TÉCNICO LISBOA

Building chatbots for customer support: fast and serious

Diogo André Barradas Fernandes

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur

Examination Committee

Chairperson: Prof. Manuel Fernando Cabido Peres Lopes

Supervisor: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur

Member of the Committee: Prof. Ricardo Daniel Santos Faro Marques Ribeiro

October 2021

Acknowledgments

First of all, I would like to thank all the people that have contributed to this thesis!

Next, I would like thank Diogo and Luísa for all the help they have given me. After all the doubts we faced during the first months regarding what this work would be, it was thanks to them that I was able to complete it. Thank you both for the connection we created during this year, which was beyond a work relationship. As for Diogo, thank you for the hours you spent with me, aiding me find solutions for any problems that I've faced during this work.

Next I want to specially thank my mom. After years of battling, she was always there for me, giving me strength and the guidance I needed to achieve my goals, and be a better human being. Thank you for giving me all the love and teachings!

I also want to thank my family and friends, that are always there both in good times and in times of need, that have helped me grow as a person.

For last but not least, I want to thank Ana from the bottom of my hearth, who has been a pillar for me during this year. Thank you for being with me in this journey, and for all the love, support and guidance you have given me.

Resumo

Agentes conversacionais são usados numa grande variedade de áreas que auxiliam o utilizador a completar uma tarefa. No entanto, para criar um agente conversacional, é necessária uma grande quantidade de dados, o que dificulta o processo de desenvolvimento para a criação de agentes para novas tarefas. Para abordar este problema, investigamos como acelerar o processo de desenvolvimento de agentes conversacionais na ferramenta Rasa. Primeiro, oferecemos duas ferramentas que através do uso da informação disponível nas bases de dados do MultiWOZ e Taskmaster-1, é capaz de criar um agente conversacional. Posteriormente, conduzimos avaliações automática e humana, que demonstram que é possível acelerar o processo de desenvolvimento de agentes conversacionais. Os resultados das nossas avaliações realçam a importância da informação anotada, tal como *slots*, os valores correspondentes, as *intents* das mensagens do utilizador e também a informação que é perguntada por parte do utilizador. Por último, fornecemos uma ferramenta que usando um agente conversacional do Rasa criado para o domínio de restaurantes, é capaz de criar um agente conversacional para o domínio dos hotéis. Os resultados obtidos na avaliação sugerem que é possível acelerar o processo de desenvolvimento de agentes conversacionais para novas tarefas usando informação de agentes conversacionais pré-existentes.

Palavras-chave: Agentes Conversacionais, Rasa Open Source, MultiWOZ, Taskmaster-1, Transferência de conhecimento, Desenvolvimento

Abstract

Conversational Agents are systems that are used in a wide variety of areas to assist the accomplishment of a task by the user. However, creating a conversational agent usually requires a great amount of data, which difficulties the development process to create agents for new tasks. To tackle this issue, we investigate how to expedite the development process. Firstly, we provide two separate frameworks that are able to create conversational agents in Rasa using the information available in the MultiWOZ and Taskmaster-1 datasets. We conduct an automatic and a user evaluation, which show that it is possible to expedite the creation of conversational agents using datasets, highlighting the importance of annotating information such as the slots, the correspondent values, the user utterance intents and also the information required by the user. Lastly, we provide a framework that using a Rasa conversational agent created for the restaurant domain, is able to create a conversational agent for the hotel domain. Our results suggest that it is possible to expedite the development of conversational agents by creating agents for new tasks using information from a preexisting conversational agent.

Keywords: Conversational Agents, Rasa Open Source, MultiWOZ, Taskmaster-1, Knowledge transfer, Development

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Chapter 1

Introduction

Conversational agents, also known as dialogue[1] systems, have been a hot topic in Natural Language Processing in recent times. These type of systems have been adopted in our social and business life (Liu and Lane 2018), where we can interact with them for fulfilling a task, or even just for fun. Conversational agents can be divided in two types: chit-chat (conversation without a reason) and task-oriented (the user has a goal in mind). This work focuses on the latter.

Dialogue systems are usually composed by three different modules: a Natural Language Understanding (NLU) module, a Dialogue Manager and the Natural Language Generation component. To create the best dialogue systems, many have tried to improve the quality of each of these modules. For example, Goo et al. (2018) attempted to improve the intent classification and slot filling tasks for the NLU module. For the dialogue manager, rule-based (Habib, Zhang, and Balog 2020) and machine learning based (Li et al. 2017) solutions were proposed. Moreover, many frameworks have been proposed in order to develop conversational agents. Some of these frameworks are Microsoft Bot Framework[2], Amazon Lex[3], Dialogflow[4] and Rasa[5].

However, the authoring of conversational agents faces two great obstacles: first, there is the need of a great amount of training data in order to train the agents on a task. Often, the lack of training data limits the number of tasks the agent is able to fulfill. Second, the behaviour of the agent during the conversation also needs to be defined. This means that a dialogue manager must be able to handle every conversation path the user can pursue to perform a task while engaging with the agent. Moreover, not only is there a considerate amount of paths to be considered, but there is also the need to consider deviations from the conversation path by the user.

As a result of the reasons presented above, developing a conversational agent is often difficult. This constitutes a problem for small business owners that want to create a bot, since the development process is expensive. For these reasons, it is important to expedite in any way the process of developing a conversational agent, which is the focus of this work. To do so, we concentrate on automatizing the

[1] The work reported in this dissertation relates with the research target of project MAIA (CMU-PT MAIA Ref. 045909)
[2] https://dev.botframework.com/
[3] https://aws.amazon.com/pt/lex/
[4] https://cloud.google.com/dialogflow/docs
[5] https://rasa.com/docs/rasa/, version 2.2.0

development of the NLU module and the Dialogue Manager module. To test this process, we will use the Rasa framework to create conversational agents, which is an open source framework.

To tackle the lack of data, many datasets have been proposed. This is the example of the MultiWOZ (Budzianowski et al. 2018) and Taskmaster-1 (Byrne et al. 2019) datasets, that contain task-oriented dialogues for several domains, developed using Wizard-of-Oz approach (Budzianowski et al. 2018). In both datasets, the dialogues are annotated. However, the annotations in each of the datasets is different, which leads to a different process of using the available information to create a conversational agent. With these datasets, we want to verify what is the process required to create a conversational agent, and if the information in each of the datasets is enough to create a complete dialogue system.

Furthermore, another possibility to expedite the development of conversational agents is reusing existing conversational agents. We believe that it is possible to transfer the knowledge gathered from one domain to create a conversational agent in a different domain. In this work, we conduct an experiment on transferring the knowledge acquired from the restaurant domain to the hotel domain.

1.1 Objectives

With this work, our main research question is the following:

RQ0 *How can we expedite the process of developing conversational agents?*

One of the possibilities to expedite the process of development is by using datasets containing dialogues. However, these datasets often have different annotations with different information, which can lead to either a lack of information or the existence of noisy data that is not required. Thus, we have another research question:

RQ1 *What is the information required from the datasets in order to automatically develop a conversational agent?*

As we presented above, the lack of information in the datasets might hinder the process of creating a conversational agent. This leads to a disparity between the capabilities described on the dialogues of the dataset, and the capabilities of the resultant conversational agent, with the last one lacking capabilities. For this reason, using the knowledge acquired from a different dataset might improve the quality of the training dataset. This leads us a third research question:

RQ2 *If one of the datasets is lacking information, can we use the knowledge from other dataset in order to improve its quality?*

For last, as previously mentioned, conversational agents are part of our day to day lives. This means that in order to complete a new task, a new conversational agent is developed. For this reason, one possibility of expediting the process of development of conversational agents is by reusing the knowledge from conversational agents developed for other tasks. The last research question is:

RQ3 *What knowledge can we reuse to create a conversational agent in a different domain of the original agent?*

1.2 Contributions

The contributions of this work are the following:

- A framework that creates a Rasa agent using the dialogues and the annotations in MultiWOZ[6] to help understand the required information to create conversational agents agents (RQ1);
- The framework used for the experiments with the Taskmaster-1 to create a Rasa agent and increasing the dataset information using the MultiWOZ knowledge[7]. This framework is used for understanding the required information from the datasets to create agents (RQ1) as well as improving the quality of the dataset by transferring knowledge from other dataset (RQ2);
- A framework that by using the restaurant agent of the MultiWOZ, creates a Rasa agent for the hotel domain[8], where we verify if it is possible to transfer knowledge between domains (RQ3).

1.3 Document Outline

In Chapter 2, we present some concepts that are important to be considered during our work. We also explain how we define a conversational agent in Rasa. For last, we present the datasets we use and how can they be used to our benefit. Chapter 3 provides information on other frameworks, as well as other works regarding the modeling of different components of a conversational agent. In Chapter 4, we present the details of the MultiWOZ dataset, as well as our process to transform its data in a Rasa agent. In Chapter 5, we address the transformation of the Taskmaster-1 dataset to a conversational agent, as well as knowledge transfer of one domain to the other. In Chapter 6 we explain the objectives, procedure and materials for the evaluation of the agents created and present the results we obtained. Chapter 7 discusses the results presented and make conclusions regarding our objectives and if they have been reached. For last, in Chapter 8 we draw the conclusions of our work, and present some future work.

[6] https://github.com/Fogoid/MultiwozToRasa
[7] https://github.com/Fogoid/TaskmasterToRasa
[8] https://github.com/Fogoid/KnowledgeTransfer

Chapter 2

Background

Conversational agents is a very vast topic in which the definition of concepts may vary across different authors. Firstly, we define important concepts related with conversational agents. Then, we introduce Rasa, an open-source framework and the requirements to create a conversational agent. For last, we present two datasets and relate them with the objectives we are trying to fulfill.

2.1 Core Concepts

When talking about conversational agents, Natural Language Understanding (NLU) is one of the most important parts, because it allows the agent to understand what a user sentence means. The most common tasks for NLU are:

Intent Classification Allows the agent to know what the user wants to accomplish, by labeling the user sentence as an *intent*. An example intent can be *reserve_restaurant*.

Entity Extraction This task enables the extraction of information present on the user sentence. These are called *"entity"* and can be names, locations or anything else.

Slot Filling For last, this one focus on setting values for *slots*. Slots are pieces of information that correspond to an important piece of data for fulfilling a goal, such as booking a restaurant.

#	Speaker	Utterance
1	**USER**	Hi! I would like to book a table in Pizza Hut for 4 people
2	**ASSISTANT**	What is the date and time of the reservation?
3	**USER**	It is for next saturday at 21:00
4	**ASSISTANT**	Your booking for Pizza Hut, for saturday at 21:00 for 4 people was successful.

Table 2.1: Example conversation between a user and a conversational agent to perform a booking at a restaurant

.

Consider Table 2.1 as a reference. In utterance 1, one possible intent for the utterance is *book_restaurant*, and the entities extracted would be "Pizza Hut" as *restaurant-name* and "4" as *n-seats*. Also, if we look at utterance 3, NLU module could identify the utterance as part of an *inform* intent, and extract "saturday" as *bookday* and "21:00" as *booktime*. Moreover, from the dialogue in Table 2.1, we can see that in order to book a table, the agent needs to know the *restaurant name*, the *number of seats*, as well as the *day* and *time* of the booking. For this reason, four slots would be created, where the values of the extracted entities would be mapped to the corresponding slot, fulfilling the slot filling task.

Also, we present some definitions regarding the dialogue management that are crucial for our work. We define these concepts since they can have different meanings or will be used several times throughout this document.

Policy A policy makes the decision on what is the next best action to take. There are several ways to implement a policy, but we will focus on two: the machine learning policies, that learn what is the best next action using training data, and rule-based policies, that choose an action following a set of defined rules.

Story This can either be a single-turn or a multi-turn conversation between the user and the assistant.

Context We define context as the entire conversation history between the user and the assistant until the current turn.

Form A form represents information that must be collected to complete a task. For example, to book a restaurant, we can have a form with the number of people, the restaurant name, and the day and time of the booking.

2.2 Rasa Framework

Rasa is an open-source framework that helps building conversational agents that can be text-based or speech-based, using machine learning. Use Listing 1 for illustration purposes. To create a new agent, we need to specify concepts such things as intents, entities, slots, responses, stories and forms, which together form the domain. The definition of the domain is what makes the assistant know what are the topics in which it will be able to work.

Slots in Rasa are extremely important. In a Rasa agent, slots are the information the agent requires to be able to perform the task correctly. Slots can be set by two ways: the first is if the slot has the same name as an entity that was extracted. The second is by being set by custom actions. Slots can assume a variety of types. They can be text, a boolean, categorical, a float, or we can even create any other slot type we want.

Listing 1 Example of domain definition of a Rasa agent

```
version: "2.0"

intents:
  - intent_A
  - intent_B
  ...

entities:
  - entity_A
  - entity_B
  ...

slots:
  slot_A:
  - type: text
  slot_B:
  - type: float
  ...

forms:
  example_form:
    required_slots:
    slot_A:
      type: from_entity
      entity: entity_A
    slot_B:
      type: from_entity
      entity: entity_B
  ...


responses:
  utter_response_A:
  - text: this is an example response
  ...

actions:
  - action_example_custom_action
```

General Architecture

Consider Figure 2.1 as a reference. To create the conversational agent, Rasa provides six modules. These are the Action Server, Tracker Store, Lock Store, Filesystem, NLU and Dialogue Manager, which we describe below.

Action Server This module allows us to create a special type of responses for the agent: custom actions. Custom actions can be an API call, an access to a database, or even just setting slots. There are two ways to run custom actions: the first one is using Rasa SDK, which implements the Action Server automatically. We can also create our own action server, from which Rasa will contact using REST calls in order to make the actions. This last option is especially good if there is the need to use code or business logic in a language other than Python.

Tracker Store Just as the name suggests, tracker store handles the storage of the history of a conversation.

Lock Store This module handles the processing of messages in the right order. In order to do so, Rasa provides a ticket lock mechanism ensuring that the messages received for a conversation are processed correctly by locking the conversation when a message is being processed. These can be obtained either by storing the locks in a single computational process or by creating a persistence layer to handle the processing.

Filesystem Once the agent is trained, a model is created that Rasa will use. From this module, we can choose where to load the trained model from. The model can either be loaded from local storage, from a server or even from a cloud storage.

NLU The NLU module is what enables the agent to extract information from a user sentence.

Dialogue Management This last module allows the agent to select what is the best action to perform according to the dialogue history.

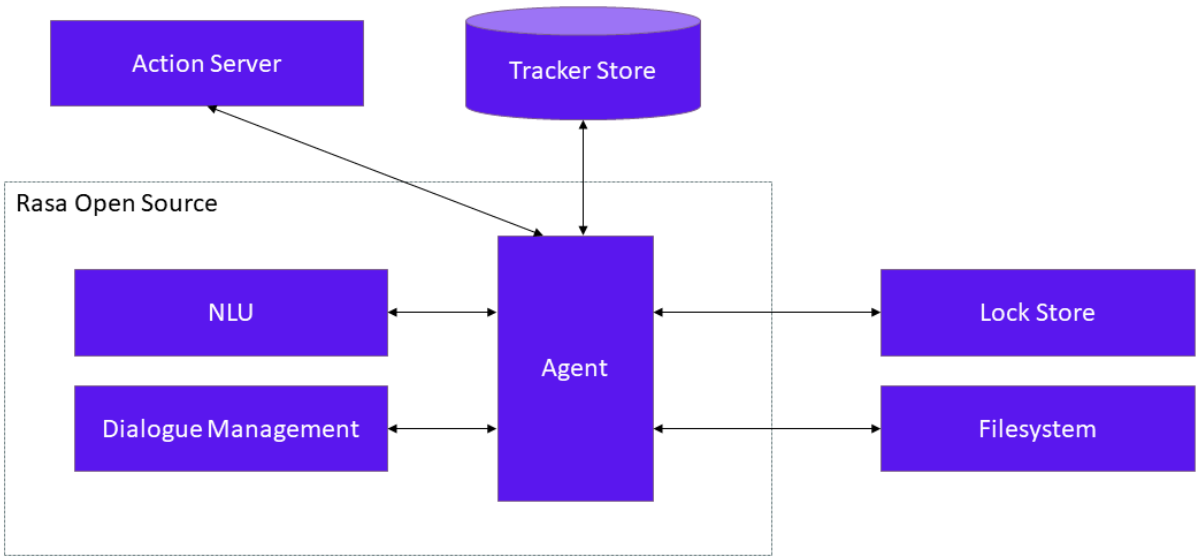

Figure 2.1: Rasa general architecture

Just as we said previously, the NLU and Dialogue Manager modules are the most important modules to define in order to have a conversational agent. For this reason, we explain in further detail how Rasa implements both of these modules in the next subsections.

2.2.1 NLU

Just as mentioned before, the NLU, also called NLU pipeline in Rasa, allows the extraction of information from a utterance. Just as the name suggests, this module is implemented using a pipeline, where

the developer has total control of which components to use to train the agent. Each of the components have a different task, and the result of the pipeline is a structured output of all the information combined from the components. The control of the NLU pipeline is given to the developer in order to have a NLU module that is best suited to the agent that is is created. To better understand how the NLU pipeline works, given the user message, and a NLU pipeline with n components, the input of the component x is the user message, plus the output of the components 1 to $x-1$. We also illustrate this process in Figure 2.2.

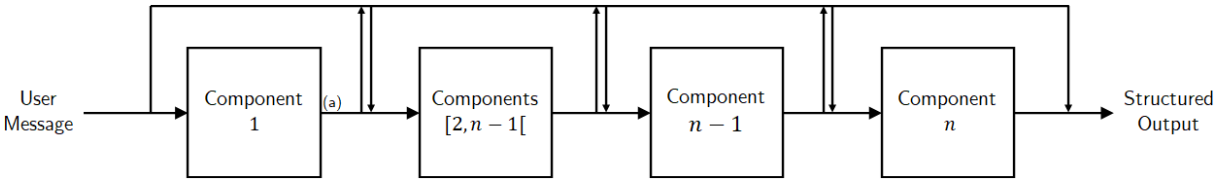

Figure 2.2: Process of a user utterance until reaching the output. (a) represents the output of the component. For a specific component, the input is the the output of all previous components, together with the user utterance

To customize the NLU pipeline, Rasa provides several types of components to complete different tasks. The types of components they offer are:

- **Language Models** These components bring pre-trained models, containing pre-trained word vectors. The language models offered by Rasa are the MitieNLP, the spaCyNLP, and HFTransformersNLP. With the first two components, we can specify the models that we want to use, either by choosing a model of a specific language, such as Portuguese, and we can also load our own pre-trained model. In the case of HFTransformersNLP, we need to specify the transformer we want to use. Also, this last component uses featurization and tokenization, computing the representations for the training examples in our agent.

- **Tokenizers** If there is the need to split the text into tokens, then tokenizers should be used. Some of the tokenizers available are a whitespace, and tokenizers for the messages, responses and intents. Note that some of these tokenizers require some specific language models components. For example, the MitieTokenizer requires the MitieNLP language model to work properly.

- **Featurizers** This creates vector representations for the user messages and responses, that can be used later by Intent Recognizers, Entity Extractors and Response selectors.

- **Intent Recognizers** This type of component is responsible for assigning an intent from the domain to a given user message. Depending on the component chosen, this can, or not, be a list of the intents in the domain with the correspondent confidence value.

- **Entity Extractors** Entity extractors are used to extract any entities from the user message. The entities extracted are the custom entities defined with the training examples and pre-defined entities, such as times, numbers, names or even distances.

- **Combined intent recognizer and entity extractor** Just as the name suggests, these types of components perform both the intent recognition and entity extracting tasks.
- **Response selectors** For last, this type of components try to select an action given a set of candidate actions. The information given by the response selector will be later used by the dialogue management to select the action to perform. However, this component only works if the agent has retrieval intents defined. Retrieval intents are a special type of intents that can be divided into smaller intents. This can be used, for example, for an FAQ, if there are two questions that are completely different but about the same topic.

Listing 2 Example of the training data used to train the NLU pipeline to find flights

```
nlu:
  - intent: find_flight
    examples: |
      - I'm looking for a flight
      - Find [moderately](price) priced flights for [2](n_seats) people
      - I want a flight from [Lisbon]("entity": "location", "role": "origin")
        to [London]("entity": "location", "role": "destination")
    ...

  - synonym: moderate
    examples: |
      - moderately

  - regex: n_seats
    examples: |
      - (\d)+

  - lookup: location
    examples: |
      - Lisbon
      - London
      ...
```

Take Listing 2 as reference. After specifying the NLU, we need to specify what are the intents, entities and training data. The training data composed by sets of utterances for each intent defined in the domain. Moreover, in order for the NLU to recognize the entities, we also need to have utterances that have the entity values annotated in the form *[value](entity)*. Furthermore, Rasa also offers different ways (which require specific components) of improving the intent classification and entity extraction tasks:

Synonyms Just as the name suggests, synonyms represent a set of words that all mean the same. When one of these words is recognized, it is mapped into the defined keyword.

Regular Expressions This type of training data can either be used to improve intent classification or entity extraction. This can be used for example to match numbers in a utterance to find an entity, or to match specific words that only occur in a specific intent, such as *"hello"* in a greeting.

Lookup Tables Rasa uses lookup tables to search for a value in a list and maps it to an entity. This can be seen as similar to regular expressions, since they also search for a match to identity the value as an entity.

Entity Roles and Groups Roles and Groups can be useful when we want to specify what is the purpose of an entity in a utterance. For example, looking at the sentence *"I want a flight from Lisbon to London"*, Lisbon and London are both locations, where Lisbon has the *"origin"* role, and London has the *"destination"* role. Note that the use of roles require a special annotation in the form *[value]("entity": "entity name", "role": "role name")*.

2.2.2 Dialogue management

After obtaining the information from the NLU pipeline, the agent needs to choose on how and what will respond to a user sentence. For reference purposes, consider Table 2.2.

#	Speaker	Utterance
1	**USER**	Hey
2	**ASSISTANT**	Hello. I hope you are well
3	**USER**	How many flights are there from Lisbon to London tomorrow?
4	**ASSISTANT**	Hmm... Let me see There are 3 flights tomorrow: at 9:00, at 15:00, and at 19:00

Table 2.2: Conversation between a user and an agent with the goal of finding the available flights for a day

With Rasa, the agent can respond to these two questions using different types of answering. In utterance 2, the answer of the assistant can be accomplished using **Responses**, while the answer in utterance 4 is accomplished using **Actions**. Both of these are defined in the domain, as we exemplify in Listing 1, and although they fulfill the same purpose, they are substantially different.

Responses are template messages that are sent by the agent. These templates are mainly text, but can also contain slot values, buttons or even images. Returning to the third utterance in Table 2.2, the utterance "Hey! I hope you are well!" can represent a template of a response.

However, responses are sometimes not enough to answer the agent questions. For this, reason, we use **custom actions**. Just as we specified earlier, custom actions allow us to run some code that may be required for the assistant to be able to answer correctly a user question. In example 4 of Table 2.2, the assistant needs to use a custom action in order to perform a search in the flight database in order to find what are the available flights for London on the day after.

Assume that after the conversation in Table 2.2 unfolds, the user wants to make a booking for the 15:00 hour flight. To accomplish this task, the assistant requires information such as the flight details, as well as the user information, such as the name and personal contact. One possible way of the agent to ask this information is by using **forms**. Forms can be seen as a custom actions that allow the agent to ask for all the information required to finish the task.

For last, Rasa also provides some default actions. These are present in the dialogue manager module and are allowed to be called at any given moment. For example, for confirming a task.

Just as the NLU module, the dialogue manager module requires training data to learn what is the best action to take. As training data, Rasa uses stories and rules, which allows Rasa to model the dialogue flow by giving instructions on what action to take given a conversation history and a user utterance.

Stories Stories can be seen as a multi-turn interaction between a user and an agent. In Rasa, each story represents a path the user can take. A story is represented by a set of intents, with or without entities from a user utterance, the assistant actions, and events, such as setting a slot and starting or ending a form. Stories are good because they allow the agent to generalize conversations, allowing the coverage of more paths. An example of a Rasa story can be seen in Listing 3, which represents a conversation path taken by the user in Figure 2.1 to perform a booking for a restaurant.

Listing 3 Story representation of collection of information to make a booking on a restaurant

```
- story: make booking at restaurant
steps:
- intent: book_restaurant
  entities:
  - restaurant-name: "Pizza Hut"
  - n-people: "4"
- action: utter_ask_day_and_time
- intent: inform
  entities:
  - day: "saturday"
  - time: "21:00"
- action: action_book_restaurant
```

Rules Use Listing 4 for illustration purposes. Unlike stories, rules do not generalize conversations. Rule are composed by a single turn of a conversation, which means that it has an intent and a corresponding action. Moreover, it can also have conditions that enables it's activation, such as the start of a new conversation. The use of rules is suitable for situations where the action of the agent is the same despite having different conversation history. One example of the usage of rules is to greet the user at the beginning of a conversation, or activating a form if the agent needs to collect information for a task.

Listing 4

```
- rule: greet at the beginning of the conversation
  condition:
    conversation_start: true
  steps:
  - intent: greet
  - action: utter_greet
```

As mentioned previously, Rasa takes a machine learning approach to develop conversational agents. For this reason, in order to select the best action to take, it uses Policies to learn what is the best action. There are two types of policies in Rasa: Machine learning policies which we will describe in further detail

and Rule-based policies, which is a policy that predicts the action based on the rules defined. Moreover, Rasa allows the developer to create his/her own policies. In a conversational agent in Rasa, several policies can can be used simultaneously, and Rasa chooses to execute the action from the policy that returns the result with the best confidence level.

Rasa offers three different machine learning policies to predict the answers: The *Memoization policy*, the *Augmented Memoization* policy and the *TED policy*(Vlasov, Mosig, and Nichol 2020). All three-policies use stories in order to predict the next action. The Memoization Policy and the Augmented Memoization Policy both remember all the stories and remember the current conversation for a selected number of turns. The difference between these two is that the last has a forgetting mechanism, forgetting some steps of the conversation history. TED Policy is the most different policy. Although we are not entering into great detail in how it works, this is an important policy since it allows the agent to generalize stories. When a new user utterance is received and all structured information is obtained (such as intents and entities), these are put together with with the previous action, slots and any active forms and outputs the best next action to execute and its accuracy.

2.3 Use Cases

To study our problem, we take into consideration two datasets: the MultiWOZ dataset (Budzianowski et al. 2018) and the Taskmaster-1 dataset (Byrne et al. 2019). Both of the datasets contain annotated dialogues for several domains.

2.3.1 MultiWOZ

The lack of data concerning dialogues led to the creation of the MultiWOZ dataset(Budzianowski et al. 2018). This dataset is a major contribution since it has more than ten thousand dialogues across eight different domains. The domains this dataset covers are: restaurant, hotel, attraction, taxi, train, hospital, police and bus. These conversations were created using a Wizard Of Oz(Kelley 1984) approach, that consists basically of two people interacting, one acting as the “wizard”, and the other as the “client”. This allowed the collection of dialogues with more diversity and semantic variety.

An example dialogue in MultiWOZ can cover a single domain, or two or more domains. Moreover, the authors of the dataset define several action types, such as inform, request or recommend and also define slots, like address, name or destination. Both the action types and the slots can be universal to all domains, or apply to a single domain. For example, departure is a valid slot for taxi and train domains and the action type recommend is available only for restaurant, hotel and attraction. For each turn in a dialogue, an action type is associated, and all the entities extracted are stored to be used later in the conversation.

2.3.2 Taskmaster-1

Just as MultiWOZ, Taskmaster-1(Byrne et al. 2019) is a dataset that contain example conversations regarding a specific task. To collect these dialogues, two approaches were followed, creating two dialogue types:

Two-Person Dialogues These dialogues were followed using the Wizard-Of-Oz(Kelley 1984) approach, just as MultiWOZ. In this approach, different instructions were given to both the client and the "wizard", that both needed to follow to complete a certain task.

Self Dialogues In these dialogues, one person was responsible of coming up with dialogues acting as both the agent and the person. Instructions were also given regarding the task to be performed.

In both the self dialogue instructions and the instructions given to the worker in the two-person dialogues contained all the information the agent would ask. This information can be, for example, in a reservation for a movie, details such as name of the movie, number of people, session hour and city. For the client instructions in the two-person dialogues, it was just specified to make sure all the information required was asked by the assistant.

Taskmaster-1 dialogues fall into one of six domains: (1) pizza delivery, (2) auto repair appointments, (3) upride services, (4) movie tickets, (5) coffee drinks and (6) restaurant reservations. As for the annotation of the dialogues, for each utterance, the entities that are mentioned are specified, as well as any API calls that were performed.

Both of the datasets we just presented are important for our work. Although the two datasets provide dialogues regarding the task, the information given by both is different, which allows us to understand what is the important information to model a conversational agent.

Chapter 3

Related Work

In this chapter, we present some works where the tasks of intent classification and slot filling are improved. Also, we talk about the way some authors propose the development of the dialogue manager. For last we investigate some frameworks that have been created to develop conversational agents. We end this chapter by showing some of the evaluation methods for dialogue systems.

3.1 Improvement of Intent Classification and Slot Filling

To create a conversational agent, it is important to have a good NLU module that is able to identity correctly the information that is in a user utterance. As we stated previously, two of the most important tasks of this module are the tasks of intent classification and slot filling.

For this reason, several works have focused throughout the year trying to improve both tasks. When we talk about these two tasks, the relation between the two seems obvious, but only recently have been made proposals where the two tasks are performed simultaneously. Some of the models proposed are based on RNN with attention mechanisms(Liu and Lane 2016; Goo et al. 2018) achieving state-of-the-art performances for both tasks. Goo et al. (2018) introduced a slot gate mechanism on top of the attention mechanism, that allowed to improve the task of slot filling, by exploring the correlation between the two tasks. Qin et al. (2019) defends that is is risky to rely on the gate mechanisms, proposing a joint model using stack-propagation. Moreover, the authors also use a token-level intent prediction, that can provide additional information to slot filling.

3.2 Conversational Agents Frameworks

In recent years, many companies have developed their own frameworks to create conversational agents. Some of the most popular frameworks are Bot Framework + LUIS[1], Amazon Lex, Dialogflow, Wit.ai[2] and Watson Conversation[3]. Most of these systems provide either an NLU and Dialogue Manage-

[1] https://www.luis.ai/
[2] https://wit.ai/
[3] https://www.ibm.com/cloud/watson-assistant

ment component. In this section, we provide some of the functionalities of each framework and make a comparison to Rasa.

In order to improve the workflow while developing the conversational agent, all the frameworks we mentioned provide a simple user interface for the NLU module. In order to extract information from user utterances, all of the frameworks (besides Amazon Lex) present two concepts: *intents* and *entities*. Just like Rasa, in order to extract intents and entities, the developer needs to add training examples to the framework, where the information is annotated in each utterance. For example, to add the sentence *"I am looking for a flight for tommorow"* as a training example, the developer needs to specify the intent as *find_flight* and label the value *tomorrow* as a *date* date.

In order to label the information, Amazon Lex does not provide entities, but provides what they call slots. To annotate the training data, instead of providing examples for the values the slots can take, the utterance is modified to have the slot in the correct location. For example, in the utterance we mentioned previously, the correct annotation in Amazon Lex would be *"I am looking for a* $\{$*flight*$\}$ *for* $\{$*date*$\}$*"*. Moreover, when an intent is created, it contains a list of slots associated. In order to complete the intent, all the slots associated must be provided.

As we have seen previously, Rasa provides all of these concepts. In order to annotate the information in the user message, Rasa uses intents and entities, and although using a different approach, Rasa also implements slots to store all the important information in order to complete a task. Moreover, one advantage that Rasa has over other frameworks is that these frameworks do not provide any information about the learning process nor allows the choice over the learning algorithms, which is attained in Rasa by having a fully customizable NLU by using modular components, which allows further customization capabilities and transparency in the process.

For last, while all of the frameworks are created and maintained in a cloud environment, Rasa gives this choice of where to keep the conversational agent to the developer. This way, Rasa still has the advantages of scalability and managed hosting, but also allows the developer to have more control over its data and more adaptability to its needs.

Now that we have covered the how each framework implement the NLU module, let us investigate how some of these frameworks define the dialogue manager module and use it to respond to a user utterance. Although all of these frameworks besides Amazon Lex provide a way to model the conversation, we will look closely at the implementations of Dialogflow and Microsoft Bot Framework.

3.2.1 Dialogflow

Take Figure 3.1 for illustration purposes. To manage the dialogue, Dialogflow offer contexts, which can be divided in two types: input and output contexts and can either be active or inactive. Contexts can be set in the agent intents. For example, in Figure 3.1, there are three intents: Flight Reservation Intent, Confirm Reservation Intent, and Cancel Reservation Intent. The Flight Reservation Intent has two output contexts: Confirmation and Cancel, while the Confirm Reservation Intent and Cancel Reservation Intent have each one input context: Confirm and Cancel respectively. Consider now that a user starts a new

conversation the following message:

I am looking for a flight for London.

At this moment of the conversation, there are no contexts active. This means that only one intent will be considered to be matched, the Flight Reservation Intent. After being matched, then the contexts Confirm and Cancel are activated. Consider now the user sends the next message:

I don't want to find a flight anymore

Now that the Confirm and Cancel input contexts are active, all three intents are considered for the matching, being the Cancel Reservation Intent chosen for this specific utterance. After this utterance, there are no active context.

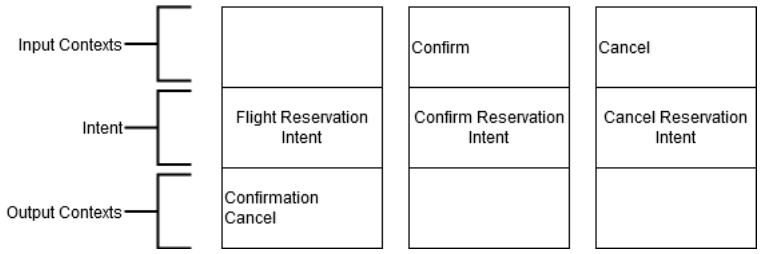

Figure 3.1: Example intent definition on Dialogflow. The upper space of the intents represent the input contexts while the lower space represent the output context

By looking at this example, we can see that although the implementation of a conversation path in Dialogflow is different to Rasa's, we can also model stories in both frameworks, since they have the ability to specify what is the possible paths the user can take in a dialogue. Moreover, in to respond to the user, dialogue offers Responses and Fullfillments, that can be defined in each intent. While Responses are template messages, and fullfillments, allows dynamic responses to the user, which is the equivalent of Responses and Custom Actions in Rasa.

3.2.2 Bot Framework

Bot Framework has two ways to create a conversational agent which will deal with the flow of the conversation between a bot and a user: (1) By Composer[4] where the bot is modeled by a visual interface that does not require any type of coding or (2) the SDK[5] where all the conversational agent is developed using code. Even though they have this difference, they both implement the same way of dealing with flow using Adaptive Dialogs.

Before talking about these, we first need to establish the concept of a Dialog in the context of Bot Framework and Adaptive Dialogs. A Dialog can be seen as a way to manage conversations with the user and can represent only a part of the whole conversation with the user. Taking this into account, a dialog can begin, continue and end.

[4]https://docs.microsoft.com/en-us/composer/
[5]https://docs.microsoft.com/pt-pt/azure/bot-service/index-bf-sdk?view=azure-bot-service-4.0

Looking now at Adaptive Dialogs in more depth, there are two types of dialogs: the main/root dialog and the child dialogs. To begin a child dialog, either the root dialogue or another child dialogue must use an action. To create a dialog, we have to define five main components:

- **Recognizer** This is what interprets what the user means with the input he just gave, by giving structured information which are intents and entities. This information is gathered using several recognizers, such as LUIS, Regular Expressions or even custom recognizers.
- **Trigger** These represent the functionality of a Dialogue. Triggers can be seen as a function that processes the input messages and can define bot behaviours. However, these require rules that tell when to execute a trigger. Some of these rules are either recognized intents (OnIntent), or even custom events.
- **Action** A trigger is composed by several actions. These actions are what the bot will do in order to complete the request. An action can be various things such as making computational tasks and making API calls, or even storing and loading values, prompting the user and managing dialogues, such as starting or ending dialogues.
- **Language Generator** This component allows the bot to create messages using variables and templates developed and gathered during the dialogue.
- **Memory** This last component stores all the important properties for the dialogue, such as the properties (for us called slots), conversation history and turn information.

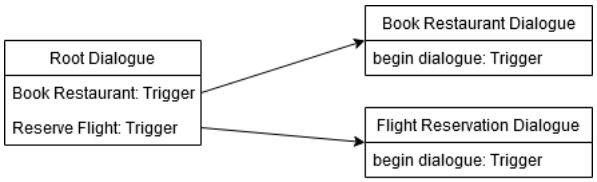

Figure 3.2: Representation of an Adaptive Dialogue. Each dialogue contains triggers. Each arrow represents the connection between the root dialogue trigger and child dialogues

To fully understand how adaptive dialogs manage the conversation, take Figure 3.2 as a reference. As we can see in this figure, there are three dialogs: the Root dialog, the Book Restaurant Dialogue, and the Flight Reservation dialogue. Moreover, the Root dialog has two onIntent triggers, the Book Restaurant trigger that is required to activate the Book Restaurant dialogue, and the Reserve Flight trigger that is used to activate the Flight Reservation dialogue. Both the Back Restaurant Dialogue and the Flight have a begin dialogue trigger. Finally, let us assume that in a conversation the Root dialog and the Book Restaurant dialog are active.

If the user sends a message with *ReserveFlightIntent*, then Bot Framework will find the first dialogue in the set of active dialogs that has an OnIntent trigger with ReserveFlightIntent. This would mean that the trigger matched would be the Reserve Flight trigger from the root dialogue. This makes Book Restaurant Dialogue pause, and the trigger from the root dialogue to be executed, starting the dialogue for flight reservation, setting it active. Once this dialogue finishes, then the conversation flow returns to

where it was previously, that was the execution of the book restaurant dialogue. This means that nothing is lost in the conversation, making it more flexible.

By providing Adaptive Dialogs, Bot Framework provides a very different but interesting approach to manage a conversation with the user. Although this approach does not make use of machine learning, it still provides flexibility in the way the dialogue is managed. However, this also means if a conversational path is outside of the logic defined in the dialogs, the agent will not know how to respond. This is one of the advantages of Rasa, where by using a machine learning approach and using the conversation paths defined in the stories as training data, the conversational agent is able to generalize and learn how to answer, even in a conversion that was not considered during the development.

3.3 Dialogue Management

In this section, we address how some works define the dialogue management. 1995, Aust and Oerder defined the flow to inquiry systems. In this paper, the authors divide the agent questions in three different types:

- **Disambiguation questions** - Questions that are used to choose the correct value for a slot that has already been mentioned previously.
- **Extension questions** - Ask for the value for slots that the agent requires to complete the task but does not have yet.
- **Follow-up questions** - Are used to ask if the user wants to continue the conversation after the task has been completed

Moreover, to choose what question must be chosen, the authors also define several rules, which are executed in the following order:

1. If there is any slot that needs to be validated, ask a disambiguation question
2. If there are still slots that are required by the agent, choose one as ask the user for it
3. If the task has been completed, ask if the user requires anything else
4. Finish the dialogue

Then, based on the type of questions and the rules defined by the authors, they create a flow by specifying every question that can be made, add the preconditions to be asked, and let the interpreter decide what is the question that should be asked by taking into consideration the dialogue history.

Xu and Seneff (2010) also proposed a system to deal with dialogues that instead of being an action-based, is entity-based and constraints-based. In this paper, the authors suggest that at the end of a task, the system presents an entity with the goal. For example, in the reservation of a flight, it can be a flight itinerary. Moreover, entities have properties that are relevant information, such as the flight numbers. Also, the constraints define what sis the required information to get a valid flight, such as the *source* and

destination. By having both the entities and constraints, the authors can then create dialogue rules that lead to the action to take.

Moreover, the authors define language understanding component and a language generation component, and also several knowledge sources, that have information about a specific domain such as times, dates or locations, or even a flight database, that provide information about flights during the conversation. In this paper, the user is considered as a knowledge source. To understand how the knowledge sources work, consider a conversation where the user has stated that he intents to fly to London. One of the requirements for the system to perform the booking of a flight is knowing the departure city. In order to learn what is the departure city, first the system goes to all knowledge sources, and if it can not find the value for the departure, then asks for the value to the user, by using the natural generation component.

But how do the authors use the entities to define the flow? For that, the developer needs to provide the form of the goal entity, and specify any relationships between the relevant entities. There are two steps to accomplish this: 1) the required knowledge sources need to be declared and 2) all entity types must be defined. For one domain, there is one goal entity, being "goal" its type, and can be an infinite number of other entity types. A type of an entity can be for example an itinerary or a flight.

3.4 Evaluating Dialogue Systems

Evaluation is a very important part of the development of a dialogue system. With it, we can measure how good our dialogue system is in a point in time. However, evaluating these systems is by no means easy, since the evaluation of each dialogue system has different purposes. In an attempt to evaluate these agents, many metrics have been proposed throughout the years and although progress has been made in creating automatic evaluation metrics, human based evaluation continues to be an important part of the evaluation.

For task-oriented dialogues, there are two main things to be evaluated: the success of a task success and how efficient is the dialogue. The first focuses on if the information was both gathered by the user and the dialogue system. The latter is normally related to how much time and turns of interaction is required to complete a task or just filling a slot.

One way of evaluating these metrics is by human evaluation. The users start by using the conversational agents, asked to perform certain tasks, and latter asked to fill a questionnaire about the interaction. Using this method, both the task success and the dialogue efficiency can be evaluated, since it is possible to retrieve information on how does each user interact with the model. Moreover, the satisfaction of the user can also be modelled using the ratings from the questionnaires.

Some frameworks have also been proposed to model the user satisfaction. One of these frameworks is PARADISE (PARAdigm for DIalogue System Evaluation)(Walker et al. 1997). The purpose of PARADISE is that by taking several metrics that we mentioned above into consideration, it computes the overall performance of the conversational agent. To better comprehend how does the framework operates, consider the reservation of a flight with the following slots: departure_date, origin, destiny and

time_range.

To calculate task success, PARADISE starts by transforming a task in a attribute value matrix (AVM), that represents the information flow between the agent and the user during a dialogue. To create an AVM for the reservation, we specify the slots, the possible values for each, and specify the receptor of the slot information (the agent or the user). For example, for origin, possible values are such as Lisbon, London or Madrid, and the information flow is from the user to the agent.

After creating the AVM for the task, PARADISE uses the Kappa coefficient to compute the measure. The Kappa coefficient uses a confusion matrix that represents the summary of how good is the agent at collecting the information for a task from the example dialogues.

Also from the AVM representation and the example dialogues, PARADISE takes information such as repair utterances and number of utterances for each dialogue and to fill a slot.

For last, it measures user satisfaction by the realization of surveys and taking an overall score. Finally, it combines all the metrics just mentioned, along with other metrics such as task duration and calculates the overall performance of the dialogue system using a linear regression. Also, PARADISE has a great advantage of being able to create a representation for any task, meaning that is independent of the domain we create the task for.

PARADISE gives us a great advantage of being able to create a representation for any task, meaning that is domain independent.

In some cases, we must create a baseline when there are no other systems to be compare our system. To evaluate his system Baptista (2020) developed three prototypes, with three user studies during the implementation cycle. This allowed him, on each user study, to collect metrics about the prototypes and created a baseline for the second and third prototypes, on which he was latter able to work on to improve further the final prototype.

On another note, Paek defends that the “perfect” baseline consists of human-to-human dialogues on performing a specific task, using the Wizard-Of-Oz approach, which we described earlier in Section 2.3.1.

Chapter 4

From the MultiWOZ dataset to Rasa

In this chapter, we begin to address what is the information required from the datasets to develop conversational agents (RQ1) using the MultiWOZ dataset. In the process, we show the details of MultiWOZ, and explain our process to transform the dataset in a Rasa model.

4.1 The MultiWOZ Corpus

As said previously, MultiWOZ is a dataset that contains dialogues created using the Wizard-Of-Oz approach and it has conversations from 8 different domains, where one dialogue might have messages regarding a single domain or more.

Utterance	**Dialogue State**	
USER: Can you help me find an expensive restaurant in the west?	**Active Intent**	find-restaurant
	Slots	restaurant-pricerange: expensive restaurant-location: west
ASSISTANT: Yes, what type of food are you looking to eat?	-	
USER: I would really like to have Indian tonight.	**Active Intent**	find-restaurant
	Slots	restaurant-pricerange: expensive restaurant-location: west restaurant-food: indian

Table 4.1: Example of a dialogue being updated after user giving information about the price range and location in the first utterance, and the type of food in the second utterance.

Consider Table 4.1 for illustration purposes. Each dialogue is composed by a set of turns, the user turn and the assistant turn. In both the user and assistant turns, the identity of the speaker and the correspondent utterance is given. Furthermore, an updated state of the dialogue is also given in every user turn. This can be seen in Table 4.1, that in the second user utterance, the state contains both the information given in the previous turns and the restaurant-food slot it just mentioned.

The dialogue state is composed by three different types of information. First, there is the Active Intent. The Active Intent represents the general task and objective that the assistant is trying to achieve. For example, in Table 4.1, the conversation being made is for the purpose of finding a restaurant for the user.

Moreover, we also have the slots, which is a representation of all the information the user has given and can collect during the course of the conversation, as seen in Table 4.1. When there is a new slot, the value of the same is part of the utterance. Furthermore, each domain has specific slots. For example, for the restaurant domain, the slots are: restaurant-area, restaurant-pricerange, restaurant-location, restaurant-food, restaurant-name, restaurant-bookpeople, restaurant-location, and restaurant-bookday.

Utterance	**Dialogue State**	
USER: I don't need a ticket right now, but could you tell me the travel time and price?	**Requested Slots**	train-duration train-price
ASSISTANT: A single ticket costs 16.50 pounds and total travel time is about 50 minutes. Will that be all today?	-	

Table 4.2: Dialogue state when the user requests the price and duration of a train.

For last, it is also important to take a look at the requested slots. Requested slots are represented by its name. An example of requested slots can be seen in Table 4.2. In the conversation, the user asks what is the *travel time* and the *price* of the train that is looking for. As this happens, the state of the dialogue contains the name of the slots being requested, *train-duration* and *train-price* respectively, but does not have a value associated.

Now that we stated all the information of the MultiWOZ dataset, we are now able to come up with a solution to design a Model for Rasa. An overview of the conversion process is in Table 4.3.

MultiWOZ data	**Rasa Open Source**
Active Intents Slots Requested Slots Utterance	Intents NLU Training Data
Slots Requested Slots	Entities and Slots
Active Intents Slots Requested Slots Mandatory task slots*	Actions and Responses Stories and Rules

Table 4.3: Overview of the information used to create the Rasa details. *These slots are computed for each active intent in MultiWOZ (Explained in detail in Section 4.2.2).

4.2 Converting MultiWOZ to a Rasa Model

To develop a conversational agent in Rasa, there are three properties to be defined: the agent's domain, the training data for the NLU and the conversational paths the agent can take during a conversation, which define the flow (Rasa stories).

4.2.1 Domain and NLU Training Data

Since the information for both the domain and the NLU training data is analogous, the retrieval of this information results from a single analysis of the dataset dialogues. In this subsection, we address how we define the Rasa intents, entities and slots, and how we annotate the user utterances as training data.

Creating Rasa Intents

Just as stated above, MultiWOZ gives in each state what is the the Active Intent. Also, the Active Intent can change during the conversation. One example of that is represented in Table 4.4. In this example, the user starts by looking for an hotel and then proceeds to a booking. During this section, we call the Active Intent as the task of the user.

Utterance	**Active Intent**
USER: Do you have any information on the University Arms Hotel? I've heard it's a nice place.	find-hotel
ASSISTANT: I sure do. The University Arms hotel is a more expensive hotel in the centre of town. It has 4 stars and includes free wifi and free parking.	
USER: I need a room for one person starting sunday for 3 nights.	book-hotel

Table 4.4: Example of conversation where the task-intent of the user starts by looking for a hotel and then decides to make a booking

Since the task can change during the dialogue, every time we have a utterance where the task changes, there are two steps we perform: if the task was never seen before, we create a new intent for the Rasa agent with the same name. Furthermore, we label the utterance as part of the same intent. The intents that originated from the tasks, as well as an example of utterances that have been labeled with each intent can be seen in Table 4.5.

Take Table 4.1 as example. Many of the times where there is a new utterance, the user is giving details about the task he/she is looking to complete. For this reason, it is also important for the assistant to collect this information. To this purpose, we create a new type of intent: the **inform** intent. In a first approach, every utterance where the task is the same of the previous state is labeled as part of **inform**. In Table 4.1, since the task is the same in both user turns, then we label this utterance as **inform**.

However, we quickly came to realise that not all conversations that continued with the same task had additional information from the user. Take Table 4.6 as a reference for the rest of this subsection. Sometimes the user could ask for a recommendation (1), affirm (2), deny (3), request information (4), or

Intent	Utterance
find_restaurant	I'd like to find a restaurant called cotto, please.
book_restaurant	Great can you book a table for 8 people at 19:00 on wednesday?
find_hotel	My husband and I are celebrating our anniversary and want to find a great place to stay in town
book_hotel	Yes please. I'll need it booked on sunday for 5 nights and it will only be 1 person
find_train	I need a train going to bishops stortford
book_train	Yeah, can you book 4 tickets for me?
find_attraction	I want to check out some colleges around here
find_taxi	I'll also need to get a taxi to go between the 2 places
find_hospital	I have a stomach ache, is there a hospital or pharmacy nearby?

Table 4.5: Example of utterances for each of the task intents that gave origin to Rasa intents.

at most cases, give information (5). Since each of these lead to different actions by the assistant, then there is the need to differentiate all these utterances.

For the cases where the user gave information, the differentiation to the other messages types was simple, given the properties of the dataset. This way, we take information from the updated state of the dialogue. If there are new slots in the current utterance, such as utterance 5 in Table 4.6, then we know for sure that the utterance contains new information, labeling it as **inform** (the process of finding slots in the utterances is described in Section 4.2.1).

Moreover, since the dataset contains the requested slots and these are updated in the state of the dialogues, we are able to annotate utterances such as utterance 4 in Table 4.6. For this reason, we create the **request** intent. If the user asks for information instead of giving to the user, then we annotate the utterance is as **request** (the search for requested information in the utterance is described in Section 4.2.1).

However, we are not able to differentiate the meaning of utterances like 1, 2 and 3 in Table 4.6. This is because there is no information in the dataset that can indicate what is the intention of the user in these utterances. Although we recognize that these utterances are important for a dialogue, we decided to not use the dialogues containing these types of utterances as part of the Rasa model, since we believe they would add noise to the training of this model.

#	Utterance
1	Not really. I'm down for anything what would you recommend?
2	Yes that will be fine
3	No, that doesn't matter
4	What is the address, please
5	I would like to leave on sunday and arrive by 17:15

Table 4.6: Example of followup utterances after stating the intent of finding a restaurant.

For last, we also noted that at the end of almost every dialogue, there would be utterances where the task is **NONE**. This means that the dialogue continued, although there was no task to complete. To verify if it was the end of the dialogue, when a **NONE** task is found, we search until the end of the dialogue if all utterances are also part of the task **NONE**.

Since we wanted the created models to be as more complete as possible, we assume that in the end of the dialogue, every utterance that has the task **NONE** is for the purpose of ending the dialogue. With this assumption, we create the intent **goodbye**, and label the utterances that fulfill these properties as **goodbye**. Some examples of the labelled utterances are in Table 4.7.

Utterance
Thanks, that's all I need. Have a nice day
No thank you. You have helped me tremendously and I have everything I need to know. Thanks and have a good day
That is all. Thanks for you help
Nope. I'm all set. Thanks again
Great! Thanks a lot. That's all I will be needing. Have a good day. Bye!

Table 4.7: Example of utterances labeled as **goodbye**

Slot Recognition and Rasa Entity mapping

Just as we seen previously, utterances can contain important information to complete the task correctly, which is represented as slots in MultiWOZ. To use this information in Rasa, we create in the domain an entity and slot for each of the slots that exists in the MultiWOZ dialogue. Moreover, during the dialogue, we use the value of the slot and the utterance to annotate it for the training data.

To find the values of the slots in the utterance, recall the properties of the MultiWOZ dataset. Every time we have a user sentence, the state of the dialogue contains the slots with the values until that utterance, as well as the new slots values found in the last utterances (both user and assistant utterances). Since we have not only the slot name but also the associated value, the annotation of the slots in the utterances is simple. To do so, we only require to access the value of the slot in the dialogue state, and perform a search in the utterance. If the value is found, then we update the value with the format *[slot_value](slot_name)*, which Rasa uses during training. An example of the annotation is in Table 4.8.

Utterance	**Slots**	**Value**	**Result**
I'm looking for a local place to dine in the center that serves chinese food.	restaurant-food	chinese	I'm looking for a local place to dine in the [centre](restaurant-location) that serves [chinese](restaurant-food) food.
	restaurant-location	centre	

Table 4.8: User utterance that contains information regarding the information about the food he wants to eat and the result after it is annotated.

However, this simple approach might give some problems. This can happen for example in the hotel dialogues. Consider the following utterance:

I want a booking to 4 people.

Assume this utterance is from the hotel domain, and it contains a slot, the *hotel-numpeople* with a value of *4*. Moreover, the state of the dialogue already contains a *hotel-stars* that also have the value *4* which was given in a previous turn. If we simply perform the search of the values of the slots in the state, then the annotated value in the utterance will be *hotel-stars*, which is incorrect.

To prevent this from happening, before searching for the slots in the utterance, we verify if they fulfill the two properties described below:

- The slot never appeared in a previous state
- The value of the slot in the current state is different from the previous state

If any of these two properties are not fulfilled, then we do not search for the slot in the utterance. Table 4.9 illustrate these properties. Using this properties, we confirm that the slot we search for is a slot that is present in the utterance and is new in the dialogue state.

Previous State	**Current State**	**Slots to find**
-	hotel-stars: 1 hotel-location: centre	hotel-stars hotel-location
hotel-location: centre	hotel-type: guesthouse hotel-location: centre	hotel-type
train-leaveat: 15:00	train-leaveat: 17:00	train-leaveat

Table 4.9: Slots to find after changes in the state of the dialogue

Unfortunately, there is also a downside to this decision. Values that are present in the user utterance, but have also been present in a previous utterance end up not being annotated, leading to a wrongfully annotated utterance. This can be seen in the sentence below:

What about [indian](restaurant-food) restaurants? Are there any ***expensive*** *ones in the [west](restaurant-area)?*

In this sentence, both *restaurant-food* and *restaraunt-area* are slots that have been annotated. This is because in this case, the value of both slots have changed from the previous state to the current state. However, the value in bold, which should have been annotated as *hotel-pricerange* was not due to our selection properties. Since this slot was mentioned before in the dialogue, and its value did not change, then the sentence is not correctly annotated.

Consider now Table 4.10 as reference. During this procedure, we noticed that, in some slots, the value in the state of the dialogue does not correspond to the value of the slot in the utterance.

As we see in the first utterance of Table 4.10, although it references the *hotel-internet* slot, the value stored in the state is completely different, which means that our algorithm will not be able to find the value to annotate. However, a different thing happens on the second and third utterances. If we look closely, the utterance values are composed by the state value with a suffix. This means that our algorithm will

Utterance	Slot	Utterance Value	State Value
Expensive guesthouse. it should have free parking and wifi	hotel-internet	wifi	Yes.
I'm looking for a moderately priced Spanish restaurant.	restaurant-pricerange	moderately	moderate
Don't care Looking for a hotel that includes free parking and should be cheaply priced.	hotel-pricerange	cheaply	cheap

Table 4.10: Difference of value between the state and the utterance

find the state value in the utterance, but without the suffix. For this reason, when the length of the word in the utterance is greater than the the state value, then we search for the state word, go until the end of the utterance word, and annotate it. Moreover, after annotating the correct value, we also mark it as a *Synonim* to the state value. We have decided to add synonims because of the simplicity of dealing with slot values later during the execution of custom actions.

For last, we also noted a problem in the slots where its values could be represented with numbers or text. Two examples of these type of slots are the slot *restaurant-bookpeople* and *train-arriveby*. For example, for the first slot, the user can state he/she wants a booking either for **two** people, or **2** people. For the second slot, the same can happen. The user can state he/she wants to arrive either by saying at **8:00** or at **8 o'clock**. Even though including more ways of stating the value of slots is good for the training data since it increases variety of speech, this also means that it would be required a vast range of training examples for the model to be able to learn all the possible numbers and hours. Since we have limited data, we opted to limit the values each slot could take, by including in our solution a *regex* for entity recognition for each of these types. During the labeling of the entity values, we first test the hour recognition and then the number recognition. If the value of the entity matches one of the regex, then we label the entity as a regex and add it to the Rasa NLU, just as we exemplify in Listing 5.

Listing 5 Example of regex for the entities train-arriveby and restaurant-bookpeople.

```
nlu:
- regex: train-arriveby
  examples: |
    - (\d){2}:(\d){2}

- regex: restaurant-bookpeople
  examples: |
    - (\d+)
```

Using this process, it is possible to recognize any whole positive number or any hour in the form of *hh:mm* that the user might give during the interaction. Besides, even though several entities will have the same regex, the agent labels entities based on the context and determine the correct entity for a value in a user utterance. The final algorithm to slot labeling can be seen in Listing 1.

Algorithm 1 Annotation of valid slots in the user utterance

```
FindSlotsInUtterance(Utterance, Slots):
    ValidSlots = GetValidSlots(Slots)
    for Slot in ValidSlots:
        StateValue = GetStateValue(Slot)
        Found = Search(Slot, Utterance)
        if Found:
            UtterValue = GetValueInUtterance(Slot, Utterance)
            if length(UtterValue) > length(StateValue):
                Annotate(UtterValue, Utterance)
                AddAsSynonim(UtterValue, StateValue)
            else if UtterValue == StateValue:
                Annotate(UtterValue, Utterance)

            if Is_Match(HourRegex, UtterValue):
                SetSlotAsRegex(Slot, HourRegex)
            elif Is_Match(NumberRegex, UtterValue):
                SetSlotAsRegex(Slot, NumberRegex)
```

Mapping Requested Slots to Rasa Entities

Reminding the knowledge of the MultiWOZ dataset, requested slots are slots that appear in the state of a user turn of a dialogue, and represent the information that the user requests in the utterance during the current turn. Moreover, requested slots do not have a value in the user utterance. An example of this slot is *restaurant-phone*.

Even though MultiWOZ did not store the value in the utterance that identified that slot, it was not difficult to understand that the right side of the name of the slot could be found in the user utterance. To annotate the values found in the utterance, we create a new Rasa entity called **requested_info**.

#	Utterance	Slot	Search Word	Result
1	Nope but can I get the postcode for that?	restaurant-postcode	postcode	Nope but can I get the [postcode] (required_info) for that?
2	How many stars is it?	hotel-stars	stars	How many [stars] (required_info) is it?
3	Sure, can I please have the phone number, post code and entrance fee information.	attraction-entrancefee attraction-postcode attraction-phone	entrancefee postcode phone	Sure, can i please have the [phone] (required_info) number, [post code] (required_info) and [entrance fee] (required_info) information.
4	Can you give me the reference number, please?	hotel-ref	ref	Can you give me the [reference] (required_info) number, please?

Table 4.11: Example of finding the requested slots in utterances

Take Table 4.11 as a reference for the next paragraphs. By analysing the table, we can conclude that we need to follow different strategies to find the required values to annotate in the utterance. In examples 1 and 2, there is no need to change the value to find. This is because if we search for the value in the utterance, there is a match.

On example 3, we can also find the slot *attraction-phone* by just searching the value *phone*. However, for the slots *attraction-entrancefee* and *attraction-postcode*, we can not find the slots with this approach. However, we noticed that although the value of the slot does not match the value in the utterances, we could find the values *post code* and *entrance fee* if we had a space to the values to search. To accommodate this property, if the value of the slot is not found, then we divide the value in two and search for the words originated from the division. The division of the value in two is done from left to right, and is repeated until either the end of the value is reached, or if the value originated from the division is found in the utterance. For example, for the slot *attraction-postcode*, the initial search value is **"postcode"**. Since this value is not found, we change the value to **"p ostcode"**. Since we did not find the value on the utterance, we keep repeating this process until the divided value is **"post code"**, where we are find the value in the utterance. Following this approach, we guarantee that we can find the values of slots which name have different lengths and require a division in different parts of the slot value to be found.

For example 4, which is a particular case in MultiWOZ, there is still the need to not only find the slot value in the utterance, but also the word that comes after it. As we can see, in the utterances, the value *ref* is not present in the utterances. However, every time a user requests this slot, is by referring the word **reference**. For this reason, we search for the word **ref** in the utterance, go until the end of the word **reference** and annotate it..

Given all the properties to annotate the information requested by the user in the utterances given, the code created for this purpose can be seen in Algorithm 2.

Algorithm 2 Annotate requested slots in a utterance

```
FindSlotsInUtter(Slots, Utter):
    for Slot in Slots:
        Slot_Value = Slot.split("-").end
        Utter = FindSlotWithSpaces(Slot, Utter)
    return Utter

FindSlotWithSpaces(Slot, Utter):
    modified_slot = Slot
    index = 0

    while index != len(slot):
        found = searchSlot(Slot, Utterance)
        if not found:
            modified_slot = slot[:index] + " " + slot[index:]
            index = index + 1
        else:
            Utter = replaceValueInUtterance(modified_slot, Utter)
        return Utter
```

4.2.2 Creating Rasa Stories

In Rasa, a story is composed by a set of the user intent and entities, followed by the action of the conversational agent. In the previous subsection, we discussed how to create Rasa intents and identify

the entities in each utterance from the information presented in the dataset. In this subsection, we proceed to annotate the assistant utterances by creating Rasa actions and Responses, and then use these to create Rasa stories.

Listing 6 Example of a designed story using our solution

```
- story: dialogue 182
  steps:
  - intent: find_restaurant
    entities:
    - restaurant-name: cafe uno
  - action: action_find_restaurant
  - intent: book_restaurant
    entities:
    - restaurant-bookday: sunday
    - restaurant-bookpeople: 7
  - action: utter_ask_restaurant_booktime
  - intent: inform
    entities:
    - restaurant-booktime: 16:15
  - action: action_book_restaurant
  - intent: goodbye
  - action: utter_say_goodbye
```

Take Listing 6 as an example. The story represented in this listing starts by the user stating its will to find a restaurant called *cafe uno*. The agent gives the information about the restaurant and the user proceeds to make a reservation, giving the information about the *day of booking* and also the *number of people*. As there is information missing, the agent asks for the time of booking. The user gives then the missing information and then the booking is completed by the assistant. The dialogue ends just as the user states its intention to terminate the conversation.

To create this story, we make two assumptions: first, we consider that for every Active Intent in the dataset, there is an action with the name *action_active_intent*. Each of these actions is responsible to fulfill the requests of the user and also make any verifications that are required. In Listing 6, there are two actions that derived from Active Intents, which are the actions **action_find_restaurant** and **action_book_restaurant**.

However, the assistant will not always have all the information required to fulfill a task. For example, to make a reservation for a restaurant, the assistant needs to know the day and time of the reservation, as well as the number of people. Although the required slots could be considered as a verification for the actions defined previously, this also means that for two conversations where the order in which the slots are given, the story representation would be the same. For this reason, we came up with a second assumption: in each of the tasks, if any of the requested slots are missing, then a Rasa response that asks for the slots is created. This process is repeated for each of the valid dialogues of MultiWOZ, creating the number of conversation paths equal to the number of valid dialogues on the dataset. We consider that a dialogue is valid if all the user utterances have been labeled correctly with Rasa intents (utterance labeling is explained in Section 4.2.1).

However, in order to realize what responses to create, there was the need to define what are the re-

quested slots for a task. First, we divided the dialogues in tasks. Next, we took two different approaches. In the first approach, we considered every slot that appeared at least once in a task as mandatory. This approach would fail, since this means that for finding a restaurant, the user would always need to state the restaurant *food*, which does not happen.

For our second approach, we decided to calculate the mandatory slots based on the following formula:

$$appearance\%(slot, active\_intent) = \frac{\#appearances(slot, active\_intent)}{\#dialogues(active\_intent)} * 100$$

On the right side, we start by calculating the number of times a slot appears for every dialogue with a specific active intent and divide that number by the total of dialogues for the same active intent. Thus, we compute the product of the index obtained by hundred, getting the percentage of active intent dialogues where the slot appears. We repeat this calculus for each slot of every Active Intent that it applies to. For example, the slot **restaurant_bookday** can appear in both of the tasks **find_restaurant** and **book_restaurant**, so we calculate the value for each one.

After calculating this value for every slot and every active intent, we compare this value with a threshold value. If the value of the appearances of the slot is above or equal the threshold value, then the slot is considered a mandatory slot for the task of the active intent. Since the value of the threshold is variable, then we adjusted the value to obtain the best results of slots, which was set to 85%. This means that only and only if a slot has 85% or more appearances in the active intent dialogues, then it is considered mandatory for that active intent. The results obtained for the Active Intents **find_train** and **book_train** are represented in Table 4.12, and the results for the other Active Intents are in Appendix A.

Active Intent	Slots	Mandatory Slots
find_train	train_day train_leaveby train_departure train_arriveby train_destination train_bookpeople	train_day train_departure train_destination
book_train	train_day train_leaveby train_departure train_arriveby train_destination train_bookpeople	train_day train_departure train_destination train_bookpeople

Table 4.12: Mandatory slots calculated for both find_train and book_train active intents with a threshold of 85%.

Given that we already determined what are the mandatory slots for each dialogue, the selection of the action for both a task-specific intent and the inform intent we defined in the previous section

for the story becomes simple. If the Rasa intent of the user is a task-specific intent, then we check if all mandatory slots have been given. If so, then we select the action of the assistant to be of the type **utter_ask_slot_name_other_slot_name**. Otherwise, the action selected is **action_task**. The same logic applies to the **inform** intent, since this intent is an extension of the task the user is trying to fulfill, meaning that the action selected is done based on the main task. For example, if the task is **book_hotel** then a possible action is **action_book_hotel**, but if the task is **find_taxi**, then the action can be **utter_ask_taxi_departure**.

We need to determine what are the actions for two other intents: **goodbye** and **request**. For the goodbye intent, we create a simple response in the form of **utter_say_goodbye**. For the request intent, we took a different approach. Since a request represents a question from the user to the assistant and the action to be performed by the assistant is always the same, then instead of making the request a part of stories, we use a Rasa **rule**. A Rasa rule defines the behaviour the agent must have when its properties are met and is good for when the action of the agent does not depend of the past conversation. Thus, we can specify that every time that the user asks for a request, the action performed is **action_get_requested_info**. The rule created for this intent is in Listing 7.

Listing 7 Rule created for the handling of the intent request.

```
- rule: give requested info
  steps:
  - intent: request
  - action: action_get_requested_information
```

Concluding, following the process described in this chapter, we were able to define the domain, the training data for the NLU, such as the utterance labeling, synonyms and regex and also the training data for the Dialogue Manager module by specifying the Rasa stories and rules. For last, in order to prepare the model to training and execution, we need to define the text to the created Responses, as well as the configurations and the implementation of the Custom Actions, which we describe in Section 6.1.

Chapter 5

Knowledge Transfer

In Chapter 4, we approach the MultiWOZ dataset and create a Rasa agent using only the information available. In this chapter, we continue the research on what is the important information from datasets using the Taskmaster-1 dataset to create a conversational agent in Rasa (RQ1). While creating the conversational agent using Taskmaster-1, we also address the possibility of using the conversational agent created from MultiWOZ to improve the dataset information (RQ2). Later in this chapter, we explore how can the knowledge learned for one domain can be used in another domain (RQ3).

5.1 Adapting Taskmaster-1 Dataset to Rasa

The Taskmaster-1 dataset, just as MultiWOZ, is a dataset that contains task-oriented dialogues. Just as we specified in Chapter 2, the dialogues were created either using a self-dialogue or a Wizard-Of-Oz approach. Moreover, each dialogue can belong to one of six domains: car repairing, restaurant reservation, ordering movie tickets, ordering coffee drinks, pizza delivery and taxi service. For a better comparison with the MultiWOZ dataset, we will focus on the restaurant reservation domain, that is a common domain to both datasets. Thus, examples and assumptions done on this point forward are only in regard to the restaurant domain.

Utterances	**Value**	**Annotation**
USER: Tell me good Chinese restaurants in new york	new york	restaurant_reservation.location.restaurant.accept
ASSISTANT: Hakkasan and uptown restaurant Philippe Chow are top rated	Hakkasan	restaurant_reservation.name.restaurant
	Hakkasan and uptown	restaurant_reservation.name.restaurant
	Philippe Chow	restaurant_reservation.name.restaurant
USER: which one is near to airport?	near the airport	restaurant_reservation.location.restaurant

Table 5.1: Taskmaster dialogue example for the restaurant domain and information available by utterance.

Take Table 5.1 for illustration purposes. In Taskmaster-1 dialogues, in each turn (user and assistant)

it is given the utterance, and information about the slots. The information given by a slot is its value, which is a part of the utterance, and the annotation of the slot. If we look at the annotation, we see that it can be broke down in different parts. The slot is composed by the domain, followed by the slot type, then the name of the slot and, optionally, ends with the acceptance, or not, of the parameter by the agent at the moment of the booking. For example, for the last utterance in Table 5.1, we consider that the domain, type and name are *restaurant_reservation*, *location* and *restaurant*, respectively. It is important to note that the slots annotated only refer to slots that are required to complete the task. For example, if the user says he/she is looking for a restaurant that has **italian**, or **asian** food, none of these two attributes are considered a slot as they are optional for the task of a reserving a restaurant.

From this information, there is information that we are able to parse. We can already create all the **entities** and **slots** in the Rasa domain. Also, we can annotate the slot values for each user utterances in Rasa annotation to use as training data. To make this annotation, we only need replace the slot value by the format *[slot_value](slot_name)*, since the slot values are always part of the utterance.

Aside from the slots, there is no more information in the dialogues. This is a problem, since we still need to identify the intents of the user. For that reason, after analysing the dialogues of Taskmaster-1, we noticed that the structure of the restaurant domain dialogues were similar to the MultiWOZ's restaurant domain, and could be broken in two parts: finding and booking a restaurant. Moreover, we also noticed that just as in MultiWOZ, there were also utterances where the user asks for information (Table 5.2 exemplifies the cases we mention).

#	Utterance
1	I'm looking for a nice sit down restaurant in San Francisco, California.
2	No, that'd be fine. Let's go with the second restaurant, please.
3	Okay, and is there good actions for kids?

Table 5.2: Examples of utterances where the user has different intentions. The first and the second are for finding and making a booking for a restaurant, respectively, and the last one is to get information about the restaurant.

Thus, to try to find this information, we took two approaches: the first one is by making assumptions by dividing the dialogue in parts and labeling the utterances, and the second is to use the the MultiWOZ's restaurant domain agent to understand some intents and extra entities from the utterances.

5.1.1 Creating Assumptions to Label Utterances

Since the information from the dataset is limited, assumptions on the dialogues can allow to retrieve more information. Thus, the assumptions we create are based on the presence of the slots in the dialogues. Moreover, we focus on the search of the restaurant, instead of the whole dialogue. The assumptions are the following:

- All the user utterances until the slot **location** is found are merged in one and labeled as **find_restaurant**.
- After finding the location, we label each of the following user messages as part of **inform** until the

first restaurant name is given by the assistant.

Intent	Utterances
find_restaurant	I'm looking for a nice sit down restaurant in [San Francisco, California.](location)
	I need to find places to eat in [San Francisco.](location) Want to book table.
	Good afternoon. I was hoping to place reservations for a sit-down Asian restaurant in [San Francisco](location). I was hoping for suggestions.
inform	Okay. I'm looking for a Greek restaurant with Greek food.
	Thank you.
	No problem.
	What is that one?
	Quiet, cheap, and well rated.

Table 5.3: Utterances annotated as find_restaurant and inform, created using assumptions and the information from the slots in the dialogues.

Some examples of the utterance labeling with each assumption are in Table 5.3. In the *find_restaurant* results, we can notice the intention of the user of finding, and later booking, a restaurant. However, the same can not be said for the *inform* utterances. In the results of the last assumption, although we got some of the utterances add information to the dialogue, such as the first and the last utterances, the rest of the utterances do not give an additional information to finding a restaurant.

With the results of both assumptions, we can conclude that by giving only the slots, we are not able to create the intents of the Rasa agent without having noise in the training data. Thus, with the Taskmaster-1 dataset, a simple approach of using information in the dataset is not sufficient to create a Rasa agent.

5.1.2 Using MultiWOZ to Add Information to Taskmaster-1

In this subsection, we use the MultiWOZ restaurant agent and try to add new information to the Taskmaster-1 dataset, such as the annotation of optional slots, that are important to find the best restaurant for the user and that are not given in the Taskmaster-1 dataset. Moreover, we also try to differentiate the utterances labeled as **inform** by the second assumption we described earlier.

To get the information from the MultiWOZ dataset, we apply the agent to each utterance labeled as **inform**, and retrieve the information. The information we can collect is the most probable intent (as well as the confidence of the rest of the intents) and the entities predicted by the agent. In Table 5.4, we represent some of the results obtained.

Looking at the examples, in the first utterance, we expected to extract the **Thai** entity as a **restaurant_food**, but the agent does not extract any entity. On the second example, although there are no entities, the utterance is not recognized as a **request**, but as an inform. For the last example, the agent is actually able to extract the correct entity, but classifies it as a goodbye, when it is a **request**.

Analysing the results of this experiment, we realised that the MultiWOZ restaurant agent is insufficient

Utterance	Intent	Entities
I was thinking Thai food.	inform	-
Okay, is it is it cheap?	inform	-
Okay, what's the price like on that?	goodbye	requested_info: price

Table 5.4: Example of classification of the user utterances by the MultiWOZ's restaurant agent.

to improve the annotation of the Taskmaster-1. This can be for two reasons: one possibility is that the utterances used in the training for the NLU are not alike the utterances in the Taskmaster-1 dataset, another possibility is that Taskmaster-1 has more diversity on the questions that can be asked. For example if we look at the sentence below

Does it have kids menu?

We see that this example is completely different from the examples used in the MultiWOZ agent, since in the first place there is no slot representing a kids menu and therefore there will not be a correct classification from the agent for this utterance. Besides these examples, it is sure that utterances like *"no problem"* can not be classified correctly. This is because if we recall the design of our MultiWOZ agent, any sentences that do not add any new information are not considered for the training of the agent which lead to the lack of existence of this type of examples in the agent.

From this experiment, we can conclude that the knowledge acquired from the MultiWOZ restaurant agent is insufficient to add information to Taskmaster-1. However, we argue that if the training data of the agent was similar to the Taskmaster-1, then it is possible to add new information.

5.2 Creating New Agent Using Knowledge From the Restaurant Domain

In this section, we proceed to create a Rasa agent for the hotel domain, using the agent previously created for the restaurant domain. With this experiment, we expect to understand if it is possible to transfer the knowledge of an agent that we defined previously to another agent, in a different domain. This possibility would allow the improvement of development process, mainly the development time.

For the target domain, we believe that the tasks of the target domain must be compatible with the original domain. For that reason, we chose the hotel domain as the target, since the number of tasks is the same. Moreover, each task has the same number of mandatory slots.

Just as we stated before, to define a Rasa agent we need to define the **domain**, the **NLU training data** and the **stories**. In this transfer, we maintain the **intents**, **responses** and **actions** in the domain, and the **stories** and **rules** we defined. Also, for the **entities** and **slots**, we create a mapping from the restaurant slots to the hotel slots. The mappings chosen are represented in Table 5.5. Moreover, we maintain the **required_info** entity.

For the mapping of the slots, there are two things that were considered: the mandatory slots, and the

Restaurant	Hotel
pricerange	pricerange
area	area
bookday	bookday
bookpeople	bookpeople
food	type
name	name
booktime	bookstay

Table 5.5: Mapping of slots from the restaurant domain to the hotel domain

slots required for each intent (*find_restaurant* and *book_restaurant*). The mandatory slots for the hotel domain were calculated using the same method as described in Section 4.2.2. The mandatory slots determined for the hotel agent are **hotel-bookday**, **hotel-bookstay** and **hotel-bookpeople**.

However, there are two slots from the hotel domain that are not considered from the hotel domain: the **internet** and **parking** slots. This happens for two reasons: first, we want the number of slots to stay as close to the original agent as possible. Moreover, both of these slots are not correctly labeled using the same method we used for the other slots, which means they would not be correctly predicted by the agent.

Also, there is one other slot we do not consider, which is the **stars** slot. Although this slot is important for the agent, there was no direct correspondent between the restaurant domain and the hotel domain. Moreover, the lack of this slot is not mandatory to the functioning of the agent, but this means the user is not allowed to tell how many stars the hotel he wants must have.

Apart from the entities and slots, we also change the NLU training data. To obtain the training data, we use the approach described in Section 4.2. Moreover, we also adapt the Rasa stories from the restaurant agent. For each story we make the mapping of the slots, and select a random value it can take from the annotated utterances in the training data. An example of the result of this process can be seen in Listing 8. Moreover, we also present the original story in Appendix B.

Using this process, we guarantee that the hotel agent will be able to recognize utterances with information regarding the hotel domain, and are able to check if the conversation paths defined by the stories in the restaurant agent can be used for the hotel domain. Concluding, now that the domain, as well as the training examples for the NLU and Dialogue Management components have been transferred to the hotel domain, we can now end the agent by specifying the text for the Responses, implementing the Custom Actions and choosing the agent configuration, which is done as described in Section 6.1.

Listing 8 Result of the mapping of the restaurant agent to the hotel domain.

```
- story: dialogue 11
steps:
- intent: find_hotel
  entities:
  - hotel-area: north
  - hotel-pricerange: moderately
- action: action_find_hotel
- intent: inform
  entities:
  - hotel-stars: 1
- action: action_find_hotel
- intent: book_hotel
  entities:
  - hotel-bookday: friday
  - hotel-bookpeople: 8
  - hotel-bookstay: 3
- action: action_book_hotel
- intent: goodbye
- action: utter_say_goodbye
```

Chapter 6

Evaluation

In this chapter, we describe how we complete the implementation of the restaurant and hotel agents. Then, we explain the evaluation we perform to both agents, the results of the evaluation and how does it support our objectives. Prior to this evaluation, we perform a preliminary test with 4 people, which allowed us to find issues regarding the interaction with the agents that could be fixed to perform the best evaluation possible.

Response	**Message**
utter_say_goodbye	Thank you! Goodbye.
utter_ask_restaurant_name	Can you tell me the name of the restaurant, please?
utter_ask_restaurant_name	Can you tell me the name of the restaurant, please?
utter_ask_restaurant_bookpeople_restaurant_booktime_restaurant_name	Please tell me the restaurant name, the number of people, and the time of the booking.
utter_ask_restaurant_bookday	For what day is the reservation?
utter_ask_restaurant_booktime_restaurant_name	Tell me the name and the time of the reservation, please.
utter_ask_restaurant_booktime	Can you tell me the time of the booking?
utter_ask_restaurant_bookpeople_restaurant_booktime	Tell me the day and the number of people for the booking, please.
utter_ask_restaurant_bookpeople_restaurant_name	Can you tell me the name of the restaurant and the number of people?
utter_ask_restaurant_bookday_restaurant_booktime	What is the day and time of the booking?
utter_ask_restaurant_bookday_restaurant_booktime_restaurant_name	Please tell me the name of the restaurant, and the day and time of the reservation.
utter_ask_restaurant_bookpeople	How many people is the booking for?
utter_ask_restaurant_bookday_restaurant_name	What is the name of the restaurant, and the day of the booking?

Table 6.1: Messages chosen for each response in the restaurant agent

6.1 Agent Implementation

Even though we were able to define the structure of the Rasa agents, we still need to define what is the behaviour of the agents, mainly what are the messages of the responses and the behaviour of the actions. Moreover, we also define the pipeline in which we train the NLU module.

6.1.1 Responses

As we described in Chapter 2, responses are messages that the agent sends to the user, that are composed by mainly text. In Chapter 4, we defined what are the Responses for the agents, so for each of the defined responses, we need to choose the correspondent message. In Table 6.1, we illustrate the messages chosen for the restaurant agent, and in Appendix C we show the responses for the hotel agent.

6.1.2 Custom actions

Although both agents have the NLU module complete and all the stories defined, we still need to define what is the behaviour of the actions of the agents. Thus, for each agent, we implemented the logic of two actions. For the restaurant agent, we define the actions **action_find_restaurant** and **action_book_restaurant**, whereas for the hotel agent we define **action_find_hotel** and **action_book_hotel**. Note that since both agents' actions have the same functionality, the logic implemented is similar

Action_find_restaurant and Action_find_hotel

To find a restaurant or a place to stay, we split the functionality in two parts:

- If the user gives the name of the restaurant/place, then search for the restaurant and give information about the same.
- If the user intents to find a restaurant/place, use the criteria to find a suitable one.

That said, there are some details for each of the parts. For the first part, even if the user has given the name of the restaurant/place, if there is new criteria such as the price range, then we proceed to the second part instead of giving information about the restaurant/place given. This property prevents the user from being stuck in the interaction if it has given the name of the restaurant previously and wants to find another one.

Furthermore, when displaying the information about the restaurant/place the user referenced, the agent also displays a message asking if the user wants to make a booking, as shown below:

If you wish to make a reservation, please state the number of people, the time of booking/days of your stay or the day of the week/check-in.

We decided to structure the sentence this way instead of a sentence like “Do you want to make a booking?” to guide better the user during the conversation and avoid errors. This is because our agents do not recognize the meaning of yes/no which represent an affirmation or a negation.

Moving now to the second part of the functionality, there are several details we consider. If no criteria has been given during the conversation, then the action responds with the following sentence:

Could you please tell me the price range, type of food/lodging or area of town of the restaurant/place to stay?

Then, after checking the existence of criteria, we proceed to find any restaurants/places whose properties match with the given criteria. In here, there are 3 possibilities:

- If there is only one restaurant/place, we use the same logic as we would when given the name of the restaurant/place by the user
- If there are two or three places, we show the user the eligible restaurants/places and ask him/her to select one.
- If there are four or more restaurants/places, we select randomly three restaurants/places, and show the user for him/her to select one

Given these 3 possibilities, one example sentence given by the agent is the following:

the missing sock, yu garden, backstreet bistro match your criteria. Can you please select one?

For last, there is a final property. If the user has not given all possible criteria (for example, give price range and type of restaurant/lodging but not the area of town), the agent also displays a message that the user can give more criteria. An example message can be:

Optionally, you can give the city area so we can narrow down the search.

Action_book_restaurant and Action_book_hotel

For the booking, we start by verifying the value of all slots required. For example, for the restaurant, we start by checking if the restaurant name has been given by the user, and also check if it exists. In case any of these properties fail, then a message is sent for the user to tell the restaurant name it is trying to make a booking, just as shown below:

I can not find the restaurant/place you told me. Can you please tell me the name of the restaurant/place again?

After the check on the restaurant name, we proceed to verify if all the slots required are set, such as the day of booking, time of booking and the number of people for the restaurant. In case any of these slots, the agent ask the agent for the missing slots:

Please tell me the number of nights in order to complete the booking.

After checking all the slots, we then proceed to make the booking. There are two possibilities. First, if no booking was done during the conversation with the user, then a booking is done. However, if a booking was already done, then we proceed to change the properties of the booking. This way, we are able to extend the functionalities of the agent, allowing the user to modify its booking in case of a mistake by either the user or the agent. In either case, the user is reported of the situation with a sentence specifying the details of the reservation. The message given by the agent to the user is, for example:

Your booking for autumn house for 3 days starting monday for 2 people has been done successfully. Can I help you with anything else?

Action_get_requested_info

For both of the agents, we proceed with the same logic. First, we check if the name of the restaurant or place has been given during the conversation. If the user did not give the name, then we ask the user to give the name of the place he/she is searching information for. However, if a name has been given, we then proceed to find information about the restaurant/place. After retrieving the information the user asked about, then a message is sent to the user with the same. For example, if the user asks for the number of stars of a place, the agent responds:

The stars of finches bed and breakfast is 4.

6.1.3 NLU pipeline

In Rasa, the NLU is defined by a set of components. To get the output of the NLU module, the user message is processed by each of the components. The details of this process can be seen in Chapter 2. To train the multi domain agent, the restaurant agent and the hotel agent, we used the configuration listed in Listing 9.

6.2 Objectives

There are two objectives regarding this evaluation:

- Prove the quality of the framework developed in order to expedite the creation of Rasa agents.
- Check if it is possible to transfer knowledge between domains.

For the first objective, we would like to evaluate the multi domain agent from MultiWOZ. This agent, however, although having good results regarding the NLU module, presents several mistakes of entity recognition between different domains. Moreover, a preliminary test on this agent has shown that it was difficult for the agent to predict the correct entity of a user utterance. The details of this evaluation can be seen in Subsection 6.5.1. For this reason, instead of using the multi domain agent, we perform an evaluation on a restaurant agent created with only domain-specific dialogues.

Listing 9 Components used to create the NLU for all the agents

```
pipeline:
  - name: WhitespaceTokenizer
    "intent_tokenization_flag": True
    "intent_split_symbol": "+"
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
    lowercase: True
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 80
    constrain_similarities: true
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 80
    constrain_similarities: true
  - name: FallbackClassifier
    threshold: 0.3
    ambiguity_threshold: 0.1
```

For the second objective, we perform an evaluation on the hotel agent, which is created by transferring knowledge from the restaurant agent, as explained in Section 5.2.

For each agent, a two-part evaluation was designed, which is an NLU evaluation, followed by a human evaluation. The NLU evaluation is an automatic evaluation provided by Rasa, which allows to collect metrics such as Precision, Recall and F1-Score for both the intents and the entities of the agents. These metrics are calculated as a Micro, Macro, and Weighted average, as well as for each intent/entity. The NLU evaluation, as the name suggests, is solely for evaluating the NLU module. Also, with this evaluation, we are able to understand if the user can be understood correctly while interacting with the agent. Moreover, the understanding of any issues with this module allows us to create better tasks for the human evaluation, so that the user can focus on completing the tasks, instead of trying to be understood. Also, it allows us to we have a better understanding of what is happening with the agent during the interaction.

The human evaluation is composed by two steps. The first one is the interaction with the agent, followed by the fill of the questionnaire. During the interaction, some metrics such as the NLU errors, action choice errors, and number of turns required to complete the task are noted. In the form, we collect information about the user experience during the interaction with the agent.

Thus, both the performance of the agent during the interaction as well as the results from the questionnaire will allow us to determine if the interaction with the agent was good, and take conclusions as if the objectives of this evaluation were reached.

6.3 Materials

There are several materials required for the evaluation of the agents. For the NLU evaluation, we require labeled utterances with both the intents and entities for each. Moreover, for the human evaluation, there are several materials we need. We specify different tasks for the user to perform and for also create a questionnaire for the user to fill after the interaction with the agent's agents.

6.3.1 NLU Utterances

For both agents, we create the test utterances using the methodology described in Section 4.2, using the dialogues in the MultiWOZ test set. This allows to have different type of utterances created using the MultiWOZ approach, which makes an accurate evaluation since the utterances were created by humans. Moreover, this way, we are also able to have the utterances annotated with both the intents and the entities. The dialogues from which we select the utterances are domain-specific. This means that only dialogues from the restaurant domain are used to evaluate the restaurant agent. Moreover, the same happens for the hotel agent.

6.3.2 Tasks

Restaurant	
Task #	Sentence Used
1	Find and make a booking for an **italian** restaurant in the **north** of town. The booking is for next **tuesday** at **20:00** for **2** people.
2	Ask what is the **phone number** of an **expensive** restaurant.
3	Make a reservation for **"city stop"** restaurant. The reservation must be for **3** people at **12:00** for next **sunday**. Also, change the booking to **4** people.
Hotel	
Task #	Sentence Used
1	Make a booking for 5 nights starting **monday** for a moderately priced place in the **south**. The booking is for **4** people.
2	Find a **cheap guesthouse** in the **west** and check if it has **internet**, as well as how many **stars**.
3	Make a reservation for the **"autumn house"** for **2** people starting next **sunday** for 1 night. Change the reservation to 2 nights, starting **friday**.

Table 6.2: Tasks presented to the user during the interaction with each of the agents.

For both agents, we create three different tasks for the user to perform. Each of the tasks has a different objective and tests a different functionality of the agent. The functionalities we test are:

1. Ability to find a restaurant/place to stay and make a booking.

2. Get information from a restaurant/place to stay.

3. Change some details of a booking.

Taken these objectives, we then select some properties for the entities of the agent and describe the task as a single sentence for the user. The tasks used for the tests are in Table 6.2.

6.3.3 Questionnaire

In order to understand what the users felt about the agent while they were interacting with it, we created a questionnaire with with three different parts. This questionnaire is performed after the interaction with the conversational agent. The first part of the questionnaire is composed by questions to get the general information of the users, such as the age, fluency in English and familiarity with conversational agents. For the second part, we focus on measuring the user experience, and the third part will measure the quality of the different parts of the system and the system as a whole.

To measure the user experience, we will use the questionnaire called UEQ (Schrepp, Hinderks, and Thomaschewski 2017). The questionnaire was created with the purpose of evaluating the user experience and to do so, it uses 26 items to measure 6 different scales, each answering a different question. The scales to be measured are the following:

- Attractiveness - Describes the likeability of the user towards the product
- Perspicuity - Do users learn the product with ease?
- Efficiency - How much effort does the user need to put in order to solve the tasks?
- Dependability - Is the user in control of the interaction?
- Stimulation - Does the user feel excited or motivated to use the product?
- Novelty - Is the product creative and does it catch the interest of the user?

Since the UEQ was designed to be able to be used in every product, some of these might not make sense in the context of our Rasa agents. For this reason, we can select only the scales that make sense to evaluate them. In our case, we consider the scales of Attractiveness, Perspicuity, Dependability, and Stimulation.

Furthermore, while Attractiveness which as 6 items, each scale as 4 items. Each item has a linear scale from 1-7 and each side contains a term, both having opposite meaning, for example Attractive-Unattractive. The full UEQ can be seen in Appendix C.

For the last part, we want to focus on the capabilities of the agent itself. The last part of the questionnaire is in Appendix C. To create the questionnaire, we created several statements, where the user will indicate how much he/she agrees with the statement on a 1-7 scale, where 1 represents Strongly Disagrees and 7 represent Strongly Agree. Moreover, we also categorize the statements in four different scales, each focusing on a different part of the agent. The scales we created are the following:

- **Understanding Capabilities** - Can the conversational agent understand what the user is saying?

- **Experience** - Did the users feel successful in the engagements with the agent?
- **Flow** - Did the dialogue to achieve the task feel natural?
- **Limitations** - Does the user feel the limitations of the agent during the engagement?

6.4 Procedure

The NLU evaluation was done prior to the human evaluation, since it is an automatic evaluation. After the NLU evaluation, we then proceeded for the human evaluation. To make this evaluation, we asked 14 different people to interact with both our agents. The human evaluation were made both in person and via Zoom.

Prior to starting the evaluation, the user is explained how the test will occur, and it is told whether he/she will interact first with the restaurant agent or the hotel agent. We made sure the starting agent was different for half of the users to minimize the learning effect of interacting with the agents. Moreover, during the interaction with each agent, we give the tasks for the user to complete. After interacting with an agent, the user then proceeds to fill the correspondent questionnaire.

6.5 Automatic NLU Evaluation

6.5.1 Multi Domain agent

In this section, we discuss the results of the agent containing data from all the domains in MultiWOZ. In this test, we analyse 6259 utterances and 9550 entities present among the utterances. In Table 6.3, we show the results of the Precision, Recall and F1-Score for the Micro, Macro and Weighted averages.

	Intents				Entities		
	Micro Avg	**Macro Avg**	**Weighted Avg**		**Micro Avg**	**Macro Avg**	**Weighted Avg**
Precision	0.9060	0.8660	0.9061	**Precision**	0.9125	0.8961	0.9138
Recall	0.9057	0.8668	0.9057	**Recall**	0.9541	0.9186	0.9541
F1-Score	0.9058	0.8661	0.9055	**F1-Score**	0.9328	0.9053	0.9328

Table 6.3: Results of the Precision, Recall and F1-Score, for the Micro, Macro and Weighted averages for both entities and intents.

Moreover, in appendix D, we also show the values of the Precision, Recall and F1-Score for each individual intent and entity. Aside from these metrics, Rasa test also gives the number of intents and entities that have been mistaken by another. For example, during the test, **hotel-pricerange** have been mistaken for **restaurant-pricerange** 9 times. In total, during the test, 215 (2.25%) entities have been mistaken by another entity and the same happened for 328 (5.24%) of the intents.

Although expected, the incorrect identification of entities is a concern for us, since in a conversation, the agent might think the user is referring to another domain than the one he really is. For this reason,

instead of proceeding to the user evaluation with the users, we made a pilot test of this agent. We quickly realised that even though the results from the NLU said only 2.25% of the entities were mistaken, this would be worse when interacting with the agent. This problem was mainly noticed in the **inform** intent, for two reasons: 1) two utterances with different entities show a high degree of similarity and 2) the context of the domain is not present in the decision of the entities. An example of two dialogues regarding this problem is represented in Table 6.4. In the first dialogue, even though the conversation is in the context of the restaurant domain, the values the **hotel-location** and **restaurant-location** slots can take is the same, which leads to an incorrect labeling of the *"north"*.

Dialogue A		
Turns		
U: Hey, I'm looking for a cheap restaurant that serves Italian food.	**Entity Value**	north
A: Sure, can you tell me what is your preferred location	**Expected Entity**	restaurant-location
U: I would prefer in the **north** of town.	**Labeled Entity**	hotel-location
Dialogue B		
Turns		
U: Hey, I'm looking for a hotel that is of moderate price.	**Entity Value**	north
A: Sure, do you have a preference on the location?	**Expected Entity**	hotel-location
U: I want it to be in the **north**.	**Labeled Entity**	hotel-location

Table 6.4: Two dialogues where the last utterance of the user is an **Inform** and the entity is labeled as the same, even though they refer to different domains.

6.5.2 Restaurant agent

For the NLU evaluation of this agent, we tested 242 labeled utterances with 370 entities. The results of the accuracy for the intents, as well as the precision, recall and f1-score for both the intents and the entities are in Table 6.5.

Intents				Entities			
Accuracy		**Macro Avg**	**Weighted Avg**		**Micro Avg**	**Macro Avg**	**Weighted Avg**
0.8802	**Precision**	0.8649	0.8836	**Precision**	0.9377	0.9376	0.9381
	Recall	0.8876	0.8802	**Recall**	0.9351	0.9335	0.9351
	F1-Score	0.8718	0.8773	**F1-Score**	0.9364	0.9328	0.9340

Table 6.5: Results of the NLU Rasa test for the restaurant agent.

Also, during the NLU evaluation, the slots **restaurant-area** and **restaurant-name** have been confused with the slot **restaurant-food** 2 and 1 time, respectively. Moreover, during this test, there was 25 (10.33%) mistakes between intents.

6.5.3 Hotel agent

For last, for the hotel agent, we tested a total of 290 utterances and 333 entities. The results of this evaluation is in Table 6.6.

Intents				Entities			
Accuracy		**Macro Avg**	**Weighted Avg**		**Micro Avg**	**Macro Avg**	**Weighted Avg**
0.8690	**Precision**	0.8571	0.8706	**Precision**	0.8099	0.8030	0.8244
	Recall	0.8599	0.8690	**Recall**	0.8829	0.9035	0.8829
	F1-Score	0.8552	0.8672	**F1-Score**	0.8448	0.8422	0.8495

Table 6.6: Results of the automatic evaluation for the hotel domain agent

During this test, there were also several mistakes. There were 22 (6.60%) mistakes in entities and 32 (11.03%) mistakes in the intents. For the entities, the mistakes were between the slots **hotel-bookpeople** and **hotel-bookstay** which is expected, since both are numbers.

6.6 Human Evaluation

Our sample was composed of 14 participants. From those, 7 (50%) people started by interacting with the restaurant agent, and the remaining 7 (50%) people started by interacting with the hotel agent. The age of the participants ranged from 16 to 28 years old, where 9 (64.29%) participants identified as male and 5 (35.71%) people identified as female. Moreover, the testers evaluated their fluency with a mean of 5.76 of 7 and the overall experience with these type of agents were 3.71 out of 7. In this section, for visualization purposes, we present the results of the questionnaires on a scale of -3 to 3, instead of a scale from 1 to 7.

6.6.1 Restaurant Agent Results

During the interaction with the restaurant agent, the users interacted for a total of 279 turns, where the agent had 57 (20.43%) turns with NLU errors and 22 (7.89%) turns where the action selected was not the most appropriate. To complete the first task, the users did an average of 8.21 turns, with 2.38 turns with understanding errors from the NLU and in 0.62 turns the action selected was not correct. For the second task, an average of 4.07 turns were required to complete the task, 0.84 turns of those were understanding errors. Only one tester got an action selection error during this task. Finally, in the third task, the users required an average of 7.68 turns to complete it, where there were 1.15 turns with understanding errors and 1.07 turns where the action selected was not appropriate.

By comparing our the values obtained for each scale with the benchmark values, the results suggest that the users liked the agent (Attractiveness), and felt in control of the interaction (Dependability). Moreover, although to a less degree, it also suggests that it is easy to learn how to interact with the agent (Perspicuity), and that is motivating to use the agent. On a scale of -3 to 3, the mean and variance for

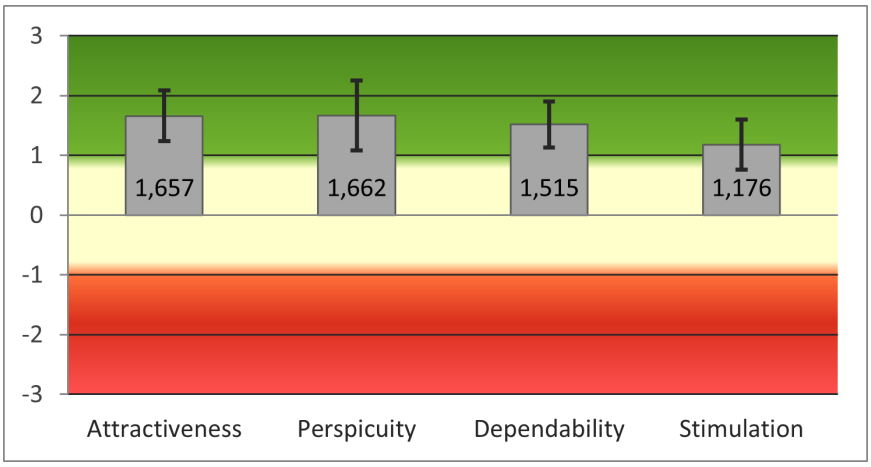

Figure 6.1: Means for each scale in the results of the UEQ in the restaurant agent

each of the scales are the following: 1.657 mean and 0.79 variance to Attractiveness (d=0.891, p=0.05, CI [1.233, 2.080]), 1.662 mean and 1.51 variance to Perspicuity (d=1.228, p=0.05, CI [1.078, 2.245]), 1.515 mean and 0.67 variance to Dependability (d=0.817, p=0.05, CI [1.126, 1.903]), and 1.176 mean and 0.76 variance for Stimulation (d=0.874, p=0.05, CI [0.761, 1.592]). These results are illustrated in Figure 6.1. We also present the mean, variance and confidence intervals for each item with p=0.05 in Appendix E.

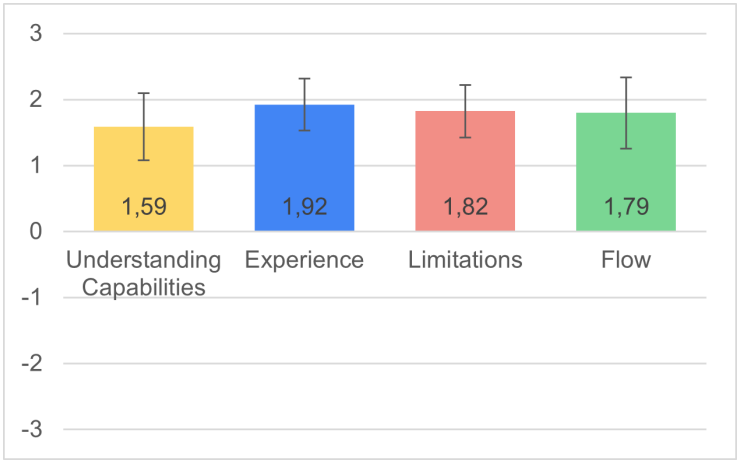

Figure 6.2: Means of each scale of the results of the third part of the questionnaire for the restaurant agent

For the last part of the questionnaire, the answers of the users suggest that the agent could understand the users, the users corresponded the expectations and they felt the flow of the dialogue was good. As illustrated in Figure 6.2, the mean and variance for the Understanding Capabilities of the agent are 1.59 and 1.15 (d=1.07, p=0.05, CI [1.079, 2.096]) respectively. For the Experience scale, the mean is 1.92 with a variance of 0.69 (d=0.829, p=0.05, CI [1.527, 2.316]). The Limitations scale has a mean of 1.82 and a variance of 0.70 (d=0.834, p=0.05, CI [1.427, 2.220]). For last, the mean and variance for the Flow scale are 1.79 and 1.27, respectively (d=1.126, p=0.05, CI [1.259, 2.329]). The metrics for each of the items in this questionnaire is in Appendix E.

Finally, regarding the opinion of the users, 7 (50%) of them mentioned interpretation errors and 1 (7.14%) mentioned being impacted by the action choice. Moreover, the users said that overall, the conversation was fluid to complete the tasks, and praised the capabilities of the agent.

6.6.2 Hotel agent Results

During the interaction with the agent, there was a total of 331 turns, where 107 (32.33%) turns had understanding issues from the NLU and there were 10 (3.02%) turns where the action selected was not correct. In the first task, there was a mean of 8.21 turns, having 2 turns with NLU errors and 0.3 turns with a action selection errors. In the second task, the number of turns required to terminate the task were 8.07, and there were 3.69 turns with NLU errors. In this task, the action selected was always the best one. For last, in the third task, the users required 7.35 turns to finish it, where 2.54 turns had NLU errors and 0.46 turns where the action selected was not the correct one.

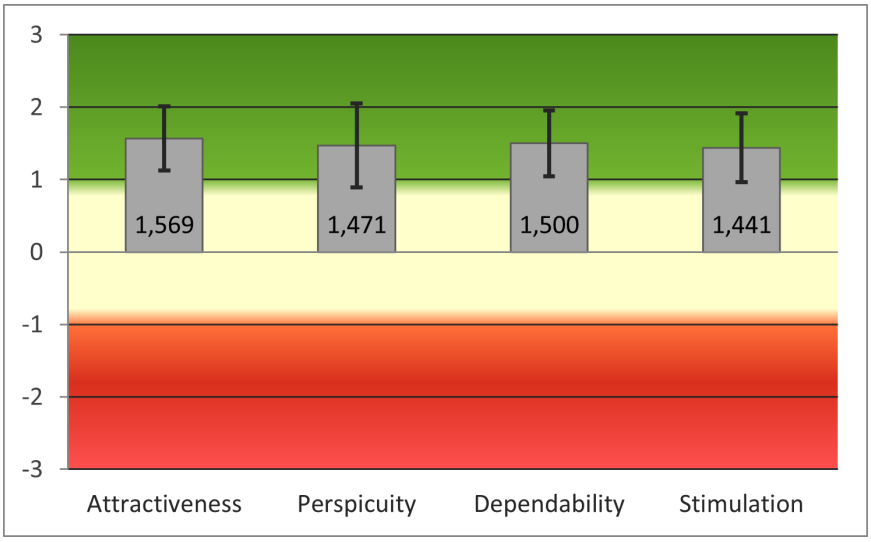

Figure 6.3: Means for each scale in the results of the UEQ in the hotel agent

Making a comparison of the results of the UEQ with the benchmark, it shows that there the agent is a good overall product (Attractiveness) where the users felt motivated during the dialogue (Stimulation) and in control of the conversation (Dependability). Although to a lesser degree, the users felt that learning to talk with the agent was easy (Perspicuity). In Figure 6.3, we illustrate the mean for each of the scales. For the Attractiveness scale, the mean is 1.569 and a variance of 0.86 (d=0.930, p=0.05, CI [1.127, 2.011]). The Perspicuity scale has a mean of 1.471 and a variance of 1.48 (d=1.215, p=0.05, CI [0.893, 2.048]). The mean and variance for the the Dependability scale is 1.5 and 0.91 respectively (d=0.952, p=0.05, CI [1,047, 1.953]) and for last, the Stimulation scale has a mean of 1.441 and a variance of 1.00 (d=0.998, p=0.05, CI [0.967, 1.916]). For also present the mean, variance and confidence intervals with p=0.05 in Appendix E for every item in the scales.

The results of the third part of the questionnaire show that although the agent could understand natural language, it was somewhat difficult for the users to express themselves in the way they wanted, which resulted in rephrasing the way they talked. However, apart from these problems, the results

suggest that the users had help from the agents and where able to conclude the tasks with some effort. That said, we present in Figure 6.4 the means of the questionnaire. The Understanding Capabilities scale had a mean of 1.49 and variance of 0.9 (d=0.951, p=0.05, CI [1.038, 1.942]), with the Experience Scale of a mean of 1.16 and a variance of 0.88 (d=0.936, p=0.05, CI [0.712, 1.602]). Moreover, we got a mean and variance of 1.57 and 1.11, respectively, for the Limitations scale (d=1.053, p=0.05, CI [1.068, 2.069]). For last, the Flow scale has a mean of 1.26 and a variance of 1.12 (d=1.059, p=0.05, CI [0.761, 1.768]). The results for each of the items is in E.

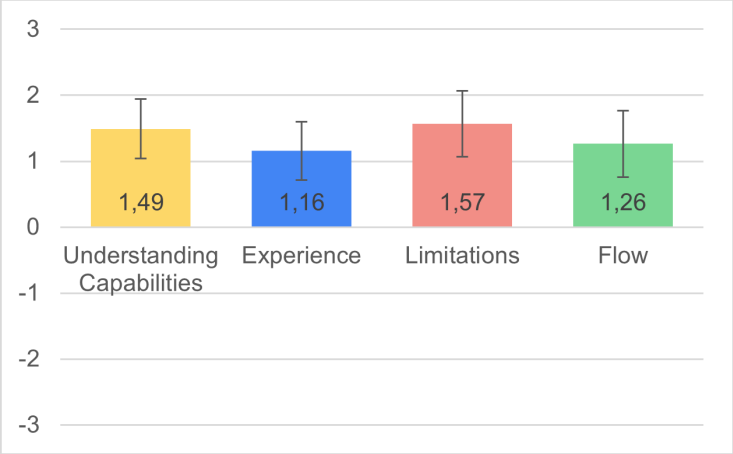

Figure 6.4: Means of each scale of the results of the third part of the questionnaire for the hotel agent

Regarding the opinion of the users, 8 (57.4%) of the users mentioned the errors of the NLU, but praised the agent for being able to help conclude the tasks. Moreover, the users mentioned that they could talk like they talk with another person.

Chapter 7

Discussion

In this chapter, we address the problems we faced creating the conversational agents, discuss the results of our evaluation and see how does it affect the objectives of this work.

7.1 Parsing Data From Datasets

To populate dialogue systems that are divided in the three modules (NLU, dialogue manager and NLG), the correct annotation of the dialogues is extremely important to create these type of systems (RQ1). This could be seen by the analysis of the two datasets (MultiWOZ and Taskmaster-1), that are two datasets with dialogues where the annotations of both differ. In Taskmaster-1, where the annotated data are only the slots at each turn, we were only able to label the entities for the Rasa agent. However, for the MultiWOZ dataset, we were able to define the NLU and dialogue manager modules of Rasa.

7.1.1 Natural Language Understanding

The Active Intents and Slots of MultiWOZ allow the labeling of a great part of the utterances as a part of an Intent such as the find_restaurant intent or the inform intent. However, this was not possible to do to utterances such as Affirmations, Refusals or Recommendations that are part of a dialogue. The absence of such intents blocks the creation of a more complete NLU module, and more consequently, the conversational agent. This could be seen during the definition of the custom actions for the restaurant/hotel agents, where the agent message was designed in order to guide the user to not write utterances that would affirm or negate, but would give more information, thus not creating understanding errors from the agent.

To address this problem, one possibility is to create in the dataset an Intent annotation. With this annotation, each of the user messages would have an Intent associated, which would allow the developers to differentiate all the utterances required for training data in the NLU module.

Besides, although the majority of the slot values in the dataset could be found in the utterances, there are slots that were not considered for the agent, such as the hotel wifi because its state value was not found in the utterance. The lack of this slot hindered the multi domain agent, since it the NLU module did

not have training data regarding this slot, diminishing the functionalities of the agent. Moreover, for the slots where the state value was found, the search for the slots could get complicated if the state value was only partly equal to the value in the utterance.

One possible solution for this problem is by storing two values during the annotation of slots in the dataset: the state value and the utterance value. The state value is the value that the agent recognizes and uses to perform it's actions. The utterance value is a transcript of the part of the utterance that mentions the correspondent slot. This would simplify the search of the value in the utterances and make possible the mapping of the utterance value to the state value, if both differ.

From the results of both evaluations, we can conclude that using the information from the dataset, it is possible to create an agent that is able to understand natural language. This is supported not only by the results of the automatic evaluation, but also by the third part of the questionnaires filled by the users.

However, the agent is by no means perfect. This can be seen by the results of the interaction, where there were 20.43% of the turns in the restaurant agent and 32.33% of the turns in the hotel agent had understanding errors. Furthermore, the results of the automatic evaluation also suggest that are improvements to be made. While the turn percentage with NLU errors of the restaurant agent is in concordance with the results of the automatic evaluation, there is a disparity of the two values in the hotel agent. There are two possibilites for this happening. First, although unlikely, since we performed the same process to create the training data for the restaurant agent, the utterances from the users in the test differ from the ones used in the training data. Another possibility is the fact that Rasa is not able to distinct correctly between the slots **hotel-bookstay** and **hotel-bookpeople**, since both use a number regex. To solve this last problem, we could create a single regex, and change the training data to be able to specify the type of number, based on the context. For example, in the sentence:

Please book it for 3 nights in a room for 2 people

We can identity the **3** as {*"entity": "number", "role": "bookstay"*} and **2** as {*"entity": "number", "role": "bookpeople"*}, where the NLU module would then learn using this training data what is the correct role of the number given.

7.1.2 Dialogue Manager

By looking at the results of the human evaluation, we can see that we were able to create a good base for the dialogue paths. This is supported by the low percentage of turns where the action selected was not the correct one (7.89% of all turns in the restaurant agent). Moreover, the users stated in the questionnaire that the agent helped to conclude the tasks and knew what to say to the agent most of the times. This means that using the information at a specific turn of the dialogue, more specifically the Intents, Slots and also the mandatory slots that we computed, it is possible to create the required actions, as well as the stories to define the dialogue manager module.

Just as we mentioned, only 7.89% of the turns had errors while choosing the action to take. We consider that the main reason for this issue is the lack of stories due to the presence of some errors in the dialogues of MultiWOZ. This would lead often to missing Intents on some utterances, which often led

to the elimination of the dialogues. One other possibility is the existence of too many dialogues, where two stories that are equal until a certain turn t, take different actions at turn $t+1$. To bridge this problem, we could perform a filter on the dialogues, eliminating any redundancies that might happen during the training of the dialogue manager module. For last, we consider that creating an annotation for the agent actions in the datasets contain the name of the action could also help in the development process. This way, it would be possible for the developer do agent the dialogue manager with more ease, since it already knows what are the actions the agent can take.

However, we consider that regarding the dialogue manager, the results of this evaluation might not be completely accurate. First, the number of users that interacted with the agent for the evaluation is low. Second, we have not put any restrictions in the way users talked and gave information to the agent. Since the agent needs to deal with all possible interactions, we believe that a restriction where users could only tell the value of one slot per turn would allow us to realize how specific are the stories we created and how complex is the dialogue manager.

Concluding, we believe that the benefits of using datasets to create a conversational agent outweigh the constraints imposed by some of the annotation properties. As supported by the results of the UEQ part of the questionnaire, using the datasets can create an agent that is good, motivating to use, where the users feel in control the conversation and interact the way they intend, qualities that are required in a conversational agent.

7.2 Knowledge Transfer

By looking the results of our user study on the hotel domain and comparing them to the restaurant domain, we consider that the knowledge transfer between domains (RQ3) was a success. In fact, there were less errors concerning the action selection in the hotel agent that were in the domain agent. Furthermore, although the results of the third questionnaire represent a significant diminish of the quality of the flow, we believe that this scale had worse results due to the influence of the errors caused by the NLU module of the agent. Moreover, as the results of the UEQ suggest the agent is still a good exciting product to use, despite all the understanding errors that it presented.

However, we consider that this method of transferring is still extremely limited. First, the only module to be retained is the dialogue manager. This is obvious, since not only the training data for the NLU module is specific for each domain, but also the behaviour of the agent is also different. This can be confirmed by our attempt at adding information to the Taskmaster-1 dataset using the MultiWOZ restaurant agent, where even though the dialogues of both were regarding the restaurant domain, the complexity and language of both datasets differed (RQ2).

Moreover, in order to transfer knowledge, it is mandatory that the tasks are compatible. In our approach, to have compatible tasks, it is obligatory that the domains have the same number of tasks, and the number of mandatory slots for each task of the destination domain is the same that in the original domain, which sometimes is not feasible. For this reason, it would be important to verify how can the knowledge be transferred, if the number of mandatory slots of both domains differ.

Chapter 8

Conclusion and Future Work

The goal that we wanted to achieve with this work was how to expedite the production of conversational agents while having production quality. To do so, we analysed two major datasets with task-oriented dialogues, the MultiWOZ dataset and the Taskmaster-1 dataset, and examined how to parse the data available to automatically create a Rasa agent. Also, we used the information acquired for one of the domains and verified if it was possible to transfer the knowledge to a different domain, creating a new Rasa agent. Finally, to validate the two approaches, we performed on two agents an automated test on the NLU module, as well as a human evaluation. In this chapter, we discuss our findings regarding the different procedures. Also, we discuss some of the advantages and limitations of Rasa that we found during the elaboration of this work.

8.1 Development Of Conversational Agents Using Datasets

In this work, we wanted to understand how to use the information available in different datasets to create conversational agent using Rasa (RQ1). Thus, MultiWOZ and Taskmaster-1 were used. Both have task-oriented dialogues that were created using the Wizard-of-Oz approach, which allows the dataset to have richer conversations and better training data. Moreover, both of these datasets have different annotations, which leads to a different analysis for both.

In the MultiWOZ dataset, the information is given only in the user turns. In a user turn, the dataset contains the Active Intent, which is the task the user is trying to achieve, the Slots given by the user since the first turn of the user of the dialogue and any Slots that are requested by the user in the current turn.

In contrast, Taskmaster-1 dataset gives information in both the user and the assistant turns. Moreover, in this dataset, the information available are the slots in the current turn utterance and, optionally, if the slots are accepted by the assistant.

Note that even though both datasets give information regarding slots, in the MultiWOZ, a slot is considered any piece of information that may be important for that domain, while in the Taskmaster-1 dataset the existing slots are only the ones required to perform the task. For example, if we would like

to perform a booking for a restaurant and start by looking for one that has **italian** food, while MultiWOZ considers this value as a **restaurant-food** slot, Taskmaster-1 does not have a slot for this information.

After the analysis of each dataset, we then started to parse the data to create a Rasa agent. With the MultiWOZ agent, the parsing was close to immediate. The Active Intents served as base for the intents and, and the slots were used for both the entities and the slots of the Rasa agent. Moreover, we created intents besides the ones in the dataset active intents, so we could differentiate between different utterances. To create these intents, we used information not only from the slots, but also from the requested slots. However, even with this information, some of the utterances were not able to be labeled, which lead to an incomplete training and the lack of some type of utterances. This is the example of sentences such as *Yes* or *Go with it*. Both these sentences can represent affirmations, which are important to a dialogue. This was validated by the human evaluation. Often, the users affirmed the intent of proceeding for a booking at a restaurant, but the assistant did not understand what the users meant.

Moreover, for the Rasa agent to be complete, we defined the Rules and the Stories, together with the actions and responses. This required some more processing, but using the information available from the active intents, slots and requested slots, we are able to choose the action to perform by the agent. For each of the active intents, we created an action. The use of of the requested slots lead to the creation of a single rule, which finds information for the user. Finally, using the slots and all the dialogues available, we were able to define the mandatory slots, which together with the dialogues, created the responses, while also creating the stories for the assistant.

From our user evaluation, we are able to conclude that although with some errors, the parsing of the data with the information available in the MultiWOZ dataset can create a good dialogue flow to perform the tasks. For example, in our evaluation, every user was able to conclude the tasks presented. However, the flaws of the dialogue flow are still noticeable. This could be seen in the third task given to the users, where approximately one turn of the user was to answer the assistant about the day of the booking, after changing the number of people (even if the user had already given the day of booking).

In Taskmaster-1, the process was more difficult. Since the only available information are the slots, which can be used to create the entities and slots of the Rasa agent, we also needed to label the utterances as intents. To do this, we used assumptions. Using this approach, we concluded that using only slots, we could not label correctly most of the utterances, since we do not have knowledge on the context of each part of the dialogue. Given the lack of information in this dataset, and the fact that one of MultiWOZ domains is the restaurant domain, we had the opportunity to verify if the information acquired to create a conversational agent (MultiWOZ Restaurant agent) can be used to increase the information of Taskmaster-1 (RQ2). From this experience, we realized that even using a agent that is for the same domain, the training data was not similar, which lead to incorrect predictions by the agent. Since both these methods failed, we then failed to produce a working Rasa agent for the Taskmaster-1 dataset.

That said, there are several conclusions we can make regarding required information in datasets to create the parsing of data to create conversational agent (RQ1). Only the information alone from the slots is not sufficient to create a working Rasa agent. In fact, to create a Rasa agent, it is required

intents, slots and required slots. Note that even having this information, there are still possible errors, such as the labeling problems we mentioned in the MultiWOZ dataset. Moreover, to create the dialogue flow, we need to use all the information given and create the actions and responses.

In order to create better and more complete conversational assistants, we address these two problems with some possible changes to the dataset annotation process. For example, creating more modular intents for the utterances, that represent what the user is trying to accomplish with the intent. Moreover, when the dialogues are created, create a set of answers the assistant can give to the user, and label each of the assistant answers. The changes we just mentioned can be seen in Table 8.1.

Utterance	**State**	
USER: Hello. I'm looking for a restaurant called city stop restaurant.	**Task**	find_restaurant
	Intent	find-restaurant
	Slots	restaurant-name: city stop restaurant
ASSISTANT: City stop restaurant is a place located in the north of town that serves expensive european food. Would you like to perform a booking?	**Action**	action_find_restaurant
USER: Yes, please.	**Task**	book_restaurant
	Intent	affirm
	Slots	-
ASSISTANT: Can you tell me the number of people, please?	**Action**	action_ask_number_of_people

Table 8.1: Part of dialogue to find and make booking for a restaurant using the changes proposed for the dataset annotations.

8.2 Knowledge Transfer Between Domains

After parsing the information and creating the agent in Rasa, we wanted to assess the possibility of using the knowledge from a agent with the restaurant domain to create a new agent with another domain (RQ3). More specifically, we assess if by changing the training data used for the Rasa NLU module, we can create a working agent. This means that we preserve the domain, the stories and the rules of the origin agent.

From the results of the user test, we confirm that the agent was worse in comparison to the restaurant agent on the experience scale. This was expected since we changed the training data. However, regarding the Flow scale, although it was verified a decline, this can be related to the difficulties seen in the the experience scale, but still kept a good conversation flow. This is also supported by the the number of turns where existed a action selection error, where on average, there was on maximum one error regarding the choice of action.

That said, we believe that the transfer of the knowledge acquired from the restaurant agent was done successfully, which is supported by the results of our evaluation. Specifically, the stories defining the dialogue manager can not only be used for one domain but to several domains. However, from this

process, we believe that in order to transfer knowledge between domains, the tasks of the target domain must be compatible with the tasks of the original domain. In our work, we defined this compatibility by using the number of mandatory slots per task. We believe that number of mandatory slots must be the same, since the stories created for the restaurant agent were created using that specific number of slots, and if the new agent has a different number of slots or mandatory slots, then there will be different conversation paths that are not covered by the stories on the origin agent.

8.3 Rasa Open Source

From this work, we were also able to understand some of the advantages and limitations of Rasa Open Source. For starters, the definition of a new agent is simple, since it has few properties to define. Moreover, using stories and rules, it is simple to define the dialogue flow. Although very effective, the definition of every conversation path in the dialogue manager can be extremely difficult, since there are a great amount of paths a user can take. This was shown during the parsing of data from the MultiWOZ dataset to create a Rasa agent, since each dialogue led to a different story. Also, another advantage of using stories to define the dialogue flow is the ease of transferring knowledge from one domain to the other, using a agent from an existing agent.

Furthermore, we consider that Rasa is limited to single domain agents. After parsing the information from MultiWOZ do Rasa, it was noticeable that there were several errors labeling entities, specially in the inform intent, when both have similar training data. One of the possibilities that could solve this issue is by introducing information of the previous turns of the dialogue, such as the previous intents, slots and actions taken by the agent. However, there is another possibility by the creator of the agent. During the design of this agent, if there are several entities with the same training data, create a slot for each of those entities, and replace the multiple entities with a single general entity. The downside of this solution is that it requires the creator of the agent to add new actions and stories, in order to define the values for these slots correctly during a conversation.

Overall, the results we obtained from the data parsing of datasets and knowledge transfer seem to indicate that it is possible to expedite the creation of conversational agents. However, the expedition of the conversational agents are still limited by several dataset properties and in the case of transferring knowledge between domains, it is limited by the configuration of the base agent. Moreover, in the case of multiple domain agents, Rasa still presents some limitations that hinders the process of creating a agent.

8.4 Future Work

There are several things that could be done in order to explore in further detail our research questions, as well as the capabilities of Rasa:

- Create a agent from the datasets using a single entity for the dataset slots where the values are

the same and use the training data to assign the entity to the correct slot.

- Rearrange the way the multi domain agent is being created. For example, have entities that are for several domains, and use the actions to choose what is the correspondent domain.
- Check how having a different number of mandatory slots per task affect the knowledge transfer between domains, such such as the MultiWOZ restaurant domain and hotel domain.
- Have an Human Evaluation where the users were encouraged to give less information per turn to assess in detail the quality of creating the dialogue manager using the dataset dialogues.
- Check if is possible to modulate the dialogue manager without using the dialogues in the dataset, for example using Rasa Forms that allows the definition of a conversation based on the mandatory slots.

Bibliography

[1] Harald Aust and Martin Oerder. “Dialogue control in automatic inquiry systems”. In: *Spoken Dialogue Systems-Theories and Applications*. 1995.

[2] Gonçalo Baptista. “The Virtual Suspect Meets Alexa”. MA thesis. Instituto Superior Técnico, Oct. 2020.

[3] Pawel Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gasic. “MultiWOZ - A Large-Scale Multi-Domain Wizard-of-Oz Dataset for Task-Oriented Dialogue Modelling”. In: *CoRR* abs/1810.00278 (2018). arXiv: 1810.00278. URL: http://arxiv.org/abs/1810.00278.

[4] Bill Byrne, Karthik Krishnamoorthi, Chinnadhurai Sankar, Arvind Neelakantan, Daniel Duckworth, Semih Yavuz, Ben Goodrich, Amit Dubey, Kyu-Young Kim, and Andy Cedilnik. “Taskmaster-1: Toward a Realistic and Diverse Dialog Dataset”. In: 2019.

[5] Chih-Wen Goo, Guang Gao, Yun-Kai Hsu, Chih-Li Huo, Tsung-Chieh Chen, Keng-Wei Hsu, and Yun-Nung Chen. “Slot-Gated Modeling for Joint Slot Filling and Intent Prediction”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 753–757. DOI: 10.18653/v1/N18-2118. URL: https://www.aclweb.org/anthology/N18-2118 (visited on 12/08/2020).

[6] Javeria Habib, Shuo Zhang, and Krisztian Balog. “IAI MovieBot: A Conversational Movie Recommender System”. In: *CoRR* abs/2009.03668 (2020). arXiv: 2009.03668. URL: https://arxiv.org/abs/2009.03668.

[7] John F Kelley. “An iterative design methodology for user-friendly natural language office information applications”. In: *ACM Transactions on Information Systems (TOIS)* 2.1 (1984), pp. 26–41.

[8] Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, and Asli Celikyilmaz. “End-to-End Task-Completion Neural Dialogue Systems”. In: *the 8th International Joint Conference on Natural Language Processing*. IJCNLP 2017, Nov. 2017. URL: https://www.microsoft.com/en-us/research/publication/end-end-task-completion-neural-dialogue-systems/.

[9] Bing Liu and Ian Lane. “Attention-Based Recurrent Neural Network Models for Joint Intent Detection and Slot Filling”. In: *CoRR* abs/1609.01454 (2016). arXiv: 1609.01454. URL: http://arxiv.org/abs/1609.01454.

[10] Bing Liu and Ian Lane. “End-to-End Learning of Task-Oriented Dialogs”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Student Research Workshop*. New Orleans, Louisiana, USA: Association for Computational Linguistics, June 2018, pp. 67–73. DOI: 10.18653/v1/N18-4010. URL: https://www.aclweb.org/anthology/N18-4010 (visited on 12/08/2020).

[11] Tim Paek. “Empirical methods for evaluating dialog systems”. In: *Proceedings of the ACL 2001 Workshop on Evaluation Methodologies for Language and Dialogue Systems*. 2001.

[12] Libo Qin, Wanxiang Che, Yangming Li, Haoyang Wen, and Ting Liu. “A Stack-Propagation Framework with Token-Level Intent Detection for Spoken Language Understanding”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 2078–2087. DOI: 10.18653/v1/D19-1214. URL: https://www.aclweb.org/anthology/D19-1214.

[13] Martin Schrepp, Andreas Hinderks, and Jörg Thomaschewski. “Construction of a Benchmark for the User Experience Questionnaire (UEQ).” In: *Int. J. Interact. Multim. Artif. Intell.* 4.4 (2017), pp. 40–44.

[14] Vladimir Vlasov, Johannes E. M. Mosig, and Alan Nichol. “Dialogue Transformers”. In: *arXiv:1910.00486 [cs]* (May 2020). arXiv: 1910.00486. URL: http://arxiv.org/abs/1910.00486 (visited on 12/15/2020).

[15] Marilyn A. Walker, Diane J. Litman, Candace A. Kamm, and Alicia Abella. “PARADISE: A Framework for Evaluating Spoken Dialogue Agents”. In: *CoRR* cmp-lg/9704004 (1997). URL: http://arxiv.org/abs/cmp-lg/9704004.

[16] Yushi Xu and Stephanie Seneff. “Dialogue Management Based on Entities and Constraints”. In: *Proceedings of the SIGDIAL 2010 Conference*. Tokyo, Japan: Association for Computational Linguistics, Sept. 2010, pp. 87–90. URL: https://www.aclweb.org/anthology/W10-4317 (visited on 12/06/2020).

Appendix A

MultiWOZ Mandatory Slots

Active Intent	Slots	Mandatory Slots
find_hotel	hotel-pricerange hotel-bookday hotel-type hotel-parking hotel-wifi hotel-bookpeople hotel-area hotel-bookstay hotel-name hotel-stars	-
book_hotel	hotel-pricerange hotel-bookday hotel-type hotel-parking hotel-wifi hotel-bookpeople hotel-area hotel-bookstay hotel-name hotel-stars	hotel-bookday hotel-bookpeople hotel-bookstay hotel-name

Table A.1: Mandatory slots for the hotel domain.

Active Intent	Slots	Mandatory Slots
find_restaurant	restaurant-food restaurant-area restaurant-pricerange restaurant-name restaurant-bookpeople restaurant-booktime restaurant-bookday	-
book_restaurant	restaurant-food restaurant-area restaurant-pricerange restaurant-name restaurant-bookpeople restaurant-booktime restaurant-bookday	restaurant-name restaurant-bookpeople restaurant-booktime restaurant-bookday

Table A.2: Mandatory slots for the restaurant domain.

Active Intent	Slots	Mandatory Slots
find_taxi	taxi-departure taxi-destination taxi-arriveby taxi-leaveat	taxi-departure taxi-destination
find_attraction	attraction-area attraction-type attraction-name	-
find_hospital	hospital-department	-
find_police	police-name	-
find_bus	bus-day bus-departure bus-destination bus-leaveat	-

Table A.3: Mandatory slots for all the remaining domains

Appendix B

Knowledge Transfer

Listing 10 Original story from the restaurant dataset used to create the hotel story

```
- story: dialogue 11
steps:
- intent: find_restaurant
  entities:
  - restaurant-area: centre
  - restaurant-pricerange: cheap
- action: action_find_restaurant
- intent: inform
  entities:
  - restaurant-food: italian
- action: action_find_restaurant
- intent: book_restaurant
  entities:
  - restaurant-bookday: wednesday
  - restaurant-bookpeople: 7
  - restaurant-booktime: 11:45
- action: action_book_restaurant
- intent: goodbye
- action: utter_say_goodbye
```

Appendix C

Evaluation

C.1 Agent Implementation

Response	Message
utter_say_goodbye	Goodbye!
utter_ask_hotel_name	Can you tell me the name of the lodging place?
utter_ask_hotel_bookpeople_hotel_bookstay_hotel_name	Can you tell me the name of the place, how many people are staying and for how many nights, please?
utter_ask_hotel_bookday	What is the day of your check-in?
utter_ask_hotel_bookstay_hotel_name	What is the name of the place and how many nights are you staying?
utter_ask_hotel_bookstay	Can you tell me how many nights are you staying?
utter_ask_hotel_bookpeople_hotel_bookstay	How many nights are you staying and for how many people?
utter_ask_hotel_bookpeople_hotel_name	What is the name of the place, as well as the number of people?
utter_ask_hotel_bookday_hotel_bookstay	Please, tell me when do you wish to check-in, as well as how many nights you are staying.
utter_ask_hotel_bookday_hotel_bookstay_hotel_name	Can you tell me the name of the place, the day of check in, and the duration of your stay, please?
utter_ask_hotel_bookpeople	How many people is this booking for?
utter_ask_hotel_bookday_hotel_name	Tell me the name of the place, as well as the day of check in, please.

Table C.1: Text defined for the responses for the Hotel model

Item	Scale	Left	Right
1	Attractiveness	annoying	enjoyable
2	Perspicuity	not understandable	understandable
3	Perspicuity	easy to learn	difficult to learn
4	Stimulation	valuable	inferior
5	Stimulation	boring	exciting
6	Stimulation	not interesting	interesting
7	Dependability	unpredictable	predictable
8	Dependability	obstructive	supportive
9	Attractiveness	good	bad
10	Perspicuity	complicated	easy
11	Attractiveness	unlikable	pleasing
12	Attractiveness	unpleasant	pleasant
13	Dependability	secure	not secure
14	Stimulation	motivating	demotivating
15	Dependability	meets expectations	does not meet expectations
16	Perspicuity	clear	confusing
17	Attractiveness	attractive	unattractive

Table C.2: UEQ questionnaire items for each scale

C.2 Materials

C.2.1 Questionnaire

Statement	Scale
The agent was able to understand what I wrote	Understanding Capabilities
The agent is able to understand natural language	Understanding Capabilities
The agent is intelligent	Understanding Capabilities
The agent's capabilities met my expectations	Experience
The agent's answers often broke my line of thought	Experience
I had difficulty formulating my sentences to be understood	Experience
I had all the information needed to engage properly with the agent	Limitations
I was able to learn more than I initially knew during the interaction	Limitations
There were questions the agent could not answer	Limitations
I felt the conversation was fluid	Flow
I managed to conclude my tasks without effort	Flow
I did not know what do say next to the agent	Flow
The agent helped me conclude my tasks	Flow

Table C.3: Questions to evaluate the capabilities of the model

Appendix D

Automatic Evaluation

D.1 Multi Domain Model

D.2 Restaurant Model

D.3 Hotel Model

Intent	Support	Precision	Recall	F1-Score
goodbye	1020	0.9701	0.9853	0.9776
book_restaurant	297	0.9017	0.8653	0.8832
find_taxi	212	0.9231	0.9057	0.9143
request	783	0.8986	0.9502	0.9236
find_hotel	401	0.9211	0.8728	0.8963
book_train	297	0.8702	0.8350	0.8522
find_train	490	0.9154	0.9490	0.9319
book_train	252	0.8352	0.8849	0.8593
inform	1624	0.8935	0.8571	0.8749
find_police	2	0.5	0.5	0.5
find_attraction	427	0.9069	0.8899	0.8983
find_restaurant	454	0.8565	0.9075	0.8813

Table D.1: Precision, Recall and F1-Score, as well as the number of tests for each of intents in the multi domain model

Entity	Support	Precision	Recall	F1-Score
hotel-type	177	0.7296	0.8079	0.7668
restaurant-bookday	256	0.9762	0.9609	0.9685
attraction-type	289	0.9218	0.9792	0.9497
train-departure	611	0.9537	0.9771	0.9652
taxi-departure	232	0.9159	0.8922	0.9039
train-bookpeople	198	0.8826	0.9495	0.9148
hotel-area	135	0.8163	0.8889	0.8511
train-arriveby	296	0.8947	0.9764	0.9338
restaurant-food	422	0.9589	0.9953	0.9767
required_info	1883	0.9261	0.9846	0.9544
taxi-arriveby	56	0.9070	0.6964	0.7879
train-leaveat	214	0.8682	0.8925	0.8802
attraction-name	363	0.9164	0.9669	0.9410
hotel-bookday	231	0.9051	0.9913	0.9463
hotel-name	353	0.9462	0.9972	0.9710
taxi-leaveat	107	0.9775	0.8131	0.8878
hotel-pricerange	206	0.7846	0.9369	0.8540
restaurant-booktime	328	0.9486	0.9573	0.9530
hotel-bookstay	204	0.8354	0.9951	0.9083
restaurant-area	251	0.8677	0.8884	0.8780
taxi-destination	219	0.9289	0.8950	0.9116
restaurant-bookpeople	236	0.9205	0.9322	0.9263
train-day	461	0.9614	0.9718	0.9666
hotel-stars	128	0.8591	1.0000	0.9242
hotel-parking	17	0.8000	0.7059	0.7500
restaurant-pricerange	264	0.8764	0.8598	0.8681
hotel-internet	22	0.8947	0.7727	0.8293
attraction-area	202	0.8632	0.9059	0.8841
hotel-bookpeople	193	0.8750	0.9430	0.9077
train-destination	650	0.9286	0.9800	0.9536
restaurant-name	346	0.9380	0.9624	0.9501

Table D.2: Results of the Precision, Recall and F1-Score as well as the number of tests for each of the entities of the multi domain model.

Intent	Support	Precision	Recall	F1-Score
request	20	0.7500	0.9000	0.8182
find_restaurant	58	0.8333	0.9483	0.8871
goodbye	60	0.9836	1.0000	0.9917
book_restaurant	39	0.8750	0.8974	0.8861
inform	65	0.8824	0.6923	0.7759

Table D.3: Metrics obtained for the intents by the automatic evaluation for the restaurant model

Entity	Support	Precision	Recall	F1-Score
restaurant-food	67	0.9155	0.9701	0.9420
required_info	51	0.9804	0.9804	0.9804
restaurant-booktime	49	0.9800	1.0000	0.9899
restaurant-name	45	0.9394	0.6889	0.7949
restaurant-pricerange	41	0.9268	0.9268	0.9268
restaurant-bookday	39	0.9512	1.0000	0.9750
restaurant-bookpeople	35	0.9189	0.9714	0.9444
restaurant-area	43	0.8889	0.9302	0.9091

Table D.4: Metrics obtained by the automatic evaluation of the Entities for the restaurant model

Intent	Support	Precision	Recall	F1-Score
goodbye	69	0.9855	0.9855	0.9855
request	26	0.8182	0.6923	0.7500
book_hotel	41	0.7600	0.9268	0.8352
inform	91	0.8415	0.7582	0.7977
find_hotel	63	0.8806	0.9365	0.9077

Table D.5: Metrics obtained by the automatic evaluation for the intents of the hotel model

Entity	Support	Precision	Recall	F1-Score
hotel-bookpeople	34	0.6000	0.6176	0.6087
hotel-bookday	51	0.9623	1.0000	0.9808
required_info	44	0.9750	0.8864	0.9286
hotel-type	36	0.6078	0.8611	0.7126
hotel-parking	1	0.5000	1.0000	0.6667
hotel-pricerange	37	0.8780	0.9730	0.9231
hotel-bookstay	39	0.6000	0.6923	0.6429
hotel-area	25	0.9259	1.0000	0.9615
hotel-name	43	0.9762	0.9535	0.9647

Table D.6: Metrics obtained by the automatic evaluation for the entities of the hotel model

Appendix E

Human Evaluation

E.1 UEQ Results

E.2 Capabilities Questionnaire Results

Item	Mean	Variance	Std. Dev.	Confidence Interval	
annoying - enjoyable	1.882	0.495	0.697	1.551	2.214
not understandable-understandable	1.882	1.985	1.409	1.213	2.552
easy to learn - difficult to learn	1.588	3.257	1.805	0.730	2.446
valuable - inferior	1.491	2.015	1.419	0.730	2.446
boring - exciting	0.941	2.069	1.435	0.796	2.145
not interesting - interesting	1.000	0.000	0.000	0.259	1.625
unpredictable - predictable	0.941	1.809	1.345	0.302	1.581
obstructive - supportive	1.176	1.654	1.286	0.565	1.788
good - bad	1.647	1.493	1.222	1.066	2.228
complicated - easy	1.882	1.235	1.111	1.354	2.411
unlikable - pleasing	1.471	1.765	1.328	0.839	2.102
unpleasant - pleasant	1.706	0.971	0.985	1.238	2.174
secure - not secure	2.176	0.779	0.883	1.757	2.596
motivating - demotivating	1.294	1.596	1.263	0.694	1.895
meets expectations - does not meet expectations	1.765	1.691	1.300	1.147	2.383
clear - confusing	1.294	1.596	1.264	0.694	1.895
attractive - unattractive	1.000	1.875	1.369	0.349	1.651
friendly - unfriendly	2.235	1.441	1.200	1.665	2.806

Table E.1: Mean, Variance, Standard Deviation and Confidence Interval with p=0.05 for each item of the UEQ obtained in the restaurant model user test.

Sentence	Mean	Variance	Sdt. Dev.	Confidence	Confidence Interval	
The agent was able to understand what I wrote	1.235	1.316	1.147	0.545	0.690	1.781
The agent is able to understand natural language	1.941	1.559	1.249	0.594	1.348	2.535
The agent is intelligent	1.588	1.382	1.176	0.559	1.029	2.147
The agent's capabilities met my expectations	1.941	0.934	0.966	0.459	1.482	2.401
The agent's answers often broke my line of thought	1.882	1.485	1.219	0.579	1.303	2.462
I had difficulty formulating my sentences to be understood	1.941	1.934	1.391	0.661	1.280	2.602
I had all the information needed to engage properly with the agent	2.294	0.721	0.849	0.404	1.891	2.698
I was able to learn more than I initially knew during the interaction	1.882	1.360	1.166	0.554	1.328	2.437
There were questions the agent could not answer	1.294	3.096	1.759	0.836	0.458	2.130
I felt the conversation was fluid	1.294	2.721	1.649	0.784	0.510	2.078
I managed to conclude my tasks without effort	1.824	1.154	1.074	0.511	1.313	2.334
I did not know what do say next to the agent	1.941	1.684	1.298	0.617	1.324	2.558
The agent helped me conclude my tasks	2.118	1.235	1.111	0.528	1.589	2.646

Table E.2: Results for each of the items in the third part of the questionnaire of the human evaluation for the restaurant model

Item	Mean	Variance	Std. Dev.	Confidence Interval	
annoying - enjoyable	1.529	1.419	0.675	0.855	2.204
not understandable-understandable	1.647	1.412	0.671	0.976	2.318
easy to learn - difficult to learn	1.588	1.417	0.673	0.915	2.262
valuable - inferior	1.588	1.326	0.630	0.958	2.218
boring - exciting	1.176	1.131	0.538	0.639	1.714
not interesting - interesting	1.588	1.064	0.506	1.082	2.094
unpredictable - predictable	0.795	1.480	0.704	0.061	1.468
obstructive - supportive	1.471	1.375	0.653	0.817	2.124
good - bad	1.765	1.300	0.618	1.147	2.383
complicated - easy	1.529	1.463	0.695	0.834	2.225
unlikable - pleasing	1.471	1.068	0.507	0.963	1.978
unpleasant - pleasant	1.529	1.068	0.507	1.022	2.037
secure - not secure	2.118	0.928	0.441	1.677	2.559
motivating - demotivating	1.412	1.326	0.630	0.782	2.042
meets expectations - does not meet expectations	1.647	1.057	0.503	1.145	2.150
clear - confusing	1.118	1.269	0.603	0.514	1.721
attractive - unattractive	0.882	1.691	0.804	0.078	1.686
friendly - unfriendly	2.235	1.091	0.519	1.716	2.754

Table E.3: Mean, Variance, Standard Deviation and Confidence Interval with p=0.05 for each item of the UEQ obtained in the hotel model user test.

Sentence	Mean	Variance	Sdt. Dev.	Confidence	Confidence Interval	
The agent was able to understand what I wrote	1.000	1.625	1.275	0.606	0.394	1.606
The agent is able to understand natural language	1.941	0.684	0.827	0.393	1.548	2.334
The agent is intelligent	1.529	1.640	1.281	0.609	0.921	2.138
The agent's capabilities met my expectations	1.529	0.890	0.943	0.448	1.081	1.978
The agent's answers often broke my line of thought	0.706	2.971	1.724	0.819	-0.113	1.525
I had difficulty formulating my sentences to be understood	1.235	2.816	1.678	0.798	0.438	2.033
I had all the information needed to engage properly with the agent	2.529	0.265	0.514	0.245	2.285	2.774
I was able to learn more than I initially knew during the interaction	1.941	2.559	1.600	0.760	1.181	2.702
There were questions the agent could not answer	0.235	4.441	2.107	1.002	-0.766	1.237
I felt the conversation was fluid	0.765	3.691	1.921	0.913	-0.149	1.678
I managed to conclude my tasks without effort	1.294	1.596	1.263	0.600	0.694	1.895
I did not know what do say next to the agent	1.294	2.471	1.572	0.747	0.547	2.041
The agent helped me conclude my tasks	1.706	1.096	1.047	0.498	1.208	2.203

Table E.4: Results for each of the items in the third part of the questionnaire of the human evaluation for the hotel model