então, esta função tem a seguinte assinatura

$$responde : Pergunta \times Estado \mapsto Resposta$$

e abaixo está a sua definição:

$$
\begin{aligned}
&responde(pergunta,\ estado):\\
&\quad pos = analisa(pergunta) && (1)\\
&\quad v = [\ (fr,\ verifica(pos, erl)) \quad \textbf{for}\ (erl, fr) \in Regras\ ] && (2)\\
&\quad (fr,\ tokens) = escolhe(v,\ estado) && (3)\\
&\quad resposta = fr(tokens) && (4)\\
&\quad \textbf{return}\ resposta && (5)
\end{aligned}
$$

Como pode ser visto na definição desta função, *responde* necessita de consultar as *Regras* (e poderia até alterá-las num modelo mais rico). Daí a ligação no diagrama da Figura 2.

Esta função está estruturada em 5 partes: (1) a análise sintática da pergunta; (2) a verificação de todos os padrões linguísticos para detetar as palavras-chaves e contexto; (3) a escolha de uma reação apropriada conforme a verificação feita no passo anterior; (4) a execução da função escolhida no ponto (3); e, por fim, em (5), é devolvida a resposta à pergunta inicial.

Em (1), a análise morfossintática irá devolver uma lista de pares que contém cada palavra que constitui a *Pergunta* e a etiqueta gramatical referente a essa palavra, aquilo que é vulgarmente designado por *Part-of-Speech tag*.

Em (2) é executada a função que verificará se a *pergunta* tem alguma correspondência com os padrões linguísticos definidos nas regras do núcleo do *Bot*, extraindo os elementos essenciais da *pergunta*, que designamos por *tokens*. Esta fase da verificação será explicada mais abaixo com mais detalhe, devido à sua complexidade.

Após verificar as correspondências, em (3), utilizando algoritmos diversos e a(s) base(s) de conhecimento integradas no estado do sistema, será escolhida a melhor e mais apropriada reação, que trará agregada os *tokens* necessários à sua execução. Esta escolha além de utilizar o conhecimento prévio e os diálogos passados, poderá também utilizar outros métodos, como por exemplo *machine learning*, para que quando existam várias correspondências, ou nenhuma correspondência, o resultado obtido seja tão próximo quanto possível da reação esperada pelo utilizador. Na Secção 3.6 é explicada a implementação destes métodos.

Escolhida a função apropriada, o próximo passo é a execução da mesma. Ou seja, em (4), é executada a função com os *tokens* extraídos e o estado atual. Além

de devolver a resposta, o estado também será atualizado, em particular com o registo da interação entre o *Bot* e o utilizador.

Por fim, em (5), será devolvido o resultado obtido na parte anterior.

Analisemos agora com mais precisão a parte (2) da função *responde*. Nesta parte, a função $verifica$ é chamada para cada regra do núcleo. Esta função é responsável por encontrar correspondências entre a *pergunta* e as ERL das regras. A função aceita como entrada a pergunta, já com as etiquetas gramaticais que resultaram da análise sintática (*pos*), e devolverá um dicionário com os *tokens* da correspondência encontrados. Assim sendo, a função tem a seguinte assinatura

$$verifica : Pos \times ERL) \mapsto \perp | Dicionário$$

onde *Dicionário* é constituído pelo par chave-valor (*id*, *pal*)A definição da função *verifica* é a seguinte:

$$\begin{aligned}
&verifica\ (pos,\ erl) : \\
&\quad r = \{\ \} && (1)\\
&\quad \textbf{for}\ (pal, tag, id) \in erl : && (2)\\
&\qquad \textbf{if}\ search\ (pal,\ tag,\ pos) \neq \textbf{empty} && (3)\\
&\qquad\quad \textbf{then}\quad r\ +=\ (id \hookrightarrow pal) && (4)\\
&\qquad\quad \textbf{else}\ \perp && (5)\\
&\quad \textbf{return}\ r && (6)
\end{aligned}$$

A definição e explicação das ERL (*Expressão Regular Linguística*) é efetuada com na seguinte detalhe secção. No entanto, para se entender o funcionamento desta função *verifica*, apresentamos desde já, sucintamente, a estrutura geral das ERL. As ERL são sequências de triplos da forma

$$(\ pal,\quad tag,\quad id\ ) \text{ ou } (\ pal,\quad tag,\quad \_\ )$$

onde *pal* é uma *word pattern*, grosso modo uma expressão regular sobre palavras, a *tag* é uma *tag pattern* (uma expressão regular sobre etiquetas gramaticais) e a terceira componente dos triplos é ou um identificador *id* (caso interesse registar a palavra) ou o símbolo _, a denotar que não será importante registar essa palavra.

Voltemos agora à descrição da função *verifica*. Em (1) é inicializado o dicionário r, que no final vai conter os *tokens* da verificação. Em (2) é inicializado um ciclo que percorre cada um dos elementos da ERL da entrada, com o qual é testado

em (3) se existe alguma correspondência com a palavra recebida em *pos*. Se sim, o dicionário r é atualizado com o par chave-valor (*id*, *pal*) (em 4), prosseguindo com a próxima componente da ERL, caso contrário a função *verifica* termina, sinalizando (em 5) que a palavra recebida não corresponde à ERL.

2.2 Expressões Regulares Linguísticas

As expressões regulares linguísticas (ERL) constituem a linguagem que usaremos na nossa modelação para definir os padrões linguísticos para os quais um *Bot* terá reações associadas, através das suas regras. Embora tipicamente os *Bots* sejam construídos para interagir com o utilizador dentro de um certo domínio de aplicação, estes domínios, em geral, são muito diversos, e torna-se importante que a nossa linguagem de expressões regulares linguísticas seja tão expressiva e tão flexível quanto possível, de modo a que a nossa modelação possa abarcar uma grande heterogeneidade de Bots. As ERL são baseadas em expressões regulares, envolvendo possivelmente etiquetas gramaticais, e permitem indicar quais são as palavras-chave que importa extrair do correspondente padrão linguístico.

Em concreto, as ERL são dadas pela seguinte gramática:

$$ERL ::= (WP, TP, (ID|\_))^*$$

ou seja, uma sequência de triplos (a que chamaremos elementos da ERL), cujas componentes descrevemos de seguida.

WP (word pattern) representa uma expressão regular base, que permitirá descrever padrões que queremos identificar em partes da frase proveniente do utilizador. Por exemplo:

- ".*" será um WP que permite qualquer palavra numa dada parte da frase;
- "ficar|localizar|situar" será um WP que permite exatamente uma das três palavras indicadas numa dada parte da frase.

TP (tag pattern) é uma expressão regular sobre etiquetas gramaticais que dá a possibilidade de introduzir padrões sobre etiquetas gramaticais. Por exemplo:

- "nome_proprio" será o TP que permitirá indicar que numa dada parte da frase se espera um nome próprio;

- No Anexo A.1 está descrita a lista de etiquetas gramaticais que serão permitidas na prototipagem de Bots, desenvolvida no capítulo seguinte.

A última componente de um dado elemento da *ERL* introduz um mecanismo para guardar a palavra na frase proveniente do utilizador, correspondente a esse elemento da *ERL*. Quando a última componente é um *ID* (identificador) esse identificador corresponderá ao nome da variável onde ficará guardada a palavra extraída da frase do utilizador, a que chamaremos uma palavra-chave. Quando a última componente é "_", tal significará que não é importante guardar a palavra "capturada" na frase do utilizador. Por exemplo,

- se o elemento da ERL for (.*, nome_proprio, lugar), espera-se nessa parte da frase um nome próprio, que ficará guardado com o identificador "lugar";
- se o elemento da ERL for (.*, determinante, _), espera-se nessa parte da frase um determinante, mas esse determinante não será guardado.

A escolha destas três componentes para os elementos das *ERL* vem do facto de nos parecerem indispensáveis para a identificação e utilização de palavras-chave nas frases provenientes dos utilizadores, e de permitirem já alguma flexibilidade na descrição dos padrões que poderão corresponder a frases diversas, mas poderão corresponder a frases com o mesmo significado ou a uma mesma intenção por parte do utilizador

Uma propriedade que é importante garantir numa *ERL* é que as últimas componentes de cada um dos seus elementos sejam todas distintas entre si. Com esta propriedade, garante-se que durante a função *verifica* não existirão palavras-chave capturadas em diferentes partes da frase associadas ao mesmo identificador.

Com este modelo de *ERL* baseado em expressões regulares, por um lado temos já alguma flexibilidade para exprimir padrões linguísticos a identificar nas frases do utilizador, mas ao mesmo tempo permitimos que o modelo seja implementado com simplicidade, uma vez que as *ERL* poderão ser compiladas para expressões regulares comuns. Um tal compilador será descrito no capítulo seguinte.

Regressando agora ao ELIZA e aos princípios prescritos por *Weizenbaum*, verificamos que o nosso modelo, genericamente, cumpre esses princípios. A nossa estruturação de *Bots* permite que, através das regras e das *ERL*, consigamos facilmente adicionar aos Bots novos padrões linguísticos a reconher e respetivas ações associadas. Com base nas *ERL* indicadas nas regras do *Bot*, a função *verifica* é

capaz de encontrar contexto para as frases dos utilizadores e identificar as palavras-chave. A função *responde* tem capacidade de gerar uma resposta a dar à pergunta do utilizador, mesmo que não encontre nenhuma regra com padrões que consiga associar à pergunta, e tem capacidade de empreender transformações ao *Bot*, através da alteração do seu estado (que terá consequências no processamento de perguntas subsequentes), ou mesmo através da alteração do seu conjunto de regras.

Concluímos esta Secção apresentando um exemplo concreto que ilustra a ação de um *Bot* perante uma pergunta de um utilizador, seguindo o nosso modelo. Este exemplo, poderá ser corrido no *Bot* protótipo desenvolvido no próximo capítulo.

Considere-se a seguinte pergunta por parte do utilizador:

Onde fica a Universidade do Minho?

Considere-se que o *Bot* inclui uma regra com a seguinte expressão regular linguística e com a seguinte função reação:

ERL:$(.^*,\ Adverbio,\ \_),\ ((ficar|localizar|situar),\ Verbo,\ \_),$
$(.*, Determinante, \_),\ (.^*,\ Nome\ Próprio,\ lugar),\ (?,\ Interrogação,\ \_)$
FR: *Googe Maps Search*

Quando esta pergunta passa pela função *responde*, o primeiro passo é a sua análise sintática que dá como resultado

PoS = (Onde, Adverbio), (ficar, Verbo Presente Indicativo), (a, Determinante), (Universidade do Minho, Nome próprio), (?, Interrogação).

De seguida, no passo (2) da função *responde* é executada a função *verifica* com as ERL e a frase e obtemos

v = [(Google Maps Search, { lugar = Universidade do Minho })],

assumindo que o *Bot* não tem nenhuma outra regra cuja ERL possa ser associada a esta pergunta. Neste caso a função *escolhe* devolverá naturalmente a única alternativa de resposta encontrada, ou seja:

(fr, tokens) = (Google Maps Search, { lugar = Universidade do Minho }).

Quando é executada a função *Google Maps Search* como o token *Universidade do Minho*, a resposta devolvida ao utilizador pelo *Bot* será:

Universidade do Minho - Campus de Gualtar situa-se em R. da Universidade, 4710-057 Braga, Portugal.

3 *Diabrete*: Prototipagem de *ChatBots*

Neste capítulo apresentaremos a ferramenta que desenvolvemos durante o estágio para a prototipagem de ChatBots, designada Diabrete. Esta ferramenta implementa o modelo de ChatBots, que descrevemos ao longo do capítulo anterior, e não corresponde a um *ChatBot* concreto, mas antes a um motor geral para a criação de ChatBots. O Diabrete fornece de antemão as componentes fundamentais ao funcionamento de um Chatbot. A geração de um *ChatBot* concreto a partir do Diabrete passa por lhe adicionar regras específicas, relativas ao domínio de aplicação do *ChatBot*.

Ao longo das secções deste capítulo apresentaremos as ideias seguidas na implementação do Diabrete. Prestaremos especial atenção à forma como se encontram implementadas as várias componentes do núcleo. Em particular, explicaremos a forma como se encontram implementadas as expressões regulares linguísticas, as funçoẽs *responde*, *verifica* e *escolhe*, esta última baseada num classificador construído com técnicas de machine learning.

A implementação do Diabrete foi efetuada na linguagem de programação Python, versão 3, com a base de dados em MySQL. O Diabrete recorre adicionalmente a algumas bibliotecas externas. Duas dessas bibliotecas são o FreeLing e o NLTK. Estas bibliotecas são utilizadas na fase da análise sintática das perguntas do utilizador e na implementação da função *escolhe*, respetivamente, e são apresentadas neste capítulo. O Diabrete está já dotado de algumas regras específicas (que em particular servem para ilustrar o seu funcionamento) e na implementação das funções reação destas regras específicas foram usadas outras bibliotes externas, tais como Google Maps Place API e Wikipedia API.

No apêndice A.2 encontra-se a ligação para o repositório que contém o código fonte desta implementação.

3.1 Bibliotecas Externas

Na implementação do *Diabrete* recorremos à utilização de algumas ferramentas *open-source* para levar a cabo as tarefas da análise sintática das frases provenientes dos utilizadores e para a construção de um classificador, que está na base da im-

plementação da função que escolhe a reação que o *ChatBot* deve apresentar a uma dada pergunta. Para a implementação da primeira tarefa recorremos à biblioteca *Freeling* e para a construção do classificador recorremos à biblioteca *NLTK*. Além destas ferramentas serem *open-source*, ambas revelaram funcionalidades adequadas à implementação das fases do nosso modelo relativas à análise sintática e à escolha de reações, pelo que a sua adoção nos pareceu uma opção natural e deixou mais tempo para o desenvolvimento de outros aspetos de implementação do modelo. No resto desta Secção apresentamos as bibliotecas *Freeling* e *NLTK*.

3.1.1 *FreeLing*

FreeLing é uma biblioteca *open-source* escrita em C++, criada por Lluís Padró, que analisa frases de linguagens naturais morfossintáticamente [4]. Esta biblioteca oferece um leque variado de funcionalidades, mas a anotação de texto processado é a única funcionalidade do *FreeLing* usada na implementação do *Diabrete*.

A análise sintática de frases é feita utilizando técnicas de *machine learning*, designadamente dois classificadores *AdaBoost* e *Support Vector Machine* (SVM) [6], que são previamente treinados com um conjunto de frases etiquetadas. Este conjunto é chamado *Bosque* e insere-se no projeto *Floresta Sintá(c)tica*, sendo composto por 9368 frases analisadas e revistas por linguistas, ou seja, é o conjunto perfeito para treinar tal tipo de classificador. (Na Secção 3.6 fazemos uma muito sucinta apresentação do uso de técnicas de *machine learning* em *ChatBots* e, em particular, descrevemos sucintamente a técnica *Naive Bayes* para a construção de classificadores.)

A interface da biblioteca utilizada neste protótipo é o servidor escrito em *Java*, configurado para a língua Portuguesa e devolve em formato *JSON* a frase etiquetada e a árvore de dependências sintáticas, chamada *analyze*. Portanto, e seguindo documentação da biblioteca, corre-se o comando:

```
analyze -f pt.cfg --output json --outlv dep --server --port 5000
```

A biblioteca também oferece uma interface para comunicar com o servidor, chamada *analyzer_client*. A comunicação entre o *Diabrete* e o servidor é feita através deste comando que é integrado no módulo *freeling_client.py*, e permite assim a obtenção da frase analisada e etiquetada.

3.1.2 *NLTK*

A biblioteca *open-source* NLTK é um plataforma que permite construir e implementar programas que interajam com linguagens naturais, fornecendo assim diversas interfaces para tratamento de linguagens, como por exemplo a extração de *tokens*, a etiquetagem gramatical, vários algoritmos para construir classificadores de *machine learning*, etc.

Como já é utilizado a *FreeLing* para a análise de frases, esta biblioteca será utilizada apenas como forma de construir um classificador *Naive Bayes* para implementar a função do núcleo do *ChatBot* que irá escolher a reação para uma frase, descrito na Subsecção 3.6.2.

O método para a construção de tal classificador está inserido na classe *NaiveBayesClassifier*. Para o construir tal classificador e classificar algo, basta efetuar o seguinte:

```
classificador = nltk.NaiveBayesClassifier.train(training_set)
```

Para a classificação de um reação, é necessário muito mais do que uma chamada ao objeto *classificador*. Tal envolve a criação de uma função que irá extrair características e juntar o conhecimento prévio do *ChatBot*. Na Secção 3.6 será apresentado com detalhe todo este processo.

3.2 Implementação das *Expressões Regulares Linguísticas*

Como vimos no capítulo anterior, as expressões regulares linguísticas (*ERL*) são uma peça fundamental no nosso modelo de *ChatBots*, uma vez que são a ferramenta que permite definir padrões linguísticos a reconhecer pelo *ChatBot* e palavras-chave a extrair das frases dos utilizadores. Nesta Secção descrevemos uma implementação das *ERL* que contempla todos os aspetos definidos na Secção 2.2.

Para a implementação das *ERL* foi previamente criada uma *Domain Specific Language*. Desta forma, construimos uma descrição gramatical das *ERL* livre ainda de pormenores de implementação, mas já específica para o domínio de *ChatBots*. A gramática criada está implementada em *Python*, mas para mais fácil

compreensão apresentamos abaixo esta gramática seguindo a linguagem *Bison*:

```
ERL  : elem
     | elem " " ERL
     ;
elem : pal "/" tag "?"?
     ;
pal  : WP
     | "$" WP
     | "(" id "=" WP ")"
     ;
tag  : TP
     | "$" TP
     ;
```

Assim, na nossa implementação de expressões regulares linguísticas, consideremos que ERL ou é um *elem* (elemento de uma ERL), ou uma sequência de vários *elem* separados por um espaço, e seguindo a modelação de padrões linguísticos descrito na Secção 2.2.

Um *elem* é a unidade básica do padrão, ou seja, representa um triplo *(pal, tag, id)* tal como estipulado pelo modelo. Diz-se então que temos *pal* separado de *tag* pelo símbolo /, e, no final, pode ou não aparecer o símbolo *"?"* (sinalizado pela presença do operador gramatical ?). A presença do símbolo "?" significa que este *elem* é opcional na ERL. A justificação da utilização deste operador vem do facto de poderem ser ignoradas na verificação certas palavras da frase que não sejam relevantes ao padrão. Uma aplicação exemplo será mostrada abaixo.

Tanto *pal* como *tag* seguem a mesma construção, onde WP é abreviatura para *word pattern*, ou seja, um padrão sobre palavras, e TP é abreviatura de *Tag Pattern*, ou seja, um padrão sobre as etiquetas gramaticais. Mais ainda, ao ser utilizado o símbolo terminal $, diz-se que o padrão escrito será substituído, no momento da compilação, por outro padrão. Assim, caso algum padrão seja de grande dimensão, pode ser definido noutro lado e, na expressão linguística, basta colocar o seu nome precedido do símbolo $. Nota-se que este pormenor foi implementado para facilitar a criação e introdução de padrões linguísticos, constituindo uma funcionalidade e não uma necessidade no modelo.

A última componente de *pal* corresponde no modelo ao mecanismo para captu-

$$\underbrace{.^*/RG}_{(1)} \quad \underbrace{.^*/PP.^*?}_{(2)} \quad \underbrace{\$verbos\_lugar/\$verbo\_indicativo\_presente}_{(3)}$$
$$\underbrace{.^*/D.^*?}_{(4)} \quad \underbrace{(lugar = .^*)/.^*}_{(5)} \quad \underbrace{?/Fit?}_{(6)}$$

Figura 3: *ERL* de localização

rar uma palavra-chave com um certo identificador. Ou seja, ao escrever-se entre parênteses o identificador seguido do símbolo terminal = e da *word pattern*, diz-se que esse identificador guardará a palavra-chave capturada.

Um exemplo de uma *ERL* na sintaxe concreta do *Diabrete* é apresentado na Figura 3.

O exemplo de padrão linguístico do Figura 3 pretende descrever uma forma geral de apresentar uma questão acerca da localização. Ou seja, procura identificar frases que tenha a intenção perguntar onde se situa algum lugar ou algum ponto de interesse. Por exemplo:

Onde fica a Universidade do Minho?
ou
Onde se situa a o santuário do Bom Jesus?

são frases possíveis que correspondem ao padrão descrito na Figura 3.

Antes de iniciar a explicação de cada elemento da *ERL* da Figura 3, é importante referir que a utilização do *word pattern* ".*" significa que qualquer palavra é aceite. Quando existir um *tag pattern* da forma $NC.^*$ significa que qualquer etiqueta que comece por *NC* é aceite. Neste caso, *NC* corresponde à etiqueta para um nome comum e qualquer tipo de nome comum, masculino, feminino ou indefinido, será aceite.

Expliquemos agora em detalhe, elemento a elemento, a *ERL* da Figura 3. O primeiro elemento, (1), aceitará qualquer palavra que corresponda a um advérbio *RG*. O elemento (2) aceitará qualquer palavra que seja um pronome pessoal. O elemento (3) indica que a expressão *$verbos_lugar* é uma abreviatura, que neste caso irá ser substituída pelo padrão *(ficar/estar/situar/localizar)* e que *$verbo_indicativo_presente* é também uma abreviatura que será substituída pelo

padrão $VMIP^*$ (que permite qualquer conjugação no presente do indicativo dos verbos em $\$verbos\_lugar$ será aceite). O elemento (4) aceitará qualquer palavra que seja um determinante. O elemento (5) aceitará qualquer etiqueta gramatical e qualquer palavra, sinalizando que essa palavra será uma palavra-chave e será guardada no identificador *lugar*. Por fim, o elemento (6) aceitará o sinal de pontuação "?" (sendo Fit a etiqueta gramatical associada pelo $FreeLing$ a este sinal de pontuação), sendo que o "?" final indica que este elemento é opcional, ou seja "?" pode ou não estar presente na frase do utilizador.

Na implementação do *Diabrete*, são usadas ERL que podem ser classificadas em um de três tipos: as ERL que apresentam padrões de "introdução" e "animação" do $ChatBot$, por exemplo, capazes de identificar frases dos utilizadores como a palavra *"Olá"*; as ERL das regras do $ChatBot$ usadas para responder a pedidos do domínio de aplicação específicos do $ChatBot$; as ERL de reserva, que correspondem às situações em que a pergunta do utilizador não foi associada a nenhuma das outras ERL.

3.3 Implementação das Funções *responde* e *verifica*

As funções responde e *verifica* são duas das funções centrais na nossa modelação de ChatBots, conforme vimos na Secção 2. A função *responde* é o "coração" do *ChatBot*, no sentido em que recebe as perguntas do utilizadores, processa-as, com base nas suas regras e no seu estado, e devolve as respostas aos utilizadores. A função *verifica* é uma das fases fundamentais do processamento efetuado pela função *responde*. Nas duas subsecções seguintes apresentamos cada uma destas funções.

3.3.1 *responde*

A função *responde*, conforme vimos na Secção 2, está dividida em cinco fases : (1) a análise sintática da pergunta colocada pelo utilizador; (2) a verificação dos padrões nas regras que concordam com a pergunta; (3) a escolha de uma função reação; (4) a execução dessa função reação; e (5) a devolução da resposta ao utilizador. Para implementar a função *responde*, recorremos em particular a duas funções: analisa e *verifica*. A biblioteca *FreeLing* que descrevemos na Secção 3.1.1

é usada na implementação da função analisa, pois esta função tem por objetivo devolver a frase de entrada etiquetada gramaticalmente. A função *verifica* será abordada com detalhe na Secção seguinte. Sucintamente, esta função tem por objetivo verificar se há correspondência entre uma frase (já na sua formato com etiquetas gramaticais) e uma ERL,devolvendo um dicionário de pares (identificador,palavra-chave). Apresentamos abaixo a função *responde*:

```
1  def responde(frase):
2      # Análise da frase
3      analyzed = freeling.analyse(frase)
4      dot, tagged = freeling.extract(analyzed['sentences'][0], view=False)
5      parsed_tag = parse_analise(tagged)
6
7      # Verificação
8      res = verifica(parsed_tag)
9
10     if res != {}:
11         res = substitui_por_originais(res, tagged, frase)
12         return call_func(Actions.escolhe(classificador, res))
13     else:
14         res_esp = verifica_especiais(parsed_tag)
15         if res_esp != {}:
16             res = substitui_por_originais(res, tagged, frase)
17             return call_func(Actions.escolhe(classificador, res))
18         else:
19             res_back = verifica_backup(parsed_tag)
20             res = substitui_por_originais(res, tagged, frase)
21             if res_back != {}:
22                 res = substitui_por_originais(res, tagged, frase)
23                 return call_func(Actions.escolhe(classificador, res))
24             else:
25                 return 'Não␣consegui␣entender...'
```

A parte (1) está compreendida entre as linhas 3 e 5, onde na linha 3 é feita a análise sintática, com recurso ao *FreeLing*, e as seguintes linhas são de extração do resultado da análise. As variáveis *dot* e *tagged* correspondem ao grafo de dependências sintáticas e à frase etiquetada, respetivamente. Já a variável *parsed_tag* corresponde ao *parsing* da frase etiquetada devolvida pelo *FreeLing*.

A segunda parte está implementada na linha 8 com a função *verifica*, que será descrita na seguinte Secção. Nesta linha, *res* guardará o dicionário com os *tokens* extraídos pela função *verifica*.

Da linha 10 à linha 28, é feita a escolha da reação, bem como a chamada e devolução do resultado da execução da função reação escolhida. Esta etapa inicia-se com o teste de uma condição para verificar se o resultado da função $verifica$ contém alguma correspondência com as regras específicas do $ChatBot$. Em caso positivo procede-se à escolha e execução da função escolhida (linha 13). Caso contrário, e verificada novamente a frase, mas desta vez, contra as regras de "introdução" e "animação" (linha 15), e havendo correspondência procede-se à escolha e execução da função de reação (linhas 18 e 19). Caso não haja correspondência na verificação das expressões linguística anteriores, verifica-se, por fim, as ERL de reserva que tentam dar uma resposta com base nas palavras encontradas na frase. Por exemplo, suponhamos que o utilizar introduz uma frase "Onde viveu Albert Einstein" e não existe nenhuma ERL que faça correspondência. Então, como o *Diabrete* está pré-programado com uma ERL de reserva que faz corresponder todas as frases que contenham um nome próprio, a função reação escolhida será *procura_nome* que invocará a função externa *wikipedia_search* com o parâmetro "Albert Einstein" e retornará o primeiro parágrafo da página da *Wikipedia* encontrada.

Ao testar a correspondência de uma frase do utilizador com as expressões regulares linguísticas pela ordem acima descrita, priviligiamos as ERL que contêm relevância informativa no domínio específico do $ChatBot$.

Por fim, na linha 28, caso não haja qualquer tipo de resposta possível, o *Diabrete* devolve uma resposta pré-programada.

3.3.2 *verifica*

A função *verifica*, conforme referido na Secção 2.1 e na subsecção anterior, desempenha uma das tarefas principais envolvidas na função *responde*. A função *verifica* irá procurar correspondências entre as perguntas dos utilizadores e os padrões linguísticos especificados nas regras do *ChatBot*. Ao contrário da descrição na Secção 2.1, a implementação em Python desta função tem apenas um argumento (a frase já com a informação acerca da sua análise sintática), uma vez que esta função também implementa o ciclo que irá percorrer todas as regras do *ChatBot*, com o objetivo de extrair as respetivas expressões regulares linguísticas definidas. A função *verifica* está descrita abaixo.

```
def verifica(frase):
    r = {}
    for reg_exp in Regexs.keys():
        matched = Regexs[reg_exp]['exp'].search(frase)
        if matched:
            r[reg_exp] = res.groupdict()
    return r
```

Esta função inicia com a criação de um dicionário vazio, que será usado para ir guardando todas as correspondências encontradas durante o ciclo. A cada iteração e após a verificação, caso haja alguma correspondência, são inseridas no dicionário as correspondências associadas ao nome da expressão regular linguística. Como pode ser visto, esta é uma função que se torna simples pelo motivo de se utilizar as expressões regulares comuns, e assim, permitir a utilização dos seus mecanismos. Por exemplo, o método utilizado, *search*, irá procurar na expressão se há correspondências, e devolverá o dicionário com as mesmas.

3.4 Função *Main* e Bases de Dados

Nesta Secção abordar-se-ão duas outras componentes essenciais ao funcionamento do *Diabrete*, designadamente a função Main e a base de dados, que implementará o estado do *ChatBot*.

3.4.1 *Main*

Conforme a modelação descrita na Secção 2.1 e a respetiva implementação descrita na Subsecção 3.3, a função *responde* tem apenas por objetivo devolver uma resposta a cada uma das perguntas colocadas pelos utilizadores. Ou seja, é necessária uma outra função responsável pela invocação da função *responde*. Esta tarefa será desempenhada pela função "principal" do *Diabrete*, chamada Main, como é habitual. Na função Main estão também implementadas as configurações iniciais, que são essenciais para o arranque do *ChatBot*, é assegurada a ligação entre todas as funções e módulos que implementam o *Diabrete* e estão implementados os procedimentos necessários a guardar os dados relevantes no momento de encerrar o *ChatBot*.

O módulo *Configurations* alberga todas as variáveis de configuração necessárias, tais como o nome do *ChatBot*, neste caso *Diabrete*, o esquema da sua base de dados juntamente com o nome de utilizador, password e o endereço da base de dados. Qualquer tipo de configuração que se pretenda adicionar para o funcionamento do *Bot* é feita neste módulo.

Após a inicialização, começa a interface com o utilizador que é nada mais nada menos que um ciclo infinito que só é interrompido quando ocorrer um erro grave no sistema ou o utilizador deseja sair. Durante o ciclo, é esperado um *input* do utilizador e, quando recebido, é chamada a função *responde* que irá processar esse *input*. O código da função *Main* é o seguinte:

```
1  def main(args):
2      global users_db
3      global current_user
4
5      init_dbs()
6
7      try:
8          first_conversation()
9          try:
10             while True:
11                 s = input(current_user + ": ")
12                 if s == "sair":
13                     break
14                 else:
15                     print(bot_name,':',responde(s))
16         except EOFError and KeyboardInterrupt:
17             pass
18     except ValueError:
19         pass
20     finally:
21         dump_users(users_db, "users_db.json")
```

É utilizada uma função auxiliar, *init_dbs*(), para tratar da inicialização das base de dados. Após a inicialização, inicia-se a interação com o utilizador. A primeira interação passa por um processo de pedir o nome do utilizador, para poder carregar um estado passado ou criar um estado novo referente a esse utilizador. Também existe a opção do utilizador interagir com o *ChatBot* como um convidado.

Após isto, inicia-se o ciclo com a captura de exceções. A captura destas exceções é essencial, pois mesmo se um utilizador introduzir *EOF* ou o atalho para terminar processos, o *ChatBot* deve ser capaz de terminar corretamente, ou seja, capaz de

guardar o estado do sistema. Este modo de terminar está definido após *finally* e usa a função *dump_users*.

3.4.2 Bases de Dados

Uma das componentes da nossa modelação de *ChatBots* é o estado. Conforme descrito na Secção 2, o estado prevê uma base de dados para guardar informações base do *ChatBot*, como por exemplo a informação relativa aos seus utilizadores, e uma base de dados para o armazenamento de conhecimento relativo a interações passadas do *ChatBot*. Esta última revelou-se ser de extrema importância para a construção do classificador que está na base da função que escolhe a reação do *ChatBot* a uma dada pergunta. A base de dados de conhecimento do Diabrete segue o modelo relacional MySQL e armazena informações como as frases processadas e as ações desencadeadas pela *ChatBot*, ou as palavras-chave observadas e qual o contexto onde apareceram.

A base de dados utilizada pelo *Diabrete* para armazenamento de conhecimento tem a estrutura descrita na Figura 4.

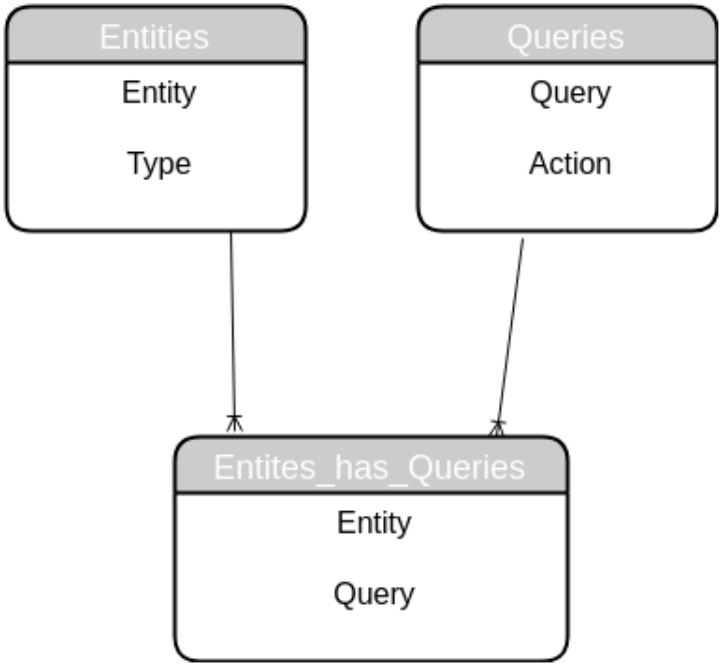

Figura 4: Modelo relacional da base de dados de conhecimento

O modelo da base de dados de conhecimento do *Diabrete* contem três tabelas:

Entities, *Queries* e *Entities_has_Queries*. Em cada tabela só há dois campos.

A tabela *Entities* refere-se às palavras-chave e, a chave principal é a palavra-chave e o outro campo é o seu tipo.

A tabela *Queries* refere-se às perguntas de utilizadores, onde a chave principal é a pergunta e o outro campo a ação tomada.

A última tabela, *Entities_has_Queries*, representa a relação entre as duas tabelas anteriores, para que seja possível que uma entidade esteja em várias perguntas, e várias perguntas tenham a mesma entidade.

A base de dados descrita na Figura 4 só armazena o conhecimento resultante das interações tidas com os utilizadores. A base de dados dos utilizadores é um ficheiro à parte, em formato *JSON*, e que, nesta implementação, só guarda os nomes dos utilizadores.

3.5 Implementação de *Funções de Reação*

Por forma a poder ilustrar mais amplamente o nosso modelo no que se refere às funções de reação, dotámos o Diabrete de algumas regras específicas, que definem alguns padrões linguísticos para lidar com perguntas do utilizador relativas a lugares ou a pontos de interesse, como monumentos ou outros. As correspondentes funções de reação estão implementadas num módulo chamado Actions e recorrem a duas bibliotecas externas, designadamente *Google Maps Place* API e *Wikipedia* API, que contêm as funções que irão dar o resultado ao utilizador.

Um exemplo de uma função reação implementada em *Actions* que encapsula uma função proveniente de uma biblioteca externa é a seguinte:

```
def procura_lugar(**kargs):
    if kargs['dict']:
        return gmaps.procura_lugar(kargs['dict']['locality'])
    else:
        return 'Não consegui entender o lugar...'
```

Aqui, a função *procura_lugar* encapsula a função da biblioteca do *Google Maps Places* que procura um ponto de interesse. A função *procura_lugar* aceita um número indeterminado de *inputs* da função, o que é útil não só para poder articular

a chamada da função externa, como para mais tarde poder encadear com outras funções.

Para que uma função reação seja chamada conforme o nome associado nas regras do *ChatBot*, é necessária uma função que faça essa ligação. Esta função é chamada *call_func* e será útil para a implementação da função *responde*. A função $call\_func$ é definida da seguinte forma:

```
def call_func(func, args):
    return getattr(Actions, func)(dict=args)
```

Utilizando os atributos de classe do *Python*, pode-se chamar através de uma *string* uma função que tenha esse nome. Este tipo de mecanismo torna a prototipagem de *ChatBots* através do *Diabrete* mais simples para o programador.

3.6 *Machine Learning* em *ChatBots*

3.6.1 Classificador *Naive Bayes*

As técnicas de machine learning permitem dar a um programa capacidades de aprendizagem através de dados, sem que sejam explicitamente programados para tal, recorrendo, por exemplo, a técnicas estatísticas.

Os programas construídos com base em machine learning podem assim ultrapassar dificuldades encontradas com os programas tradicionais, que seguem um conjunto de regras restritas ao prever e ao tomarem decisões, com base num modelo construído através dos dados fornecidos. Os algoritmos baseados em machine learning são tipicamente divididos em duas grandes categorias: algoritmos de aprendizagem supervisionada e algoritmos de aprendizagem não-supervisionada. A primeira categoria, aprendizagem supervisionada, requer que um conjunto de dados seja fornecido juntamente com o resultado pretendido para cada elemento do conjunto, para que o algoritmo possa ser construído. Este conjunto é denominado de conjunto de treino (training set). Além disso, é necessário determinar quais as variáveis ou características (features) que o modelo deve analisar para poder prever um resultado para um dado input. A segunda categoria, aprendizagem não-supervisionada, ao contrário da supervisionada, não necessita de um conjunto de treino nem características que identifiquem propriedades nem padrões,

para construir um algoritmo capaz de prever uma resposta para um input. Os algoritmos não-supervisionadas, eles próprios, procuram detetar padrões e relações nas entidades em estudo, com base nos dados que lhe são fornecidos. Por isso, estes algoritmos necessitam tipicamente de um grande conjunto de dados para que possam fazer previsões de resultados mais acertadas para os inputs.

Os classificadores são, como o nome sugere, programas que tem por objetivo classificar os seus inputs numa classe, de entre um número finto de classes. O classificador do tipo Naive Bayes é um algoritmo de aprendizagem probabilística que deriva da teoria de decisão bayesiana, e corresponde a um modelo de aprendizagem supervisionado. Este modelo de aprendizagem é simples e pode ser implementado com muita eficiência e complexidade linear, o que o torna rápido. O princípio base deste modelo é de cada feature de um input é independente de todas as outras. Apesar deste princípio não ser verificado por linguagens naturais, uma vez que o contexto em que ocorre uma palavra é fundamental para lhe dar significado, este classificador é muito usado na classificação de documentos e texto pela sua simplicidade e pelas propriedades que o tornam muito eficaz neste domínio [10].

3.6.2 Implementação da Função *escolha*

Nesta Subsecção explicaremos como é implementada a função *escolhe* invocada pela função *responde* e responsável pela seleção de ações a executar como resposta à pergunta do utilizador. Esta função é implementada com um classificador do tipo *Naive Bayes*. Grosso modo, a pergunta de entrada, após pré-processamento para extração das características relevantes, irá ser classificada com um das funções de reação do *ChatBot*, procurando que a reação escolhida seja a mais apropriada para a pergunta. Para a construção do classificador *Naive Bayes* primeiro é necessário definir quais as características das frases que queremos extrair.

As características que se notam mais importantes são: as palavras-chave e as ações escolhidas em sessões passadas e também as funções de reação definidas nas regras do *ChatBot*. Estas palavras-chave e ações são provenientes da análise e verificação efetuadas à frase com as *ERL*, e da base de dados com o conhecimento de sessões passadas. Para facilitar a compreensão desta extração de características, começa-se por apresentar um conjunto de treino para o classificador:

```
{"queries" : {
    "Onde fica Braga?" : {
        "action" : "procura_lugar",
        "keywords" : ["braga"]},
```

```
    "Quem é Mozart?" : {
        "action" : "procura_pessoa",
        "keywords" : ["Mozart"]},
    "Onde se situa o Porto?" : {
        "action" : "procura_lugar",
        "keywords" : ["Porto"]}
        },
"entities" : {
    "braga"  : {"TYPE": "locality"},
    "porto"  : {"TYPE": "locality"},
    "mozart" : {"TYPE": "person"}
    }
}
```

Este exemplo de um conjunto de treino está dividido em duas partes importantes: os pedidos feitos por utilizadores, que contêm informação sobre a(s) palavra(s)-chave e reação referentes a esse pedido; e o tipo associado a cada uma das palavras-chave.

Assim, percebe-se que a função que irá extrair as características de um pedido do utilizador, às palavras-chave e ações provenientes da função *verifica*, acrescentará palavras-chave e ações provenientes da base de dados do conhecimento. Apresenta-se abaixo a definição desta função:

```
1  def features(frase, verifica, db_keys, db_frases, dep_tree=None):
2      actions_from_exp = [a for exp in verifica for a in mapa_acoes[exp]]
3      actions_from_knowledge = get_action_from_frase(frase, db_frases)
4      entities_from_knowledge = [get_entities(verifica[exp][exp2], db_keys)
5                                 for exp in verifica for exp2 in verifica[exp]]
6      entities_from_exp = []
7      if len(verifica.keys()) > 0:
8          for key1 in verifica:
9              for key2 in verifica[key1]:
10                 entities_from_exp.append((verifica[key1])[key2])
11
12     features = {}
13     if len(actions_from_exp) > 0:
14         features['action'] = actions_from_exp[0]
15
16     if len(actions_from_knowledge) > 0:
17         if (len(actions_from_knowledge[0]) > 0):
18             if 'action' not in features.keys():
19                 features['action'] = features['action'] +
20                                      actions_from_knowledge[0]
21             else:
```

```
22                  features['action'] = actions_from_knowledge[0]
23
24      if len(entities_from_knowledge) > 0:
25          if len(entities_from_knowledge[0]) > 0:
26              features['entities'] = entities_from_knowledge[0]
27
28      if len(entities_from_exp) > 0:
29          if 'entities' not in features.keys():
30              features['entities'] = entities_from_exp
31          else:
32              features['entities'] = features['entities'] + entities_from_exp
33
34      return  features
```

Esta função pode ser separada em 3 partes: busca de ações (linhas 2 a 10), junção das características (linha 12 a 32) e devolução das características (linha 34).

A busca de ações baseia-se, em particular, no parâmetro *verifica*, que contém as reações e as palavras-chave que resultam do processamento da frase pela função *verifica*. Com esta informação é então percorrida a base de dados de conhecimento para identificar ações passadas associadas a palavras-chave provenientes da verificação.

A junção destas características (ações e palavras-chave) é feita para um dicionário chamado *features*. A junção é feita de modo a evitar repetições de *features*.

A função *features* será responsável pela produção dos parâmetros que serão passados ao classificador, que irá escolher a ação a executar, como resposta ao utilizador.

O classificador é do tipo *Naive Bayes* e, com base nos parâmetros passados, associa uma probabilidade a cada uma das ações possíveis, sendo então escolhida a ação com maior probabilidade associada. Para criar o classificador *Naive Bayes* é necessário ensiná-lo com um conjunto de treino, composto por frases e pelas ações que devem ser desencadeadas por estas frases. Como referido na Secção 3.1, esta tarefa é feita com recurso à biblioteca NLTK, através da seguinte invocação:

```
classificador = nltk.NaiveBayesClassifier.train(conjunto_de_treino)
```

Em síntese, para classificar uma frase proveniente do utilizador, primeiro executa-

se a função que extrai as características e de seguida passa-se esse resultado ao classificador, através das seguintes invocações:

```
caracteristicas = features(frase, analise, expressões, base_dados)
funcao = classificador.classify(caracteristicas)
```

O método do classificador devolverá em *funcao* o nome da função reação a ser executada.

4 Conclusão

Neste trabalho abordámos o problema da construção de *ChatBots*. Por um lado, desenvolvemos uma modelação original de *ChatBots* e, por outro lado, construímos uma ferramenta que permite a criação de *ChatBots* que seguem o modelo que desenvolvemos. A modelação revelou-se como o elemento mais importante do nosso trabalho. O modelo que desenvolvemos tem como peça central do *ChatBot* o seu *núcleo*, responsável por processar as perguntas de utilizadores, com recurso a regras pré-estabelecidas, compostas por expressões regulares linguísticas associadas a funções de reação. As expressões regulares linguísticas são uma linguagem que desenvolvemos para exprimir padrões linguísticos presentes em linguagens naturais e dar informação ao *ChatBot* sobre o contexto em que ocorrem esses padrões. As expressões regulares linguísticas utilizam como base expressões regulares, não só sobre palavras, mas também sobre etiquetas gramaticais utilizadas na análise sintática de frases, e permitem indicar quais são as palavras-chave a extrair quando um dado padrão linguístico é observado. Um outro aspeto importante das expressões regulares linguísticas é que são de implementação simples, uma vez que podem ser compiladas para expressões regulares ordinárias. Assim, as expressões regulares linguísticas revelaram-se também um desenvolvimento importante do nosso trabalho.

Como referido, neste trabalho também fizemos uma prova de conceito com a implementação da ferramenta *Diabrete*. O *Diabrete* foi pensado e implementado como um motor gerador de ChatBots, e não como um *ChatBot* específico, para um domínio de aplicação específico. Este gerador permite uma rápida prototipagem de *ChatBots* em diversos domínios aplicacionais, bastando ao programador construir as regras relativas ao domínio específico de aplicação, o que envolve a identificação dos padrões linguísticos de interesse, as funções reação associadas e a implementação das funções reação. Na implementação do Diabrete foram usadas ferramentas externas, *open-source*, para desempenhar algumas tarefas associadas ao nosso modelo, nomeadamente a análise morfossintática das frases através da biblioteca *FreeLing* e a construção de um classificador do tipo *Naive Bayes* para a escolha de uma função de reação através da biblioteca *NLTK*. A análise sintática feita pelo *FreeLing* utiliza classificadores de *machine learning* treinados com o Bosque (um conjunto muito vasto de frases analisadas e certificadas por linguistas), o que torna este analisador muito eficaz nesta tarefa. É ainda de salientar que na implementação que desenvolvemos, construímos algumas regras específicas, para lidar com perguntas relativas a localização de sítios ou de pontos de interesse, em

geral. Estas regras específicas recorrem também a bibliotecas externas como a *Google Maps Place* API e a *Wikipedia* API.

Devido à escassez de trabalhos e literatura no domínio da modelação de *ChatBots*, decidimos criar o nosso próprio modelo, procurando abstrair na organização e funcionamento de alguns *ChatBots* em concreto. A nossa modelação contempla já muitos dos aspetos presentes em *ChatBots*, mas pode, naturalmente, ser enriquecido em várias direções. Uma dessas direções é a possibilidade de prever a presença de "Regras Comportamentais", como forma de poder especificar o tipo de comportamento esperado pelo *ChatBot*, por exemplo, num contexto de presença e num contexto de ausência do utilizador. Outra direção importante para o enriquecimento do nosso modelo é prever a possibilidade de alteração e adaptação das regras pré-existentes no núcleo do *ChatBot*, como forma de exibir reações mais adequadas às perguntas dos utilizadores. Uma outra direção poderia ainda ser a geração de novas regras através da introdução de triplos RDF numa base de dados específica, automatizando assim o processo de introdução de expressões regulares linguísticas mais concretamente, definindo triplos RDF, definimos relações entre colunas de uma base de dados, que por sua vez originam regras linguísticas que podem ser convertidas para expressões linguísticas. Por exemplo, com o triplo (rio, desaguar, cidade) define-se a relação desaguar entre a coluna rio e cidade, ou seja, definimos uma relação que indica que rios desaguam em quais cidades. Com esta relação seria possível gerar ERL capazes que identificar estes padrões automaticamente.

Referências Bibliográficas

[1] Edward Loper & Ewan Klein Bird Steven. *Natural Language Processing with Python*. 2009.

[2] X. Carreras et al. «Freeling: An Open-Source Suite of Language Analyzers». Em: (2004).

[3] Jason D M Rennie; Lawrence Shih; Jaime Teevan; David R Karger. *Tackling the Poor Assumptions of Naive Bayes Text Classifiers*. Rel. de pesquisa. Artificial Intelligence Laboratory; Massachusetts Institute of Technology; Cambridge, 2003.

[4] Lluís Padró. *FreeLing Home Page*. URL: `http://nlp.lsi.upc.edu/freeling/node/1`.

[5] Bayan Abu Shawar e Eric Atwell. «Chatbots: are they really useful?» Em: *Zeitschrift für Computerlinguistik und Sprachtechnologie* 22.1 (2007), p. 29.

[6] Lluís Stanilovsky Evgeny; Padró. *FreeLing 3 . 0 : Towards Wider Multilinguality*. Universitat Politecnica de Catalunya, pp. 2473–2479.

[7] Alan Turing. «Computing Machinery and Intelligence». Em: *Mind* 59.236 (1950), pp. 433–460.

[8] Michael W.Berry e Jacob Kogan. *Text Mining Application and Theory*. Vol. 39. 5. 2010.

[9] Joseph Weizenbaum. «ELIZA - A computer program for the study of natural language communication between man and machine». Em: *Communications of the ACM* (1966). URL: `http://portal.acm.org/citation.cfm?doid=365153.365168`.

[10] Harry Zhang. «The Optimality of Naive Bayes». Em: *Florida Artificial Intelligence Research Society Conference* 2 (2004), pp. 1–6.

A Apêndices

A.1 Tagset do FreeLing

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*pastparticiple*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*; **M**:*plusquamperfect*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*ratio*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*personal*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*; **P**:*possessive*
2	degree	**S**:*superlative*; **V**:*evaluative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessorpers	**1**:*1*; **2**:*2*; **3**:*3*
6	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*possessive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**A**:*augmentative*; **D**:*diminutive*

A.2 Código Fonte

https://github.com/mrcalixe/BOT