Pontifícia Universidade Católica
do Rio de Janeiro

Jefry Sastre Pérez

A Framework to automate data science tasks through personalized chatbots

Tese de Doutorado

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor : Prof. Helio Cortes Vieira Lopes
Co-Advisor: Dr. Marx Leles Viana

Rio de Janeiro
November 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Jefry Sastre Pérez

A Framework to automate data science tasks through personalized chatbots

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática. Approved by the Examination Committee:

Prof. Helio Cortes Vieira Lopes
Advisor
Departamento de Informática – PUC-Rio

Dr. Marx Leles Viana
Co-Advisor
Pesquisador Autônomo

Profª. Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Prof. Bruno Feijó
Departamento de Informática – PUC-Rio

Prof. Luiz André Portes Paes Leme
UFF

Prof. Marcos de Oliveira Lage Ferreira
UFF

Prof. Cassio Freitas Pereira de Almeida
ENCE

Prof. Marco Antonio Casanova
Departamento de Informática – PUC-Rio

Rio de Janeiro, November 19th, 2021

Jefry Sastre Pérez

The author graduated in Computer Science from the University of Havana in 2015. In 2018, he obtained his master's degree in Computer Science at PUC-Rio. He has an interest in Data Science, Machine Learning, and Software Engineering.

Bibliographic Data

Sastre Pérez, Jefry

A Framework to automate data science tasks through personalized chatbots / Jefry Sastre Pérez; advisor: Helio Cortes Vieira Lopes; co-advisor: Marx Leles Viana. – 2021.

78 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. ciencia de dados. 2. automação de processos. 3. engenharia de software. I. Cortes Vieira Lopes, Helio. II. Leles Viana, Max. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Acknowledgments

In the first place, to my advisors, Hélio Lopes and Max Vianna, for the opportunity, confidence, creativity, passion, and dedication to the research and the valuable teachings through this period. To my colleagues for providing me an exceptional environment and spirit. Also for their friendship and will to contribute to this work.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. Finally, I would like to thank Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), for partially financing this research under grant #140685/2018-9.

For you all, my sincere thank you!

Abstract

Sastre Pérez, Jefry; Cortes Vieira Lopes, Helio (Advisor); Leles Viana, Max (Co-Advisor). **A Framework to automate data science tasks through personalized chatbots**. Rio de Janeiro, 2021. 78p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Several solutions have been created for automating specific data science scenarios and implementations of personalized content in conversational interfaces. However, the overall understanding of these conversational interfaces that provide personalized suggestions for data scientists is still poorly explored. We identify the need to automate data science procedures up to different levels of automation. Our research focuses on helping data scientists during the automation of these procedures by using conversational interfaces. We propose a framework for creating a chat-bot system to facilitate the automation of data science common scenarios. In addition, we instantiate the framework in two different data science scenarios. The first scenario focuses on outlier detection, and the second scenario on data cleaning. We conducted a study with 28 participants to demonstrate that data scientists can use the proposed framework. All participants completed the activities correctly, and 75 to 80% found the framework relatively easy to extend and use. Our analysis suggests that the use of conversational interfaces can facilitate the automation of data science tasks.

Keywords

data science; automation; software engineering.

Resumo

Sastre Pérez, Jefry; Cortes Vieira Lopes, Helio; Leles Viana, Max. **Um framework para automatizar tarefas de ciencia de dados através de interfaces conversacionais**. Rio de Janeiro, 2021. 78p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Diversas soluções foram criadas para automatizar cenários específicos de ciência de dados e implementações de conteúdo personalizado em interfaces de conversação. No entanto, o entendimento geral dessas interfaces de conversação que fornecem sugestões personalizadas para cientistas de dados ainda é pouco explorado. Identificamos a necessidade de automatizar procedimentos de ciência de dados até diferentes níveis de automação. Nossa pesquisa se concentra em ajudar os cientistas de dados durante a automação desses procedimentos usando interfaces conversacionais. Propomos um framework para a criação de um sistema chat-bot para facilitar a automação de cenários comuns de ciência de dados. Além disso, instanciamos a solução em dois cenários diferentes de ciência de dados. O primeiro cenário se concentra na detecção de valores discrepantes e o segundo na limpeza de dados. Conduzimos um estudo com 28 participantes para demonstrar que os cientistas de dados podem usar a solução proposta. Todos os participantes concluíram as atividades corretamente e 75 a 80 % acharam o framework relativamente fácil de estender e usar. Nossa análise sugere que o uso de interfaces conversacionais pode facilitar a automação de tarefas de ciência de dados.

Palavras-chave

ciencia de dados; automação de processos; engenharia de software.

Table of Contents

List of Figures

List of Abbreviations

AutoML – Automated machine learning

IA – Artificial Intelligence

ML – Machine Learning

NLP – Natural Language Processing

1
Introdução

1.1
Motivation

Data Science (DS) and Machine Learning (ML) have been gaining popularity and strength in different areas such as Informatics in healthcare Liu et al. (2020); Knoop et al. (2019); Alemu and Huang (2020), Internet of Things Tallyn et al. (2018), and Software Engineering Smestad (2018); Sun and Zhang (2018), to name a few. The application of ML and DS technology is a priority for many organizations Idoine et al. (2018). But, according to Wang et al. (2021), it is such a complex and time-consuming activity that there is a lack of professionals to fill the job demands. Recently, to widen the adoption and improve results, the community has started to explore diverse levels of automation of the execution of DS/ML projects. However, developing automations to support users throughout DS/ML processes is not a trivial task, as it requires the creation of systems capable of providing answers for data scientists with different levels of knowledge and expertise.

Wang et al. (2021) conducted a survey with 217 DS and ML professionals to identify desired levels of automation during the lifecycle of DS/ML project, several of whom interested in a DS/ML project: stakeholders, programmers, and domain experts. However, the authors consider "data scientists" only those who code and build models. They classified automations into five levels: no automation (L0), human-directed (L1), system-suggested (L2), system-directed (L3), and full automation (L4).

Figure 1.1 depicts the interest of all participants concerning the levels of automation during the model building and feature engineering stages. However, most of the time, data scientists prefer system suggestions over full automation. Conversely, domain experts prefer full automation in most of the stages involving coding. It presents different points of view about automation in DS/ML projects and show how much practitioners want the automation of those projects.

To automate the different stages of the DS/ML process, we need to define the different levels of automation we will provide to data scientists. There are

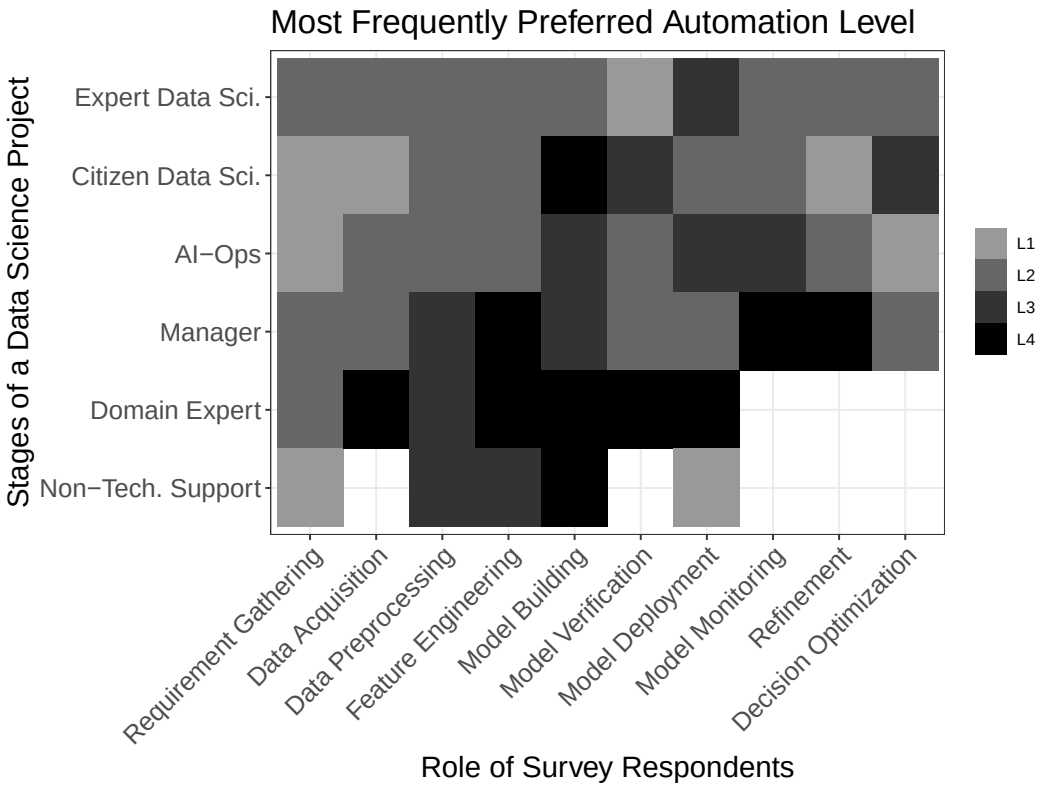


Figure 1.1: Desired levels of automation in every stage of the DS/ML project. Image taken and adapted from (Wang et al., 2021).

several attempts to describe and formalize the data scientists' profile (Costa and Santos, 2017; Harris et al., 2013; Berthold, 2019). Harris et al. (2013) relate the use of the term data science to describe different profiles. In addition, they present the use of the term for hidden the expectations of the role. So, the authors named data scientist as a general concept comprising traditional areas such as mathematics and computer science, among others. Such areas have clear expectations, but all together creates confusion. To better define data scientists' expectations, the authors conducted online surveys to gather data about the required or desired skills. They summarized 22 skills divided into five categories: Business, ML, Math, Programming, and Statistics. The authors clustered the results in four profiles: Data Businesspeople, Data Creatives, Data Developers, and Data Researchers.

Data Businesspeople focus on organization and business value through data analysis. They see themselves as leaders or advisors. Data Creatives concentrate on a broad set of tools to attack a problem. They have academic and professional experience and see themselves as artists or hackers. Most of the participants had contributed to open source projects. Data Developers focus on technical problems about data, such as: where to store it and how to extract it. They are definitively the group with most coding skills, with

computer science or engineering degrees. The Data Researchers group gathers people with academic publications and most of them have a Ph.D. degree. They focus on the use of data to understand complex processes.

Costa and Santos (2017) present a conceptual model for a data scientist profile, dividing the data scientists' characteristics into two groups: knowledge base and skill set. The knowledge base contains formal requirements such as security, data ethics, information systems, and data management, among others. On the other hand, the skill set focuses on tasks that a data scientist must perform, such as visualizations, gather data, statistical methods, to cite a few. The authors define data scientists as capable of extracting patterns from data no matter the challenges, communicating their results, generating data artifacts, and improving decision-making processes. We can observe that the union of the four groups described by Harris et al. (2013) complete the conceptual model presented by Costa and Santos (2017). Porter (2015) showed how clustering from a population sample of STEM educators confirms the groups presented by Harris et al. (2013), contributing to reaffirm the validity of the data scientists groups.

Berthold (2019) reviews the required data scientists' skills considering three categories: novice, apprentice, and expert. The novice data scientists can improve the results of a well-defined data science problem. The apprentices will have to deal with data collection and processing, and can answer hypothetical questions about the data. In turn, the expert group must deal with poorly described situations and suggest new exploratory guidance. They are expected to continuously refine their models with user feedback and use the knowledge from previous experiences. The profiles described by Berthold (2019) hold strong theoretical backgrounds, augmented with insights from previous experiences. These categories complement previous works, concluding that data scientists should have both theoretical knowledge and practical experience.

We foresee the use of personal assistants to automate parts of the DS/ML projects. Fast et al. (2018) exemplify the use of conversational interfaces to create personal assistants in the data science area. One of the main advantages they point out is productivity, and their experiments showed that users using data science assistants complete specific data science tasks 2.6 times faster than using python scripts. Experienced and novice users highlight diverse benefits: experienced users notice that the use of assistants can save time as wrappers of python functions, and novice users take advantage of the structural guidance provided by the personal assistant. However, to reach all the desired levels of automation proposed by Wang et al. (2021), we must enhance the conversational interfaces with the ability to suggest further steps (L2 system

suggestions). The chatbot suggestions may vary depending on the situation. For instance, novice users may require more guidance, while expert users may want to create personalized execution pipelines. In any case, the conversational interface must adapt its suggestions to fit the user's needs.

Recent research on conversational interfaces has raised the concern of personalized interactions (cf. Abdellatif et al. (2020); Suta et al. (2020); Reis et al. (2020)). In most cases, the authors mention personalized content as a means to improve overall user satisfaction. However, it is a way to provide specific content for different groups of users. Suta et al. (2020) highlight two limitations: understanding the context and generating a relevant response. Their work is a bit more specific about these limitations and states that bots should answer with emotions or/and personalized content. Abdellatif et al. (2020) show how developers have real concerns about Natural Language Understanding, specifically, the ability to answer. According to the authors, it has a real impact on user satisfaction. Therefore, the ability to understand is crucial, but a proper response is critical as well. Reis et al. (2020) provide another example, bringing the physician and the patient concerns, use cases, and limitations about the use of chatbots in healthcare environments. However, such restrictions can be generalized as the ability to deal with different groups of users. In their specific case, the authors refer to sensitive information and patient-doctor confidentiality. In a more general sense, the authors engage with previously presented work on the need to provide personalized content for different groups of users.

1.2 Problem definition

With the growing demand for intelligent software solutions, the data science area has gained immense recognition. Data science processes are complex and time-consuming and lack qualified professionals to fill the job demands. The complexity and the lack of qualified professionals raise the need for some degree of automation. In this work, we address the current necessity of automating DS/ML procedures. Conversational interfaces emerge as solutions to automate different processes and interact with technology using natural language. We believe that personalized content inside conversational interfaces will facilitate the automation requested by data scientists in some of the DS/ML stages. In particular, our goal is to propose a framework for automating DS/ML processes through conversational interfaces with personalized interactions.

To achieve our goal, we address the following research questions:

– **RQ1:** Can we use personalized interactions in conversational interfaces to reach the system-suggestion level of automation (L2) inside the framework's core functionalities?
– **RQ2:** Will data developers be able to extend the proposed framework for specific scenarios?

1.3 Methodology

In Figure 1.2, we present our plan to answer the research questions.

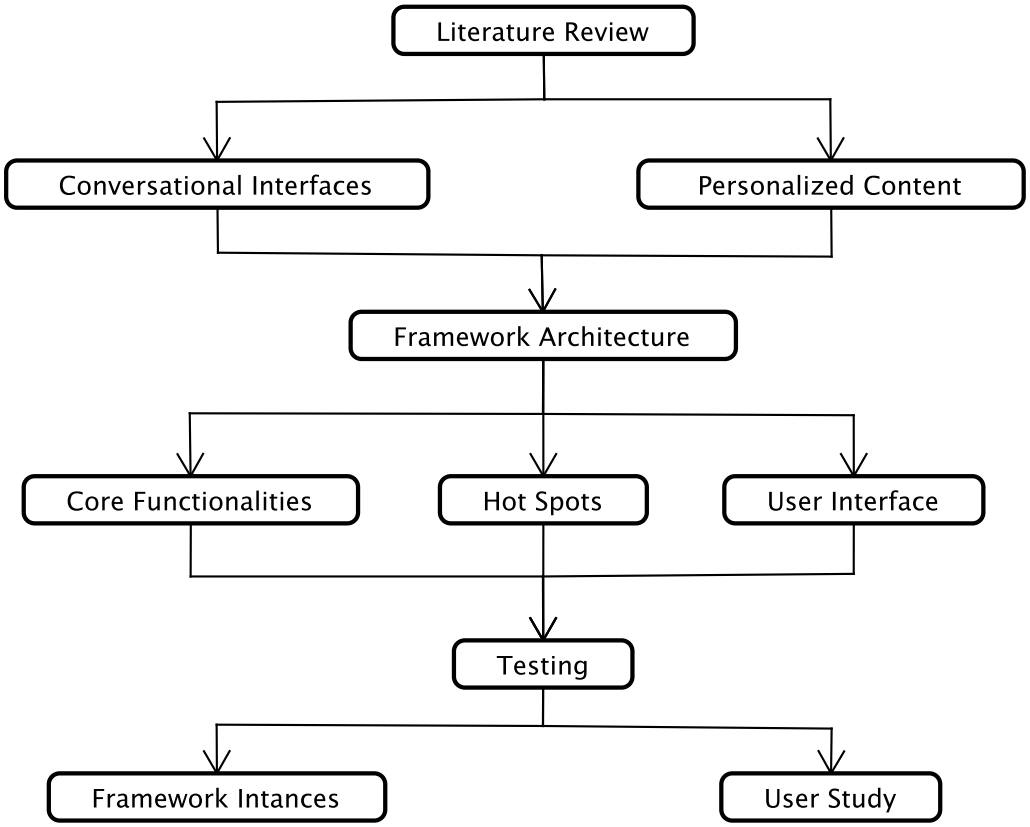


Figure 1.2: Our plan to answer the research questions.

First, we propose a literature review about personalized content answers and chatbots. After the literature review, we will create the framework macro architecture. This initial architecture might be adjusted over the course of the implementation. The architecture must comprise the framework's core functionalities and the hot spots. We also plan to integrate our conversational agent with user interfaces. Finally, we will instantiate our framework for some DS scenarios and carry out a study to check whether other developers/ data scientists can extend the proposed solution.

1.4 Contributions

There is a necessity to automate data science tasks and. To the best of our knowledge, no one proposed the usage of conversational interfaces with personalized interactions to reach the L2 level of automation. We use conversational interfaces to interact with the data scientist and adopt a strategy to learn from the user history and personalize the interactions. In particular, we are focussing on is the suggestions that the user receives from the chatbot. We extended previous related works such as (Fast et al., 2018). The authors introduce the code generation feature and the commands hierarchy to help the data scientists. We extend their approach and include pipelines and other new features such as support for some automl tasks, tracking user dialog history, and personalized suggestions. To evaluate our contributions, we report a study to analyze the use of the framework by external developers and data scientists. Participants provided suggestions and desires about new features or modifications and showed their main drawbacks while using the proposed solution, such as issues while generating code or confusion whether the data was loaded or not. Opinions, suggestions, and issues expressed by data developers about the proposed solution can encourage future works. The framework is available online at *https://pypi.org/project/dsbot*.

1.5 Document details

The remaining of this document is organized as follows. In Chapter 2, we present the main concepts to follow up with this work while. Chapter 3 summarizes recent works extracted from literature. Chapter 4 shows our proposed solution and Chapter 5 details how the proposed solution was extended and presents our study results. Finally, Chapter 6 contains a summary of the presented work and possible future lines.

2
Backgorund

This chapter presents the main concepts used in this work. In the Section 2.1 we will provide a common definition of chatbot systems and Section 2.2 presents tools to support automatic execution of data science tasks (AutoML).

2.1
Chatbots

A chatbot system can be defined as a software agent that uses natural language to provide access to data and services Følstad and Brandtzæg (2017). Chatbots are commonly used in a variety of scenarios such as health Dey and Zhang (2019); Kadariya et al. (2019); Knoop et al. (2019) e-commerce Følstad et al. (2018a); Rhee and Choi (2020) and customer support Portela and Granell-Canut (2017); Tallyn et al. (2018). According to Følstad et al. (2018b), chatbots can be characterized by two dimensions: the control of the dialogue and the duration of the relationship. The control of the conversation determines who is in control of the dialog and can be one of the following options:

- Chatbot-driven dialogue: In this category, we found bots with predefined interactions. In this case, the dialogue is somehow forced by the chatbot.
- User-driven dialogue: This kind of bot allow more flexible conversations and adapts to user input variations. In this case, the conversation is up to a certain level, controlled by the user.

Another characteristic of chatbots is the duration of the relationship with the user.

- Short-term relation: This kind of bot focus on providing interaction without a sustained relationship.
- Long-term relation: In this category, bots are prone to extract information from previous conversations and use it in further interactions with users.

In our case, we propose a Chatbot-driven, long-term relation chatbot. The interactions are bounded into a specific domain and with syntactic restrictions. So, the bot will guide the interactions. On the other hand, the

relationship between the bot and the user is refined along with multiple conversations. Therefore, the is a long-term relationship.

2.2 AutoML

According to He et al. (2019), AutoML can be defined as the automation or partial automation of a machine learning pipeline. Most machine learning pipelines contain the data preparation, feature engineering, model generation, and model evaluation stages. However, in Guyon et al. (2015), the author enhances the pipeline with a more complex vision of automation and proposes other related tasks such as automatic report generation, matching problems to algorithms, transfer learning, among others. Those complements provide a more wide comprehension of the acting field and the possibilities that came together with the automl challenge. In Yao et al. (2018), AutoML is defined as maxing the performance of a learning tool with less human assistance and limited computational budget. Therefore, it has been a hot topic in the industry, and there are several available tools on the internet such as AutoWeka Kotthoff et al. (2017), TPOT Le et al. (2020), Auto-Pytorch Mendoza et al. (2018), etc.

The basic pipeline described in He et al. (2019) contains four major steps. The data preparation step focuses on the treatment of the data. Common tasks within this step are normalization, cleaning, fill missing values, etc. This step prepares the data for further processing. In the feature engineering step the goal is to transform the data into the final usable dataset by applying dimensionality reduction algorithms, features importance algorithms, data argumentation techniques, among others. Note that in the first step, the number of features remains the same. However, in this step, the number of features can be reduced with dimensionality reduction algorithms or can be increased by generating new features. Some strategies to generate new features are data transformations to more complex representations. For instance, algebraic linear applications can transform $\mathbb{R}^n$ space into $\mathbb{R}^m$. The model generation step contains the mapping of the data into an algorithm and the tuning of the hyperparameters. Nowadays, there is a wide variety of algorithms and each one can have uncountable parameters. Most traditional algorithms such as Decision trees and SVM have a limited number of hyperparameters. However, modern neural networks can have multiple layers with different architectures. Automatic tuning of neural network architectures can deal with millions of hyperparameters, such as the case of the VGG-16 Simonyan and Zisserman (2014) with more than 130 million parameters. Finally, the model evaluation

step has determined the quality of the generated models. However, the training of several models can be a time-consuming task and there are several strategies to improve the model evaluations. For example, early stopping is a strategy used to avoid overfitting in traditional methods, and resource-aware is a strategy to monitor the consumption of resources (Memory, CPU, and GPU) and compensate with other metrics such as accuracy, precision, recall, and f1-metric. In this case, even when the model may lose performance it can gain a significant resources economy.

3
Related works

This chapter presents works related to personalized content generated by conversational interfaces. In Section 3.1 we present an overview of personalized content and provide authors to highlight the importance of personalized content in conversational interfaces. In Section 3.2 we bring a summary of selected previous works about conversational interfaces where users suggest some form of personalized content and present an examples of personalized content in conversational interfaces implementations. In Section 3.3 we show our final considerations about the analyzed works.

3.1
Overview

The authors in Chaves and Gerosa (2019) propose a survey of the literature to derive a conceptual model of social characteristics for chatbots. The authors argue that chatbots should be enriched with social characteristics to avoid users dissatisfaction. The literature reviewed in this survey strongly supports chatbots with social characteristics. The authors found three major social characteristics groups in chatbots (Conversational intelligence, Social intelligence, and Personification). Personalization is located inside the Social intelligence groups and focuses on providing personalized interactions. According to the authors, the main drawback of personalization is data privacy: the system must have personal information to adapt itself and to provide personalized interaction. However, the user's personal information, sensible and requires security and becomes a privacy concern. Personalized interactions are separated into three main advantages. First, to enrich interpersonal relationships: the user should control the amount of personal information the bots have access to. Second, provide unique services. For instance, a tour assistance system could use geolocation services to provide conveniently located places. Finally, reduce interaction breakdowns. In this case, the UI should adapt to different kinds of users. For example, the system could use larger fonts for older people.

On the other hand, Rönnberg (2020) focuses on persuasive and personalized chatbot conversations. First, the authors raise the importance of creating

personalized interactions between chatbots and users. According to Brandtzaeg and Følstad (2018) bots are generally implemented as "one-size-fits-all". However, the author presents the personalized content as a demanding challenge. The author focuses the research on the driving and safety areas. But, the importance of the challenges and guidelines presented applied to multiple areas. The author presents three dimensions of the personalizations: the parts of the systems, the target, and the automation. The first refers to the parts of the system that can be or need personalization. The second refers to the target or group of targets that the personalization address. Finally, the last dimension refers to the automation of the personalization or if it requires manual actions. The authors present the final version of the guidelines. The guidelines, as the author state, are supposed to help in the design of persuasive and personalized chatbots. According to the authors, personalization is an important part of the chatbot's design and provides a tailored user experience.

In Brandtzaeg and Følstad (2018), the authors focus on presenting needs and challenges emerging in the chatbot area. The authors present the motivation for future use of chatbots from the user's point of view. First, the authors state that chatbots are not only a change in the interface with technology but also a change in the dynamics and ways to use the technology. This implies that bots are not only new ways to get to technology as it is, but technology needs to move forward and adapt to the new tendencies coming together with the use of chatbots. To achieve this purpose, the author shows the need to improve the user and context models. According to the authors, the users need to feel in control of technology. It means being able to do tasks fast and efficiently. Finally, the author brings an important lesson learned through their research: chatbots are not a unique solution for all users because people's motivations and purposes are diverse. Instead of a "one-solution-fits-all", the author suggests multiple use cases in contexts. This is closely related to the personalized content response because it reveals the need to behave properly in different scenarios.

3.2 Articles Selection

In this section, we present researches related to personalized content in conversational interfaces. First, in Subsection 3.2.1 we gather general chatbot works where the users ask for personalized content of any form and Subsection 3.2.2 present previous approaches.

3.2.1 Conversational Interfaces

In Ghandeharioun et al. (2019), the authors present a comparison of how users interact with an emotionally expressive bot against a neutral bot. The authors design and implement an emotion-aware chatbot that answers with emotionally appropriate conversations. The authors conclude that participants showed more positive when dealing with the emotional bot. The authors added emoticons to smooth the conversations. The author conducted a study with 39 participants and identified several possible guidelines for future works. One limitation of the work is that authors scripted all textual interactions, and eventually the response of the bot became predictable. In this way, the authors propose the use of machine learning techniques to generate automatic answers enhanced by sentiments and emotions. This is an example of personalized content answers. In this case, the authors use the emotional components to determine the personalized answer for every emotion included in the research. We can map the emotion to a group of users. In this way, the system generates personalized answers for different groups of users.

The work Smestad (2018) presents a framework to design user-centered personality chatbots and how the personality influences the user experience while interacting with chatbots. The motivation of the work was to investigate to what extent personality affects user perception of chatbots. The author shows that two chatbots offering the same service and effectiveness can be rated differently based on pragmatic qualities. The framework proposed provides a stable path to build personality chatbots prototypes in conformance with personality theory. The authors conducted an experiment to evaluate the influence of personality on the user experience and came to the conclusion that it significantly enhances the user experience. As future work, the authors propose to extend the framework and include a tone analyzer, which can determine the mood of the user input and dynamically change the response. The tone analyzer could benefit from the framework to generate a response according to the personality. This future work is an example of personalized content responses, in which the bot will generate different answers for different moods.

In Neururer et al. (2018), the authors were motivated by the lack of social competence of conversation agents and what characteristics might contribute to the authenticity of the chatbots in future implementations. The authors conducted a series of interviews with experts from different areas to reach a consensus definition of authenticity in conversation agents. The main contribution of the work is a theoretical definition of authenticity through

the fusion of concepts found in artificial intelligence and social ethics. The author concludes there are several characteristics that facilitate authenticity such as having a transparent purpose, learning from experience, showing strong conversational behavior, among others. However, the authors agreed that the main point is to learn from experience. By learning from experience the author means keeping track of the conversations, behavior, and being coherent. The authors state that providing personalized information can strengthen the relationship with the user and maintain the user's engagement.

The authors in Portela and Granell-Canut (2017) present and empirical research about engagement and affection within conversational agents. The purpose of the research is to find how the users feel having empathy relations with conversational agents. To accomplish the objective, the authors conducted a series of interviews with 13 participants and two different conversational agents. The first agent was created using the traditional AIML specifying language format with predefined conversations, and the second bot was personified by a Wizard-Of-Oz strategy. The first chatbot mapped words or phrases to respond accordingly while the second bot gave more contextualized answers. The survey reflected that users' experience with the second chatbot was better than the experience with the first chatbot. The authors state that with the current level of AI technologies, conversational agents are focused on specific purposes instead of broad contextual intelligence. However, the authors conclude that including different behavior strategies in the design of chatbots can positively influence user engagement and the user experience.

The authors in Tallyn et al. (2018) present Ethnobot, a bot that collects ethnographic data and presents opportunities and challenges of collecting this information during an event. The authors conducted a study with 13 participants to gather information. The Ethnobot is an example of the user of IoT technologies to enhance user interaction, however, some of the participants felt frustrated by the understanding capabilities of the bot. Nevertheless, some participants found the chatbot pleasant company during the event. The authors demonstrate the effectiveness of the Ethnobot form collection information. However, some participants reported that Ethnobot should use their current location to guide them to more convenient places to visit. Also, some participants reported that the bot should ask for references to direct the participant to areas meeting their expectations. The information the participants reported is a close example of personalized content answers. The conversational agent should collect the personal information of the users to provide a more specialized answer.

3.2.2 Personalized Content Implementation Examples

In Dey and Zhang (2019) the authors propose a mechanism in a data processing system to provide personalized drugs response for patients. A key piece of the proposed mechanism is the drug response estimation engine, composed of a patient similarity network, a lasso regression analysis, and a patient clustering component. The author uses the drugs response estimation engine in a flow to predict drug responses for new patients. The flow begins when the mechanism receives new world information. Then, the engine processes the information and generates the group patients via clustering so there are specific drugs in each group patient. Based on these groups the mechanism can predict drug responses for new patients. The authors present the cognitive system as a general-purpose mechanism and provide a specific example in the healthcare area. The presented example uses a question/answer pipeline to receive information. However, the means to receive information may vary in different applications. Anyway, the proposed cognitive system aims to personalize the drug treatment based on similarity networks and population clustering analysis.

In Kadariya et al. (2019) the authors present kbot, a chatbot for monitoring asthma patients under 15 years old. The bot alert the patient about medication, environmental changes, and asthma management in general. The authors emphasize the need to contextualize, learn and personalize to maintain a meaningful dialog. Personalization is achieved through the conversation. In the beginning, the patient profile is constructed based on existing medical records. Within the whole process, kbot respects the data privacy and uses HIPPA compliant anonymized data. The personalization of the bot comes in two circumstances: First, by checking the medication history and found poor compliant medications, and second by checking environmental conditions that may trigger asthma reaction based on previous asthma reports. Kbot is presented as an Android application with a NoSQL database and Google DialogFlow to identify the user intentions. The author performed usability tests with 16 participants(8 physicians and researchers) and came to the conclusion that chatbots can be more effective when using patient history and domain knowledge to generate personalized answers.

This work Knoop et al. (2019) presents a mechanism to dynamically generate personalized questions for health risk assessment. The proposed system begins with a battery of questions and analyzes the results to generate the initial patient profile. With the initially gathered information, the system fits the patient into a patient group and determines the next question to present

to the patient based on the fitted patient group similarities. This process generates a loop in which patients are presented with a question, the answer fits the patient into a new patient group and using the patient's similarity inside the new group the system determines the new question to present. This loop repeats until certain conditions meet, and the systems calculate the health risk scores and present them with an uncertainty score. The authors present QA systems to communicate with the patients effectively and search through large sets of documents and records. In this case, the proposed system segment question is based on patient groups and fits patients into groups to find more personalized questions to present. The authors state that clustering and machine learning techniques can be used to generate the question segmentation database. Even when this work presents strategies to generate personalized questions, it is an example of personalized content generation for different user groups.

In Musto et al. (2020), the author presents MyrrorBot, a conversational interface based on a user profile strategy called Myrror. MyrrorBot contains two principal components, the intention recognizer, and the answer generator. the intentions recognizer receives the user request in natural language, query the Myrror user profile and send the results to the answer generator. The answer generator has a composite answer. All the answers have a static part equal to every user and a dynamic part depending on the user profile results. In this case, the dynamic part represents a personalization of the content provided to the user by the answer generator. The authors carried out a controlled experiment with 67 subjects to evaluate the proposed system and concluded that almost 90% of the participants argued that MyrrorBot provided the answer they were looking for.

The work Liu et al. (2020) shows research about mirroring during a dialogue with a conversational interface. The authors propose a technique to learn user speech patterns during the dialogue, and adapt the answers according to the learned patterns. The method consists of two steps: the n-grams extraction and the pattern injection. The n-gram extraction selects common structures used by the users while speaking and stores the n-grams for further learning. To inject patterns, the authors use two approaches: the BERT+Explicit pattern and the seq2seq neural network. The BERT+Explicit patterns inject sentences into patterns wildcard. For instance, the pattern "I said *" could be transformed into "I said take a look ...". On the other hand the seq2seq network with the original sentences without the n-gram as the dataset and the full sentence as the annotation. In other words, the seq2seq network is trained to inject the n-grams into the sentences. The authors test the

methods using a dataset of the 17k sentences in Donald Trump's 2016 campaign speeches, and the results showed that with small amounts of data the seq2seq has lower perplexity values than the BERT+Explicit pattern. However, both methods reflect the speaking patterns. Those two techniques are examples of personalized content in conversational interfaces. Both techniques could be used to mirror speaking styles back to the user to gain authenticity.

In Rhee and Choi (2020), the authors present a base-voiced conversational agent to explore the impact of personalized content on customer preferences in a shopping scenario. The authors also evaluate the impact of the different conversational agent personalities such as friends or secretaries. The personalized message is a well-known marketing strategy, and there is research result behind the personalization of advertising messages on the web. The conversational agent was simulated by a speaker placed on the participant desktop and connected to another desktop where a researcher followed a scripted conversation. The author used text-to-speech technologies to simulate a personal assistant such as Alexa or Siri. In fact, in the experiments, the conversational agent responds to "Jenny". The personalization of the messages was made individually for each participant. Before the experiment, the authors asked the participants to answer a survey about preferences and usages. With this information, the authors prepare one script for each participant with personalized content. The authors conducted experiments with 124 participants varying the role and the personalization level of the conversational agent and analyzed the data of 122 participants. The results showed a significant effect of the personalization, and participants seem to be more positive with a recommendation as a friend rather than a personal assistant.

In Sun and Zhang (2018), the authors present a framework to merge recommender systems with conversational interfaces. According to the authors, traditional dialog systems often use only the current conversation and ignore past conversations. They take advantage of past purchases of the user that can improve metrics such as conversion rate. The proposed framework has three components: the believe tracker, the policy network, and the recommender. The believe tracker transforms user inputs into vector representations called "belief". The policy network receives the user's belief and generates an answer. Sometimes, the policy network calls the recommender to receive a personalized list of suggestions for the user. The personalization is achieved through a factorization machine model that uses past information and the current conversation to generate recommendation rankings. To evaluate the proposed framework, the authors use the Yelp challenge recommendation dataset and simulate online and offline evaluations of the framework. The results showed the impact of

incorporating recommendations systems to conversational interfaces and the merits of taking action to optimize long-term rewards.

Sun and Zhang (2018) presented a framework to merge recommender systems with conversational interfaces. They take advantage of user's history to improve metrics such as conversion rate. According to the authors, traditional dialog systems often use only the current conversation and ignore past ones. Their proposed framework has three components: the tracker, the policy network, and the recommender. The policy network calls the recommender to receive a personalized list of suggestions for the user. The framework presents the recommender feature as a separate component of the solution. It interacts with other components to provide personalized content. Chittò et al. (2020) propose a framework for generating a chatbot out of a website equipped with bot-specific HTML annotations. The authors used the tree representation of the HTML elements to insert specific tags with intentions. In this way the authors create a tree of intentions linked with the HTML elements. We adopt the ideia of representing the intentions of the conversation in a tree data structure.

Ed-douibi et al. (2021) propose a framework to generate chatbots tailored to Open Data API technologies. The APIs are modeled as annotated UML schemas and then used to generate the corresponding chatbot to access the Open Data source. The generated chatbot transforms the conversation into API queries. We found this approach useful to access data sources, and adapted the proposed algorithms to our data science scenario. Pérez-Soler et al. (2021) introduced a platform called CONGA, a model-driven solution for forward and backward chatbot engineering with a recomender system to help select an adequate chatbot development tool. CONGA provides a domain-specific language (DSL) to model chatbot functionalities, a questionnarie to selected the best development tool, a code generator to build the chatbot, and an integration tool to deploy the chatbot in some social networks and applications. This platform uses the idea of code generation to build chatbots. By contrast, we will create chatbots with code generation on each intention. So, our plan is to generate code from the conversation with the bot.

3.3 Considerations

With the analysis of the previous works, we demonstrate that the need for personalized content conversational interfaces contributes to the authenticity of the chatbot (Neururer et al. (2018)), to avoid bot answers to became predictable (Ghandeharioun et al. (2019)) or to show empathy based on the

user mood (Smestad (2018)). We identify two main strategies to personalize the content. The first, used in (Knoop et al. (2019) and Dey and Zhang (2019)) tends to separate the users into clusters and provide different answers per cluster. On the other hand, Kadariya et al. (2019); Musto et al. (2020); Tallyn et al. (2018) look after specific information such as historical records or geolocation data to provide more precise content in special situations.

Our solution aims to merge, adapt, and continue insights from previous works. We plan to generate personalized content in a separate component based on the architecture proposed by Sun and Zhang (2018). Also, we take advantage of the tree structured proposed by Chittò et al. (2020) to store our conversation, and use the strategies proposed by Pérez-Soler et al. (2021); Fast et al. (2018) to generate code. None of the revised work provides native support for the L2 level of automation. The main advantage of our work over the articles mentioned above is that we provide support to personalized content as a native functionality of the framework. In addition, we create pipelines, responsible for gathering a group of commands. Based on the previously analyzed techniques, we will present our proposal to use conversational interfaces with personalized content to reach the L2 level of automation in DS/ML projects. Level L2 is the main goal of our work, given the research carried out in (Wang et al., 2021), to which several experts in the field of data science replied that they would like to receive suggestions during the DS/ML process.

4
Proposed Solution

This chapter presents our proposed solution. First, in Section 4.1, we present an overview of its features and then we go into details about how they work and communicate with each other (Section 4.2 and Section 4.3). Finally, in Section 4.4, we present the solution as a framework.

4.1
Overview

The proposed solution evolved through planning and implementation iterations. Figure 4.1 shows how we add features to our solution over time. First, we created a module capable of identifying the user's intention: as the conversation flows, the system maps text input with a predefined set of intentions.

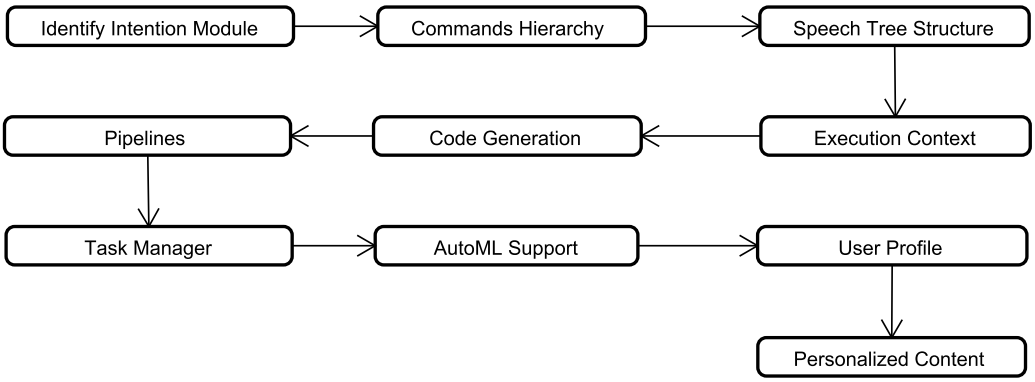


Figure 4.1: Features timeline.

The commands hierarchy module collects the base classes and core functionalities of every command. The speech tree structure stores a conversation with the user and transforms the conversation into a tree-based data structure. The context feature saves all the variables during the execution of the commands. The code generator exports all commands into usable python code and pipelines merge several commands into one. The task manager separates the commands execution thread from the user interface. The proposed solution supports automl features to enhance pipeline usage. The system creates a user profile through the user inputs. Finally, the personalized content module use the information collected in the user profile to provide suggestions.

4.2 Architecture

The proposed system has four modules (Command Manager, Speech Manager, ChatBot, and User Profile). Figure 4.2 depicts how the features are organized in modules.

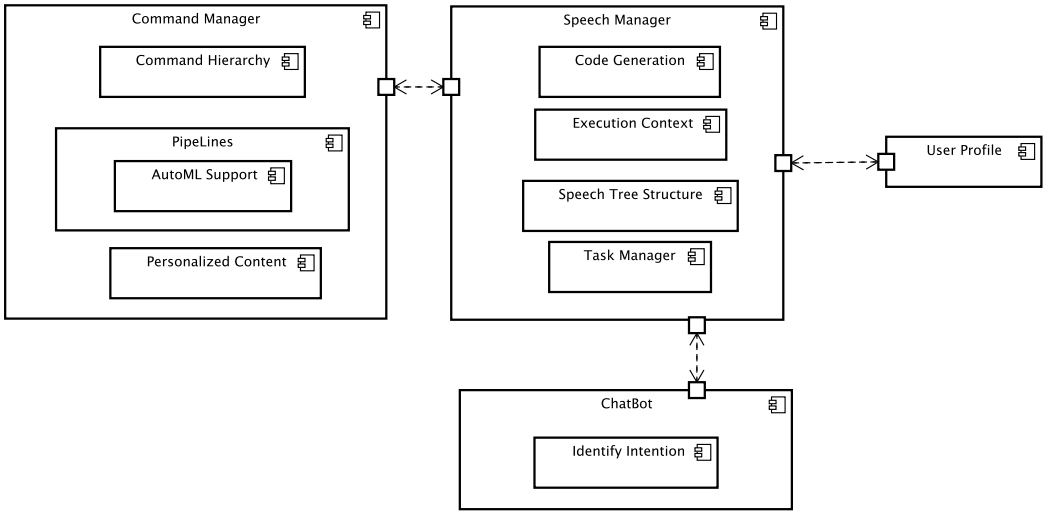


Figure 4.2: Architecture of the proposed solution.

The *Command Manager* module includes the Command Hierarchy, the Pipelines with the AutoML support, and the Personalized Content features. The ChatBot module contains the Identify Intention feature, but the User Profile Module is a feature by itself. Finally, the Speech Manager includes the Code Generation, the Execution Context, the Speech Tree Structure, and the Task Manager features. The Speech Manager Module connects all other modules and works as a command center and entry point of the solution as illustrated in Figure 4.3.

The *Speech Manager* receives user messages. A message is sent to the Chatbot to identify the user's intention. With the user's intention, the Speech Manager calls the Command Manager to create an instance of the specified command and organize it into the tree structure. Finally, the speech manager executes the command to return the proper response to the user. Commands execution is asynchronously handled inside the speech manager by the Task Manager feature, so each sequence of events is independent.

4.3 Implementation details

In this section, we are going to present implementation challenges we went through the construction of each feature.

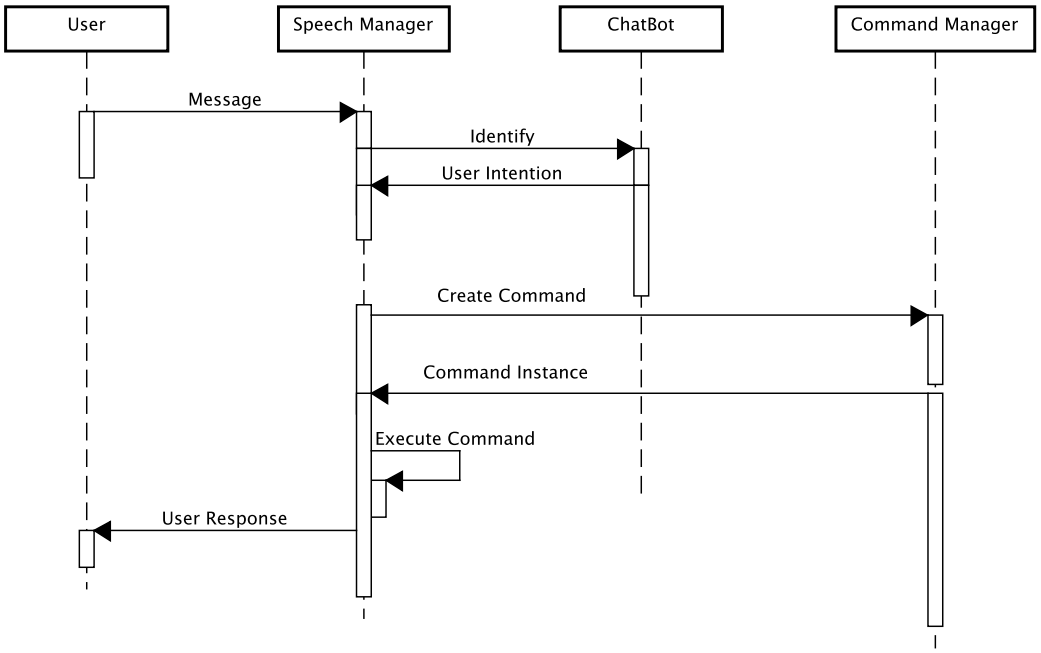


Figure 4.3: Sequence of events triggered after a new user input message.

$$\overbrace{Lets\ load\ a\ dataset}^{x} \mid \overbrace{load\_dataset}^{y}$$
$$\overbrace{load\ the\ dataset}^{x} \mid \overbrace{load\_dataset}^{y}$$
$$\overbrace{load\ a\ csv\ file}^{x} \mid \overbrace{load\_dataset}^{y}$$

Figure 4.4: Example of phrases dataset.

4.3.1 Identify Intention Bot

The ChatBot module contains the Identify Intention feature. It is in charge of recognizing what the user wants with each input. We define the problem of identifying the user's intention as a supervised learning classification problem. On system startup, the chatbot module creates a dataset of phrases and maps each phrase to a single intention. For example, the sentence "Let's load a dataset" will be mapped onto the *load_dataset* intention. We have a finite number of intentions, one for each command available on the chatbot. So, as the number of commands grows up, the complexity of this task grows as well. Figure 4.4 shows how the intentions dataset was built. The x represents the learning data while the y holds the labels.

We preprocess our phrases dataset to build a classifier and detect the user intention as shown in Figure 4.5. First, we apply a tokenization strategy to split the sentences into vectors. Then, we remove the stop words and apply an orthographic correction. However, in some scenarios like system paths or

dataset references, we should avoid orthographic correction.

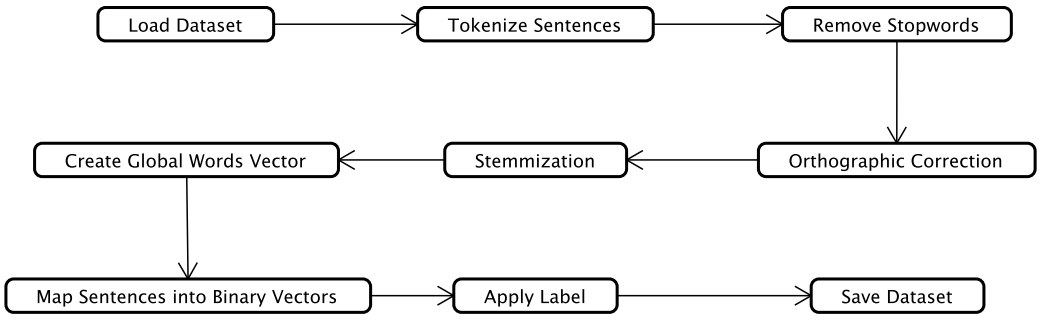
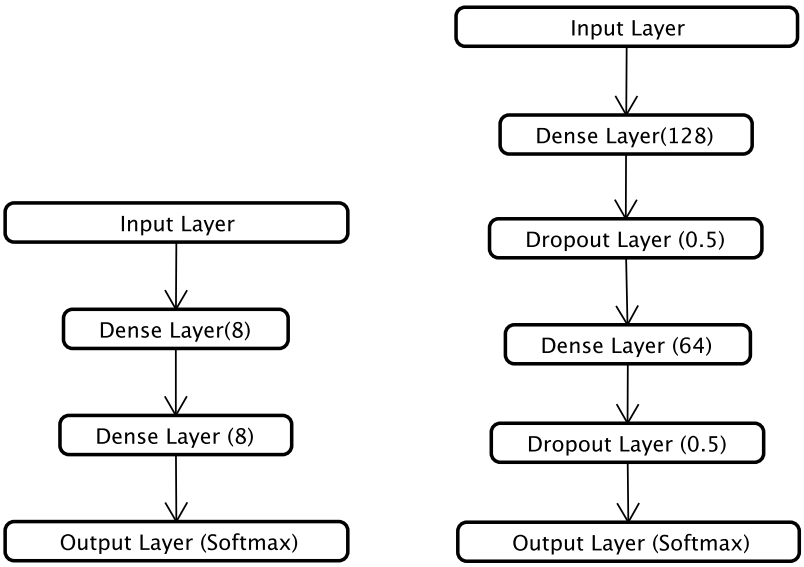


Figure 4.5: Sequence of dataset preprocessing steps

We applied a stemming process to keep the root of the words. Next, we create a global words vector and transform each sentence by setting to ones the indexes of its words. So, we create a vector of zeros with the length of the global words vector and change to one the position corresponding to the words of the sentence. In this way, we represent a sentence as a binary vector. Then we map each vector to its classification y and save the dataset for further learning. We use neural network models described in Figure 4.6a and Figure 4.6b to train classifiers.

Figure 4.6a shows the first attempt to classify the user's intentions. This model was usable for the first stages of the project, but as the number of commands grew, we had to migrate to our second model presented in Figure 4.6b. The ChatBot module contains a trained model to classify each user input. However, note that the last layer of both models is a softmax layer. Therefore, the classifier will return the probability of the user input belonging to every possible intention. It means that the classifiers will not return the intention directly. However, the bot uses a threshold value to compare the softmax probabilities and returns a valid intention only when any probability value is greater than the predefined threshold. In case no probability is greater than the threshold, the bot will return a wildcard text and the *no_understand* state to the speech manager. Summing up, the bot will preprocess every user entry and transform it into a valid binary representation. After the transformation, it will predict the probabilities of every possible intention and finally compare against a threshold value to return the most probable intention. If two probabilities are greater than the threshold and very close to one another, the bot will return the greater probability.



(4.6(a)) Initial classification model

(4.6(b)) Classification model with drop out layers

Figure 4.6: Models used for classification

4.3.2 Commands Hierarchy

In this subsection, we show how we transform the user intentions into commands, explain how we group them by categories, and detail the class hierarchy behind them.

Fast et al. (2018) created a mechanism to capture triggers and complement baseline commands. For example, when the system identifies a new input for using a command, the system adds this new input to the triggers collection of the selected command. Therefore, the DSL and the trigger-enhanced mechanism provide a base transformation from functions to commands. So, we create an initial mapping of functions to commands and complement the triggers with new inputs. Once we include a new trigger phrase, we must retrain our identify intention model. Figure 4.7 shows some user intentions separated by categories.

We created seven categories to group the commands: Base, Context, Algorithms, Preprocess, Visualization, Pipes, and Metrics. In the Base category, we put base classes of the hierarchy and commands related to a basic conversation (greetings and goodbye). The Context category gathers commands related to the running environment. For example, the *LoadDatasetCommand* accesses the file system to load a CSV file. The Algorithms category contains the implementations of different algorithms, and the Metrics category has commands to evaluate the algorithm results. The Pipe category has our

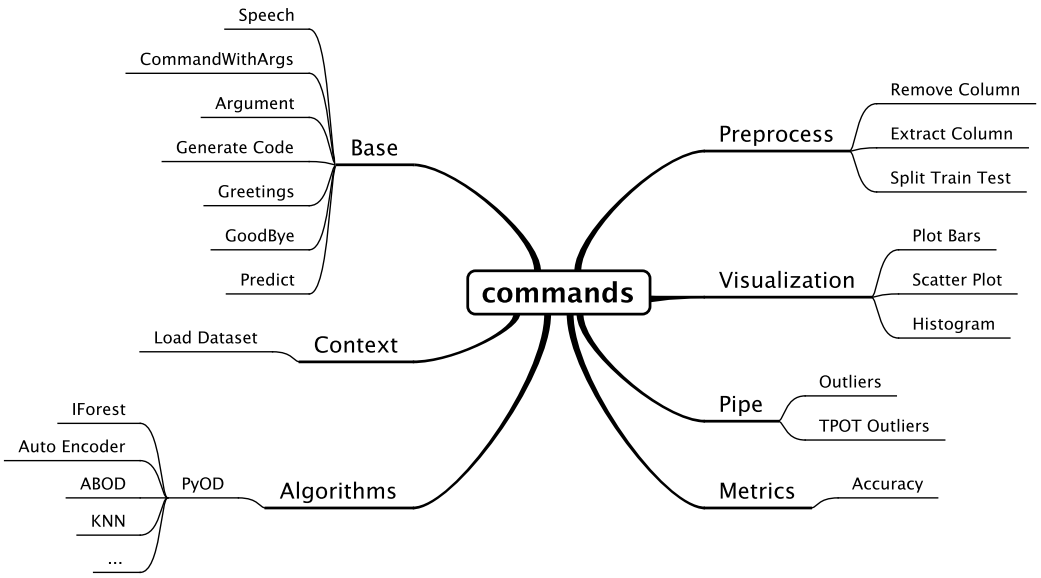


Figure 4.7: Commands separated by groups.

implemented pipes, and the Visualization category presents commands to generate charts. Finally, the Preprocess category has the tools to work with the datasets after the loading stage.

The base category contains the super classes of the commands hierarchy:*Command*, *CommandWithArgs*, and *Argument*. The *Command* class represents all the commands, while the *CommandWithArgs* represents all commands with arguments. For instance, the *LoadDatasetCommand* requires two arguments: the path of the dataset and the name. Note that there are a few commands that do not require any parameters, like *GreetingsCommand*. Finally, the *Argument* class represents the arguments of the commands. Figure 4.8 detailed the methods and properties of these classes.

In particular, the *Argument* class uses regular expressions to capture the value of the arguments. Figure 4.9 show an example.

The complete user input "Let's load the dataset outliers_sample in data/outliers.csv" is identified with the *load_dataset* intention. The system transforms the *load_dataset* intention into a *LoadDatasetCommand*. The *LoadDatasetCommand* has two arguments (dataset_name and dataset_path). Each argument is an instance of the Argument class. In this case, the dataset_name uses the `dataset ([a-zA-Z0-9_]+)` regular expression to extract the dataset name, while the dataset_path uses `at ([a-zA-Z0-9_]+)` to select a file path. Arguments can have multiple regular expressions attached and will evaluate all until one finds a match. For example, the dataset_path argument contains three regular expressions: `in ([a-zA-Z0-9_.]+)`, `in the file ([a-zA-Z0-9_.]+)` and `dataset_path = ([a-zA-Z0-9_.]+)`.

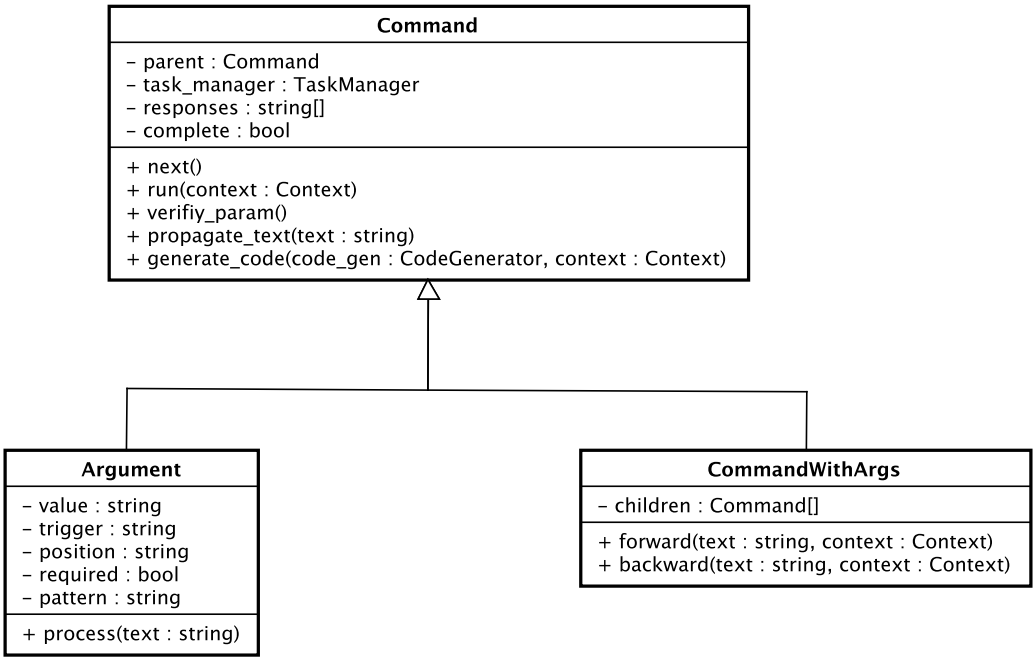


Figure 4.8: Base classes of the commands hierarchy

$$Lets\ load\ the\ dataset\ \overbrace{outliers\_sample}^{dataset\_name}\ in\ \overbrace{data/outliers.csv}^{dataset\_path}$$

$$Lets\ load\ the\ dataset\ \overbrace{iris}^{dataset\_name}$$

Figure 4.9: Example of arguments in user inputs.

Arguments can be optional, such as the dataset_path argument of the *LoadDatasetCommand*. In the sentence "Let's load the dataset iris", the dataset_path is missing. However, the iris dataset is a well-known dataset. The *LoadDatasetCommand* commands contain a collection of well-known datasets. In these cases, the command does not require a path to load the dataset.

4.3.3 Speech Tree Structure

This subsection describes how commands create a tree data structure and how we iterate over a conversation. All commands inherit from the base *Command* or *CommandWithArgs* classes and contain a list of children commands. Also, every command has a parent property referencing their parent command. These two properties (children and parent) create a tree structure during a conversation.

In particular, we use an Abstract Syntax Tree (AST) to capture the meaning of the conversation because it can formally represent the critical structure of the input while avoiding syntactic resources (Jones, 2003). AST

implementations benefit from Object Oriented Programming (OOP) languages due to inheritance and polymorphism. In this case, all the AST nodes derive from base classes that share common attributes. A useful pattern to complement AST and OOP is the Builder pattern. The builder pattern allows the expansion of the AST dynamically with new nodes.

The Tree Structure of a conversation contains three levels. The first level is the *SpeechManager*, the second level includes the children of the *SpeechManager*, which are *Command* or *CommandWithArgs* instances, and the third level contains *Argument* instances. Figure 4.10 shows a speech tree example of the following conversation.

- User: Hi
- Bot: Hello
- User: Lets load the dataset outliers_sample_data in data/outliers.csv
- Bot: Done
- User: ...
- Bot: ...
- User: Bye
- Bot: Bye

In the beginning, there is only the *SpeechManager* node in the tree. This node is the root and has no parent nodes. After the first user input "Hi", the speech manager identifies the *greetings* intention and creates a child node *GreetingsCommand*. The *GreetingsCommand* has no arguments, so it has no children of its own. When the user asks to load the dataset, the speech manager identifies the *load_dataset* intention and appends a *LoadDatasetCommand* child. The *LoadDatasetCommand* has two arguments extracted from the user input and creates two *Argument* children nodes (one for each argument). Finally, when the user says "Bye", the *SpeechManager* recognize the *goodbye* intention and creates a *GoodbyeCommand* node.
In this example, the first level of the tree has the *SpeechManager*, the second level contains the *GreetingsCommand*, *LoadDatasetCommand*, and *GoodbyeCommand* commands, and the third level has the arguments.

The Speech Manager has a context to save execution variables and data. This context is empty at the beginning of the conversation, but, as soon as commands execute, it holds the variables and the results. Each *Command* has a property named complete, and it notifies the *SpeechManager* when it is ready to execute. Commands execution is always children first and left to right. For instance, When the user says "Hi", the *GreetingCommand* generated has no arguments, so it is complete and able to run. However, the *LoadDatasetCommand* is ready only when all its children are ready. So,

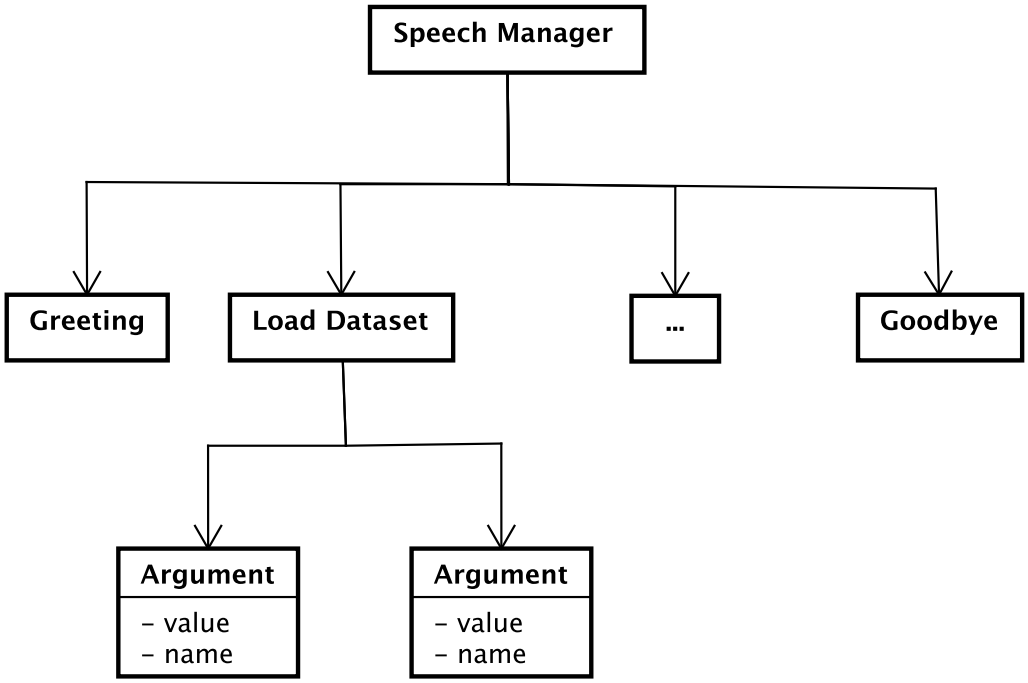


Figure 4.10: Example of conversation into tree data structure.

it depends on the completion of their *Argument* children. In this case, the arguments are in the same user input. In the case where required arguments are missing, the bot asks for them, and the *SpeechManager* saves the last non-complete children. The bot will resume the execution once the required arguments are with proper values. When the *LoadDatasetCommand* loads the dataset in memory, it saves a reference in the context object for future commands.

4.3.4 Pipelines

In this subsection, we describe pipeline objects. Firstly, we show how we group commands into pipelines. Then we show how pipelines are represented in the speech tree structure and finally some execution details. In (Fast et al., 2018), the authors present an interactive data science tool and argument that extensive data science tasks would require structured pipelines of commands. Extensive data science tasks can take days to complete. Pipelines provide an instant follow-up after each command and visualize only partial or final results.

Pipeline commands are a collection of other commands, and by definition, a Pipeline can not be recursive, so it can not have a child of the same class type. For example, in the case of outliers detection, we created the *OutlinePipe* command. The *OutlinePipe* command has the following activities presented in Figure 4.11.

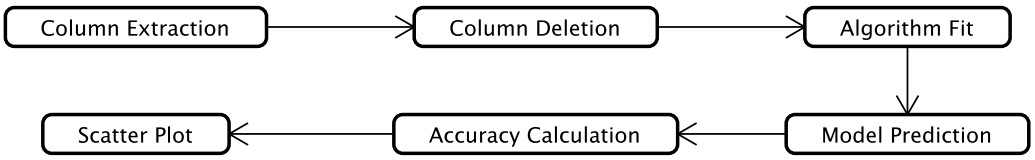


Figure 4.11: Activities of the OutlierPipeCommand.

All the activities grouped inside the OutlierPipe command are commands (Figure 4.8). The ColumnExtraction and ColumnDeletion commands belong to the Preprocessing group, and the Algorithm and Prediction are inside the Algorithms Group. The Accuracy belongs to the Metrics group. Finally, the ScatterPlot is located in the Visualization group.

- User: Hi
- Bot: Hello
- User: Lets load the dataset outliers_sample_data in data/outliers.csv
- Bot: Done
- User: Do an outlier detection analysis in the dataset outliers_sample_data
- Bot: Done
- User: Bye
- Bot: Bye

The dialog above is an example of pipeline command usage. When the user says "Do an outlier detection analysis in the dataset outliers_sample_data", the Speech Manager recognizes the outline_pipe intention and creates an instance of the *OutliersPipe* command. The *OutliersPipe* command is located between the *LoadDataset* command and the *Goodbye* command, as shown in Figure 4.12.

The *OutliersPipe* command contains six children nodes, one for each activity listed in Figure 4.11. Notice that with the addition of pipes, the levels defined for the speech tree structure change. The first level remains only with the SpeechManager, and the second level now can contain instances of *Command*, *CommandWithArgs* and pipes. Therefore, the third level, which only had *Argument* instances, can have *Argument*, *CommandWithArgs*, *Command* and other pipes. Notice that a pipe can not contain itself as a child, but it can contain other pipes.

The execution of a pipe depends on their children nodes. However, the arguments of the commands inside a pipe are not specified by the user. In this case, the pipe generates the values of their children's commands. For example, the Algorithm Fit and the Model Prediction are subsequent activities of the *OutliersPipe* command. After the execution of the Algorithm command, the

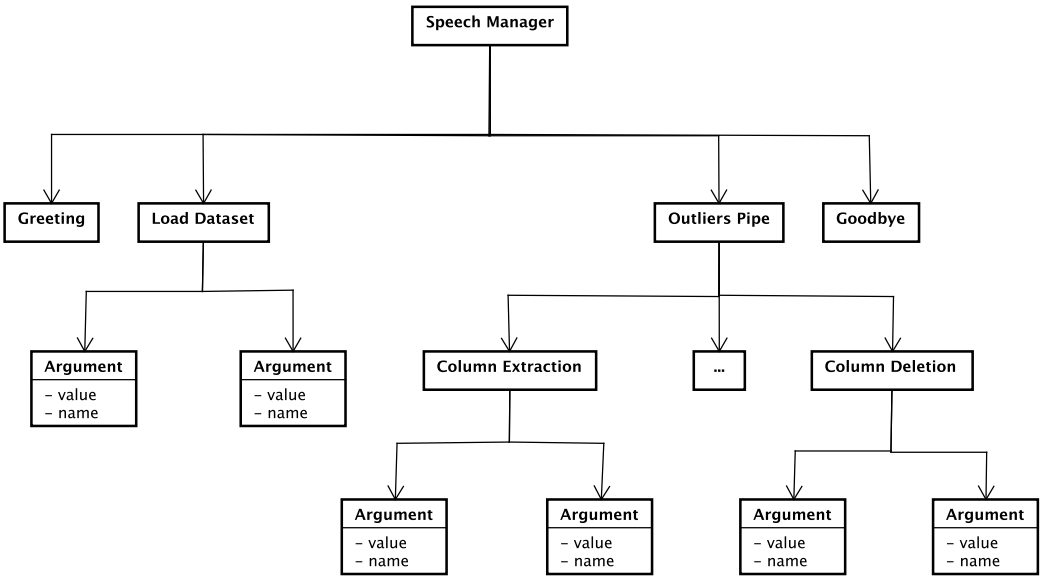


Figure 4.12: Example of speech tree with pipe.

OutliersPipe saves the result in the execution Context and passes the reference of the model to the Model Prediction command. In this way, the *OutliersPipe* command links its children commands for a straightforward execution.

The pipes can test several algorithms and select the one with the best results. But we needed to try several combinations of parameters as well. To achieve our goal, we support some automl tasks inside the system to complement the pipes. We found several possible automl packages available on the internet: TPOT, auto-sklearn, autoPyTorch, and Hyperopt-Sklearn. However, they are all specific and build upon existing libraries. For example, Auto-sklearn performs automl tasks with the scikit-learn library algorithms while the autoPyTorch uses pytorch networks. We need a tool capable of extent functionalities outside those libraries and adapt to new algorithms. We selected the TPOT library to execute automl tasks within the system. The TPOT library is based on the scikit learn library, but it can be adapted to other algorithms which maintain the same programmable interface and methods. So we could adapt the TPOT library to other algorithms outside the scikit-learn library such as the PyOD library.

We created the TPOTCommand, which has access to the TPOT optimization algorithms and extends the pipe base class. In this way, the TPOT command can integrate with the system while executes the TPOT algorithms to find the best arguments combination.

4.3.5 Task Manager

The task manager deals with the execution thread in the application. Some commands demand high computing resources, and the execution time became greater than acceptable to hold the user waiting for an answer. In this case, we decided to separate a thread for the conversation and other threads to execute the *Command* instances. A *Command* instance sometimes depends on previously executed commands and uses results from other commands. So, we created an execution pool where all the tasks are gathered and organized for execution. Figure 4.13 details the members of the *TaskManager* feature.

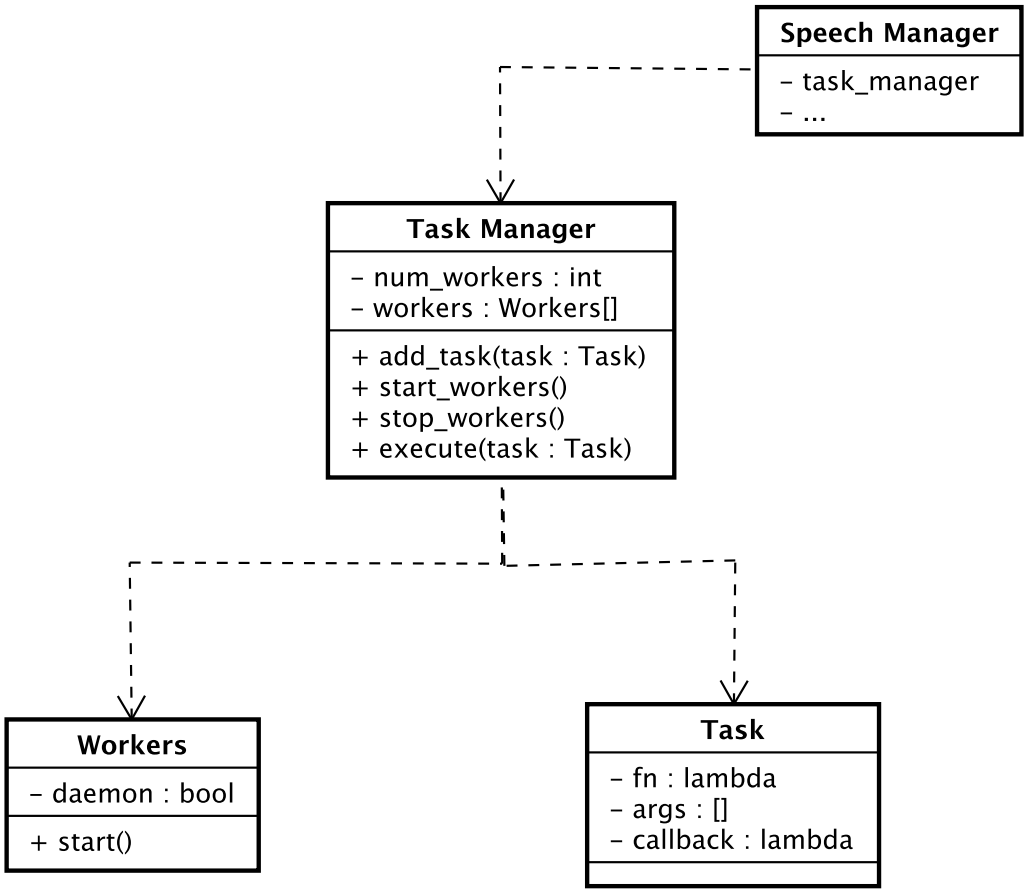


Figure 4.13: Class diagram of the Task Manager.

The *Task Manager* has a collection of *Workers*, capable of running as a background execution thread. It returns the result after the *Task* completion. The *Task* class is a definition of a job to be executed. It contains a function to execute, the arguments, and a callback to notify the results. The *TaskManager* inherits from the basic *Queue* and contains a queue of *Task* and a group of *Workers*.

The *Speech Manager* contains an instance of the *Task Manager* available to all its *Command* children. The sequence of events triggered after a new user message Figure 4.3 changes a little bit. From now on, the *Speech Manager* creates a *Task* instance with the execution of the child *Command* embedded and sends an asynchronous message to the *Task Manager* to run the *Task*. The *Task Manager* adds the received *Task* instance into its processing queue, and the *Workers* will execute it at some point. Notice here that the Speech Manager returns to the user with a resulting promise. When the *Task* ends, the *Speech Manager* will receive a notification via *Task* callback and present the actual results to the user.

4.3.6 Code Generation

In this section, we introduce the code generation feature. The code generation feature allows the system to generate usable python code from the speech tree structure. The user triggers the generation of the code through a command. The *CodeGenerationCommand* inherits from the *CommandWithArgs* base class and contain one *Argument*. The required argument is the path to save the generated code file. When the user asks for code generation, the *SpeechManager* recognizes the *code_generation* intention and sets the *Task* as it would normally do. The code generation begins with the execution of the *CodeGeneration* command. This command goes up in the speech tree structure and calls the generate_code method from the top of the tree. Figure 4.14 presents the *CodeGenerator* object.

The *CodeGenerator* has the code and the imports as a text attribute and support operations to handle python code (indent and dedent). The *CommandManager* module contains an instance of the *CodeGenerator* passed as a parameter in the generate_code method to every command to generate their code blocks. The code generation follows the same execution order and uses the context to access variables and data. For instance, the *LoadDatasetCommand* code generation applies the following steps detailed in Figure 4.15. Initially, the command writes a code comment to set up the dataset loading code block. All commands generate comments and separate their code into blocks. Next, there come the general usage imports such as numpy and pandas. The *LoadDatasetCommand* verifies if the dataset is a well-known dataset available in a package or a dataset loaded from an external file.

In the case of a known dataset, the command generates the required imports of the dataset and uses the load function provided by the package to use the dataset. When dealing with external files, the command uses a read

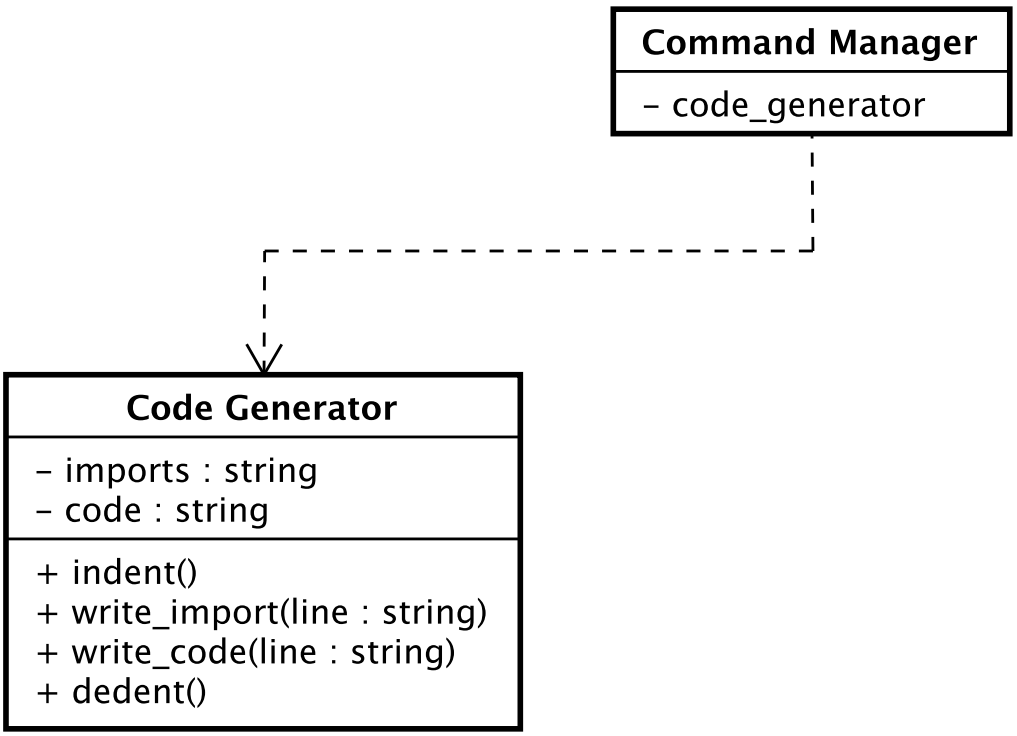


Figure 4.14: Class diagram of the Task Manager.

function provided by the pandas library to read the file. Notice that the path of the dataset is the dataset_path *Argument* of the *LoadDatasetCommand*.

4.3.7 User Profile

In this subsection, we present the user profile feature. We explain the structure used to save a user profile and how we update it after each interaction with the system. We categorized all available commands into eight groups as shown in Figure 4.22: preprocessing, algorithm, visualization, pipe, metrics, code generation, speaking, and others. The preprocessing group contains commands related to the data manipulations such as the *ColumnsExtractionCommand*, and the algorithm category has commands related to algorithm usage like *KNNCommand*. The visualization category gather commands related to charts, statistical plot and other commands with visual feedback. The pipe group has all the *PipelineCommand* instances, and the metrics group has the commands related to the metrics. The *CodeGenerationCommand* has its category, and the speaking group contains the commands needed to maintain a conversation with the bot. The others category comprises all the commands that do not fit into any other category.

The created categories summarize a group of actions. With continuous use of

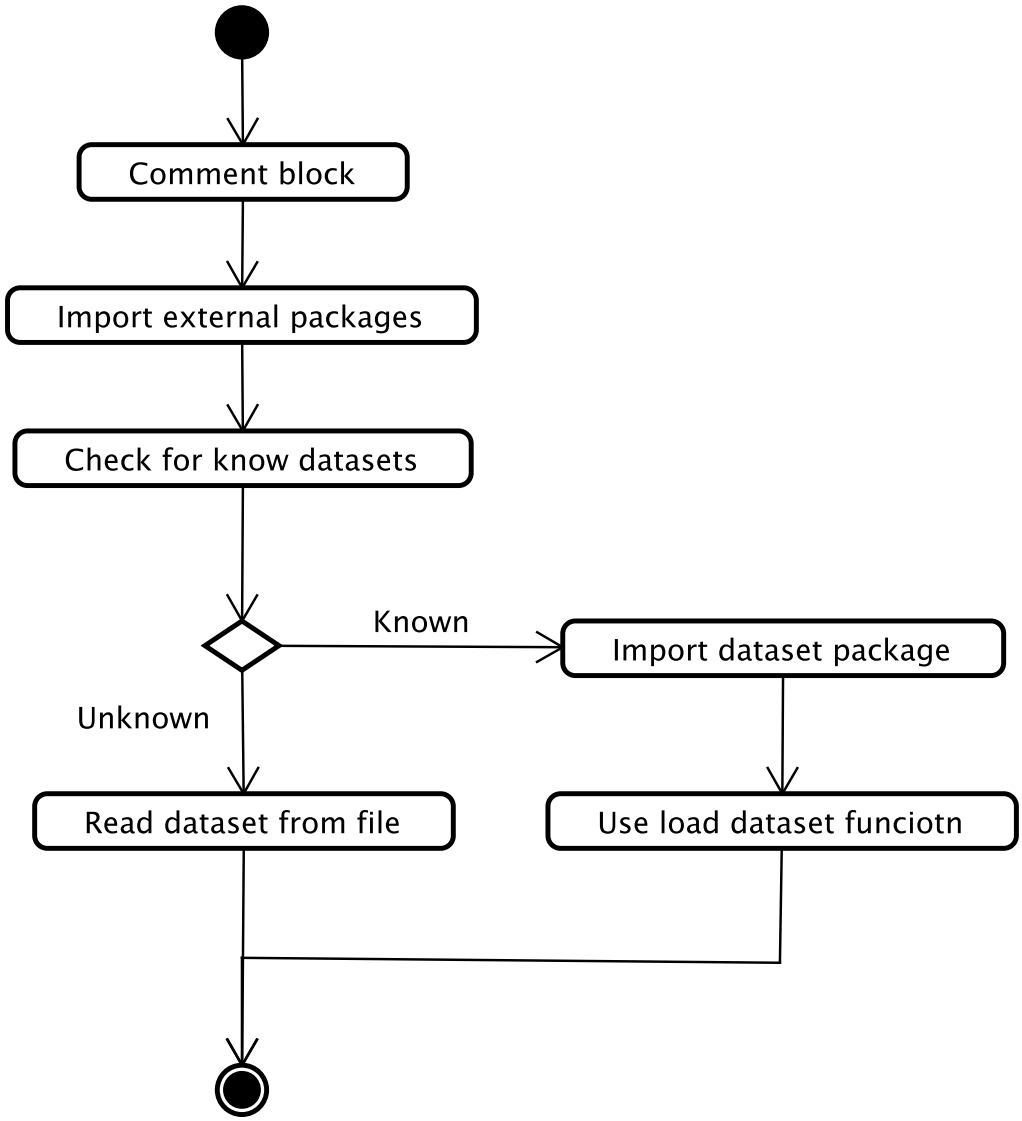


Figure 4.15: Code generation example of the LoadDataset command.

the system, each user may apply the commands inside a category more than the rest. In this case, we assume that the user has a preferred category. We create a structure to log the commands usage and ponder the categories presented in Figure 4.17.

Initially, all categories receive the same initial value, and the sum is equal to 1. This structure persists across multiple executions of the system. In this way, the user preferences reflect the user's continual usage. When the user executes a command, the system updates the user profile with a predefined rate. Algorithm 1 detailed the steps to update the user profile.

Algorithm 1 receives the current user profile and the last command executed. In the first line (2), the algorithm saves the command, and in line 3 it increases the value of the category by the update rate (0.01). The algorithm normalizes

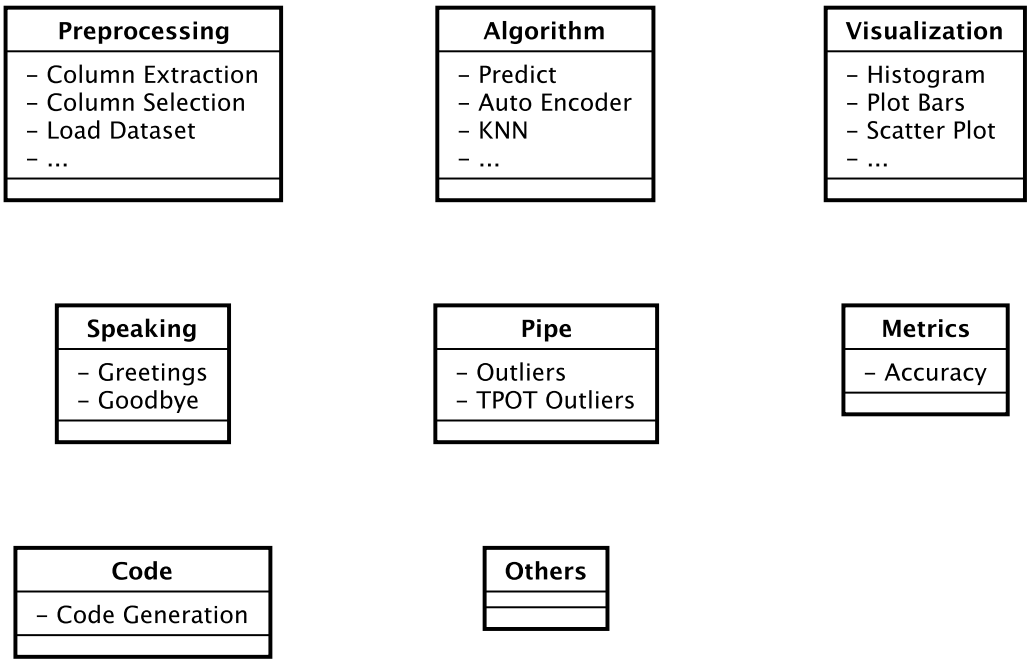


Figure 4.16: User profile types and their related commands.

$$user\_profile \leftarrow \begin{cases} Preprocessing = 0.125 \\ Algorithm = 0.125 \\ Visualization = 0.125 \\ Metrics = 0.125 \\ Speaking = 0.125 \\ Pipe = 0.125 \\ Code = 0.125 \\ Other = 0.125 \end{cases}$$

Figure 4.17: Example of arguments in user inputs.

Algorithm 1 Update user profile with a new entry

```
1: procedure UPDATE_PROFILE(user_profile, last_command)
2:     last_command_category ← last_command.category
3:     user_history[last_command_category] += 0.01
4:     index ← 0
5:     total_sum ← sum(user_history)
6:     while index < len(user_history) do
7:         current_command ← user_history[index]
8:         current_command.value ← current_command.value/total_sum
9:     end while
10: end procedure
```

the values from lines 6 to 9, so the sum of all categories is back to 1.0.

For example, after the creation of the user profile, assume that the first command is a *LoadDataset* command. The algorithm (in line 3) updates the value 0.125 to 0.126. Lines 6 to 9 normalize the rest of the categories, so the sum scores one. Figure 4.18 presents the values of each category after the update.

$$user\_profile \leftarrow \begin{cases} Preprocessing = 0.1257... \\ Algorithm = 0.1237... \\ Visualization = 0.1237... \\ Metrics = 0.1237... \\ Speaking = 0.1237... \\ Pipe = 0.1237... \\ Code = 0.1237... \\ Other = 0.1237... \end{cases}$$

Figure 4.18: Example of user profile after the first update by the *LoadDataset* command

Notice how the execution of one command affects the values of other categories.

4.3.8 Personalized Content

In this subsection, we describe a process to personalize the bot for a particular user. The idea is to learn from the user history like previous related works such as (Kadariya et al., 2019; Musto et al., 2020; Tallyn et al., 2018). Our goal is to reach the L2 level of automation in data science and machine learning projects. Therefore, the bot does not have the autonomy to execute commands by itself but commands given by the user. It means that the bot will not take the initiative, but it can learn the most frequent command sequences and propose a continuation of the analysis. For example, a user often uses the *PredictCommand* after training a model through the *AlgorithmCommand*. In this case, the bot could present the prediction results when the user asks for the *AlgorithmCommand*. However, our system does not execute commands proactively and will not execute the *PredictCommand* in this case. The bot can use this knowledge differently. The system can provide suggestions to the users. The suggestions execute when the user asks for them in compliance with the L2 level of automation. The difference remains in the fact that the bot remains reactive to the user commands. If the user wanted both commands executed together, they would merge both actions into one command through a pipeline. For example, the user could create a pipe command with *AlgorithmCommand* and *PredictCommand* commands.

$$\overbrace{greetings,\ load\_dataset,\ ...,\ predict}^{Last\ five\ commands},\ \overbrace{0.123,\ 0.153,\ ...,\ 0.138}^{user\_profile}\ |\ \overbrace{plot\_bar}^{clf}$$

Figure 4.19: User history dataset.

The system builds a dataset with the command history and the user profile to generate suggestions. This dataset holds the most recent command chains of the conversation. Figure 4.19 shows a row of the dataset. The row comprises a fixed number of previous commands, the current user profile, and the current command. It reads as follows: A user with this kind of profile executes the current command after these previous commands. In this way, each new command turns into a row of the dataset. This dataset stores personalized information of every user and interactions with the system. We created a DecisionTree model with the user dataset and saved it for later predictions. This decision tree model is used in the *PersonalizationManager* class as shown in Figure 4.20.

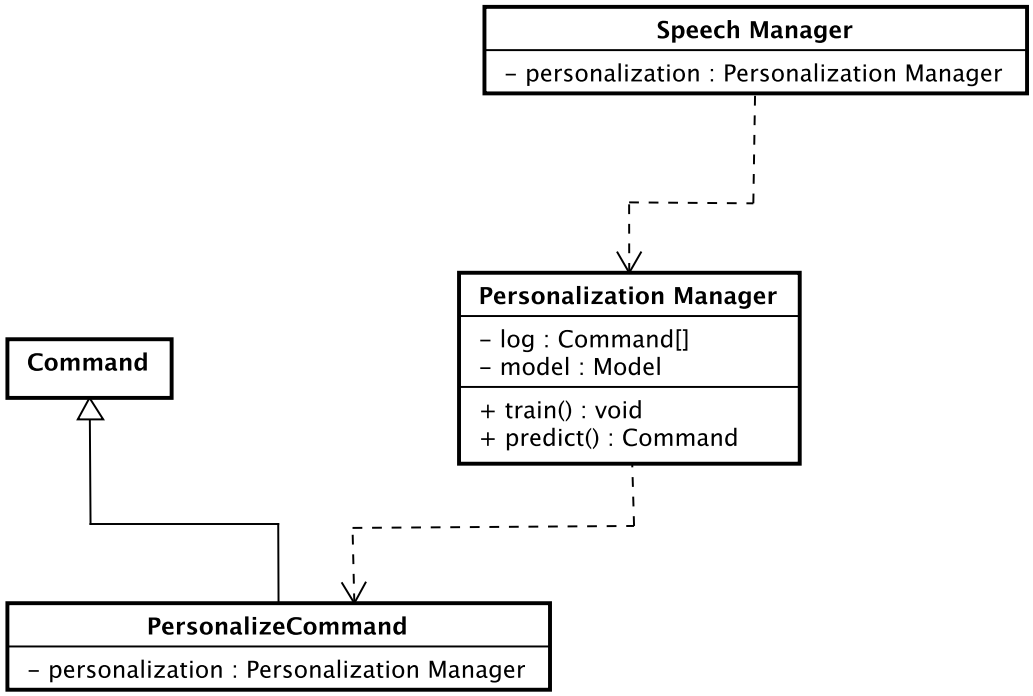


Figure 4.20: Personalization manager types and their related commands.

The *SpeechManager* passes an instance of the *PersonalizationManager* to the tree structure inside the execution context. The *PersonalizationManager* is in charge of providing personalized suggestions for the user through model predictions. We add the *PersonalizeCommand*, which inherits from the *Command* class. The *PersonalizeCommand* is the intended way to access the *PersonalizationManager* by the user. For example, in the following conversation, the

user asks for a suggestion.

- User: Hi
- Bot: Hello
- User: What can I do?
- Bot: Load a dataset.
- ...

When the user asks: "What can I do?", the bot identifies the *personalized_suggestion* intention and creates an instance of the *PersonalizedCommand*. The *PersonalizedSuggestionCommand* inherits directly from *Command* instead of *CommandWithArgs*; therefore it does not require arguments. It creates a row for prediction with the previous commands plus the user profile. In this case, we represent the row as follows:

$$\overbrace{null,\ null,\ ...,\ greetings}^{Last\ five\ commands},\ \overbrace{0.123,\ 0.153,\ ...,\ 0.138}^{user\_profile}$$

Figure 4.21: User history dataset.

We use the row described in Figure 4.21 as input parameter to predict further commands. In this case, the model predicts the *load_dataset* intention. The *load_dataset* intention belongs to the *LoadDatasetCommand*, and the *PersonalizedCommand* can finally suggest the *LoadDatasetCommand* to the user.

4.3.9 Telegram Integration

This subsection presents how we connect our conversational agent to the Telegram API interface. The Telegram API (*https://core.telegram.org/bots/api*) offers several services to execute a chatbot inside the Telegram environment. This environment includes multi-platform user interfaces for mobile, computer, and tablet applications. To implement our solution, we used the library python-telegram available in *https://pypi.org/project/python-telegram-bot/*. Among the services offered by the Telegram API, we can find gaming, message editing, among others. We implement the following services in our chatbot: receive text, send image, send text, and inline keyboard.

The text services (receive and send a text) are the base of our conversational interface and solve most of the user's needs. By combining these two services, bots can receive user input, process it, and respond with a text message. However, some user inputs should return charts. In these cases, the bot uses the *send image* service to send an image. The inline keyboard allows mul-

tiple selections through the Telegram interface and can request confirmation dialogs.

The bot also includes a list of specific commands. Commands are words preceded by a slash, for instance, */start*. The commands allow the user to interact with the bot in a different scope and adjust the bot settings. Notice that commands do not affect the current conversation history. We implement the following commands:

- /hi to check if the bot is online.
- /spelling to enable/disable the spelling corrector.
- /learn: to map a phrase with a new intention. In this case the phrase is added to the dataset for future learning.
- /suggest: to ask the bot for recommendations about how to continue the analysis.

All previous commands will facilitate a more flexible conversational interface.

4.4 Framework Overview

This subsection presents the proposed solution as a framework and highlights the fixed and variable components. Figure 4.22 illustrates a framework overview of the solution.

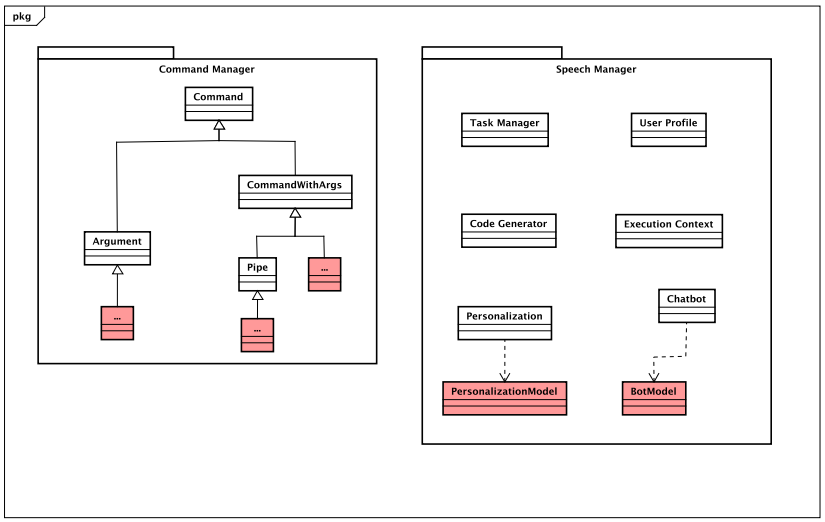


Figure 4.22: Overview of hot spots.

The components with a white background are static, while the ones with a red background are variable. In this case, the commands base classes (*Command*, *CommandWithArgs*, *Argument* and *PipelineCommand*) are parts

of the core of the solutions. However, new commands can extend those commands according to the situation. Other core functionalities are the *TaskManager*, *UserProfile*, *CodeGenerator*, *ExecutionContext*, *Personalization*, and *IdentifyIntention* modules. In the case of the *IdentifyIntention* module, the model used by the bot to identify intentions needs to be adapted to new commands. The model used in the *Personalization* module is in the same situation, and we can explore other techniques to improve bot suggestions. Finally, we can modify the categories of the *UserProfile* module according to new commands, and the *UserProfile* module will adapt to the new categories. In summary, the base of the modifications is new commands derived from the *Command* base classes. However, other components such as *UserProfile*, *IdentifyIntention*, and *Personalization* can be extended to be coherent with new commands.

5
Evaluation

This chapter describes how we evaluate our proposed solution, present the results obtained, and discuss their implications. We carried out two experiments. In the first experiment, Section 5.1, we instantiate our framework for two different situations: outlier detection and data cleaning. In the second, Section 5.2, users were asked to solve a data science task using the framework components.

5.1
Framework instances

This section describes how we instantiate our framework in two data science scenarios.

5.1.1
Outlier detection

One common data science problem is outlier detection, a widely explored problem in the literature. There are many techniques and algorithms to face such situations Chandola et al. (2007). Also, there are several applications in credit card fraud detection (Yu and Wang, 2009), network security (Zhang et al., 2010; Gogoi et al., 2011), to cite a few. However, all techniques rely on data analysis to identify odd patterns. Outlier detection aims at identifying anomalies in data (Hodge and Austin, 2004). Those anomalies can be human mistakes, fraud, or simply natural deviation of data distributions. Simple examples of outlier observations are negative ages due to human errors or abnormal transactions from cloned credit cards.

We selected a set of actions related to outlier detection: (a) extract a column from a dataset, (b) remove a column from a dataset, (c) train a specific outlier detection algorithm, (d) predict using a created model, and (e) visualize scatter plots. We map each action to a user intention as follow:

- Extract a column from a dataset: column_extraction
- Remove a column from a dataset: column_deletion
- Train specific outlier detection algorithm: outlier_algorithm

– predict using a created model: outlier_prediction

– visualize scatter plots: scatter_plot

Using the hot spots of the proposed solution, we create one command for each new intention, extend the personalization model and create a new chatbot model. The new commands require arguments; therefore, we inherit from the *CommandWithArgs* base class instead of the *Command* base class.

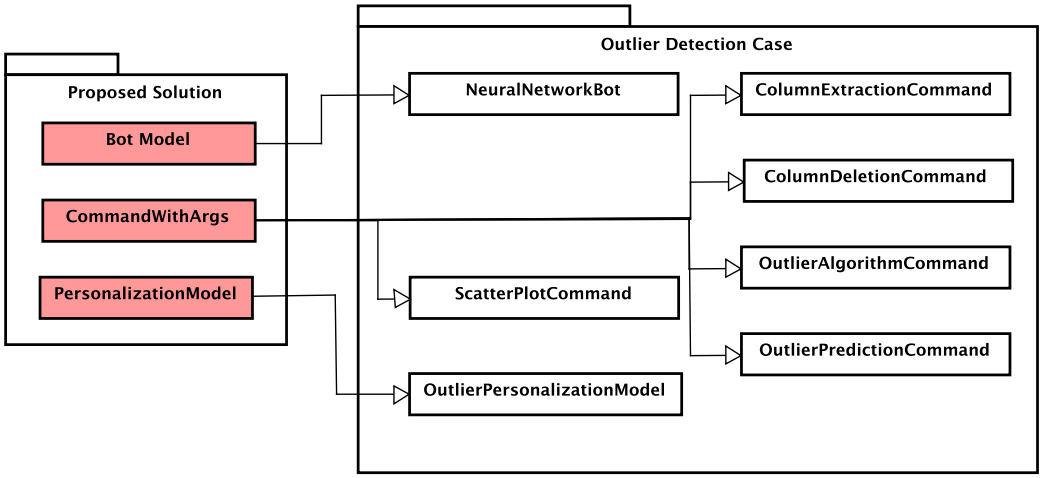


Figure 5.1: Outlier detection use case framework implementation.

Figure 5.1 presents the commands created for this example: *ColumnExtractionCommand*, *ColumnDeletionCommand*, *OutlierAlgorithmCommand*, *OutlierPredictionCommand*, and *ScatterPlotCommand*. The *OutlierAlgorithmCommand* receives the algorithm to be trained. We incorporated the following algorithms available in (Zhao et al., 2019): Principal Component Analysis (PCA), One-Class Support Vector Machines (OCSVM), Local Outlier Factor (LOF), Clustering-Based Local Outlier Factor (CLOF), Histogram-based Outlier Score (HBOS), K-Nearest Neighbors (KNN), Angle-Based Outlier Detection (ABOD), Isolation Forest (IForest), Extreme Boosting Based Outlier Detection (XGBOD), Lightweight On-line Detector of Anomalies (LODA) and Fully connected AutoEncoder (AutoEncoder). The *NeuralNetworkBot* uses the same architecture detailed in Figure 4.6, while the *OutlierPersonalizationModel* uses the default personalization model available in the proposed solution. We extended both models to add debug logs of their operations.

To represent this scenario we generate a simple outlier detection dialogue:

- User: Hi
- User: Lets load the dataset dataset outlier_sample_data in data/outlier.csv
- User: Select the target column and save it in outlier_target

- User: Remove the target column form the dataset outlier_sample_data
- User: Run the LODA algorithm and save it in autoencoder_model
- User: Predict using the autoencoder_model and save it in model_predictions
- User: Plot a scatterplot with model_predictions as colors
- User: Generate code in data/sample.py
- User: Bye

The previous dialog contains possible user inputs. However, we omitted bot responses and missing parameters interruptions to summarize the script. Note that in the example script, the user triggers all the created commands and uses some other general command such as *LoadDatasetCommand* or *GenerateCodeCommand*.

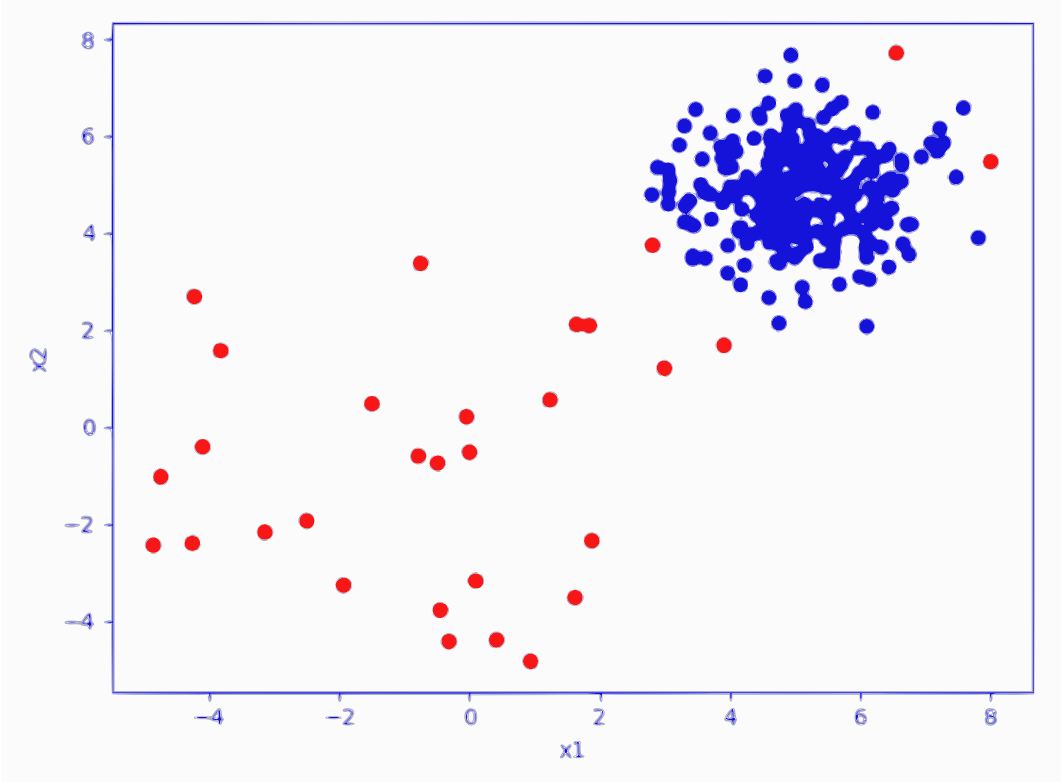


Figure 5.2: Scatter plot result of the outlier.

Figure 5.2 shows the result of the *ScatterPlotCommand*. The plot uses the predictions generated by the *OutlierPredictionCommand* to color the outliers in red and the regular data in blue. The *OutlierPredictionCommand* used the model generated by the *OutlierAlgorithmCommand* and the LODA algorithm. The commands *ColumnExtractionCommand* and *ColumnDeletionCommand* were used to prepare the dataset. The LODA algorithm used the default arguments to generate the model. However, we wanted to search for a better model. To overcome this situation, we created a pipeline to find a more accurate model.

The new pipeline command created *OulierPipeCommand* extends the base class *BasePipeCommand* and defines the sequences of command to execute and the execution order. In our case, we created the following sequence of commands: *ColumnExtractionCommand*, *ColumnDeletionCommand* and *TPOTCommand*. The TPOT command defines a grid search with several algorithms and their parameters. In our example, the TPOT returned the *CBLOF(input_matrix, contamination=0.1, n_clusters=8)* combination with 98% accuracy.

5.1.2 Data Cleaning

Data cleaning is one of the early stages of a data science process and is contained inside the data preprocessing step. According to (Wang et al., 2021), the data science process can be divided into tenv stages. One of the ten stages is Data Preprocessing and contains data cleaning as one of the approaches to process data. Data cleaning comprises all the techniques, strategies, and methods to transform incorrect or inconsistent data (Hellerstein, 2008). In this way, data should be transformed to grant the requirements of a specific data science task (Krishnan et al., 2016). Common examples of data cleaning are missing cells, text in numeric columns, or duplicated observations.

To implement our proposed solution framework we selected a set of data cleaning actions: (a) duplicate in a column, (b) transform column to lowercase/ uppercase, (c) remove additional white spaces, (d) replace Null/NaN observations, (e) normalize column and (f) sort column. All selected activities are handy in different situations. Transform to lowercase is useful when data comes with the same entity written with a combination of lower and upper case. For example, country names like (Brazil, BRAZIL, and brazil). All previous observations should be narrowed down to a single expression. Replace Null/Nan observations comes in handy when there is a null cell in the data that can potentially reduce performance in later stages of the data science process. The null/ NaN observations can be replaced by the mean, median, default value, etc. according to each problem specification or domain context. However, in our implementation we allow the media and the median values to replace the empty observation. Column normalization is key when there are scaling problems with data and may mislead the rest of the data science process. In summary, all selected data cleaning tasks help the outgoing data science process in a very specific way. We map every task to a user intention and generate a new command for every intention:

– duplicate in a column: column_duplication

– transform column to lowercase/uppercase: lower_upper_case

– remove additional white spaces: remove_white_spaces

– replace Null/NaN observations: replace_empty

– normalize column: column_normalization

– sort column: sort_column

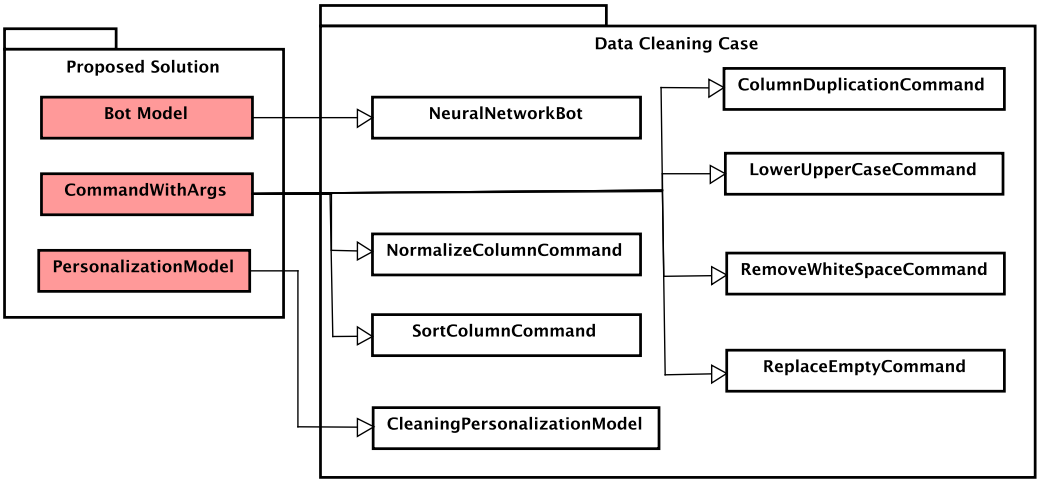


Figure 5.3: Data cleaning use case framework implementation.

Figure 5.3 demonstrate how every created intention turned into a new system command triggered by the user's input. To extend the framework architecture, we inherit from the appropriate base classes, retrain the intentions bot and the personalization model. To evaluate the created command, we generate a sample script:

- User: Hi
- User: Lets load the dataset cleaning_sample_data in data/data_cleaning.csv
- User: Remove duplicate lines from column x2
- User: Convert to lowercase the column x1 from the dataset cleaning_sample_data
- User: Replace empty values with the mean of the column x1
- User: Normalize the column x1 of the dataset cleaning_sample_data
- User: Sort the column x1 of the dataset cleaning_sample_data
- User: Bye

The previous dialogue was created to trigger all implemented commands and process a test dataset. Note that we omitted bot answers and parameter interruptions to simplify the example.

Figure 5.4(a) shows the dataset before the cleaning process while Figure 5.4(b) shows the dataset after the last sort command finished. Each command receives the dataset from the execution context, processes the data, and

	x1	x2
0	4.008780414158585437e+00	4.875245777243090828e+00
1	6.659941198333887868e+00	3.776461834377619731e+00
2	4.910271640185587039e+00	6.145712733467220801e+00
3	5.070319429876489536e+00	
4	6.500727364543041453e+00	4.879499841615219324e+00
5	3.888119031544130522e+00	4.623883074311895491e+00
6	4.611637611170428031e+00	3.803856355819496216e+00
7		3.925670017337897999e+00
8	6.102074501084556957e+00	3.498066480873270656e+00
9	NaN	3.482103922782001870e+00
10	NaN	4.448994311108052102e+00

(5.4(a)) Before

	x1	x2
280	0.000000	-2.435304519972205650e+00
279	0.008925	-1.026126124762823721e+00
288	0.046668	-2.396337462174522592e+00
291	0.049046	2.688009640539730682e+00
290	0.058914	-4.084455996308253845e-01
275	0.080510	1.573295326844569075e+00
299	0.133189	-2.167288367657778103e+00
296	0.183241	-1.934377887070082291e+00
289	0.227191	-3.260185843517168003e+00
295	0.261056	4.786269668930384484e-01
292	0.316443	-5.983108124258205152e-01

(5.4(b)) After

Figure 5.4: Dataset before and after the data cleaning

replaces the same reference with the updated dataset. In this way, the cleaning is made on the same dataset instead of duplicating the data on every command. Also, by using the same reference, the commands can set a sequence similar to the builder pattern where one command receives the result of the previous cleaning command and generates the input for the next cleaning command. By the end of the process, we remove unnecessary white spaces, normalize the x1 column, replace the null/nan entries by the column mean value, and sort the x1 column. Note how sorting messes up with the ids of the observations.

5.2 User Study

The study comprised four stages: Installation, Tutorial, Activities, and Questionnaire. Each stage has its specific purpose and depends on the completion of previous stages. The purpose of this study is to verify if the data developer can use the framework.

The Installation stage is to install the framework and its dependencies on the participant's computer. In this stage, we installed the framework on different operating systems (Windows, Ubuntu, Debian, and Mac-OS) and python versions (from 3.5.9 up to 3.8.5). The goal of this stage is to set up the environment, and it was guided most of the time by the participants. The analyst could act on installation errors that participants could not fix. Once the participant had created the proper environment, we proceeded to the Tutorial stage. The tutorial stage comprised several code chunks to execute and play. The tutorial shows how to extend the framework with a new command *ColumnNamesCommand*. The *ColumnNamesCommand* lists the columns of a dataset already loaded by the framework. Every code chunk comes with an explanation and enhances a complete but simple example of usage. For more details, the tutorial is available at *https://github.com/jefrysastre/dsbot*. The goal of this stage is to provide basic usage examples and an understanding

of the framework. The analyst guided this stage, and the participant followed the instructions to get the framework with the examples working. This stage represents a dialogue where participants ask about the framework and the analyst answers all questions, even questions deviating from the tutorial content. For example, the *ColumnNamesCommand* has a parameter to select the dataset and show the columns. However, many participants wanted to understand how the framework extracted the dataset name internally.

After the participants stated that they believed they could create their commands, we proceeded with the activities stage. The activities stage comprised two tasks: first, to create a command to show the header of a dataset already loaded in the framework. Our goal with this task was to get a general idea about how much of the tutorial the user understood. Note that the command of the Tutorial *ColumnNameCommand* shows the columns of a dataset. The user should make minor changes to get this task done. The second task asked the participant to apply a logarithm function to the first column of a dataset. This command requires a deeper understanding of the framework and increases the difficulty of the activity. Finally, we asked the participants to execute their code and test the two previous commands in the IRIS dataset. Our goal was to assess whether the commands were implemented correctly and fix the execution bugs. Finally, after a successful code test, the participants proceeded to the questionnaire stage. The questionnaire comprises four sections: agreement, profile, activities, and code. The agreement section was created to request permission to use participant's data in our research. The profile section has general questions about age, working areas, and python experience. The activities section contains general questions about the framework. It also has specific questions about each independent task (each command requested). Finally, in the code section, the participants had to upload their python code for analysis.

The proposed solution is intended for assisting data scientists who want to automate some steps of the work process. Help novice data scientists who can take advantage of commands made by other data scientists. Generate data analysis scripts that can be shared and manually enhanced by other data scientists. In other words, a data scientist arrives at a partial analysis of data and can generate a script to share the findings made in the proposed solution with other colleagues. The users should know the domain problem from a data science point of view, and must define, model, and formulate the problem.

On the other hand, the users who extend the proposed solution and create new commands will need the following background: - knowledge of python, frameworks, and Object-Oriented Programming (OOP) - knowledge of the

data science process and the main strategies used in each stage.

5.2.1 Results

A total of 28 people participated in the study. 68% was between 31 and 40 years old, 25% was between 21 and 30 years old, and the remaining were above 40. All participants have engineering degrees, 63% have a master's degree and 14% a Ph.D. degree. 82% of the participants come for computer science programs while 18% are from other areas, such as chemistry or telecommunication. From the 82% in computer science, 56% work with data science. Most participants (71%) have between one and four years of experience with python, 10% have more than four years of experience, and 18% have less than a year of python usage. Almost all participants (90%) had previously worked with the numpy and pandas libraries.

All participants completed both tasks answering our RQ2. In the first task, 75% ($CI_{95\%}(59\%, 91\%)$) of the participants reported that they concluded with little effort, while the remaining 25% claimed to have spent a moderate amount of effort to complete the first task. None of the participants found the first task very hard. More participants 82% ($CI_{95\%}(67\%, 96\%)$) found the second task very easy, but they had already completed the first task. However, one participant (3.6%) found the second task very hard to complete, and the remaining 14% found the second task required a moderate amount of work. The first command was very similar to the tutorial example and required minor changes to complete. The second was more distant and required a deeper understanding of the framework. However, more participants stated that the second task was very easy to complete (82%) against the (75%) of the first task. On the other hand, one participant found the second task harder than the first one. In general, the margin of error varies from +-16% in the first task and +-15% in the second task.

We created a set of variables to characterize each participant's profile and performance in the study, shown in Table 5.1.

Table 5.1: Variables created to quantify relevant aspects during the study.

Variable Name	**Description**
participant	This variable identifies the participants with a unique code.

installed_python	This variable flags if the user had a python version installed on the computer. Users that had not used python recently might have greater difficulties completing the activities.
python_experience	This variable sets three levels of python experience: (1) for participants with less than one year of experience, (2) for participants with one to four years of experience, and (3) for participants with more than four years of experience.
crawl_debugged	This variable identifies the users who investigated the source code by crawling the git files or debugging the tutorial examples.
code_generation	This variable identifies the participants who asked questions, did not understand or skipped the code generation methods.
left_super	This variable identifies the participants who left the tutorial superclass in their code.
imports	This variable identifies the participants who asked questions or missed the import numpy package during the activities.
dataset_reference	This variable identifies the participants who omitted the dataset reference during the execution of the activities.
bot_extension	This variable identifies the participants who did not register their commands in the framework.
unused_code	This variable identifies the participants who noticed the unnecessary code chunk in the tutorial examples.
suggest_updates	This variable identifies the participants who suggested at least one relevant improvement on the framework. This variable means that the participant achieved a reasonable level of understanding of the framework limits and proposed an improvement of some kind.
first_activity difficulty	This variable identifies the level of difficulty to complete the first activity. The possible values are: (1) very easy, (2) reasonable work, (3) very hard, (4) I couldn't do it.

second_activity difficulty	This variable identifies the level of difficulty to complete the second activity. The possible values are: (1) very easy, (2) reasonable work, (3) very hard, (4) I couldn't do it.

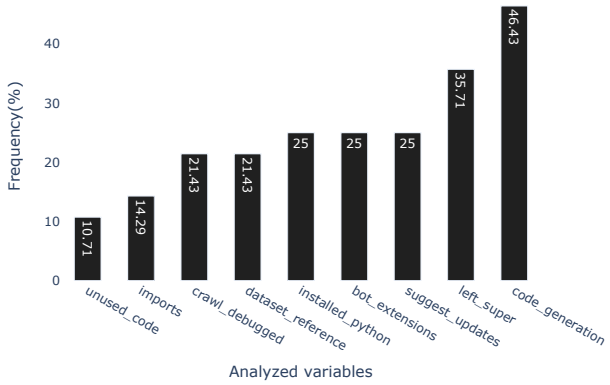


Figure 5.5: Frequency of events in the dataset.

In Figure 5.5, we can observe the frequency (in percentages) of the boolean variables of interest collected in the study. Almost half of the participants (50%) presented difficulties with code generation during the study. However, all of them ended up completing the activities after several attempts. Note that 25% of the participants made relevant suggestions that prove a deeper understanding of the framework. 20% read or debugged the source code to clarify some framework behaviors. 25% of the participants did not have a python version installed on the computer, and only 10% noticed the unused code lines in the tutorial. Therefore, 90% of the participants accepted those lines of code and did not question their role in the tutorial.

We created a correlation matrix to analyze how the variables are related. Figure 5.6 present the correlation matrix.

We can observe that the highest correlation happens between python experience and installation concerning the perceived difficulty of the activities with value -0.6. This negative correlation suggests, as expected, that users less experienced with python found the activities more challenging than users with more experience using python. Another strong correlation occurred between python installation and code generation, with value $+0.6$. This suggests that users without a python installed on their machine had more trouble completing

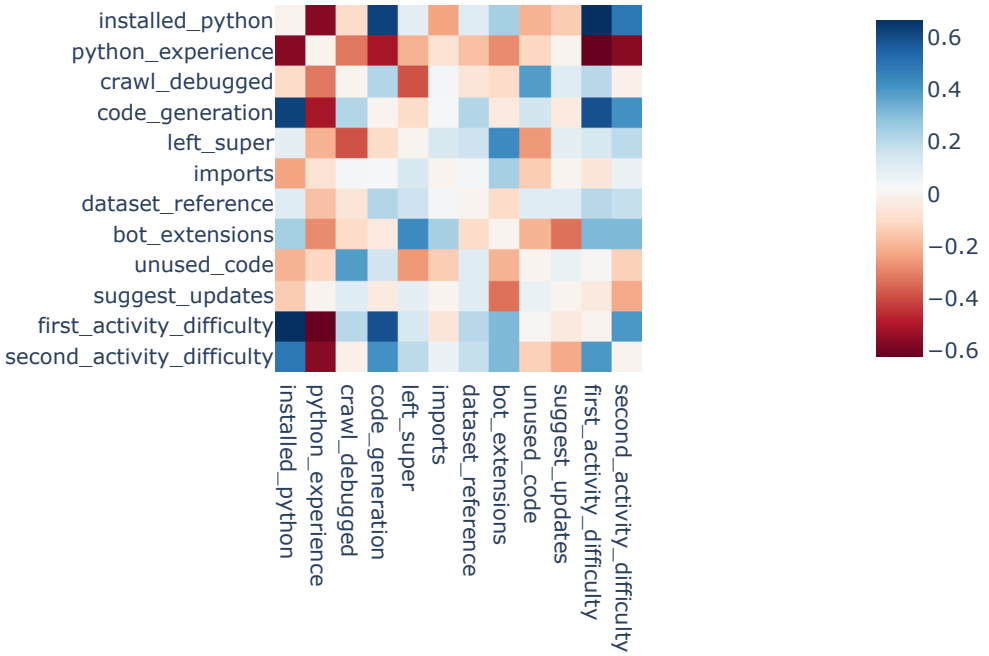
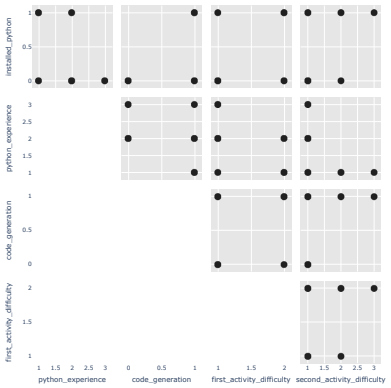


Figure 5.6: Correlation of the variables.

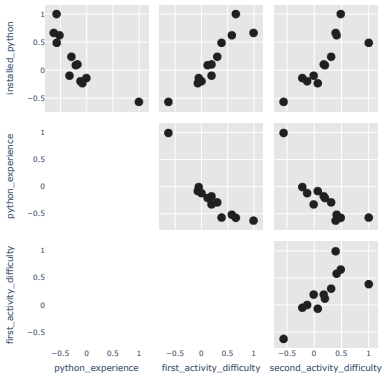
the code generation part of the activities than users with a python installed. People with python installed are expected to be more frequent python users and to be more familiarized with coding activities. To investigate this in more detail, we generated a scatterplot (Figure 5.7).

In Figure 5.7(a), we can see the occurrence of the value combinations of the more correlated variables. For instance, in the case of the *code_generation* and the *python_installed* variables we can see that the combination *python_installed = 0* and *code_generation = 1* did not happen for any of the participants. Therefore, all the participants who did not have python installed present difficulties with the code generation. Moreover, we can see that all participants with more than one year of python experience found the second activity very easy to complete. Only users who claimed to have less than a year of python experience found the second activity more difficult. Also, all participants who found the second activity with a moderate to a high level of complexity presented problems during code generation. Therefore, all the participants that did not have difficulties with code generation found the second task very easy to complete.

Figure 5.7(b) presents the correlation matrix between *python_installed*, *python_experience*, *first_activity_difficulty* and the *second_activity_difficulty*. As expected, we can observe a positive linear correlation between the *installed_python* and the activities difficulty. There is a negative linear cor-



(5.7(a)) Scatterplot of single value occurrences.



(5.7(b)) Scatter correlation matrix of the python_installed, python_experience and the activities difficulty.

Figure 5.7: Scatter matrix of the strongest correlated variables.

relation between the *python_experience* and the activities difficulty. Those correlations agree with the findings from the scatter matrix. Also, we can observe that there is a strong correlation between the participants who claimed to have more than one year of python experience and the participants who had to install python for the study.

We filtered the collected dataset by the *python_experience* variable to keep the participants with more than one year of python experience. There are 22 participants with more than one year of python experience and, in the first task 82% ($CI_{95\%}(69\%, 95\%)$) found the task very easy to complete. In this case, the margin of error varied from 16% (28 participants) to 13% (22 participants with more than one year of python experience) and maintained the original sample distribution. However, all participants with more than one year of python experience found the second task very easy to complete. In this case, the margin of error varies from 15% (28 participants) to near 0% (22 participants with more than one year of python experience). We believe that users with more than one year of python experience learned from the first task and found the second one easier to complete. Note that four participants with more than one year of python experience found the first task with moderate difficulty, and all participants found the second task very easy to complete. However, we expected it to be the other way because the first task was similar to the tutorial example while the second was further away from the tutorial.

5.2.2 Discussion

In this subsection, we first describe some recurrent scenarios during the application of the study and then present some analysis of the collected variables.

Five participants had wrong or broken Pip (Python installer Packages) links due to old Pip versions. We commonly solve this error by upgrading Pip to the latest version. In some environments, participants used packages conflicting with another project. We create a new virtual environment with a clean python setup. Some participants had low internet speed. This problem did not affect the installation but instead forced two participants to schedule a second session to continue with the study. Some computers did not have enough disk space available. We solved this problem by freeing space on the disk. However, we needed to free up to 3GB on the main computer partition to complete the installation, although the installation required only 1.5GB. Note that the Pip tool uses the main partition to download the packages, so even when participants installed on different hard drive partitions, the main partition needed free space. Almost all participants posed questions about the code generation section, but only a few inspected the source code files to find out. Other participants debugged the tutorials to gain a clear understanding of the execution context.

Seven participants skipped the code generation for the commands or pasted the tutorial code generation example. Some participants misunderstood the purpose of the study. Others did not find it necessary to complete the activities. The participants could get a false sense of completion if they skipped the code generation because the bot executes the commands and shows results. However, the framework did not generate code for those commands and this might affect future code generation blocks. Almost all participants left the *ColumnNameCommand* superclass for their commands and received an execution error during the tests. All participants quickly realized and fixed the error.

Four participants assumed that there was no need to import their packages inside their commands. The second command requires the numpy library, and some participants argued that the framework already loaded the numpy library. This error could be because the numpy and pandas libraries appear together frequently, and the framework uses the pandas library. So, participants could use the pandas dataframe object without importing the pandas library. Some users said: 'I already have a dataframe object, and I didn't import the pandas library.'. The *force_training* parameter determines whether

the framework can use old serialized files or it must train a new chatbot. Seven participants left the *force_training* parameter set to true even when loading an old chatbot to test several ideas. This parameter affected only the loading time of the chatbot at every execution. However, when training heavy models, the training should be executed only on demand. Almost every participant used their commands without specifying the dataset value. All commands in the Tutorial and the Activities required a dataset argument to complete the action. However, almost no one realized that they were not specifying the dataset value, and the framework completed the action correctly. The framework resolved the last referenced dataset in the dialogue and used that dataset to complete the action. Nevertheless, participants did not know about it, and none of them questioned the framework behavior. Some participants argued: 'Naturally, the framework uses the dataset we are talking about.'.

Seven participants did not register their commands to the *extension* option of the framework. The *extension* option defines a list of new commands that the framework must incorporate and train all the models. In those cases, the code did not break, but the bot failed to recognize those commands and some participants struggled to find out why. Most of them tried to find errors inside their command, but the problem was in the framework initialization. In the tutorial code, we placed an unused import of a random class from the framework. This line of code had no explanation or links to any other code on the tutorial. However, almost no participant (3 participants) questioned or removed the unused code chunk. The rest of the participants (25 participants) ignored or just accepted the code without knowing the reason for its being there.

Some participants suggested that the framework should have a function that receives all the hot spots and create a class internally. This approach should remove the object-oriented programming (OOP) dependency and reduce the amount of code required to extend the framework. However, it becomes more cryptic. Other participants suggested that the bot should maintain the loaded object through several executions. This case happened when the participants tried to use a dataset from old conversations.

6
Conclusion

This work describes a framework to automate data science tasks by using personalized chatbots. To do so, we instantiated the framework in an outlier detection scenario. We also conducted a user study to demonstrate the use of the tool. The proposed solution and the outlier detection instance of the framework answer our RQ1, as discussed in Section 5.1. The outlier detection instance shows how we instantiate framework components to automate specific data science tasks; specifically, the personalization model to reach the desired L2 level of automation. The user study carried out answered our RQ2, as discussed in Section 5.2.1. It shows that data developers are able to extend the proposed framework for specific scenarios.

We also analyzed that the highest correlation happens between python experience and installation concerning the difficulty of the activities. We show that this negative correlation infers that users less experienced with python found the activities more challenging than users with more experience using python. Our results would seem to suggest that code generation depends on python installation. We conjecture that the main reason is that users without python installed on their machine had more difficulty in finalizing the coding activities than users with python installed. In addition, during the interview, some participants suggested that the framework should have a function that receives all the hot spots and create a class internally.

We see two main contributions of this work. First, the main contribution is the proposed workflow to reach L2 level of automation. We achieved our goal through conversational interfaces and personalized suggestions. Second, the presented framework to automate some DS/ML processes.

Next, we point out some limitations of this work. First of all, our study did not evaluate the impact of the suggestions provided by the framework to guide the user on DS/ML procedures. For this reason, we recommend a future study to evaluate how such recommendations influence the users' DS/ML procedures execution. Moreover, how new data collected from data might improve the system recommendations. A second limitation is the monitoring of the background activities. We did not propose tools or commands to keep track of background activities. Therefore, the data scientists will only receive

the final results. However, sometimes the data scientist needs to collect partial results, or the working task does not make sense anymore. Finally, a third limitation is that the participants of our evaluation study were mostly data developers. However, a more heterogeneous group can show different findings and improve the user study.

6.1 Future works

As future work, we can also add another level of automation to the framework, such as the L3 automation level as a user configuration. In this case, the bot should have more flexibility to take action. However, the bot may waste some time due to poor decisions. We could take advantage of the bot suggestions and the confidence of the proposed continuation steps. The bot can automatically initiate the suggestions above a certain confidence threshold to achieve the desired level of automation.

Another future work can be the preparation of the framework to accept new commands as functions. This approach should remove the object-oriented programming dependency and reduce the amount of code required to use the framework. The only way to extend the framework is through inheritance and overriding the correct classes. However, several participants suggested that they would have preferred a different approach to create new commands. In this case, the bot instance requires methods to receive the framework hot spots and internally create the corresponding class.

Some participants assumed that their datasets remain loaded through several executions, but the framework releases the memory resources upon exit. We believe that this scenario could be an interesting future work, where users can make loaded datasets and models permanent until the user explicitly deletes those objects. In this way, the instances will remain loaded through several framework executions. But, we must consider memory consumption. The bot should serialize the permanent object before exiting the application and load it back again after the initialization.

We see that more than half of participants had some issue while generating code based on the variables collected. We propose a new study focusing on the code generation feature to reveal why the participants had trouble generation code. From the data scientist's point of view, the code generation feature can turn a conversation into a script to manually work on later. This study will clarify the code generation contributions.

Finally, one could create a domain-specific language(DSL) below the chatbot layer to export/import conversations in a pseudo-language. In this

case, the users can modify some aspects of the conversation and run the conversation again without going through every interaction with the bot.

Bibliography

Abdellatif, A., Costa, D., Badran, K., Abdalkareem, R., and Shihab, E. (2020). Challenges in chatbot development: A study of stack overflow posts.

Alemu, E. N. and Huang, J. (2020). Healthaid: Extracting domain targeted high precision procedural knowledge from on-line communities. *Information Processing & Management*, 57(6):102299.

Berthold, M. R. (2019). What does it take to be a successful data scientist? *Harvard Data Science Review*, 1(2).

Brandtzaeg, P. B. and Følstad, A. (2018). Chatbots: changing user needs and motivations. *Interactions*, 25(5):38–43.

Chandola, V., Banerjee, A., and Kumar, V. (2007). Outlier detection: A survey. *ACM Computing Surveys*, 14:15.

Chaves, A. and Gerosa, M. (2019). How should my chatbot interact? a survey on human-chatbot interaction design.(2019). *arXiv preprint arXiv:1904.02743*.

Chittò, P., Baez, M., Daniel, F., and Benatallah, B. (2020). Automatic generation of chatbots for conversational web browsing. In *International Conference on Conceptual Modeling*, pages 239–249. Springer.

Costa, C. and Santos, M. Y. (2017). A conceptual model for the professional profile of a data scientist. In *World Conference on Information Systems and Technologies*, pages 453–463. Springer.

Dey, S. and Zhang, P. (2019). Estimating personalized drug responses from real world evidence. US Patent App. 15/855,314.

Ed-douibi, H., Cánovas Izquierdo, J. L., Daniel, G., and Cabot, J. (2021). A model-based chatbot generation approach to converse with open data sources. In *International Conference on Web Engineering*, pages 440–455. Springer.

Fast, E., Chen, B., Mendelsohn, J., Bassen, J., and Bernstein, M. S. (2018). Iris: A conversational agent for complex tasks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12.

Følstad, A. and Brandtzæg, P. B. (2017). Chatbots and the new world of hci. *interactions*, 24(4):38–42.

Følstad, A., Nordheim, C. B., and Bjørkli, C. A. (2018a). What makes users trust a chatbot for customer service? an exploratory interview study. In *International Conference on Internet Science*, pages 194–208. Springer.

Følstad, A., Skjuve, M., and Brandtzaeg, P. B. (2018b). Different chatbots for different purposes: towards a typology of chatbots to understand interaction design. In *International Conference on Internet Science*, pages 145–156. Springer.

Ghandeharioun, A., McDuff, D., Czerwinski, M., and Rowan, K. (2019). Towards understanding emotional intelligence for behavior change chatbots. In *2019 8th International Conference on Affective Computing and Intelligent Interaction (ACII)*, pages 8–14. IEEE.

Gogoi, P., Bhattacharyya, D., Borah, B., and Kalita, J. K. (2011). A survey of outlier detection methods in network anomaly identification. *The Computer Journal*, 54(4):570–588.

Guyon, I., Bennett, K., Cawley, G., Escalante, H. J., Escalera, S., Ho, T. K., Macià, N., Ray, B., Saeed, M., Statnikov, A., et al. (2015). Design of the 2015 chalearn automl challenge. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Harris, H., Murphy, S., and Vaisman, M. (2013). *Analyzing the analyzers: An introspective survey of data scientists and their work.* " O'Reilly Media, Inc.".

He, X., Zhao, K., and Chu, X. (2019). Automl: A survey of the state-of-the-art. *arXiv preprint arXiv:1908.00709*.

Hellerstein, J. M. (2008). Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 25.

Hodge, V. and Austin, J. (2004). A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126.

Idoine, C., Krensky, P., Brethenoux, E., Hare, J., Sicular, S., and Vashisth, S. (2018). Magic quadrant for data science and machine-learning platforms. *Gartner, Inc*, page 13.

Jones, J. (2003). Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, page 26.

Kadariya, D., Venkataramanan, R., Yip, H. Y., Kalra, M., Thirunarayanan, K., and Sheth, A. (2019). kbot: Knowledge-enabled personalized chatbot for asthma self-management. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 138–143. IEEE.

Knoop, S. E., Ng, T. H. M., and Timm, J. T. (2019). Personalized questionnaire for health risk assessment. US Patent App. 15/716,819.

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., and Leyton-Brown, K. (2017). Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830.

Krishnan, S., Haas, D., Franklin, M. J., and Wu, E. (2016). Towards reliable interactive data cleaning: A user survey and recommendations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–5.

Le, T. T., Fu, W., and Moore, J. H. (2020). Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics*, 36(1):250–256.

Liu, S., Leng, Z., and Wijaya, D. (2020). Learning to mirror speaking styles incrementally. Available at http://arxiv.org/pdf/2003.04993v1 (2020/03/05) | 4 pages, 3 tables, 1 figure.

Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., Urban, M., Burkart, M., Dippel, M., Lindauer, M., and Hutter, F. (2018). Towards automatically-tuned deep neural networks. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *AutoML: Methods, Sytems, Challenges*, chapter 7, pages 141–156. Springer. To appear.

Musto, C., Narducci, F., Polignano, M., de Gemmis, M., Lops, P., and Semeraro, G. (2020). Towards queryable user profiles: Introducing conversational agents in a platform for holistic user modeling. In *UMAP 2020 Adjunct - Adjunct Publication of the 28th ACM Conference on User Modeling, Adaptation and Personalization*, pages 213–218. Association for Computing Machinery.

Neururer, M., Schlögl, S., Brinkschulte, L., and Groth, A. (2018). Perceptions on authenticity in chat bots. *Multimodal Technologies and Interaction*, 2(3):60.

Pérez-Soler, S., Guerra, E., and de Lara, J. (2021). Creating and migrating chatbots with conga. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 37–40. IEEE.

Portela, M. and Granell-Canut, C. (2017). A new friend in our smartphone? observing interactions with chatbots in the search of emotional engagement. In *Proceedings of the XVIII International Conference on Human Computer Interaction*, pages 1–7.

Porter, T. T. (2015). *Identifying the data scientist amongst stem educators: An introspective survey of work skills*. PhD thesis, Capella University.

Reis, L., Maier, C., Mattke, J., and Weitzel, T. (2020). Chatbots in healthcare: Status quo, application scenarios for physicians and patients and future directions. In *ECIS*.

Rhee, C. and Choi, J. (2020). Effects of personalization and social role in voice shopping: An experimental study on product recommendation by a conversational voice agent. *Computers in Human Behavior*.

Rönnberg, S. (2020). Persuasive chatbot conversations: Towards a personalized user experience.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Smestad, T. L. (2018). Personality matters! improving the user experience of chatbot interfaces-personality provides a stable pattern to guide the design and behaviour of conversational agents. Master's thesis, NTNU.

Sun, Y. and Zhang, Y. (2018). Conversational recommender system. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 235–244.

Suta, P., Lan, X., Wu, B., Mongkolnam, P., and Chan, J. (2020). An overview of machine learning in chatbots. *Int J Mech Engineer Robotics Res*, 9(4):502–510.

Tallyn, E., Fried, H., Gianni, R., Isard, A., and Speed, C. (2018). The ethnobot: Gathering ethnographies in the age of iot. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–13.

Wang, D., Liao, Q. V., Zhang, Y., Khurana, U., Samulowitz, H., Park, S., Muller, M., and Amini, L. (2021). How much automation does a data scientist want? *arXiv preprint arXiv:2101.03970*.

Yao, Q., Wang, M., Chen, Y., Dai, W., Yi-Qi, H., Yu-Feng, L., Wei-Wei, T., Qiang, Y., and Yang, Y. (2018). Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*.

Yu, W.-F. and Wang, N. (2009). Research on credit card fraud detection model based on distance sum. In *2009 International Joint Conference on Artificial Intelligence*, pages 353–356. IEEE.

Zhang, Y., Meratnia, N., and Havinga, P. (2010). Outlier detection techniques for wireless sensor networks: A survey. *IEEE communications surveys & tutorials*, 12(2):159–170.

Zhao, Y., Nasrullah, Z., and Li, Z. (2019). Pyod: A python toolbox for scalable outlier detection. *Journal of Machine Learning Research*, 20(96):1–7.

A
Appendix

The appendices contains extra material related to our work. In Appendix we present a detailed installation guide and several tutorial examples. The installation guide will facilitate further reproductions of this work.

A.1
Installation guide and tutorial

In this section we plan to setup an environment to run the proposed solution. We list all the prerequisites and provide a brief explanation.

DSBot is a chatbot framework to automate data science and machine learning processes. The framework provide several common base commands and the capability to adapt to other scenarios by creating new commands. Install and update using pip *https://pypi.org/project/pip/* tool.

The framework can be installed in Windows, MacOS and Linux using python versions from 3.5.x to 3.8.x. The installation process takes between 15 and 30 minutes depending on the internet link speed. The framework depends on several heavy packages such as tensorflow and pytorch. The framework requires around 2GB to download all the dependencies on the installation folder and up to 5GB on the main partition, specifically in the temporary cache folder.

A.1.1
A Simple Example

This first example show the minimal settings required to execute the framework. The *bot_options* object contains all framework options. In this case the *save_path* option points to a folder to store the generated files. The framework will generate files to save and load the bot after the training. Also will serialize the user history and the preprocessed datasets.

```python
from dsbot.DSBot import DSBot

if __name__ == '__main__':

    bot_options = {
        'save_path': "data/",
    }

    dsbot = DSBot(**bot_options)
    dsbot.start()
```

Figure A.1: Basic framework example.

Note: The system will create the *data* directory if not exist.

A.1.2 Creating New Command

In this subsection we will extend the framework to create a new command. The command is going to be the. *ShowColumnNamesCommmand*. This command will show the names of the columns in a specified dataset. Figure A.2 shows how to create a command using the framework. We can see that we import the *CommandWithArgs* base class and create a new class extending *CommandWithArgs*. The *tag* holds the unique command code required to identify the command. The triggers stores some patterns to execute the command. In other words: how the command will be called during a conversation.

```python
from dsbot.commands.base.command_with_args import CommandWithArgs

class ColumnNameCommand(CommandWithArgs):

    tag = "show_column_name"
    patterns = [
        "Mostre os nomes das colunas",
        "Quais sao as colunas"
    ]
```

Figure A.2: New command class.

Our command *ColumnNameCommand* must know from which dataset

it has to list the columns. So, we need to add a parameter *dataset_name*. The method *create_dataset* receives the dataset name and a trigger phrase. Also, we have to prepare the bot answers *responses*. To do this, we setup the *___init___* method as shown in Figure A.3

```
from dsbot.commands.base.command_with_args import CommandWithArgs
from dsbot.commands.base.argument import Argument

class ColumnNameCommand(CommandWithArgs):

    tag = "show_column_name"
    patterns = [
        "Mostre os nomes das colunas",
        "Quais sao as colunas"
    ]

    def __init__(self, parent, task_manager):
        super(ColumnNameCommand, self).__init__(parent, task_manager)
        self.responses = ["Posso mostrar as colunas do dataset"]
        self.create_argument("dataset_name", "dataset")

    def run(self, context):
        dataset_name = self.dataset_name.value

        try:
            df_columns = context[dataset_name].columns
            print(df_columns)
        except Exception as e:
            print(e)
```

Figure A.3: Command initialization.

The *create_argument* function will create a property *dataset_name* to save the dataset parameter. In fact, the argument extraction will use regular expressions to select what the user types after the word *dataset*. For instance, when the user types "vamos carregar o dataset iris". The system will understand the *dataset_name* values as "iris". Just the word after the "dataset" trigger.At this point, we have a base command with a dataset argument, but with the default functionality. To override the default behavior we override the *run* and *generate_code* methods.

```python
from dsbot.commands.base.command_with_args import CommandWithArgs
from dsbot.commands.base.argument import Argument

class ColumnNameCommand(CommandWithArgs):

    tag = "show_column_name"
    patterns = [
        "Mostre os nomes das colunas",
        "Quais sao as colunas"
    ]

    def __init__(self, parent, task_manager):
        super(ColumnNameCommand, self).__init__(parent, task_manager)
        self.responses = ["Posso mostrar as colunas do dataset"]
        self.create_argument("dataset_name", "dataset")

    def run(self, context):
        dataset_name = self.dataset_name.value

        try:
            df_columns = context[dataset_name].columns
            print(df_columns)
        except Exception as e:
            print(e)

    def generate_code(self, code_generator, context):
        code_generator.write("")
        code_generator.write("# Select column from dataset")

        dataset_name = self.dataset_name.value

        code_generator.write("print({0}.columns)".format(dataset_name))
```

Figure A.4: Overriding default functionalities.

At this point, we have a new command ready to be used by the framework. However, to execute the bot with the new command, we need to pass it in the initial configuration options as you can see in the following example.

```
from dsbot.DSBot import DSBot
from extensions.ColumnNameCommand import ColumnNameCommand

if __name__ == '__main__':

    bot_options = {
        'save_path': "data/",
        'force_training': True,

        'extensions': [
            ColumnNameCommand
        ],
    }

    dsbot = DSBot(**bot_options)
    dsbot.start()
```

Figure A.5: Basic framework example.

The system requires several files such as the corpus, the bot (tensorflow files), among others. When we add new commands or extend the base commands with new ones, we must recreate those files. otherwise the system will load old files without the new commands. To summarize, the main.py needs to have the same number of commands as the files in the data folder. For example, if we add the *ColumnNameCommand* in the extend parameter without the *force_training* parameter set to true, the system will load old serialized files and return an error. The loaded commands does not match the extended commands. To force the training and serialization of new files, we can use the *force_training* parameter set to true.

A.1.3 Study Activities

Nosso estudo consiste em estender um framework para criar chatbots, mas antes de tudo vamos entender o tutorial da ferramenta no link *https://github.com/jefrysastre/dsbot*.

Crie um chat bot capaz de executar as seguintes tarefas utilizado a ferramenta *DSBot*. Para criar os commandos novos vc tem que estender a ferramenta. Para estender a ferramenta siga os exemplos do link: *https://github.com/jefrysastre/dsbotcreating-new-commands*

– Crie um comando para mostrar o cabeçalho do dataset (as 5 primeiras linhas). Pode usar a funcão head do pandas para mostrar as linhas.

– Aplique uma transformação logaritmica na primeira coluna. Pode usar a função np.log do pacote numpy.

– Carrege o dataset iris e teste os comandos que vc implementou. Para carregar o dataset escreva algo do tipo: "vamos carregar o dataset iris".