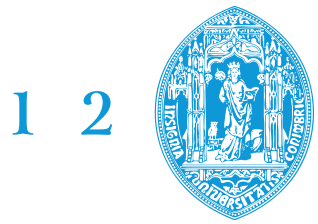
UNIVERSIDADE D
COIMBRA

Ricardo Jorge Rodrigues Vieira

LEXIA 2.0

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Sistemas Inteligentes, orientada pelo Professor Doutor Ernesto Jorge Fernandes Costa e pelo Doutor Rui Lopes da Critical Software apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2022



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Ricardo Jorge Rodrigues Vieira

Lexia 2.0

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Sistemas Inteligentes, orientada pelo Professor Doutor Ernesto Jorge Fernandes Costa e pelo Doutor Rui Lopes da Critical Software apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2022

Agradecimentos

Quero agradecer aos orientadores da Critical Software, Doutor Rui Lopes, Engenheira Ana Guarino e ao orientador do departamento Professor Doutor Ernesto Costa por todo o acompanhamento e conhecimento transmitido, bem como toda a disponibilidade demonstrada no decorrer do estágio.

À minha família, especialmente aos meus pais, que sempre me apoiaram, me incentivaram e por me terem proporcionado a possibilidade de estudar e de ingressar no ensino superior. Sem eles certamente que nunca seria quem sou hoje, nunca teria chegado onde cheguei nem teria alcançado os meus objetivos.

A todos os amigos, sejam eles de infância ou aqueles que conheci durante estes cinco fantásticos anos, que sempre estiveram do meu lado nos bons e maus momentos e com os quais partilhei histórias incríveis que certamente irei para sempre relembrar.

Um especial agradecimento aos Condeses do Ameal que para além de todos os momentos mais animados e descontraídos, de todas as nossas vivências, de todos os momentos passados juntos, acima de tudo sempre fomos como uma família e foram um dos pilares essenciais do meu percurso académico.

Abstract

The constant evolution of the technological industry has changed the way humans interact with different devices. During this evolution, virtual assistants have risen, which have been focus of research due to their popularity and ability to perform everyday tasks.

Lexia is a platform created by Critical Software that aims to create custom virtual assistants that can be easily integrated with external APIs and that present themselves with augmented cognitive capabilities. The main objectives of this internship are to improve Lexia and the interaction between the user and the virtual assistant. To achieve these goals, a set of features has been identified as required: i) to add support for stories in order to maintain the state of the conversations making them more fluid; ii) to add support for entity-related memory in order to enable form-based conversations and keep user data between intents; and iii) to migrate the natural language core to a newer version, which will allow the use of more robust models compared to the ones used in the first version of Lexia and to add support for incremental models improving configuration management.

The intern contributed to the evolution of the platform by implementing a new solution in order to make the conversation between the user and the assistant more fluid. Stories became part of Lexia, improving the conversation flow managment. Lexia can also save crucial information given by the user in order to use it in different points of the conversation. Lastly, it now uses a more recent version of the natural language core, making it now able to use more robust models and consequently improve the classification of the messages sent by the user. This migration also allowed Lexia to use more recent features, namely incremental training.

Keywords

Natural Language Processing, Intent Classification, Entity Extraction, Virtual Assistants.

Resumo

A constante evolução da industria tecnológica tem mudado a forma como o ser humano interage com diferentes dispositivos. Durante esta evolução surgem os assistentes virtuais, que têm sido foco de investigação devido à sua popularidade e capacidade de realização de tarefas do quotidiano.

O Lexia é um sistema criado pela Critical Software que tem como objetivo criar assistentes virtuais customizados de forma rápida e simples, de maneira a que possam ser facilmente integrados com APIs externas e que apresentem capacidades cognitivas aumentadas. Este estágio tem como principais objetivos melhorar o Lexia e a conversação entre o utilizador e o assistente virtual. Para que tal seja cumprido, foram identificadas algumas funcionalidades necessárias: i) adicionar suporte para histórias de maneira a manter estados das conversas e torná-las mais fluidas; ii) adicionar suporte para memória relativa às entidades de maneira a implementar conversas baseadas em histórias e guardar informação entre intenções; e iii) migrar o núcleo de processamento de linguagem para uma versão mais recente, permitindo a utilização de modelos mais robustos comparativamente àqueles usados na primeira versão do Lexia e permitindo também a adição de suporte para o treino incremental melhorando a gestão da configuração da plataforma.

O estagiário contribuiu para a evolução da plataforma implementando uma nova solução para tornar a conversação entre o utilizador e o assistente mais fluida. O Lexia passou a integrar histórias para uma melhor gestão do fluxo da conversação. Passou também a guardar determinadas informações recebidas pelo utilizador para que pudessem ser reutilizadas em diversos pontos da conversa. Por fim, foi também migrado o núcleo de processamento de linguagem para uma versão mais recente, podendo fazer uso de modelos mais robustos e consequentemente melhorar a classificação das mensagens enviadas pelo utilizador. Esta migração permitiu também ao Lexia utilizar funcionalidades mais recentes, nomeadamente o treino incremental.

Palavras-Chave

Processamento de Linguagem Natural, Classificação de Intenções, Extração de Entidades, Assistentes Virtuais.

Conteúdo

Acrónimos

BERT *Bidirectional Encoder Representations from Transformers*.

CSW Critical Software.

DIET *Dual Intent Entity Transformer*.

MLM *Masked Language Model*.

NLU *Natural Language Understanding*.

NSP *Next Sentence Prediction*.

POC *Proof of Concept*.

SDK *Software Development Kit*.

SVM *Support Vector Machine*.

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nas últimas décadas presenciámos uma mudança na maneira como interagimos com os computadores dada a constante evolução nas tecnologias usadas por estes. Durante esta evolução surgiram os assistentes virtuais, que são aplicações que utilizam Inteligência Artificial. O objetivo destes assistentes é perceber mensagens em linguagem natural enviadas pelo utilizador devolvendo uma resposta ou realizando ações de acordo com a intenção identificada. Estes assistentes continuam a ser alvo de investigação, uma vez que demonstram ser bastante úteis em diversos contextos, por exemplo, em situações de atendimento ao cliente, casas inteligentes, carros inteligentes, entre outros. Assim sendo, estes assistentes também podem ser aplicados em ambiente empresarial, uma vez que apresentam diversas vantagens nomeadamente a capacidade de automatizar tarefas demoradas, estar continuamente disponível e a facilidade de atender diversos pedidos em simultâneo. Todas estas vantagens contribuem para economizar tempo e recursos dentro da empresa.

Aplicações específicas que recorrem a este tipo de tecnologias requerem também implementações específicas para que possam não só responder às necessidades do cliente mas também de cumprir os principais objetivos para os quais foram desenvolvidas. É neste contexto que surge o Lexia, um sistema desenvolvido pela Critical Software, criado para o desenvolvimento de assistentes virtuais que ambiciona facilitar o processo de desenvolvimento dos mesmos. Tal é possível, uma vez que fornece a base para a construção, não havendo a necessidade de criar tudo de raiz. Deste modo, o Lexia permite personalizar estes assistentes consoante as necessidades da empresa e economizar tempo com o processo de desenvolvimento. Por sua vez, os executores criados com o Lexia integram funcionalidades que facilitam a realização de determinadas tarefas internas à empresa, como é o exemplo do Redy que permite marcar reuniões, verificar a disponibilidade de horários, reservar salas, entre outros.

1.1 Motivação

O Lexia demonstrou a capacidade de reconhecer a maioria das intenções dos utilizadores e de dar resposta às mesmas. No entanto, com a sua constante utilização, foram identificadas algumas lacunas.

Certas mensagens nem sempre eram classificadas corretamente e a conversação poderia ser melhorada, tornando a interação entre o utilizador e o assistente mais fluida. A fim de melhorar o Lexia e de o tornar mais completo é necessário adicionar métodos mais modernos de processamento das mensagens introduzidas pelo utilizador para que se possa melhorar a classificação destas e, por sua vez, dar uma melhor resposta aos utilizadores. Para além da classificação de mensagens e da conversação em si, pretende-se também melhorar a gestão da configuração. Para tal, pretende-se adicionar a funcionalidade de treino incremental para que novos dados de treino possam ser inseridos no modelo de forma mais rápida e simples.

É também notória a falta da funcionalidade para manter estado das conversas de uma forma genérica, uma vez que esta implementação está feita ao nível de cada executor. Esta funcionalidade permite aos assistentes não só seguir um determinado fluxo de conversação mas também guardar informação que já tenha sido fornecida em mensagens anteriores, evitando pedir ao utilizador a introdução de informação repetida. Pretendeu-se implementar uma solução que conseguisse gerir o estado de todos os assistentes a partir da plataforma do Lexia, melhorando o processo de gestão de memória.

Para que o Lexia pudesse evoluir a nível de classificação de mensagens e as funcionalidades necessárias fossem implementadas, tomou-se a decisão de desenvolver o Lexia 2.0.

1.2 Objetivos

Devido às lacunas apresentadas na primeira versão do Lexia e à necesidade de tornar a conversa entre o assistente virtual e o utilizador mais fluida, surgiu a necessidade de melhorar esta plataforma. Depois de finalizada a implementação, o assistente virtual que está em execução na Critical Software será atualizado para tirar partido destas funcionalidades, não sendo este um objetivo deste estágio.

Para testar as funcionalidades e verificar o cumprimento dos objetivos foi utilizado um pequeno projeto de maneira a conseguir trocar mensagens com o assistente e realizar comparações entre ambas as versões.

Os objetivos de implementação do Lexia 2.0 são os seguintes:

- Melhorar a experiência de conversação com o utilizador, tornando-a mais fluida.
- Permitir manter estado da conversa, para que possam ser mantidas infor-

mações que vão sendo fornecidas pelo utilizador no decorrer da mesma e guiar a conversa consoante as intenções que vão sendo classificadas.

- Melhorar a gestão da configuração incluindo treino incremental e modelos de domínio.
- Aperfeiçoar o modelo de deteção de linguagem para frases curtas (Português e Inglês), embora este seja um extra.

Para que tais objetivos fossem cumpridos e de maneira a possibilitar a implementação das funcionalidades necessárias e a utilização de novos modelos de classificação, foi essencial fazer a migração para que a plataforma Lexia utilize a versão mais recente do Rasa.

Para além dos objetivos referentes à implementação em si, este estágio tem também como finalidade a aquisição de conhecimentos científicos e o desenvolvimento de competências inerentes à minha integração profissional.

1.3 Estrutura

O restante documento encontra-se dividido nos seguintes capítulos:

O segundo capítulo descreve o funcionamento do Rasa e a arquitetura da primeira versão do Lexia, explicando cada módulo e a sua respetiva função na plataforma.

O terceiro capítulo apresenta uma comparação entre algumas das alternativas ao Rasa e as funcionalidades atuais desta plataforma, bem como as novas funcionalidades que foram implementadas no Lexia 2.0.

O quarto capítulo é direcionado para a abordagem, onde são apresentados os requisitos funcionais, não funcionais, as alterações realizadas durante o segundo semestre e a análise de riscos.

O quinto capítulo descreve a metodologia seguida durante o primeiro semestre e a fase de desenvolvimento no segundo semestre. Neste capítulo estão também apresentadas as tarefas realizadas com a respetiva duração num Diagrama de Gantt.

No sexto capítulo está descrita a implementação que permite ao Lexia manter estado das conversas, bem como algumas das decisões tomadas.

No sétimo capítulo são apresentadas as diferenças no fluxo das conversações entre as versões do Lexia e os resultados obtidos tanto na classificação de intenções como na extração de entidades.

No oitavo capítulo é apresentado um resumo do trabalho realizado no decorrer do estágio e é também abordado o trabalho futuro.

Capítulo 2

Conhecimento Prévio

Neste capítulo são apresentados alguns conceitos utilizados tanto no Rasa [1] como no Lexia e que servem de base ao funcionamento deste tipo de assistentes virtuais. É também apresentada uma descrição dos componentes que constituem a framework do Rasa e como é que estes interagem entre si para classificar as mensagens do utilizador de forma a identificar a ação pretendida.

Seguidamente, é apresentada a plataforma do Lexia desenvolvida pela Critical Software (CSW) e como é que esta tira partido da *framework* Rasa. Inicialmente, é explica a arquitetura de alto nível do sistema e de seguida o modelo de domínio da plataforma Lexia. São também explicados os diferentes módulos da plataforma e quais as suas funções.

2.1 Conceitos

Nesta secção serão apresentados alguns conceitos utilizados durante o desenvolvimento da plataforma. Estes conceitos surgem no contexto do Rasa e têm como objetivo guardar informação relativa à classificação da mensagem do utilizador, à extração de informação que possa existir na mensagem e ao controlo do fluxo da conversa consoante a intenção recebida. Enquanto os conceitos de intenção e entidade estão diretamente relacionados com a mensagem do utilizador, os conceitos de história, formulário e *slot* estão relacionados com o fluxo da conversa e com o armazenamento da informação recebida.

2.1.1 Intenção

A intenção refere-se ao que o utilizador tentou transmitir ou realizar com a mensagem introduzida. É extraído o significado da frase para que esta possa ser categorizada e tomada a respetiva ação.

2.1.2 Entidade

As entidades estão associadas às intenções e adicionam informação adicional, não alterando o sentido da frase. As entidades podem estar associadas a diversas intenções, da mesma forma que uma intenção pode ter diversas entidades. Estas informações adicionais podem ser cidades, cores, números, dias da semana, entre outros.

2.1.3 História

As histórias são a representação de uma conversa entre um utilizador e o assistente. Estas utilizam um formato específico, pois guardam apenas as intenções do utilizador ao invés de toda a mensagem. A deteção da história é determinada com base na última intenção introduzida pelo utilizador e também no fluxo da conversa. As histórias são definidas pelo programador na configuração do assistente virtual para que durante a conversação possam ser reconhecidas à medida que o utilizador vai introduzindo as mensagens. Para além disso, as histórias são também utilizadas para determinar qual a próxima ação a ser executada pelo assistente, complementado assim a parte de classificação da mensagem, isto é, a mesma mensagem em contextos diferentes pode ter ações e respostas distintas. Na Figura 2.1 podemos observar um exemplo de como as histórias são definidas nos ficheiros de configuração do Rasa, bem como a definição de cada ação para cada intenção introduzida pelo utilizador. Por sua vez, na Figura 2.2 podemos verificar a aplicação da história referida anteriormente numa conversa real.

```
stories:
- story: Context switch from credit card payment to money transfer
  steps:
  - intent: pay_credit_card
  - action: credit_card_payment_form
  - active_loop: credit_card_payment_form
  - intent: transfer_money                          # - user requests a money transfer
  - active_loop: null                               # - deactivate the credit card form
  - action: transfer_money_form                     # - switch to the money transfer form
  - active_loop: transfer_money_form
  - active_loop: null
  - action: utter_continue_credit_card_payment      # - once the money transfer is completed,
                                                    #   ask the user to return to the
                                                    #   credit card payment form
```

Figura 2.1: Definição de uma história nos ficheiros de configuração [2].

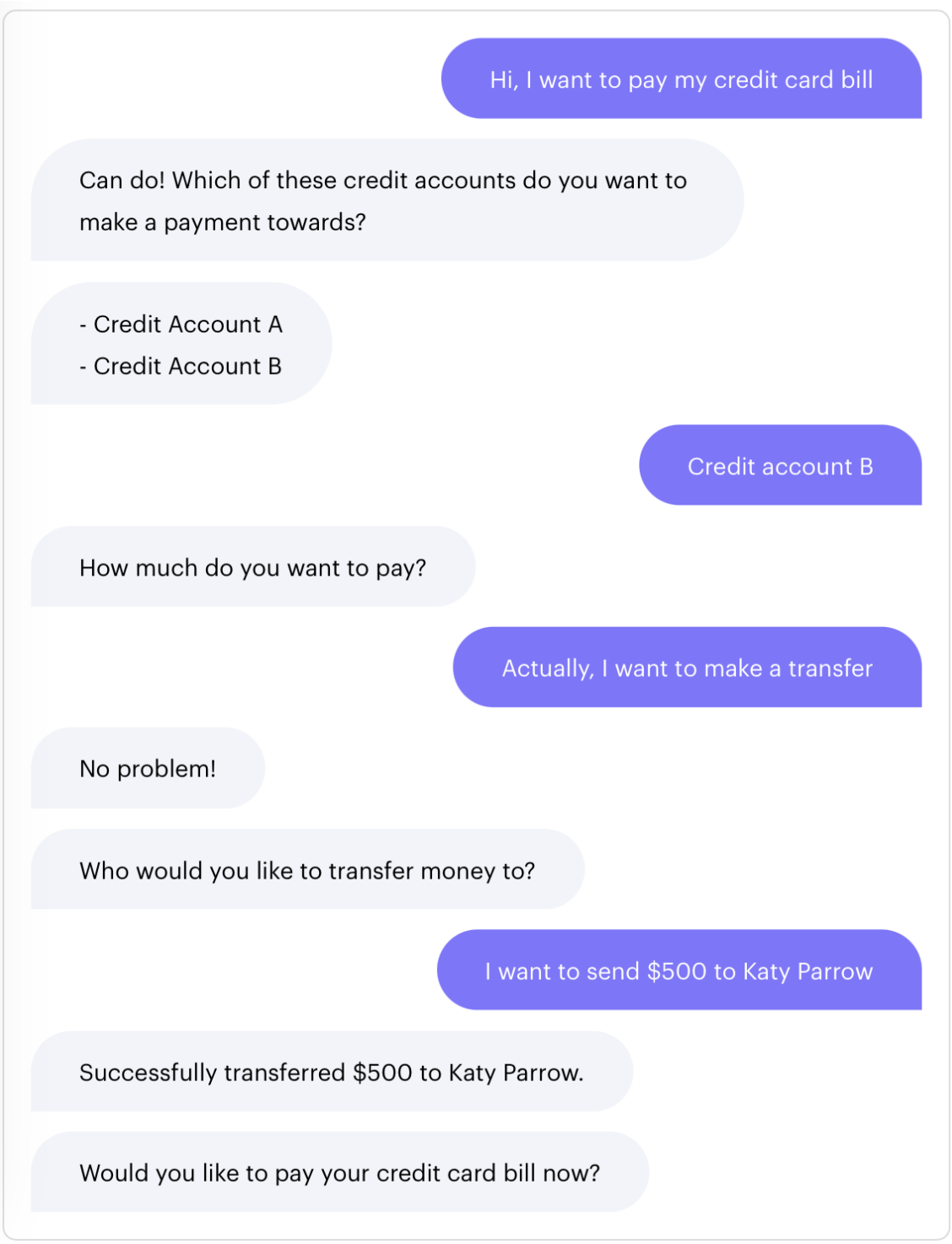


Figura 2.2: Aplicação da história da Figura 2.1 numa conversa real [2].

2.1.4 Slot

Os *slots* representam a memória do assistente virtual. Estes podem ou não estar associados a uma intenção e guardam a informação das entidades da mesma. Quando todos os *slots* associados a uma intenção estiverem preenchidos, a tarefa

está em condições de ser executada.

2.1.5 Formulário

Os formulários são um padrão de conversação em que o assistente vai pedindo parte da informação ao utilizador para a realização de uma determinada tarefa. O assistente apenas consegue realizar a tarefa após a obtenção de toda a informação necessária. A Figura 2.3 representa as iterações de um determinado formulário que foi ativado após a deteção da intenção. A figura mostra também a respetiva interação entre o assistente e o utilizador através da troca de mensagens. Podemos observar que à medida que o utilizador vai fornecendo a informação requisitada pelo assistente, os *slots* relativos a cada entidade vão sendo preenchidos. Assim que toda a informação necessária à realização da ação se encontrar reunida, o formulário é desativado e o assistente pode completar a ação.

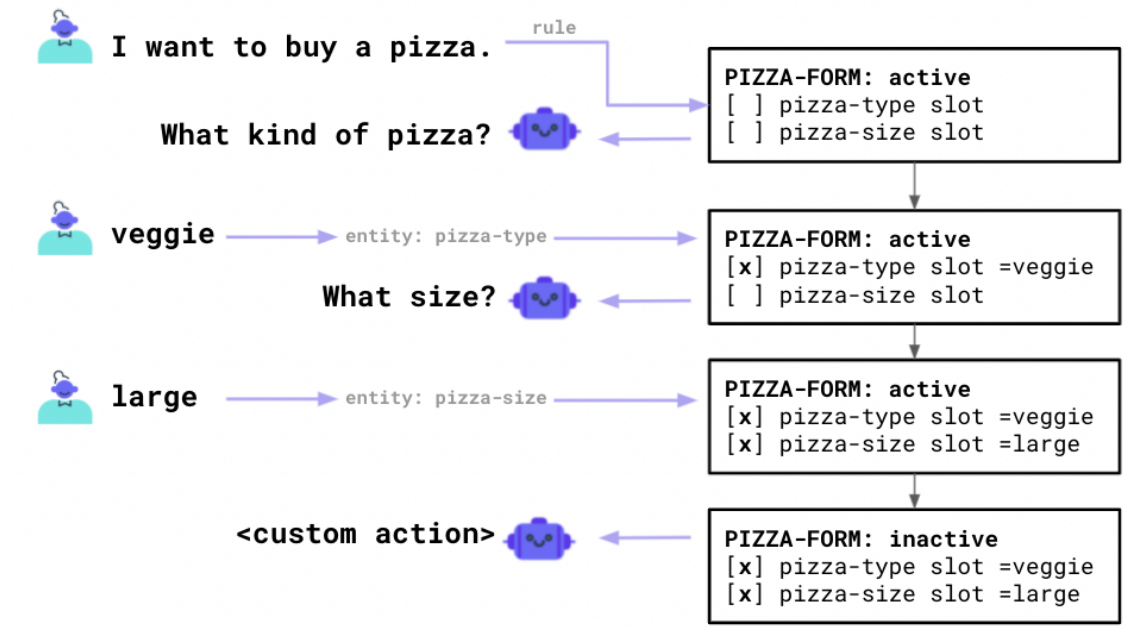


Figura 2.3: Exemplo da aplicação de um formulário numa conversa real [3].

2.2 Rasa

O Rasa [1] é uma ferramenta criada para o desenvolvimento de assistentes virtuais. Esta é constituída essencialmente por dois módulos - o núcleo e o módulo de *Natural Language Understanding* (NLU). O núcleo é responsável por tomar as decisões do assistente e gerir o fluxo da conversa, enquanto que o NLU é responsável pela classificação das mensagens recebidas. Tal como a maioria das ferramentas disponíveis para a implementação destes assistentes, possui uma estrutura baseada no conceito de intenção. Esta utiliza inteligência artificial para classificar e extrair a intenção na mensagem do utilizador de maneira a conseguir devolver

uma resposta consoante o que foi pedido. Estas intenções podem também ser associadas com entidades, que são complementos que acrescentam informação adicional à intenção do utilizador (cor, tamanho, data, entre outros).

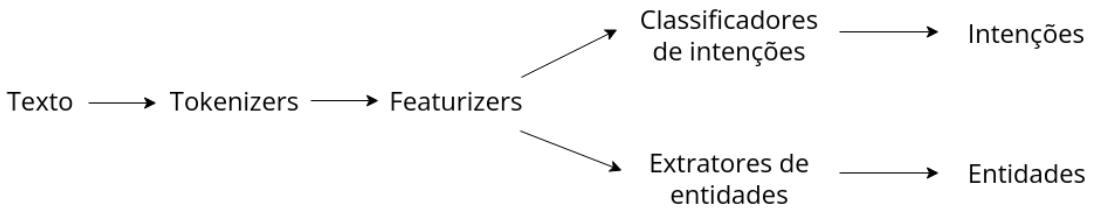


Figura 2.4: Fases da *pipeline* do Rasa relativas à classificação de texto.

De forma a obter as intenções e entidades, o Rasa segue uma determinada *pipeline* que é constituída por quatro tipo de componentes. Estes componentes podem ser visualizados na Figura 2.4 que representa as diversas etapas desde que a mensagem é recebida até aos resultados devolvidos pela classificação. Estes componentes são os seguintes:

- Os *Tokenizers* que são responsáveis por separar a frase introduzida pelo utilizador em *tokens*. Usualmente, cada *token* representa uma palavra. Isto é, quando o utilizador insere mais do que uma palavra, cada *token* corresponde a cada palavra e os espaços brancos entre elas atuam como separação. Alguns *tokenizers* estão preparados para realizar o processo de lematização que consiste em transformar uma palavra na sua forma canónica. Normalmente, estes algoritmos retornam uma lista de palavras prontas a serem usadas na próxima fase da *pipeline*.

- Os *Featurizers* que convertem a lista de palavras devolvida pela fase anterior em *features* numéricas para que estas possam ser compreendidas pelos modelos de *Machine Learning*. Estas subdividem-se em *sparse features* e *dense features*. As *sparse features* devolvem vetores que são constituídos maioritariamente por zeros, levando facilmente a um aumento na memória utilizada. Para erradicar este problema de memória, as *dense features* apenas guardam os valores que são diferentes de zero e as suas respetivas posições no vetor, possibilitando o treino de grandes *datasets*.

- Os classificadores de intenções que são modelos de classificação que recebem as *features* convertidas pelos *featurizers* e atribuem-lhes as respetivas classificações.

- Os classificadores de entidades que, tal como a fase anterior, são reponsáveis pela classificação das entidades contidas nas *features* recebidas como parâmetro.

Após termos uma visão mais concreta do que acontece dentro do Rasa, torna-se possível interpretar o fluxo da mensagem representado no exemplo da Figura 2.5.

Isto é, após o utilizador enviar a sua mensagem para o assistente virtual, o Lexia fará uso do Rasa para classificar as intenções e extrair as entidades que estejam presentes nesta. Consoante a intenção classificada pelo Rasa, o Lexia executará as ações necessárias devolvendo uma resposta ao utilizador. Como podemos ver pelo exemplo, na frase "Qual a meteorologia para amanhã?"a intenção é saber o estado da meteorologia, a entidade é "amanhã"e a resposta devolvida ao utilizador seria algo semelhante com a que se apresenta na figura abaixo.

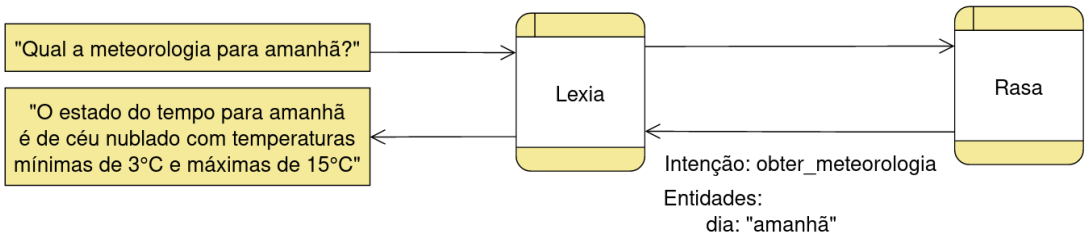


Figura 2.5: Exemplo do fluxo de uma mensagem entre o utilizador, Lexia e Rasa.

2.3 Lexia 1.0

Tal como já foi referido anteriormente, o Lexia é uma plataforma que foi implementada com o propósito de desenvolver assistentes virtuais personalizados. Este é responsável pelo fluxo da mensagem desde que é recebida pela plataforma até à realização da ação pedida pelo utilizador e devolução da respetiva resposta.

Para a realização da classificação das mensagens é utilizado o Rasa. Uma vez que o Lexia já faz a gestão do fluxo da mensagem, este utiliza apenas o módulo referente ao NLU do Rasa, que é o principal reponsável pela classificação das mensagens introduzidas pelo utilizador.

O Lexia é também responsável por realizar a deteção de idioma. Esta etapa é relevante para que possa ser utilizada a respetiva *pipeline* com os componentes configurados para essa linguagem específica e também para que o assistente possa responder no mesmo idioma utilizado pelo utilizador.

2.3.1 Arquitetura

Na Figura 2.6 podemos observar a arquitetura de todo o sistema, incluindo todos os serviços externos utilizados pelo Lexia. Esta arquitetura é constituída pelo Lexia - que será explicado com maior pormenor de seguida -, pelos canais responsáveis pela troca de mensagens, pelos modelos utilizados para a classificação das mensagens e por um repositório para armazenar as mensagens recebidas pelo utilizador.

Em primeiro lugar, a mensagem é introduzida pelo utilizador através de um dos canais disponíveis, nomeadamente a *Shell* e o *Microsoft Bot Framework*. Este último

é o canal mais utilizado, uma vez que a troca de mensagens é realizada através do Microsoft Teams o qual está ligado ao Lexia através de um *endpoint*. Após a mensagem ter sido enviada, esta é colocada numa fila de mensagens que serão processadas pelo Lexia. Inicialmente, será feita a deteção do idioma, para que a mensagem possa ser reencaminhada para o respetivo modelo de classificação (RasaModel PT ou RasaModel EN). De seguida, a mensagem e a respetiva classificação são armazenadas num repositório e é criada a tarefa consoante a intenção extraída pelo modelo de classificação. Por fim, essa tarefa é colocada numa fila de tarefas que serão processadas pelo respetivo executor. Cada executor é responsável pela realização da tarefa e por devolver uma resposta ao utilizador através do mesmo canal pelo qual foi recebida a mensagem inicial.

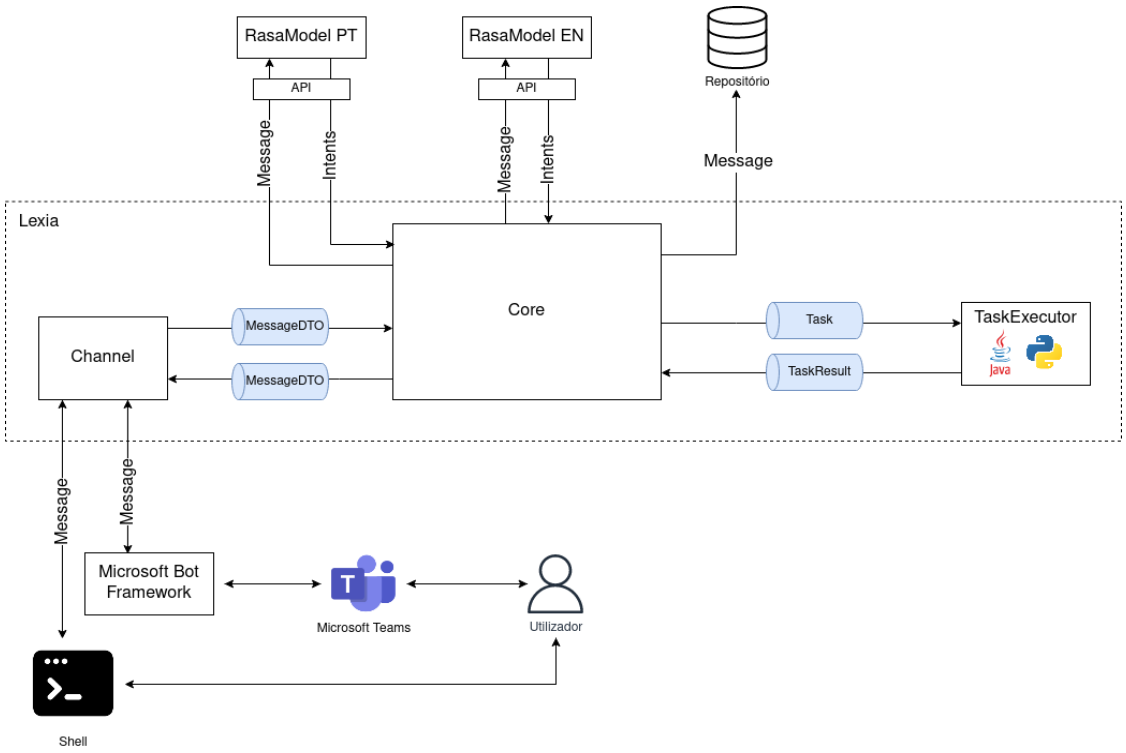


Figura 2.6: Arquitetura de alto nível do sistema.

O Lexia divide-se em quatro principais módulos que podem ser visualizados na Figura 2.7, bem como as interações entre eles. De seguida será explicado cada módulo detalhadamente.

- Módulo "Message": Este módulo (Figura 2.8) engloba três classes que se designam como *Message*, *Address* e *Channel*.

  A classe *Message* é responsável por guardar todos os dados referentes a uma determinada mensagem - como é o caso do emissor e recetor da mensagem -, o seu conteúdo, o idioma e a sessão do utilizador referente à mensagem.

  A classe *Address* guarda as informações acerca dos intervenientes numa mensagem (emissor e recetor), como por exemplo o canal utilizado e o ID do utilizador em questão.

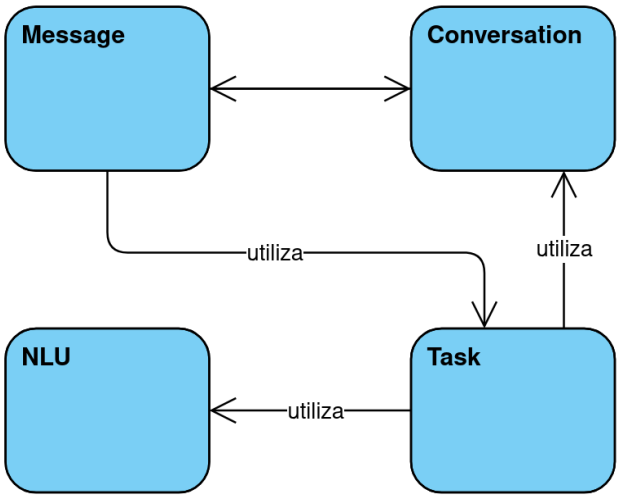


Figura 2.7: Arquitetura de domínio do Lexia.

A classe *Channel* contem informação sobre o canal a ser utilizado para o envio das mensagens. Este módulo comunica com os módulos "Conversation"e "Task", de maneira a obter a sessão do utilizador e a tarefa identificada na mensagem, respetivamente.

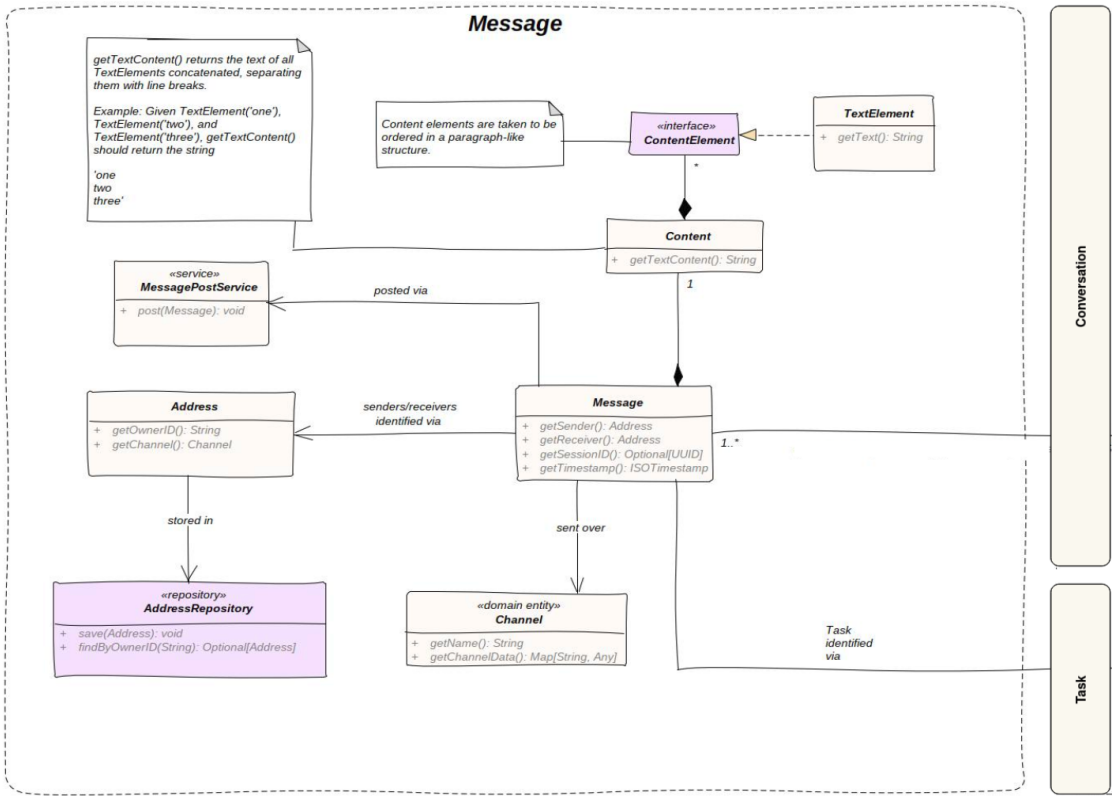


Figura 2.8: Módulo Message da plataforma Lexia 1.0.

- Módulo "Conversation": Embora este módulo (Figura 2.9) já existisse na

arquitetura do Lexia 1.0, ainda não estava implementado, havendo apenas uma prototipagem do mesmo. Este módulo era constituído pelas classes *Story*, *Conversation* e *Session*.

A classe *Story* estava prototipada com o intuito de guardar informações acerca de uma determinada história, como por exemplo, o nome e as intenções associadas a esta. As histórias são a representação de uma conversa num formato específico, onde as mensagens do utilizador são representadas como intenções, ao invés da mensagem de texto introduzida. Estas histórias são identificadas através do serviço *StoryIdentificationService* e as respetivas intenções são identificadas através do *StoryIntent*.

A classe *Conversation* seria responsável por manter a lista de mensagens de uma determinada conversa e utiliza a classe *StoryExecutionState* para identificar o estado atual da conversa, ou seja, em que ponto da história a conversa se situa.

Por fim, a classe *Session* guardaria o ID da sessão, bem como o último idioma utilizado.

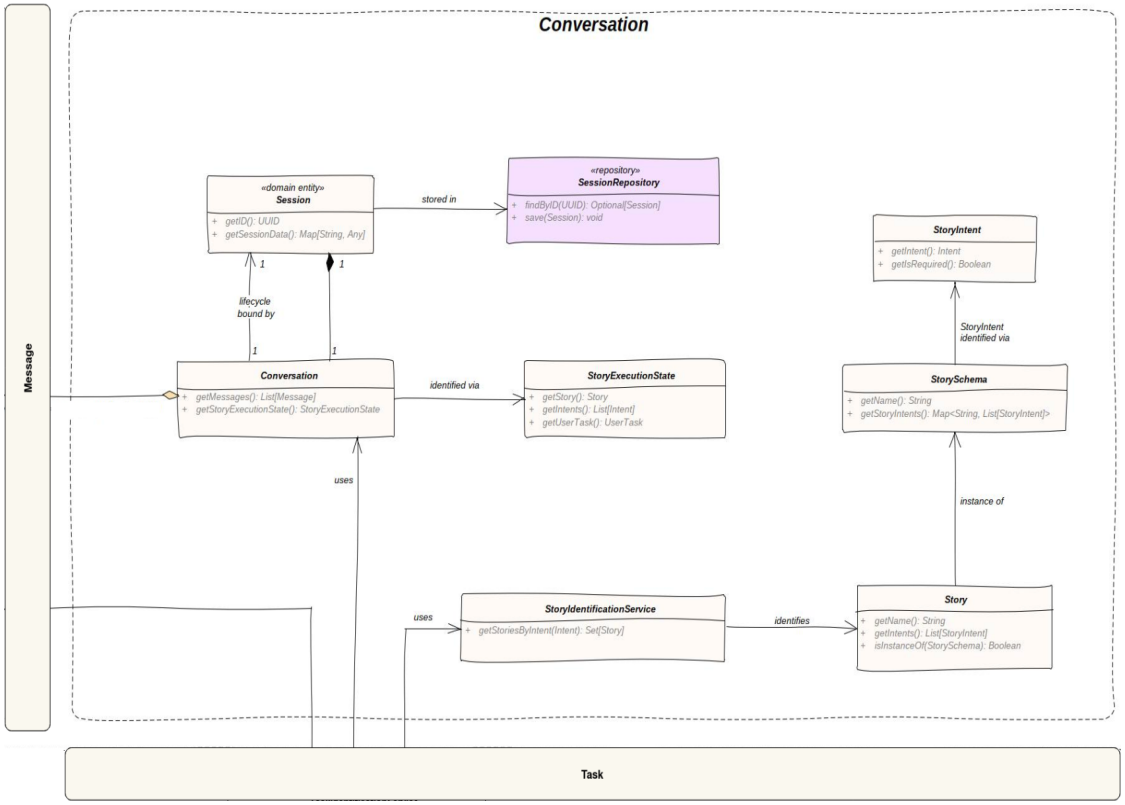


Figura 2.9: Módulo Conversation da plataforma Lexia 1.0.

- Módulo "Task": Este módulo é composto essencialmente pelas classes *Task* e *TaskIdentificationService*.

  A primeira contém a informação acerca da tarefa como é o caso do nome, a quem pertence a tarefa, a sessão, entre outros.

  A segunda é responsável por criar tarefas a partir das intenções classificadas na mensagem do utilizador, ou seja, faz uso da componente NLU para obter

as intenções a partir da mensagem recebida e cria os respetivos objetos da classe Task com as intenções que são devolvidas pelo NLU. Assim sendo, este módulo estabelece uma dependência direta com os módulos "Conversation"e "NLU".

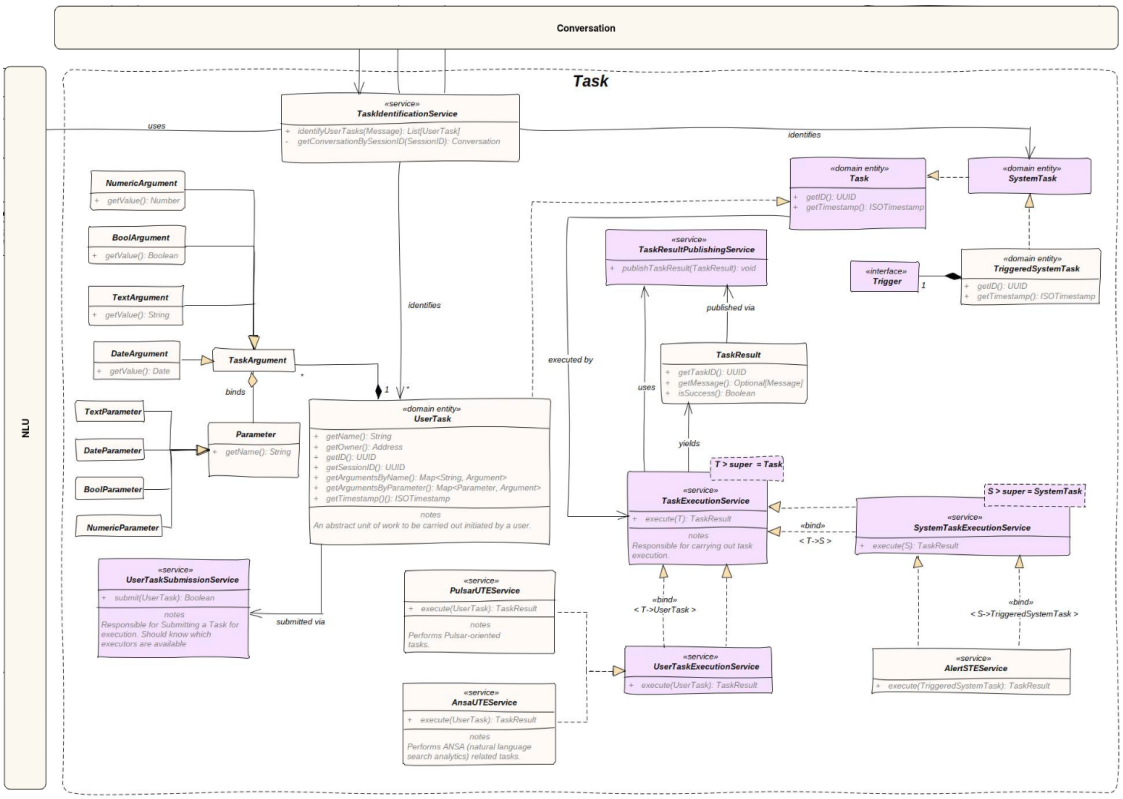


Figura 2.10: Módulo Task da plataforma Lexia 1.0.

- Módulo "NLU": engloba as classes *Intent*, *Entity*, *Language*, *NLUController* e *NLUService*.

  A classe *Intent* guarda informação sobre as intenções classificadas na mensagem do utilizador, essencialmente o nome e as entidades associadas a esta.

  A *Entity* é responsável por guardar o nome da entidade e o seu tipo.

  A classe *Language* guarda o nome e o código referente à linguagem (ex: PT, EN, etc.).

  Este módulo é gerido pela classe *NLUController* que controla toda a *pipeline*, recorrendo à classe *NLUService* para fazer a classificação das intenções e ao serviço de deteção de linguagem para a respetiva deteção do idioma da mensagem. É também neste módulo que é utilizado o Rasa Framework, uma vez que é necessário realizar todo o processamento de linguagem e a classificação de intenções e entidades.

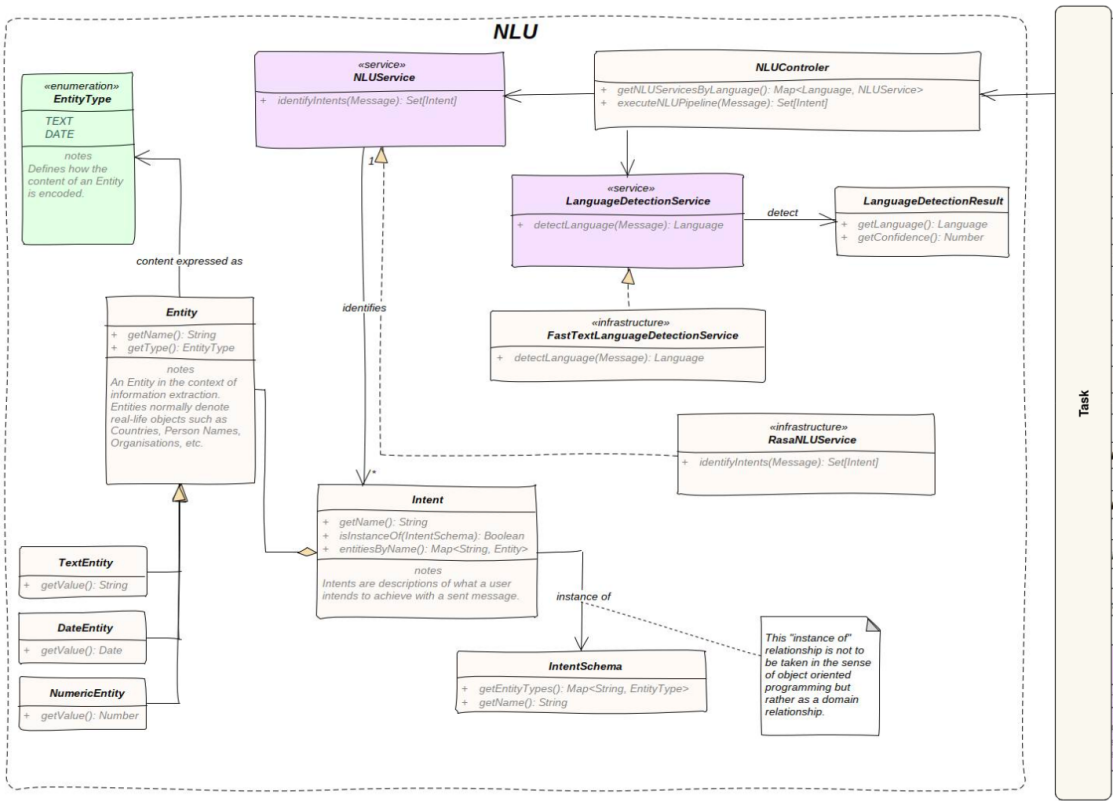


Figura 2.11: Módulo NLU da plataforma Lexia 1.0.

Toda a arquitetura do Lexia descrita anteriormente pode ser vista na íntegra na Figura A.1, com todas as classes devidamente representadas e as respetivas ligações entre os diferentes módulos.

Capítulo 3

Estado da arte

Esta secção aborda as diversas alternativas ao Rasa, comparando-as entre si numa tabela onde são listadas as diversas vantagens de cada uma. É também feito um estudo das funcionalidades já existentes na versão 1.7.0 do Rasa e as novas funcionalidades que foram introduzidas a partir desta versão. Por fim, são apresentadas breves descrições de modelos de linguagem, *tokenizers*, *featurizers*, classificadores de intenções e extratores de entidades introduzidos nas versões mais recentes do Rasa.

3.1 Alternativas ao Rasa

Com o grande crescimento e a consequente procura por estes assistentes e pelas tecnologias por estes utilizados, rapidamente começaram a surgir diversas opções no mercado de maneira a conseguir dar resposta às diferentes necessidades dos clientes.

3.1.1 Microsoft Bot Framework

Microsoft Bot Framework [4] é uma framework desenvolvida pela Microsoft. Esta possui um *Software Development Kit* (SDK) disponível para diferentes linguagens de programação, um núcleo capaz de fazer toda a parte de processamento de linguagem, assim como reconhecimento de entidades. Para além disso, possui uma interface gráfica de maneira a facilitar o desenvolvimento destes assistentes e é compatível com diferentes idiomas.

3.1.2 Wit.ai

Wit.ai [5] é uma plataforma adquirida pelo Facebook que permite também a criação de assistentes virtuais. Diponibiliza um SDK disponível para diferentes linguagens de programação e possui intenções e entidades embutidas na *framework*,

sem que seja necessário a implementação dessas por parte do programador. Possibilita a criação de aplicações para *wearable devices* e possui suporte para vários idiomas.

3.1.3 Dialogflow

Dialogflow [6] é uma plataforma desenvolvida pela Google que tem a capacidade de construir assistentes baseados tanto em formato de texto como também por voz. Para além de ser possível de ser integrado com plataformas de comunicação, como as restantes alternativas, permite também a integração com assistentes virtuais, como por exemplo a Google Assistant e a Amazon Alexa. Possui ainda uma interface de maneira a facilitar a implementação a pessoas inexperientes.

3.1.4 IBM Watson

IBM Watson [7] é uma pataforma concebida pela IBM. Esta é bastante completa com uma interface gráfica direcionada para pessoas inexperientes. Apresenta várias certificações ISO relacionadas com segurança e proteção de dados e possui ainda a possibilidade de analisar o número de mensagens das diversas conversas, nomeadamente, o número médio de mensagens por conversa e o número total de conversas, complementando com gráficos para melhor visualização da informação. É também possível integrar com outras funcionalidades implementadas pela IBM, entre as quais podemos destacar a transcrição de voz para texto, e vice-versa, e a tradução de idioma.

3.2 Funcionalidades do Rasa 1.7.0

A versão do Rasa utilizada pelo Lexia 1.0 possui diversas funcionalidades que suportam este assistente.

- É possível executar determinadas tarefas como a criação de um projeto, treino/teste de um modelo e iniciar o *interactive learning* a partir da linha de comandos.
- *Interactive learning* que permite criar novos dados de treino à medida que a conversa se desenvolve com o assistente. Durante essa conversa, há a possibilidade de visualizar a probabilidade associada a cada decisão tomada pelo assistente, de corrigir a classificação atribuída às inteções e de corrigir as ações do assistente caso haja essa necessidade.
- Opção para criar um *template* para uma nova mensagem introduzida pelo utilizador na funcionalidade *interactive learning*.
- Suporte para spaCy 2.1. O spaCy é uma biblioteca dedicada ao processamento de linguagem natural.

- Testar a *performance* das configurações NLU num determinado *dataset* de treino através do comando *rasa test nlu*.
- *FallbackPolicy* que pode ser configurada para ser ativada quando a confiança entre duas intenções previstas é muito próxima.
- Classificador *KeywordIntentClassifier* baseado apenas em palavras-chave para a classificação de intenções. Este classificador é apenas destinado a pequenos projetos ou projetos de iniciação.
- *ConveRTFeaturizer* baseado no modelo *ConveRT* lançado pela PolyAI. A PolyAI é uma companhia que se foca no desenvolvimento de assistentes virtuais a nível empresarial.
- *ConveRTTokenizer* que deve ser usado sempre que o *ConveRTFeaturizer* também for utilizado.

3.3 Novas Funcionalidades no Rasa3

Após a análise do estado atual da *framework* Rasa e das funcionalidades implementadas no Lexia, foi realizado um estudo das novas funcionalidades implementadas após a versão 1.7.0. Assim sendo, foi criada uma lista de novas funcionalidades presentes nas versões mais recentes que são relevantes para o Lexia 2.0.

1. Alterações obrigatórias:
    - *EmbeddingPolicy* foi removida. A nova denominação para esta política é TEDPolicy.
    - *EmbeddingIntentClassifier* e *CRFEntityExtractor* foram substituídos pelo *DIETClassifier*.
    - *RulePolicy* irá substituir as políticas *Mapping Policy*, *Fallback Policy*, *Two-Stage Fallback Policy* e *Form Policy*, que foram removidas.
    - *SklearnPolicy* foi removida, sendo necessário utilizar a política *TEDPolicy*.
    - Tornou-se necessário especificar o modelo de linguagem no ficheiro config.yml.
    - Tornou-se necessário fazer a conversão dos ficheiros em formato Markdown para YAML.

Para além das funcionalidades referidas anteriormente foram também adicionadas outras das quais não é necessário fazer uma migração e que podem ser aplicadas diretamente, nomeadamente:

- Adicionada a possibilidade de agrupar e atribuir funções às entidades.

- Adicionado o comando *rasa data convert nlu | core -f yaml* de maneira a facilitar a conversão dos ficheiros de treino do formato Markdown para YAML.
- Adicionada a funcionalidade de treino incremental. A partir do comando *rasa train –finetune* é possível calibrar o modelo com novos dados de treino sem que seja necessário retreiná-lo novamente de raíz.
- Atualização para que o Rasa seja compatível com o spaCy 3.0, apresentando assim suporte de mais funcionalidades para mais línguas.
- Introdução de uma nova política denominada *UnexpecTEDIntentPolicy* com o intuito de adicionar a capacidade de reação ao assistente sempre que o utilizador insere algo fora do contexto da conversa.
- Atualização para o Tensorflow 2.6. Este quebra a compatibilidade com os modelos anteriores, sendo necessário retreinar os modelos.

3.4 Comparação entre as alternativas

A principal razão para o Rasa ser uma excelente opção para o desenvolvimento desta plataforma é que este permite ter bastante controlo do NLU, ao contrário das restantes opções, cujo *backend* é bastante limitado e desconhecido. Embora sejam apresentadas outras opções *open-source*, nenhuma delas oferece o nível de customização do Rasa onde, para além de ser possível alterar os diversos componentes que constituem a *pipeline* - como modelos de linguagem, *tokenizers*, *featurizers*, classificadores e extratores de entidades -, é também possível adicionar componentes customizados de maneira a executar tarefas específicas ou adicionar modelos que não estejam implementados no Rasa.

Enquanto que as restantes alternativas estão limitadas devido ao facto de serem executadas na sua própria *cloud*, esta *framework* tem a vantagem de poder ser instalada em qualquer sistema operativo, tornando-a uma alternativa mais versátil, e podendo ser desta forma executada localmente guardando todos os dados sem necessitar de *software* de terceiros. Para além disso, a primeira versão do Lexia já utilizava o Rasa, pelo que implementar todas as funcionalidades já existentes numa nova plataforma iria levar a um aumento do tempo e esforço necessários não compensando a nível de funcionalidade.

Para complementar, a *framework* possui ainda uma excelente documentação, revelando-se bastante completa e estruturada. O Rasa dispõe também de um *blog* onde são publicados diversos artigos - tutoriais, descrição das funcionalidades introduzidas em novas atualizações e artigos relativos ao estado da arte - e de um Fórum onde são respondidas diversas perguntas relacionadas com a plataforma bem como os seus componentes. Todos estes pontos facilitam bastante o processo de implementação, permitindo também desenvolver um produto com a devida qualidade.

Por fim, um ponto fulcral é também a importância que o Rasa dá ao estado da arte e ao avanço do mesmo. Estes constituem uma equipa de investigação que

se foca em manter o Rasa atualizado e em estudar e evoluir novos modelos a fim de obter resultados de classificação competentes e de contribuir para o avanço na área de processamento de linguagem natural.

A Tabela 3.1 apresenta uma comparação mais detalhada das alternativas referidas anteriormente.

3.5 Treino incremental

O treino incremental é uma funcionalidade que permite a calibração de um determinado modelo após a adição de novos dados de treino sem que este tenha que ser treinado novamente de raíz. Este modo de treino permite reduzir significativamente o tempo de treino de um determinado modelo, uma vez que utiliza um menor número de *epochs* para calibrar o mesmo. Ou seja, é utilizado um modelo já treinado anteriormente e é treinado novamente com os novos dados de treino mas com um menor número de *epochs*. Este treino utiliza todo o *dataset*, incluindo os dados de treino antigos, de forma a evitar que o modelo esqueça exemplos de treino do modelo original.

Para que este processo seja possível, é necessário que as configurações utilizadas durante o treino do modelo original sejam as mesmas utilizadas para o treino do novo modelo calibrado. O treino incremental também não permite a adição de novas classes ao *dataset*, uma vez que para que estas fossem reconhecidas teriam de ser feitas alterações ao modelo já existente. Assim sendo, esta funcionalidade permite apenas adicionar novos dados de treino a classes já existentes no modelo original.

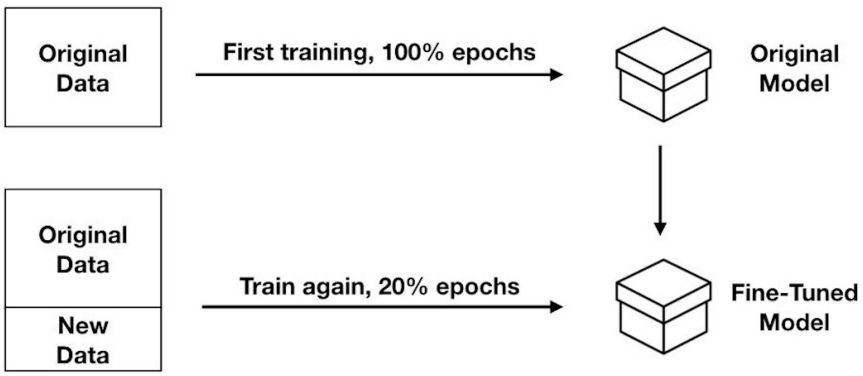


Figura 3.1: Exemplo relativo ao treino incremental.

3.6 Modelos de linguagem

Nesta secção serão apresentados novos modelos que foram introduzidos nas versões mais recentes do Rasa, descrevendo um pouco o seu funcionamento.

3.6.1 BERT

O *Bidirectional Encoder Representations from Transformers* (BERT) [8] é um modelo de linguagem publicado pelos investigadores da Google que se baseia na arquitetura de um Transformer [9]. Este modelo é capaz de aprender relações contextuais entre palavras, uma vez que, ao contrário de outros modelos, utiliza um treino bidirecional de maneira a conseguir fazer as associações entre as palavras. Deste modo, este modelo é capaz de determinar o contexto de uma determinada palavra tendo em conta o que o rodeia, tanto à esquerda como à direita.

O BERT possui duas etapas essenciais durante a sua fase de treino: pré-treino e *fine-tuning*. Para o pré-treino destes modelos é utilizada aprendizagem não supervisionada em duas tarefas, com o objetivo de construir uma camada base de conhecimento para o modelo. A primeira tarefa é denominada por *Masked Language Model* (MLM) e o seu funcionamento baseia-se em mascarar um *token* da frase recebida como parâmetro para que o modelo preveja qual a palavra contida nessa posição, ou seja, é retirada uma palavra aleatória da frase para que o modelo possa prever essa palavra com base no restante contexto da frase. A segunda tarefa é a *Next Sentence Prediction* (NSP) e tem como objetivo permitir ao modelo prever se duas frases estão relacionadas e se apresentam uma sequência lógica. Para esta fase de treino são utilizados grandes corpos de texto como dados não supervisionados, de maneira a fornecer informação suficiente ao modelo para que este aprenda as relações semânticas entre as palavras. A segunda fase de treino é baseada em aprendizagem supervisionada, permitindo melhorar o modelo e especificá-lo para um certo conjunto de dados. Durante esta fase são fornecidos parâmetros de entrada e os respetivos parâmetros de saída para que o modelo possa ser aperfeiçoado e adaptado às necessidades dos utilizadores. Como é demonstrado no artigo, este modelo consegue atingir melhores resultados de precisão quando comparado com outros algoritmos [8] [9].

3.6.2 RoBERTa

O RoBERTa é um modelo de linguagem baseado no BERT, que foi apresentado por um grupo de investigadores do Facebook com o objetivo de avançar o Estado da Arte no que diz respeito ao processamento de linguagem natural. De maneira a tornar este avanço possível, o modelo foi treinado sobre uma maior quantidade de dados e com um maior número de *batches*, uma vez que havia espaço para progredir nesse aspeto e melhorar o desempenho do modelo. Foi também removida a tarefa NSP no modelo RoBERTa, uma vez que na pior das hipóteses igualava o resultado obtido com o modelo BERT e em alguns casos esse resultado melhorava minimamente. Por fim, introduziram o conceito de *dynamic masking*. Ao

contrário do modelo BERT que seleciona os *tokens* a serem mascarados antes da fase de treino, permanecendo os mesmos durante todas as iterações, no modelo RoBERTa estes *tokens* vão sendo alterados durante a fase de treino, evitando mascarar o mesmo *token* múltiplas vezes.Deste modo, o RoBERTa ultrapassa os resultados obtidos com o original BERT, aproximando-se de uma precisão de quase 90% [10].

3.7 Tokenizers

O *tokenizer* mais utilizado será provavelmente o *WhiteSpaceTokenizer* [11] que, por sua vez, é também dos mais simples, uma vez que utiliza os espaços brancos para separar a frase em diversos *tokens*. Para além deste, existem outros *tokenizers* como é o caso do *JiebaTokenizer* [12] que é específico para o processamento do idioma chinês, devido aos seus caracteres especiais. Por fim, existem ainda outros dois algoritmos para a separação dos *tokens* que necessitam da utilização dos modelos de linguagem fornecidos pelas respetivas bibliotecas. Estes são o *MitieTokenizer* [13] e o *SpacyTokenizer* [14] que necessitam de ser utilizados com os modelos de linguagem MitieNLP e SpacyNLP, respetivamente.

3.8 Featurizers

Tal como para a secção anterior, as bibliotecas Mitie e Spacy também fornecem algoritmos [15] [16] para transformar os *tokens* recebidos da fase anteior em *features*. Existem também o *RegexFeaturizer* [17] que tal como o nome indica, utiliza expressões regulares para para a conversão das *features*. Entre outros algoritmos, o Rasa disponibiliza também o *LanguageModelFeaturizer* [18] que utiliza modelos de linguagem pré-treinados (BERT, RoBERTa, etc.) para criar representações vetoriais da mensagem do utilizador.

3.9 Classificadores de Intenções

Existem vários classificadores de intenções que podem ser utilizados durante o processamento de linguagem natural. Um exemplo destes classificadores é o *MitieIntentClassifier* [19] que utiliza uma *Support Vector Machine* (SVM) multiclasse com um *kernel* linear para a classificação das intenções. Outro exemplo, é o *SklearnIntentClassifier* [20] que treina uma SVM linear e que utiliza pesquisa em grelha (*Grid Search*) para a otimização dos parâmetros de treino. Esta pesquisa em grelha consiste na construção de uma grelha com as combinações entre os diversos parâmetros necessários para o treino de uma SVM e é escolhida aquela que obtém melhor resultado. Entre estes parâmetros estão os valores de C, Gama e as funções de Kernel a serem utilizadas pelo modelo. Durante a classificação, para além das intenções retiradas da frase são também devolvidas como parâmetro de

saída as restantes intenções que não foram consideradas presentes na frase e o respetivo grau de confiança.

3.10 Extratores de Entidades

Para a extração de entidades são disponibilizados os algoritmos fornecidos pelas bibliotecas Mitie e Spacy, tal como para as secções anteriores [21] [22]. Para além destes, existem outros componentes como é o caso do *DucklingEntityExtractor* [23] que permite extrair as entidades mais comuns da mensagem do utilizador, como por exemplo datas, números, distâncias, contactos, etc. Após o reconhecimento das entidades, este componente normaliza a informação, convertendo-os em dados estruturados.

3.11 Classificador DIET

Esta secção tem o objetivo de apresentar um novo classificador introduzido nas versões mais recentes do Rasa.

Assim sendo, o classificador introduzido foi o *Dual Intent Entity Transformer* (DIET) [24]. Este apresenta uma arquitetura multitarefa para a classificação de intenções e reconhecimento de entidades, permitindo obter as intenções e entidades a partir de um único algoritmo. Esta arquitetura segue um modelo baseado num Transformer, que é utilizado para as duas tarefas referidas anteriormente, no entanto este pode ser configurado para realizar apenas umas das tarefas.

Alguns modelos de linguagem são bastante pesados no sentido em que necessitam de um grande poder computacional. No entanto, o DIET supera o Estado da Arte e apresenta-se seis vezes mais rápido a treinar que os restantes modelos, igualando os resultados de precisão obtidos por estes. Para além disso, apresenta uma arquitetura modular que permite aos programadores uma maior flexibilidade na realização de experimentações [25].

O erro total calculado pelo modelo é resultado da soma entre os erros obtidos na classificação de intenções, na extração de entidades e no processo de previsão do *token* mascarado. Este último vem desativado por defeito e é apenas aconselhada a sua utilização caso o dataset utilizado seja bastante grande de maneira a que o modelo se adapte aos dados e se possa espeificar no domínio em questão. É também recomendada a utilização da extração de entidades e classificação de intenções em simultâneo, mesmo que se queira realizar apenas uma das tarefas, para não influenciar a classificação final, uma vez que o erro total depende da soma dos dois erros [26].

Tal como todos os restantes modelos e classificadores presentes no Rasa, é possível alterar as configurações utilizadas durante a fase de treino do modelo, tal como o tipo de função a ser utilizada para o cálculo do erro, o número de iterações para treinar o modelo, o número de camadas escondidas, entre muitos

outros. Este pode ser utilizado com o modelo de linguagem BERT, conseguindo bons resultados de precisão e tempos de treino relativamente baixos [27].

3.12 Sumário

O Rasa apresenta-se como uma excelente opção para o desenvolvimento da segunda versão do Lexia devido à sua elevada personalização e facilidade de integração e instalação em qualquer máquina ou sistema operativo. Este disponibiliza diversos componentes e modelos para cada fase da *pipeline*, de maneira a realizar o processamento e a respetiva classificação das mensagens. Tem também a vantagem de utilizar apenas um classificador (DIET) para a realização das tarefas de classificação de intenções e extração de entidades. Para além dos modelos disponibilizados, permite também adicionar componentes personalizados e específicos para a realização de determinadas tarefas. Estes novos modelos introduzidos nas versões mais recentes acompanham os desenvolvimentos mais recentes do estado da arte, demonstrando-se bastante competentes e capazes de melhorar a classificação de mensagens e por sua vez a conversação com o utilizador.

	Rasa	MS Bot Framework	Wit.ai	Dialogflow	IBM Watson
Entidades pré treinadas	×	✓	✓	×	×
Agentes pré treinados	×	×	×	✓	×
SDK para diferentes linguagens	×	✓	×	✓	✓
Open-source	✓	✓	✓	×	×
Suporte para várias línguas	✓	✓	✓	✓	✓
Host num servidor local	✓	×	×	×	×
Pipeline customizável	✓	×	×	×	×
Componentes customizáveis	✓	×	×	×	×

Tabela 3.1: Comparação entre o Rasa, Microsoft Bot Framework, Wit.ai, Dialogflow e IBM Watson.

Capítulo 4

Abordagem

Neste capítulo são apresentados os requisitos funcionais e não funcionais do *software* e de seguida as alterações que serão realizadas pelo aluno na arquitetura do Lexia durante o segundo semestre. É também feita a análise de riscos caracterizando cada um com o seu impacto, probabilidade e o respetivo plano de mitigação.

4.1 Requisitos Funcionais

Nesta secção serão abordados os requisitos funcionais de maneira a cumprir com as necessidades e o propósito do projeto. Estes requisitos estão listados na Tabela 4.1 que se apresenta abaixo.

O primeiro requisito refere-se à capacidade de interpretação do assistente, uma vez que este deve conseguir fazer as devidas associações entre as entidades e o significado que estas têm no contexto da frase. Assim sendo, os assistentes desenvolvidos devem conseguir agrupar as entidades e atribuir-lhes a respetiva função.

Em alguns casos pode ser necessário pedir informação ao utilizador de maneira a realizar alguma ação ou enviar parâmetros que sejam necessários para executar determinadas funções. O segundo requisito refere-se à implementação de formulários de maneira a tornar mais simples a recolha dessa informação para que possa ser utilizada para a execução da respetiva tarefa.

O terceiro requisito está relacionado com a integração de histórias nos assistentes e a possibilidade de ver o estado da conversa, podendo aceder às diversas intenções do utilizador ao longo das mensagens trocadas com o assistente.

Para tornar os assistentes mais intuitivos, foi também ser necessário implementar uma memória onde o assistente pode guardar informações que vai obtendo durante a conversa com o utilizador. Esta funcionalidade está relacionada com o quarto requisito funcional apresentado na tabela.

O seguinte requisito refere-se à integração do treino incremental, com o objetivo

de facilitar a adição de novos dados de treino ao modelo. Deste modo, será possível economizar tempo durante o desenvolvimento dos assistentes, uma vez que não será necessário treinar o modelo de raiz sempre que for preciso adicionar novos exemplos de treino ao assistente virtual.

Por fim, o último requisito relaciona-se com o aperfeiçoamento da deteção de idioma. Este requisito opcional acabou por não ser cumprido, uma vez que não se apresentava como prioritário e, por questões de tempo, houve a necessidade de dedicar o tempo disponível a tarefas mais relevantes.

Nome	Descrição
Melhorar a classificação de entidades	Atribuir funções e grupos às entidades de maneira a tornar os assistentes desenvolvidos com o Lexia mais completos e inteligentes.
Integração de formulários	De maneira a obter os parâmetros necessários para a execução de determinadas funções, pode ser necessário que o assistente virtual peça informações ao utilizador através do uso de formulários.
Integração de histórias	A integração de histórias permite aos assistentes manter informação acerca da conversa e um contexto da mesma.
Adição de Slots	Os Slots podem ser vistos como a memória do assistente. Estes podem ser utilizados para guardar informações como o nome do utilizador, a sua cidade, etc. Podem também ser definidos para terem influência na conversa e nas ações tomadas pelo assistente.
Adicionar a funcionalidade de treino incremental	Integração da funcionalidade de treino incremental para que possam ser adicionados novos dados aos assistentes sem haver a necessidade de treinar o modelo novamente de raiz.
(Opcional) Melhorar a deteção de idioma	Deve ser melhorada a funcionalidade de deteção de linguagem, uma vez que esta apresenta falhas quando as mensagens introduzidas são demasiado curtas.

Tabela 4.1: Requisitos funcionais.

4.2 Requisitos Não Funcionais

Enquanto que os requisitos funcionais estão mais relacionados com as funcionalidades do *software* desenvolvido, os requisitos não funcionais estão relacionados com o uso da aplicação - muitas vezes associados com as tecnologias envolvidas -, o desempenho do programa ou a usabilidade. Estes requisitos estão ligados à forma como o *software* desempenha as suas funções e muitas vezes restringem os requisitos funcionais.

O primeiro requisito não funcional foi a necessidade de utilizar a linguagem de programação Kotlin, uma vez que a primeira versão do Lexia foi também desenvolvida com esta linguagem. Deste modo, foi mantido o mesmo critério, utilizando esta linguagem para o desenvolvimento da nova versão.

O outro requisito não funcional foi a comparação de *pipelines* para que pudessem ser escolhidas aquelas que obtivessem melhores resultados, não comprometendo a *performance* dos assistentes.

Nome	Descrição
Linguagem de programamção Kotlin	De maneira a seguir com a primeira versão do Lexia, foi necessário utilizar a linguagem de programação Kotlin.
Comparação entre *pipelines*	Foi feita uma comparação do desempenho das diversas *pipelines* implementadas, a fim de utilizar aquela que obteve melhor resultado. Esta comparação foi realizada através da linha de comandos, uma vez que o Rasa possui um comando para tal.

Tabela 4.2: Requisitos não funcionais.

4.3 Análise de Riscos

O projeto esteve suscetível a riscos que poderiam afetar o produto final de uma forma negativa. Tal como em todos os projetos de *software*, esta é uma análise que deve ser feita para mitigar os efeitos dos riscos aos quais o projeto está sujeito. Esta análise deve ser contínua para que seja mantida uma noção do que pode acontecer no decorrer do mesmo e para que seja aplicado um plano de mitigação caso tais riscos venham a acontecer.

Cada risco tem um impacto de acontecimento que caracteriza o efeito resultante no projeto, bem como no sucesso do mesmo. O impacto pode ser classificado como Alto quando o risco tem a possibilidade de impedir a conclusão do projeto

com sucesso; Médio quando o risco permite terminar o projeto com muitas dificuldades; e Baixo quando este praticamente não afeta os critérios de sucesso ou a conclusão do projeto. Os riscos têm também uma probabilidade de acontecer, que se classifica como Alta quando a probabilidade de este ocorrer é acima dos 70%; Média quando a probabilidade se situa entre 40% e 70%; e Baixa quando a probabilidade está abaixo dos 40%. Por fim, estes devem ter um plano de mitigação que é a estratégia que deve ser adotada para cada risco de maneira a diminuir o impacto deste. Quanto maiores forem o impacto e a probabilidade, tipicamente mais perigoso será o risco associado e melhor deverá ser a preparação do plano de mitigação para que este possa facilmente ser posto em prática e afetar o mínimo possível o projeto final.

Risco	Familiarização com a plataforma Lexia
Impacto	Alto
Probabilidade	Baixa
Plano de mitigação	Esclarecer as dúvidas com os orientadores ou alguém da equipa do Lexia

Tabela 4.3: Risco 1 - Familiarização com a plataforma Lexia.

Risco	Familiarização com a linguagem de programação Kotlin
Impacto	Alto
Probabilidade	Baixa
Plano de mitigação	Estudo da linguagem antes da fase de implementação e esclarecimento das dúvidas com membros da Critical Software

Tabela 4.4: Risco 2 - Familiarização com a linguagem de programação Kotlin.

Risco	Tarefas não completadas dentro dos prazos
Impacto	Alto
Probabilidade	Média
Plano de mitigação	Rever periodicamente o plano de tarefas e os tempos estimados

Tabela 4.5: Risco 3 - Tarefas não completadas dentro dos prazos.

Risco	Alta complexidade do projeto, não devidamente percebida nas etapas iniciais
Impacto	Alto
Probabilidade	Baixa
Plano de mitigação	Esclarecer possíveis dúvidas e aplicar tempo extra na compreensão do projeto

Tabela 4.6: Risco 4 - Alta complexidade do projeto.

Risco	Erro nas estimativas dos tempos das tarefas
Impacto	Médio
Probabilidade	Média
Plano de mitigação	Avaliar estimativas anteriores de maneira a estimar corretamente os tempos das próximas tarefas

Tabela 4.7: Risco 5 - Erro nas estimativas dos tempos das tarefas.

Risco	Problemas relacionados com a plataforma *open-source* (Rasa)
Impacto	Alto
Probabilidade	Baixa
Plano de mitigação	Avaliar alternativas caso haja necessidade de optar por outra opção

Tabela 4.8: Risco 6 - Problemas relacionados com a plataforma *open-source* (Rasa).

4.4 Casos de Uso

Os casos de uso representam uma lista de ações que descrevem a interação de um determinado utilizador ou alguém que representa uma função no sistema e o sistema em si. Estes representantes denominam-se de atores e são os responsáveis por executar a ação que desencadeia a resposta do sistema.

A criação de casos de uso representa uma função importante pois ajudam a perceber a maneira como o sistema deve agir em diversas situações. Este comportamento esperado serve de guia para ser comparado com o comportamento real do sistema.

Assim sendo, são apresentados diferentes casos de uso desde a classificação de intenções até à maneira como a informação é gerida pelo assistente.

Os casos de uso identificados encontram-se apresentados nas tabelas seguintes.

Nome	Manter estado da conversa.
Descrição	Os assistentes devem conseguir manter estado da conversa para que não haja necessidade de pedir informação repetida.
Ator	Utilizador.
Pré-condição	O utilizador deverá já ter fornecido informação de maneira a haver um contexto da conversa.
Fluxo	O utilizador envia uma mensagem ao assistente. O assistente executa a ação e devolve uma resposta sem haver a necessidade de pedir a informação que já tinha sido fornecida em interações anteriores.

Tabela 4.9: Caso de Uso 1 - Manter estado da conversa.

Nome	Pedir informação em falta.
Descrição	Os assistentes devem analisar se no fluxo da conversação já reúnem toda a informação necessária para realizar a ação pretendida pelo utilizador.
Ator	Utilizador
Pré-condição	Não tem.
Fluxo	O utilizador envia uma mensagem ao assistente. O assistente verifica se existe informação em falta e pede essa informação ao utilizador. Após o utilizador fornecer toda a informação necessária, o assistente executa a ação e devolve uma resposta ao utilizador.

Tabela 4.10: Caso de Uso 2 - Pedir informação em falta.

Nome	Assistente classifica a intenção corretamente
Descrição	Os assistentes devem conseguir classificar a intenção do utilizador corretamente para que possam realizar a ação necessária.
Ator	Utilizador
Pré-condição	Não tem.
Fluxo	O utilizador envia uma mensagem para o assistente. De seguida, o assistente classifica a intenção contida na mensagem e devolve a devida resposta ao utilizador.

Tabela 4.11: Caso de Uso 3 - Assistente classifica a intenção corretamente.

Nome	Assistente não classifica a intenção corretamente
Descrição	Caso o assitente não consiga classificar corretamente a intenção contida na mensagem do utilizador este deve enviar uma mensagem de erro para que o utilizador possa introduzir novamente a sua intenção.
Ator	Utilizador
Pré-condição	Não tem.
Fluxo	O utilizador envia uma mensagem para o assistente. De seguida, o assistente não consegue classificar a intenção corretamente e devolve uma resposta ao utilizador para que este volte a reescrever a sua mensagem de maneira a que o assistente consiga perceber.

Tabela 4.12: Caso de Uso 4 - Assistente não classifica a intenção corretamente.

Nome	Adição de novos exemplos de treino ao assistente sem a necessidade de treinar o modelo de raiz
Descrição	O programador poderá melhorar o reconhecimento de intenções através da funcionalidade de treino incremental, onde esta utiliza a técnica de *fine-tuning* para ajustar o modelo sem haver necessidade de o treinar novamente de raiz.
Ator	Programador
Pré-condição	Não tem
Fluxo	O programador adiciona os novos dados de treino relativos às intenções. De seguida, utiliza o método de treino incremental para adaptar o modelo aos novos dados de treino introduzidos.

Tabela 4.13: Caso de Uso 5 - Adição de novas intenções ao assistente sem a necessidade de treinar o modelo de raiz.

Capítulo 5

Metodologia e Planeamento

5.1 Metodologia

Durante o primeiro semestre foram realizadas reuniões mensais com o orientador do DEI, Professor Ernesto Costa, com a presença dos orientadores por parte da Critical Software, Engenheiro Rui Lopes e Engenheira Ana Guarino, de maneira a fazer o acompanhamento do aluno e verificar se os objetivos estabelecidos estavam a ser cumpridos. Para além destas reuniões mensais com orientador do departamento, houve também reuniões apenas com os orientadores da Critical Software para esclarecimento de dúvidas e também de acompanhamento do trabalho e respetivo progresso.

Relativamente ao segundo semestre, a metodologia adotada durante o desenvolvimento foi baseada nas metodologias agéis. Estas metodologias baseiam-se no desenvolvimento do produto de forma incremental, permitindo ir observando a evolução do produto ao longo do tempo e permitem também uma maior flexibilidade no decorrer do mesmo, conseguindo responder mais facilmente a mudanças. A implementação das funcionalidades foi feita de forma gradual, mantendo-se as reuniões regularmente para acompanhamento e esclarecimento de dúvidas.

5.2 Planeamento

Nesta secção é apresentado o planeamento inicial das tarefas e a duração final das mesmas no decorrer do semestre. Os diagramas referentes ao primeiro semestre encontram-se nas Figuras 5.1 e 5.2.

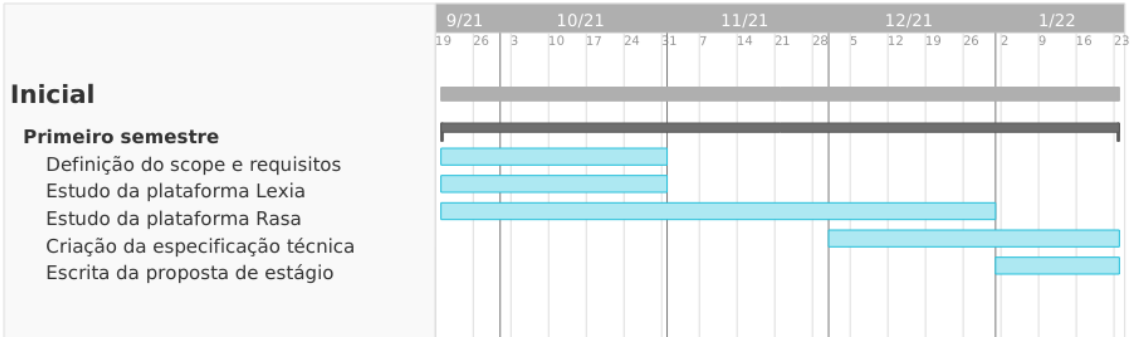


Figura 5.1: Diagrama de Gantt inicial relativo ao primeiro semestre.

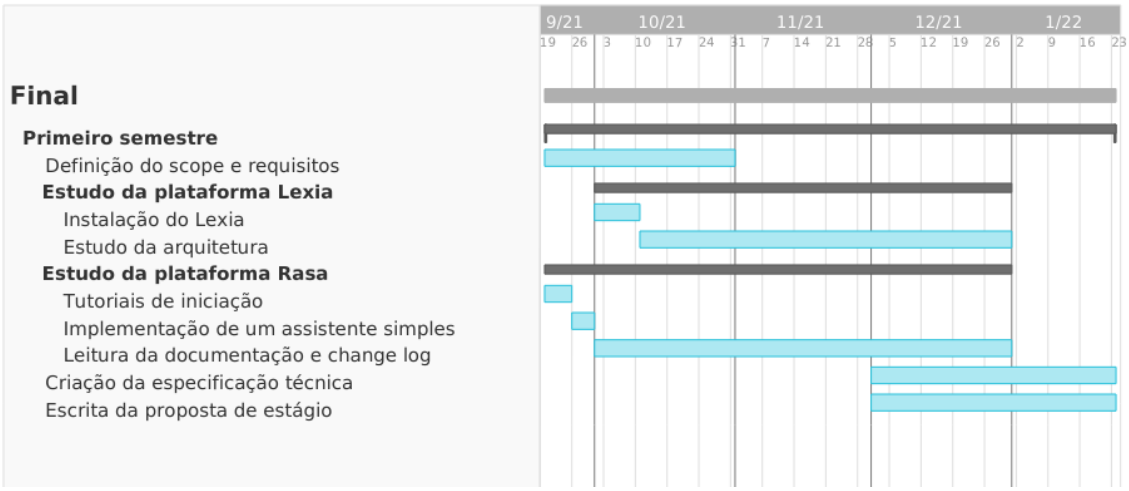


Figura 5.2: Diagrama de Gantt final relativo ao primeiro semestre.

Podemos observar que houve ligeiras alterações entre o planeado e a duração final das tarefas relativas ao primeiro semestre. Após o término da tarefa que corresponde à definição do scope e requisitos dei início ao estudo do Rasa, assistindo alguns tutoriais disponibilizados pela plataforma e desenvolvendo um assistente virtual relativamente simples de maneira a ter um contacto mais direto com a mesma. De seguida passei ao estudo da plataforma Lexia que acabou por se prolongar, uma vez que esta estava de certo modo relacionada com a leitura e estudo da documentação e respetivas alterações do Rasa. Por fim, foi criada a especificação técnica e a escrita da proposta de estágio.

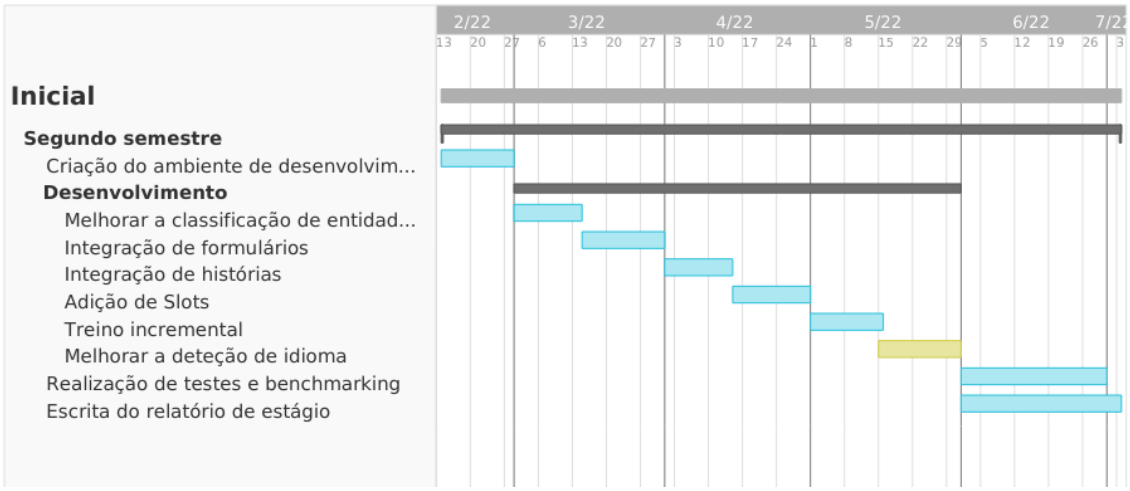


Figura 5.3: Diagrama de Gantt inicial relativo ao segundo semestre

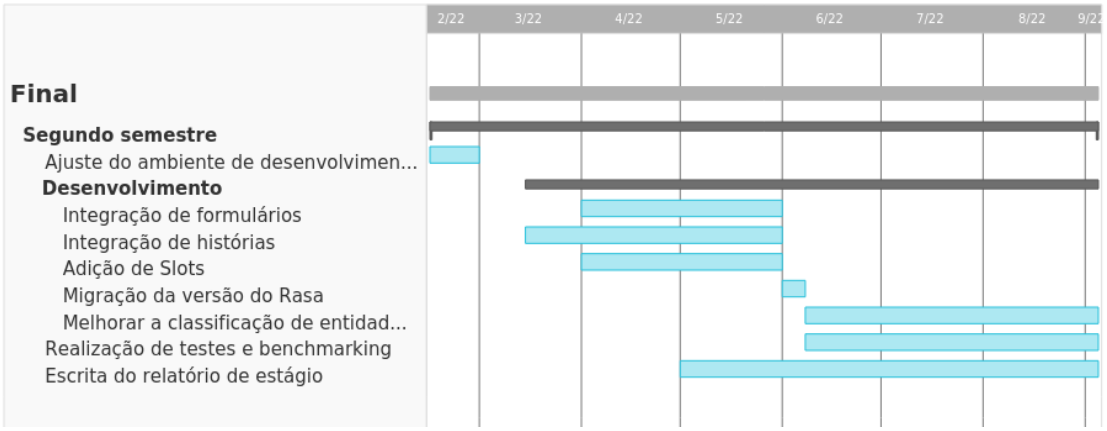


Figura 5.4: Diagrama de Gantt final relativo ao segundo semestre

Relativamente ao segundo semestre, podemos observar uma ligeira alteração no número de tarefas, devido ao facto da tarefa relativa ao treino incremental não requerer qualquer tipo de implementação, uma vez que esta funcionalidade está presente no Rasa através da linha de comandos e da tarefa opcional relativa à deteção de idioma não ter sido realizada devido ao tempo disponível. Por outro lado, foi adicionada a tarefa da migração da versão do Rasa, uma vez que não tinha sido considerada no diagrama inicial do segundo semestre.

No que diz respeito aos tempos de execução das tarefas podemos observar uma grande variação entre a expectativa inicial e o tempo que realmente foi despendido nas mesmas, uma vez que as tarefas relativas à integração de histórias, formulários e *slots* acabaram por ser realizadas em simultâneo.

Capítulo 6

Implementação

Neste capítulo será descrita a implementação que permite ao Lexia manter estado das conversas. Inicialmente, é abordado o executor criado como prova de conceito para que as funcionalidades pudessem ser testadas. De seguida, é explicado o motivo pelo qual as histórias tiveram de ser implementadas na plataforma Lexia e é descrita um pouco a implementação e o seu funcionamento. É também explanada a implementação dos formulários, dos slots e do executor auxiliar cuja função é pedir informação ao utilizador sempre que esta está em falta.

6.1 Arquitetura Lexia 2.0

A primeira alteração a ser feita foi o formato dos ficheiros que contêm os exemplos de treino, uma vez que o formato Markdown deixou de ser suportado. O formato suportado pelo Rasa passou a ser o YAML.

Foi também necessário realizar alterações no módulo *Conversation* do Lexia. Inicialmente deu-se a implementação da classe *Conversation* para manter um histórico de toda a conversa, uma vez que esta classe já estava prototipada na arquitetura mas ainda não tinha sido implementada. Assim sendo, a partir de agora é possível manter uma lista de todas as mensagens trocadas durante uma determinada conversa. Sempre que uma nova conversa for iniciada, as mensagens anteriores são descartadas.

Foi também criada a classe *StorySchema* para guardar as histórias lidas dos ficheiros de configuração.

De seguida foram implementadas as classes *Story* e *StoryIntent* para guardar as informações relativas às histórias e as intenções associadas a estas, respetivamente.

Por fim, foi implementada a classe *StoryExecutionState* para guardar o estado da história ativa no momento. Esta é responsável por manter informações acerca do ponto em que a história se situa, as intenções anteriormente introduzidas pelo utilizador, os *slots* que estão a ser preenchidos bem como aqueles que já foram preenchidos caso o utilizador tenha guardado os *slots* da intenção anterior e o

idioma utilizado na última mensagem enviado pelo utilizador.

Na Figura 6.1 podemos observar as alterações realizadas no modelo de domínio relativas ao módulo *Conversation*. Destacado a cinzento representa as classes que já existiam na primeira versão do Lexia e que não sofreram qualquer alteração e a amarelo a classe que já existia mas sofreu alterações. Todas as restantes foram implementadas no Lexia 2.0.

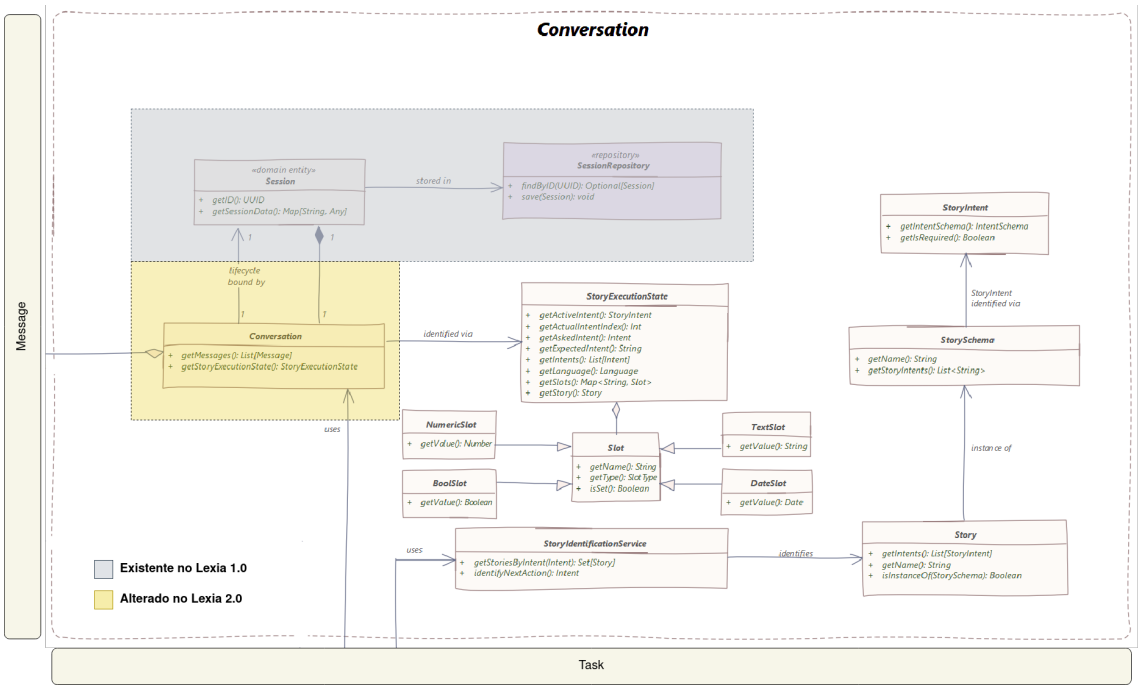


Figura 6.1: Alterações no modelo de domínio referentes ao módulo *Conversation*.

Outro módulo que sofreu alterações foi o módulo *Task*. Foi necessário fazer alguns ajustes à classe *TaskIdentificationService* para que esta fizesse a gestão das mensagens à medida que estas iam sendo recebidas, de maneira a guardar todas as mensagens de uma determinada conversa.

Por fim, foi necessário adicionar a classe *AskInfoTask*, responsável por guardar a informação das tarefas responsáveis por pedir informação ao utilizador e o respetivo serviço (*AskInfoTaskExecutionService*) para que tais tarefas sejam executadas.

Na Figura 6.2 podemos observar destacado a amarelo a classe já existente na primeira versão do Lexia e a verde as classes que foram implementadas no Lexia 2.0. Todas as restantes já existiam no Lexia 1.0 e não sofreram alterações.

O modelo de domínio final pode ser visto na Figura A.2.

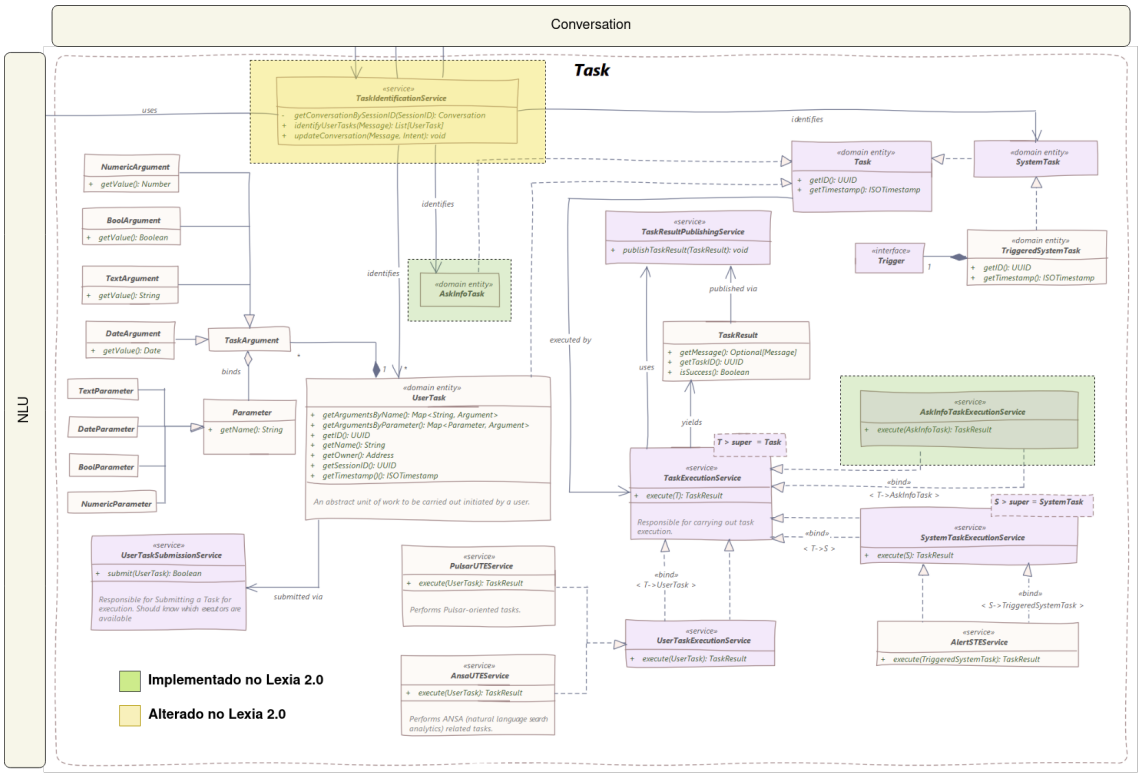


Figura 6.2: Alterações no modelo de domínio referentes ao módulo *Task*.

6.2 Executor *Proof of Concept* (POC)

O primeiro passo desta implementação foi a criação de um executor para que as funcionalidades implementadas pudessem ir sendo testadas à medida que iam sendo terminadas. Este executor é responsável pela marcação de viagens e necessita que o utilizador lhe forneça informação acerca da cidade de destino e do número de passageiros para que possa concluir a tarefa com sucesso. Para além desta intenção, este executor é também encarregue de simular a marcação de um hotel cujas entidades são a cidade de destino e o número de dias para a estadia. Estas intenções podem ser visualizadas na Tabela 6.1.

Para que as respetivas tarefas possam ser reencaminhadas para este executor, foi criada uma nova fila de espera responsável apenas pelas tarefas tanto da marcação de voos como da visualização do último voo marcado. Para que fosse possível guardar os voos e apresentar ao utilizador o último voo marcado sempre que este o requisitasse, foi configurada uma estrutura de dados em Redis [28] que armazena o ID de um determinado voo, a cidade de destino e o respetivo número de passageiros, através da classe *Flight*.

Após a criação do executor e de toda a lógica para o seu funcionamento, foi necessário adicionar dados de treino para que o Rasa detetasse as intenções de marcar voo e visualizar o último voo marcado. Para tal, foram treinados novos modelos, tanto para PT como para EN para que estas intenções fossem reconhecidas em qualquer um dos idiomas.

Nome	Descrição	Entidades
book_flight	Esta intenção é responsável por simular a marcação de um voo para uma determinada cidade de destino e o respetivo número de passageiros.	Cidade de destino Número de passageiros
book_hotel	Semelhante à intenção anterior, esta é responsável por simular a reserva de um hotel para uma determinada cidade de destino e o respetivo número de dias para a estadia.	Cidade de destino Número de dias
get_last_flight	Intenção responsável por devolver o último voo guardado na estrutura de dados Redis ao utilizador.	Não tem entidades

Tabela 6.1: Intenções associadas ao executor POC.

6.3 Histórias

Uma vez que o Rasa já possuía suporte para histórias na sua plataforma, o primeiro passo foi verificar se era possível tirar partido destas na plataforma do Lexia. Assim sendo, antes de iniciar a implementação das histórias, foi necessário avaliar as diferenças nos pedidos realizados pelos *endpoints* do Rasa entre as versões 1.7.0 e 3.0.0, de maneira a verificar se a integração das histórias disponibilizadas pelo Rasa era possível com o Lexia. Para tal, foi treinado um modelo no Rasa com um conjunto de histórias de maneira a observar se o pedido do *endpoint* devolvia em que história o fluxo de conversação melhor se encaixava. Foi utilizado o Postman [29] para fazer essa análise, onde foi observado que não havia alteração na resposta devolvida pelo Rasa, ou seja, não era passada informação acerca das histórias. Assim sendo foi tomada a decisão de implementar as histórias diretamente no Lexia.

As histórias são representadas pela classe *Story* que possui uma lista ordenada de todas as intenções que a constituem bem como as entidades a estas associadas. Estas histórias são identificadas através do *StoryIdentificationService* que devolve o conjunto de histórias que possuam as intenções introduzidas pelo utilizador.

Após a identificação da história, é mantido um estado da mesma através da classe *StoryExecutionState*, que guarda informação sobre a história ativa no momento, bem como o ponto da história em que a conversa se situa. Esta classe é também responsável por guardar a informação dos *slots* relativos à intenção que está a ser preenchida no momento, ou seja, à medida que o assistente vai pedindo a

Intenção	Dados de treino
book_flight	- Quero marcar um voo para Paris (*destination_city*) para 3 (*num_passengers*) pessoas. - Podias agendar um voo para Munique (*destination_city*) para 1 (*num_passengers*) pessoa?
book_hotel	- Quero marcar um hotel para Paris (*destination_city*) durante 4 (*num_days*) dias. - Podias-me marcar um hotel durante 3 (*num_days*) dias para Lisboa (*destination_city*)?
get_last_flight	- Qual o ultimo voo que eu marquei? - Mostra-me o meu ultimo voo. - Nao me lembro da ultima viagem marcada.

Tabela 6.2: Alguns exemplos de dados de treino das intenções relativas ao executor POC.

informação ao utilizador, esta vai sendo guardada nos respetivos *slots* até reunir toda a informação necessária para realizar a tarefa. Sempre que alguma intenção não pertencer a nenhuma história, o estado das histórias volta ao estado inicial e será feita uma nova identificação da história na próxima mensagem do utilizador.

6.4 Formulários

Após a implementação das histórias, deu-se início à implementação dos formulários para que a informação pudesse ser pedida ao utilizador sempre que estivesse em falta. Assim sendo, caso o utilizador forneça toda a informação necessária em uma só mensagem, a intenção está completa e a tarefa é reencaminhada para o respetivo executor para que possa ser executada. Caso contrário, passamos para o passo seguinte na história, no caso da intenção introduzida pelo utilizador necessitar de entidades para a sua execução, os passos seguintes serão essas entidades, para se certificar que estas são pedidas caso não tenham sido fornecidas. É feita a verificação do primeiro *slot*, se este ainda não estiver preenchido é pedida a informação ao utilizador, se já estiver preenchido é feita a verificação do próximo *slot* e assim sucessivamente.

6.5 Slots

Por fim, foi criada a classe *Slot* para que fosse possível guardar o nome do mesmo, o seu valor e o seu tipo. Tal como já foi referido anteriormente, os *slots* guardam a informação retirada das entidades que são extraídas através do Rasa. Estes são preenchidos no decorrer da história e podem ser reutilizados em intenções que necessitem do mesmo tipo de informação, ou seja, sempre que o utilizador introduz uma nova intenção no contexto da história, caso esta possua *slots* em comum com as intenções anteriores é-lhe perguntado se deseja guardar a informação já fornecida anteriormente. Se o utilizador recusar, esta informação é pedida novamente para que os *slots* sejam preenchidos de novo. Por outro lado, se o utilizador aceitar, os *slots* mantêm-se preenchidos para que possam ser reutilizados.

6.6 Executor Auxiliar

Após a preparação do Lexia para pedir informação ao utilizador, foi necessário criar um método para o fazer. Para tal, foi implementado um executor com o intuito de apenas pedir informação ao utilizador. Sempre que o núcleo do Lexia precisar de preencher algum *slot* em falta, este reencaminhará a tarefa responsável por pedir a informação relativa àquele *slot* para o executor, o qual enviará uma mensagem ao utilizador com o respetivo pedido. Uma vez que esta informação é fornecida através de intenções genéricas, ou seja, tanto pode ser necessários pedir um número para saber a idade do utilizador como para saber o número de passageiros de uma viagem, as perguntas são definidas no ficheiro de configuração tendo em conta a história em questão para que possam ser realizadas de maneira a fazerem sentido no contexto da conversa.

Capítulo 7

Avaliação

Neste capítulo serão apresentadas as diferenças no fluxo de conversação entre ambas as versões do Lexia, demonstrando com exemplos reais. Serão também apresentados os testes realizados e discutidos os resultados dos mesmos. Por fim, serão comparados os resultados obtidos entre o Lexia 1.0 e o Lexia 2.0, tanto para a classificação de intenções como para a classificação de entidades.

7.1 Alterações na conversação

As alterações descritas no capítulo anterior permitem ao Lexia manter estado da conversa, bem como gerir o fluxo da mesma. Na primeira versão do Lexia (Figura 7.1), podemos observar que não era possível manter estado de uma conversa, sendo necessário fornecer toda a informação para a realização da tarefa. Caso não se verificasse o assistente apresentava uma mensagem de erro, pedindo toda a informação necessária. Após as alterações realizadas, podemos observar que o assistente é capaz de manter estado da conversa, pedindo a informação em falta de uma maneira sequencial (Figura 7.2).

7.2 Introdução aos testes realizados

Para a obtenção dos resultados foi utilizada a funcionalidade do Rasa [30] para testar os dados de treino e a respetiva classificação dos modelos. Assim sendo, foi utilizado o método de *cross-validation* [31] que consiste na separação dos dados de treino em N subconjuntos, todos com o mesmo número de dados de treino. Esta técnica visa avaliar a capacidade de generalização do modelo de classificação treinando diversos modelos alternando os N subconjuntos. Para tal, foi utilizado este método com um valor de N=5 e os resultados obtidos estão apresentados nos gráficos abaixo. Após a conclusão dos testes, o Rasa devolve os valores de três métricas diferentes, são elas a exatidão, F1-score e precisão.

A exatidão representa o número de classes que foram categorizadas corretamente tendo em conta o número total de exemplos no *dataset*. No entanto, esta métrica

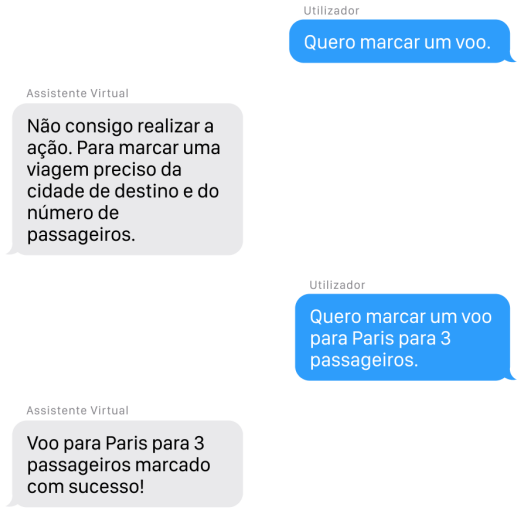


Figura 7.1: Exemplo de uma conversa no Lexia 1.0.

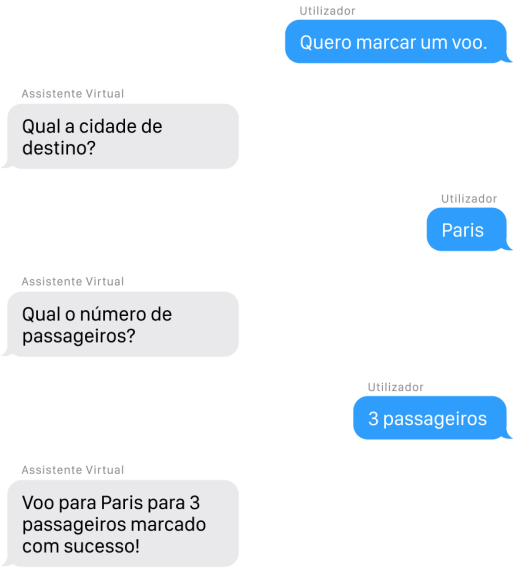


Figura 7.2: Exemplo de uma conversa no Lexia 2.0.

não tem em conta o balanço das classes, podendo facilmente induzir em erro. A fórmula para para calcular a exatidão é:

$$Exatidão = \frac{Verdadeiros\ Positivos + Verdadeiros\ Negativos}{Verdadeiros\ Positivos + Falsos\ Negativos + Verdadeiros\ Negativos + Falsos\ Positivos}$$

A precisão é a fração de verdadeiros positivos entre os exemplos que o modelo classificou como positivos. A fórmula para calcular a precisão é:

$$Precisão = \frac{Verdadeiros\ Positivos}{Verdadeiros\ Positivos + Falsos\ Positivos}$$

O *recall* é utilizado para medir a contagem de número de verdadeiros positivos tendo em conta o número de positivos existentes no *dataset*. Embora esta métrica não seja devolvida pelo Rasa ao utilizador, ela é necessária para o cálculo do F1-score. A fórmula para calcular o *recall* é:

$$Recall = \frac{Verdadeiros\ Positivos}{Verdadeiros\ Positivos + Falsos\ Negativos}$$

O F1-score combina a precisão e o *recall* e é bastante útil em cenários onde as classes não estão equilibradas. A fórmula para calcular o F1-score é:

$$F1 - score = \frac{2 * Precisão * recall}{Precisão + recall}$$

7.3 Comparação de *Pipelines*

Nesta secção serão comparadas as diferentes *pipelines* testadas para o Lexia 2.0, tanto para o idioma inglês como para o português. Para ambos os idiomas, inicialmente é apresentada a *pipeline* utilizada no Lexia 1.0 e de seguida são apresentadas as *pipelines* testadas na nova versão do Lexia, seguidas pela respetiva justificação para o qual essas *pipelines* terem sido usadas ou testadas.

7.3.1 Idioma Inglês

Lexia 1.0

O primeiro passo foi obter os resultados relativos à *pipeline* utilizada pelo Lexia 1.0, que se encontra apresentada na Figura 7.3, para que se obtivesse um meio de comparação para as *pipelines* que iriam ser testadas no Lexia 2.0.

```
language: en

pipeline:
  - name: "WhitespaceTokenizer"
  - name: "RegexFeaturizer"
  - name: "CRFEntityExtractor"
  - name: "DucklingHTTPExtractor"
    url: "http://duckling-server:8000"
    dimensions: ["time", "number", "duration"]
    timezone: "Europe/Lisbon"
  - name: "EntitySynonymMapper"
  - name: "CountVectorsFeaturizer"
  - name: "CountVectorsFeaturizer"
    analyzer: "char_wb"
    min_ngram: 1
    max_ngram: 4
  - name: "EmbeddingIntentClassifier"
```

Figura 7.3: *Pipeline* utilizada pelo modelo de classificação do Lexia 1.0 para o idioma inglês.

Lexia 2.0

O passo seguinte para escolher a *pipeline* que seria utilizada para o Lexia 2.0, foi utilizar uma *pipeline* equivalente àquela utilizada no Lexia 1.0 (Figura 7.3). No entanto, uma vez que o *EmbeddingIntentClassifier* foi removido nas versões mais recentes do Rasa, foi necessário remover este e substituí-lo pelo *DIETClassifier*. Para além disso, uma vez que o *DIETClassifier* é capaz de realizar a classificação

de intenções e a extração de entidades, foram removidos os extratores de entidades da *pipeline* utilizada no Lexia 1.0, ficando o *DIETClassifier* responsável por estas duas tarefas, dando origem à *pipeline* apresentada na Figura 7.4.

```
language: en

pipeline:

  - name: "SpacyNLP"
    model: "en_core_web_md"
  - name: "SpacyTokenizer"
  - name: "SpacyFeaturizer"
  - name: "RegexFeaturizer"
  - name: "CountVectorsFeaturizer"
  - name: "DIETClassifier"
    epochs: 100
  - name: "EntitySynonymMapper"
```

Figura 7.4: Primeira *pipeline* testada no Lexia 2.0 para o idioma inglês.

De seguida, repetiu-se a experiência com a mesma *pipeline* aumentando o número de *epochs* de 100 para 300, com o objetivo de melhorar a classificação de intenções e a extração de entidades, obtendo a *pipeline* representada na Figura 7.5.

```
language: en

pipeline:

  - name: "SpacyNLP"
    model: "en_core_web_md"
  - name: "SpacyTokenizer"
  - name: "SpacyFeaturizer"
  - name: "RegexFeaturizer"
  - name: "CountVectorsFeaturizer"
  - name: "DIETClassifier"
    epochs: 300
  - name: "EntitySynonymMapper"
```

Figura 7.5: Segunda *pipeline* testada no Lexia 2.0 para o idioma inglês.

Por fim, foi testada mais uma *pipeline*, onde se manteve o *DIETClassifier* com um número de *epochs*=100 para a classificação de intenções e se alterou o extrator de entidades para o *CRFEntityExtractor*, tal como era utilizado no Lexia 1.0. As alterações descritas deram origem à *pipeline* representada na Figura 7.6.

```
language: en

pipeline:

  - name: "SpacyNLP"
    model: "en_core_web_md"
  - name: "SpacyTokenizer"
  - name: "SpacyFeaturizer"
  - name: "RegexFeaturizer"
  - name: "CountVectorsFeaturizer"
  - name: "DIETClassifier"
    epochs: 300
    entity_recognition: false
  - name: "CRFEntityExtractor"
  - name: "EntitySynonymMapper"
```

Figura 7.6: Terceira *pipeline* testada no Lexia 2.0 para o idioma inglês.

7.3.2 Idioma Português

Lexia 1.0

Tal como para o idioma anterior, o primeiro passo foi realizar os testes para a *pipeline* que era utilizada no Lexia 1.0 na classificação de mensagens em português, para que fosse possível comparar com os valores mais tarde obtidos nos testes realizados às *pipelines* relativas ao Lexia 2.0.

```
language: pt

pipeline:
  - name: "SpacyNLP"
    model: "pt_core_news_md"
  - name: "SpacyTokenizer"
  - name: "SpacyFeaturizer"
  - name: "RegexFeaturizer"
  - name: "EmbeddingIntentClassifier"
  - name: "EntitySynonymMapper"
  - name: "CRFEntityExtractor"
  - name: "DucklingHTTPExtractor"
    url: "http://duckling-server:8000"
    dimensions: [ "time", "duration" ]
    locale: "pt_PT"
```

Figura 7.7: *Pipeline* utilizada pelo modelo de classificação do Lexia 1.0 para o idioma português.

Lexia 2.0

Após a obtenção dos resultados com a *pipeline* utilizada pelo Lexia 1.0 para a classificação de mensagens introduzidas pelo utilizador em português, foi construída uma *pipeline* com o intuito de ser semelhante àquela que era utilizada na primeira versão do Lexia. Mais uma vez, como o *EmbeddingIntentClassifier* foi removido pelo Rasa, este foi substituído pelo *DIETClassifier* que ficou também responsável pela extração de entidades, removendo todos os restantes extratores de entidades da *pipeline*. Esta encontra-se representada na Figura 7.9.

```
language: pt

pipeline:
  - name: "SpacyNLP"
    model: "pt_core_news_md"
  - name: "SpacyTokenizer"
  - name: "SpacyFeaturizer"
  - name: "RegexFeaturizer"
  - name: "DIETClassifier"
    epochs: 100
  - name: "EntitySynonymMapper"
```

Figura 7.8: Primeira *pipeline* testada no Lexia 2.0 para o idioma português.

```
language: pt

pipeline:
  - name: "SpacyNLP"
    model: "pt_core_news_md"
  - name: "SpacyTokenizer"
  - name: "SpacyFeaturizer"
  - name: "RegexFeaturizer"
  - name: "DIETClassifier"
    epochs: 300
  - name: "EntitySynonymMapper"
```

Figura 7.9: Segunda *pipeline* testada no Lexia 2.0 para o idioma português.

Por fim, foi testada a *pipeline* representada na Figura 7.10. Nesta configuração manteve-se o *DIETClassifier* para a classificação de intenções e alterou-se o extrator de entidades para o *CRFEntityEXtractor*, uma vez que era aquele utilizado no Lexia 1.0.

```
language: pt

pipeline:
  - name: "SpacyNLP"
    model: "pt_core_news_md"
  - name: "SpacyTokenizer"
  - name: "SpacyFeaturizer"
  - name: "RegexFeaturizer"
  - name: "DIETClassifier"
    entity_recognition: false
  - name: "CRFEntityExtractor"
    epochs: 300
  - name: "EntitySynonymMapper"
```

Figura 7.10: Terceira *pipeline* testada no Lexia 2.0 para o idioma português.

Visto que os resultados ficaram abaixo do esperado, a *pipeline* escolhida para a classificação de mensagens em português foi a que se encontra na Figura 7.9, uma vez que foi a que apresentou melhores resultados.

7.4 Resultados Finais

Nesta secção serão comparados e analisados os valores entre o Lexia 1.0 e o Lexia 2.0 das três métricas utilizadas para ambos os idiomas. Para a obtenção dos gráficos foram utilizadas as *pipelines* que obtiveram melhores resultados em cada cenário.

	Classificação de intenções			**Classificação de entidades**		
	Precisão	Recall	F1-score	Precisão	Recall	F1-score
Figura 7.3	0.957	0.956	0.955	0.850	0.708	0.759
Figura 7.4	**0.965**	**0.965**	**0.964**	0.848	0.786	0.808
Figura 7.5	0.955	0.955	0.953	**0.877**	**0.871**	**0.871**
Figura 7.6	0.947	0.946	0.944	0.725	0.542	0.597

Tabela 7.1: Resultados das *pipelines* utilizadas para o idioma inglês. A Figura 7.3 representa os resultados obtidos com a *pipeline* do Lexia 1.0. As Figuras 7.4, 7.5 e 7.6 representam os resultados das *pipelines* testadas no Lexia 2.0.

Dado que os resultados obtidos para a classificação de entidades (Tabela 7.1) com a *pipeline* representada na Figura 7.4 ficaram um pouco aquém do esperado, decidiu-se repetir a experiência com a mesma *pipeline* aumentando o número de

epochs de 100 para 300, obtendo os resultados apresentados na Tabela 7.1 na linha referente à Figura 7.5. Embora, se tenha observado uma ligeira descida na classificação de intenções, os resultados da extração de entidades obtiveram uma melhoria significativa com o aumento do número de *epochs*.

Ainda assim, foi testada a *pipeline* representada na Figura 7.6 para avaliar o desempenho do *CRFEntityEXtractor* na nova versão do Rasa. Após a obtenção dos resultados dos testes realizados para a última *pipeline* e uma comparação com as restantes *pipelines* testadas, podemos observar que esta foi aquela que obteve os piores valores relativamente à extração de entidades e classificação de intenções. Assim sendo, a *pipeline* da Figura 7.5 foi a escolhida para a classificação de mensagens com o idioma inglês.

	Classificação de intenções			**Classificação de entidades**		
	Precisão	Recall	F1-score	Precisão	Recall	F1-score
Figura 7.7	0.974	0.974	0.974	0.779	0.453	0.529
Figura 7.8	**0.984**	**0.984**	0.983	0.785	0.627	0.683
Figura 7.9	**0.984**	**0.984**	**0.984**	**0.924**	**0.756**	**0.824**
Figura 7.10	**0.984**	**0.984**	**0.984**	0.757	0.500	0.572

Tabela 7.2: Resultados das *pipelines* utilizadas para o idioma português. A Figura 7.7 representa os resultados obtidos com a *pipeline* do Lexia 1.0. As Figuras 7.8, 7.9 e 7.10 representam os resultados das *pipelines* testadas no Lexia 2.0.

Relativamente aos resultados obtidos para o idioma português (Tabela 7.2), podemos observar um comportamento semelhante àquele observado anteriormente, onde a *pipeline* com 300 *epochs* foi aquela que apresentou os melhores resultados. A *pipeline* que utiliza o *CRFEntityExtractor* como extrator de entidades foi a que apresentou os piores resultados. Assim sendo, a *pipeline* representada na Figura 7.9 foi a que ficou definida para a classificação de mensagens em português.

7.5 Discussão de Resultados

7.5.1 Classificação de Intenções

Em relação à classificação de intenções para o idioma inglês, embora os valores sejam bastante semelhantes, observamos uma pequena melhoria no Lexia 2.0.

Tal como para o idioma inglês, a diferença obtida nos resultados na classificação de intenções para a língua portuguesa entre ambas as versões é mínima. Ainda assim conseguimos notar uma ligeira melhoria no Lexia 2.0.

Esta melhoria em ambos os idiomas comprova a capacidade do *DIETClassifier* na classificação de intenções, mesmo utilizando um número inferior de *epochs*, uma

vez que apresentou um baixo número de falsos positivos e falsos negativos.

Observando as matrizes de confusão para ambos os idiomas (Figuras B.1 e B.3), podemos confirmar a fiabilidade dos modelos, não havendo grandes erros na classificação das intenções.

7.5.2 Classificação de Entidades

Na classificação de entidades o *DIETClassifier* também se mostrou bastante competente, melhorando os resultados obtidos tanto para o idioma inglês como para o português. Foi na classificação de entidades onde se observou a maior diferença nos resultados obtidos entre ambas as versões do Lexia.

Observando a precisão e o recall, podemos reparar que o número de falsos positivos é inferior ao de falsos negativos, uma vez que a precisão é superior ao recall em todas as *pipelines* testadas.

Quando comparados os dois idiomas entre si, podemos observar que o F1-score é superior para o idioma inglês, uma vez que este apresenta um menor número de falsos positivos e de falsos negativos.

Podemos observar também pela matriz de confusão relativa à classificação de entidades para o idioma inglês (Figura B.2) que o número de dias é a entidade que apresenta um maior número de classificações incorretas. Relativamente à matriz de confusão referente à classificação de entidades para o idioma português observamos que as cidades não são reconhecidas num número considerável de vezes (Figura B.4).

Capítulo 8

Conclusão

Durante o primeiro semestre foi realizada a parte mais explorátoria deste estágio, que está relacionada com o estudo das plataformas Rasa e Lexia, uma vez que estas seriam os principais focos para a fase de desenvolvimento e implementação de novas funcionalidades.

Foram analisadas algumas alternativas ao Rasa e feita uma comparação entre estas, destacando os pontos positivos de cada uma. Após a comparação, foi justificada a escolha do Rasa como a *framework* para suportar o processamento de liguagem natural do Lexia, devido à sua facilidade de instalação em qualquer servidor, à sua customização, à importância que a plataforma dá ao avanço do estado da arte e à facilidade de melhorar as suas funcionalidades uma vez que a primeira versão do Lexia já utiliza o Rasa. Em relação ao Rasa, foi estudado todo o núcleo NLU que este diponibiliza e o seu funcionamento baseado na classificação de intenções e entidades, bem como todos os seus componentes e a maneira como estes são configurados na *pipeline*. Para além das funcionalidades já existentes na versão que suporta o Lexia, foram também estudadas as novas funcionalidades que permitiriam integrar estado ao Lexia, novos modelos de linguagem e modelos de classificação de maneira a melhorar toda a configuração do mesmo.

De seguida foi realizado o estudo da plataforma Lexia. Este estudo incluiu a sua integração com o Rasa e a maneira como a mensagem é processada desde que é recebida pelo Lexia até à sua classificação e devolução da resposta ao utilizador. Para tal, foi também necessário estudar cada módulo que constitui a arquitetura do Lexia.

Após o estudo das duas plataformas foram definidos os requisitos funcionais e não funcionais e descritas as principais alterações que iriam ocorrer ao nível da arquitetura do Lexia e na *pipeline* do Rasa.

No segundo semestre foi implementada toda a lógica necessária para que o Lexia conseguisse manter estado das conversas, uma vez que os endpoints do Rasa não passavam informação acerca das histórias. Assim sendo, o Lexia passou a possuir o conceito de histórias, formulários e slots. Juntamente com a lógica, foram também implementados dois executores, o primeiro foi utilizado como prova de conceito para testar as novas funcionalidades e o segundo foi implementado com

o intuito de pedir informação ao utilizador sempre que esta estivesse em falta.

Após a implementação terminada, deu-se início à comparação de diversas *pipelines* com o objetivo de alcançar aquelas que obtivessem melhor resultados e consecutivamente criassem os melhores modelos de classificação para as mensagens do utilizador.

Assim sendo, o objetivo foi cumprido pois foram concretizados os pontos principais do trabalho. O Lexia passou a integrar histórias, para uma melhor gestão do fluxo da conversação. Passou também a guardar determinadas informações recebidas pelo utilizador para que pudessem ser reutilizadas em diferentes pontos da conversa. Por fim, foi também migrado o núcleo de processamento de linguagem para uma versão mais recente, podendo fazer uso de modelos mais robustos e funcionalidades mais recentes.

O Rasa mostrou-se bastante competente na sua mais recente versão, uma vez que os valores obtidos nos testes realizados para a classificação de intenções e extração de entidades para os idiomas de inglês e português foram todos superiores no Lexia 2.0.

A exploração dos parâmetros disponibilizados pelo Rasa nos seus modelos de classificação é algo que deve ser feito num trabalho futuro, com o objetivo de melhorar e aperfeiçoar o funcionamento dos modelos e consecutivamente a classificação das mensagens, uma vez que esta *framework* oferece uma vasta quantidade de parâmetros para a configuração dos mesmos.

Referências

[1] Rasa, open source conversational ai. Disponível em: https://rasa.com/. Último Acesso: 2022-01-24.

[2] Writing conversation data. Disponível em: https://rasa.com/docs/rasa/writing-stories/. Último Acesso: 2022-06-24.

[3] Custom forms. Disponível em: https://learning.rasa.com/conversational-ai-with-rasa/custom-forms/. Último Acesso: 2022-06-24.

[4] A comparison of eight chatbot environments. Disponível em: https://dev.botframework.com/. Último Acesso: 2022-01-20.

[5] A comparison of eight chatbot environments. Disponível em: https://wit.ai/. Último Acesso: 2022-01-20.

[6] A comparison of eight chatbot environments. Disponível em: https://cloud.google.com/dialogflow/. Último Acesso: 2022-01-20.

[7] A comparison of eight chatbot environments. Disponível em: https://www.ibm.com/watson. Último Acesso: 2022-01-20.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]*, May 2019. arXiv: 1810.04805.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv:1706.03762 [cs]*, December 2017. arXiv: 1706.03762.

[10] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[11] Rasa WhitespaceTokenizer. Disponível em: https://rasa.com/docs/rasa/components/#whitespacetokenizer. Último Acesso: 2022-05-16.

[12] Rasa JiebaTokenizer. Disponível em: https://rasa.com/docs/rasa/components/#jiebatokenizer. Último Acesso: 2022-05-16.

[13] Rasa MitieTokenizer. Disponível em: https://rasa.com/docs/rasa/components/#mitietokenizer. Último Acesso: 2022-05-16.

[14] Rasa SpacyTokenizer. Disponível em: `https://rasa.com/docs/rasa/components/#spacytokenizer`. Último Acesso: 2022-05-16.

[15] Rasa MitieFeaturizer. Disponível em: `https://rasa.com/docs/rasa/components/#mitiefeaturizer`. Último Acesso: 2022-05-16.

[16] Rasa SpacyFeaturizer. Disponível em: `https://rasa.com/docs/rasa/components/#spacyfeaturizer`. Último Acesso: 2022-05-16.

[17] Rasa RegexFeaturizer. Disponível em: `https://rasa.com/docs/rasa/components/#regexfeaturizer`. Último Acesso: 2022-05-16.

[18] Rasa LanguageModelFeaturizer. Disponível em: `https://rasa.com/docs/rasa/components/#languagemodelfeaturizer`. Último Acesso: 2022-05-16.

[19] Rasa MitieIntentClassifier. Disponível em: `https://rasa.com/docs/rasa/components/#mitieintentclassifier`. Último Acesso: 2022-05-16.

[20] Rasa SklearnIntentClassifier. Disponível em: `https://rasa.com/docs/rasa/components/#sklearnintentclassifier`. Último Acesso: 2022-05-16.

[21] Rasa MitieEntityExtractor. Disponível em: `https://rasa.com/docs/rasa/components/#mitieentityextractor`. Último Acesso: 2022-05-16.

[22] Rasa SpacyEntityExtractor. Disponível em: `https://rasa.com/docs/rasa/components/#spacyentityextractor`. Último Acesso: 2022-05-16.

[23] Rasa DucklingEntityExtractor. Disponível em: `https://rasa.com/docs/rasa/components/#ducklingentityextractor`. Último Acesso: 2022-05-16.

[24] Rasa DIETClassifier. Disponível em: `https://rasa.com/docs/rasa/components#dietclassifier`. Último Acesso: 2022-02-10.

[25] Ryan Ong. Day 101 of #NLP365: In-Depth Study Of RASA's DIET Architecture. https://towardsdatascience.com/day-101-of-nlp365-in-depth-study-of-rasas-diet-architecture-3cdc10601599, April 2020.

[26] Mohit Saini. Using the DIET classifier for intent classification in dialogue. Disponível em: `https://medium.com/the-research-nest/using-the-diet-classifier-for-intent-classification-in-dialogue-489c76e62804`, July 2020.

[27] Tanja Bunk, Daksh Varshneya, Vladimir Vlasov, and Alan Nichol. DIET: Lightweight Language Understanding for Dialogue Systems. *arXiv:2004.09936 [cs]*, May 2020. arXiv: 2004.09936.

[28] Redis. Disponível em: `https://redis.io`. Último Acesso: 2022-06-25.

[29] Postman api platform. Disponível em: `https://www.postman.com`. Último Acesso: 2022-06-26.

[30] Evaluating an nlu model. Disponível em: `https://rasa.com/docs/rasa/testing-your-assistant/#evaluating-an-nlu-model`. Último Acesso: 2022-06-29.

[31] Evaluating an nlu model. Disponível em: `https://rasa.com/docs/rasa/testing-your-assistant/#using-cross-validation`. Último Acesso: 2022-06-29.

Apêndices

Apêndice A

Arquitetura Lexia

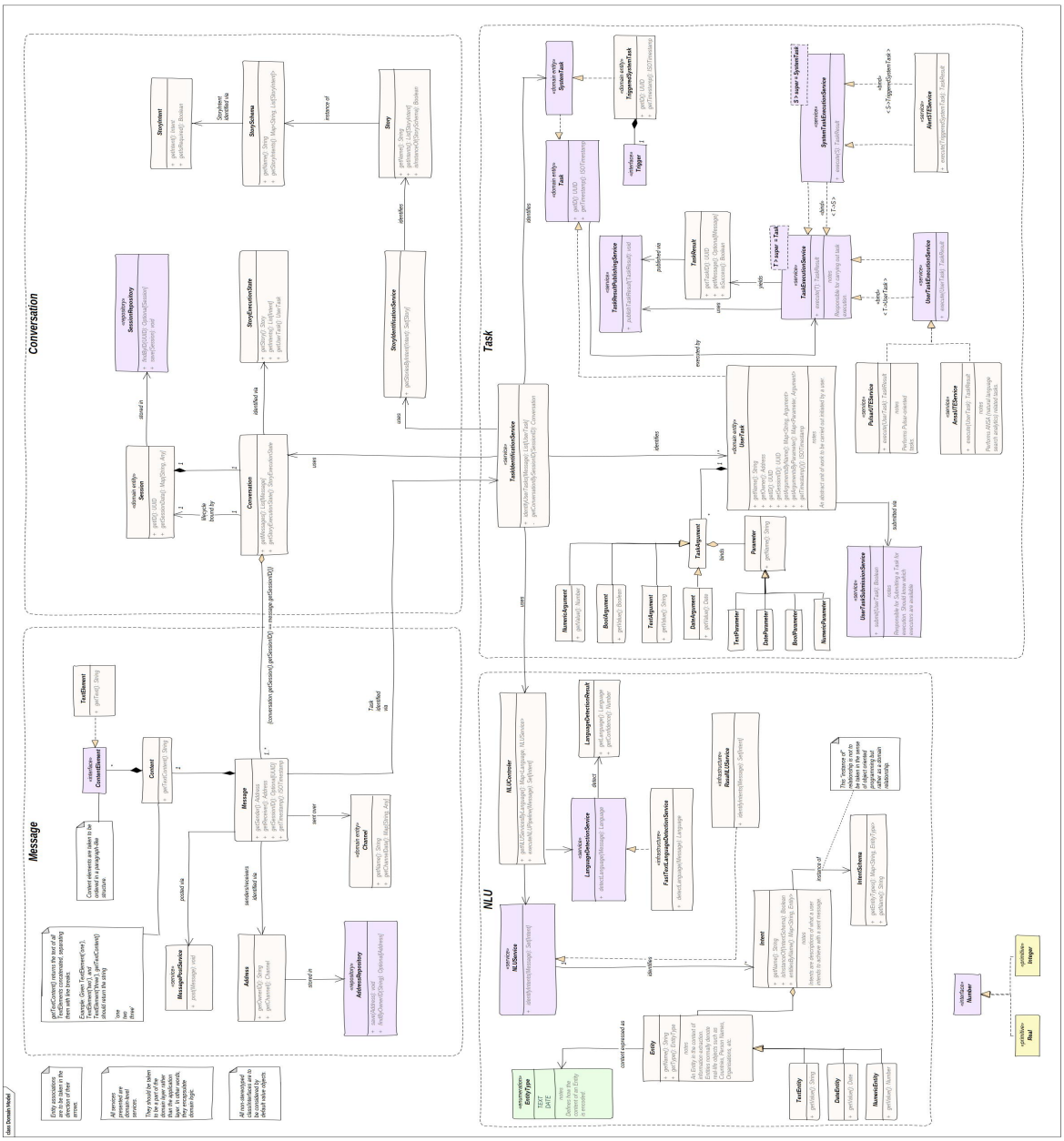


Figura A.1: Modelo de domínio do Lexia 1.0.

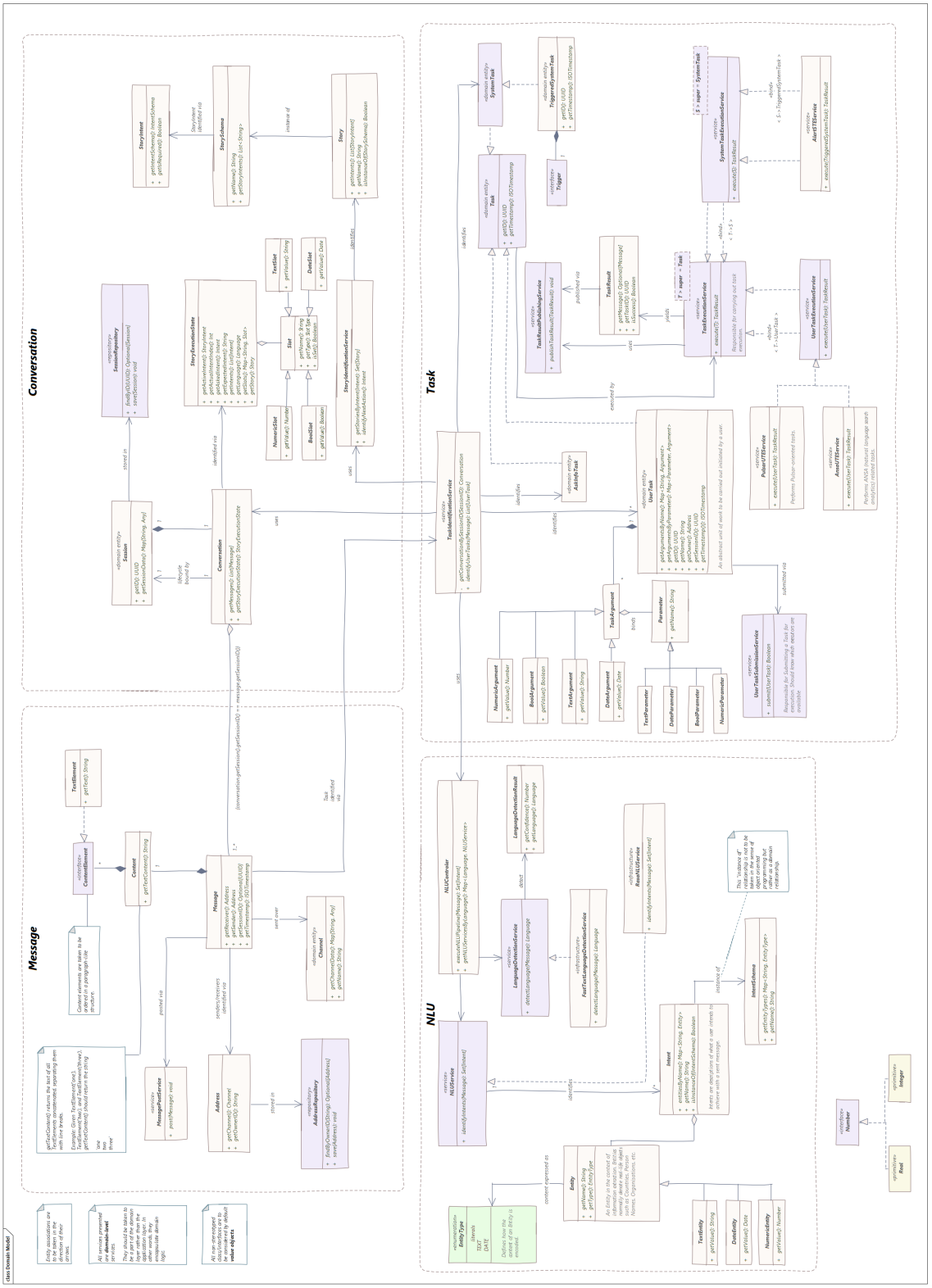


Figura A.2: Modelo de domínio do Lexia 2.0.

Apêndice B

Matrizes de confusão

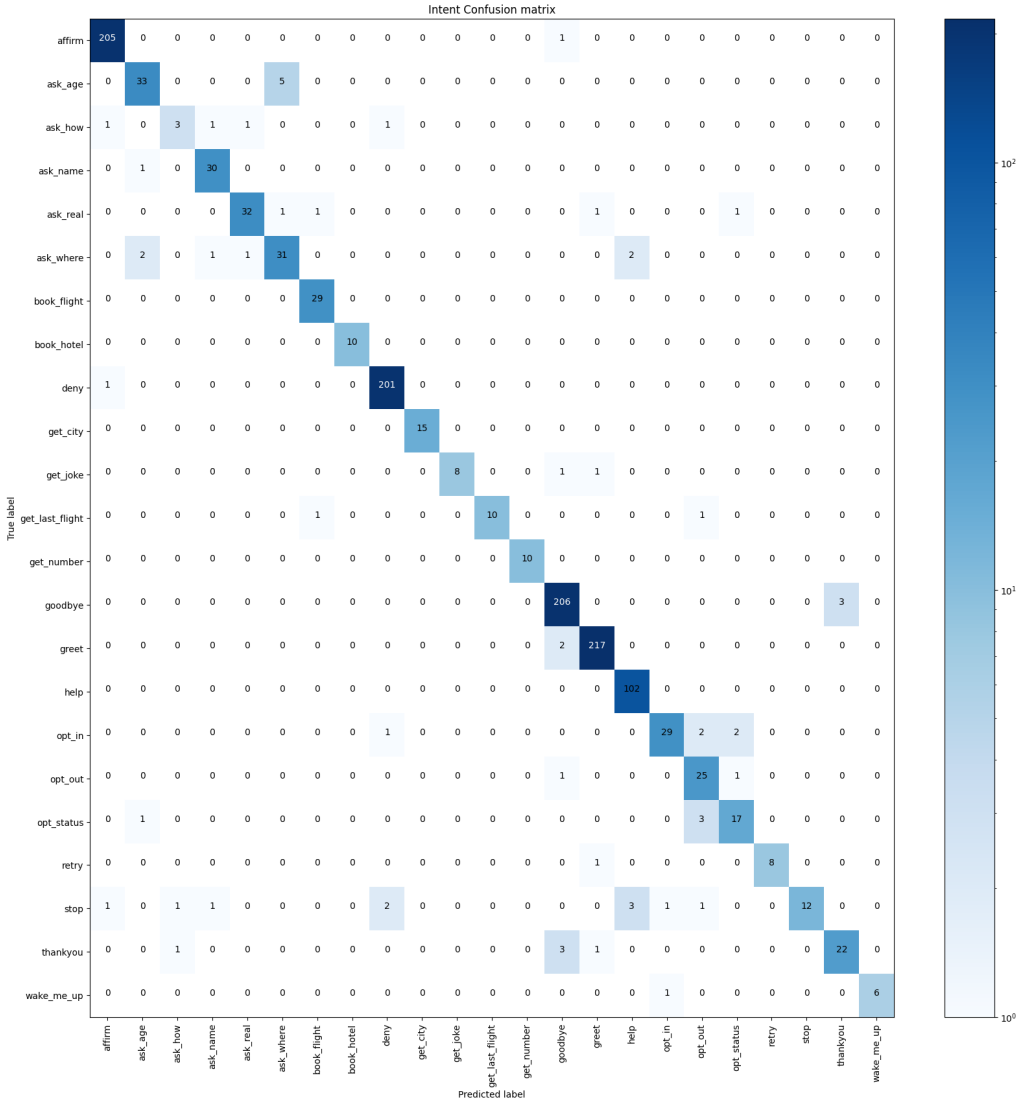


Figura B.1: Matriz de confusão relativa à classificação de intenções em inglês no Lexia 2.0.

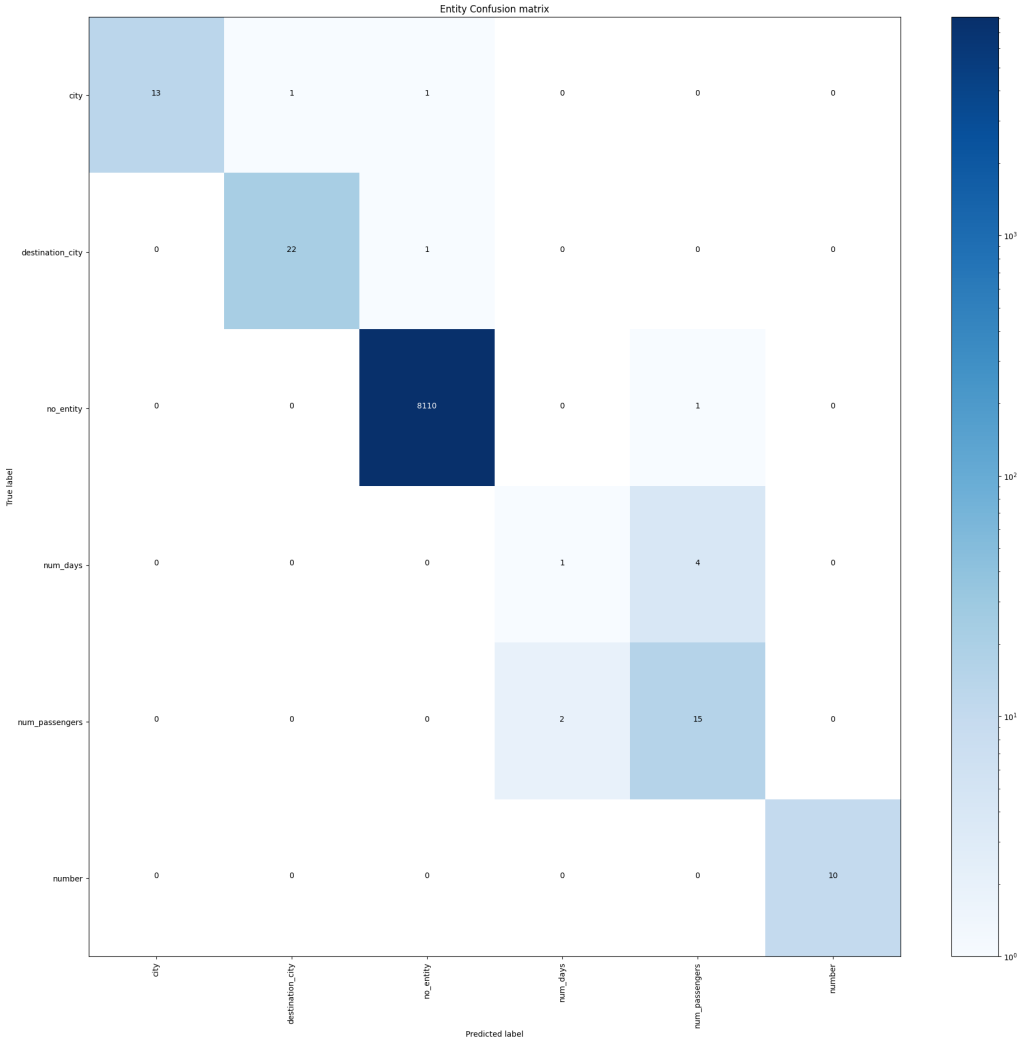


Figura B.2: Matriz de confusão relativa à classificação de entidades em inglês no Lexia 2.0.

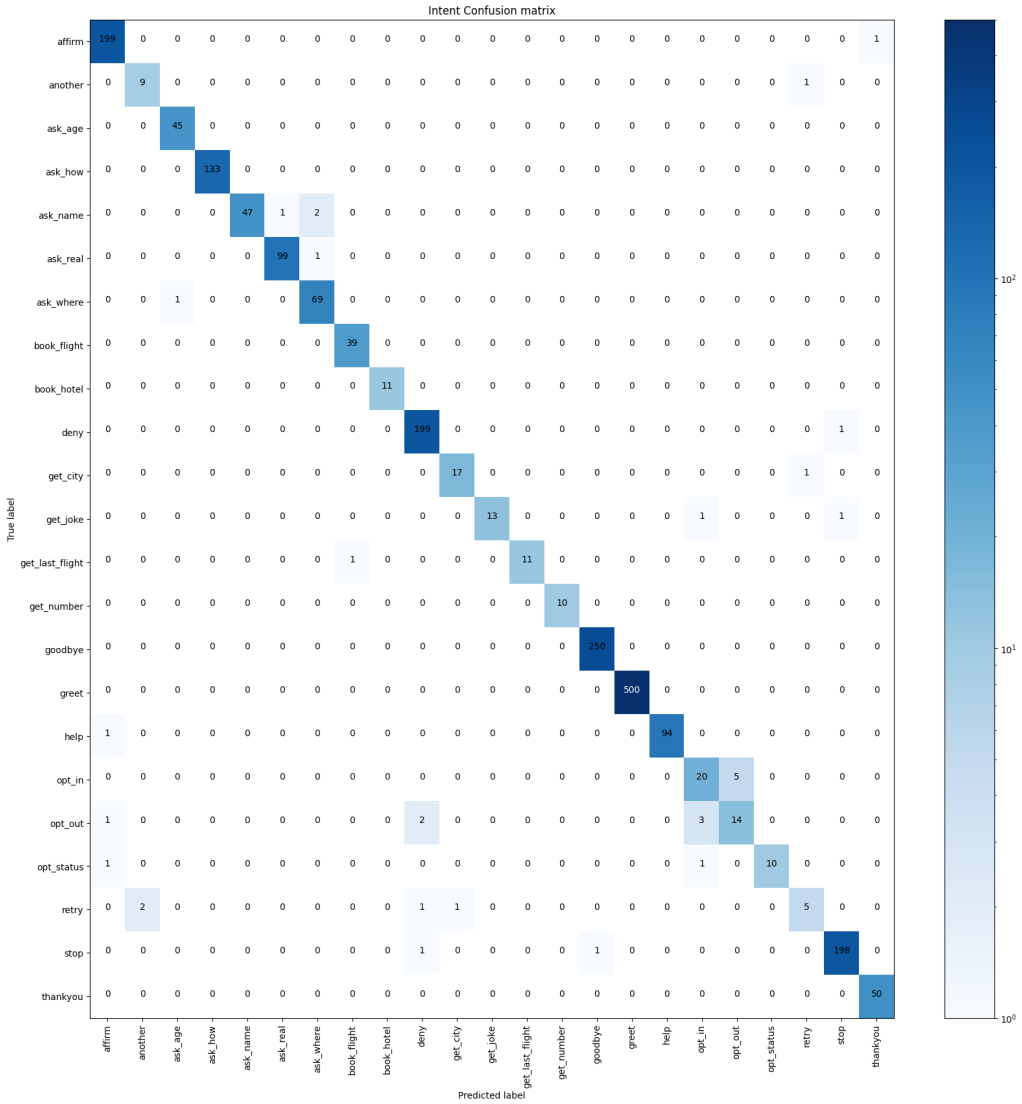


Figura B.3: Matriz de confusão relativa à classificação de intenções em português no Lexia 2.0.

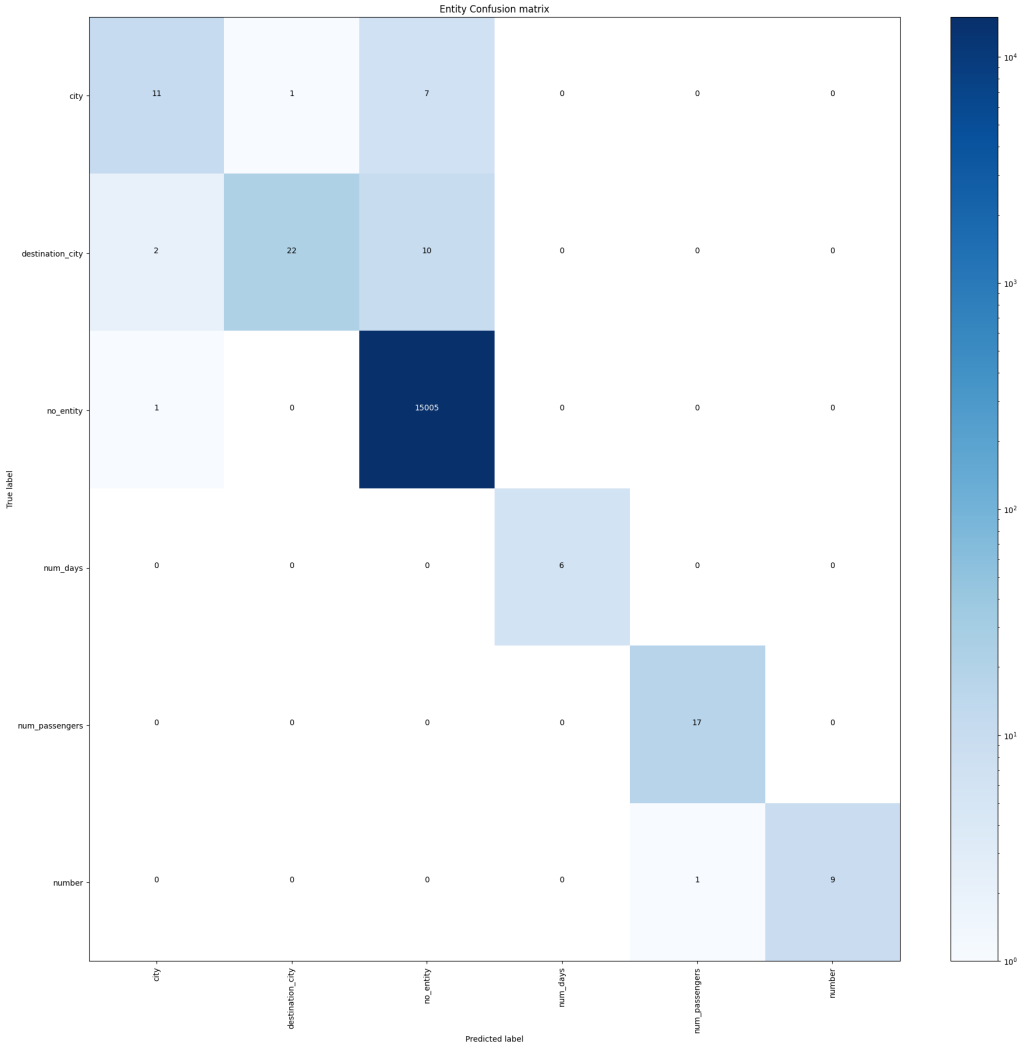


Figura B.4: Matriz de confusão relativa à classificação de entidades em português no Lexia 2.0.