SAPIENZA
UNIVERSITÀ DI ROMA

Customer-driven design, implementation and evaluation of an intent-based conversational agent using NLP techniques

Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti"

Corso di Laurea Magistrale in Engineering in Computer Science - Ingegneria Informatica

Candidate

Nicolò Palmiero
ID number 1918734

Thesis Advisor

Prof. Ioannis Chatzigiannakis

Academic Year 2020/2021

Thesis defended on 21 March 2022
in front of a Board of Examiners composed by:

Prof. Maurizio Lenzerini (chairman)
Prof. Roberto Beraldi
Prof. Roberto Capobianco
Prof. Ioannis Chatzigiannakis
Prof. Marco Console
Prof. Giorgio Grisetti
Prof. Daniele Nardi

Customer-driven design, implementation andevaluation of an intent-based conversational agent using NLP techniques
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: nicolo.palmiero@gmail.com

Abstract

Conversational agents are becoming more and more popular products in these last few years, for example, contact centers make large use of these kinds of software to automatize the customer assistance when his or her problem is well known and it is easily resolvable from a human operator. Many big companies like *Vodafone*, *GE Appliances* and *Vanguard*, to cite some, are partners of a chatbot service provider. The market is plenty of solutions for building chatbots and conversational agents in an easy way, but are they effective for 100% of the cases? This thesis contains a market analysis to discover the most famous solutions in this field and the design and the implementation of a chatbot building system that aims to satisfy potential customer needs under particular circumstances in which is not convenient to use the most famous products. After explaining the architecture of the product and the technologies hidden behind it, there is a use cases analysis and a final evaluation that aims to understand if the developed conversational agent system is smart enough to be used in a real context.

Introduction

A conversational agent is a piece of software that is capable to interact with a human through a textual or vocal chat. To implement these kinds of solutions nowadays neural networks are often employed, in particular, there are state-of-the-art techniques of natural language processing and natural language understanding that raise the effectiveness of these pieces of software to a good enough level that many big companies like *Google*, *Amazon* and *Microsoft* have created their own chatbot service providers integrated with their cloud services. This thesis aims to find the best solution to integrate a conversational agent service in *Opencommunication* a unified communication product that is sold from the company in which I actually work, *Bytewise*. The first step is to analyze the strengths and the weaknesses of the most famous products that are present on the market, then decide if it is the case to use one of these products or if there are some advantages in implementing a brand new solution. This decision is heavily influenced by the needs and desires of our customers. In fact, it is obvious that in the first step it is very unlikely that the features of a new product overcome the ones of a well-known product like *Dialogflow* or *Amazon Lex* but in certain cases, customers' needs could be in conflict with the choosing of one of these products. Also, the costs of implementation of the chosen solution are a big factor to take into account, in particular in the long term. These considerations are the starting point that led to the design and the implementation of *OC-Chatbot*. This service offers the possibility to the user to build a chatbot that is made suitably for his needs. In fact, all that the user needs is a bunch of examples of questions to which he or she wants the chatbot to answer. These questions have to be categorized in *intents*, to allow the system to build a dataset on which the conversational agent will be trained. So, refining what I have said above, this system allows the user to build a conversational agent that is capable to answer categories of questions that he or she has designed in advance and makes it possible for every client to build a conversational agent that well suits in their domain of interest. During the evolution of this thesis, we will explain this approach in more detail and we will go deep on the weaknesses and the strengths of it evaluating the resulting product. Here is a short summary of each topic that is treated:

In Chapter 1 there is a market analysis of the most famous products and there are some of the reasons which pushed us to decide to implement our own product instead of using one of the above mentioned

Chapter 2 summarizes the technologies giving also a historic background on the evolution of the NLP from the neural networks to the state of the art, and the techniques used to implement our conversational agent solution

Chapter 3 describes the techniques used to implement the conversational agent solution and give details of the architecture of the entire service

Chapter 4 analyzes 3 possible use cases of our chatbot system, one of which comes from one of my company's customers, that deals with legal consulting for doctors

In Chapter 5 are described the results of this work, and the way in which it was decided to evaluate performances of the conversational agent system

In Chapter 6 there are some ideas to improve the product and to make it more effective and more appealing for a potential customer

Contents

List of Figures

List of Tables

Chapter 1

Market analysis

This is an initial market analysis on the feasibility of a solution of an integrated chatbot for *Opencommunication*, a unified communication system for call centers. This analysis is forwarded to discover what the market offers for what concerns the development and the integration of digital assistants for call centers. At first sight, we can easily discover that there are three alternative ways to follow:

- **Cloud solutions**: this kind of solution is the most common, in fact, big companies like Google, Amazon, and Microsoft are on the market with their own tools to develop a chatbot.
- **Open-source solutions**: these solutions are also present on the market but the fact that they are open-source does not necessarily mean that they are completely free.
- **Develop a new solution**: the final goal of this analysis is for sure to evaluate if it is worth buying an already implemented solution to develop a chatbot or to build our own in terms of costs and flexibility to our needs.

We are going to analyze three big cloud products:

- *Google Dialogflow*
- *Amazon Lex*
- *Microsoft bot services*

For the second category, we are going to analyze *RASA*, which is a product that offers an open-source module. The challenge behind this analysis is to choose the most flexible and cheap solution to present to a potential customer, showing if it is worth developing a custom chatbot while a lot of products of this kind are already available in the market. Let's see in more detail what are the strengths and the weaknesses of all the considered solutions.

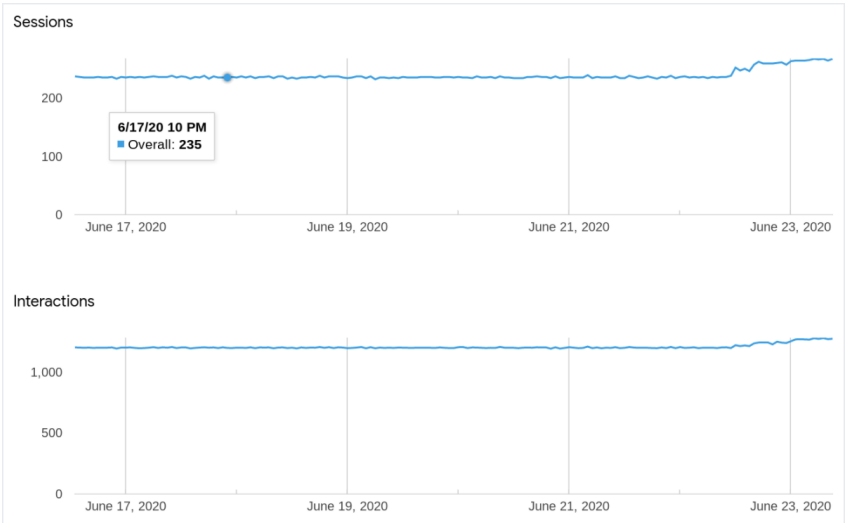
1.1 Cloud solutions

Cloud solutions have a big advantage w.r.t. the other two alternatives: they are well-tested products coming from big companies, so the expectation is that they will be continuously developed and updated with state-of-the-art language models and with new features that will make it simpler to develop an effective chatbot. Coming

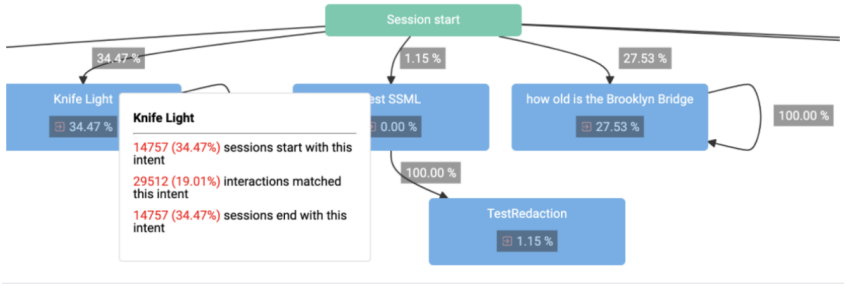
to the negative aspects the pricing is not a point in favor of adopting this kind of solution, in fact, a potential customer could not be interested to build an expensive solution to implement a digital assistant, the second negative aspect is that there are some customers, that cooperate with the company where I work, that do not want their chats data to flow out of their servers due to company policies, in this case, to adopt these solutions could not be afforded. Although these last sentences could make us think about taking another way, it is certainly worth making a deeper analysis of the strengths and the pricing of the biggest three products that are on the market at this time, because when the above limitations do not occur they are the fastest and the easiest way to develop a chatbot.

1.1.1 Dialogflow

One of the most appealing features of Dialogflow is that it is software that is running for years, so its maturity is certainly superior to any custom solution that could be developed. It is a service offered by Google and it offers a wide variety of customizations that are thought to match with a wide variety of services, from the most common applications like Whatsapp, Telegram, and slack, to the major CRM services and contact center products. This is possible through a REST API, that is also well documented. One fundamental aspect is that it has support to almost any language, and for our use case, support to Italian and English language is strictly required. Dialogflow implements a very intuitive analytics system that is not so trivial to develop starting from scratch, and this is a big point in favor to adopt this solution. Through the analytics page of Dialogflow, the developer or the system administrator could have access to some data concerning the usage of the last deployed chatbot, like the number of sessions and interactions, or other plots concerning the behavior of the chatbot during a specific chat, like the intent path view.



(a) Sessions and interactions trend plot



(b) Intent path view

Figure 1.1. Two examples of analytics plots in Google Dialogflow
Source: https://dialogflow.cloud.google.com/

To develop and deploy a chatbot, Dialogflow offers a dashboard that is pretty straightforward to use: after the creation page, in which the user is prompted with information like the name of the agent and the desired language, there is the possibility to insert a number of intents and a number of entities, that will be useful to the chatbot in its training phase. After the training phase of the chatbot is over it will be ready to be used and to be deployed in production.

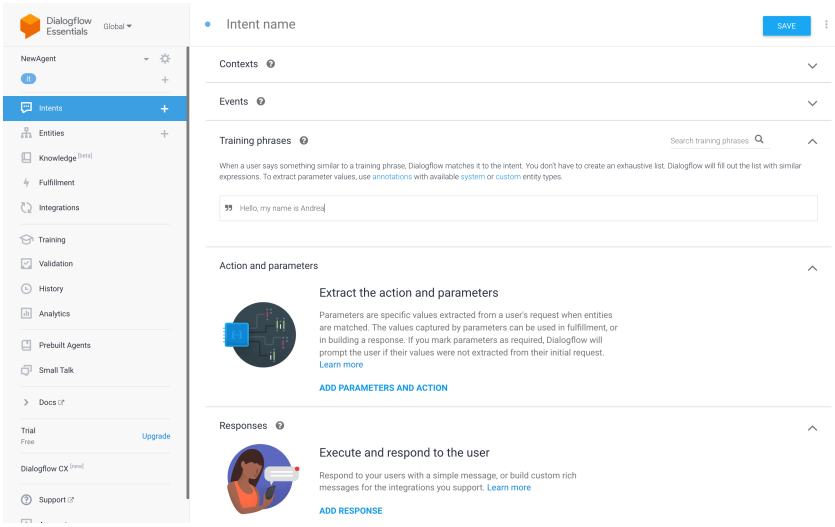


Figure 1.2. A screen from Dialogflow intent creation dashboard
Source: https://dialogflow.cloud.google.com/

Coming to the prices that Google established to make use of these services, it could be a good idea to show a low-level architecture, only equipped with a GPU for training models in an acceptable time, and a high profile machine that has two GPUs in parallel and more memory.

1 x Low-budget	
Region: Iowa	
312.857 total hours per month	
VM class: regular	
Instance type: a2-highgpu-1g	EUR 199.14
Operating System / Software: Free	
GPU dies: 1 NVIDIA TESLA A100	EUR 790.08
Estimated Component Cost: EUR 989.21 per 1 month	

(a) Low-budget machine: it is equipped only with the less expensive GPU available on Dialogflow

1 x High-budget	
Region: Iowa	
312.857 total hours per month	
VM class: regular	
Instance type: a2-highgpu-2g	EUR 398.27
Operating System / Software: Free	
GPU dies: 2 NVIDIA TESLA A100	EUR 1,580.15
Local SSD: 2x375 GiB	EUR 22.13
Estimated Component Cost: EUR 2,000.56 per 1 month	

(b) High-budget machine: it is equipped with 2 GPUs and an SSD storage

Figure 1.3. Two examples of monthly costs to use Dialogflow
Source: https://dialogflow.cloud.google.com/

1.1.2 Amazon Lex

Amazon Lex is the chatbot developing service offered by Amazon AWS, is exposes pretty much the same functions that are offered by Dialogflow, but it relies on a different model, that is the one which is used for Alexa. Like Dialogflow it supports both vocal and chat messages and its functioning is to recognize the user intent and then to associate it an action to answer the user needs in the best way. This service is fully integrated with all the AWS environments, so if the entire system runs on AWS it could be the best choice that a company can do: for example, there is the built-in integration with Amazon Kendra and Amazon Polly. Amazon Kendra is a service that allows the user to define more refined answers to the users' intents,

doing a query through non-structured documents and FAQs. Amazon Polly is a service of TTS (text to speech) that allows the company to build spoken interactions with users starting only from textual data. The third example of integrations that is worth presenting is the one with Amazon lambda functions. These allow the chatbot, once it has recognized an intent, to apply any arbitrary backend logic that will be executed in the AWS environment. This is very important because this solution allows the service to make, for example, REST calls going out of the AWS environment, and give the whole service a lot more flexibility. Amazon Lex is of course already integrated with the contact center software powered by AWS: Amazon Connect, but it is also true that there is the possibility to easily integrate it with other contact center services that are Amazon Partners like Genesys, 8x8, Xapp.ai, Clevy and many others. What is completely different from Dialogflow is the pricing system, in fact, Amazon Lex, like many other services powered by AWS is paid at consumption. This means that every voice or chat message that is elaborated by Amazon Lex has a cost, and the monthly cost is the sum of every single computation in one month. This makes it very difficult to compare prices with Dialogflow, but it is very clear that Amazon Lex is way more convenient for contact centers that do not receive many requests. It is difficult to make a comparison also because Amazon Lex will be used together with many other services from AWS like lambda functions, that are also paid at consumption. Below there are the prices for streaming conversations, both vocal and textual with a little usage example.

Input requests	Cost per unit	Number of units	Total
8,000 speech intervals*	$0.0065	8,000 intervals	$52.00
2,000 text requests	$0.0020	2,000 requests	$4.00
Total Amazon Lex charges for the month			$56.00

*Every 15 seconds of input (including silence) is counted as one interval; any input is rounded up to the nearest 15-second interval.

Figure 1.4. A usage example with pricings from Amazon Lex
Source: https://aws.amazon.com/it/lex/

1.1.3 Azure bot services

The last cloud solution is Azure bot services, developed by Microsoft. It offers a quite different user experience w.r.t. the other two, because there is obviously the possibility to develop a chatbot with a no-code approach, and the basic mechanism with which the chatbot works is the one based on the intent recognition, but in this case, there is the possibility to use predefined open-source modules to improve the quality of the answers, and for each intent, it is possible to define a conversational flow, rather than a simple example sentence, on which the final chatbot will base its answer.

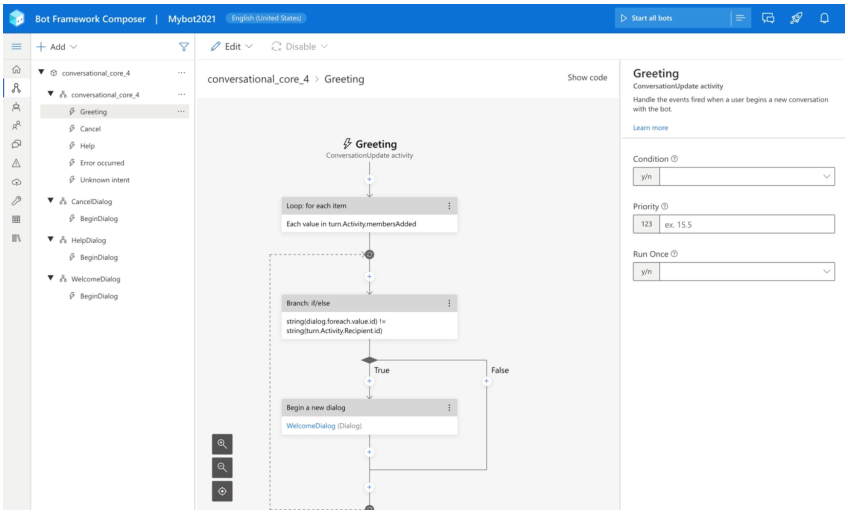
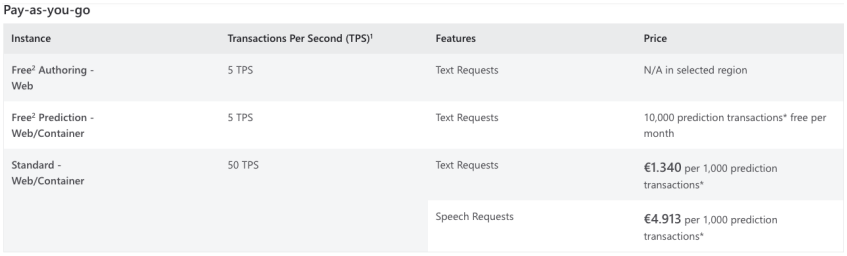


Figure 1.5. An example of a flow diagram used by azure to define conversational flows

Azure, like the other two services, provides an analytics service that is capable to catch important data to understand the effectiveness of the chatbot. In particular, through Azure application insights, the user is capable to catch data concerning both the chatbot's usage and the chatbot's effectiveness, in a similar way as Amazon Lex and Dialogflow do.

Figure 1.6. An example an azure analytics view on the infrastructure usage
Source: https://azure.microsoft.com/it-it/services/bot-services/

Azure has decided to apply a mixed strategy for what concerns the economic aspects of its service. There is the possibility to have a pay-as-you-go subscription in which there is a limited flow of transactions that the system can handle per second, like 5 or 50 *Transactions Per Second (TPS)*. Or the second approach is to use commitment tiers and have a monthly subscription based on the number of requests that the system has received. This leaves the possibility to the client to choose the best way to use the Azure bot service without spending more money than he is supposed to. Below there are two example tables that show both the costs of the pay-as-you-go and the least commitment subscriptions.



Pay-as-you-go

Instance	Transactions Per Second (TPS)[1]	Features	Price
Free[2] Authoring - Web	5 TPS	Text Requests	N/A in selected region
Free[2] Prediction - Web/Container	5 TPS	Text Requests	10,000 prediction transactions* free per month
Standard - Web/Container	50 TPS	Text Requests	€1.340 per 1,000 prediction transactions*
		Speech Requests	€4.913 per 1,000 prediction transactions*

(a) Pay-as-you-go subscription

Commitment Tiers

This pricing is limited access. Apply here.

Instance	Features	Price (per month)	Overage
Azure - Standard	Text Requests	€1,071.764 per 1M transactions €4,554.995 per 5M transactions €19,425.714 per 25M transactions	€1.072 per 1,000 prediction transactions €0.911 per 1,000 prediction transactions €0.778 per 1,000 prediction transactions
Connected Container - Standard	Text Requests	€857.411 per 1M transactions €3,643.996 per 5M transactions €15,540.571 per 25M transactions	€0.858 per 1,000 prediction transactions €0.733 per 1,000 prediction transactions €0.626 per 1,000 prediction transactions

(b) Least commitment subscription

Figure 1.7. Two examples of the sustainable costs using Azure bot services
Source: https://azure.microsoft.com/it-it/services/bot-services/

1.2 RASA

RASA is an alternative to all the three cloud solutions studied so far. Rasa has almost the same capabilities as the cloud solutions, but RASA has one big important advantage: it is possible to deploy it on a proprietary architecture. This solves the problem of keeping out chat data from the customer's environment, but still, there is a problem with this approach. Although RASA is a very flexible solution, there is no way to have any control over the model, and in particular on its output. To make things practical, talking with our customers we found out that one desired feature is that in the situation in which the chatbot does not understand what the customer wants for two consecutive times, it automatically passes the conversation to an operator without any further action from the customer. In this way, they tried to find a solution to offer a good user experience even if the chatbot fails in understanding the user's intent. This is something that could be easily implemented in a custom solution with the introduction of confidence intervals on the model's logits, while with the use of RASA this is not something that can be done in a very straightforward way, because we only have access to the final predictions. So in general we can say that it is legit that the customer wants some kind of customizations that concern the behavior of the model, and the only way to foresee and to try to satisfy them is to create custom software, in which we have full control on the model.

1.3 OC-Chatbot

OC-Chatbot is a custom solution that was thought to integrate a chatbot in Open communication without making the customer sustain relatively high costs but leaving to him a high grade of customization. So it could be the best solution for some of our customers' needs. With a custom architecture, we have more control

over the pricing, in fact, one idea to lower the costs sustained by the client could be to use a GPU equipped machine only for training the model, and to deploy it to a general-purpose machine that will do predictions at runtime in production. This is possible because the prediction task of a neural network is way lighter than the training task, which must be massively parallelized to be completed in acceptable timings. Also with this solution, it will be possible to deploy all the necessary software for the functioning of the chatbot to the customers' environment, and it will be possible to have the maximum grade of customization of the prediction phase of the model.

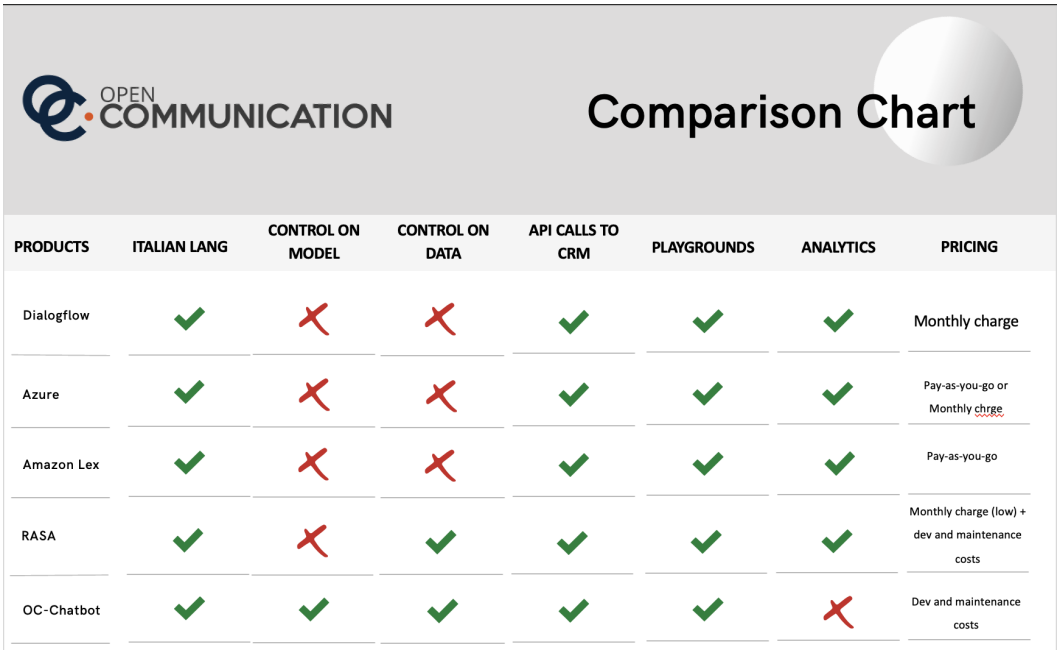


Figure 1.8. A comparison chart between all the alternatives

1.4 Spoken interactions

After having seen the variety of opportunities that the market offers we have seen that there are cases in which is convenient to choose pre-existing products and some other cases in which a brand new product is the best choice to go after the customer's needs. But we can imagine that it is very difficult that a brand new product could arrive in a short time to implement all the features that the most famous products have. One of the biggest features that the market offers, and it is not yet implemented in OC-Chatbot is the spoken interaction. As a future development, it could be very interesting to create another service, that implements a text-to-speech engine and a speech-to-text engine. This service is not very straightforward and it could become an important bottleneck for performances if it does not get almost perfect results. For this reason for this task, it is more reasonable to use already existing services offered by the most famous automated speaker systems like Alexa, Siri, and Google. These are clearly cloud services, and this at first could raise a conflict with the customer's need to keep all its chat data within its environment. But these kinds of services do not need to have any active context on the current chat, and they are completely unaware of any intent that hides behind a particular sentence and they do not need any example of a real sentence to work. They are simply services that offer a REST API and when they are called sending to them a voice message, they answer with a transcription of that message (or vice-versa). With these powerful tools, it is easier to develop a spoken interaction for our product, and we could achieve the goal of following an entire conversation only through spoken interactions. Let's see some examples of these services.

1.4.1 Alexa Voice Services

Alexa Voice Service is a service that is powered by Amazon, that offers the developers the possibility to add a spoken interaction to their applications. It implements this by offering a text-to-speech API and also a speech-to-text one, to allow full-duplex voice interaction. Obviously, to guarantee that these services are not used without authorization, they implement strong security mechanisms, and each request has to be authenticated through a token that is obtained only after having registered the client application to AWS, using the *Login With Amazon* service. These requests' body is a JSON that has to follow a convention that was meant to be used specifically by Alexa Voice Service. The basic spoken interaction, in this setting, is perfectly integrable with OC-Chatbot, in fact, just in case the user will send a voice message, our service will send a request to transcribe it. The model will work on the transcription of the voice message, and after having found the corresponding intent and having computed the most relevant answer, it will be translated to a voice message through a call to another AVS endpoint.

1.4.2 Google API

Another valid alternative to Alexa Voice Sevice is for sure the service offered by Google Cloud, which includes both text-to-speech and speech-to-text APIs. The most significant difference with AVS is that the service offered by Google can also be deployed to private data centers, this means that also a potential customer that in every case does not want that data flows out from its data center can implement this solution. Also, Google gives the possibility to the customer to select the best model that was trained on data coming from a specific domain. This means that a customer that offers contact center services within only a single or a few linked branches could

be very advantaged by this choice, in fact in this way the text-to-speech and the speech-to-text models will be more capable to translate technical terms that are strictly linked to that branch of interest for the company, increasing in this way the overall accuracy of the chatbot service.

1.4.3 Siri SDK

Another interesting future development could be to create a dedicated version of the Open communication Chat widget through an IOS app. In this case, the integration with Siri SDK could be a big point in favor of this approach. In fact, since the release of IOS 10, it is possible to communicate with Siri also for third-party apps. This suggests the first kind of interaction that the app could do with Siri: it could be possible to tell Siri to send a text message only through voice, and the app could read the response that the chatbot has computed on the server-side. Another important point in developing an IOS application could be that Swift (the most used programming language to develop IOS apps) offers text-to-speech and speech-to-text classes, they are respectively so AVSpeechSynthesizer and SpeechRecognizer. So leaving the chatbot service untouched, it could be possible to have a full-duplex voice interaction with the smartphone. These two approaches obviously do not involve the backend part of the OC-Chatbot architecture, but they are client-side developments, so in this case, they could be available only for an IOS app. On the other hand, in the second scenario, this kind of spoken interaction could lower the load of data computed on the cloud, and it could matter to investigate if the performances in terms of speed of computation and quality of the results are good enough. For the second category, we are going to analyze *RASA*, which is a product that offers an open-source module. The challenge behind this analysis is to choose the most flexible and cheap solution to present to a potential customer, showing if it is worth developing a custom chatbot while a lot of products of this kind are already available in the market. Let's see in more detail what are the strengths and the weaknesses of all the considered solutions.

Chapter 2

Used technologies

The *OC-Chatbot* service is a piece of software that makes use of various technologies and it is thought to have the highest flexibility to the customer needs also in its architecture. Before going into detail on the architecture it is important to give the reader a background on the technologies used to implement this service. Once these topics have been assimilated it will be simpler to understand the final architecture. The core technologies employed in this project concern for sure some NLP and NLU topics, so in this section will be a deepening into the concepts behind the *neural networks* that evolved in NLP until arriving at the state of the art, which is *transformers*. The other notable technologies employed in this project are for sure *relational databases* with the use of *ORMs* and the employment of React and Material-UI to build the frontend part.

2.1 Neural Networks

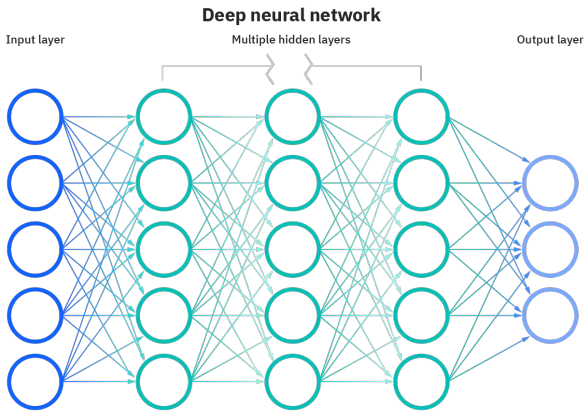


Figure 2.1. A graphical representation of an artificial neural network
Source: https://www.ibm.com/it-it/cloud/learn/neural-networks

As we said the first topic that we meet if we approach the world of Natural Language processing is the concept of Neural Networks. All of the Natural Language processing state-of-the-art techniques have at their bases the employment of a neural network. This is a fundamental concept not only in NLP but in all the machine learning world. Unfortunately for us, there is not a unique and strict definition

of a neural network, so we will start from the definition given in a paper called *"Definition of artificial neural networks with comparison to other networks"* [4] and we will try to explain it. The definition is the following:

Definition 1. *A directed graph is called an Artificial Neural Network (ANN) if it has*

- *at least one start node (or Start Element; SE),*
- *at least one end node (or End Element; EE),*
- *at least one Processing Element (PE),*
- *all the nodes used must be Processing Elements (PEs), except start nodes and end nodes,*
- *a state variable n_i associated with each node i,*
- *a real-valued weight w_{ki} associated with each link (ki) from node k to node i,*
- *a real-valued bias bi associated with each node i,*
- *at least two of the multiple PEs connected in parallel,*
- *a learning algorithm that helps to model the desired output for a given input.*
- *a flow on each link (ki) from node k to node i, that carries exactly the same flow which equals to n_k caused by the output of node k ,*
- *each start node is connected to at least one end node, and each end node is connected to at least one start node,*
- *no parallel edges (each link (k_i) from node k to node i is unique).*

Definition 2. *Start Element (SE) k is a node in a directed graph, which gets an input I_{ij} from the input matrix $I = I_{ij}; i = 1, 2, ..., n, j = 1, 2, ..., m$ of n attributes of m independent records, and starts a flow in the graph.*

Definition 3. *End Element (EE) i is a node in a directed graph, which produces an output O_{ij} from the output matrix $O = Oij; i = 1, 2, ..., n, j = 1, 2, ..., m$ of n desired outputs of m independent input records and ends a flow in the graph.*

So the first thing that we understand from what we said above is that an artificial neural network can be represented as a graph in which we can modify the weights associated with the graph's edges to obtain the desired output given a certain input. This is a sufficient condition to assert that with an artificial neural network we have the possibility to approximate at least any given linear function. Now we have to extend the definition to give the possibility to the neural network to approximate any function adding a non-linearity.

Definition 4. *Let G be a directed graph with the following properties;*

- *a state variable n_i is associated with each node i,*
- *a real valued weight w_{ki} is associated with each link (ki) from node k to node i,*
- *a real valued bias b_i is associated with each node i,*

- *has no parallel edges (links).*

Let $f_i[n_k, wki, b_i, (i \neq k)]$ *be the following function in graph G for node i;* $f_i(u_i, b_i) = n_i\phi(u_i + b_i)$ *(1) where* $\phi(.)$ *is the activation function and* u_i *is as follows;*

$$u_i = \Sigma_{j=1}^{m} w_{ji} n_j$$

The non-linearity that we were searching for is included by the activation function that can be inserted after each node. There are several kinds of activation functions, within the most used ones we have to cite *Sigmoid*, *Tanh ReLu* and *Softmax*. The thing that these functions have in common is that they are non-linear, and they can be used to transform the linear input given by the linear combination of weights and state variables. In this way, by changing the value of the weights, the neural network can approximate any given function.

Definition 5. *Learning Algorithm in an ANN is an algorithm that modifies weights of the ANN to obtain desired output(s) for a given input.*

So the way in which we modify the weights of a neural network is as a learning algorithm that aims to minimize locally a utility function. Very often the utility functions used in this field are measures of the error that is there between the output of the neural network and the label of the sample that was given in input to the NN. The most famous error function is called *Mean Squared Error* and is defined in this way:

$$MSE = \frac{\Sigma_{i=1}^{n}(x_i - y_i)^2}{n}$$

where x_i is the output of the NN and y_i is the label associated to the current sample. n is the number of samples that were considered, and this number is called *batch*.

2.2 Recurrent Neural Networks

Once we have understood what is an artificial neural network we have to know that there exist many kinds of neural networks, that are specifically designed for different tasks. The first kind of neural network that is worth citing in the context of this thesis is *Recurrent Neural Network*. A recurrent neural network is designed specifically to work with sequences, like text, audio, or in any case in which data presents a strong relationship between each other, either temporal or spatial. An RNN takes into account these relations between data making the information flow through a loop instead of going directly from a starting node to an end node like a classical artificial neural network. To be more precise in recurrent neural networks the process is divided into steps, and the output from the previous step is given as the input of the next step. Here there is a graphical representation of a recurrent neural network, to give the reader a good idea of the structure of a recurrent neural network before we go on.

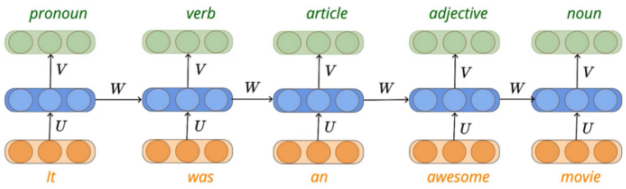


Figure 2.2. A graphical representation of a recurrent neural network
Source: https://towardsdatascience.com/recurrent-neural-networks-rnn-explained-the-eli5-way-3956887e8b75

This picture is a representation of a recurrent neural network that is made to point out the relation between the input, that in this case is a written sentence and the processing unit, but a more exact representation of a recurrent neural network could be this:

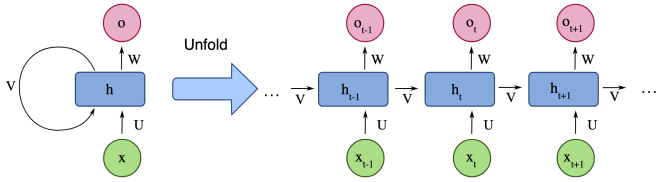


Figure 2.3. Folded and unfolded recurrent neural network
Source: https://it.wikipedia.org/wiki/Rete_neurale_ricorrente

In this last picture is perceptible that the processing unit is the same for each token contained in the input. So the processing unit of a recurrent neural network takes two inputs: the output of the precedent step (or a starting feed if we talk about the first step) and encoding of the current input token. In this way, the neural network is able to memorize information from all the input tokens and correlate this information with other tokens in order to make the final prediction. This is very important in the field of Natural Language Processing because each word in a sentence assumes a different meaning depending on the context in which it is. To make an example, the English word *bank* has two meanings that are completely different in these two sentences:

- Today I will go to the *bank* to take some money
- Today I will take a walk near to the river *bank*

so the context of the sentence is an information that is fundamental in Natural Language Processing. From the mathematical point of view we know that an artificial neural network is represented by the formula $y_i = \phi(\Sigma_{j=1}^{n} w_{ij} x_i + bi)$ or in matrix form : $y = \phi(Mx + b)$. This expression becomes more complex when we have to represent a recurrent neural network, in fact before giving the complete mathematical representation is is useful to define the variable h_t that is the hidden state of the processing unit at time t:

$$h_t = \phi(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$$

This expression is obviously recursive, because the state at time t is bounded to the value of the state at time $t-1$. The activation function ϕ that is used in this context

is a *Tanh* or a *Sigmoid*. W is the matrix containing all the weights of the neural network at a certain step. Once we have defined h_t we can give a mathematical definition of an RNN that is similar to the one we gave for the ANN:

$$y_t = softmax(W(s)h_t)$$

Here we apply the softmax because it is a function that given any function reduces its co-domain to [0,1] making it a probability distribution. If we take the argmax of this probability distribution we are able to find the most probable output within the domain of the input function (i.e. the output of the recurrent neural network at a given stage). The most used loss function that is used together with an RNN is the *multi-class cross-entropy loss function* and this is strictly related to the choosing of the *softmax* as our final activation function. This was a description of how a recurrent neural network works, but what we have not said is that the recurrent neural networks suffer, at least in their most basic forms, from a big problem, that is called *vanishing gradients*. This problem makes the computations of an RNN less and less precise as the size of the input increases. This is due to the fact that in each step always the same weights W are used to compute the output y_t, and during the phase of back-propagation to compute the gradient of the loss function these multiplications between W and h_{t_i} are repeated. So the more two words are distant the more the RNN cannot propagate the error to the initial steps of the computation.

2.3 Long Short term Memory (LSTM)

A way to solve the *vanishing gradients problem* is represented by the LSTM recurrent neural networks. This approach was first presented in a paper by Hochreiter and Schmidhuber in 1997 [5]. The functioning of this particular category of RNNs is appreciable if we compare the basic structure of an RNN with the one of an LSTM

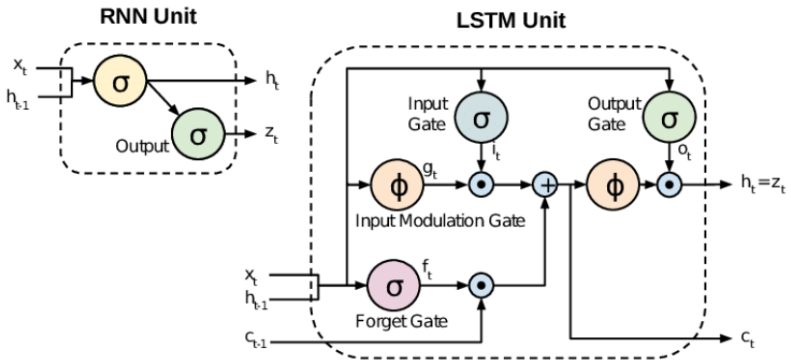


Figure 2.4. Difference between a RNN and LSTM
Source: https://ashutoshtripathi.com/2021/07/02/what-is-the-main-difference-between-rnn-and-lstm-nlp-rnn-vs-lstm

The first impression that these two figures give us is that the difference between an RNN and an LSTM is in their hidden states, in fact, while the hidden state h_t in an RNN has a very simple structure, with only one activation function (*sigmoid* in this case), the LSTM presents a more complex structure. This complex structure takes the name of *Memory cell*. In this representation, the rows point out the flow

that the input token does in the memory cell, while we can notice an important concept that is introduced in this kind of architecture, *gates*. The gates are a way to decide if the information that was given in input can be included or not. The gates are implemented with a *sigmoid*, that is a function that yields in output a number contained in $[0, 1]$. There are three kinds of gates, and their function changes according to their position in the flow of the initial input of the memory cell:

- **Forget gate**: this gate receives the concatenation of the input x_t and the preceding hidden state h_{t-1} and outputs a vector f_t containing numbers from 0 to 1 and very close to these two to decide how much of information to keep from the concatenation of these two elements. For example, if there is a piece of information from the preceding hidden state that is worth keeping, this gate will return a value close to one, instead of this information could be misleading for the current prediction we expect that the gate will output a number close to 0

- **Input gate**: the output of the *Input gate* is multiplied with the output of the *Input modulation gate* that is another activation function applied to the concatenation of x_t and h_{t-1}, like a *tanh* to decide how much information to keep to create the current hidden state h_t. The result of this operation is then added to the output of the forget gate f_t multiplied with another information from the previous step, called c_{t-1}. The result of this process is exactly c_t, which is a representation of the hidden state of the current state memory cell.

- **Output gate** the last gate that we will examine decides how much information will be outputted by the memory cell and contextually how much information will be the input for the next step of the computation. This is implemented by giving it in input the concatenation of x_t and h_{t-1} and adding it to the result of applying the same activation function as the input modulation gate (often this could be a *tanh*) to c_t, this creates the output of the current step memory cell h_t.

The LSTM approach solves the vanishing gradient problem, but it suffers from another problem that until a few years ago didn't allow NLP to be very effective, specifically in some tasks like *machine translation* and *speech recognition*. This problem is that an LSTM can handle only problems in which input and output matrices have to be a fixed size. But this condition in NLP is very often violated two examples of this are the two problems mentioned above.

2.4 Sequence to sequence learning

In 2014 Google's researchers with the publication of *"Sequence to sequence learning with neural networks"*, [9] introduced a new approach in which they proved that the use of LSTMs, in a certain configuration, could overcome even the inputs and outputs size problem. This gave a big boost to the evolution of the NLP, that in fact in the last few years is in continuous evolution. The solution that Google's researchers gave is to use two LSTMs in an Encoder/Decoder fashion. The Encoder/Decoder architecture is a solution that is employed also in other fields of machine learning, and it consists to put together two neural networks with a layer that is called latent space. A basic application of this principle could be the *Autoencoder*

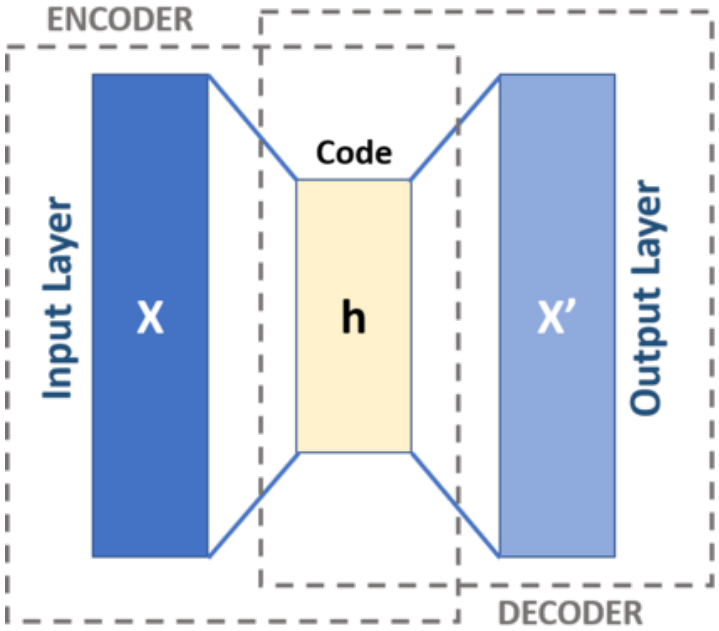


Figure 2.5. High level structure of an autoencoder
Source: https://en.wikipedia.org/wiki/Autoencoder

In the figure, we can see that the first neural network from the left encodes the input from its original representation to a reduced dimension, but yet maintaining the most of the original information, this process takes the name of *Encoding* and the resulting vector is the *Latent space representation of the input*. The right hand of this architecture takes in input the latent vector and tries to reconstruct the original input. This is called *Decoding* and the training of this structure consists of training together both the encoding and the decoding part, with the encoder that creates the latent vector end the decoder that tries to reconstruct the original input starting from the output of the encoder. From this last statement, we can understand the success that this method had in unsupervised learning. The idea behind *Sequence to sequence learning* is simply to use two LSTMs, one as the Encoder and the other as the Decoder.

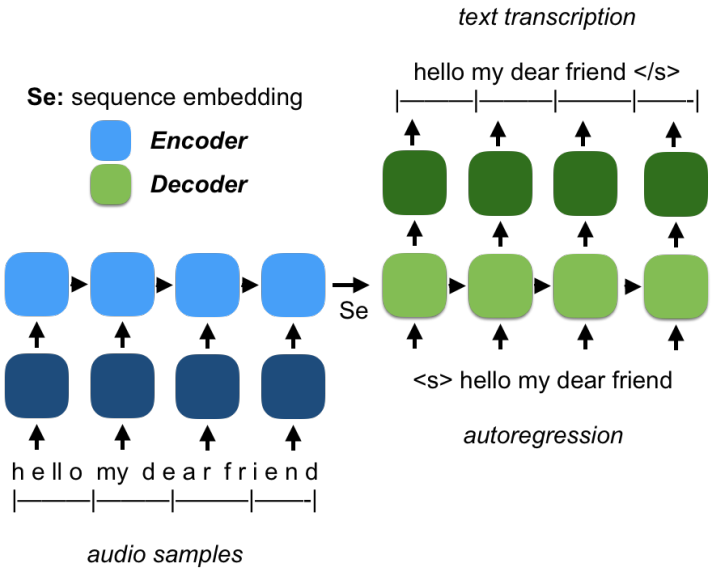


Figure 2.6. High level structure of a sequence to sequence learning architecture
Source: https://culurciello.medium.com/sequence-to-sequence-neural-networks-3d27e72290fe

The big power of this model is that the encoder and decoder are strongly decoupled, so the encoder can have an input size that is totally different from the decoder output size. In the example that is represented in the picture, we have that that the encoder takes in input a vector that is large t temporal samples (the speech is sampled with a frequency f), while the size of the output depends on the length l of the sentence that is represented in the input.

2.5 Word Embeddings

Now that we have seen how an Encoder/Decoder architecture is made before going on it is worth seeing how sentences are manipulated to be fed to a neural network, that accepts only numerical vectors and matrices as input. *Word Embeddings* is one of the most popular ways to represent numbers as vectors. The goal of Word Embeddings is to represent similar words in meaning nearby each other, such that it is easy to capture the meaning of a sentence for any processing unit. To be precise, the angle between the vectors representing two similar words has to be little. We will see that this task is quite hard to perform, and various techniques were tried to do this. One of the most famous techniques to create Word Embeddings, but not the most effective is Word2Vec. It exploits two methods, both employing neural networks *CBOW* and *Skipgram*. The first method is called *CBOW Model*: takes the context of each word as the input and tries to predict the word corresponding to the context[1]. For example, consider the sentence *"Have a great day"*. Let the input to the Neural Network be the word, great. Notice that here we are trying to predict a target word (day) using a single context input word great. More specifically, we use the one-hot-encoding of the input word and measure the output error compared to a one-hot-encoding of the target word (day). In the process of predicting the target word, we learn the vector representation of the target word.

[1] https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

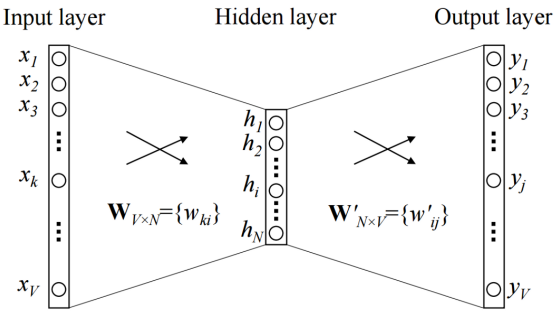


Figure 1: A simple CBOW model with only one word in the context

Figure 2.7. CBOW architecture with one word in context
Source: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

The only activation function that is present in this model is the *softmax* that is applied to the outputs. The input is the one-hot-encoded representation of the context word of size V. The hidden layer contains N neurons and then the output has also size V. The hidden layer neurons just copy the weighted sum of inputs to the next layer, without an activation function. But the model above shows how this works given in input only one context word. The real case is to use this model with k context words in input, and its output will be anyway a single vector of size V that summarizes the context vectors.

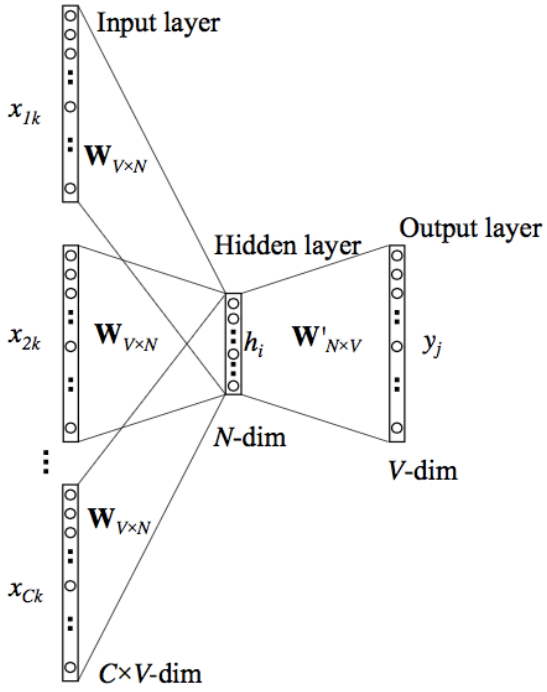


Figure 2.8. CBOW architecture with more than one word in context
Source: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

For what concerns *Skipgram* we could say that is like an inverted *CBOW* because we input the target word into the network, and the model outputs C probability distributions. For each context position, we get C probability distributions of V probabilities, one for each word.

2.6 Attention

The negative aspect of the *Sequence to Sequence* architecture is that for long input sequences, the latent vector cannot keep enough information to work optimally, to improve this aspect, yet keeping a similar architecture, the concept of *Attention* was introduced. In fact, in 2014 the researchers realized that the encoding in the latent state of a long enough sequence could lead to an important loss of information, this is because the reduced and fixed dimensions of the latent space worked as a bottleneck for the entire system. To overcome this difficulty the concept of *Attention* was introduced by the paper *Neural machine translation by jointly learning to align and translate* [1]. It consists in spreading the contextual information of the input across the layers of the encoder, in such a way that in the decoding operation, the Decoder could use that information to take them into account during the computing. In more detail this is implemented allowing the Decoder not only to access the information from the latent vector, but now it can access all the hidden layers of the Encoder, so the formula for computing H_k that is the k^{th} hidden state of the decoder becomes also a function of C_k that is called context vector, and it contains a weighted sum of all the hidden states from the Encoder

$$H_k = f(H_{k-1}, Y_{k-1}, C_k)$$

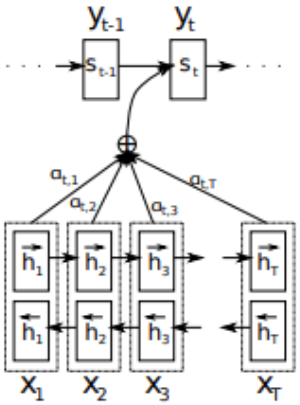


Figure 2.9. Graphical illustration of a Sequence to sequence model using attention
Source: https://arxiv.org/pdf/1409.0473.pdf

The context vector weights α_{ti} are called *global alignment weights* and they are learned by a feed-forward neural network that is called *attention layer*. The fact that also these weights are learned is fundamental because they cover an important role in the flow of information that arrives at the decoder LSTM. They have the role to determine which size of the input is important to decode correctly the latent vector.

2.7 Transformers

In 2017 Google's researchers took this Attention concept to the next level, implementing an architecture that relied only on this concept instead to be bound to the Encoder/Decoder architecture, this idea took the name of *Transformers*. Transformers were presented in a paper that was called *"Attention is all you need"* [10] and this gave birth to the state of the art technologies that even today are still used in many tasks of NLP. The kind of Attention that we saw in the Sequence learning is also called *Encoder/decoder Attention*, while the type of attention that the Transformer exploit took the name of *Self-attention*. The Self-attention mechanism consists in trying to relate each other to different positions within a given sentence to compute an appropriate representation of it. Beyond the concept of Self-attention, two other concepts hide:

- **Self-attention in the input sentence**: that is comparing all the words in the input sentence
- **Self-attention in the output sentence**: that is comparing all the words in the output sentence, but we should be aware that the scope of Self-attention is limited to the words that occur before the current one. And this is done by masking the words that occur after it each step. So for step 1, only the first word of the output sequence is NOT masked, for step 2, the first two words are NOT masked, and so on.

To correctly calculate the Self-attention we need three elements[2], these are:

- **Query vector**: $q = X \cdot Wq$ Think of this as the current word.

[2] https://towardsdatascience.com/transformers-89034557de14

- **Key vector**: $k = X \cdot Wk$ Think of this as an indexing mechanism for the Value vector. Similar to how we have key-value pairs in hash maps, where keys are used to uniquely index the values.
- **Value Vector**: $k = X \cdot Wk$ Think of this as the information in the input word.

Once we have these data structures the task that we want to solve is to find within the sentence the most similar vector to the current token, which is represented by the vector q. We do this by doing the dot product $q \cdot k$. The closest query-key couple will result in having $softmax(q \cdot k)$ with smaller values close to 0 and larger values close to 1. Then this *softmax* distribution will be multiplied with v, in this way the components multiplied with 1 will stay, while the components multiplied with 0 will be filtered. The sizes of v, q, k are referenced as *hidden size*.

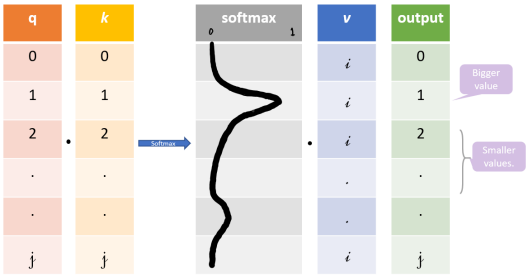


Figure 2.10. Representation of the vectors q, k and v
Source: https://towardsdatascience.com/transformers-89034557de14

Now we are ready to analyze the full *Transformer* architecture. Here there is an image in which we can observe both a high-level representation and both a more detailed one

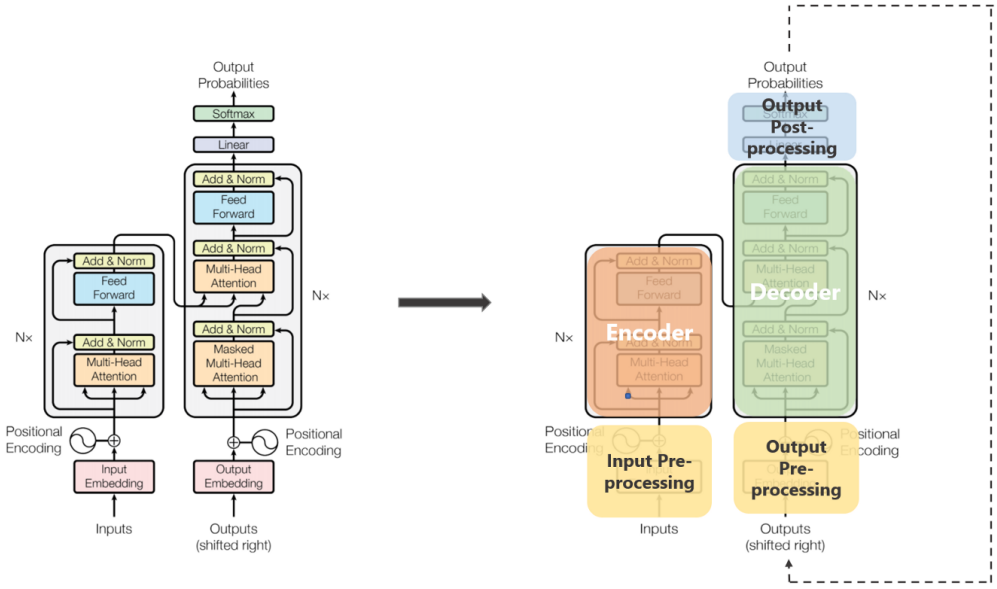


Figure 2.11. Representation of the architecture of a transformer, at different levels of detail
Source: https://towardsdatascience.com/transformers-89034557de14

Let's explain this image starting by analyzing the blocks that are represented on the right hand. The first block is the *Encoder*, which is in charge to encode the input to its representation in a reduced dimensions space, it achieves this by computing the outputs from Self-attention for each word with some post-processing. Each encoder has two sub-modules:

- **A Multi-head self attention layer**
- **A simple, position-wise fully connected feed-forward network** we can consider it like the post-processing mentioned above.

The second element of this architecture is the *Decoder* that has three sub-layers: the last two are similar to the encoder, but in the first position a **masked multi-head self-attention mechanism** on the output vectors of the previous iteration is added. Then the **A Multi-head self-attention layer** of the Decoder takes into input the output from the encoder and the masked multi-headed self-attention layer. Now we want that the transformer is able to focus on different positions of the sequence computing many times the Self-attention with different q, k, v, each computing of this kind is called attention-head. When all these processes end the resulting outputs are concatenated that is scaled down to a lower dimension with another matrix W_0 that is also trained. The third element that we have to analyze is the *Input and Output pre-processing* in fact each word is represented through an embedding, and this is done for both the Encoder end the Decoder part. But word embedding does not contain any positional information of the token that they represent, and the computing of the Self-attention the positional information of the starting tokens is lost. To solve this problem before applying Self-attention the word-embeddings are multiplied with some periodical functions that the transformer is able to learn, and so to restore the positional information that was lost by the

application of the *softmax* while computing Self-attention. The result of this pre-processing is called *positional encoding*. The last element that has to be explained is the *Decoder stack* in its whole. We can notice from the image that the output of the decoder will be the input of it in the next step of the computation. This means that the new input is pre-processed and becomes a positional embedding too. This is given in input to the Multi-head masked self-attention layer. This layer is modified to consider only positions concerning the previous steps of the computation, assuring that the current output can depend only on the known outputs. The outputs from the encoder stack are then used as multiple sets of key vectors k and value vectors v, for the *encoder-decoder attention layer*. It helps the decoder focus on the contextually relevant parts in the input sequence for that step. The q vector comes from the *output self attention layer*. Once we get the output from the decoder, we do a *softmax* again to select the final probabilities of words.

2.8 BERT

Bidirectional Encoder Representations from Transformers better known as *BERT* is explained one of the most recent papers by Google AI researchers [3], it made the popularity of the NLP increase because it obtained the state-of-the-art results in most of the NLP tasks, like *question answering* or *Natural language inference*. BERT's greatest innovation was to apply the bidirectional training of a transformer, that is a popular attention model, to language modeling. The paper has described a technique called *Masked LM (MLM)* which allows bidirectional training in models in which before it was impossible[3]. Another big success that made BERT so popular is that its training phase can be split up in two parts: pre-training and fine-tuning. The third good feature of BERT is that it uses word-piece tokenization to produce embeddings. This is useful as opposed to the traditional word embeddings like GloVe or Word2Vec it is not necessary to maintain a dictionary with the handling of unknown tokens, so from this, the model can gain more accuracy in its predictions. As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore it is considered bidirectional, though it would be more accurate to say that it's non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

2.8.1 Masked LM (MLM)

Before feeding a sentence to BERT 15% of the words are substituted with the [MASK] token, the model then tries to predict the original value of the masked words, based on the context, that is red bidirectionally

[3] https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

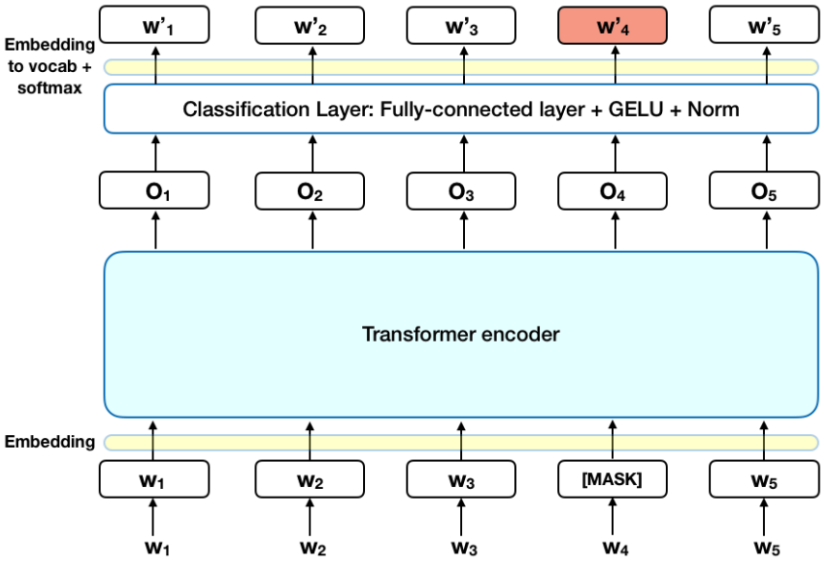


Figure 2.12. BERT structure highlighting the MLM process
Source: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

BERT loss function takes into account only predictions about masked tokens and does not compute the error on non-masked ones, this makes the model converge slower than other models. In the pre-training phase, BERT is fed with two sentences and it has to predict if one sentence is subsequent to the other. The dataset with which BERT is trained is made from 50% of sentences that precede each other in the document corpus, and the other half are two sentences sampled at random. To help the model distinguish between the two sentences during training the input is pre-processed adding some tokens like [CLS] which is a token that is inserted at the beginning of the input and [SEP] that is inserted between each couple of sentences. Then this enriched sentence passes through a sentence embedding layer, which for each token produces an embedding that indicates if each token belongs to sentence A or sentence B. After this, the last step of the pre-processing is to add a positional embedding to each token, in a similar way in which this is implemented in the traditional Transformers.

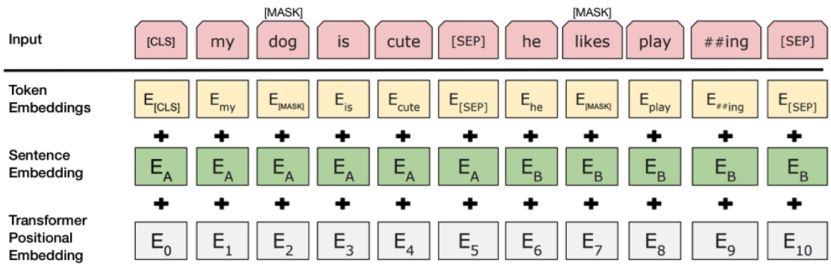


Figure 2.13. BERT structure highlighting the MLM process
Source: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

To predict if the two sentences are concatenated the following steps are performed:

1. The pre-processed sentences are fed to the Transformer model

2. The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases)

3. using *softmax* the probability of the two sentences to be concatenated is computed

When training the BERT model, Masked LM and Next Sentence Prediction are trained together, with the goal of minimizing the combined loss function of the two strategies.

2.8.2 Fine tuning

After going through the pre-training phase BERT can be used for some different tasks, like *Sentence classification*, *Question answering* and *Named Entity Recognition (NER)*. There is a specific way to use BERT depending on the task in which it is applied:

1. *Sentence Classification* tasks such as *Sentiment Analysis* or *Intent Recognition* are done similarly to Next Sentence classification, by adding a classification layer on top of the Transformer output for the [CLS] token.

2. In *Question Answering* tasks (e.g. SQuAD v1.1), the software receives a question regarding a text sequence and is required to mark the answer in the sequence. Using BERT, a Q&A model can be trained by learning two extra vectors that mark the beginning and the end of the answer.

3. In *Named Entity Recognition (NER)*, the software receives a text sequence and is required to mark the various types of entities (Person, Organization, Date, etc) that appear in the text. Using BERT, a NER model can be trained by feeding the output vector of each token into a classification layer that predicts the NER label.

In the fine-tuning phase, most hyper-parameters stay the same, and the paper gives specific guidance on the hyper-parameters that require tuning.

2.9 Natural Language Processing tasks

Until now we have described the technologies that boosted the spread of Natural Language Processing, but now the time has come to go down in more detail about the tasks in which we are interested in the design and the development of this project, that are two:

- Intent Recognition
- Named Entity Recognition (NER)

Both of these two tasks are possible employing BERT or a similar model. Let's explain the problems that they hide and the way in which is possible to solve them.

2.9.1 Intent Recognition

Intent recognition is a classification task that consists in taking a written or spoken input, and classifying it based on what the user wants to achieve[4]. To better understand this sentence it is useful to make an example: let's say that we pick four categories that are labeled in this way: *Greeting, Goodbye, Weather, Date*. Now we want to classify an input like "What's the weather like today" in one of these four categories. If the Intent recognition model works in the right way, in this case, its prediction will be 'Weather'. Now that we have a category to which the input sentence belongs it is easy to associate an action or an answer to this one. Training data is a representative sample of raw data that is manually labeled or organized in the way you eventually want your model to do automatically, often there will be an input sentence and its ground truth that will be the label that is associated with it. Labeling the data, in this case, is very time consuming, because there is no simple way to do it automatically, also in this sense the use of BERT helps a lot, because it is a pre-trained model that needs only to be fine-tuned, so it does not need such a big amount of data to perform well. Once we have labeled the data, we can feed it into the ML model so it can learn what we expect it to do. This task is at the base of the functioning of most of the chatbots that are on the market, including the ones that were described in Chapter 1.

2.9.2 Named Entity Recognition (NER)

Named Entity recognition is our second task of interest in this project, and it is one of the most successful techniques to build a conversational agent. Let's see why. Named-entity recognition (NER) (also known as (named) entity identification, entity chunking, and entity extraction) is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc[5]. An example of a problem of this kind could be: a NER model takes into input the sentence "My name is Nicolò and I study at Sapienza University", and we have some predefined categories like: *Person*, *Place*, *University*, *MISC*. If the model works correctly it will output for sure something like that: *My: MISC, name: MISC, is: MISC, Nicolò: Person, and: MISC, I: Person, study: MISC, at: MISC, Sapienza University: University*. This task is particularly useful for the implementation of a conversational agent that has the goal to give the impression of talking with a real person. This is because using NER a conversational agent could potentially memorize some important information from the previous messages of a conversation to reuse that data to give a precise response without the need that the customer repeats a piece of information that he has already said.

[4] https://medium.com/mysuperai/what-is-intent-recognition-and-how-can-i-use-it-9ceb35055c4f

[5] https://en.wikipedia.org/wiki/Namedentity_recognition

Chapter 3

Architecture

After we have seen most of the background knowledge that hides behind this project we are ready to analyze its architecture. The first thing to know about this project is its core functionalities, so the Natural Language Processing tasks that it accomplishes, are *Intent Recognition* and *Named Entity Recognition (NER)*. Solving these two tasks allows the system to create the basics to implement a conversational agent. How this is possible will be explained during this chapter. Another thing that we have to point out, is that this architecture aims to be as scalable as it is possible, this means that instead to be a simple implementation of a conversational agent it gives the user (any customer) the possibility to create a conversational agent that will answer only to the kinds of questions that he is interested in. Doing this means adopting a customer-driven approach to the problem, so potentially each customer could use this conversational agent system to fulfill its purposes. The system can be divided into five micro-services:

1. **Prediction module** a Python module that is capable to take an input sentence and output the predicted intent from the Intent Recognition model and the categories predicted from the Named Entity Recognition model

2. **Training module** a Python module that is in charge to train a new model for the Intent Recognition task starting from the example questions that the customer will choose

3. **Coordinator backend** a Javascript module that has the role of orchestrating all the processes in which the system is involved. It has a key role both in the training phase and in the prediction phase. It has also the role to communicate with a relational database in which the examples data that the customer has submitted is stored

4. **Administration panel** this is the administrators' frontend, in which it is possible to create a new model, select the model to use in production, and test the existent models through a playground.

5. **Chat widget** this is a frontend that will be embedded in the customer's website and from which the potential users will chat with the conversational agent. This module will not be treated in the scope of this thesis.

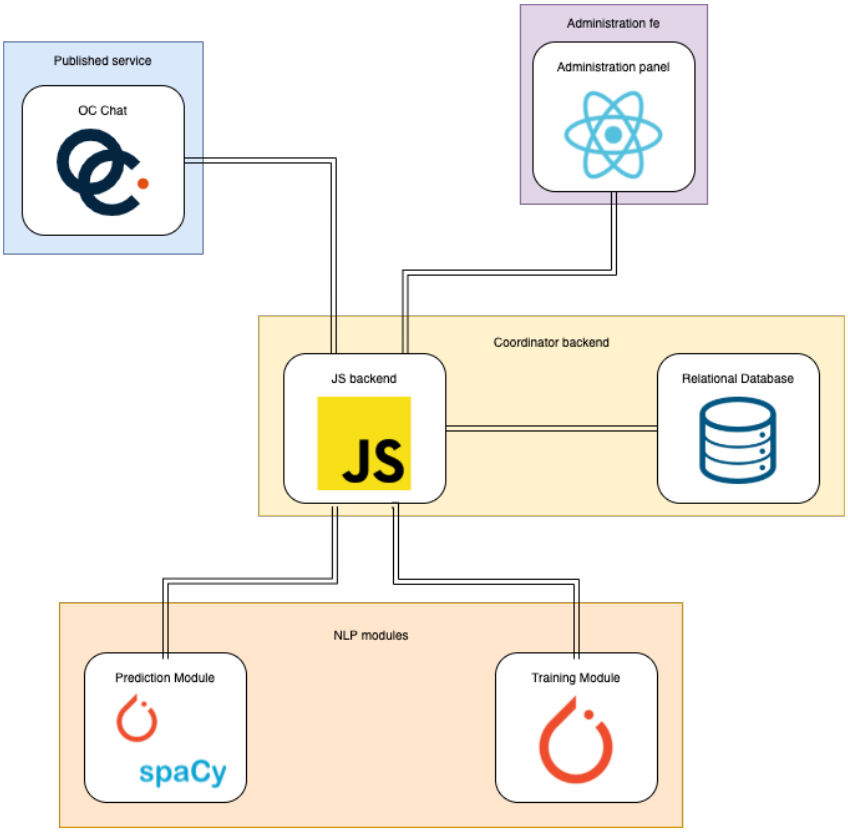


Figure 3.1. High level abstraction of the system's architecture

3.1 NLP services

As we said before the Natural Language Processing micro-services are the core of this project, for obvious reasons. So it is particularly important to analyze their structure and the way in which they were designed. These two modules are intended to give a complete implementation of more than one machine learning model with the possibility to train them and make them do predictions. To be more precise the training module has the assignment to train only the Intent Recognition Model, which was implemented from scratch, while The Named Entity recognition part is implemented with a library called *SpaCy*[1]. This aspect will be explored in section 3.1.3.

3.1.1 Intent Recognition Model

Now let's focus on the implementation of the Intent Recognition Model. The first thing that is quite straightforward is that this is a classification model, so this means that this has to be a neural network that in input receives a tokenized sentence of length l and in the output, it has to return the label of the class to which the input sentence belongs. From chapter 2 we learned that the state-of-the-art way to implement this task is to use BERT or a similar model. In this case, after some tests on various models (see chapter 5) RoBERTa, a slightly different version of BERT, proved to be a good solution to implement this task. In particular, the

[1] https://spacy.io/

implemented solution comes from *Huggingface*[2], a very famous library that offers the implementation of different pre-trained models for NLP. Before going on it is useful to represent the model with an image:

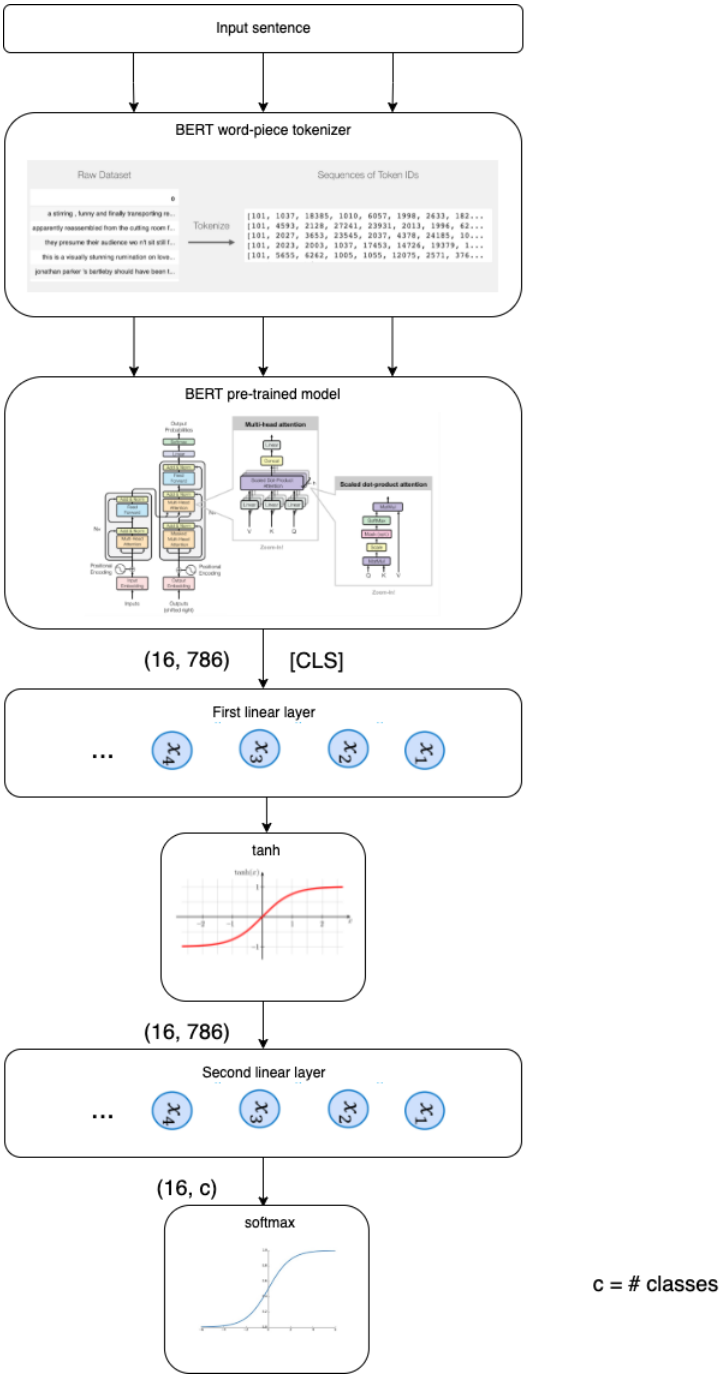


Figure 3.2. High level abstraction of Intent Recognition Model

We can see how the original sentence is given to the word-piece tokenizer that produces the input for the pre-trained BERT model to do its computations. But

[2]https://huggingface.co/

BERT's output does not directly suit our purposes, in fact, BERT's output with input with dimensions (b, l, e), where b is the batch size and e is the word-embeddings size, is (b, l, h) where h is the hidden size of BERT, so 768. So we need a way to go from the output tensor of BERT to a number that represents the class of belonging of the input sentences. This is done by the classification head, which is a simple feed-forward neural network that contains two linear layers: The first one takes in input the [CLS] output token from BERT like is recommended in section 2.8.2 and outputs a tensor with the same dimensions. So if we consider 16 as our batch size, it goes from (16, 768) to (16, 768). The second step of this Neural Network is to apply a *tanh* activation function, and the output of this operation is passed to a second linear layer, that goes from (16, 768) to (16, c), where c is the length of the predefined classes list. The last step of this model is to apply the *softmax* activation function, which is justified by the fact that Intent Recognition is a multi-class classification task. Now to get a prediction from the model we have to apply the *argmax* to its output so that we get the index of the predicted class.

```
   | Name          | Type             | Params
-------------------------------------------------
0  | model         | BartModel        | 139 M
1  | lin1          | Linear           | 590 K
2  | lin2          | Linear           | 16.9 K
3  | dropout       | Dropout          | 0
4  | tanh          | Tanh             | 0
5  | softmax       | Softmax          | 0
6  | loss_function | CrossEntropyLoss | 0
7  | train_acc     | Accuracy         | 0
-------------------------------------------------
140 M     Trainable params
0         Non-trainable params
```

Figure 3.3. Model summary that shows the quantity of trainable parameters

3.1.2 Training module

The first module of the architecture that makes use of this Machine Learning model is the *Training module*. This is a micro-service that exposes a REST interface and it has the task to train a model out of the examples that the customers supplied. There is a big consideration to do here: we want that our model learns from example data that the customer inputs by hand, but we do not want the customer to insert thousands of examples in the system, because this could become time-consuming and difficult to do more than once. The proposed solution here is that before training the model with the few data that the customer has inserted, we augment this data to obtain a dataset that is built in a semi-automatic way. We augment the sentences using a python library called *nlpaug*[3], and we use four techniques to augment the data:

- **Insert word augmentation**: from the input sentence, we are capable to generate another sentence with an inserted word using BERT.
- **Substitute word augmentation**: We apply the same technique as before, but instead of inserting a new word we substitute an existing one

[3]https://github.com/makcedward/nlpaug

- **Back translation augmentation**: We use a machine translation model to translate the sentence from a starting language to a second language, and then we do the inverse operation. In this way, the goal is to obtain a slightly different sentence that still has the same meaning as the input sentence
- **Add typos augmentation**: This kind of augmentation aims to get the final model more robust against typos char swaps and human mistakes

Once the data is set, the module can start the training phase. During the training, it saves the best model in terms of validation f1_score locally and will expose it to the prediction model after the training is over.

3.1.3 Prediction module

The other service that makes use of the Intent Recognition model is the *Prediction module*. This service is the core of the project, in fact, it allows the whole system to interact with the user making predictions on the sentence that was typed by the user. Not only it can give predictions on a sentence's intent, but it implements a mechanism of named entity recognition that is built on top of the named entity recognition library called *SpaCy*. It is capable to recognize five categories of words in a sentence: NUM, MISC, PER, LOC, ORG. The prediction module, to interact with the other nodes of the architecture, exposes a REST interface, with 3 main endpoints: the first one is to select the model to use in production with *OC-Chat* within the available ones, the second one allows to make a prediction on the intent of a given sentence, the third one is to tag the input sentence with the categories from the Named Entity Recognition task. There is also a fourth endpoint that implements the same logic as the first one, but this is to select the model to use in the playground that is offered from the system to try a model before deploying it in production.

3.2 Coordinator backend

The coordinator backend is the service that solves many tasks that allow the system to work correctly. Let's analyze them in more detail. The first fundamental function that it covers is to communicate with a relational database that is responsible to store all the information that the system needs. A representation of the DB is provided through this entity-relation schema that helps to clarify the ideas about its structure and gives a glimpse of the other functions that the Coordinator backend accomplishes:

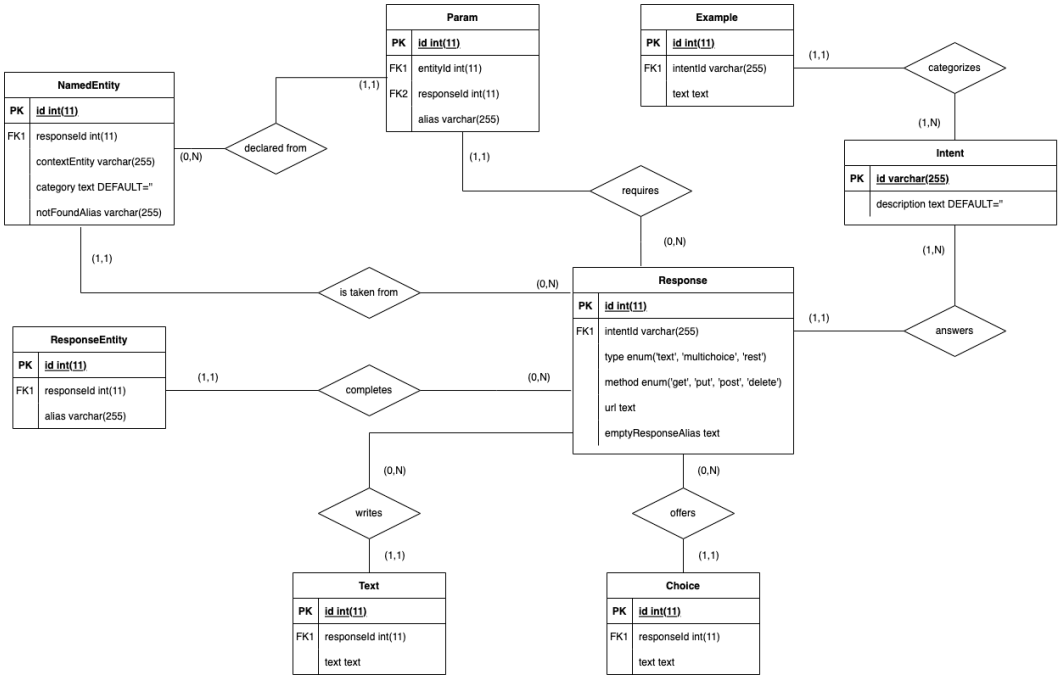


Figure 3.4. Entity-relation schema of the database

The structure of the database was built automatically with a Javascript implementation of the *Object Relational Mappings (ORMs)* called *TypeORM*[4]. Once the connection with the DB is established, the coordinator backend has other two functions:

- Make a training start with data from the database
- Build text responses on top of the data from DB and data from predictions by the prediction module

This service exposes a REST API that allows a user to have access to all these functions from the *Administration panel*. For the first task, the flow is quite simple, in fact, the Coordinator backend sends a POST to the Training module, that gets all the examples with their relative intents from the JS service and starts the procedures of data augmentation and training. While for the second task, the story is more complex. The coordinator backend maintains a context for each active chat in which with a key-value pairing, it puts all the entities that the Named Entity Recognition model from the prediction model has outputted for each sentence of the chat. We know from the entity-relation schema that each intent has associated a response. A response can be of three types: *Text*, *Rest*, and *Multi-choice*. For all kinds of responses, it is implemented a mechanism that substitutes special tokens, that are wrapped by <> with the corresponding entity in the context. Let's take an example to clarify this: suppose that the active chat's context is something like *"person": "Nicolò Palmiero"*, now the outputted answer from the JS backend is "Hello <person>, my name is OC-Chatbot", so the output of the described process will be *"Hello Nicolò Palmiero, my name is OC-Chatbot"*. But how the chat's context is created and maintained? This is all thanks to the Named Entity recognition task,

[4]https://typeorm.io/#/

with some added logic on the JS service. When a chat starts the chat's context is obviously empty, when the first message by the user is emitted the prediction module is asked to extract all the named entities from the new sentence. Once we have the result of this computation, we can now go on to substitute the existing keys in the context chat with the newer ones. In this way, the conversational agent can maintain and update the state of the conversation. The last thing we need to know is the difference between the three kinds of responses. The text response is only a direct text response that can be manipulated with the mechanism described above. The REST response is thought to answer questions whose response does change depending on the user that is asking or it simply changes over time. All that this kind of answer needs is a server that is built from the customer, that gives all the information that it needs through a REST API. For example, if a user has to ask a personal question to an operator, but the answer is still very common upon the requests that users make, the REST response could be a good way to implement automation of this. The multichoice response is useful when the chatbot has to prompt the user for a limited number of choices, in this case, the user is forced to choose between the alternatives that the chatbot already knows in advance, giving it an advantage on the decision of the next action to do. The coordinator backend offers another minor function: to create and maintain a web socket connection with the administration panel to implement a playground chat, in which it is possible to try the available trained models.

3.3 Administration panel

The last part of the architecture that we have to explore is the *Administration panel*, which together with the *OC-Chat*, build up the frontend part for our service. So, just why this is a frontend service there will be some screenshots to explain the behavior of the system.

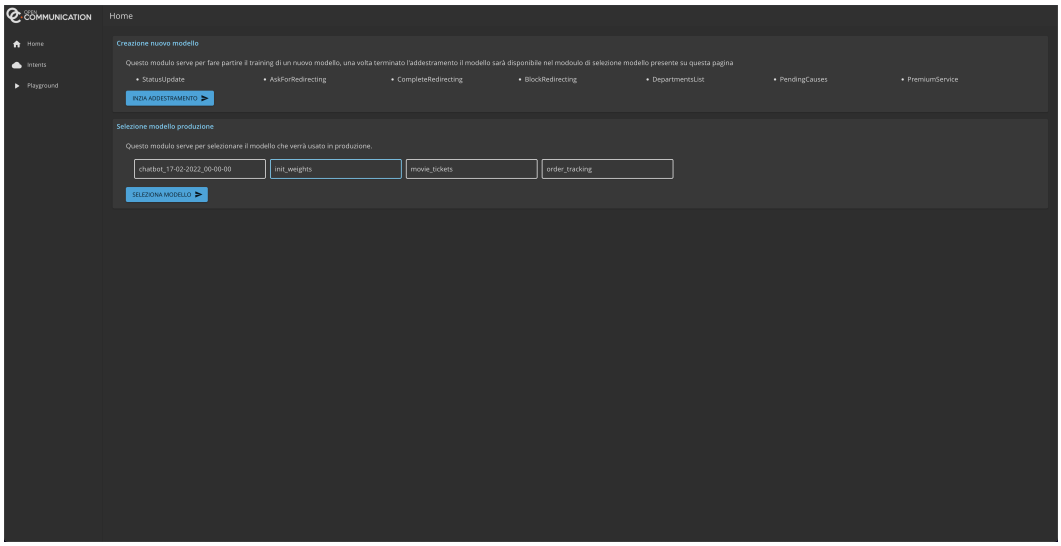


Figure 3.5. Screenshot of the homepage of the Administration frontend

This is the homepage, what the administrator can do from this page is to select the intent recognition model that is actually in production and answers to the customers' requests, and he can start training with the currently selected intents to

create a new model. This brings us to the explanation of the second page:

Figure 3.6. Screenshot of the intents page of the Administration frontend

This page is the one in which the administrator can handle the kind of model he wants to create, creating editing, or removing an intent. In the table, all the intents that are currently configured are listed, and from here it is possible to access the detail page of each intent

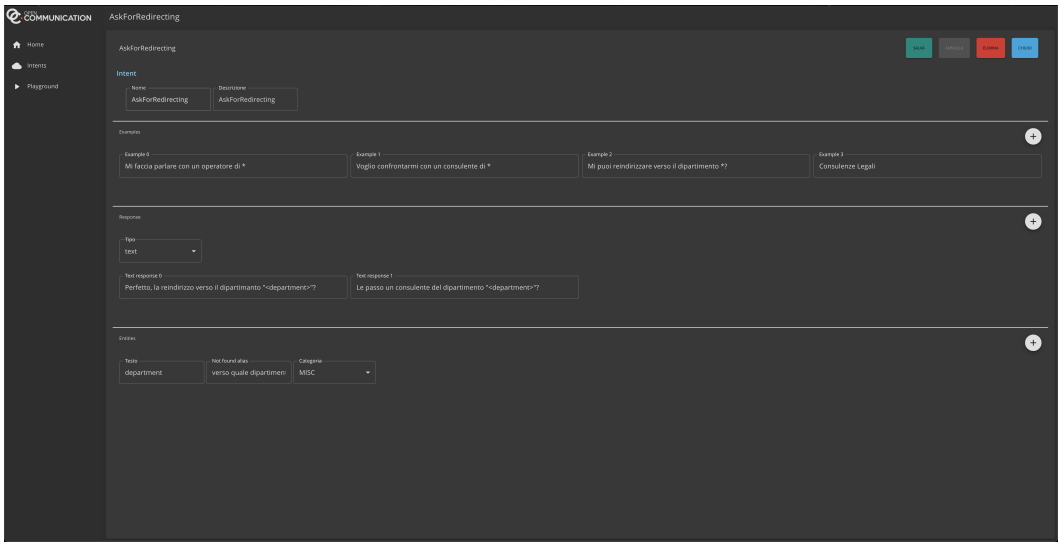


Figure 3.7. Screenshot of an intent detail of the Administration frontend

Here there is a form that allows the user to access all the fields of the database. The first row aims to simply give the intent a name. The second row is a list of all the examples that are associated with the intent, so the ones that will be fed to the training module to start the training of a new model. The third row allows the administrator to choose the kind of answer that is associated with the intent. In the case of the image it is a text response, and below are listed the possible text responses that will be picked at random from the conversational agent. This form

changes in base to the kind of response that the administrator picks, in fact, the REST response has different associations than a text response, and this reflects also on the frontend.

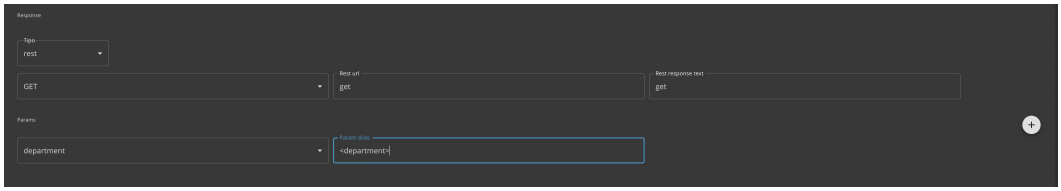


Figure 3.8. Detail of a rest response

The fourth row is the response entities part, in which an association of a recognized entity to the context of the chat can be created. To use the content of the variable created in the context with this association in response, the administrator has to wrap the variable name like this *<department>* in the text of the response. The last section of this frontend is the playground, in which, it is possible to try a model asking it something just like a real customer could do.

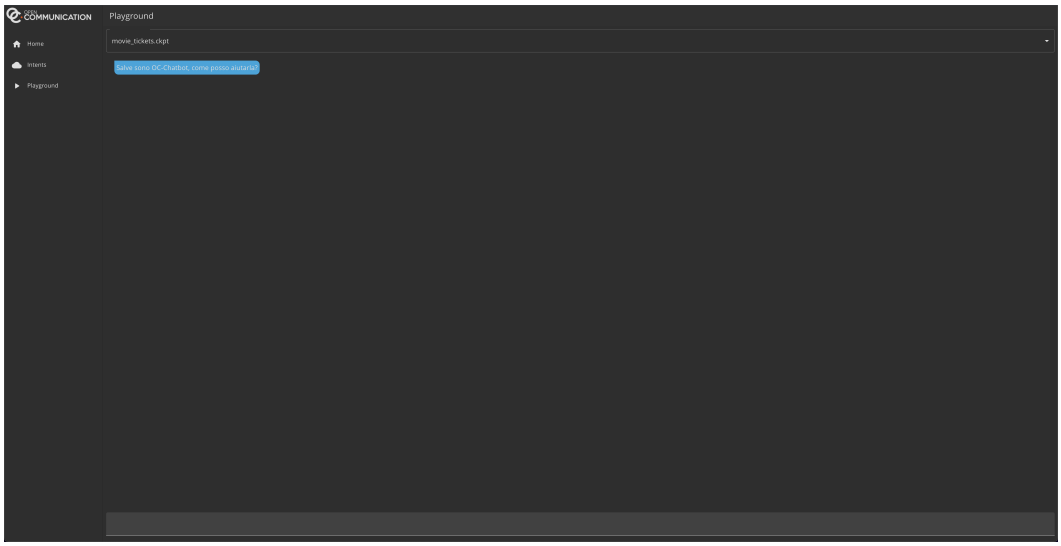


Figure 3.9. Screenshot of the playground section of the administration panel

This is very important because this creates the possibility to test a model after it is created, but before deploying it to the production phase. So this gives the possibility to improve the model, for example, if an administrator realizes that the conversational agent's answers are not particularly effective if the question is asked in a particular way, he can put new examples in the training set and start a new training creating like this an improved version of the model. If this process is repeated the conversational agent can arrive at a very good understanding of the customer's requests.

Chapter 4

Use cases

Now it is the case the start talking about some situations in which the conversational agent could be useful and could help to lower the load of requests that are submitted to operators. There are various fields of interest in which a conversational agent can be a powerful solution, for example, contact centers are one of the most common environments in which it is employed. In this chapter three use cases are proposed, creating a flow diagram to idealize and design the aimed flow of a conversation, and then trying to reply to it using the playground on the administration panel. These three use cases are:

- **Movie tickets booking**
- **Order tracking**
- **Legal assistance contact center**, which comes from a real customer in the company where I work

Before starting to see the details it is important to underline one thing: all the use cases are in Italian, this is because all the customers of the company where I work are Italians, but if in the future there will be the need to have an English model, this thanks to *Huggingface* is absolutely possible and very easy to do. All these use cases are evaluated by giving four examples for each intent to the training module to create the model. This is important because due to the fact that the architecture allows the improvement of the model by submitting more examples for each intent, it was thought that without fixing a limit for the number of intents this task could be distorted. The requests coming from rest responses for the goals of the examples are handled simulated a real server from the customer company with a *JSON-server*[1].

4.1 Movie tickets booking

In this use case, it is possible to start a conversation with a chatbot that is thought to give information and to allow the user to book a ticket for a movie. The designed intents for this use case are:

- **Reservation** the user expresses the intent to book a ticket
- **TimeInput** expresses the time in which the user wants to watch the movie

[1] https://github.com/typicode/json-server

- **FilmList** the chatbot will give the user the list of the films in the program
- **OpeningTimes** the user wants to know the opening times of the cinema
- **FilmScheduling** the chatbot tells the user the schedule of the programmed movies
- **TicketCosts** the user wants to know the cost of the tickets
- **TalkToOperator** the user is not satisfied with the chatbot's answers and wants to talk with an operator
- **FilmDuration** the user wants to know the duration of a film
- **PurchaseMethods** lists all the modalities to purchase a ticket

Once we have seen the possible intents that the conversational agent can recognize, we move forward to see one typical flow of a conversation in this context:

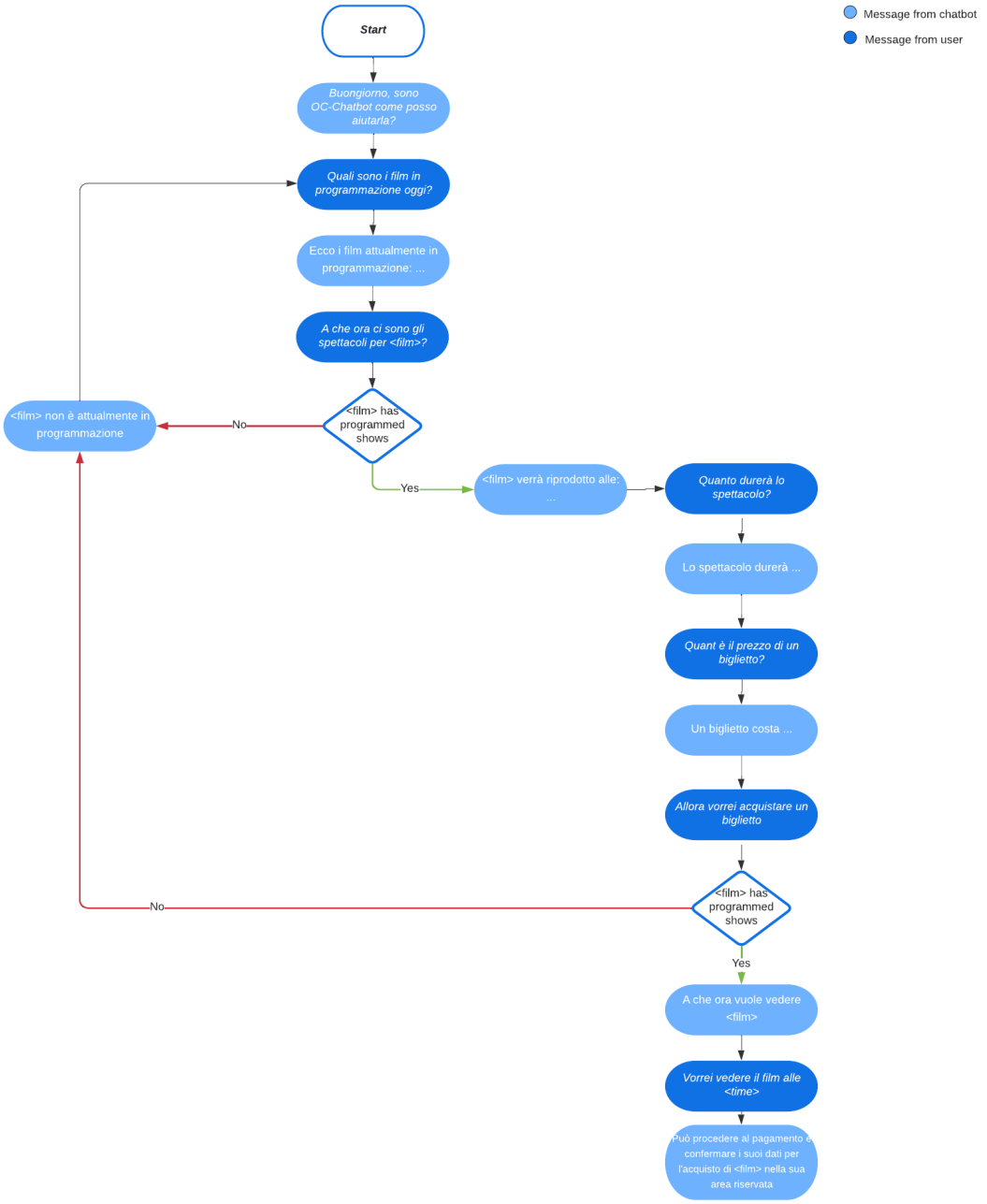


Figure 4.1. Flow diagram of an example conversation for ticket booking

now that we have seen this flow, we try to reproduce it in the playground with an ad-hoc trained model.

We can see that the accuracy, in this case, is quite high, the conversational agent maintains correctly the context remembering that the film that the user wants to see is The Batman. There is to underline the fact that in this example there is a misclassification at the moment in which the user asks to book the ticket. This is due to the fact that the intents *PurchaseMethods* and *Reservation* have some terms in common. The solutions to this error are two: the first is to add more examples of a reservation request, the second is to balance the number of examples on the number

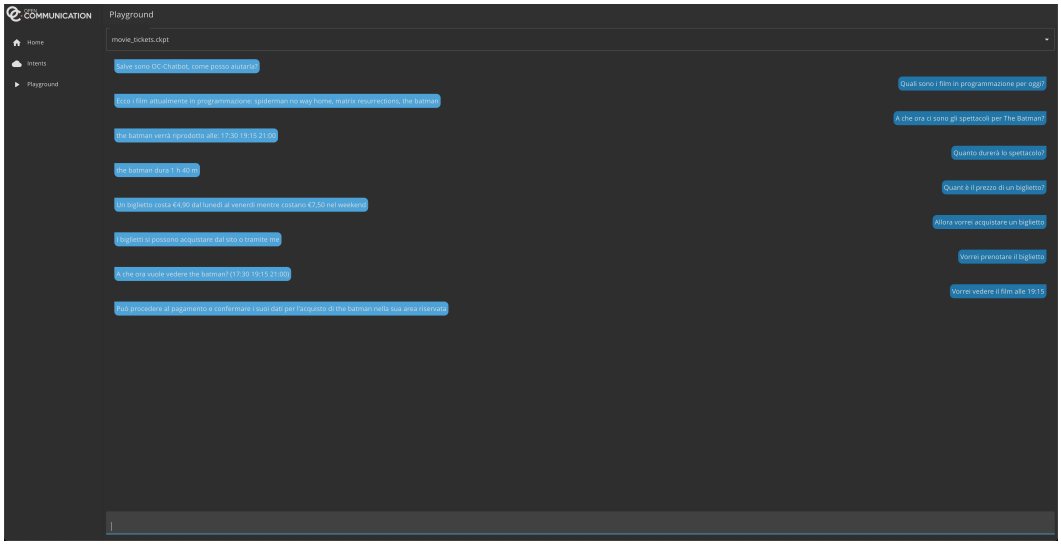


Figure 4.2. Real conversation reproducing the example for ticket booking

of times each intent is required. In this case just because this is a conversational agent whose goal is to guide the user to the booking of a ticket, the *Reservation* examples must have a bigger cardinality than the other intents. This shows in a good way the potentialities of the system, that were described in chapter 3.

4.2 Order tracking

The second use case that is analyzed is a situation which many delivery companies in the world have to handle: the tracking of their shippings. The designed intents for this task are:

- **MyOrders** the user wants to know what are his pending orders
- **TrackOrder** the user wants to track one of his pending orders
- **DeliveryPlaces** the chatbot will give the user the list of the delivery places in which the company ships
- **StatusesMeaning** the user wants to know the meaning of the name of the status of shipping
- **CancelOrder** the user wants to cancel an order
- **Confirmation** the user confirms that he wants to cancel the order
- **Negation** the user does not confirm that he wants to cancel the order
- **TalkToOperator** the user is not satisfied with the chatbot's answers and wants to talk with an operator

and here there is an example of a conversation in this context

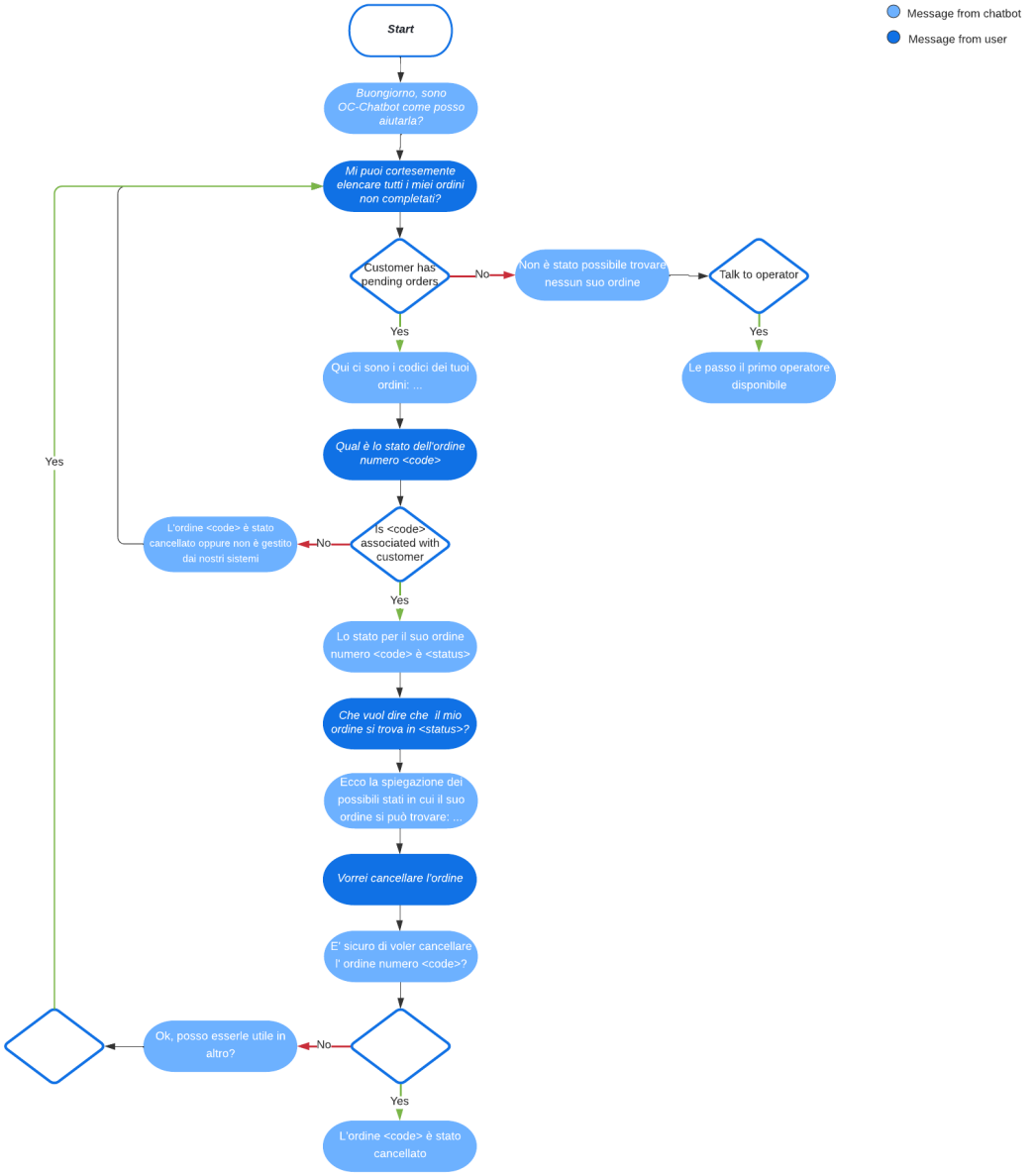


Figure 4.3. Flow diagram of an example conversation for order tracking

Also here the conversation requires that the conversational agent memorizes some information from the questions of the customer.

Figure 4.4. Real conversation reproducing the example for ticket booking

The conversation this time proceeded without big obstacles, but obviously, there will be cases in which the chatbot misunderstands the correct intent, another time it is fundamental to use the playground to prevent the biggest part of the errors before deploying the model in production.

4.3 Legal assistance contact center

This use case comes from one of our customers, that had the need to use a chatbot to answer their most frequent requests or to redirect the customer to the right department to which ask, now these tasks are assigned to a specific department. So the intents for this use case are:

- **PremiumService** the user wants to know if he has subscribed to the premium service, and in the positive case when the service will expire
- **StatusUpdate** the user wants a piece of information about the state in which one of his pending causes is
- **DepartmentsList** the chatbot will give the user the list of all the departments with which it is possible to talk
- **BlockRedirecting** the user does not give the consent to be redirected to another department
- **CompleteRedirecting** the user gives the consent to be redirected to another department
- **AskForRedirecting** the user is asked if he wants to continue to be redirected to another department
- **PendingCauses** the conversational agent gives the user the list of all his pending lawsuits.

And the flow of a conversation could be like this:

Figure 4.5. Flow diagram of an example conversation for legal assistance contact center

The real counterpart for this flow diagram could be split into two cases because it develops more in breadth than the other two examples

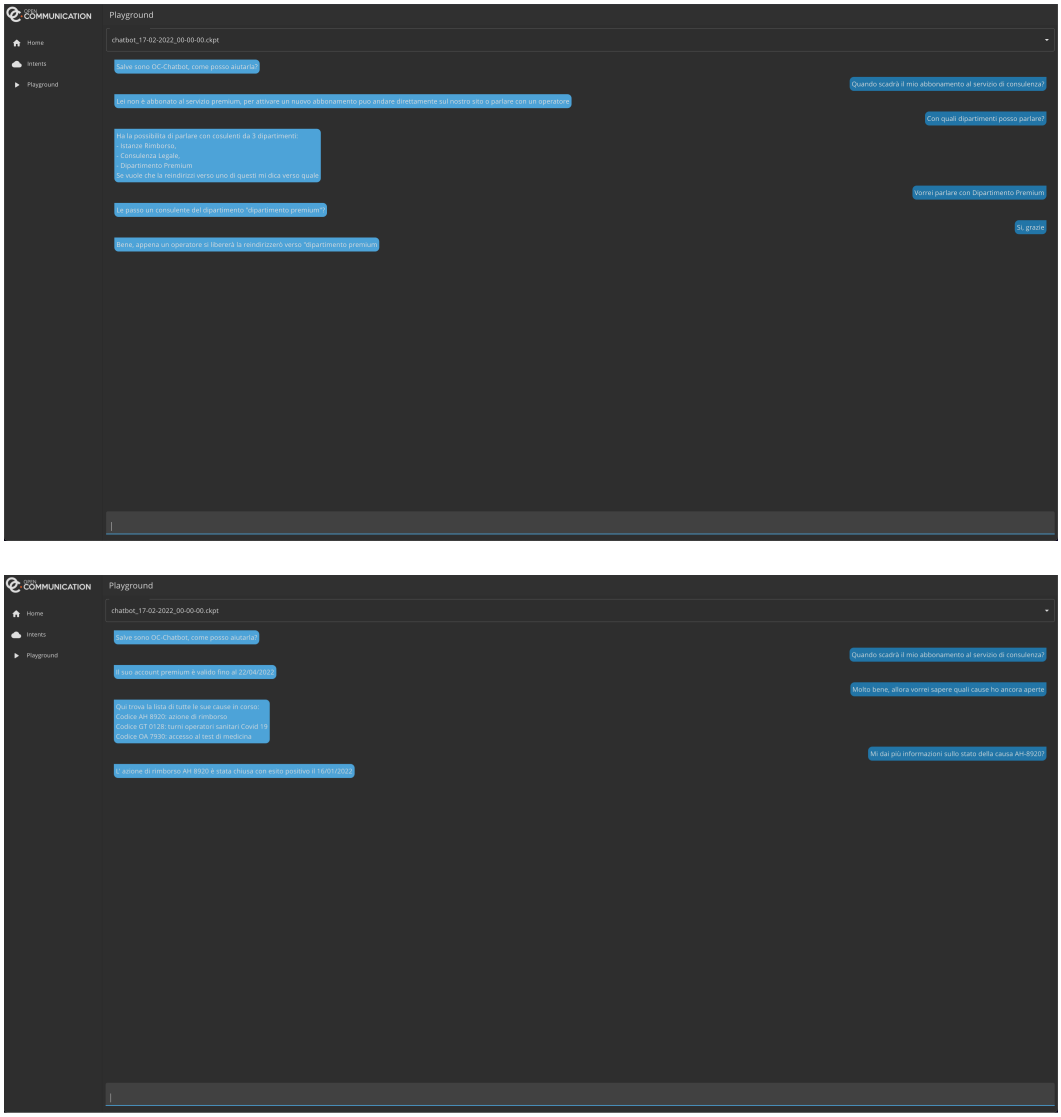


Figure 4.6. Two examples of conversations from the flow diagram

The integration between the conversational agent and the customer's systems is at an initial stage, in the future there will be a study that aims to integrate more functionalities of the chatbot in a way that will lower the load of requests to the operators with an increasing impact. But this integration is not only hindered from a technological point of view but mainly from the point of view of privacy and data protection. In fact, the main operation that will be done in this study is to point out all the tasks that are enough frequent to allow the chatbot to handle them, and that do not treat data that is confidential and has to be handled directly from an operator.

Chapter 5

Evaluation

A quantitative evaluation of a conversational agent is not something easy to put into practice, and there is not yet a precise standard to follow to make a complete evaluation of its performance. This is true because they are almost certainly composed of more modules that interact with each other, so it is useful for sure to evaluate them one by one as if they were isolated, but this is not a complete evaluation of a conversational agent because these results will not guarantee that it will work optimally when it will be deployed in production. So, in addition, to carry out the evaluation of each chunk of the architecture it is useful to use a method that allows to point out the effectiveness of the conversational agent as its whole. Preceding step by step we will start with the evaluation of each part of the Natural Language Processing services.

5.1 Intent Recognition evaluation

The *Intent recognition* task is the heart of the conversational agent and this technique must work as expected. For the evaluation of this module, a small dataset from *Kaggle*[1] was picked, which is very good to simulate the most common situation in which the conversational agent will work, because it is a dataset that has 144 examples, a little number, and in perspective a big number of intents, 22. To conduct the evaluation, the dataset was augmented in this way: it was first augmented with *Insert word augmentation*, *Substitute word augmentation* and *Back translation augmentation*, then it was split into training, validation, and test splits, and only after typos were added to the training set. This was made to add some noise during the training phase, but it is not useful to add typos also in the evaluation part, because this would increase the similarity between training data and evaluation data, and this could in some sense simplify the work of the classifier in the evaluation phase, giving boosted results. Here is a bar plot representing the balancing of the training set after the augmentation phase

[1] https://www.kaggle.com/elvinagammed/chatbots-intent-recognition-dataset

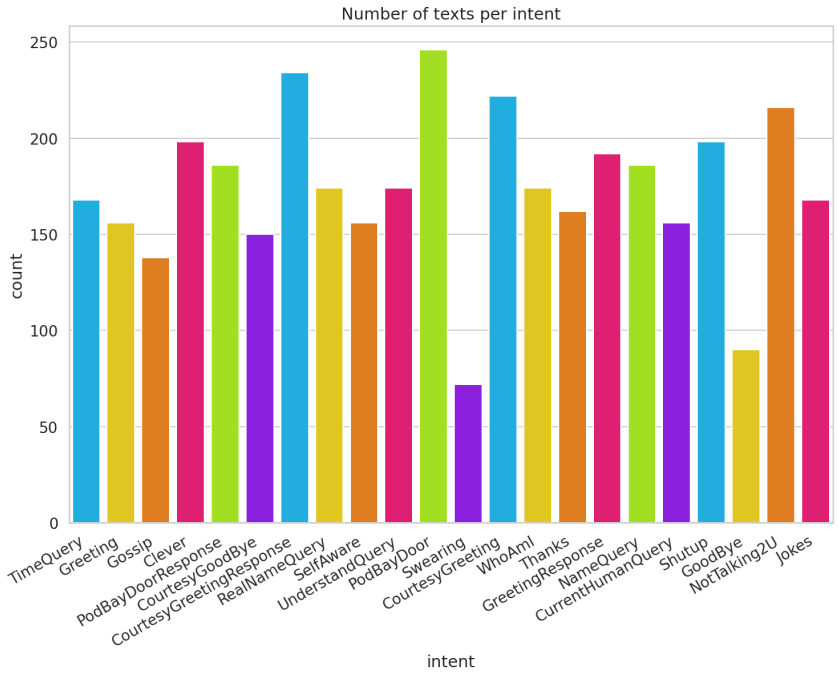


Figure 5.1. Balancing of the dataset after the augmentation

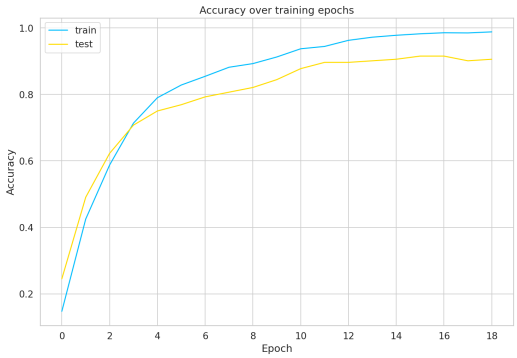
This evaluation phase is fundamental to have some criteria to choose which architecture of the model is worth to be used as the final configuration of the *Intent recognition model*. About this it was have decided to evaluate three different models on this Kaggle dataset and pick the one that had the best performance in terms of *accuracy*, *precision*, *recall* and *f1_score*. These three models are all transformers architectures:

- **BERT**
- **RoBERTa**, a modified version of BERT in handling the Masked Language Model, that still is the core of the training of RoBERTa [7]
- **BART**, it is a generalization of BERT, adding noising functions like *Token Masking, Token Deletion, Token Infilling, Sentence Shuffling*, and *Document Rotation* to prepare the encoder's noisy input during training phase [6]

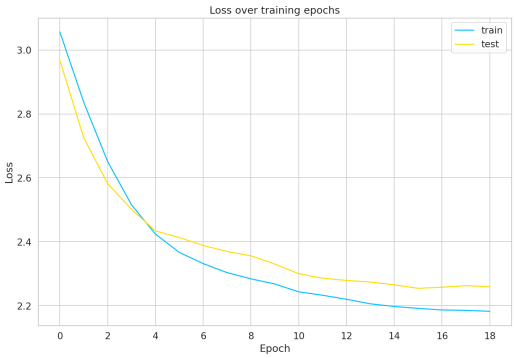
The classification head is left as it is described in chapter 3 in all the three model configurations.

5.1.1 Models comparison

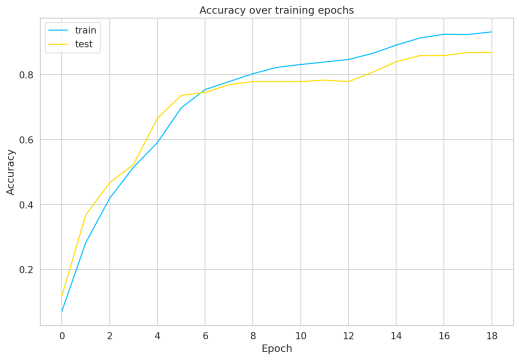
The comparison of the models is brought using both plots and metrics that allow to catch differences between them, this is important because as it is intuitable, results from these tests are very similar between each other, because the philosophy behind the three models is the same, but they differ in details aiming at gaining some little boosts in performances. The first plots that we analyze are the training loss/accuracy plots, which allow us to understand the behavior of a model during training, and to point out the point in which a model overfits. All the training sessions lasted for 20 epochs, at a learning rate of 0.00001 and with a *Categorical CrossEntropy* loss function



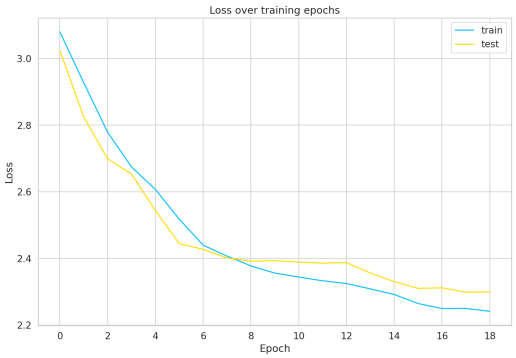
(a) BERT accuracy plot



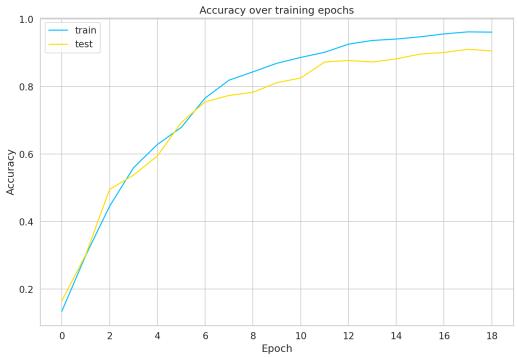
(b) BERT loss plot



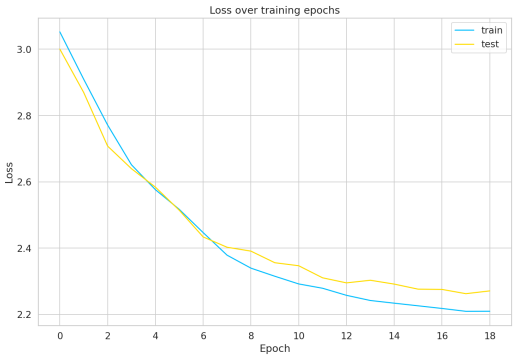
(c) BART accuracy plot



(d) BART loss plot



(e) RoBERTa accuracy plot



(f) RoBERTa loss plot

Figure 5.2. Training accuracy/loss plots

These plots are very similar for all the three models, in fact in all these plots we can see that there is no proper overfitting, but the training accuracy converges in a number that is very close to 1, this means that the model cannot learn any more from the current training set, and it is worth to stop the training to prevent unexpected behaviors. One more clarification, the model that is saved at the end of the training is not the last epoch's one, but it is the one that outperformed the others during the training phase on the validation set This technique is called model checkpointing. We need other data to pick the best model between the three because basing our choice only on these plots the differences are too few. So now we pass to analyzing classification reports made evaluating the models on the test set. It is important to have this split and to use only to evaluate the performances of the model, because the results are close to real performances only if the model

is evaluated on data that it has never seen before, both in the training and in the validation phase. The classification reports give us information about the reached precision, recall f1_score, and accuracy of each class, and macro/micro precision, recall, and f1 giving us also information on the support for each entry of the table.

	precision	recall	f1-score	support
Clever	0.89	1.00	0.94	8
CourtesyGoodBye	0.62	1.00	0.76	13
CourtesyGreeting	0.88	1.00	0.93	7
CourtesyGreetingResponse	0.92	0.92	0.92	13
CurrentHumanQuery	0.92	1.00	0.96	11
GoodBye	0.83	0.83	0.83	6
Gossip	0.85	1.00	0.92	11
Greeting	0.90	1.00	0.95	9
GreetingResponse	1.00	1.00	1.00	18
Jokes	1.00	0.56	0.71	9
NameQuery	0.75	0.75	0.75	4
NotTalking2U	1.00	1.00	1.00	14
PodBayDoor	1.00	1.00	1.00	8
PodBayDoorResponse	1.00	1.00	1.00	10
RealNameQuery	1.00	0.86	0.92	7
SelfAware	1.00	1.00	1.00	12
Shutup	0.90	0.82	0.86	11
Swearing	1.00	0.88	0.93	8
Thanks	1.00	0.22	0.36	9
TimeQuery	1.00	0.83	0.91	12
UnderstandQuery	0.82	1.00	0.90	9
WhoAmI	1.00	1.00	1.00	3
accuracy			0.91	212
macro avg	0.92	0.89	0.89	212
weighted avg	0.92	0.91	0.90	212

(a) BERT classification report

	precision	recall	f1-score	support
Clever	0.80	1.00	0.89	8
CourtesyGoodBye	0.57	0.92	0.71	13
CourtesyGreeting	1.00	0.86	0.92	7
CourtesyGreetingResponse	0.92	0.92	0.92	13
CurrentHumanQuery	0.90	0.82	0.86	11
GoodBye	0.00	0.00	0.00	6
Gossip	1.00	1.00	1.00	11
Greeting	0.67	0.89	0.76	9
GreetingResponse	0.95	1.00	0.97	18
Jokes	0.89	0.89	0.89	9
NameQuery	0.60	0.75	0.67	4
NotTalking2U	1.00	1.00	1.00	14
PodBayDoor	0.89	1.00	0.94	8
PodBayDoorResponse	0.90	0.90	0.90	10
RealNameQuery	1.00	0.86	0.92	7
SelfAware	1.00	1.00	1.00	12
Shutup	0.73	0.73	0.73	11
Swearing	0.00	0.00	0.00	8
Thanks	0.50	0.44	0.47	9
TimeQuery	1.00	0.83	0.91	12
UnderstandQuery	0.69	1.00	0.82	9
WhoAmI	1.00	1.00	1.00	3
accuracy			0.84	212
macro avg	0.77	0.81	0.79	212
weighted avg	0.80	0.84	0.81	212

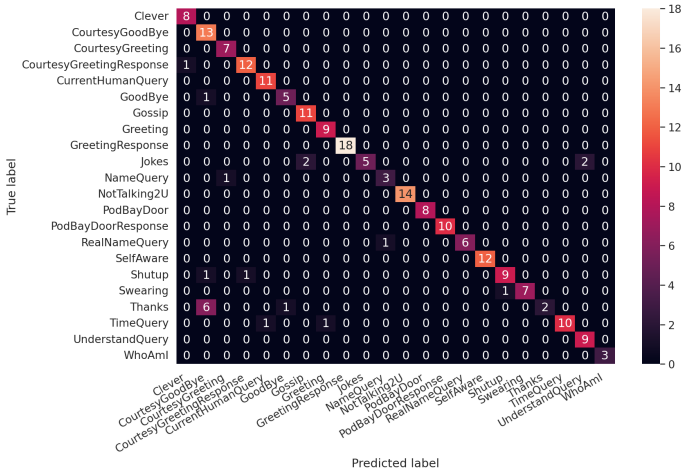
(b) BART classification report

	precision	recall	f1-score	support
Clever	1.00	1.00	1.00	8
CourtesyGoodBye	0.92	0.92	0.92	13
CourtesyGreeting	1.00	1.00	1.00	7
CourtesyGreetingResponse	0.92	0.92	0.92	13
CurrentHumanQuery	1.00	1.00	1.00	11
GoodBye	0.75	1.00	0.86	6
Gossip	1.00	1.00	1.00	11
Greeting	0.90	1.00	0.95	9
GreetingResponse	0.95	1.00	0.97	18
Jokes	1.00	1.00	1.00	9
NameQuery	1.00	1.00	1.00	4
NotTalking2U	1.00	1.00	1.00	14
PodBayDoor	1.00	1.00	1.00	8
PodBayDoorResponse	1.00	1.00	1.00	10
RealNameQuery	1.00	1.00	1.00	7
SelfAware	1.00	1.00	1.00	12
Shutup	0.82	0.82	0.82	11
Swearing	0.00	0.00	0.00	8
Thanks	0.58	0.78	0.67	9
TimeQuery	1.00	0.92	0.96	12
UnderstandQuery	0.82	1.00	0.90	9
WhoAmI	1.00	1.00	1.00	3
accuracy			0.93	212
macro avg	0.89	0.93	0.91	212
weighted avg	0.90	0.93	0.91	212

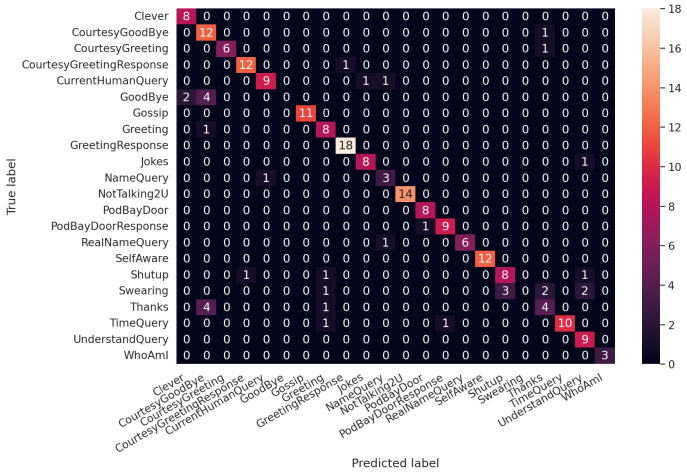
(c) RoBERTa classification report

Figure 5.3. Classification reports from the three models

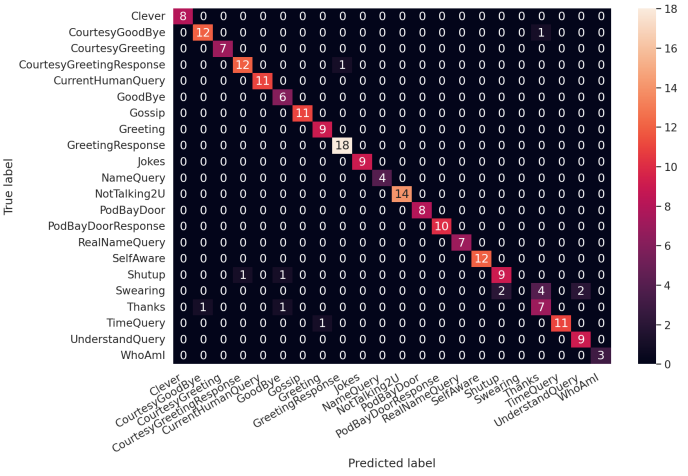
This is the detail of each classification report, in which we can appreciate some differences between the three models. Among these three RoBERTa is the model that scored the best, and BART was the one that was the worst. In particular, RoBERTa outperforms BERT in *TimeQuery* and *Jokes*, and this gave it a boost of two points in micro-f1_score. But the best news is that RoBERTa was capable to obtain an f1_score bigger than 67% in all classes, and if we do not consider *Thanks* intent this threshold increases to 82%, that is a big result. While the other two models had classes in which there were some errors, in particular, the lowest f1_score set from BERT and BART was respectively 36% and 47%, but BERT in the average behaved better than BART, so it is more appealing than it. To confirm this data there are confusion matrices, that point out the exact number of correct and not correct classifications for each class.



(a) BERT confusion matrix



(b) BART confusion matrix



(c) RoBERTa confusion matrix

Figure 5.4. Confusion matrices from the three models

5.2 Named Entity Recognition evaluation

In this section, the results of the evaluation of the Named Entity Recognition task will be shown. At the actual state, the NER task is implemented through SpaCy, so it is not useful to give you other data than the one that is reported on the official source, but for the seek of completeness this data is reported here. The only thing that is worth pointing out is a justification of the selected Italian model between the three that were available:

1. **it_core_news_sm**
2. **it_core_news_md**
3. **it_core_news_lg**

All the three models were trained on sentences from Italian news, so the only difference between the three is the number of parameters of each model. Now there is obviously an advantage to using more parameters (trainable and not trainable) to improve performances, but there is a trade-off between accuracy and performance, in fact, the largest model will be slightly slower than the small one. Here is data from the official site on the NER task of all three models:

	it_core_news_sm	it_core_news_md	it_core_news_lg
Precision	0.86	**0.88**	**0.88**
Recall	0.85	0.87	**0.88**
F1	0.85	0.87	**0.88**

Table 5.1. Results of the evaluation of SpaCy models
Source: https://spacy.io/models/it

My choice, considering these data was to use the it_core_news_md model, which has a significant boost from the smaller model but is not far from the performances of the larger model.

5.3 Conversational agent evaluation

The last thing that remains to see is the evaluation of the conversational agent as its whole, in fact, the success of a conversation does not only depend on the accuracy of the Intent Classification task nor the Named Entity Recognition taken alone, although they are at the base of the success of a conversational agent. There are other factors that influence the flow of the conversation. In our case two of them could be for sure the choice of the intents and the examples with which the chatbot was trained, and how smart the Coordinator backend is in maintaining an active context of the current chat. In the last years, there was not a standardized way to conduct this evaluation and there were no common metrics shared with everyone to make precise comparisons. There are a bunch of papers on the web that point out this problem and try to create a standard that is accepted by everyone. One of these is *"Trends & Methods in Chatbot Evaluation"* [2], it makes an analysis on the used evaluation methods from 2016 to 2020 and identified three main areas in which a conversational agent can be evaluated: *Effectiveness, Efficiency, Satisfaction.* Another interesting paper in this field is *"Evaluating Quality of Chatbots and Intelligent Conversational Agents"* [8] that proposes an evaluation method based on different hierarchical quality

assessments that can be done when evaluating a conversational agent. But still, considering the resources that we have available, the customer's needs, and the fact that what we are building is a goal-based conversational agent, it was decided to evaluate the conversational agent, for each of the three use cases in chapter 4, using two metrics:

- **Goal Completion Rate (GCR)**: it is the percentage of the attempted conversations in which the user has reached his goal
- **Average Response Time (ART)**: it is the average time, measured in seconds, in which the system answers a user's request

The conversational agent has been given to three potential users to which was asked to have 10 conversations focusing on one use case each, trying to never ask questions in the same way and to consider a conversation not successful if the conversational agent gave an unexpected answer for two times during the conversation. This is a summary of what this analysis has produced:

	GCR	ART
Movie tickets booking	0,7	3.653
Order Tracking	0.8	3.637
Legal assistance contact center	0.7	3.649
Average	0.73	3.650

Table 5.2. Results of the evaluation of the use cases scenarios

The final row is the average of all the results obtained from each conversation, and not the average computed on the rows above. The average response times are the ones that we expect from an architecture that runs on a simple CPU-equipped machine, if the Prediction Module runs on a machine with a GPU, these times can be lowered. This is due to the fact that the implementation of the Prediction Module was designed to detect the presence of a GPU, and if it is the case it is capable to move the model and the inputs on the detected device and do the computation there. This can speed up the prediction phase thanks to the big computational power that the GPU has. These results are also boosted by the simplicity of the use cases that were analyzed but they are a starting point to understand that the system is going in the right direction. Once the system will be deployed to the customer it will be possible to make studies on more quality assessments, starting for example from the customers' satisfaction, that is something that is not quantifiable in a significant way if a large number of users using the system is not available, and this will allow refining also the quality assessments that we have already done and improve the system following the customers' directives.

Chapter 6

Future developments and conclusions

This thesis project aimed to create from scratch a conversational agent that could satisfy our customers' needs and at the same time be competitive in terms of pricing with the other alternatives on the market. So the first thing that was made was to evaluate the most famous solutions to build a conversational agent in terms of features, costs, and data treatment, having seen that this last one is a key point for the customers with whom a consultation was made. After this market analysis, were identified the key features that had the priority to be implemented to have a starting point to build a new product and was designed an architecture that allowed us to use servers equipped with a GPU only for that training phase of a model, while for the prediction phase it is recommended but not required to improve the speed with which the system answers a question. Then was started an analysis of three possible scenarios in which the chatbot could be useful, designed some flow diagrams, and tried to reproduce the conversations asking our prepared questions to the conversational agent. In the last point, an evaluation of the product from a quantitative point of view was tried, and what emerged is that there is no one single way to evaluate the performances of a chatbot, the approach of this project was to first evaluate the single components that composed the conversational agent, and then a small test scenario was built up, evaluating the goal completion rate of each conversation, and the average response time of the system.

6.1 Future developments

At the end of this analysis of this conversational agent system, we have seen some features that are solid and others that can be improvable, for example, the customer-driven approach allows every customer to build a conversational agent in a few hours, and the playground system is the tool with which the customer can improve the quality of the model testing it against real sentences and real questions from one user or a pool of users. Starting from this point and having seen what the other conversational agent systems on the market offer, two new ideas to improve the maintainability of the models are given:

- Include an analytics part on the administration panel to monitor the behavior of the model that is deployed in production, some of these statistics could be: *Number of interactions*, *Bounce Rate* that is the number of times in which a user is redirected to an operator, the already cited *Goal Completion Rate*

and the *Average Response Time*. This data is grouped in one view is very helpful to understand if the model is behaving as expected or the users avoid the chatbot because it rarely solves their problems.

- Integrate a flagging tool in the chat frontend to give the possibility to the user to report a bad answer from the chatbot. What the administrator will see in a specific section of the administration panel, is the question that the user asked, surrounded by n messages to give context to the report, and the answer that the chatbot gave, so that the administrator can realize the problem, and try to solve it using the mechanism that is described in this work

Another idea to improve the interactions with users of the conversational agent is to implement confidence intervals in intent classification. This means that the Intent Recognition model, in the prediction phase will output the predicted intent if and only if the maximum of the probability distribution yield from the softmax activation function is bigger than any other value by a quantity greater or equal to the confidence interval. If this does not happen, the model could output a fallback intent like *TalkToOperator* or *DidNotUnderstand*. The last idea for future development is to implement a Named Entity Recognition module from scratch because although the SpaCy models, as we have seen have good evaluation metrics and good performances is limited from the point of view of the number of categories that can be predicted, that is only 4. Instead, if we could train a transformer model Like BERT or RoBERTa to perform Named entity recognition with more categories the overall performances of the conversational agent system would have a great boost in my opinion. What held me back to implement directly a Named Entity Recognition model from scratch is the lack of Italian labeled data, but recently one candidate to use as a training set for this model was discovered, it is the OSCAR dataset[1] that offers, through the other features Italian labeled data to be used to train a NER model.

[1] https://oscar-corpus.com/

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *ArXiv* 1409 (Sept. 2014).

[2] Jacky Casas et al. "Trends & Methods in Chatbot Evaluation". In: Oct. 2020, pp. 280–286. DOI: 10.1145/3395035.3425319.

[3] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[4] Erkam Guresen and Gulgun Kayakutlu. "Definition of artificial neural networks with comparison to other networks". In: *Procedia Computer Science* 3 (2011). World Conference on Information Technology, pp. 426–433. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2010.12.071. URL: https://www.sciencedirect.com/science/article/pii/S1877050910004461.

[5] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[6] Mike Lewis et al. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *CoRR* abs/1910.13461 (2019). arXiv: 1910.13461. URL: http://arxiv.org/abs/1910.13461.

[7] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

[8] Nicole M. Radziwill and Morgan C. Benton. "Evaluating Quality of Chatbots and Intelligent Conversational Agents". In: *CoRR* abs/1704.04579 (2017). arXiv: 1704.04579. URL: http://arxiv.org/abs/1704.04579.

[9] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf.

[10] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.