

UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN**



**MÁSTER UNIVERSITARIO DE INGENIERÍA DE
TELECOMUNICACIÓN**

TRABAJO FIN DE MÁSTER

**DESIGN AND IMPLEMENTATION OF A
GOOGLE ACTION ENABLED SMART
AGENT SYSTEM FOR MOBILE APP
REVIEW MONITORING BASED ON
SENTIMENT ANALYSIS TECHNIQUES**

ANTONIO FÉRNANDEZ LLAMAS

2017

PROYECTO FIN DE CARRERA

Título: Desarrollo de un Sistema de Agente Inteligente basado en Google Actions para la Monitorización de Opiniones de Aplicaciones Móviles utilizando técnicas de Análisis de Sentimientos.

Título (inglés): Design and Implementation of a Google Action enabled Smart Agent System for Mobile App Review Monitoring based on Sentiment Analysis Techniques

Autor: Antonio Fernández Llamas

Tutor: Carlos A. Iglesias Fernández

Departamento: Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:

Vocal:

Secretario:

Suplente:

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN**

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes



TRABAJO FIN DE MÁSTER

**DESIGN AND IMPLEMENTATION OF A
GOOGLE ACTION ENABLED
SMART AGENT SYSTEM FOR
MOBILE APP REVIEW MONITORING
BASED ON SENTIMENT ANALYSIS
TECHNIQUES**

Antonio Fernández Llamas

Junio de 2017

Resumen

Actualmente, el procedimiento para desarrollar nuevas aplicaciones móviles resulta más sencillo que nunca. Un desarrollador independiente es capaz de tener una idea, implementarla por sí mismo y publicarla en la tienda con una barrera de entrada muy inferior. Como desarrollador de producto móvil, es de vital importancia tratar de situar tus aplicaciones lo más alto posible dentro del ranking, manteniendo el producto lanzando nuevas funcionalidades y corrigiendo pequeños errores.

Sin embargo, las plataformas de distribución de aplicaciones iOS y Android ofrecen retroalimentación por parte del usuario con una valoración media entre 1 y 5, y en ocasiones adjuntando una corta opinión sobre la experiencia de usuario. En la mayoría de los casos esta información es insuficiente para identificar una mala aceptación por parte de los usuarios, y en consecuencia un mal posicionamiento en la tienda.

Esta falta de datos proveniente del mercado de aplicaciones acerca de cómo se siente el usuario al utilizar una aplicación supone un grave problema para los desarrolladores. Extrayendo las opiniones de los usuarios de una aplicación, y llevando a cabo un análisis de sentimientos o emociones puede dar a conocer nuevos datos sobre los gustos del usuario, pudiendo incluso hacer recomendaciones en base a esos datos. Además, el uso de clasificadores que detecten errores o mejoras en los comentarios puede ayudar al equipo de desarrollo a redirigir estas respuestas siendo previamente etiquetadas.

Para realizar estas soluciones hemos definido varios objetivos. En primer lugar, desarrollar un agente inteligente para terminales Android donde el usuario pueda ejecutar análisis de sentimientos y emociones de las aplicaciones publicadas en la Play Store. El usuario interactuará con el agente mediante comandos de voz o texto, utilizando sistemas para entender el lenguaje natural. Además el sistema permitirá la automatización de respuestas a opiniones publicadas en apps desarrolladas por el usuario, dependiendo de la clasificación obtenida. Por último, ofrecerá un servicio para analizar el sentimiento en Play Store.

Palabras clave: Agente Inteligente, Android, Google Actions, API.AI, Análisis de sentimientos, Análisis de emociones, Python, Clasificador de errores, Java, Clasificación de mejoras.

Abstract

Nowadays, app development for smart phone ecosystem is quite fast and easier than ever. A freelance developer can have an idea, implement it by himself and publish the app in the market with a low barrier of entry. As a mobile product developer, it is essential to place your apps in the top of the store rankings, maintaining your product by implementing new features and minor bug fixes.

However, iOS and Android market platforms only provide customer feedback with an average rating of 1 up to 5, and sometimes including a short review with the user experience. In most cases, this information is not enough to identify why your app has a bad acceptance, and consequently a bad positioning in the store.

This lack of data provided by marketplaces about how user feels using an app means a problem for developers. Mining the reviews of a certain application, and performing a sentiment or emotion analysis, we can obtain an implicit more detailed information about the product, identifying how the user feels, and evaluate his accordance with the product. Even comparing same user reviews in similar apps, we can extract some information of what do my users like about other related apps?, and make recommendations based on this data. Furthermore, bug and features classification it's also a good implementation, so developer teams composed by low number of programmers can redirect the feedback obtained from the store previously tagged.

To carry out these solutions we have defined several objectives. First of all, develop a smart agent for Android devices where user can perform sentiments and emotions analysis for different Play Store apps. The smart agent interacts with the user through voice or text sentences, which will be converted into specific actions using a Natural Language Understanding system. Moreover, the system allows us to automate Play Store review response for applications developed by the user, depending on the classification result obtained. Finally, offering a service oriented to developers, where they can manage and analyze mobile application market status from their Android terminal.

Keywords: Smart Agent, Android, Google Actions, API.AI, Sentiment Analysis, Emotions Analysis, Python, Bug Classification, Java, Features Classification

Agradecimientos

En primer lugar quiero agradecer a mi familia y amigos por haberme apoyado durante todo este tiempo cursando el Máster Universitario.

Gracias a mi tutor Carlos Ángel Iglesias por ayudarme y orientarme a la hora de realizar este proyecto, así como a los compañeros del Grupo de Sistemas Inteligentes (GSI) por brindarme su ayuda siempre que lo he necesitado.

Como mención especial, dedico este trabajo en conjunto a Alberto Luján Martín, por enseñarme a disfrutar de la vida, y valorar cada momento como si del último se tratara.

Contents

Resumen	VII
Abstract	IX
Agradecimientos	XI
Contents	XIII
List of Figures	XVII
List of Tables	XXI
1 Introduction	1
1.1 Context	3
1.2 Overview	5
1.3 Master thesis goals	5
1.4 Structure of this Master Thesis	6
2 Enabling Technologies	9
2.1 Python	11
2.1.1 Flask	11
2.2 Java	12
2.2.1 Android basis	12
2.3 Natural Language Understanding Systems	14
2.4 Google Actions	17

2.4.1	Conversation API	18
2.4.2	API.AI	19
2.4.2.1	Machine Learning	22
2.5	Senpy	23
2.6	Google Play Developer API	23
2.6.1	Reply to Reviews API	24
3	Requirement Analysis	27
3.1	Use Cases	29
3.1.1	Actors dictionary	29
3.1.2	UC1: Request a new analysis	31
3.1.3	UC2: Request a new classification	32
3.1.4	UC3: Authenticate with Google Play Developers API	33
3.1.5	UC4: Request a review	34
3.1.6	UC5: Reply a review	35
3.2	Requisites capture and analysis	35
4	Architecture	37
4.1	Overview	39
4.1.1	Server Module	39
4.1.2	API.AI Agent	40
4.1.3	Client Module	41
4.2	Server Module	42
4.2.1	Controller	42
4.2.2	Play Store module	44
4.2.3	Senpy module	45
4.2.4	Classification module	48

4.2.4.1	Dataset	49
4.2.4.2	Performance	50
4.2.4.3	Workflow	51
4.3	API.AI Agent Module	53
4.3.1	Elements	54
4.3.1.1	Entities	54
4.3.1.2	Intents	55
4.3.1.3	Conversational Context	55
4.3.1.4	Responses	56
4.3.2	Workflow	57
4.3.3	Machine Learning techniques	57
4.4	Client Module - Android Smart Agent	58
4.4.1	Overview	58
4.4.2	General application	60
4.4.3	Chat agent module	61
4.4.4	Developer-oriented module	63
4.4.5	Ranking sentiment analysis module	65
5	Case study	67
5.1	Scenario overview	69
5.1.1	Market Explorer	70
5.1.2	Feature mining	71
5.1.3	Trending apps	75
5.1.4	Automated reply for bugs detection	77
5.2	Conclusions	79
6	Conclusions and future work	81

6.1	Conclusions	83
6.2	Achieved goals	84
6.3	Future work	85
	Bibliography	86

List of Figures

2.1	Comparative between most services, evaluating most important categories offered by them [1]	14
2.2	Evaluation process for intent detection applied to <i>Places</i> extension 2.4. . . .	16
2.3	Evaluation process for intent detection applied to <i>Weather</i> extension 2.4. .	16
2.4	Graph which represents precision over recall in parameter identification for most common NLU services 2.4.	17
2.5	Conversation action lifecycle inside Google Actions environment.	18
2.6	Conversation API workflow representation including requests and responses.	19
2.7	API.AI architecture inside application scenario.	21
3.1	UC1. Request a new analysis	31
3.2	UC2. Request a new classification	32
3.3	UC3. Authenticate	33
3.4	UC4. Request a review	34
3.5	UC5. Reply a review	35
3.6	Use cases diagram	36
4.1	Global achitecture of the system developed.	39
4.2	Listing of the API REST calls enabled in the server. HTTP Method, request name and required parameters are specified with a brief description about the response obtained.	43
4.3	Example of an output JSON file structure generated for <i>MyApp</i> test application.	45

4.4	Sequence diagram with the analysis operation work-flow, since the user request to the final output with the application analysis.	46
4.5	Sequence diagram with the classify operation work-flow, since the user request to the final output with the bug or feature binary classification. . . .	52
4.6	Sequence diagram with the classify operation work-flow for developers-oriented side.	53
4.7	Example of intent communication using contexts inside the API.AI agent. .	56
4.8	Sequence diagram with API.AI agent behaviour when receives a text or voice input from the user.	57
4.9	Zoom in about client module inside the global architecture of the project. .	59
4.10	The figure represents an UML diagram modelling with those classes that actively participate in the application.	61
4.11	The figure represents the sequence followed by the mobile agent when a new query is made by the user. The process starts when the input trigger, calling the <i>AIManager</i> service and finally obtaining the JSON response from API.AI agent, which is translated into a new list element.	62
4.12	Sequence diagram of the answering process. First the user scans a self-developed application and detects possible bugs. After he posts the recommended replies.	64
5.1	Voice input	71
5.2	Text input	71
5.3	Example query made from the smart agent chat module. The user requests a sentiment analysis for the application named <i>Trello</i>	71
5.4	Sentiment analysis for <i>Trello</i>	72
5.5	Detailed sentiment analysis view	72
5.6	Dialog interface between user and the API.AI agent. By tapping the card (1) with global results, the <i>Trello</i> analysis details will be shown (2).	72
5.7	Voice input	73
5.8	Text input	73

5.9	Example query made from the smart agent chat module. The user requests a features classification for the application named <i>The Guardian</i> so they can extract improvements from the user feedback.	73
5.10	Features classification dialog for <i>The Guardian</i> app	74
5.11	Features classification details	74
5.12	Dialog interface between user and the API.AI agent. By taping the card (1) with global results, the <i>The Guardian</i> classification details will be shown (2), showing those reviews tagged as features.	74
5.13	Ranking list of top 60 free applications	76
5.14	Query result from search <i>arcade</i> category	76
5.15	Filter process inside the ranking static sentiment analysis placed in the smart agent.	76
5.16	Input parameters requested for developer-oriented scanning.	78
5.17	Result obtained after retrieving recent app reviews, classify them as a <i>bug</i> or not, and finally recommend a suitable reply.	78
5.18	Automated reply visualization inside Play Store website.	79

Tables Index

3.1	Actors list	30
4.1	Sentiment polarity representation	47
4.2	Emotions centroids representation, with their values for <i>arousal</i> , <i>dominance</i> and <i>valence</i> parameters	47
4.3	Dataset sizes for supervised learning. There are represented the row count about four datasets inside <i>Re_2015_Set</i> [2] (<i>Bug_Report_Data</i> , <i>Not_Bug_Report_Data</i> , <i>Feature_Or_Improvement_Request_Data</i> and <i>Not_Feature_Or_Improvement_Request_Data</i>	50
4.4	Classification parameters for both classifiers, bugs and features, after working with test dataset. Some of these parameters are accuracy, recall, f-measure and finally the most informative features for each classifier ordered by de- scending in possitive terms.	51
5.1	Scenario actors list	69
5.2	Execution environments	70
5.3	Results of sentiments and emotions analysis in project management applica- tions.	72
5.4	Results	74
5.5	Results obtained from query the ranking sentiments analysis	75

Introduction

This chapter provides an introduction to the problem which will be approached in this project. It provides an overview of the main objectives to achieve in this thesis. Furthermore, a deeper description of the project and its environment is also given, going deeper inside the context that surrounds the global idea of the system.

1.1 Context

Nowadays, app development for smart phone ecosystem is quite fast and easier than ever. A freelance developer can have an idea, implement it by himself and publish the app in the market with a low barrier of entry. As a mobile product developer, it is essential to place your apps in the top of the store rankings, maintaining your product by implementing new features and minor bug fixes.

However, iOS and Android market platforms only provide customer feedback with an average rating of 1 up to 5, and sometimes including a short review with the user experience. In most cases, this information is not enough to identify why your app has a bad acceptance, and consequently a bad positioning in the store.

This lack of data provided by marketplaces about how user feels using an app means a problem for developers. Mining the reviews of a certain application, and performing a sentiment or emotion analysis, we can obtain an implicit more detailed information about the product, identifying how the user feels, and evaluate his accordance with the product. Even comparing same user reviews in similar apps?, and make recommendations based on this data. Also bugs and features classification it's also a good implementation, so developer teams composed by low number of programmers can redirect the feedback obtained from the store previously tagged.

Furthermore, another problem is how companies should interact with their users to translate bad ratings into good ones, writing a well-formed reply. It's possible to implement an automated response system, where the smart agent picks the best reply depending on the bug or feature classification extracted from the review. This can also be applied to filter frequently asked questions (FAQ), performing an automated answer manager.

To carry out these solutions we have defined several objectives. First of all, develop a smart agent for Android devices where users can perform sentiments and emotions analysis for different Play Store apps. The smart agent interacts with the user through voice or text sentences, which will be converted into specific actions using a Natural Language Understanding system. Moreover, the system allows them to automate Play Store review response for applications developed by the user, depending on the classification result obtained. Finally, offering a service oriented to developers, where they can manage and analyze mobile application market status from their Android terminal.

To realize this notion, a number of software components interact. These modules are an

endpoint server module, API.AI conversational platform, mobile smart agent and Senpy.

Endpoint server module is the element that behaves as the brain of the whole integrated system. It provides a complete scalable framework for each project module, which redirect each work-flow implemented consequently, depending on the requested action. It can be accessed through an intuitive API REST service, using HTTP requests to communicate with the smart mobile agent, which acts as client side, with the rest of the smart system developed. Definitely inside the Model - View - Controller project design pattern, it plays the controller role handling every interaction performed between components

API AI Conversational platform is the tool responsible of parsing, recognizing and extracting the meaning of the sentence provided by the user, and act consequently. It provides us the natural language process layer, being able to create the conversational scenario, design corresponding actions and analyze interactions with users. The platform learns from examples provided by developers and conversations it has with end users to continuously improve user experience. Due to the huge variety of integrations developed by third parties, it enables to connect the smart system with other messaging platforms easily. In addition, cross-platform support gives us freedom to choose client and server development technology.

Mobile smart agent allows us to interact with the service through text or voice statements. It is developed for Android mobile devices and relies in API AI conversational platform for natural language processing and events recognition, attending API AI response and displaying the action result carried out. It also includes a developer-oriented section from where an app developer can schedule bug or feature classifications, automating their response and noticing the results to his development team using external third-party applications. All functionalities and use cases implemented within mobile app will be detailed in future sections.

Senpy lets you create sentiment analysis web services easily, fast and using a well known API. As a bonus, Senpy services use semantic vocabularies (e.g. NIF, Marl, Onyx) and formats (turtle, JSON-LD, xml-rdf). This tool aids to analyze the extracted mobile-app reviews and conclude a global sentiment or emotion that resume the users opinion.

1.2 Overview

Before explaining the main thesis goals and objectives, it's necessary to make a brief overview about how the mobile app reviews extraction works and how Google behaves within this process, both for users and app developers.

Due to privacy terms and integrity policies established by Google, it's really complicated to extract raw information about mobile apps reviews directly from market website in an easy way. For this reason, this data mining extraction process requires use tools or plugins created by third parties, in order to extract complete app reviews information. However, in these cases the extraction process is quite limited, being impossible to retrieve specific user information like recent downloads historic or the latest ratings carried out, which are really interesting for the project.

In this thesis we have focused on Android Play Store exhaustive analysis, but all this work and conclusions could be extrapolated to the iOS app store marketplace, or any other with similar appearance platform. There are some libraries that enables to interact with the marketplace through a restful application programming interface (API), but they are all outdated and doesn't work as expected. Actually, the best choice is to develop a standalone interface where we can obtain required information about an app by only providing its name or package.

All these obstacles decrease the creation of new app reviews data-sets, and consequently, it becomes hard to predict how users feel about an application, which seems interesting for app developers aims to improve the application performance. Due to of this lack of facilities, the project goals proposed below make more sense after this Google marketplace overview.

1.3 Master thesis goals

The main purpose of this master thesis is to develop an interactive smart agent for Android mobile devices which allows to run sentiments and emotions analysis for mobile applications published in Google Play marketplace. The agent will provide an automatic reply section only available for Android developers, which will process the output obtained from a trained topic classifier, automating the user reports answering and market feedback extraction, being able to categorize reviews in case it is a bug report, a feature request, some relevant user experience information etc.

After this reviews classification process, the system will have extracted the value of each

review, generating an optimal reply which helps to improve the application user rating, and increases the positioning inside best rated apps ranking of Play Store marketplace.

The idea of how upcoming mobile applications are developed has been changing recently. The application as a single software installed on user smart-phones is migrating to an *all-in-one* idea, where those apps grouped as a global entity are accessed through voice commands, using natural language assistants to solve users problems. The smart agent proposed in this thesis revolves around this premise, and intends to create an assistant oriented to mobile developers.

1.4 Structure of this Master Thesis

In this section we will provide a brief overview of all the chapters of this Master Thesis. It has been structured as follows:

Chapter 1 provides an introduction to the problem which will be approached in this project. It begins with an overview of the mobile apps marketplaces platforms and several emerging projects related with review processing that have come up recently. Furthermore, a deeper description of the project and its environment is also given.

Chapter 2 contains an overview of the existing technologies on which the development of the project will rely, and briefly exposes a state of art about several Natural Language Understanding platforms.

Chapter 3 describes the requirements analysis for the project and the actors that participates in them. The are mentioned most important functional requisites, presented as use case diagrams and explaining each actor involved.

Chapter 4 describes the project itself explaining the global architecture and how modules communicates each other. It begins with a brief description about the project structure, describing the most important components involved. Afterwards, we will go deeper into each module presented before, defining different workflows between them through several use cases. Inside this section, we will explain the sentiment analysis and classification techniques used in the thesis, and how natural language processing is used to perform sentiments and emotions analysis for each mobile application.

Chapter 5 describes a selected use case inside a specific scenario, presenting a *start-up* recently created that uses our system to improve their business labor. There are several scenarios explaining how the smart agent behaves in each case.

Chapter 6 sums up the findings and conclusions found throughout the document and gives a hint about future development to continue the work done for this master thesis.

Enabling Technologies

This chapter introduces which technologies have made possible this project. First of all there must be a short presentation about the programming language used in server and client side development, Python and Java respectively. Furthermore, we will go on explaining the emerging Google Actions system and its usage through the API.AI multiplatform restful service. We will continue with the sentiments and emotions analysis techniques used in the thesis, and how they are integrated in the project with Senpy [3]. To conclude, we will detail the procedures offered by Google Play Developers API to interact with their platform, being possible to extract and reply user opinions within our developed application.

2.1 Python

In this project we are going to use Python as the programming language for our software by server side. In general, Python is an efficient language that has three main characteristics that improve its efficiency: dynamically typed, concise and compact [4]. It has an easy syntax, high readability, it is object oriented and extensible.

All of these features aid in prototyping, shorten the development cycle and result in a cleaner, smarter and more effective code. It simplifies and accelerates the development process, also making it very accessible [5]. It is worth mentioning the large amount of libraries available for Python, which reduces the coding work.

However, it has some disadvantages, such as concurrency. Concurrency and parallelism, although completely possible in Python, are not designed-in for elegant use, as with JavaScript and Go. Other disadvantage is the speed, because Python is executed by an interpreter instead of compilation, which causes it to be slower than if it was compiled and then executed. However, for most applications, it is by far fast enough.

2.1.1 Flask

Flask is a micro web framework written in Python and based on Werkzeug toolkit and Jinja2 template engine [6]. Flask is called a micro framework because it does not presume or force a developer to use a particular tool or library. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions [7].

However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools.

In this thesis we have applied this generalized web framework to develop an intermediate controller server, which manages the communication between the smart agent and the logic implemented to carry out the action requested. Flask aids to integrate all these modules and simplify our service, obtaining a restful API REST from where our agent or any third party developed application can interact with the mobile app monitoring system purposed in this project.

2.2 Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA) [8], meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

If we go deeper, referring to the project explained in this document, we have used Java inside Android environment, creating a smart agent that extends several libraries and connects with the rest of the system modules. More specifically, we have used Android Studio developer toolkit, which provides a complete user-friendly framework for Android mobile apps development in native way, to create the mobile client. This means that all the classes extend directly from super classes provided by Java.

Moreover, in this project we have intended to follow the basic standard design guides for app development established by Google, more commonly named as Material Design guidelines [9], which are very extended in every native application nowadays.

Now we are going to adapt these Java concepts to the Android development framework, explaining some basic concepts about how most applications works.

2.2.1 Android basis

In this part we are going to introduce how an Android application basically works, explaining basic elements involved in the smart agent development and used by most of the applications. With this we intend to help understanding how the application is structured, previously defining the platform key concepts. Inside every Android application, exists multiple choices in what pattern design refers, but most of them present the same component structure [10]. App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app. Some components depend on others.

There are four different types of app components. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed. The next sections describe the four types of app components [11]:

- **Activity:** An activity is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity

that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. In this thesis each module is composed by a main activity for service displaying, and some secondary activity to display specific contents such as detailed analysis or classification results.

- **Service:** A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

For our case, the chat agent module binds the API.AI communication service, in order to send in background the user queries using the SDK functions, being synchronized with the server in every moment.

- **Broadcast receivers:** A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. For our application, we use a receiver to listen upcoming events which are received from the API.AI as answer of user queries, summoning some specific functions inside the app like add a new *CardView* with the analysis result, or a failure message in case it doesn't trigger an intent remotely.
- **Content providers:** A content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content provider allows it. For example, the Android system provides a content provider that manages the user's contact information. For this application we used the system preference to store the static analysis information, so it weren't necessary to use this component.
- **Intent:** Intents bind individual components to each other at runtime. You can think of them as the messengers that request an action from other components, whether the component belongs to your app or another. For activities and services, an intent defines the action to perform, for instance, an intent might convey a request for an

activity to show an image or to open a web page. For broadcast receivers, the intent simply defines the announcement being broadcast.

Some other components are used in our application, like **asyncTasks**, that handle background process in order to send the HTTP requests and process the response asynchronously so the app don't freezes in case it takes some time. Moreover, **adapters** are also important components inside the smart agent because they link the data with the view objects. Also, the **SystemPreferences** extension enables to persist relevant data over sessions, being recoverable every time the user launches the application, and able to save there the chat historic, the authentication credentials or environment variables.

2.3 Natural Language Understanding Systems

Over the past few years, natural language interfaces have been transforming the way we interact with technology. Voice assistants in particular have had strong adoption in cases where it's simpler to speak than write or use a complex user interface. This becomes particularly relevant for IoT, where devices often don't have touchscreens, making voice a natural interaction mechanism.

Building a robust AI assistant, however, is still rather complicated. A number of companies are addressing this issue by providing natural language understanding (NLU) solutions that developers can use to augment their product with natural language capabilities.

In this section we are going to compare those NLU [12] systems approach and evaluate services they offer to developers. Some of the most relevant services NLU oriented are shown in Fig. 2.1.







	On-device solution	In-house server hosting	Third-party server hosting	Private mode on third-party server?	Built-in intents	Custom intents
 Alexa	✗ No	✗ No	✓ Yes	✗ No	✓ Yes	✓ Yes
 Api	✗ No	✗ No	✓ Yes	✓ Yes	✓ Yes	✓ Yes
 Luis	✗ No	✗ No	✓ Yes	✗ No	✓ Yes	✓ Yes
 Siri	✗ No	✗ No	✓ Yes	✗ No	✓ Yes	✗ No
 Snips	✓ Yes	✓ Yes	✗ No	N/A	✓ Yes	🕒 Soon
 Watson	✗ No	✗ No	✓ Yes	✗ No	✗ No	✓ Yes

Figure 2.1: Comparative between most services, evaluating most important categories offered by them [1] .

All these conversational services currently work in a similar way [1]:

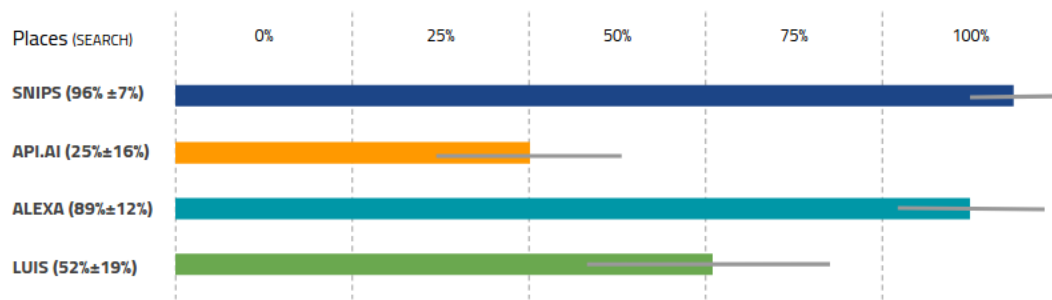
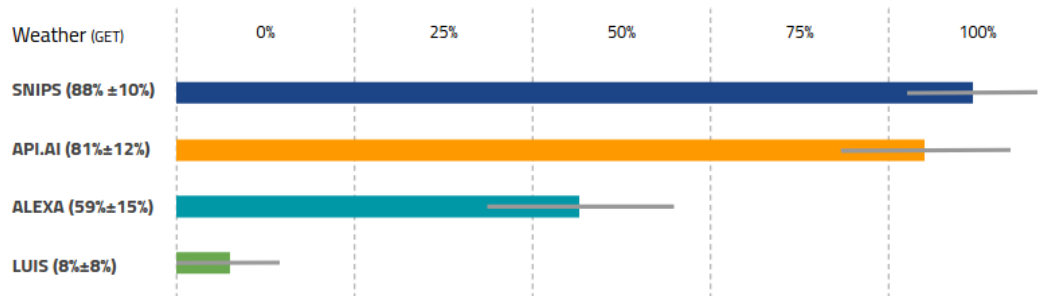
- Natural language is sent to the service, for instance *where can I buy some food*
- The intention is detected, for this example the intention of the query would be *search place*.
- The specific parameters are extracted, in our case *type: food place, coordinates= 34.34, 55,78*
- A structured description of the user query is sent back

In order to compare these natural language understanding services each other, it's necessary to evaluate them inside for each functionality, analyzing their behaviour inside several scenarios and concluding relevant conclusions about their optimal usage. We can distinguish two main steps inside understanding process field:

Detecting user intentions - Each NLU system uses their own ontology for intents understanding, so we first have to build a baseline by creating a joint ontology across services [1]. Different domains must be identified, and similar intents have to be grouped together in performance estimates. According to the benchmark, the metric provided is the fraction of properly classified queries for each intent, along with its 95% confidence interval. The latter enables more robust conclusions, given that the size of the test dataset is limited. Fig. 2.2 and 2.3 show some of the most commonly used bundles for intent validation, which are *Places*, *Reservation*, *Transit* and *Weather*.

These results suggest that intent classification remains a difficult problem, on which competing solutions perform quite differently. *Luis* is significantly outperformed by completing solutions. *API.AI* and *SNIPS* generally yield indistinguishable performances, aside from place search where the *API.AI* accuracy decrease compared with other tools. To conclude this section, *Alexa* and *SNIPS* systematically rank among the best solutions on the intents they were tested on.

Parameters identification inside the query - The second step of the NLU process is to extract the parameters related to the detected intent, a process commonly called "slot filling". Again, each service has defined its own system of intents, and each intent comes with its specific set of parameters. The intents covered in this benchmark have between 1 and 18 parameters (slots). In figure 2.4 are represented most common NLU systems in parameter recognition terms, obtaining their precision for relevant topics, which are again *Places*, *Reservation*, *Transit* and *Weather*.

Figure 2.2: Evaluation process for intent detection applied to *Places* extension 2.4.Figure 2.3: Evaluation process for intent detection applied to *Weather* extension 2.4.

The results show that all NLU systems behave similarly in terms of precision. For all solutions, average precision per intent spans between 50% and 90%. The later suggest that all solutions are perfectible, and have a certain risk of misinterpreting user queries.

This benchmark shows that there is currently no platform that stands out about the others. More precisely, there is no solution that doesn't misinterpret user queries, and no solution that understands every query. Unlike other areas of Artificial Intelligence, machines haven't reached yet the level of human performance when it comes to NLU.

What this benchmark also shows is how important such tests and methodologies are needed. There are important differences between solutions available to developers. As the saying goes, you can't optimize what you don't measure. At least we now know that some solutions lack robustness when it comes to variations in how things are asked, while all of them would benefit from improvement in how they fill slots.

After this NLU systems evaluation process we decide to choose the API.AI system which relies on Google Actions environment. We have chosen the API.AI environment because it has perfect compatibility with Google Home device, also offering a SDK for Android developers which enable us to create the smart agent for Android framework, instead of other systems like *SNIPS* that only supports Raspberri Pi implementations.

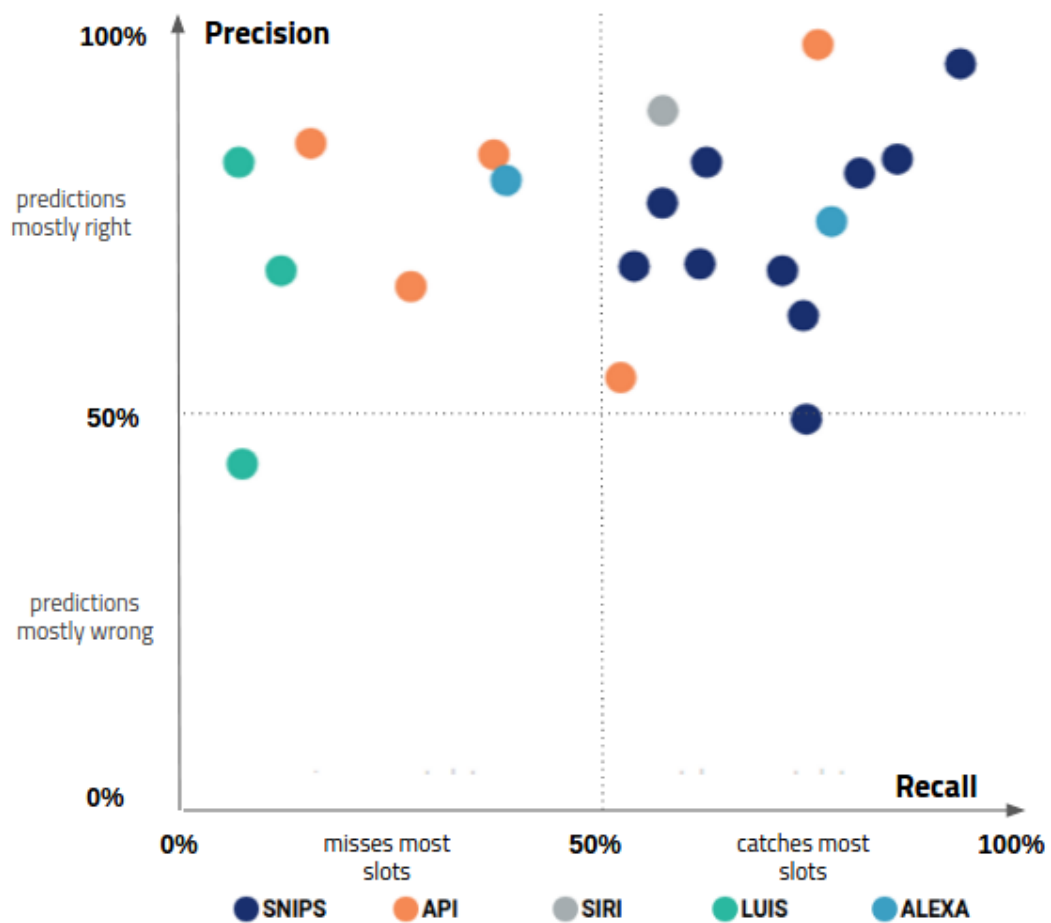


Figure 2.4: Graph which represents precision over recall in parameter identification for most common NLU services 2.4.

2.4 Google Actions

Actions on Google provide all the tools you need to build action. There are also many third-party tools that supplement and enhance the developer experience. Inside all possibilities offered by Google actions engine, we will focus on actions triggered in conversational environments.

Conversational Actions help you fulfill user requests by letting you have a two-way dialog with users. When users request an action, the Google Assistant processes this request, determines the best action to invoke by performing a pattern recognition analysis, and invokes your Conversation Action if relevant. From there, your action manages the result, including how users are greeted, how to fulfill the user's request, and how the conversation ends.

When you build conversation actions, you must define some several components:

Invocation triggers define how users invoke and discover your actions. Once triggered, your action carries out a conversation with users, which is defined by dialog.

Dialogs define how users converse with your actions and act as the user interface for your actions. They rely on fulfillment code to move the conversation forward.

Fulfillment is the code that processes user input and returns responses and you expose it as a REST endpoint. Fulfillment also typically has the logic that carries out the actual action like retrieving recipes or news to read aloud.

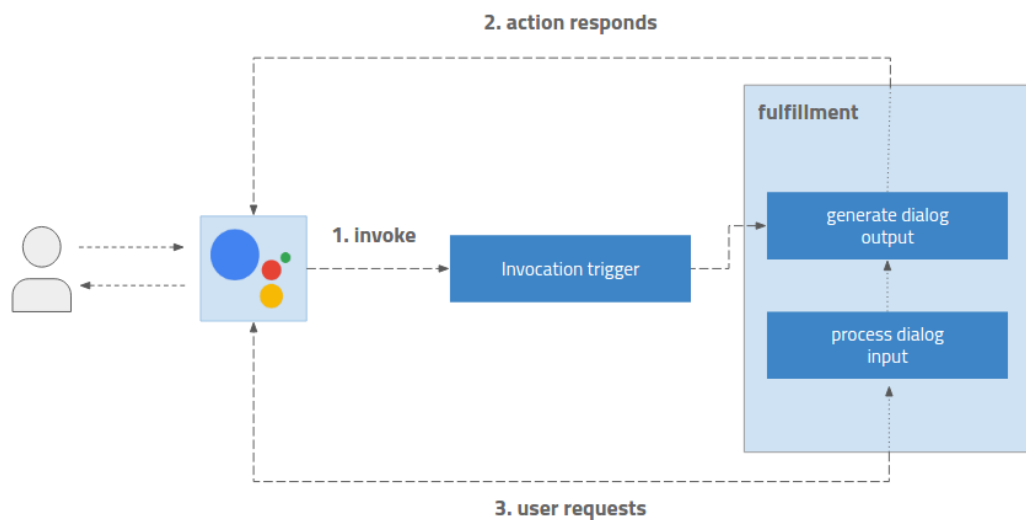


Figure 2.5: Conversation action lifecycle inside Google Actions environment.

As can be observed in figure 2.5, once the event is invoked, the whole conversational process becomes cyclical, obtaining an action from the request made by the user. All of these components require you to communicate with the Google Assistant in a standard way, which is defined by the **Conversation API**.

2.4.1 Conversation API

The Conversation API defines a request and response format that you must adhere to when communicating with the Google Assistant. Even though you control the user experience when your action is invoked, the Google Assistant still brokers and validates the communication between your action and the user. This procedure is represented in figure 2.6.

We can differentiate between requests and responses inside Conversational API communication module:

Requests contain information such as the user input, information about the user and the device the request came from, or other relevant data you need to process. The Google Assistant sends requests to your actions when it wants to invoke them, and when it has user input that needs to be sent to your action processing.

Responses that you return to the Google Assistant contain information such as the text to speak back to users and whether or not you expect the user to provide additional information. Your actions must send a response to the Google Assistant every time it receives a request.

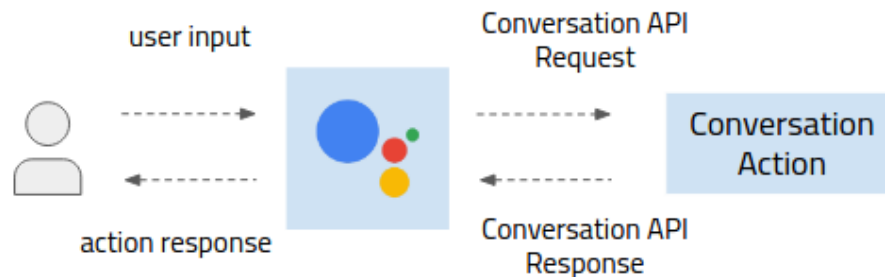


Figure 2.6: Conversation API workflow representation including requests and responses.

In order to help with requests and responses construction process, Google Actions provide several SDKs and tools available for multiple develop platforms. In this thesis, since the smart agent has been developed for Android devices, the API.AI framework seems to be the best implementation choice to connect the smart-phone and the Google Actions remote server, which will interpret the sentences provided by the end user.

2.4.2 API.AI

The Actions SDK gives you all the tools you need to build Conversation Actions, but it's preferable to use one of the supported tools to build them, such as API.AI. These tools generally provide a better developer experience by making it easier to build and deploy actions within a single interface and also provide additional features to make building actions easier.

The API.AI tool gives us some extra conveniences for our project development, such as:

API.AI NLU - The *API.AI Natural Language Understanding* is integrated into API.AI and does in-dialog processing of the input provided. This offers us with conveniences such as natural expansion of phrases and built-in features to easily distinguish and parse arguments from user input.

Furthermore, this avoids us the development of a natural language to text converter, and a post processing of the sentence obtained, performing a pattern recognition analysis to retrieve the true sense of the phrase.

Web GUI where we can define and configure action components, such as dialogs or invocations, in an easy-to-use user interface.

Conversation building features , that let us contextualize user requests and apply previously built machine learning algorithms to better understand the user request and compose a continuous conversational flow.

Moreover, the API.AI framework acts as intermediate between the user and the Google actions structure explained in figure 2.5. Inside all the possibilities offered by this toolkit, we are going to go deeper into some of them, focusing on detail those elements which are crucial inside the conversational process. In figure 2.7 is presented how API.AI is related to other components and how it processes data.

We are going to keep on with a brief presentation of some key concepts involved in any API.AI implementation.

Agents can be described as NLU (Natural Language Understanding) modules for applications. Their purpose is to transform natural user language into actionable data. This transformation occurs when a user input matches one of the intents or domains. Intents are developed-defined components of agents, while domains are pre-defined knowledge packages that can be enabled or disabled in each particular agent. Moreover, they can be designed to manage a conversation flow in a specific way by using some of the components explained below.

Entities represent concepts and serve as powerful tool for extracting parameter values from natural language inputs. The entities that are used in a particular agent will depend on the parameter values that are expected to be returned as a result of agent functioning. In other words, a developer does not need to create entities for every concept mentioned in the agent, being necessary only for those that require actionable data.

There are three types of entities:

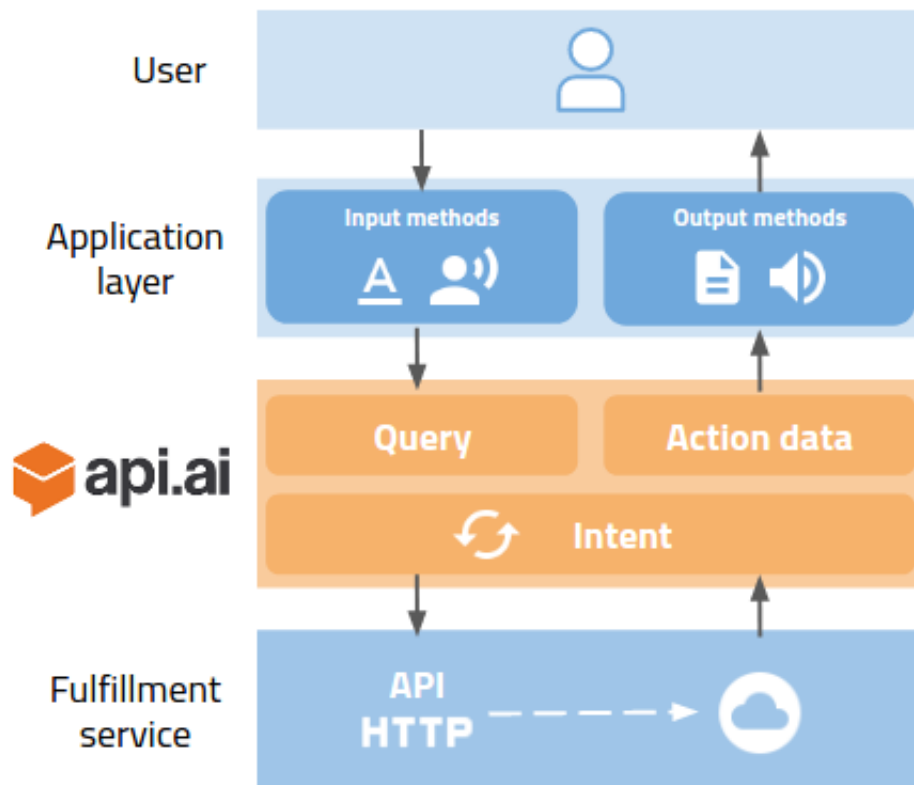


Figure 2.7: API.AI architecture inside application scenario.

- **System entities:** These are pre-built entities provided by API.AI in order to facilitate handling the most popular common concepts. Some examples of this category could be words referring to colors (red, blue yellow...), language names like Spanish or French or common given names, for example John or Mary.
- **Developer entities:** These refer to all the entities that can be created by the user. These entities can be designed in several different ways. Because of that, we can distinguish between different developer entity subtypes. First of all, *Developer Mapping Entities*, which allows the mapping of a group of synonyms to a reference value. For example, a good list of entries for music styles could be *jazz, rock, pop, indie, metal,* Secondly, *Developer Enum Type Entities*, that contains a set of entries that do not have mappings to reference values. Lastly we have *Developer Composite Entities*, which are basically enum type entities

whose entries contain other entities used with aliases. This type of entities are most useful for describing objects or concepts that can have several different attributes.

- **User entities:** These are entities that are defined for a specific end-user, for example a recommendation entity that has generic recommendations could be defined on a request or for a given session.

Intents represents a mapping between what user says and what action should be taken by your software. An intent is composed by several modules. First of all, what user says in natural language is required. Then it's necessary to set up the corresponding action, and the response, which is provided by the external application service.

Contexts are designed for passing on information from previous conversations or external sources, such as user profile or device information. Also, they can be used to manage conversation flow.

2.4.2.1 Machine Learning

Machine Learning [13] is a tool provided by API.AI platform that allows your agent to understand user inputs in natural language and convert them into structured data, extracting relevant parameters. In the API.AI terminology, your agent uses machine learning algorithms to match user requests to specific intents and uses entities to extract relevant data from them.

The agent *learns* both from the data is provided in it and from the language models developed by API.AI. Based on this data, it builds a model for making decisions on which intent should be triggered by a user input and what data needs to be extracted. The model is unique per agent.

It's possible to approach agents behaviour to humans one in a certain way. For example, if we need to teach something new to our agent, we start training it by adding new intents and entities. Machine learning allows the agent to make decisions based on the new data. It can make mistakes at first as humans do, but the more trained it is, the better it will work.

2.5 Senpy

Senpy [3] is a technology developed by the GSI of ETSIT-UPM. It is an open source software and uses a data model for analysis of feelings and emotions. It is based on NIF, Marl and Onyx [14] vocabularies.

It aims to facilitate the adoption of the proposed data model to analyze feelings and emotions, so that the services of different providers are inter-operable. For this reason, the design has focused on its extensiveness and reuse.

Some of the plugins defined in Senpy platform are:

- *emoTextAnew*: it extracts the VAD (Valence - Arousal - Dominance) of the sentence provided by matching words from the ANEW dictionary[15]. Once the VAD triple is obtained, we can compare it with the nearest emotion.
- *emoTextWordnetAffect*: it is based on the hierarchy of WordnetAffect [16] to obtain the emotion of the sentence provided.
- *sentiText*: it refers to a software developed during TASS 2015 competition and has been adapted for multilingual purpose for English and Spanish.
- *meaningCloud*: it is based on the sentiment analysis API developed by Meaning Cloud company. It requires previous authentication with the server.
- *affect*: it is created to perform a sentiments and emotion analysis simultaneously.

In this thesis we have focused on *sentiText* and *emoTextAnew* plugins to carry out sentiments and emotions analysis respectively. The connectivity with Senpy module is implemented through a Flask server in Python [4]. The interaction between Senpy, which is hosted in a remote cluster, and our server is implemented using an API REST, making HTTP request and filling all the parameters needed.

2.6 Google Play Developer API

The Google Play Developer API allows to perform a number of publishing and app-management tasks. It includes two components, the Subscriptions and In-App Purchases API and the Publishing API, which is the one used in this project.

Inside Publishing API we can find some useful APIs related with reviews request and response such as Reply to Reviews API service.

2.6.1 Reply to Reviews API

The Google Play Developer API allows you to view user feedback for your app and reply to this feedback. You can use this API to interact with users directly within our server controller.

The Reply to Reviews API enables you to access feedback only for production versions of your app. If you want to see feedback on alpha or beta versions of your app it is necessary to use the Google Play Console instead. Also, note that the API shows only the reviews that include comments. If a user rates an app but does not provide a comment, their feedback is not accessible from the API.

We will briefly explain how the API works, detailing the information that can be retrieved from it.

Gaining Access is mandatory for API usage. You have to provide authorization using either an OAuth client or a service account. Moreover, you can only retrieve/answer reviews for applications uploaded by yourself in the logged account.

Retrieve reviews requires to make a HTTP GET request to the API once authentication is granted. The application package name and the authorization token must be attached. The response obtained will follow the JSON structure defined below. Most relevant information are the review itself, the rating, publish date, and device information.

```
{
  "reviews": [
    {
      "reviewId": "12345678",
      "authorName": "Jane Bloggs",
      "comments": [
        {
          "userComment": {
            "text": "This is the best app ever!",
            "lastModified": {
              ...
            },
          },
          "starRating": 5,
          "deviceMetadata": {
```

```
        ...
    }
}
},
{
    "developerComment": {
        "text": "That's great to hear!",
        "lastModified": {
            ...
        }
    }
}
]
}
}
```

Retrieve an individual review is also possible by giving the specific review id that we want to obtain.

Translation language can be attached to the HTTP request and review comment will be automatically translated to that language.

Reply to reviews requires to make a HTTP POST request to the API. The application package name, reply text, answered review id and the authorization token must be attached.

```
{
  "result": {
    "replyText": "Thanks for your feedback!",
    "lastEdited": {
      "seconds": "1453978803",
      "nanos": 796000000
    }
  }
}
```

As courtesy to other developers, the Reply to Reviews API enforces several limits according to prevent flooding over their service. Those limits are **60** GET requests for retrieving lists of reviews and individual reviews per hour and **500** POST requests for replying to reviews.

Requirement Analysis

The result of this chapter will be a complete specification of the requirements, which will be matched by each module in the design stage. This helps us also to focus on key aspects and take apart other less important functionalities that could be implemented in future works.

3.1 Use Cases

These sections identify the use cases of the system. This helps us to obtain a complete specification of the uses of the system, and therefore define the complete list of requisites to match. First, we will present a list of the actors in the system and a UML diagram representing all the actors participating in the different use cases. This representation allows, apart from specifying the actors that interact in the system, the relationships between them.

These use cases will be described the next sections, including each one a table with their complete specification, and adding a use case diagram to clarify the concepts. Using these tables, we will be able to define the requirements to be established.

3.1.1 Actors dictionary

In this section we will resume all the actors that are involved in any use case. These use cases we be detailed in depth in sections below.

Actor identifier	Role	Description
ACT-1	User	End user that uses the mobile application, sending queries to the API.AI calling specific intents
ACT-2	API.AI agent	Refers to the agent hosted remotely in the API.AI natural language understanding system. It can accept different inputs and manage the application flows over intent calling.
ACT-4	Server	Controller of the whole system. Its where relies the project logic code, offering an API REST interface to communicate with the different submodules contained.
ACT-5	Senpy Platform	Remote restful service that processes HTTP requests and perform a sentiment or emotion analysis for the input sent.
ACT-6	Play Store website	Google application official marketplace where developers can publish their software. It contains apps information and user ratings and opinions about their experience.
ACT-7	Google Play Developers API	Service that enables to interact with the Play Store reviews, being able to reply them once the developer is authenticated.

Table 3.1: Actors list

3.1.2 UC1: Request a new analysis

Use Case Name	Request a new analysis.
Use Case ID	UC1
Actors	User, Senpy platform, API.AI agent, Play Store website and Server.
Pre-Condition	The mobile application must be linked with a valid API.AI agent
Post-Condition	-
Flow of Events	<p>1.- The user sends a new analysis query using text or voice channels through the chat interface.</p> <p>2.- The API.AI agent processes the input received from the user as a JSON file. It carries out an intent recognition analysis, and in case it calls an <i>Analysis</i> task, a web-hook request is made.</p> <p>3.- The server processes the request, extracts the application info from Play Store website and carry out the analysis with Senpy. The result is returned to the smart agent.</p>

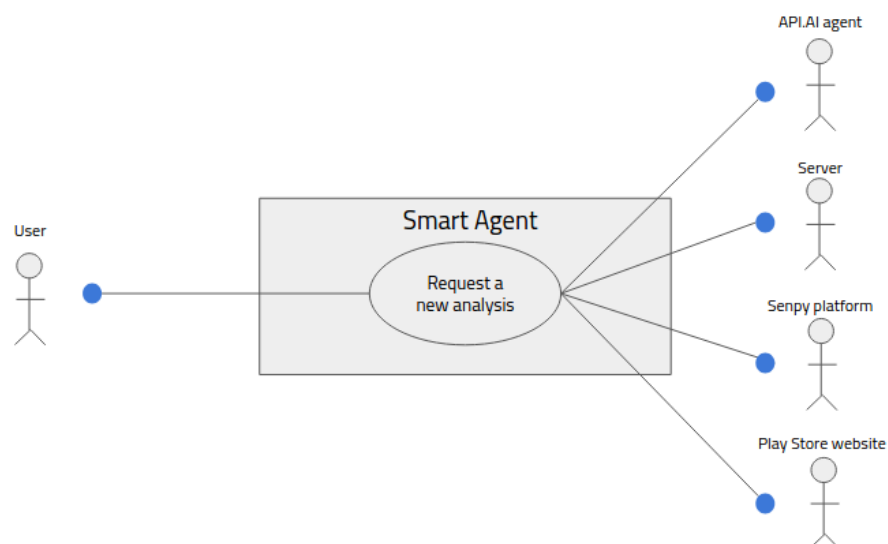


Figure 3.1: UC1. Request a new analysis

3.1.3 UC2: Request a new classification

Use Case Name	Request a new classification.
Use Case ID	UC2
Actors	User, API.AI agent, Play Store website and Server.
Pre-Condition	The mobile application must be linked with a valid API.AI agent. Both classifiers must be trained previously.
Post-Condition	-
Flow of Events	<ol style="list-style-type: none"> 1.- The user sends a new classification query using text or voice channels through the chat interface. 2.- The API.AI agent processes the input received from the user as a JSON file. It carries out an intent recognition analysis, and in case it calls a <i>Classification</i> task, a web-hook request is made. 3.- The server processes the request, extracts the application info from Play Store website and run the classifier for each review. The classification result is returned to the smart agent.

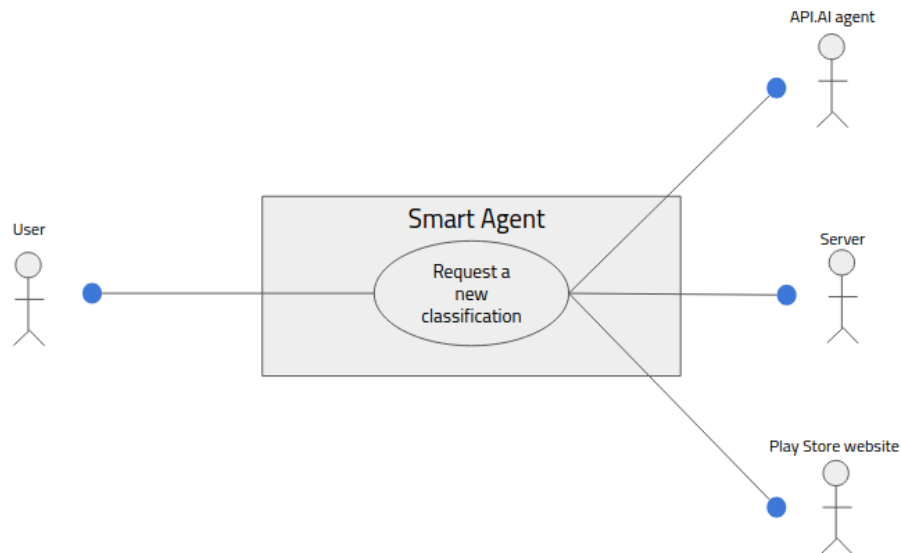


Figure 3.2: UC2. Request a new classification

3.1.4 UC3: Authenticate with Google Play Developers API

Use Case Name	Authenticate with Google Play Developers API.
Use Case ID	UC3
Actors	User and Google Play Developers API.
Pre-Condition	-
Post-Condition	-
Flow of Events	<ol style="list-style-type: none"> 1.- The user authenticates with the Google Play Developers API. 2.- If success the server provides an authentication token which should be saved for future requests to the GPD API.

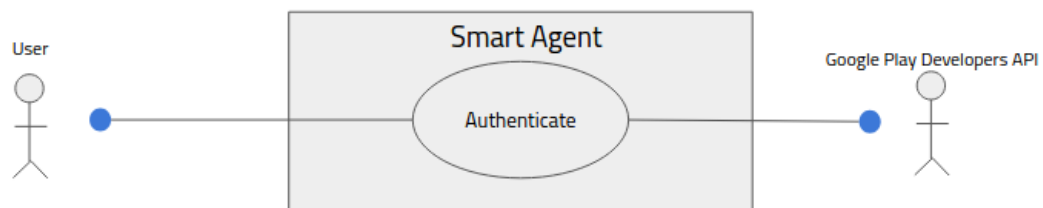


Figure 3.3: UC3. Authenticate

3.1.5 UC4: Request a review

Use Case Name	Request a review
Use Case ID	UC4
Actors	User and Google Play Developers API.
Pre-Condition	The user must be authenticated with the Google Play Developers API. App must have been published by that account
Post-Condition	-
Flow of Events	<p>1.- The user requests to the Google Play Developers API one or multiple reviews for a self-developed app. In case it's a single review the id must be attached.</p> <p>2.- If the app belongs to the authenticated account or has permissions, the Google Play Developers API return a JSON response with all the reviews made for the app during the last 15 days.</p>



Figure 3.4: UC4. Request a review

3.1.6 UC5: Reply a review

Use Case Name	Reply a review
Use Case ID	UC5
Actors	User and Google Play Developers API.
Pre-Condition	The user must be authenticated with the Google Play Developers API. App must have been published by that account
Post-Condition	-
Flow of Events	<p>1.- The user posts to the Google Play Developers API one reply for specific review of a self-developed app.</p> <p>2.- If the app belongs to the authenticated account or has permissions, the Google Play Developers API publishes the replied text in the Play Store marketplace answering the review.</p> <p>The Google Play Developers API returns a JSON response with the success code of the operation.</p>



Figure 3.5: UC5. Reply a review

Finally, figure 3.6 shows a diagram with all the use cases presented above.

3.2 Requisites capture and analysis

According to the use cases presented above, we are ready to collect the general requisites that should be accomplished in this thesis. We will also give them a priority so we can define those use cases that are most important to achieve. The functional requisites established for this project thesis are:

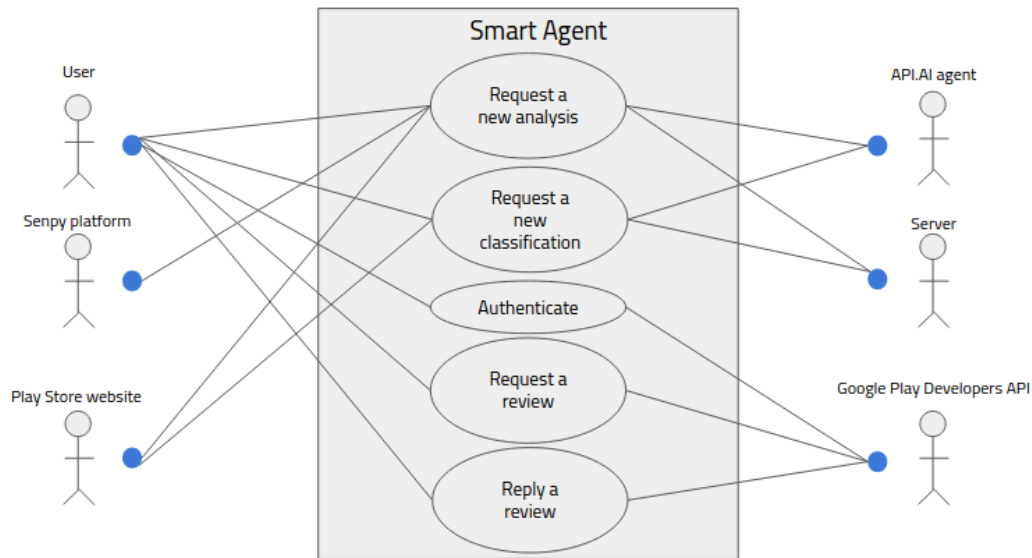


Figure 3.6: Use cases diagram

- Develop a mobile interface from where the user can request sentiments and emotions analysis for applications published in the Play Store market - **Mandatory** (*UC1*)
- This interface must be compatible with the Google Action service, using the API.AI Natural Language Understanding system, sending text or voice queries to the remote agent - **Mandatory** (*UC1*, *UC2*)
- Develop a bugs and features classifier able to determine if a certain review can be categorized as one of these concepts - **Desirable** (*UC2*)
- Implement a developer oriented screen to review reply automation, using the API REST offered by Google Play Developers. The reply will depend on the previous classification result - **Desirable** (*UC3*, *UC4*, *UC5*)
- Evaluate how the sentiment evolves over the most popular applications in order to apply these results to innovate and create similar products by freelance developers - **Optional** (*UC1*)

Architecture

This chapter describes in depth how the system is structured in different modules and how the users interact with them and also how the modules interact with other modules by themselves. First of all we will present the global architecture of the project and we will continue explaining each module in depth.

4.1 Overview

In this section we will explain the architecture of the project from a global perspective, including an overview about each element that interacts in the system, and describing how they have been implemented.

We can divide the project into three groups, the server side, the API.AI agent based on Google actions and lastly the mobile smart agent. The complete global architecture of the system is shown in Fig. 4.1.

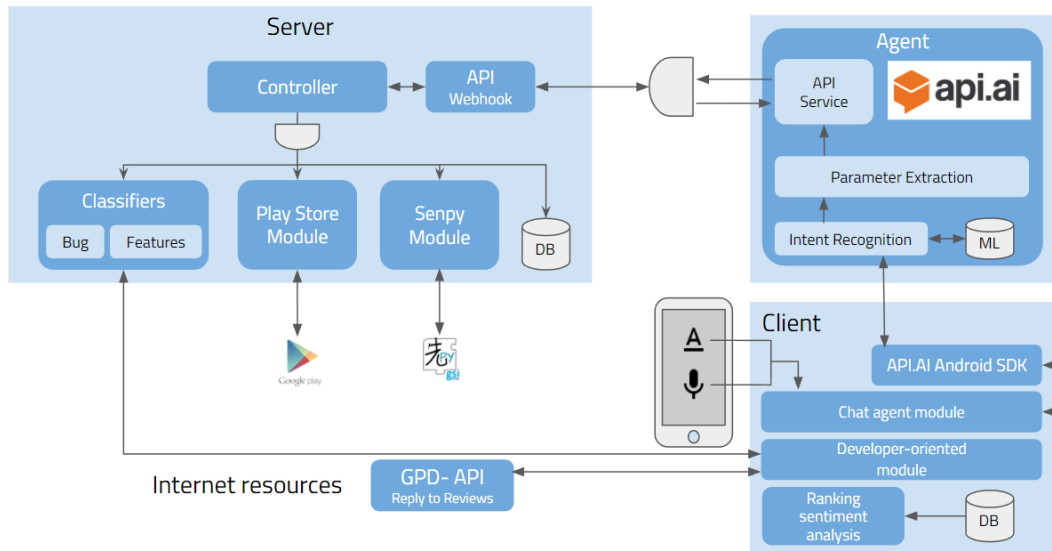


Figure 4.1: Global architecture of the system developed.

4.1.1 Server Module

Server side section is composed by several modules. At first, we have the controller class from where every request made to the server is handled. This controller adapts the entrance parameters to the final components which will carry out single functionalities.

All the requests are received through the API webhook, that is linked to the API.AI agent. The API REST is developed in Flask and contains multiple methods that redirect depending on the application work flow.

The controller interacts with three submodules. The Play Store module is formed by all the procedures required for app info extraction from Google Play website. Scrapping and filter tasks are performed inside this component. Afterwards, the Senpy module is responsible for connecting with the sentiments and emotions analysis. In this class we can

find some functions to adapt and process the response received from Senpy service. Finally, the last module is composed by bug and features classifiers which are previously trained. Each of these components interacts with external applications, like Google Play, Senpy or Slack by extracting relevant information from their website, or using other API REST services.

Some of the information obtained inside this modules is saved in a database, such as recent analysis carried out or the trained classifiers pickles, in order to cache most requested information and speeds up the communication process between the smart agent and the server.

4.1.2 API.AI Agent

The API.AI module acts as an intermediary between the logic hosted by the server and the smart mobile agent. All this behaviour is orchestrated by an Agent remotely configured using the API.AI web interface.

This component is responsible for interpreting the request made by the smartphone, translating the voice clip into a text sentence, identifying the meaning of the query and extracting the relevant parameters with pattern recognition and carry out the consequent action, which might be simple such as replying with a simple text string, or depends from the controller server being necessary to communicate with the API webhook.

Then, we are going to introduce the structure of the API.AI agent, explaining the intents and events implemented to carry out the client requests. This aids to understand the basic model of the agent and its architecture.

Intents refers to actions that our agent will execute. These intents could have dependencies from previous interactions with the agent, so it's necessary to contextualize each one of them. The intents handled by the agent are:

- **Analyze:** Reflects the analysis action. The agent receives the analysis type and the application desired to be analyzed as input parameters, offering the Senpy [17] analysis result as output. This intent use has webhook option enabled because it needs to interact with the developed server.
- **Classify:** it represents the bug or feature classifier call, being necessary to communicate with the remote server. The input parameters are the classification type and the application in case it can't be extracted from the conversational context.

Every intent is prepared for being summoned with missing parameters, training the API.AI agent to request those that are mandatory, or package them inside the conversation context, from where the system can extract them. For example, if the *classify* intent is called like *Would you kindly perform a sentiments analysis for WhatsApp application*, the system collects the app name parameter and the analysis type for a while, in case future requests have this values empty. Afterwards, if we ask the agent *Could you perform a bug classifier*, The bot will extract the missing app name parameter from the context, referring to the last app used in the conversational process.

This humanizes the natural language understanding and accelerates the conversation, interacting with the bot through real natural sentences instead of sending simple commands or instructions to a robot.

Entities refers to the element which represents concepts involved in the intent event triggering. Following the different types of entities explained in chapter 2, the entities defined for the agent are:

- **App:** refers to the application that wants to be treated. This entity hasn't an static value because it covers all the mobile applications published in the Google Play marketplace, so the value of the entity must be implicit in the request and may be different for each case.
- **Analysis Type:** refers to the analysis types that can be performed. The value is bounded, being possible to run a sentiments or emotions analysis exclusively. The scalability offered by the API.AI framework enable us to add more analysis types in the future.
- **Classifier Type:** specify which type of trained classifier would you like to call. The available values for this attribute are also limited to bug or feature.

The agent has also enabled the *Small Talk* bundle, that includes predefined phrases to the most popular requests, what makes the bot to look more like a human.

4.1.3 Client Module

The client module is developed through an Android application that receives user requests and redirects them directly to the API.AI agent. This behaviour is implemented through the API.AI extension library, which provides us all the needed methods to talk with the API.AI service.

The application is structured in several modules regardless each other. The main module manages the voice and text inputs from user in a chat bot interface, and is linked to API.AI, representing the response obtained from the server. The second module is developer-oriented, and requires previous authentication for usage. It allows to interact with self-developed apps published in Play Store, being able to define automatic response for certain reviews. Lastly, we can find an extended analysis of the top free apps in the Play Store marketplace. The analysis shows an evolution of the app sentiment over time, being able to extract conclusions about the polarity evolution among most downloaded apps, providing the final user an extended overview about products which tend to success recently in the global app market.

These three sub modules build the mobile smart agent, performing different functions to improve the feedback obtained from the market and manage that information for our own benefit inside this thesis.

4.2 Server Module

In this section we are going to go deeper inside the server side architecture, detailing how each module has been implemented and communicates with the rest of the system. We will begin introducing the *brain* of the server part, also known as controller, and we will go on with their functional modules, explaining the classifier algorithms at last.

4.2.1 Controller

The controller of the server acts as the manager of the server and every call made from the agent passes through this component. It is directly linked with the API REST webhook, which defines the different functions that can be summoned from the outside. Due to the weird API.AI architecture, it only allows to redirect server requests to a single endpoint, being able to provide a common fulfillment URL for all the intents. For this reason, the controller will have to filter the calls received in the general API address, extract the desired action and proceed with the action itself. Anyway, the server also accepts individual calls to the API for regardless modules, offering an operable access to those calls, in case it's necessary in future implementations.

The available API calls that can be made to the endpoint server are presented in figure 4.2. Some of these calls could require others summoning, for instance in case the user claims a sentiments analysis about an application, it's necessary to previously obtain the reviews

from the market, so it would implicitly call the *getAppsInfo* in case it's not cached.

Method	Name	Description
GET	<i>/getAppInfo</i> · app name	Retrieve complete information about the app requested. This information is scraped from the Play Store website and inserted in a JSON file.
POST	<i>/analyze</i> · appName · analysisType · maxReviews	Perform an analysis for specified application, obtaining the information from Play Store if necessary. Analysis types are sentiments and emotions.
POST	<i>/classify</i> · appName · classificationType · maxReviews	Perform a classification for specified application, obtaining the information from Play Store if necessary. Classification types are bugs and features.
GET	<i>/checkQueue</i> · taskId	Check a task status inside the server queue in case it requires a long execution time and the API.AI agent throws a request timeout exception.

Figure 4.2: Listing of the API REST calls enabled in the server. HTTP Method, request name and required parameters are specified with a brief description about the response obtained.

In order to ease the analysis and classification tasks, it is implemented a caching system inside the controller. The controller basically caches the response generated by a call for a limited period of time, so it can be directly obtained without processing it again if requested. This basically reduces the amount of requests carried out, decreasing the overload inside external platforms like Senpy [17] analyzer or Play Store website, and prevent redundant API calls.

Once we have explained the different requests accepted by the server we are going to present through several sequence diagrams how each module behaves when once of these

API calls are triggered, as shown in Fig. 4.1

4.2.2 Play Store module

This module handles the communication with the scraper which extracts relevant information for a concrete mobile application. The entrance parameter is only the application name, so it's required to perform some data mining procedures till we obtain the final result. Searches inside Play Store website works with the package name instead the application name because there could exist several application with the same name, but not with the same package name, so our first step to accomplish is retrieving the package name of the first application obtained from search in the web. Afterwards, once we have the package name we can search the application and mine the reviews. The process proposed in this module is:

1. Retrieve the package name by making a HTTP request to the Play Store search engine ([https://play.google.com/store/search?q="+my_app_name+"&hl=en](https://play.google.com/store/search?q=)) and obtain the response. The list of results obtained could be composed by applications, music, books or any other content provided in the market with that name.
2. Filter the search results and discard those which aren't mobile applications.
3. Extract the most relevant application obtained, in this case we will always get the first result generated by the search engine.
4. Obtain the implicit **package** of the application from the HTML code.

By this point, the package name is obtained so we can go on with the next step, retrieve the information of the application.

5. With the package name, we need to obtain the HTML of the detailed application website ([https://play.google.com/store/apps/details?id="+pkg_name+"&hl=en](https://play.google.com/store/apps/details?id=)), from where we will extract the most relevant information for the project.
6. Finally, we put together the basic information about the app and the reviews in a JSON format file, caching it for a short period of time in case the user triggers the same call again, and return the response to the user.

The output obtained from this sequence results in a JSON file, which schema is represented in figure 4.3.

```
{
  "name": "MyApp",
  "imgPath": "logo.png",
  "category": "Games",
  "avg_score": "4.3",
  "nReviews": "15",
  "package": "com.example.myapp",
  "reviews": [
    {
      "author": "John Martin",
      "authorImg": "profile-pic.png",
      "datePublished": "01-01-2017",
      "rating": "3",
      "title": "I love this app",
      "text": "This app is awesome, it's the best app I ever downloaded"
    },
    ...
  ]
}
```

Figure 4.3: Example of an output JSON file structure generated for *MyApp* test application.

4.2.3 Senpy module

This module manages the communication with Senpy [17] analysis platform, creating the request with the review and building the response by adding the analysis result. The communication between the server and Senpy is done using HTTP request, attaching the review inside the body. Depending on the algorithm used, some parameters could be required, for example the language of the text or an authentication token, in case we are using the *Meaning Cloud* [18] sentiment analyzer.

Once a sentiment or emotion analysis is triggered from the smart agent, the controller redirects the query to this module, where the task begins and the output is generated. In figure 4.4 we can observe this work-flow behaviour.

First of all, the procedure begins with the */analyze* API call method. The first step is to check if another similar analysis has been carried out recently, in that case it won't be necessary to execute again the analysis task, so the server just returns the cached output. Afterwards, the server needs the app info, more specifically the reviews that will be analyzed, to retrieve them by communicating with the Play Store module explained in section 4.2.2.

With all the reviews ready to be processed, the system starts a loop where each review is sent to Senpy, obtaining the analysis result. Depending on the analysis performed, the information obtained is different.

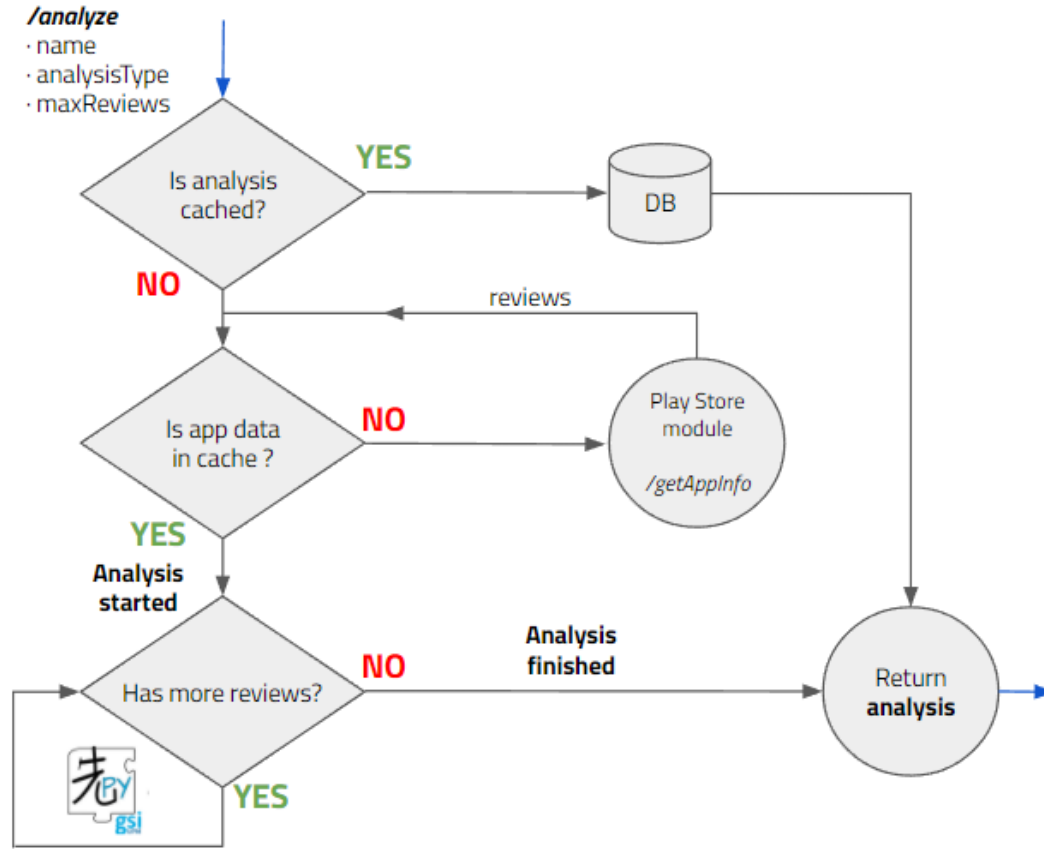


Figure 4.4: Sequence diagram with the analysis operation work-flow, since the user request to the final output with the application analysis.

Sentiment analysis response gives two relevant values, the polarity obtained after text classification and his equivalent category inside possible sentiment values, which could be *Positive*, *Neutral* or *Negative*. This explanation is represented in table 4.1.

Emotion analysis response is composed be three parameters, named *Arousal*, *Dominance* and *Valence*. The response also includes a static centroids that describe these parameter values for several emotion states, in order to calculate the nearest emotion for each review. Each of these centroids are represented in table 4.2, with it corresponding *arousal*, *dominance* and *valence* reference values.

After each evaluation, the application obtains the sentiment and emotion which globally represents the application user experience, based on the results for each individual review

result. This global emotion and sentiment is extracted applying the mean value inside all the values obtained from Senpy analyzer.




Emoji	Sentiment	Polarity
	Positive	1
	Neutral	0
	Negative	-1

Table 4.1: Sentiment polarity representation






Emoji	Emotion	Arousal	Dominance	Valence
	Anger	6.95	5.1	2.7
	Disgust	5.3	8.05	2.7
	Fear	6.5	3.6	3.2
	Joy	7.22	6.28	8.6
	Sadness	5.21	2.82	2.21

Table 4.2: Emotions centroids representation, with their values for *arousal*, *dominance* and *valence* parameters

To conclude, once Senpy task has finished the analysis process, the server returns the result in a JSON format file that contains the sentiment/emotion evaluation for each review and globally for the application requested. This result is cached in case it's requested again (figure 4.4)

4.2.4 Classification module

Inside this module relies two classification services that accept reviews related with mobile apps and return if it has been detected a bug/feature or not. In this section we are going to explain how these classifiers work and their training and testing process. Furthermore, we will present several classification methods, and the advantages of our classification system.

Both classifiers implemented in this thesis belongs to Naive Bayes Classifiers binary classifiers family, using supervised learning. Supervised machine learning algorithms can be used to classify the reviews. The idea is to first calculate a vector of properties, most frequently named as *features*, for each review. Then, a training phase begins, calculating the probability for each property to be observed in the reviews of a certain type. To sum up with the classification process, it's necessary to perform a test phase, where the classifier uses the previous observations to decide whether the review belongs to a type or not. All this process is known as binary classification. In this thesis we have applied two binary classifications: as a bug report or not and as a feature request or not. Alternatively, it is possible to assign the review to several classes at once, which is called multiclass classification [19].

Inside the huge family of binary classifiers, we are going to talk about several techniques that belongs to this category. **Naive Bayes** [20] is a very popular algorithm, which is based on applying Bayes' theorem with strong independence assumptions between the features. Giving a brief overview about the advantages obtained from it usage, it is simple, efficient and does not require a large training set like most other classifiers to obtain great accurate results. Decision Tree learning is another common classification algorithm [21], which assumes that all features have finite discrete domains and that there is a single target feature representing the classification, which is referred as *tree leaves*. At last, deeping inside multiclass classifiers, the multinomial logistic regression, or **Maximum Entropy**, is a popular one. This algorithm assumes a linear combination of the features instead of a statistical independence, and determines that some specific parameters related with reviews can be used to extract the probability of each particular review type.

To assure which of these techniques is better applied to our case, we have studied each algorithm performance regardless, examining multiple facts such as training duration, the optimal dataset size to obtain an accurate classification model etc. After this evaluation process we have concluded that:

- The dataset size affects directly to training time. Using about 200 reviews for training the classifier, the time curve seems more exponential than linear in shape terms. Model creation time is much higher for Maximum Entropy than Decission Tree or

Naive Bayes, being the last the fastest, taking between 10 - 20 minutes, depending on the computer specifications where the task is carried out.

- In terms of performance, two multiple binary classifiers, one for bugs and other for features, works significantly better than a single multiclass classifier in both cases.
- The **Naive Bayes** algorithm seems to be an appropriate classifier as it can achieve high accuracy with a small training set and less than half of the time needed by the other proposed classifiers.

Once we decided the classifier algorithm, it's time to begin the training process. The idea for this thesis is to generate a trained model and save it into a pickle, so the server controller can insert review inputs and extract the classification result without having to create, train and test the model each time, which will be really tedious. This enables to interact with the classification module fast and efficiently, being able to detect upcoming bugs from recent reviews, or filter those reviews for the developer-oriented site, where we can provide an automatic predefined answer to every type of comment and collect them to be solved as soon as possible by the app developer team.

4.2.4.1 Dataset

In this section we are going to talk about the dataset structure. This dataset was used to train and test both classifiers implemented in the project. As we introduced above, we decided to use supervised learning applied to **Naive Bayes** algorithm for binary classifiers. This set is composed by several tables with reviews previously tagged as bug or not_bug, and feature or not_feature. The data collection [2] about the feedback management was obtained from [19] paper, which offers a supervised dataset composed by multiple reviews posted inside the iOS app store. However, this data has been extrapolated, using them for Google Play marketplace context.

Going deeper inside the dataset, it's composed by 24 tables that globally contain almost 13.000 rows correctly classified inside different topics. These topics are bugs, features, rating and user experience. In this project we focused on bugs a features detection. In table 4.3 are represented our dataset sizes used to train and test both classifiers.

We used a 70% of a subset (370 entries for bugs topic and 295 entries for features topic) of each table to train the classifier, and another 30% subset to test it. This dataset contains lots of information inside table columns. For example, some of the most relevant columns are the *stopwords_removal*, which shows the text of the review only with meaningful

Dataset	Size	Train Size (%)	Test Size (%)
Bugs	379	256 (70%)	114 (30%)
Not Bugs	2752	256 (70%)	114 (30%)
Features	295	207 (70%)	88 (30%)
Not Features	1629	207 (70%)	88 (30%)

Table 4.3: Dataset sizes for supervised learning. There are represented the row count about four datasets inside *Re_2015_Set* [2] (*Bug_Report_Data*, *Not_Bug_Report_Data*, *Feature_Or_Improvement_Request_Data* and *Not_Feature_Or_Improvement_Request_Data*).

words. Other interesting columns are those referring to the tense applied inside the review, detecting how many times appear a present, past or future tense structure, in order to use grammatical structure for other analysis techniques. In section 4.2.4.2 we will analyze the accuracy obtained for each classifier after training and testing processes.

4.2.4.2 Performance

After completing train and test tasks for each classifier we are going to evaluate their performance obtaining their accuracy and the most relevant *features* for the models. The implementation of both models are based on the *textblob* [22] Naive Bayes classifier model, applying a basic feature extractor.

As we briefly presented in section before, we cache the trained model in order to access it easily inside a file extension named *pickle*. The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy [23].

With the pickle generated, the accuracy obtained for test set is presented in table 4.4 for each classifier type. We can explore which features are most used when we insert a review that clearly contains a bug or a feature. For example, bug positives usually contains negative sentences, being the ‘ character the most informative feature, followed by words like *open*, *fix*, *crash*, *last*, or *delete*. For features classifier, most common features are words like *crash*, *also*, *add*, *could* or *phone*.

Type	Accuracy	Recall	F-measure	Most Informative Features
Bugs	0.7	0.74	0.7	open, fix, crash, last, delete
Features	0.76	0.69	0.71	crash, also, add, could, phone

Table 4.4: Classification parameters for both classifiers, bugs and features, after working with test dataset. Some of these parameters are accuracy, recall, f-measure and finally the most informative features for each classifier ordered by descending in positive terms.

4.2.4.3 Workflow

To conclude this section we must explain how the classification process works inside the server, since the API call triggers the */classify* action, to the output obtained after invoking bug or feature binary classifier. As we explained in section 4.2.3, the application flows analogously like the analysis situation, as we exposed in figure 4.4. In this case, the reviews will be introduced in the classifier, depending on the type requested, obtaining a conclusion for each review, that will be contained $\{bug, not_bug\}$ and $\{feature, not_feature\}$ sets respectively. This sequence is represented in figure 4.5.

The classification result will be inserted inside the JSON which contains the app information, and returned to the user who claimed it. The classification tasks duration may vary depending on the amount of reviews the system has to process. In case it takes a long duration, it's possible that the server throws a timeout exception due to the missing response. In this case, the classification result will be saved inside a queue once the task has finished, being accessible from the smart agent.

Moreover, this module also affects to the developer-oriented functionalities implemented inside the Android smart agent. More concretely, depending on the classification result the developer can set up a specific response for the user comment inside the Play Store, and redirect that comment to other interesting team organization tool in order to solve the problem rapidly. So the workflow in this situation behaves quite different from when it's summoned through the API.AI instance.

The workflow in developers side begins with scheduler triggering, or by direct execution forced by the user. This will be better detailed in section 4.4, which is dedicated to fully explain the Android smart agent. Once the process starts, the next step will be the recent reviews extraction from the Google Play Developer API, more concisely using the Reply to Reviews API, presented in section 2.6. This API service enables us to retrieve upcoming

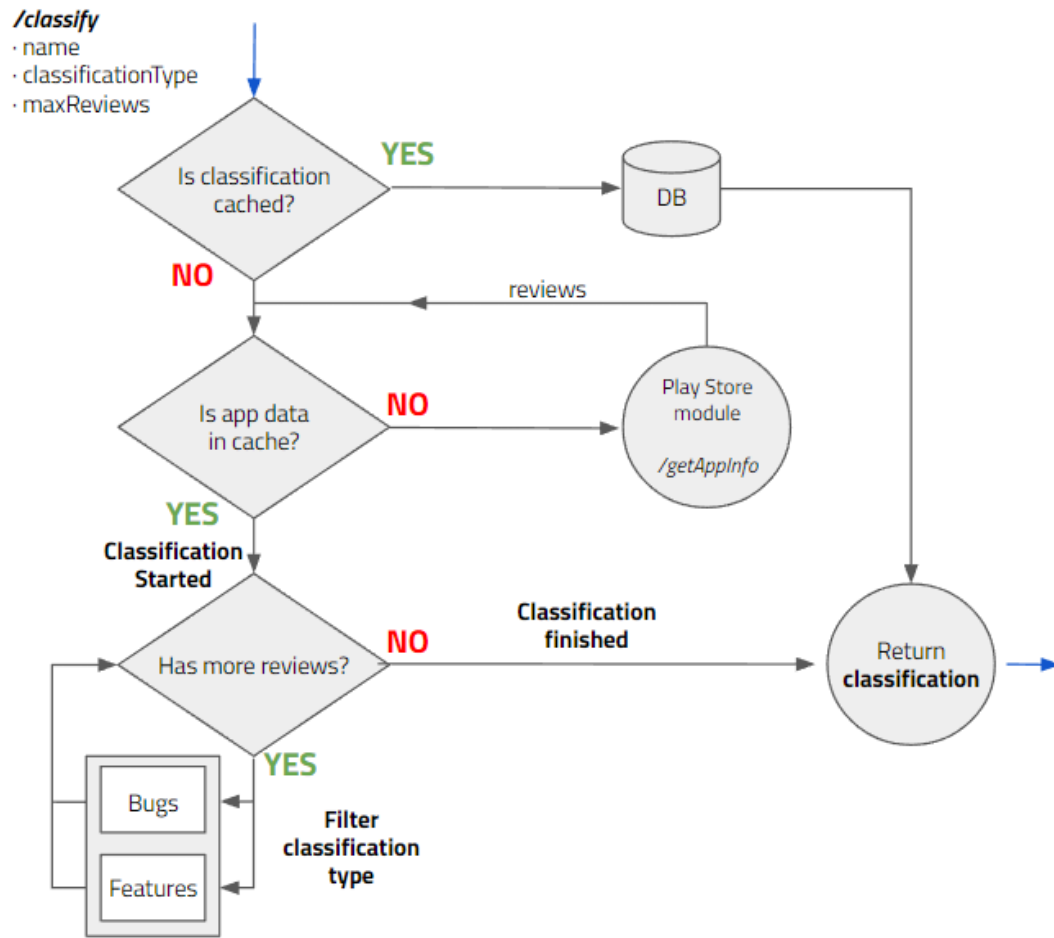


Figure 4.5: Sequence diagram with the classify operation work-flow, since the user request to the final output with the bug or feature binary classification.

reviews for applications published by the authenticated developer. When we have obtained all the reviews, it's necessary to classifying them in order to extract if they are related with bugs or new features proposals. After classify them, the agent replies automatically using predefined sentences designed by the user and posts positive results into developer's work dashboard, in order to collect those bugs and new improvements proposed by the users. All this sequence is represented in figure 4.6.

The process explained above allows to set up predefined user-friendly answers, so the developer doesn't have to worry about manually identifying review intentions, automating the feedback reception by packaging the valuable opinions and improve proposals effectively.

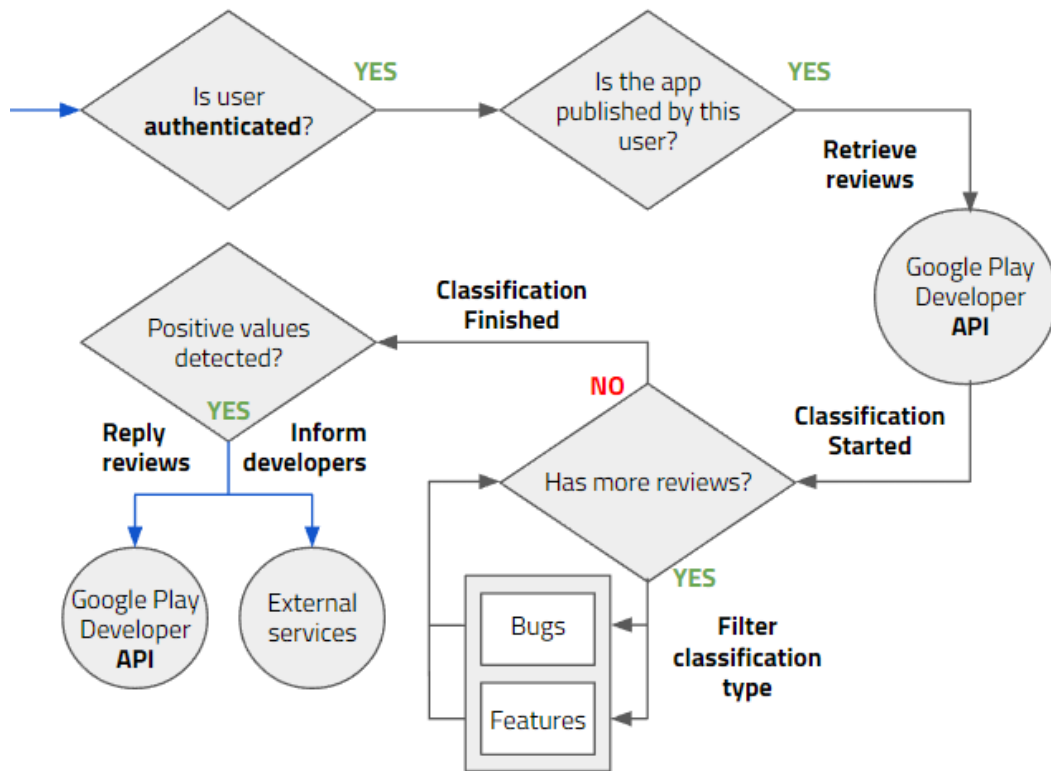


Figure 4.6: Sequence diagram with the classify operation work-flow for developers-oriented side.

4.3 API.AI Agent Module

In this section we will explain the API.AI agent architecture, detailing the entities, intents and modules created inside the framework and its connectivity within the server module and client agent. Every interaction with the agent is always made from the web interface provided by API.AI, being able to modify basic agent parameters or modify our fulfillment web server address in an easy way.

As we briefly presented in the Chapter 2, API.AI platform allows us to design an agent that will handle the request sent from our Android application, processing it and redirecting the valuable information extracted to the server controller. This avoid us to develop a natural language understanding layer, being able to define functions directly linked with server ones. These methods will be summoned through voice or text commands from the Android application, without use natural language techniques for meaning extraction.

First of all, we will explain the possible intents and entities created in the agent and shortly introduced in section 4.1.2. Afterwards, we will continue with the explanation of

the sequence followed when a new event arrives to the agent, enumerating the different phases involved until it becomes a JSON file. This process implies the explanation of those procedures that extract incomplete parameters from the conversational context, or carry out an entity recognition process to determine if exists an intent that applies for those input parameters. To sum up, we will expose some Machine Learning techniques stealthily performed inside the agent, and managed by the API.AI framework automatically.

4.3.1 Elements

Inside this section we will introduce what components take part in the conversational process with the agent, detailing their functions and structure in the system. We divided the elements into four groups, **entities** that act as the model of the agent and describe those objects that can perform multiple action depending on its value, **intents** that represent the action that the agent is able to carry out, **conversational context** which main function is to complete missing parameters when an intent triggers, being able to extract them from the previous messages inside the conversation. At last we have the **responses** that represent the way the agent answer our requests, defining a clever structure with all the necessary data available packaged in a JSON format.

The communication between intents is made through **dialogs**. This architecture enables to "jump" from an intent to another depending on the user response. For example, if the agent asks for task confirmation, depending on what user answers, where the possible answers are *Yes*, *No*, or whatever different than that, the conversation will flow in a different way, calling different intents until the action finishes, resetting the conversation contexts.

4.3.1.1 Entities

The entities that actively participate in the agent events and actions are the **Application**, which represents an object that defines the mobile app itself. This entity could have associated infinite possible values, because the Play Store market place has such a high amount of applications published. For this reason, we have declared some examples inside the entity configuration form, so the entity recognition module detects those easily. Some examples are *WhatsApp*, *Telegram*, *Instagram...* or any application placed inside the most popular apps ranking. It is also important to provide examples with a name composed by multiple words, so the system gets familiarized with them, preventing being truncated when requested. Finally, it is important to declare some weird app names which contain uncommon characters or symbols hard to interpret when the agent uses voice commands,

such as numbers, abbreviations, acronyms or punctuation marks.

Furthermore, we have the **AnalysisType** entity whose values are limited to *sentiments* or *emotions*. It's recommendable to add some synonyms to these parameters, so in case the user uses the singular form, the entity recognition module could identify the value correctly. Analogously, the same situation occurs with the **ClassificationType** entity, which is composed by *bugs* and *features* value. We also added their singular forms in case the user input vary.

4.3.1.2 Intents

Inside the API.AI agent exists multiple intents, some of them carry out a real functionality while others just act as intermediates between certain user behaviours. The intent relation could be represented as a tree, where there is a main leaf from where other path begins, building the complete architecture of the system and covering each possibility inside the conversation process. We will explain two main important intents, the Analyze and the Classify intent, both belong to the top layer of all intents defined.

4.3.1.3 Conversational Context

Thanks to conversational context we can pass information between intents, such as the application name that the user wants to analyze or what type of classifier wants to perform. Contexts link intents each other during a limited period of time named *lifespan*. This property configures how much time must live a contextual information inside the dialog sequence. To represent how contexts work inside our agent, we are going to briefly present the dialog architecture when user requires a sentiment analysis for Whatsapp application.

The process begins with a level 0 intent triggering (*Analyze*), which means that this intent has no dependencies attached, so it's the first node of the tree. The user input could be *Could you perform a **sentiments** analysis for **WhatsApp** application, please?*. You can see in bold both relevant entities, the analysis type and the application name. In case one of them is missing, the bot will prompt a default sentence asking for its value. After the user input insertion, the agent responds, *I'm about to run a **sentiments** analysis for **WhatsApp** application, are you sure?*. By this point, the decision tree is divided into three possible choices. If user answers *Yes*, the analysis will go on, starting the connection with the server controller. In case user answers *No*, the agent expects some modification in the parameters, or the task cancellation, being necessary to set up from the beginning the user request, *Better perform an emotion analysis for Telegram*. Now the entities value

has changed, and the agent is ready to start the analysis requested. In case user answers whatever different, the agent waits until it matches the *Yes* or *No* paths.

Within the sequence explained lots of intents interact to handle all the user answer possibilities. The example decision tree is represented in figure 4.7.

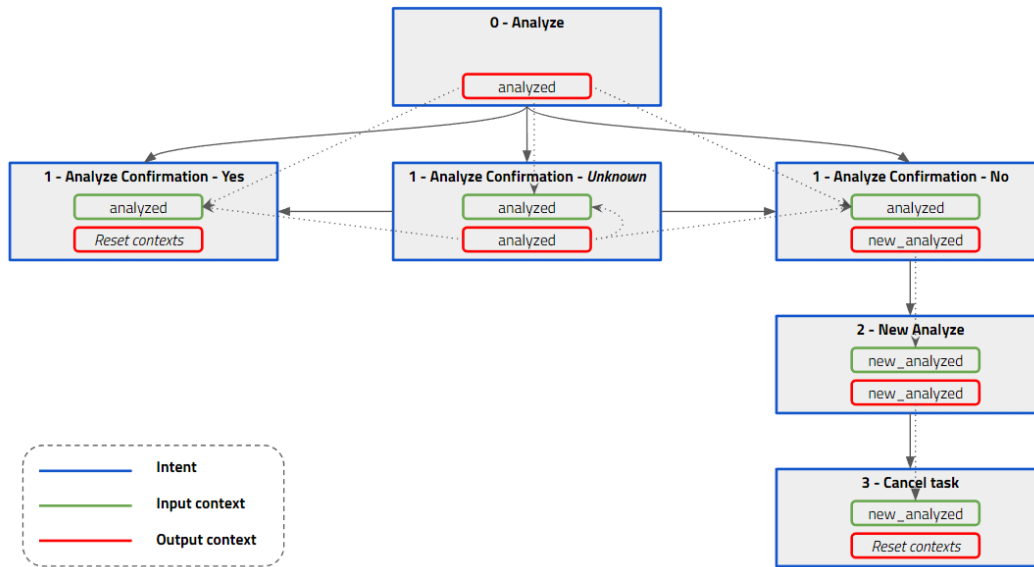


Figure 4.7: Example of intent communication using contexts inside the API.AI agent.

The classify intent acts in the same way as we exposed in figure 4.7, having their own intent to manage confirmation dialog. This is just an example of the flexibility offered by the platform, being able to set up conversational environments which interacts with the user as a human being.

4.3.1.4 Responses

To sum up we will talk about the response that API.AI agent generates, and which one is interesting for our project. The response generated is in a JSON format and contains firstly the basic information about the agent, id, timestamp and language. After this, we find the result object, that has the most important part of the response. It begins with information about the action triggered, its name, the query executed and the parameters. Then, the context is attached, detailing all the input and output contexts of the message. After some meta-data section, we reach the fulfillment section where is the agent response message and, in case there is a web-hook enabled we can find the response given by the endpoint. The last block specifies the HTTP status code, in case the request timeout or similar errors happen.

4.3.2 Workflow

After explaining the elements that compose the agent architecture and how they communicate each other, it's time to detail how the agent behaves when a new request arrives, and the decision that has to be taken until we obtain the response, or in some cases the web-hook server response. This procedure is represented in figure 4.8. First of all the agent carries out an intent recognition task, so it can identify if the query triggers any of the ones implemented in the agent. If successful, it should apply a parameter recognition based on the agent entities to extract meaningful values, such as an application name or the analysis or classification type. In case any parameter is missing, it's necessary to obtain them from the context or prompt a request dialog in case it's mandatory the slot filling. The last step is generating the response connecting with the endpoint, or by default answering with the predefined text responses.

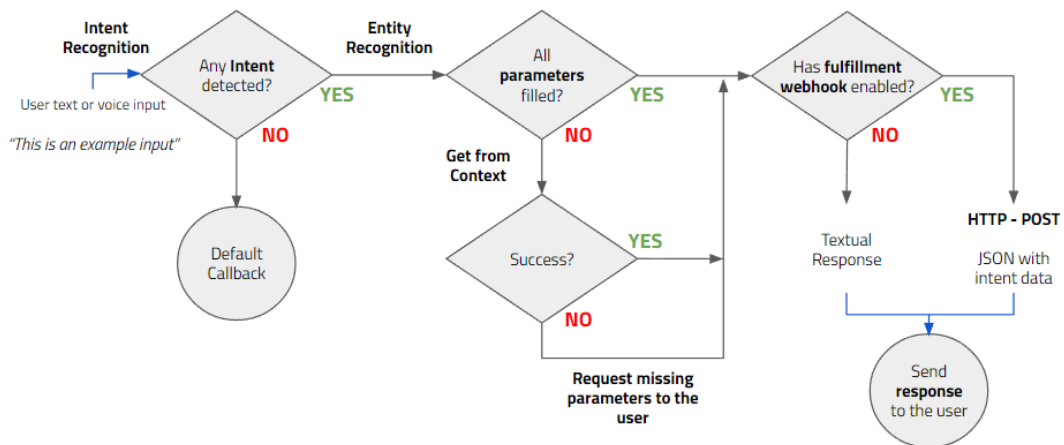


Figure 4.8: Sequence diagram with API.AI agent behaviour when receives a text or voice input from the user.

4.3.3 Machine Learning techniques

To achieve good classification accuracy, it's important to provide our agent with enough data. The greater is the number of natural language examples in the *User says* section of intents, the better is the classification accuracy. It's recommendable to use a huge variety of grammatical structures so the agent can identify user intention with no doubt.

When you create a new intent, start with examples with the maximum number parameters. This way you will define what entities should be used in this intent and name all the parameters the right way. Having annotated the first few long examples, it will be easier for

you to continue with shorter ones, as the system will start suggesting you the right entities for new examples. It also helps machine learning models train better.

To make the training process more efficient, API.AI platform provides us a *Training* tool that allows us to analyze conversation logs with the agent and add annotated examples to relevant intents in bulk. Through this module we can improve pattern recognition functionalities and validate or refuse wrong approaches. All intents developed for our agent have this plugin enabled, so the much agent receives user inputs, the better the agent analyzes our queries.

4.4 Client Module - Android Smart Agent

In this section we will explain how the smart agent has been developed. As we presented in the architecture overview, we have chosen the Android framework for its development. Inside the mobile application we can differentiate several modules that play their own functionalities inside the project objectives. We will begin with a brief overview about the client module architecture, explaining the components represented in figure 4.1 and zoom in some acts in background. Afterwards we will talk about some key concepts necessary to understand how Android applications are composed, and finally explain the architecture of regardless modules.

4.4.1 Overview

The smart agent developed for Android devices is responsible for managing user behaviour and connect with the API.AI agent, acting as an intermediate between the natural language insertion, passing through the API.AI agent and finally consume any service provided by the server. In figure is a zoom in of the global architecture of figure 4.1.

The module is composed by three sub-modules, each one responsible for carrying out a task to achieve the thesis objectives. These modules are independent each other, and interact with the server and the API.AI in different ways. The application is mainly composed by:

Chat agent module: This module shows a chat interface whose input directly connects with the API.AI framework. It offers a simple and friendly user interface where the allowed input methods are text or voice. The chat appearance gives the agent lot of personality, becoming an assistant where the user can request Play Store related tasks,

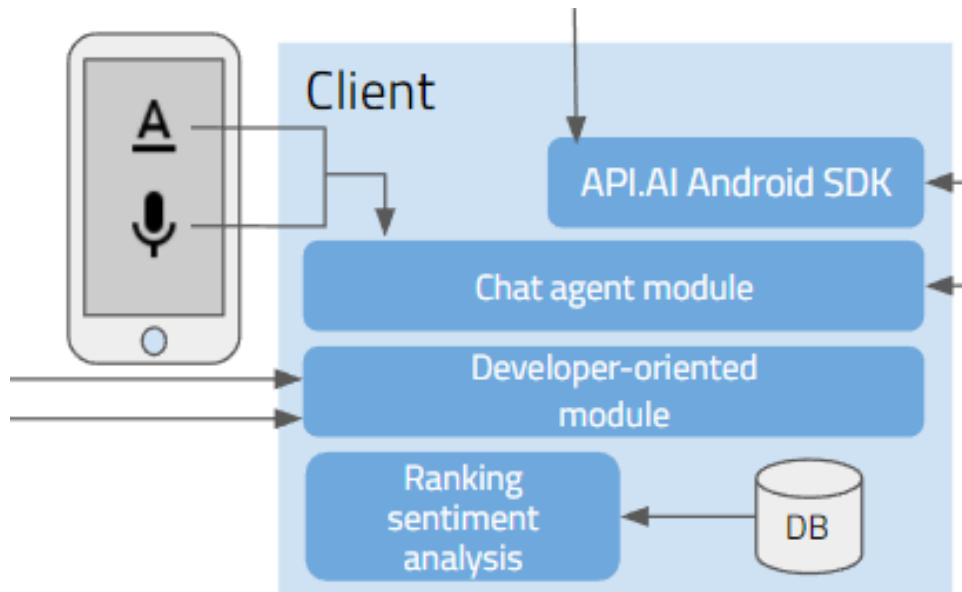


Figure 4.9: Zoom in about client module inside the global architecture of the project.

like analysis or classification, and examine the app market in real time extracting the more recent feedback posted in the market. Describing the module from a background perspective, it's basically an API.AI Android SDK implementation, using a service where the application user post his queries, and the response obtained by the remote agent is returned, being quite easy to define short conversations between the user and the bot just using natural language instead of send commands. Going deeper in the information sent and received, the analysis and classifications are shown to the user through graphs and icons that represents the sentiment or emotion results, even the evolution over time of the requested application.

Developer oriented module offers an interface from review response automation. The idea is to identify potential bugs or improvements reported by users, and response them according to the result obtained, redirecting the feedback obtained to third platforms, from where developers could stack most common problems and debug the app as fast as possible. This module is directly connected with the classification module hosted in the server, without passing through the API.AI agent. It also interacts with the Google authentication service, and the *Reply to Reviews* API, that belongs to Google Play Developers service. At last, it also communicates with some project management oriented third-party applications, like *Trello* or *Slack*.

Ranking sentiment analysis presents a static sentiment analysis made over the most downloaded free applications placed on the Play Store ranking. The main objective of

this module is to provide an external overview about the feedback obtained for such apps that succeed in the marketplace, and evaluate the sentiment evolution over time for these most common applications, to later extrapolate those results to self developed applications results in sentiment analysis environment. This section represents an static analysis so connectivity with other modules is not needed, representing the data hosted inside a local database within the Android agent.

Other resources such as the API.AI connection library, physical resources like microphone for voice detection or simply the database where the analysis is stored also contribute in the Android application lifecycle.

According to the Android basis presented in chapter 2, we will go deeper inside each module presented before. In order to simplify sub-modules architecture and facilitates their understanding, it's recommendable to check the referenced section. Finally, using all these components explained in section 2.2.1 we feel confident to go on explaining the practical implementation of them into our smart agent, detailing its implication inside each module.

4.4.2 General application

From a global perspective of the mobile application, we will focus on explaining the internal architecture and how each sub-module is accessed. Every component follows the material design guidelines [24] established by Google in order to follow the standard in design terms. The application is composed by a single main activity with a *NavigationDrawer* component inserted. This component allows users to divide the screen in multiple sections called *fragments*, by accessing them tapping the menu layer button placed in the action toolbar. The *NavigationDrawer* component handles the access to the rest of modules pressing each tab. These tabs are regardless each other and manage their own lifecycle without other modules dependency.

The whole application is based in the **Model - View - Controller** (*MVC*) pattern, adapted to the Android base classes. The model of the application is shown in Fig. 4.10 using *UML* notation. Main classes basically are *Message*, *Analysis*, *Classification* and *Review* for the chat module, and the *Response* class for the developer oriented one.

As is described in the diagram, Analysis and Classifications have a 0...* relation, but normally it will be the default value, just because it feels really weird to include the number of reviews inside the query using natural language. That's why the user usually will request the analysis, but doesn't mind the number of reviews, most of the cases because he doesn't know that the *maxReviews* actually exists.

The controller normally instantiates these model classes to represent the requests and responses involved in API.AI communication. The views inside the application are written in XML format. This allows users to define several layouts which contain the components of the app, being really simple the user interface design. All the graphic resources such as images or icons are saved inside the drawables directory, and are referenced for the layout XML files.

Once the basic design guidelines are explained for the application development, it's time to examine each module behaviour.

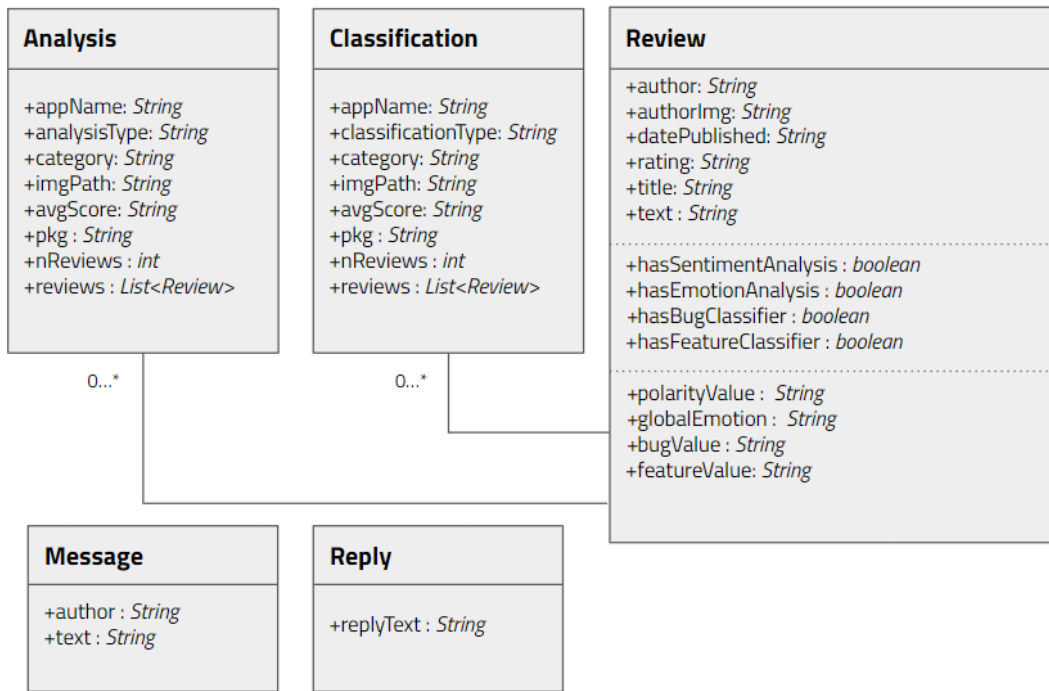


Figure 4.10: The figure represents an UML diagram modelling with those classes that actively participate in the application.

4.4.3 Chat agent module

In this chapter we will focus on detailing how the chat agent module works. As we explained in subsection 4.4.2, this part has been implemented over a *fragment* accessed using a *NavigationDrawer*. The main component of the module is a *RecyclerView*, which basically is a list that accepts objects of different class types. This way, the list has attached an adapter that manages the content that want to be shown with the view elements. Below the fragment there is an *EditText* box, from where the user can interact with the agent by text or voice sentences. Every time a message is sent or received, a new item will be

added to the *RecyclerView* list, and in case an analysis or classification result is obtained, a *CardView* object will be added. By clicking the card we can access a more detailed information about the analysis or classification result calling a new Activity for it, named *DetailedAnalysisActivity* or *DetailedClassificationActivity* respectively.

Inside the *onCreateView* fragment method, the API.AI manager is instantiated creating an *AIManager* object. This class manages every communication made with the API.AI Android SDK, which means service binding and the *SendRequest* async task so the requests can be sent to the remote agent. When the user sends a new query, the *AIManager* service needs two parameters, the query itself and the receiver object that will process the API.AI response. This receiver will obtain the JSON generated by the remote agent, processing it converting the data into the corresponding model. Depending on the response, the chat message list will add an item or another, completing the conversational process. In figure 4.11 is shown all the sequence explained before.

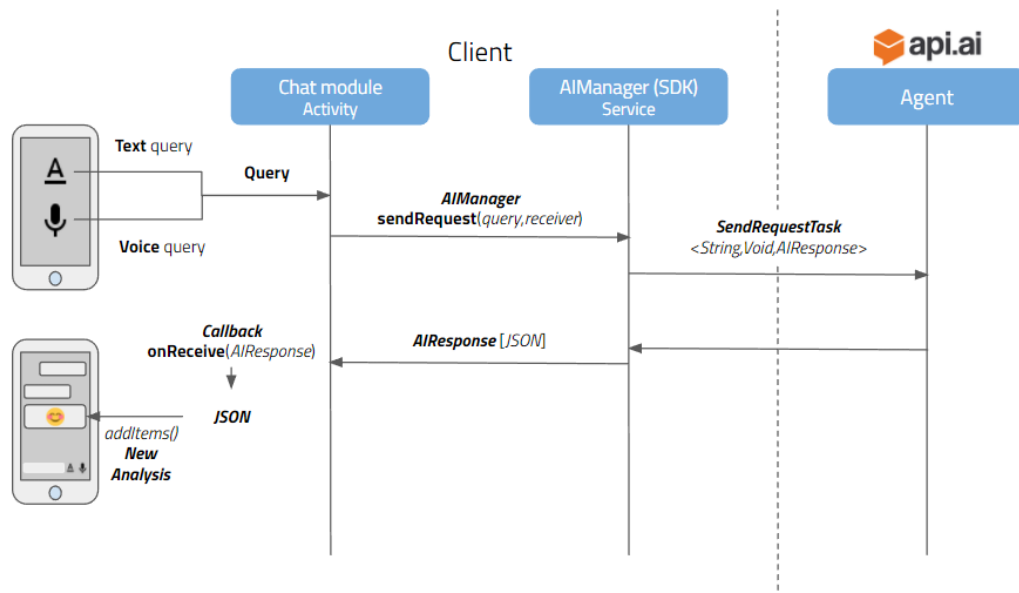


Figure 4.11: The figure represents the sequence followed by the mobile agent when a new query is made by the user. The process starts when the input trigger, calling the *AIManager* service and finally obtaining the JSON response from API.AI agent, which is translated into a new list element.

By default, the number of reviews allowed for both, analysis and classification are set to 15 in order to minimize the server response time. However it can be modified in query format or either by editing the default parameter value in the API.AI agent. In case the response takes too much work, the API.AI agent may throw a timeout exception, in which case the response will contain the task identifier. The mobile will retrieve the response

directly from the server by consulting for the specified task inside the queue. This could seem a weird approach, but API.AI was initially design for short time responses, and even it allows web-hook connections, those can't overload the communication flow, so the platform designers set 10 seconds as the maximum limit time for processing web-hook requests.

The chat activity has also implemented a message buffering system, that enables to archive last queries made by the user, or check recent analysis or classifications result. The cache system works in background and automatically, allowing a maximum of a hundred messages buffer size allowed. The database is hosted in the Android *SystemPreferences* module, which offers a String storage system for future sessions based on object serialization, in this case the *Review*, *Analysis* and *Classification* models must extend the *Serializable* class, and also the sent and received *String* messages from the remote agent.

4.4.4 Developer-oriented module

This module enables to automate the reply process for the feedback obtained from self-developed apps. This intends to speed up the review answering process, depending on the intention detected in the rating. For this thesis, we will focus on automate responses depending on the result obtained after performing a bug classifier. This way, the reply text will be in the same context as the review.

To carry out this functionality it's necessary to communicate with the Google Play Developers API, and also being authenticated inside it, because this is only available for those users who have purchased the Android developer license. For this reason, the user will only be able to interact with self-developed applications feeds.

As we presented in section 2.6, the API calls are restricted, being able to recover only those reviews published in less than 15 days from the request time. This complicates the simulation process because small applications developed by the *GSI* research group or either by myself doesn't have much reviews to analyze, and most of them will be in Spanish so the classification process won't work. For this reason we developed the interface and oriented it to applications with high audience, at least around 30 reviews in 15 days.

The architecture is pretty simple. The smart agent directly communicates with the Google Play Developers API. First of all, authentication is required. Some additional steps must be accomplished before log into the developer oriented section. The developer account manager must grant permissions to our application. For this reason, it's necessary to create an *OAuth* [25] client id so the application can retrieve a valid authentication token.

Once the authentication token is obtained, it's necessary to obtain a *refresh_token*, in

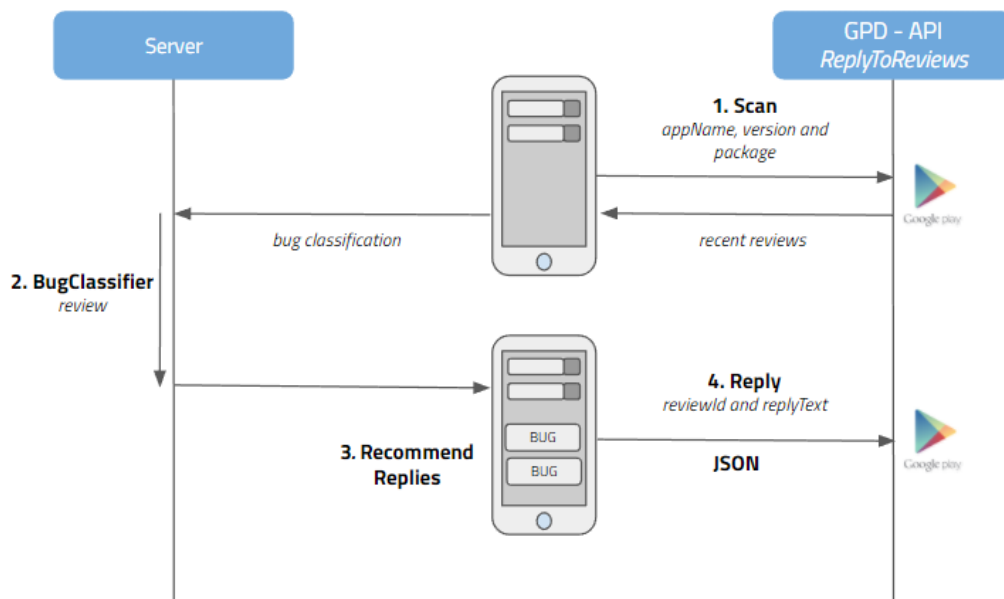


Figure 4.12: Sequence diagram of the answering process. First the user scans a self-developed application and detects possible bugs. After he posts the recommended replies.

case the *access_token* expires, so the app can renew it. Once the *access_token* is generated, the rest of the communications are made with HTTP POST requests. The input parameters are the app name, followed by the version that is currently published in the store, and the package. With all these fields, we can scan the API looking for valid reviews made within a short period of time. If any review is returned, automatically the app contacts with the bug classifier hosted in the server and tag it. A recommended reply is suggested for each review classified as bug. These recommendations come from a static list of user friendly answers in case a bug is reported, so most of them apologizes and asks for patience.

Once the recommended reviews are displayed, the user can ignore them or reply with those answers. For reply process it's mandatory the *access_token*, the review id that is going to be replied and the response text. It's sent through a JSON file attached inside a HTTP POST request.

This complete process is represented in diagram 4.12, where an example application is scanned, classified and finally answered depending on the result.

4.4.5 Ranking sentiment analysis module

This part of the application consist on a sentiment analysis about the most downloaded free applications. The dataset is formed by a total of 60 applications, with a maximum limit of 100 reviews per app. The main value of this client module is to analyze the sentiment evolution over time inside the most demanded applications in the market, so a freelance developer could compare the feedback obtained from their self-developed apps, with those which are considered as trending products.

This means that *a priori*, the information displayed in this section stays invariable, so it remains static, but it could be updated monthly so the developers can adapt their software according to those functionalities that seems to be preferable to most of users.

This static analysis can also help to freelance or recently created *start-ups* as an inspirational tool, being able to detect, with the feedback obtained from the most valuable apps, well innovative opportunities to develop similar applications.

The data extraction has been carried out using a *Python* [4] script. It's based on the *PlayStoreModule* server module, mixed with the sentiment analysis part implemented inside the *SenpyModule*. The dataset structure is represented in figure below.

```
[
  {
    app_name:"",
    package:"",
    category:"",
    imgPath:"",
    nReviews:"",
    analysisType: "sentiments",
    globalPolarityValue:0,
    avg_score:"",
    nReviews:"",
    reviews:[
      {
        sentimentAnalysis:{
          marl:hasPolarity:"",
          marl:polarityValue:""
        },
        rating:"",
        title:"",
        author:"",
        text:"",
        datePublished:"",
        authorImg:""
```

```
        }  
      ]  
    },  
    ...  
  ]
```

With this dataset, it results pretty easy to design an interactive user interface where the user can search those apps that think are relevant. The app interface will be shown inside the *Case study* chapter.

Case study

In this section we will present a complete usage scenario for the thesis. It is going to be explained the running of all the tools involved and its purpose. It is based on how a freelance developer or a newly created start-up could find real value from the smart agent developed. Afterwards we will show how the mobile application looks inside the case study.

5.1 Scenario overview

We propose a scenario starred by a new *start-up* created a few weeks ago whose business activity focuses on developing mobile applications for third companies, or by their own initiative. This new company is trying to explore the mobile application market, more concisely the Android for the moment. For this reason, they decided to use the smart agent designed in this thesis so they can study using sentiment analysis techniques how users thinks about their own applications, and similar apps that directly compete with their products having the potential users.

Due to the low number of employees that actively participate in the *start-up*, it's desirable to process the feedback obtained from their users and extract valuable data from it, such as bugs and crashes encountered, or features and improvements that could increase their audience opinion. Moreover, it will be great to redirect these results to a team management platforms like *Trello* or *Slack*, so the developer team could stack those reviews and solve it as soon as they can.

Actor identifier	Role	Description
ACT-1	CIO	<i>Chief Information Officer</i> , in charge of tasks such as perform sentiments and emotions analysis of market apps, analyze the market sentiment evolution for most popular apps and try to apply those well rated functionalities to the enterprise.
ACT-2	Developer	Representative of mobile programmers department, that focus on detecting bugs and features from review classification results, solving the bugs as soon as possible and trying to implement proposed improvements in future app versions.

Table 5.1: Scenario actors list

Finally, as an optional functionality, it will be great to explore how users feel about most popular apps, so this enterprise can reference those apps that show a good reception in sentiment terms, in order to scale their self-developed apps within the popularity rankings, opting to reach a better number of downloads per published app, increasing its value as a company.

After exposing the case study, most relevant actors are shown in table 5.1. The Chief Information Officer (CIO) is responsible for app market study, using the smart agent to extract conclusions about sentiment and emotions analysis. He also must authenticate the agent with the enterprise Google developer account, so the ACT-2 can set up the developer-oriented module and reply the reviews.

About where each system module runs, the execution environments proposed for each project component are detailed in table 5.2.

Component	Where it runs?
Mobile application (Smart Agent)	Any Android device (over API 19).
API.AI Agent	API.AI Website, using enterprise account.
Server controller	Laboratory, GSI, <i>cluster</i> with <i>docker</i> containers.

Table 5.2: Execution environments

Afterwards, we will present some scenarios of usage where the start-up presented before uses the smart agent developed in this thesis, attaching some screen captures with the result obtained inside the Android app.

5.1.1 Market Explorer

The *start-up* is looking for a project management application in order to improve their inter-department communications and track their external project status from their smartphones. Therefore, the CIO has decided to explore the market looking for applications that are able to solve this upcoming idea. For this, he will perform a sentiment and emotion analysis for popular team management apps, more concretely he will focus on *Trello*, *Slack*, *Basecamp*, *Todoist* and *Evernote* apps. All these applications are used to coordinate development groups, some of them aren't free at all, like *Basecamp*, that offers a 30 day trial period. The main goal is to observe the sentiment and emotion on these apps and choose one to

incorporate it within the activity of the company. To carry out this procedure, he will use our chat smart agent. In concrete, the CIO will perform a sentiment and emotion analysis for each published app and later compare them depending on the results.

Using the chat module, the user can insert the queries through text or voice input format. An example query viewed from the agent perspective is shown in figure 5.3.



Figure 5.1: Voice input

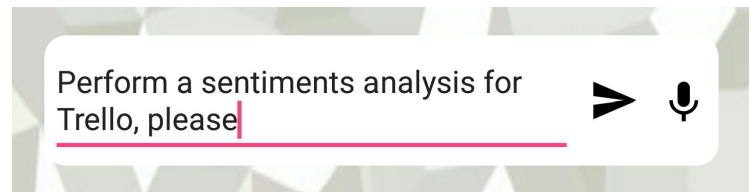


Figure 5.2: Text input

Figure 5.3: Example query made from the smart agent chat module. The user requests a sentiment analysis for the application named *Trello*.

In figure 5.6, is represented an example of the conversation the CIO has inside the proposed scenario. The agent accepts multiple input format for queries, so it could exist multiple possible dialog flows to obtain the same analysis output. The user interface given by the agent it's really simple, acting as a task assistant answering any user input thanks to the API.AI platform. After the analysis process, final results are grouped in table 5.3. The application enables to examine more in detail the results obtained by clicking the card, being able to extract even the sentiment or emotion of a single review.

To conclude, observing the results obtained from both analysis the CIO has to determine which third-party platform is better considered by the Play Store user community. Most of the applications have present the same emotion, so generally are good applications. In the other hand, the sentiment vary depending on the app, probably due to a recent updates that users don't like at all.

5.1.2 Feature mining

Lastly, our *start-up* received a project plan offer from a well-recognized digital newspaper, which requests a second version of their Android mobile app named *The Guardian*. Due to the low budget the company had to face the project in the first version, they decided to implement only basic functionalities to reach a higher audience developing a simple application for smart-phones. Now, the company wants to order a second version with advanced

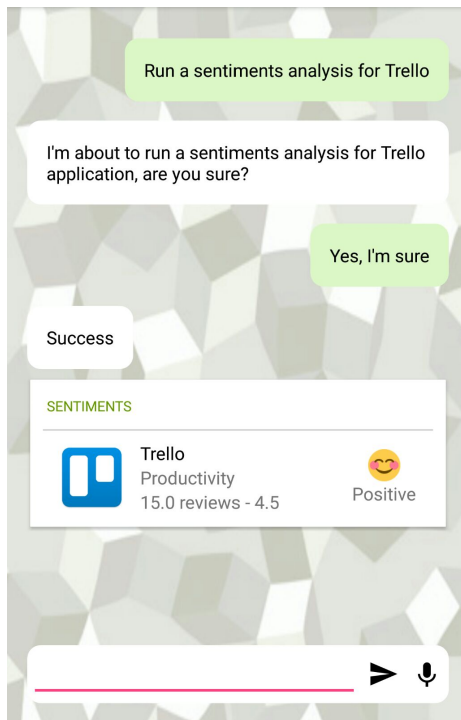


Figure 5.4: Sentiment analysis for *Trello*

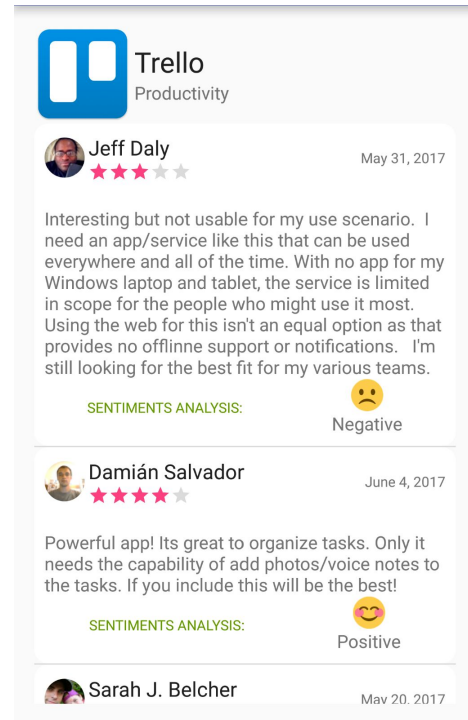


Figure 5.5: Detailed sentiment analysis view

Figure 5.6: Dialog interface between user and the API.AI agent. By tapping the card (1) with global results, the *Trello* analysis details will be shown (2).

Application	Reviews	Sentiment	Emotion
Trello	15	😊	😊
Slack	15	😞	😊
Basecamp	15	😐	😊
Todoist	15	😊	😊
Evernote	15	😞	😊

Table 5.3: Results of sentiments and emotions analysis in project management applications.

functionalities to our scenario company, and the *start-up* developer team members aren't able to identify interesting improvements or features for the application. For this reason, they decided to *obtain* them from the current published version, in order to implement those mostly demanded in the Play Store reviews section.

To carry out this operation, they will use the smart agent proposed in this thesis, performing a feature classification over the *The Guardian* application, so they can detect new features and improvements, so they don't have to read every single comment and extract if it contains an interesting value, earning lots of time. So the representative of the developer team member (ACT-2 in table 5.1) decided to run a feature classification for the certain app. The available inputs allowed, text or voice, are represented in figure 5.9.

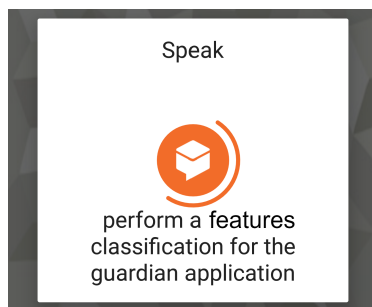


Figure 5.7: Voice input

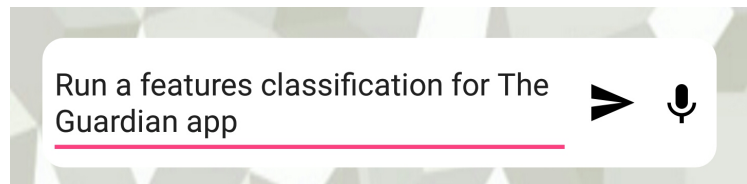


Figure 5.8: Text input

Figure 5.9: Example query made from the smart agent chat module. The user requests a features classification for the application named *The Guardian* so they can extract improvements from the user feedback.

The feature analysis is executed for a total of 30 reviews, considering that it's enough to extract a significant amount of relevant features. The conversational dialog with the developer representative and the API.AI agent as participants is represented in figure 5.12

Some of the reviews tagged as features can be checked in table 5.4. It shows the classification result and the value obtained from a natural language comprehension, being able to detect if the classification output matches with a human understands after read that user review.

As we can appreciate, most of the improvements extracted match with the classification result. After this process, the developer team meet together and brainstorm about the most common features demanded by the users, prioritizing those that seems to be more interesting for the client. Without usage, the developer team would have had to fetch every single review, filter those that refer to bugs, user experience, opinion etc, and finally evaluate their importance in the second version.

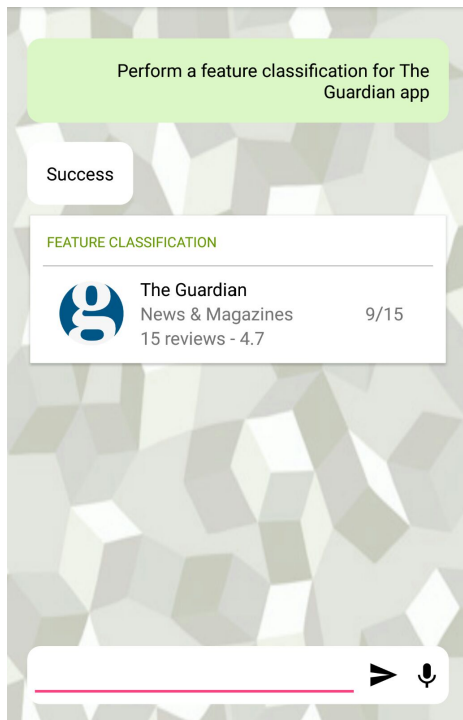


Figure 5.10: Features classification dialog for *The Guardian* app

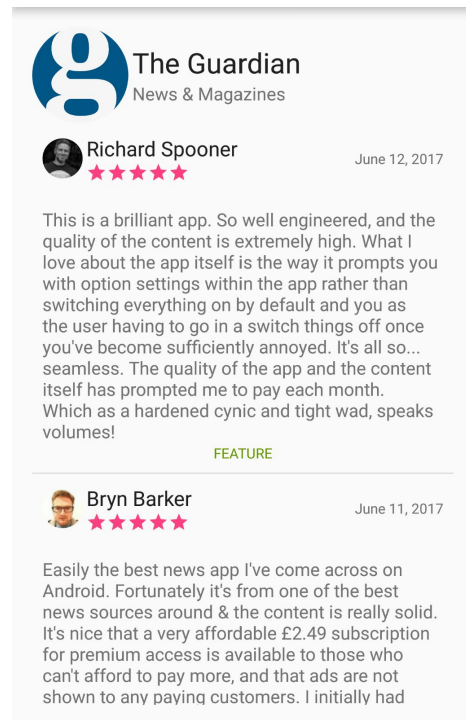


Figure 5.11: Features classification details

Figure 5.12: Dialog interface between user and the API.AI agent. By taping the card (1) with global results, the *The Guardian* classification details will be shown (2), showing those reviews tagged as features.

Review	Classification Result	Value
<i>...by default, it does tend to send quite lot of notifications...</i>	Feature	Customize notification subscription system
<i>... however, your news source doesn't update like CNN, you stay on the same old news headline...</i>	Feature	Update more frequently the news title, in case an important new come up suddenly.
<i>Very good, lots of interesting articles, good podcasts, easy to read posts and videos...</i>	Not Feature	None

Table 5.4: Results

5.1.3 Trending apps

Our project manager came up with a new idea a few weeks ago. He wanted to develop a game for Android devices based on other games that have already success in the community. He was indecisive about the game theme, considering arcade and puzzles games as the best option to succeed in his opinion. The rest of the team members didn't agree at all so they decided to check what type of games are nowadays most popular inside the Android Play Store market. For that, they used the ranking analysis offered by the smart agent, being able to filter by app category and see the sentiment and the average rating score. They definitely decided to filter by the *arcade* query, so they can observe if there are popular games of that category inside the top 60 free applications.

The process is represented in figure 5.15, where it can be observed the results extracted after filter by *arcade* query. All the results obtained from the filtering are shown in table 5.5

This analysis enables to extract those ideas that are currently trending, which offers to small companies to develop products with a high chance to success inside the market, scaling inside the position ranking faster than an idea that hasn't been studied within viability terms.






Position	App	Average Score	Sentiment
6	<i>Fidget Spinner</i>	4.1	
32	<i>Subway Surfers</i>	4.5	
46	<i>Tigerball</i>	4.6	
57	<i>Piano Tiles 2</i>	4.7	
60	<i>Dancing Line</i>	4.8	

Table 5.5: Results obtained from query the ranking sentiments analysis

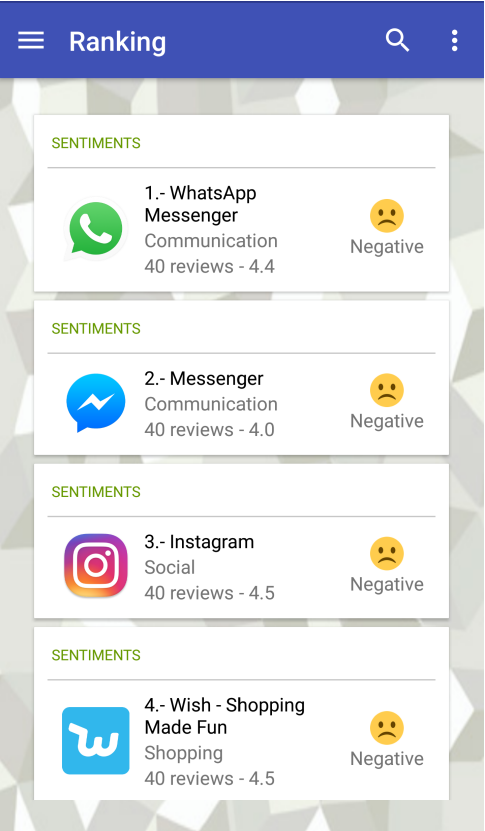


Figure 5.13: Ranking list of top 60 free applications

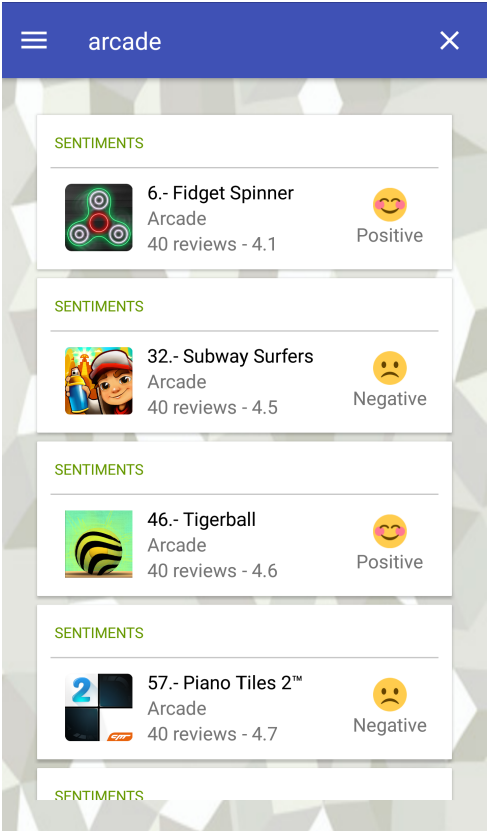


Figure 5.14: Query result from search *arcade* category

Figure 5.15: Filter process inside the ranking static sentiment analysis placed in the smart agent.

If we analyze the results obtained after executing the proposed scenario, it seems that exist almost a 10% of apps that belong to the *arcade* category inside the top 60 free applications. The Fidget Spinner it's placed at sixth position and has recently obtained positive ratings, with a 4.1 of average score.

To conclude, just observing the table 5.5, the developer team has a reference about the sentiment of most popular apps inside *arcade* category, so they can now decide if it's viable to go on with the initial idea instead of changing it to obtain better acceptance in the market. Remember that the results shown in the ranking section are illustrative, because the sentiment was obtained just from the last 40 reviews published, so it could vary in case there's a new update or features version for the game.

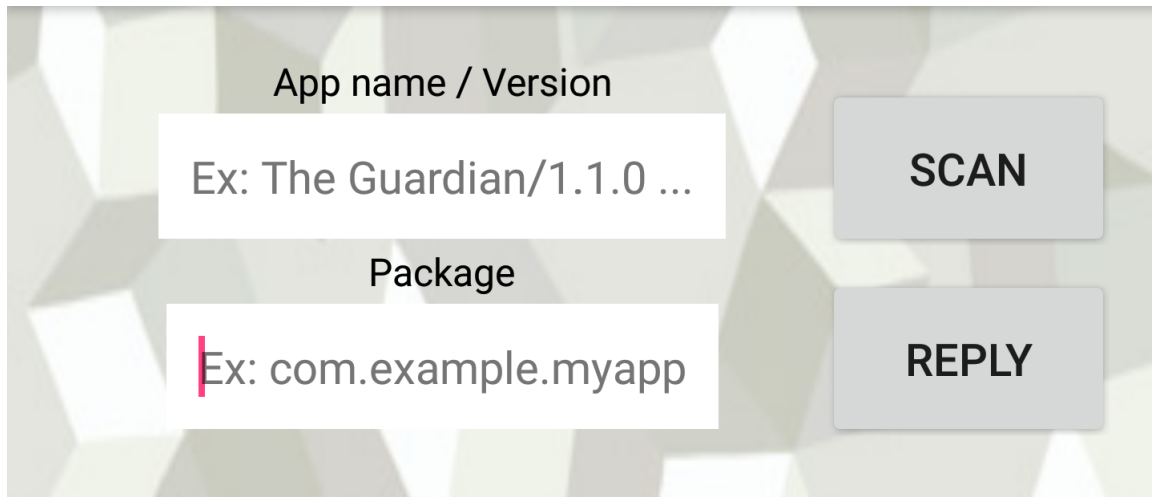
5.1.4 Automated reply for bugs detection

As we presented above for the *Feature mining* scenario in subsection 5.1.2, there much valuable information inside a Play Store application. In this scenario, our *start-up* has just published the second version for the digital newspaper introduced in 5.1.2, and the next step is to carry out a maintenance process during the first month since launch. The objective is to analyze the user experience over the app, extracting relevant comments from app reviews and try to solve the errors detected as soon as possible. It seems really tedious to read each review and manually filter those that refers to bugs, and discard the rest. For this reason, the development team think that the developer-oriented section included in the smart agent developed for this thesis could speed up this task.

The main objective is to collect the application feedback posted in Play Store, classify it using the bug classifier designed for the system which is hosted in the server, and finally answer accordingly to that bug, so the users do not feel ignored. Afterwards, for example those bugs could be grouped and posted inside *Trello's* company board, so they can be stacked and solved simultaneously.

The development team representative accesses to the developer-oriented tab and visualizes which is represented in figure ???. We suppose that the agent has previously been linked to the Google Play developer company account. The input data must be the application name and the version that is currently published in production. The package name is also mandatory in order to identify the application. As a reminder, remember that this section can only retrieve reviews from applications published under the linked developer account.

After pressing the *SCAN* button, the smart agent communicates with the Reply to Reviews API, and in case the app belongs to the authenticated account, it will obtain all



App name / Version

Ex: The Guardian/1.1.0 ...

Package

Ex: com.example.myapp

SCAN

REPLY

Figure 5.16: Input parameters requested for developer-oriented scanning.

the review comments posted during 15 day ago. After obtaining them, it's necessary to evaluate them calling the remote bug classifier. In case some review is tagged as a *bug*, the system will randomly extract a friendly response from a bug collection replies. The result is shown in figure 5.17. You can observe that the user is complaining about a crash that happens after the app log in process, so it is clearly a bug. The system has extracted the following recommended answer: *We apologize for inconvenience, we will try to solve it in the next version. Thanks for your patience.*

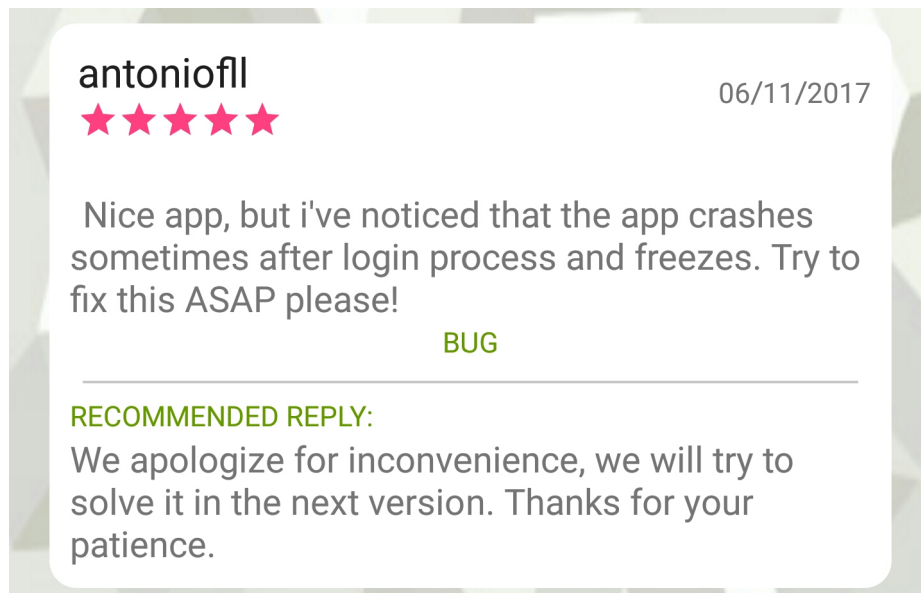


Figure 5.17: Result obtained after retrieving recent app reviews, classify them as a *bug* or not, and finally recommend a suitable reply.

When the *REPLY* button is pressed, all the recommended reviews for reviews classified as bugs will be posted to the Play Store endpoint. The reply will also be sent to the author as an email, but that process acts regardless of the agent. Anyway, the publishing process can take a while until it appears in the website. Afterwards, the output will look like the figure 5.18.

A better approach of this process could be the weekly execution of a pipeline that performs the same action, being able to use a scheduler to program its execution. We will show other possibilities in future work section.

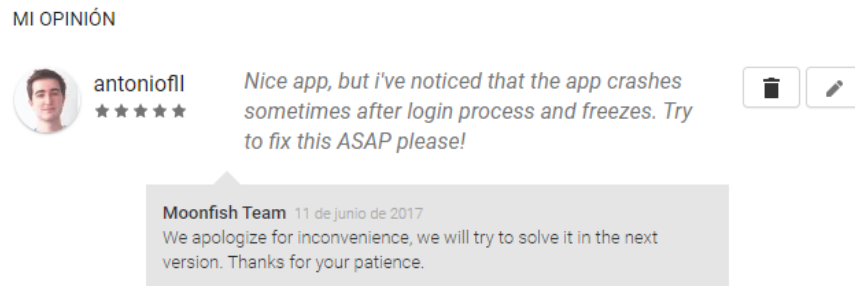


Figure 5.18: Automated reply visualization inside Play Store website.

5.2 Conclusions

To conclude the case study, we have evaluated the implemented smart agent inside several situations, explaining the high utility contributed to a small app development *start-up*. In terms of scalability, it offers lots of new functionalities implementation, being able to design new modules oriented for other purposes, not only sentiments or emotions analysis, but even suggestion mining or user experience extraction.

We have presented standard scenarios that certainly could happen in real life. Every situation has successfully been solved using the smart agent, obtaining good results in most of cases. Finally, all the technologies that participates in the system proposed can be easily deployed in almost any device, using *Docker* for server deployment and the mobile application which actually runs in every smart-phone with the Android operative system installed. To conclude, we will detail more conclusions inside chapter 6.

Conclusions and future work

To conclude this project we will resume the principal concepts that are explained in this document, evaluating the achieved goals and finally detailing some design and implementation ideas about future work that are interesting for this master thesis.

6.1 Conclusions

To conclude this master thesis document, we are going to recapitulate about the project implemented. We have developed a complete integrated system which enables Google Actions through the natural language understanding platform API.AI. The implemented smart agent for Android devices offers a great possibility to evaluate and analyze the feedback obtained from the Play Store market place using sentiment analysis techniques and binary classifiers, revealing interesting data about user experience. It also offers an innovative design for smart-phone apps, with an intuitive interface that handles voice commands to interact with all system components.

The designed system can be used by freelance developers in order to analyze the application market status and track the feedback obtained, being able to extract a real meaning from user opinions and redirect those feedback directly to improvements or error detection tasks. This kind of tool isn't common nowadays, and earn so much time and money to newly created company that has limited budget to manually analyze their user feedback.

The proposed system architecture offers much flexibility in deployment terms. The *docker* container offers a multiplatform coverage, being able to launch locally a single instance and develop other similar applications for the same server controller, such as a website dashboard that groups the same functionalities oriented to other platforms, integrating the system with iOS App Store instead of just focus on Android Play Store.

Moreover, the API.AI agent enables Google Home integration, being able to access the server resources through voice commands, obtaining the output result in audio format. However, some functionalities like bug o feature classification have less sense this way. Anyway, the Google Home integration allows to develop new oriented modules to the smart agent, for instance a recommendation module that suggest the user new applications based on their recent queries.

Finally, the complete project presents a well-formed structure, with an innovative and scalable solution, with a server side compatible with many other platforms, an API.AI agent with high performance and a mobile application that acts as the smart agent that follows the new application model based on smart assistants who listen user queries and manage the communications with the rest of the system components.

6.2 Achieved goals

The achieved goals obtained developing this project are:

Create an interactive user interface where Play Store application can be explored, being able to interact with a chat bot using natural language instead of commands.

Apply sentiments and emotions analysis techniques to the Play Store marketplace, obtaining intrinsic meaning from recent reviews, which enables to know how a certain application audience behaves.

Detect bugs and features from binary classification, using a Naive Bayes classifier to tag reviews of a certain app. This classification process allow to understand user experience and enables to mine most requested improvement from an app audience, in order to develop new features and obtain well-rated reviews.

Compare different Natural Language Understanding systems and observe advantages and inconveniences between them in performance terms, finally choosing the API.AI platform to develop an intermediate service to communicate the logic hosted by the server and the mobile application where users acts.

Create a developer-oriented section from where the feedback answering could be automated, recommending several reply in case bugs or features are detected in a review. This is really helpful and allows companies to better process the information provided by the store.

Analyze the top 60 most downloaded free applications from Play Store ranking, being able to analyze the sentiment evolution inside a limited period of time. Furthermore, knowing what is currently trendy in application market helps to innovate in similar projects and in some way 'assure' a good reception in user community.

Develop a stable architecture with a robust server controller that manages every request made from the agent. The API REST developed in *Flask* offers a service for other applications, being able to use both classifiers and analyzers inside external projects.

Machine Learning techniques have been applied inside multiple modules of the project, improving pattern recognition, and consequently the parameters or intent extraction process inside the API.AI agent, and also increasing the accuracy of classifiers training the with recent reviews, being able to obtain a better output.

6.3 Future work

The high scalability offered by the developed systems raises a lot of possible improvements or future work to be done. Then we are going to mention some of the most interesting objectives to accomplish that haven't been implemented into this project due to the time limitation:

- Adapt the server modules and the smart agent interface to other languages, more concretely to Spanish. The training data-set was only available for English, and set-up a new review set from the beginning was really tedious. Also the API.AI agent only admits a single language configuration, so replicating the agent one for each was a bit messy for agent communication. If only we can generate a well tagged review set in Spanish and clone the API.AI agent into another language, the agent possibilities will be highly increased.
- Develop new classification methods based on the average rating or the user experience detection in order to extract even more valuable information from app reviews.
- Obtain latest application downloaded by the user and develop a recommendation system based on application of the same category place in a high position inside the ranking. In the same way, if only the Play Store offers more data, it would be interesting to collect in which app comments each user, and make recommendations based on similar hobbies, but for the moment it's quite impossible to realize this approach.
- Integrate the smart agent with Google Home, adding a search module from input categories. This way users may find unknown applications that could share some similarities with other popular ones.
- Improve the conversational flow inside the chat module, specially with voice inputs because it's really hard to recognize the application names in case they are quite weird. For this it will be nice to attach an application names set to the API.AI training section.

Bibliography

- [1] D. L. F. P. Caroline Wisniewski, Clément Delpuech and J. Dureau., “Benchmarking natural language understanding systems.” <https://snips.ai/content/sdk-benchmark-visualisation/>, Jan 2017.
- [2] W. M. C. Stanik, D. Pagano, ““Review 2015 Training Set - SQL format”.” <https://mobis.informatik.uni-hamburg.de/wp-content/uploads/2015/03/Data-and-Coding-Guide-.zip>, 2015.
- [3] J. F. Sánchez-Rada, C. A. Iglesias, I. Corcuera-Platas, and O. Araque, “Senpy: A Pragmatic Linked Sentiment Analysis Framework,” in *Proceedings DSAA 2016 Special Track on Emotion and Sentiment in Intelligent Systems and Big Social Data Analysis (SentISData)*, October 2016.
- [4] “Python java: A side-by-side comparison,” May 2009.
- [5] M. L. S. C. NYC, “Python programming language advantages disadvantages,” Aug 2011.
- [6] A.Ronacher, “Opening the flask: How an aprils fools’ joke become a framework with good intentions,” 2011.
- [7] A.Ronacher, “Welcome to flask — flask documentation (0.10),” Jan 2016.
- [8] C. Weekly, “Write once, run anywhere?,” 2009.
- [9] Google, “Material design guidelines.” <https://material.io/guidelines/material-design/introduction.html>, April 2017.
- [10] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, “Appintent: analyzing sensitive data transmission in android for privacy leakage detection,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS ’13, (New York, NY, USA), pp. 1043–1054, ACM, 2013.
- [11] A. D. Website, “Application fundamentals.” <https://developer.android.com/guide/components/fundamentals.html>.
- [12] A. Soffer, D. Konopnicki, and H. Roitman, “When watson went to work: Leveraging cognitive computing in the real world,” in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pp. 455–456, ACM, 2016.
- [13] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell, “An overview of machine learning,” in *Machine learning*, pp. 3–23, Springer, 1983.

- [14] C. A. Iglesias, J. F. Sánchez-Rada, G. Vulcu, and P. Buitelaar, “Linked Data Models for Sentiment and Emotion Analysis in Social Networks,” in *Sentiment Analysis in Social Networks*, ch. Linked Dat, pp. 46–66, Morgan Kauffman, October 2016.
- [15] M. M. Bradley and P. J. Lang, “Affective norms for english words (anew): Instruction manual and affective ratings,” in *Technical report C-1, the center for research in psychophysiology*, 1999.
- [16] S. P. T. Pedersen and J. Michelizzi, “Wordnet:: Similarity: measuring the relatedness of concepts,” in *Demonstration papers at HLT-NAACL 2004*, 2004.
- [17] G. UPM, “Senpy playground.” <http://senpy.cluster.gsi.dit.upm.es/>.
- [18] M. Cloud, “Advantages of automatizing sentiment analysis applications.” <https://www.meaningcloud.com/products/sentiment-analysis>.
- [19] H. N. Walid Maalej, “Bug report, feature request, or simply praise? on automatically classifying app reviews.”
- [20] K. P. Murphy, “Naive bayes classifiers.” <https://datajobsboard.com/wp-content/uploads/2017/01/Naive-Bayes-Kevin-Murphy.pdf>, Oct 2006.
- [21] L. Torgo, “Data mining with R: Learning with case studies,” in *Data Mining and Knowledge Discovery Series. Chapman Hall/CRC*, 2010.
- [22] T. ReadTheDocs, “Tutorial: Building a text classification system.” <http://textblob.readthedocs.io/en/dev/classifiers.html>.
- [23] P. S. L. Documentation, “Pickle - python object serialization.” <https://docs.python.org/2/library/pickle.html>.
- [24] G. Inc, “Material design guidelines.” <https://material.io/guidelines/>.
- [25] Google Play Developers API, “Authorization - User guide to connect the *ReplytoReviews* API with Android devices.” <https://developers.google.com/android-publisher/authorization>.