Master thesis : Design and implementation of a chatbot in the context of customer support

Auteur : Peters, Florian
Promoteur(s) : Wehenkel, Louis
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2017-2018
URI/URL : http://hdl.handle.net/2268.2/4625

University of Liège - Faculty of Applied Sciences

Master thesis

Design and implementation of a chatbot in the context of customer support

Graduation Studies conducted for obtaining the Master's degree in Computer Science and Engineering by Florian Peters

supervised by
Prof. Louis Wehenkel

Academic year 2017-2018

Design and implementation of a chatbot in the context of customer support

Florian Peters

Abstract

Customer support is perhaps one of the main aspects of the user experience for online services. However with the rise of natural language processing techniques, the industry is looking at automated chatbot solutions to provide quality services to an ever growing user base. This thesis presents a practical case study of such chatbot solution for the company GAMING1.

First, an introduction to the market the company operates in is presented as well as a quick review of the field of conversational agents, highlighting the previous and current techniques used to develop chatbots. Then, the theory behind the techniques used is presented. Mainly deep learning techniques such as gated recurrent unit neural networks are discussed.

Afterwards, a checklist of the issues solved by the chatbot is put on paper. Then a scalable software architecture for the chatbot is proposed and explained. A way of extracting ticket data as well as a quick dataset analysis are shown.

A complete analysis of various neural network structures for user intent classification is shown alongside models for requesting a human operator if need be. The gated recurrent units were shown to be the most effective for classification whereas simpler models worked quite well for the human operator requester.

Finally, a summary of performance metrics for the chatbot's various submodules is shown. However since performance metrics are hard to interpret for dialogue systems, a series of practical test cases are presented as they show that the bot's behaviour is more than satisfactory despite certain performance metrics remaining unsatisfactory.

Acknowledgements

First and foremost, I would like to thank professor Wehenkel for his help throughout the realisation of this thesis, especially concerning its redaction as well as for initially presenting this thesis' subject during one of his academic lessons.

I would also like to personally thank the GAMING1 team for being an incredible support during the whole development of this thesis as well as for providing the adequate tools necessary for the achievement of this work. In particular, I would like to thank Christophe Boniver, Aurélie Boniver, Lucas Naldi, Cédric Dierickx, Mark Van opstal and Jennifer Parraga for their help.

Finally, a big thank you goes to my friends and family who were an incredible support for me during the redaction of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter will introduce briefly the context in which this work takes place in section 1.1. Then, a definition of chatbot will be given in section 1.2. Afterwards, a brief history of the field of conversational agents will be recalled in section 1.3. Finally, it will also motivate the need for further developments in chatbots by referencing recent surveys conducted on this subject in section 1.4.

1.1 Context

Casino games are being digitized more and more each day due to the market's growth. The task of creating digital casino games is nontrivial as it requires very fine tuning on the game designers' end to create both engaging and viable games. As such, customer service is an important point in the user experience, especially on the online scene where most users are extremely demanding both in terms of response time and quality of the answers given. Customer support talks the users through their problems with products and are sometimes a mandatory step when validating one's account.

Tools already exist for organizing and structuring customer support, for instance : Zendesk, Freshdesk, Happyfox, and others. Generally speaking, these tools provide great pieces of functionality to define levels of support, assessing agents' performances, managing communication channels and so on. However there is one area in which these tools lack proper support : automation. Indeed, even though users can potentially issue many different tickets varying in problem types, writing style and vocabulary, most of these tickets only refer to a small subset of problems that can be answered in a systematic fashion. Much like in the famous Pareto principle, only a limited part of users' problems is responsible for most of the workload. Unfortunately, most help desk software do not offer satisfying fully automated solutions to this. Partly because it heavily depends on the type of users contacting the business and because these solutions are generally not built from scratch for the specific target-audience in question. They also tend to be proprietary and difficult to customize. Therefore, a tailored solution for automation is a more suitable approach.

In order to improve their workflow, the *GAMING1* company is looking for ways to automate part of their interactions with customers on their help desk platform. In particular, they would like to deploy a dialogue system solution, also known as chatbot, that would be integrated seamlessly in the existing customer support team. The goal of this work is to design such system.

1.2 Chatbot: A definition

According to the Oxford English Dictionary, a chatbot is defined as follows:

> **chatbot** (n.):
> *A computer program designed to simulate conversation with human users, especially over the Internet.*

In the scientific literature, chatbots are more formally referred to as **conversational agents**. In the context of this document, the terms chatbot/conversational agent will be used interchangeably.

The underlying principle of every chatbot is to interact with a human user (in most cases) via text messages and behave as though it were capable of understanding the conversation and reply to the user appropriately. The origin of computers conversing with humans is as old as the field of computer Science itself. Indeed, Alan Turing defined a simple test referred to now as the *Turing test* back in 1950 where a human judge would have to predict if the entity they are communicating with via text is a computer program or not [1]. However this test's ambition is much greater than the usual use case of chatbots; the main difference being that the domain knowledge of a chatbot is narrow whereas the Turing test assumes one can talk about any topic with the agent. This helps during the design of conversational agents as they are not required to have a (potentially) infinite domain knowledge and can, as such, focus on certain very specific topics such as for instance helping users book a table at a restaurant.

Furthermore, another general assumption chatbot designers bear in mind is that users typically have a goal they want to achieve by the end of the conversation when they initiate an interaction with a chatbot. This then influences the conversation's flow and topics in order to achieve the chosen goal. This can be exploited by developers since certain patterns of behaviour tend to arise as a result.

Therefore, the definition of a chatbot adopted for this document is a computer program communicating by text in a humanly manner and who provides services to human users in order to accomplish a well-defined goal.

1.3 A brief history of chatbots

The first instance of a conversational agent was born in 1966: ELIZA was a computer program that simulated a psychiatrist and rephrased user input using basic (by today's standards) natural language processing techniques [2]. Despite being relatively simple, the program managed to give the illusion of understanding the user's problems and successfully fooled a great many people. Its creator, Joseph Weizenbaum, even mentioned that his secretary would ask him to leave her so she could have a private conversation with ELIZA.

Then during several decades, chatbots heavily followed ELIZA's approach with minor additions brought into the field like speech synthesis and emotions management. Then in 2001 came SmarterChild, a conversational agent developed by *ActiveBuddy, Inc.* (now *Colloquis*) that operated on AOL Instant Messenger and MSN Messenger. Inspired by the rise of instant messaging platforms such as SMS, SmarterChild was created to provide quick access to news, weather forecasts, sports results, etc... The main innovation was that SmarterChild was connected to a knowledge base and detained useful information for its users. Unfortunately, the technical limitations of natural language processing caught up with bots on those platforms at the time and they were forgotten by History.

The next advancement for conversational agents was made by a team at IBM through the Watson AI project that has been in development since 2006. The agent was designed with the sole purpose of winning the American TV show *Jeopardy!* which it did in 2011 when competing against two of the show's former champions. *Jeopardy!* is interesting from an NLP point of view since the questions involve a lot of play on words and require fast information retrieval in vast knowledge bases. Unfortunately this AI in its past form could only answer to one-liner questions and was unable to carry on a proper conversation with someone else.

Finally in the early 2010's came the rise of virtual assistants such as Apple's Siri, Microsoft's Cortana, Google's Google assistant, Amazon's Alexa and others. Those agents brought in the field the concept of conversation as well as goal-oriented dialog. Another major event in the field of chatbots was the release of the Messenger Platform for Facebook Messenger in 2016 and allowed the creation of conversational agents for non-AI related companies.

As shown in this brief summary of the field of conversational agents, a lot of progress has been made since the early days of NLP. This does not imply however that current solutions are without flaw as will be highlighted in the next section.

1.4 Motivation

Several studies have been conducted studying user preferences concerning customer service. In particular, a recent survey done by *Drift* in collaboration with *SurveyMonkey Audience*, *Salesforce*, and *myclever* assesses the image of chatbots in the eyes of the customers [3]. It stems from the survey the following observations :

1. Customers experience problems with traditional online communication channels : 34% state that websites are hard to navigate, and 31% say that they can't get answers to simple questions through those communication vectors.

2. Customers see the following potential benefits in chatbot services : 24-hour service (64%), instant replies (55%), answers to simple questions (55%).

It is clear that the customers' needs and expectations are not being fulfilled by traditional channels. They even see the perks of chatbots in comparison to other solutions.

However, despite all of this, they still feel that current chatbots are not as effective as they could be as observed in a survey conducted by *Chatbots.org* [4]. Across all generations, 53% of customers view chatbots as "*not effective*" or only "*somewhat effective*". It highlights a discrepancy between the potential advantages of these methods and the actual implementations' capabilities. Nonetheless, when dividing the customers' impressions of chatbots by generations, it becomes clear that younger and more tech-savvy people are more optimistic regarding conversational agents. According to this survey, millennials and generation Z respectively rated chatbots as "*effective*" or "*very effective*" 56% and 54% of the time, whereas boomers and the silent generation respectively gave chatbots that rating only 38% and 49% of the time.

All in all, this suggests that additional research and advancements be made in order to unlock conversational agents' full potential.

Chapter 2

Theoretical background

In this chapter will be formally introduced the theory behind the main problems in the conversational agents field as well as the state-of-the-art machine learning and artificial intelligence techniques used in practice for chatbots.

2.1 High-level view of a conversational agent

Conceptually, a chatbot is composed of multiple components working in unison to accomplish a common goal. Figure 2.1 summarises in visual form the relations between each parts of a conversational agent.

Upon reception of a new message, it is first processed by the language identification module. This can range from a simple tag retrieval to more elaborate statistical methods. The new message, as well as the language and potential previous conversation messages retrieved from backend, are then fed to the intent classifier module whose role is to infer the intent the user is trying to convey.

Afterwards, the message's metadata, inferred intent and other information from backend will be used to determine an appropriate action or sequence of actions. For instance, a chatbot may decide to reply with a question if the intent is still not clear, or it could decide to reactivate a user account if the user's intention is to ask permission to reactivate his/her account.

Finally, the action handler module takes as input an action and properly executes said action. This is useful to implement it this way since a same action can be executed in different ways depending on the agent's environment. The way an action is performed can be completely different depending if the bot operates on the Messenger platform than on a company's website.

2.2 Common deep learning techniques

This section will introduce the theory behind the most promising deep learning techniques for NLP. In particular, popular recurrent neural networks and word embedding techniques will be discussed extensively.

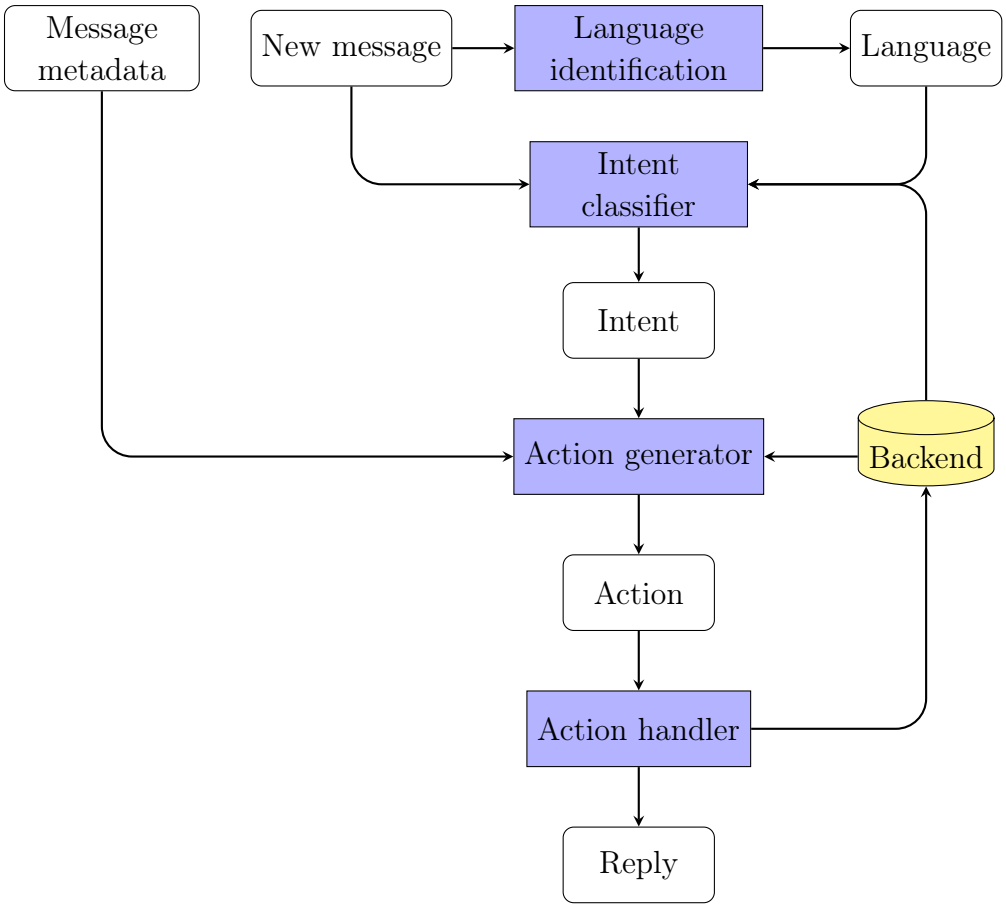


Figure 2.1: High-level schematic of a chatbot.

2.2.1 Word embedding

In a nutshell, word embedding is simply the act of mapping words into vectors. This vector representation can then be directly fed into a machine learning algorithm as features. There are multiple ways of performing this operation, ranging from a simple count vector to deep learning approaches such as Word2vec [5] and GloVe [6]. The latter will be of particular interest for this thesis as they have proven to work efficiently in the field of conversational agents. In particular, the skip-gram model will be explained extensively since it is the technique used in the Keras library for embedding layers.

Skip-gram model

First introduced in *"Efficient Estimation of Word Representations in Vector Space"* [5], the underlying principle of the skip-gram model is very simple : train a shallow neural network containing a single hidden layer of fixed size in order to predict a context given a word. Figure 2.2 is a schematic view of the skip-gram model architecture.

Essentially, artificial training examples of the form $(w_t, [w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}])$ if the window size is 5, are generated. The definition of context varies between authors but the basic principle remains the same : a sliding window of fixed size is

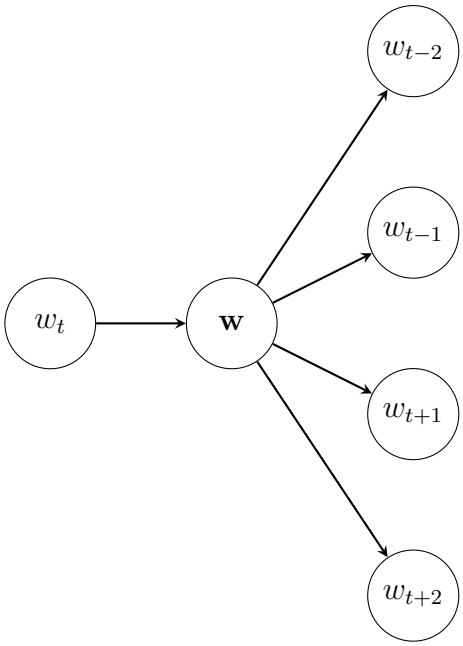


Figure 2.2: Skip-gram model in the case of a 5 words window.

run across the words sequence; the middle of it corresponds to the target word and the words preceding it and the ones after it form the context for that word. The ultimate goal is to extract the inner layer and use it as the vector representation for the trained vocabulary. Indeed, if the input words are one-hot encoded, it only serves as a lookup table of vector representations. In practice it means that the corpus is converted into sequences of indices at first, then the skip-gram model is trained on those sequences and finally the sequences of indices are converted into sequences of vectors to train the final algorithm on.

2.2.2 Recurrent Neural Networks

RNNs or Recurrent Neural Networks are a special type of neural networks specifically tailored for sequences of data. In a nutshell, they are neural networks with loops. More formally : they contain an internal state, usually denoted C_t for the state at time t, which is fed back into the neuron as input at the next timestep. The neuron also outputs a value h_t at each timestep. However, the main issue with naive RNN implementations lies in the fact that they suffer heavily from the vanishing and exploding gradient problems as was proven by Pascanu, Mikolov, and Bengio in *"Understanding the exploding gradient problem"* [7]. Several variants of RNNs have been proposed in order to alleviate this issue using various mechanisms. The most common ones will be presented in the following paragraphs.

Long-Short-Term Memory Units

Long-Short-Term Memory units (LSTMs for short) are used extensively in Natural Language Processing. They were first introduced in 1997 by Hochreiter and Schmidhuber [8] and have become increasingly more popular in recent years thanks

to advancements in hardware accelerated deep learning. They have also shown promising results in machine translation [9] and image captioning [10].

An example of unrolled LSTM layer is depicted in Figure 2.3. The main components of an LSTM unit are the *forget gate*, *input gate*, *cell state* and *output gate* all of which will be explained in this section. Let $\boldsymbol{x} = (x_0, x_1, \cdots, x_\tau)^T$ be the input sequence, and $\boldsymbol{h} = (h_0, h_1, \cdots, h_\tau)^T$ be the hidden state produced by the LSTM layer, then the relation between $\boldsymbol{x}$ and $\boldsymbol{h}$ is given by the following equations :

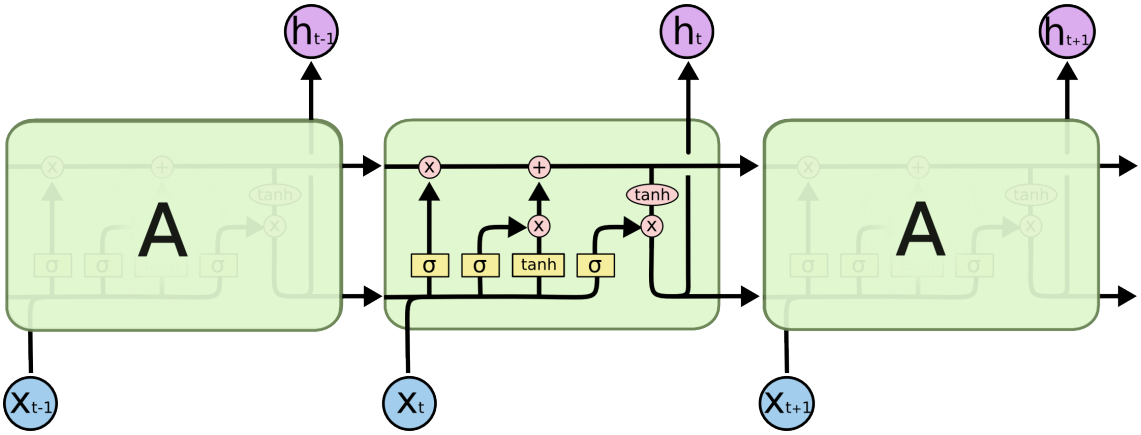


Figure 2.3: Example of unrolled LSTM layer made of 3 units.

Source: Olah, *Understanding LSTM Networks* [11]

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.1}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.2}$$
$$C_t = f_t C_{t-1} + i_t \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.3}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.4}$$
$$h_t = o_t \tanh(C_t) \tag{2.5}$$

Equation 2.1 is the forget gate's output. Intuitively, it represents which information the unit will keep or forget from the previous cell state C_{t-1} as it is directly multiplied with it. Figure 2.4 illustrates this process.

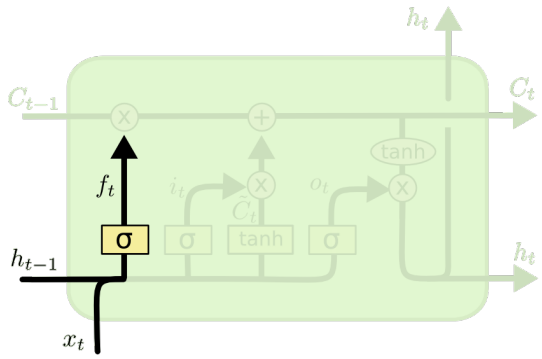


Figure 2.4: LSTM unit's forget gate.

Source: Olah, *Understanding LSTM Networks* [11]

The input gate's response is given by equation 2.2. It simply creates a new candidate value for the cell state, denoted $\tilde{C}_t$, scaled by i_t as shown on Figure 2.5.

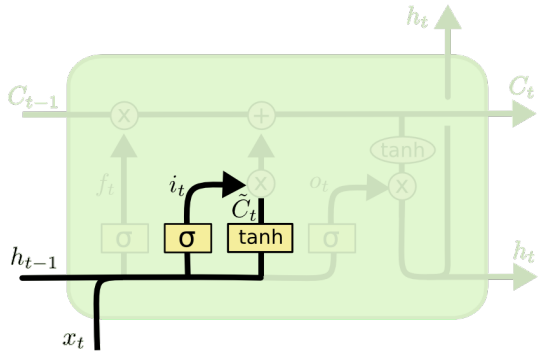


Figure 2.5: LSTM unit's input gate.

Source: Olah, *Understanding LSTM Networks* [11]

The cell state is then updated according to equation 2.3. It simply combines the effects of the forget and input gates. The relevant connections for this update are highlighted in Figure 2.6.

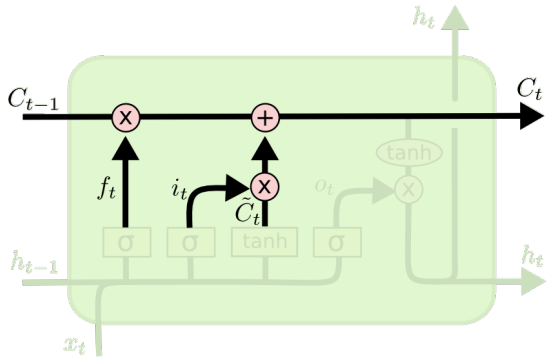


Figure 2.6: LSTM unit's cell state update.

Source: Olah, *Understanding LSTM Networks* [11]

Once the cell state is updated, the output gate's activation can be computed using equation 2.4 to generate the unit's hidden state h_t. This is shown in Figure 2.7.

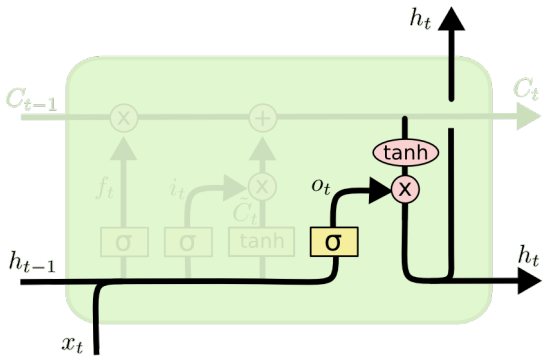


Figure 2.7: LSTM unit's output gate.

Source: Olah, *Understanding LSTM Networks* [11]

Gated Recurrent Units

Introduced in 2014 by Cho et al. [12], Gated Recurrent Units (or GRUs) are a variant of LSTMs where the forget and input gates are merged into a single update gate, as well as the cell and hidden states. This results in fewer parameters to tune and hence shorter training times for GRUs compared to LSTMs. Nonetheless their performance has been proven to be comparable to that of classic LSTM networks [13].

For the sake of completeness, the equations for GRUs are given below :

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{2.6}$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{2.7}$$
$$\tilde{h}_t = \tanh(W \cdot [r_t h_{t-1}, x_t]) \tag{2.8}$$
$$h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t \tag{2.9}$$

where z_t and r_t correspond to the GRU's inner gates as shown in the diagram in Figure 2.8.

Both the LSTM and GRU were considered for developing the chatbot. Their use will be examined further in Chapter 6.

2.3 Language identification

Inferring the language of a text is sometimes a necessary first step in a larger natural language processing chain. Some languages even share homographs (e.g. *room* which appears both in English and Dutch even though it has a different meaning in

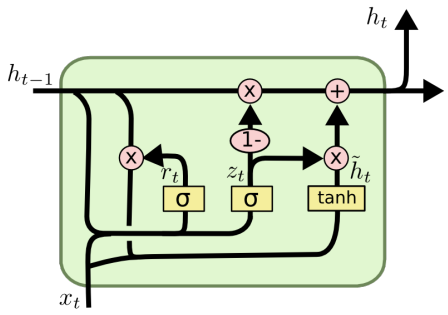


Figure 2.8: Schematic view of a GRU.

Source: Olah, *Understanding LSTM Networks* [11]

each language) which may confuse algorithms with the semantics of these particular words hence requiring the need to identify the correct language for a given text before processing it further.

In this work, the assumption that messages can only be written in a single language will be made; however, there are versions of this problem which involve detecting multiple languages in a single piece of text [14].

2.4 Intent classification

Upon receiving a new message, the conversational agent has to be able to identify the goal the user is trying to accomplish. This is usually modelled as a multiclassification problem whose labels are the names of the possible user intentions. Techniques to solve this problem vary from simple keyword extraction to Bayesian inference in order to determine the user's request based on multiple messages. LSTM networks have been previously known to work well in this area [15]. Those are used as well for the development of this project.

2.5 Knowledge management

An intelligent agent can only do so much without knowledge. The field concerned with allowing computers to handle knowledge has progressed significantly in the 1980's under the name of *knowledge engineering*. Early techniques involved usually an inference engine in order to manipulate facts and derive new knowledge using second and first order logic. It is a way of deriving answers to incomplete questions and are usually easily transcribed into API calls.

For conversational agents, knowledge engineering is extremely useful for instance to answer basic questions about general facts. Siri and Amazon Alexa use internally knowledge inference methods to retrieve facts from the web and other sources (e.g. asking Alexa about trains departing from Brussels today might trigger an internal inference operation of the form $train(brussels, D, today)$ where D is an anonymous variable representing a destination).

Nowadays, knowledge management is mostly done through API calls and optimised database requests. Although more exotic methods inspired by graph-structured ontologies are used every now and then for knowledge bases [16].

2.6 Responses generation

In order to communicate, a conversational agent must possess the ability to reply. Moreover, the replies need to be coherent according to the conversation's context. This issue can be tackled using two different modules working in pair : one that generates a list of candidate replies and the other that selects the most appropriate one or ranks them based on a certain metric. Two popular approaches have emerged from this subproblem : retrieval-based and generative-based methods.

Retrieval-based techniques simply rely on a large database of candidate responses and matches them with information from the user's message to find the most appropriate answer. This information can simply be a regular expression that looks for particular sentence structures or can be the output of a machine learning model. The main advantage of this approach is that the chatbot's maintainers can control every answer and thus avoid inappropriate replies.

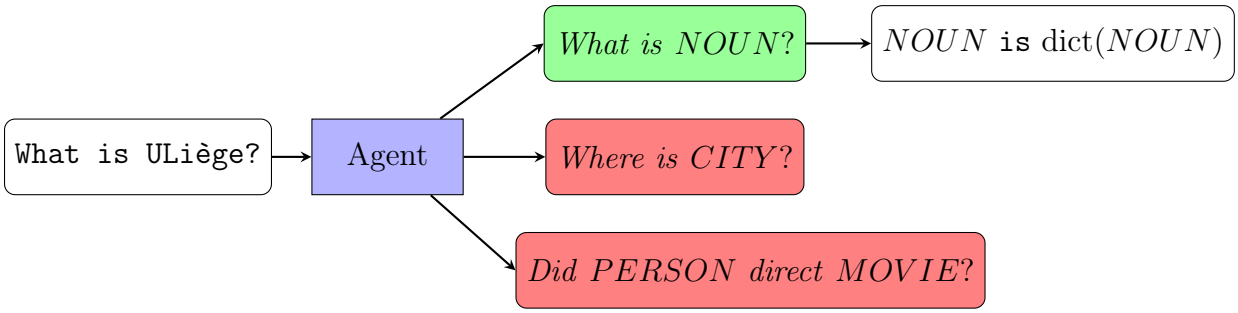


Figure 2.9: A toy retrieval-based example where the user query matched a simple rule, in green, which triggers the corresponding answer.

Generative-based techniques on the other hand rely on generative models to produce new replies without the need for an extensive database of examples. Novel responses can be generated easily provided the model is trained correctly. However as of today, generative-based approaches' performance is insufficient relative to companies' constraints. and the industry is still not convinced of its potential, as shown by the unfortunate tale of Microsoft Tay [17].

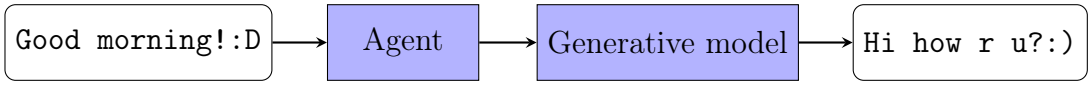


Figure 2.10: A toy generative-based response generation example.

Recent developments have attempted developing hybrid solutions where replies are generated if they cannot be retrieved as demonstrated by Tammewar et al. in *"Production Ready Chatbots: Generate if not Retrieve"* [18].

2.7 Performance assessment

One area of improvement in the field of conversational agents is the performance assessment and metrics used to quantify the quality of a chatbot's behaviour. Liu et al. in *"How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation"* show how uncorrelated standard metrics in the field tend to be against intuitive human judgement [19] such as the BLEU [20] and ROUGE [21] scores. The fundamental issue is that speech and eloquence is inherently subjective and it is therefore very difficult to accurately quantify it, as is the case with any other subjective property of a system.

Chapter 3

Objectives and milestones

This chapter will start by a brief description of the problem in section 3.1. Then, the overall objectives that were agreed upon with the *GAMING1* company will be listed in section 3.2, as well as their requirements outlined in section 3.3.

3.1 Problem description

Usually, users encountering issues contact customer service through one of the supported communication vector (E-mail, social media, website chat, etc ...) creating what is called a *ticket*. Tickets are a special data structure which contain a conversation between an agent and a customer as well as metadata such as the time and date of the messages, the language of the conversation, tags used for filtering tickets, etc ...

Customer service agents then analyse the messages through the Zendesk tool [22]. They can generate automated replies called *macros* which also assign specific tags to the ticket. Sometimes, the agent might be allowed to perform certain actions in specific cases. For example, if a player complains about not being able to log into their account and has been blacklisted, the policy is to escalate the ticket to the Risk & Fraud department.

The opportunity for automation in this case comes from the fact that certain patterns of behaviour arise in the messages sent by users. For instance, the vocabulary used for a player that is stuck in a game is very different than a player that has forgotten their password. Therefore, such patterns could be exploited to easily automate certain aspects of customer support and alleviate part of the workload to a conversational agent.

3.2 Goals

The high-level view of the final objective is to design a scalable and easily maintainable multilingual chatbot solution that can interface with the company's existing customer support software. It should serve as the first communication layer in the company's customer service and provide a way for users to send free-form text

messages to help them with the most common issues they encounter regarding the company's products. In short, it should replicate the menial tasks performed by customer service agents on a daily basis.

Ideally, the software should :

- Provide means to automatically extract messages from the company's customer support software.
- Process extracted data into a suitable format for machine learning solutions.
- Provide a scalable machine learning pipeline for the system's models, starting from the raw data.
- Be designed in such a way that it can be deployed on multiple platforms with little effort.
- Be able to reply to users in real-time.
- Help users with a subset of problems that they often experience.
- Ask the user for more information if their intent is not clear enough.
- Request human intervention when appropriate.
- Allow tuning of the bot's responses.
- Support multiple languages.

3.3 Constraints

Quite naturally, the designed solution must also follow certain guidelines enforced by the company :

- Other developers should easily understand the codebase and pick up the project later on without difficulties.
- Using the software should be straightforward and well-documented.
- Training the system's models must take a reasonable amount of time.
- The bot should make as little errors as possible to ensure that customers do not get frustrated.

The project is therefore articulated around those constraints in order to not violate them.

3.4 Identified user problems

The first question to answer is "*Which customer problems should the chatbot be able to handle?*". This is a non-trivial issue since the problems should be

1. common enough to provide enough training samples.

2. simple to resolve so as to limit the chatbot's potential mistakes.

A few user scenarios were thus identified, with the help of customer support representatives, as adequate for this project. They are summarised in Table 3.1.

User problem	Description
Stuck in game	The user is experiencing difficulties playing a game and cannot progress further. Usually it indicates a rarely occurring turn of events which makes their in-game credit stuck.
Email modification	The user would like to change its email address.
Suspended account	The user cannot access their account as they have been suspended.
Account activation	The user cannot access their account or are prevented from using certain functionalities because they forgot to activate it.
Registration	The user is unable to register on the online platform.
Withdrawal	The user has issued a credit withdrawal to his bank account but is asking why it has not been completed yet.
Forgot password	The user has forgotten their password. The proper action for customer service agents is to provide a new generated password for the user.

Table 3.1: List of considered user problems

Customer service agents then respond appropriately using macros which attach tags on the ticket. This can be useful to trace the usual responses agents make regarding particular types of tickets. An additional assumption is made : users will only talk about one problem at a time in a single ticket.

Chapter 4

Software architecture

This chapter will talk about the overall architecture of the conversational agent. More specifically, section 4.1 describes the agent's environment in details, section 4.2 will present the chatbot's class diagram, and section 4.3 will describe the agent's modules, what purpose they fulfil and how they accomplish it.

4.1 Agent environment

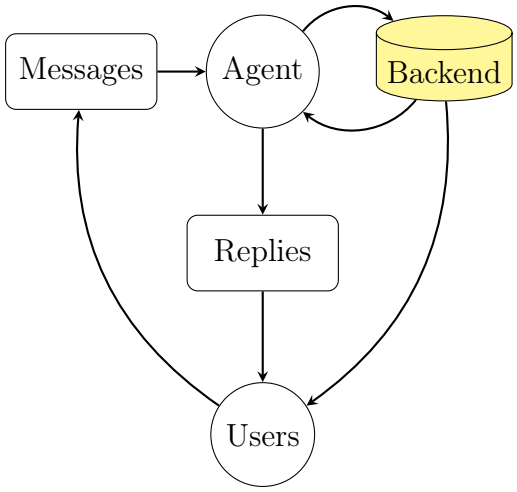


Figure 4.1: Agent environment model.

The agent environment consists of a few different parts. One of them is the user messages : they are a dynamic input that the agent can receive at any given time. They consist in a string representing the actual text sent by the user, and a metadata structure containing additional information like a pointer that links the message to the structure representing the particular conversation it belongs to, and possibly the time the message was sent, on which platform the message was sent, etc ... The chatbot can only read the information it contains with no possible means of modifying it.

Another significant part of the environment the agent has access to is the company's backend which contains additional information about the user and the database's

state. The agent can both inspect and influence certain aspects of the backend. The chatbot also has the ability to send replies to the users in order to obtain new information, or simply to tell them that their request has been accepted and treated accordingly.

A visual representation of the agent's environment is presented in Figure 4.1.

4.2 Class diagram

In this section, the connections between all the agent's modules will be highlighted and discussed. Figure 4.2 shows the class diagram of the agent's modules.

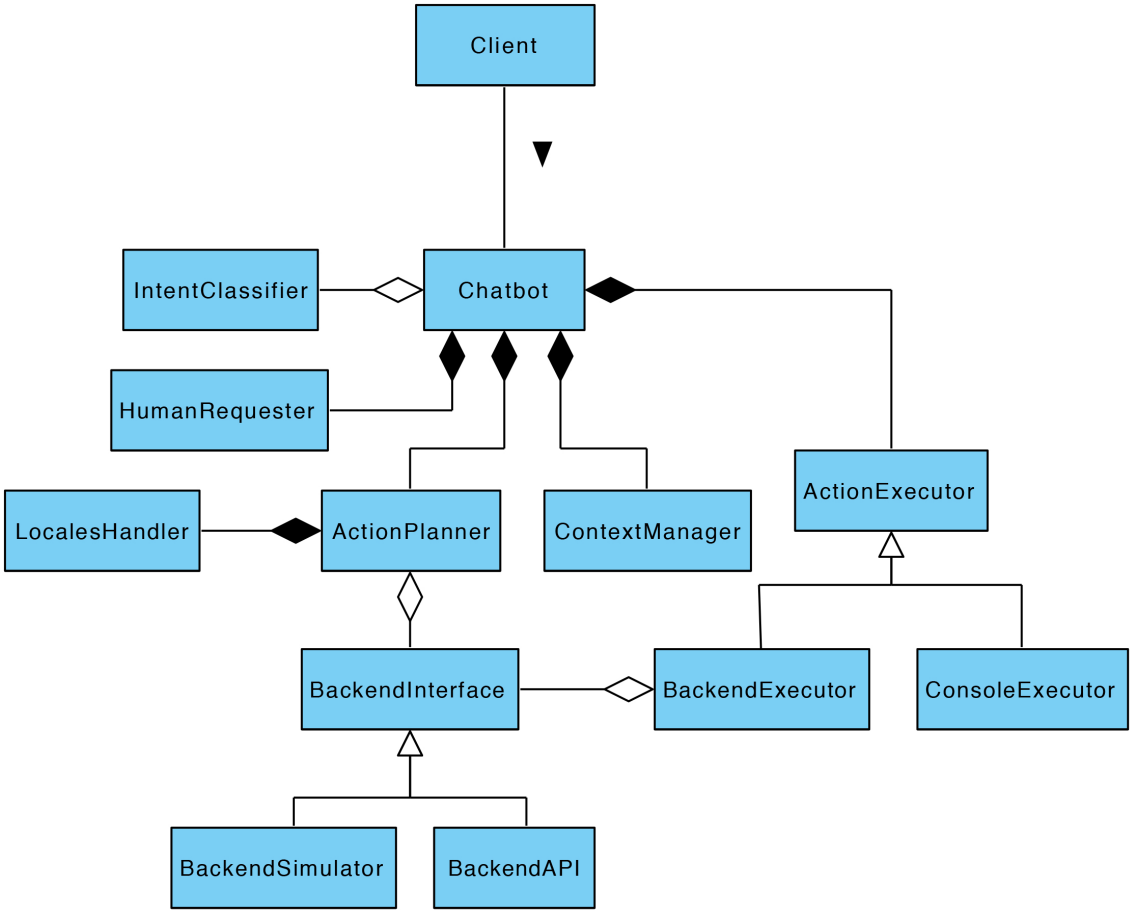


Figure 4.2: Chatbot class diagram.

As can be clearly seen from this diagram, the chatbot is made of several components each solving one particular problem. The component labelled *Client* at the top of the diagram is simply a generic client program that makes use of the chatbot. It can simply be a terminal client or a more complicated dynamic online listener that fetches new tickets when they are issued through the system. The chatbot's 5 main components are : *IntentClassifier*, *HumanRequester*, *ActionPlanner*, *ContextManager*, and *ActionExecutor*.

In particular, it is also shown that the *ActionPlanner* has access to 2 submodules, one of which (*BackendInterface*) being shared with another submodule.

4.3 System modules

This section will present the system's submodules by discussing their respective roles, and will also give their inputs as well as their outputs.

4.3.1 Intent classifier

This module is responsible for inferring the user intent based on a message. It heavily uses neural networks to perform its normal operation. Its only input is a single user message and its output is a dictionary that maps every supported user intent with its corresponding probability for that message. The training of this module will be explained more thoroughly in chapter 6 section 6.1.

4.3.2 Human intervention requester

This module operates right after the intent classifier. Its role is to request the intervention of a human customer service agent in the case the intent of the user could not be identified correctly. It is provided as a module as it is easier to change between behaviours (For instance, one might want a more conservative chatbot when dealing with high profile customers that will request human intervention more often).

Its input is the whole context of a dialogue with a user, as provided by the context manager. Its output is simply a boolean value which is true if the module judges that, given the context, human intervention is required; it returns false otherwise. Its implementation is discussed in more details in section 6.2 of chapter 6.

4.3.3 Context manager

The module whose role is to handle the structure which organises contexts. A context in this case is defined as the sequence of messages, as well as metadata attached to them, which are useful for helping a user with a single problem. In human terms, this roughly corresponds to our intuitive concept of a single conversation/ticket.

4.3.4 Action planner

The action planner's goal is, given a probability vector, decide which actions the agent should take. It is responsible for observing the backend's state and generating a payload that contains the actions to perform and in which order. In this case, the payload's format will simply be a JSON object. It is also responsible for constructing the actual reply to be sent to the user. It makes use of two submodules: the locales handler and the backend interface.

Locales handler

This module is a simple wrapper for everything related to localisation. Inspired from Chrome's `i18n` infrastructure for browser extensions, it provides a common interface for all localised text. This enable common code without having the need to localise every constant or string that appears in the codebase.

Backend interface

A simple interface class over the system's database that specifies which functions are available to the agent. It is useful to specify it as an interface since it gives the chatbot versatility. Indeed, to deploy the chatbot on another platform, one can write a new class inheriting from this interface that works with a different kind of backend.

4.3.5 Action executor

This module, as the name implies, is responsible for executing sequences of instructions. It works directly on the *Action planner*'s output and actually sends the messages and performs the planned instructions on the backend.

Chapter 5

Software implementation

This chapter will justify the use of certain libraries for the project, as well as discuss the concrete implementation details of the completed system.

5.1 Programming language and libraries

Considering the context of the project and the company's software environment, the language Python was chosen as main programming language to implement the chatbot. This is a trade-off between ease of implementation due to the language's prolific machine learning catalogue, and the ease of use for future developers at *GAMING1*. Indeed, the company mainly works with the .NET framework and as such writes a significant amount of code in the C# language. However, Python is not a difficult language to learn and was deemed appropriate in this situation.

Concerning libraries, there are a few used to implement the chatbot in practice. The first one is the *Keras* library [23] whose overall goal is to provide a common and easy-to-use interface for several deep learning frameworks such as *Tensorflow* [24], *Theano* [25], etc ... The chatbot uses *Keras* on top of the *Tensorflow* library. Another important library used is *pandas* [26] which provides ways to efficiently store and organise datasets. It is useful for manipulating large amount of data. Finally, a minor (relative to this project) but nevertheless important library is the *scikit-learn* library [27]. It is used in this context for some utilities such as splitting a dataset into train validation and test sets.

5.2 Extracting data

The starting point is to be able to extract a messages dataset from the company's customer support system. Since the company started using Zendesk since 2013, there is a significant amount of data to retrieve from their database. There are a little more than 1,400,000 tickets present in total in their system as of the writing of this report[1]. However, Zendesk unfortunately does not provide means to export large amount of tickets easily or at least not including the tickets' messages. Besides,

[1]Numbers presented in graphs may vary as more tickets were downloaded in the meantime.

it is easier to design a custom solution that can periodically export new tickets every week or so.

Therefore, a simple API crawler was designed to overcome this issue. The software is simply an infinite loop that requests ticket data in JSON format directly from the Zendesk API 100 tickets at a time since that is the maximum amount of tickets the API can send in one request. It first starts with a seed date which corresponds to the oldest date the crawler will extract tickets from. It will then process the remaining tickets in chronological order. Each time a batch of 100 tickets has been processed, the crawler requests what is called the next *page* of tickets to process. After a defined amount of processed tickets, the crawler will save the current batch of tickets to disk in order to create a backup in case of failure. This is implemented because fully extracting tickets takes a non-negligible amount of time. If it does fail, the crawler looks for the last processed ticket in the backup file and restarts the process starting from that ticket. This mechanism is also used every time the crawler is launched making it possible to execute it every week or month in order to fetch new ticket data and update the model dynamically.

Once all tickets have been extracted, they are then separated by language using the tags attached to them. There are currently 9 supported languages in the GAMING1 system : French, Dutch, English, Portuguese, Spanish, Romanian, Serbian, German and Turkish. Figure 5.1 contains the distribution of tickets amongst all languages.

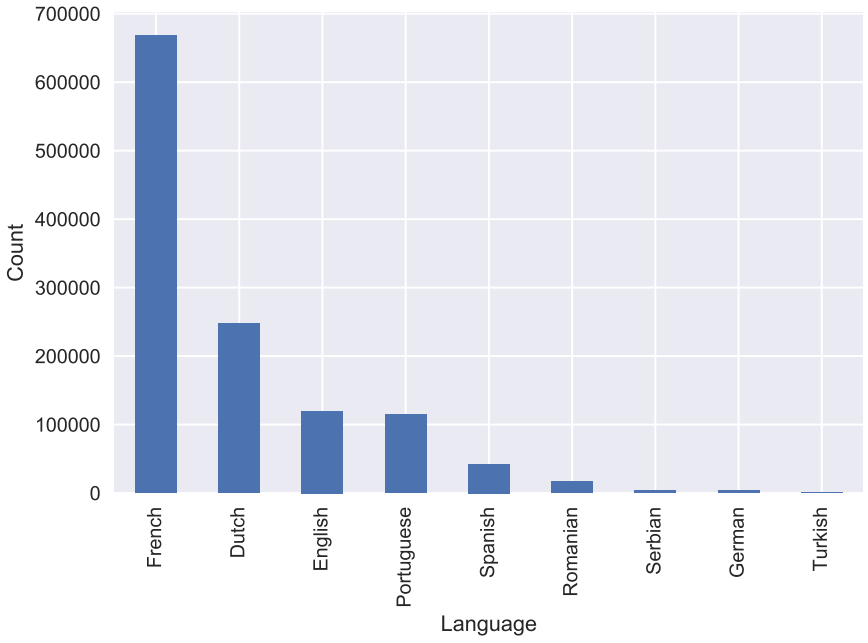


Figure 5.1: Dataset's language distribution

As can be clearly seen, the dataset is significantly biased towards French tickets, making up approximately 48% of the total amount of tickets. However, most other languages contain a satisfactory amount of tickets except for the extremely scarce ones : Spanish, Romanian, Serbian, German and Turkish. The smallest amount

of tickets considered will thus be around 100,000 tickets for Portuguese. All other languages are ignored for the time being due to a lack of data. Moreover, since there is the same amount of tickets for English than for Portuguese, the Portuguese dataset will be ignored for the time being as its analysis would be the same as for the English dataset. Therefore only the French, Dutch and English data are considered for the rest of this thesis.

The data provided by the API contains all sorts of information like the date of the ticket, the time it took for customer service agents to resolve the ticket, the VIP status of the user, which customer support level the ticket was processed, etc... Most of which is useless for the chatbot and is meant for customer support managers as they will use this data to monitor statistics about the workforce and see how well or how badly it is doing. As such, only the first message received from the user and the *Tags* field will be used to train the chatbot.

Once each language dataset has been created, the crawler will iterate through each one of them if a file labelled `tags-`<ISO-639 language code>`.csv` exists. These files contain on each line a user problem label alongside a list of tags that identify that particular problem (Usually leftovers from using certain macros). The tags are language specific hence why there needs to be a separate file for each of them. This will generate new datasets used for intent classification which will be explained in more details in section 6.1.

5.3 Generating responses

Due to the constraints outlined in Chapter 3, the approach taken for generating responses is a retrieval-based one. Indeed, doing so allows the maintainers to ensure that the agent does not say something offensive to the user. The module responsible for retrieving the responses is the `ActionPlanner` and in this case, it contains all the logic that is triggered whenever the user intent is correctly identified. For example, it might call the backend to ensure that the user's account is not blacklisted or that the delay of the user's last withdrawal is because it is performed to an international bank account. The responses have a structure that is reminiscent of what customer support agents might actually answer in real life. Thus they are formulated to look a lot like formal e-mails since that is the primary source of tickets customer service receives.

Chapter 6

Experiments

This chapter will focus on the experiments performed for tuning the chatbot's modules as well as results obtained during training and discuss them extensively.

6.1 Intent classification

The first problem to solve is the intent classification problem. It will be modelled as a multiclassification problem whose output is simply the label of the detected problem. However, in order to allow fine-tuning of the bot's behaviour, the classifier should also output the corresponding vector of class probabilities when predicting.

6.1.1 Dataset generation

The dataset obtained from section 5.2 cannot be fed directly to the intent classifier as it is. The main issue is that the samples are not properly labelled. Fortunately, due to the fact that tags are attached to each macros, most user problems are identifiable using tags attached to the ticket. Thus, the samples are labelled by going through the list of tags attached to the ticket and seeing if they match certain tag types that correspond to a given problem as explained previously in section 5.2.

Furthermore, in order to allow quick experimentation and for efficiency, only the French tickets' dataset will be used at first for exploring the possible network structures; the main reason being that it is the biggest dataset (Containing roughly 600,000 tickets). The implicit assumption is that the French dataset's class distribution, which is displayed in Figure 6.1, accurately represents the other languages' class distribution. As can be seen, the dataset is highly imbalanced and is extremely skewed towards the `Other` class.

Figure 6.2 shows both the French and Dutch distribution of user problems.

As can be observed, the two distributions look very much alike except, perhaps, for the `Stuck in game` class which is more prominent for French than Dutch. Nevertheless, this further confirms that tuning the network for French and then porting it to other languages is a suitable course of action.

Considering the amount of available samples, a simple training-validation-test scheme is used for assessing the performance of the classification model; the corre-

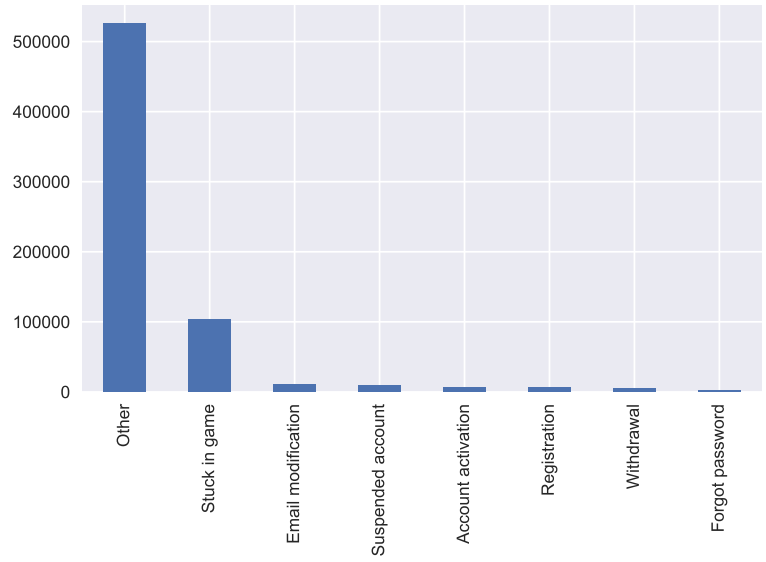
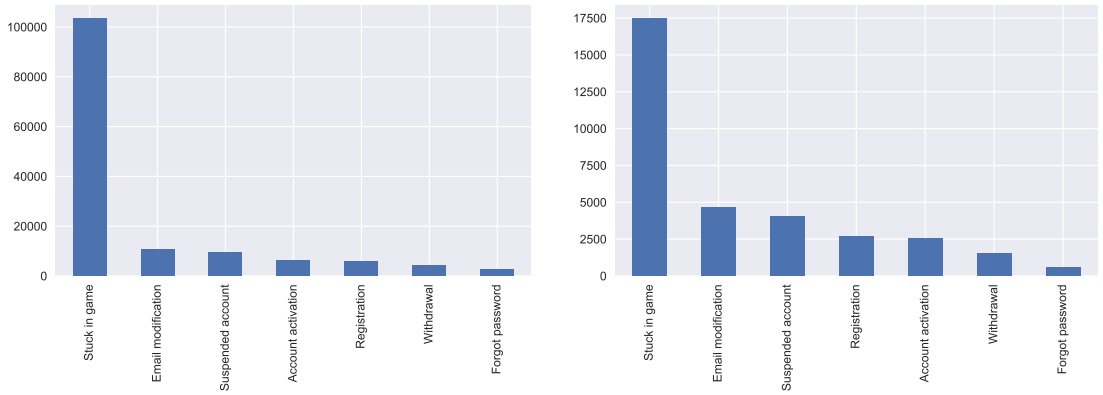


Figure 6.1: Problem distribution for French.



(a) Problem distribution for French.

(b) Problem distribution for Dutch.

Figure 6.2: Problem distributions without the `Other` class.

sponding sizes are 60/25/15. Finally, each sample's label is one-hot encoded as a vector of 0's and a 1 at the index which represents that sample's label. Due to the imbalance of classes in the dataset, the weight of each sample j will be given by

$$w_j \triangleq \frac{N_{y_j}}{\sum_{i=1}^{m} N_i} \tag{6.1}$$

where N_i is the number of samples belonging to class i and y_j the true label of sample j.

6.1.2 Preprocessing

Due to their design, textual data cannot be fed directly to the neural network. They require a small extra step of preprocessing before actual training. They need to be

transformed into sequences of integers of the same length. The preprocessing steps are listed below in the order they are performed :

1. Turn all characters to lowercase.
2. Filter unimportant punctuation characters (e.g. `"!?,^` ...).
3. Tokenize the texts by using words as tokens. A word in this context is defined as a *contiguous sequence of characters surrounded by whitespace*.
4. Compute the texts' vocabulary (i.e. the set of tokens used).
5. Swap each token in the sequences by their corresponding index in the vocabulary.
6. Pad the sequences to a fixed length of 100 tokens.

The final sequences obtained are suitable for training the neural networks.

6.1.3 Network training

Different network topologies were considered for the intent classification subtask. This section will present briefly each of them as well as their performance.

Common structure

Each network is a variant of the same base structure, which is represented in Figure 6.3. $\mathbf{x}$ is the tokenized input message. In this case, the tokenization is done word by word and each word is then converted into an integer that uniquely identifies it amidst the training vocabulary. Then, the sequence of integers is converted into a sequence of real vectors of size 128 by an embedding layer. This sequence of vectors is then sent to the RNN substructure. Finally, the RNN's output is fed through a softmax layer using the standard softmax function, $S(\mathbf{z})$, defined as

$$S(\mathbf{z})_j \triangleq \frac{e^{\mathbf{z}^T \mathbf{w}_j}}{\sum_{i=1}^{m} e^{\mathbf{z}^T \mathbf{w}_i}} \tag{6.2}$$

where $\mathbf{w}$ is the weight vector for all connections between the softmax layer and the previous layer, and m is the output size of the softmax layer i.e. the number of classes. The interesting property about the softmax function is that

$$\sum_{j=1}^{m} S(\mathbf{z})_j = 1$$

and therefore

$$S(\mathbf{z})_j \approx P(y = j|\mathbf{z})$$

This is useful in this case since it is expected that the intent classifier outputs a vector of probabilities alongside its prediction.

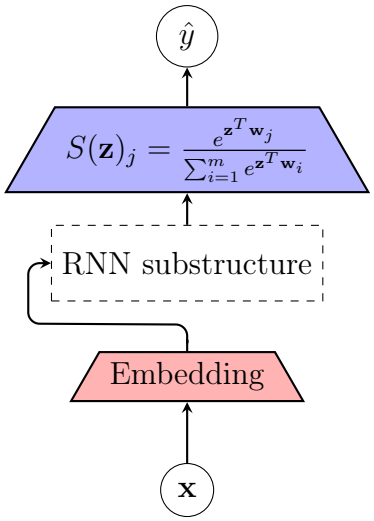


Figure 6.3: Common structure of each network variant.

The loss function that is minimised during training is the *categorical cross-entropy loss* defined as follows :

$$\mathcal{L}(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} y_{i,j} \log(\hat{y}_{i,j}(\theta)) \tag{6.3}$$

where $n \in \mathbb{N}$ is the number of samples, $m \in \mathbb{N}$ the number of classes, $y_{i,j} \in \{0, 1\}$ is 1 if the class of sample i is j and 0 otherwise, and finally $\hat{y}_{i,j} \in [0, 1]$ is the predicted probability that sample i has label j. Another metric that is generated during training is the accuracy of the model's predictions.

The Adam optimiser [28] is chosen to train the neural networks as it has been shown to produce good results in the field of deep learning. It is also computationally efficient which is particularly useful here due to the amount of samples to optimise.

Single LSTM

The first topology considered is a simple single layer LSTM. Figure 6.4 shows the full neural network's structure.

Figure 6.5 depicts the evolution of the accuracy of the network as the number of epochs increases both on the validation and training set[1]. It can be seen that the behaviour displays an erratic behaviour during training as its loss and accuracy fluctuate greatly. Nevertheless, it still almost learns completely how to correctly classify the entire training set and achieves a maximum accuracy of 87.6% on the validation set. The total training time was 314 minutes.

[1]Loss functions for every accuracy graph in this section are presented in Appendix A.

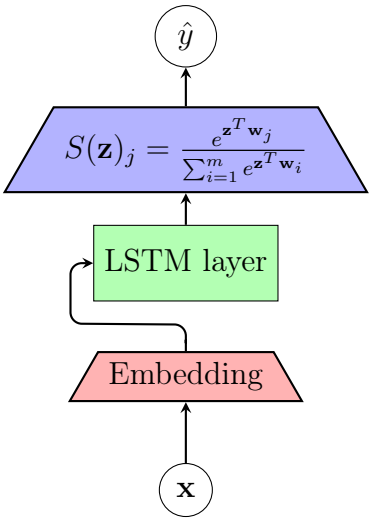


Figure 6.4: Single LSTM structure.

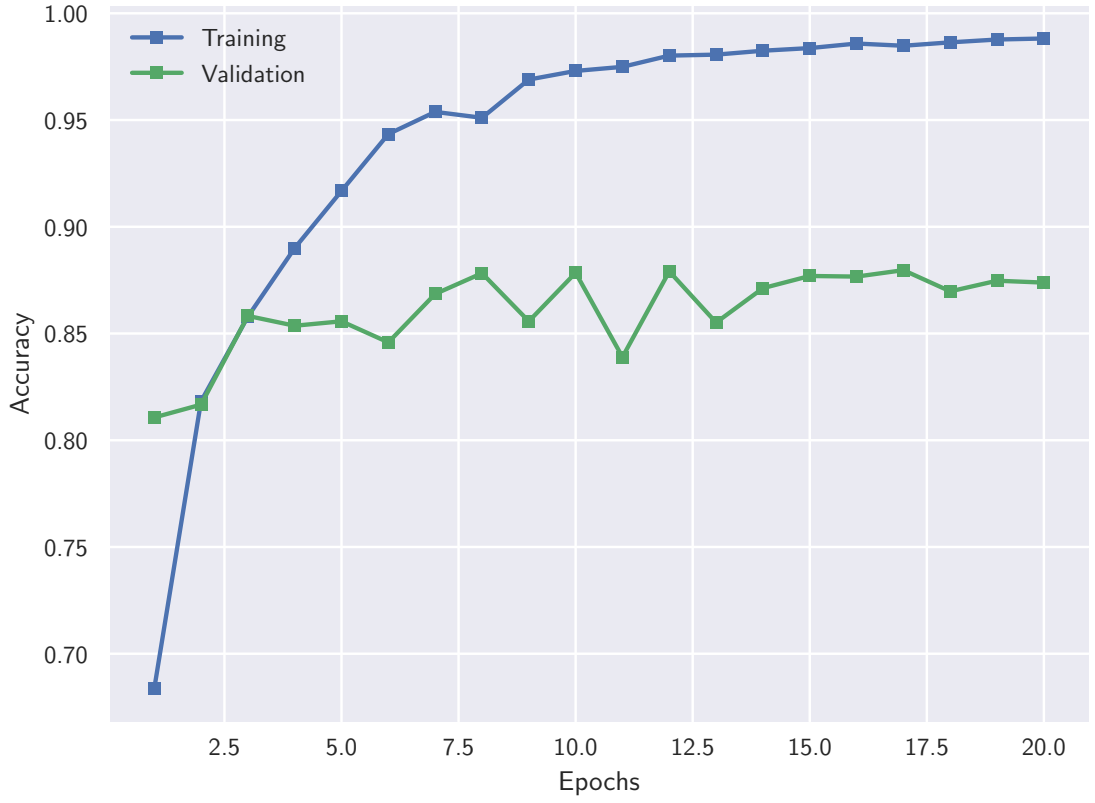


Figure 6.5: Single LSTM layer's accuracy during training.

Single GRU

The next recurrent neural network type considered is the GRU. The goal here is to study the similarities and differences between using LSTM and GRU layers. Figure 6.6 shows the single GRU network's structure.

As is shown in Figure 6.7, the single GRU layer structure performs approximately the same way as the single LSTM layer structure. It displays the same erratic be-

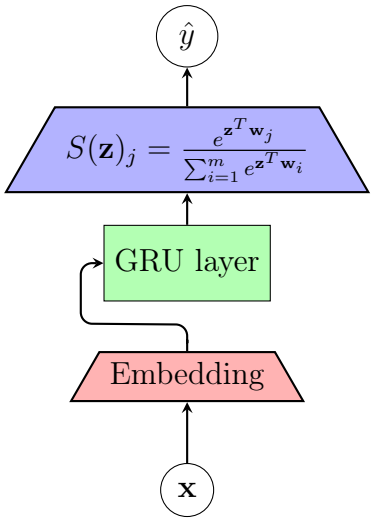


Figure 6.6: Single GRU structure.

haviour during training and achieves almost the same maximum accuracy of 87.5%. However, its total training time is significantly lower since it only took 196 minutes to fully train.

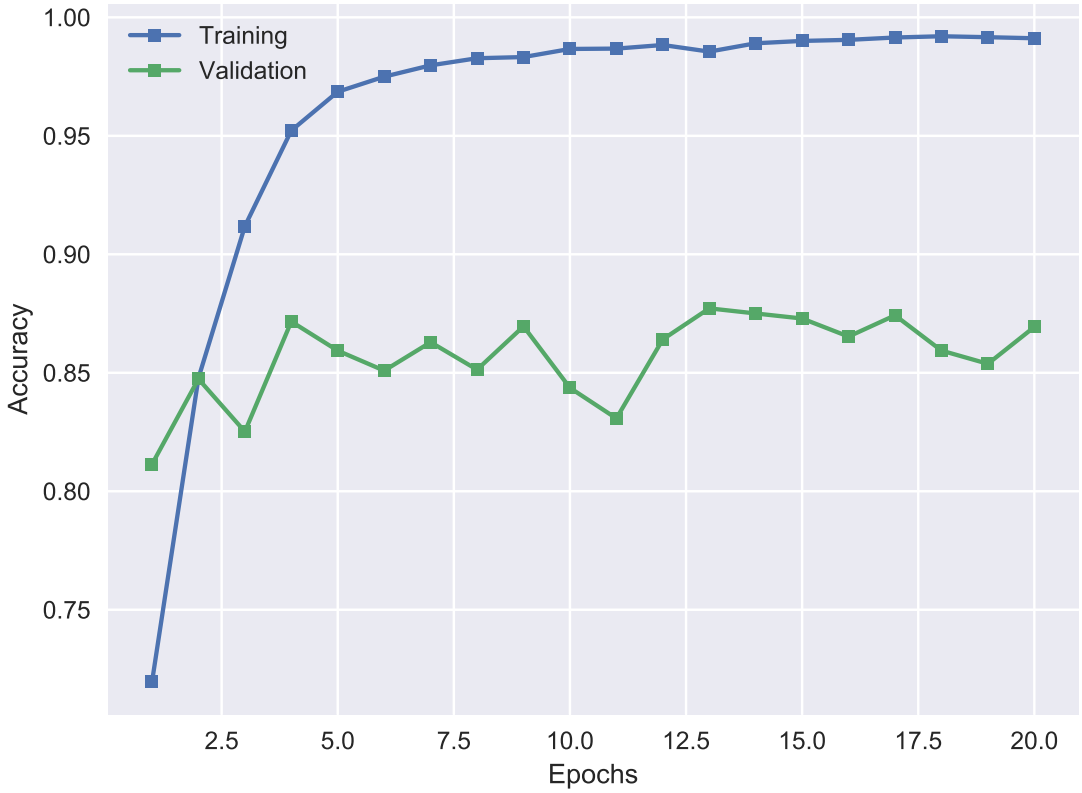


Figure 6.7: Accuracy during training of a single GRU layer network.

Inverted input sequence GRU

The next structure introduces a small variant to the single GRU layer structure by inverting the order of the input sequence. This is because inverting the input sequence has shown to be a simple heuristic that can marginally improve the performance of certain networks, particularly in the subfield of machine translation [9].

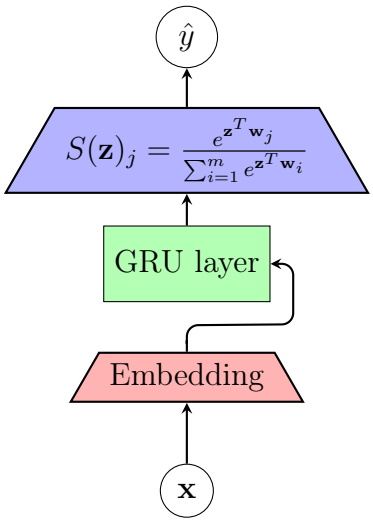


Figure 6.8: Inverted GRU sequence structure.

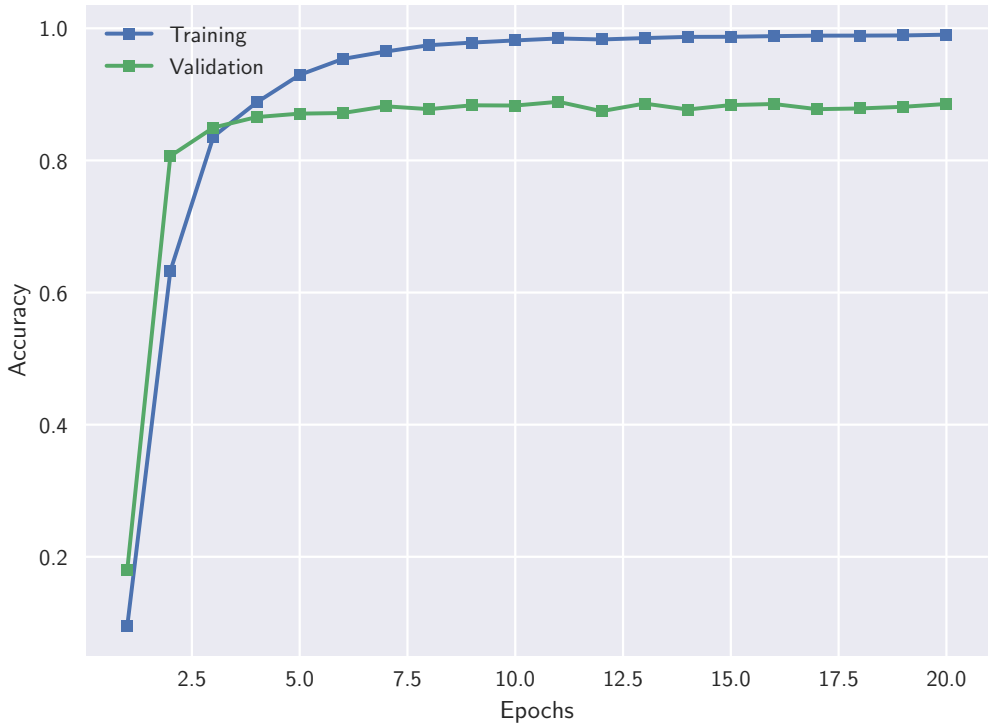


Figure 6.9: Accuracy for a single GRU layer with inverted input sequence.

The evolution of its accuracy is shown in Figure 6.9. It clearly shows that this

structure is much more stable than the previous ones during training as there is less jitter for the loss and accuracy. The training time did not suffer too much from the input sequence being reversed as it took 198 minutes to fully train.

GRU bilayer structure

Another possible way to improve accuracy would be add 2 layers of GRUs stacked on top of each other as this would be able to capture a richer information from the input sequences. Figure 6.10 shows what the underlying network structure looks like.

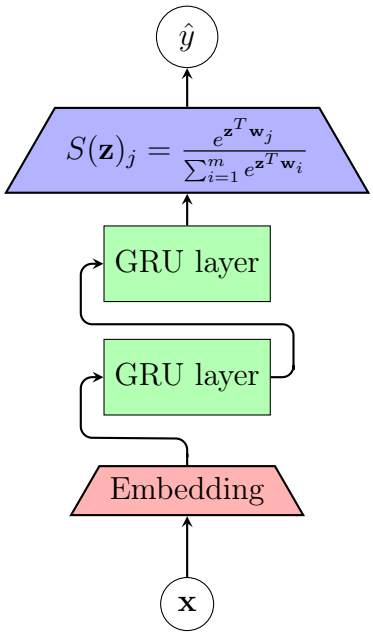


Figure 6.10: Two GRU layers structure.

As Figure 6.11 points out, this type of network does not perform better than the previous iterations. More to the point : it performs worse than a single GRU layer. This might be due to the fact that the overhead of training an extra GRU layer is much greater than the extra information gain it is supposed to provide. The maximum accuracy the network can achieve is only 86.4% for a total training time of 352 minutes which is significantly higher than a simple single GRU layer structure.

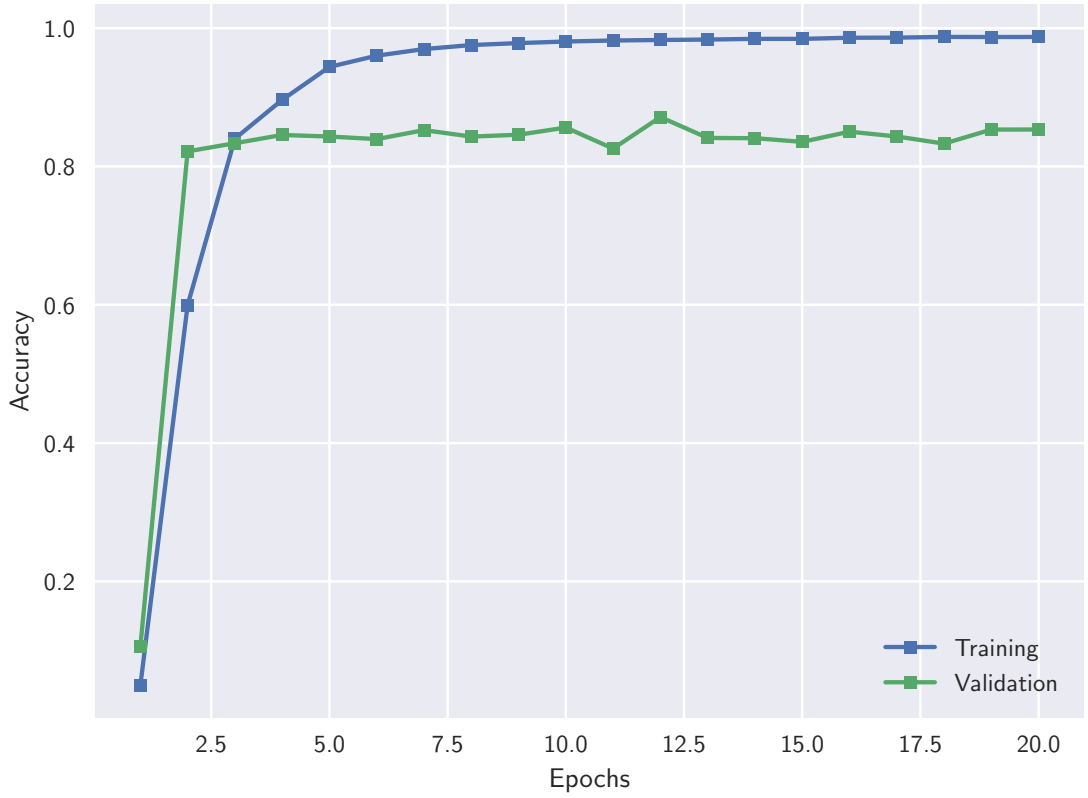


Figure 6.11: Accuracy of a network with two GRU layers.

6.1.4 Final network structure

Having explored multiple candidate structures, the best one can be chosen. Table 6.1 summarises the performance of each network type.

Network type	Best validation loss (lower is better)	Best validation accuracy (higher is better)	Total training time [min]
Single LSTM	0.45	87.6%	314
Single GRU	0.46	87.5%	196
Inverted GRU input	0.49	88.2%	198
Two GRU layers	0.52	86.4%	352

Table 6.1: Intent classification results summary.

Based on these results, the final architecture chosen for the intent classifier is the single GRU layer with inverted input sequence as it provides the greatest accuracy and trains much faster than its LSTM counterpart.

6.1.5 Towards multiple languages support

One of the requirements is for the chatbot to be able to work with multiple languages. In order to do so, there are two approaches that can be explored :

1. Train a different model for each language supported.

2. Train a single model that works on any language.

Both approaches are discussed hereafter. In a first time, two extra models were trained respectively on Dutch and English tickets using the final model architecture chosen in section 6.1.4. Figure 6.12 shows the evolution of accuracy for Dutch as a function of trained epochs.

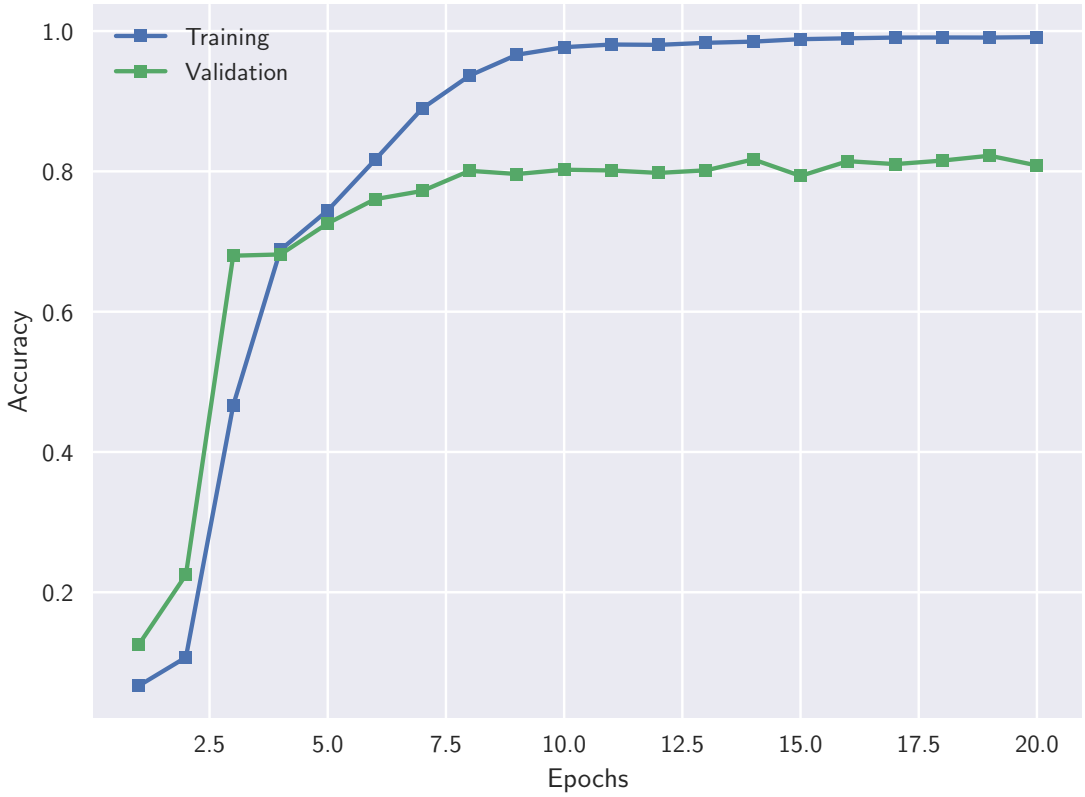


Figure 6.12: Network accuracy for Dutch.

As can be seen from the graphs, the accuracy is slightly lower for the Dutch dataset than for the French one. Indeed, on the Dutch dataset, the network only achieves a final accuracy of 80.7%. Therefore, the accuracy over both datasets would be the average between the two which is around 84.5%. This is nevertheless still an acceptable accuracy in total.

The other approach, consisting in concatenating multiple datasets into a single multilingual one and then training a single model on it, generates the training curve shown in Figure 6.13.

In this case also the accuracy stabilises at 84.3%. Therefore, strictly speaking from the point of view of accuracy, the multilingual dataset and single language dataset approaches are equivalent. However, there are some pros and cons using one or the other. The main advantage of training a model for each language is that the maintainers can fine-tune each model separately. This gives more control to the chatbot's maintainers over the classifiers. The drawback though is that each model

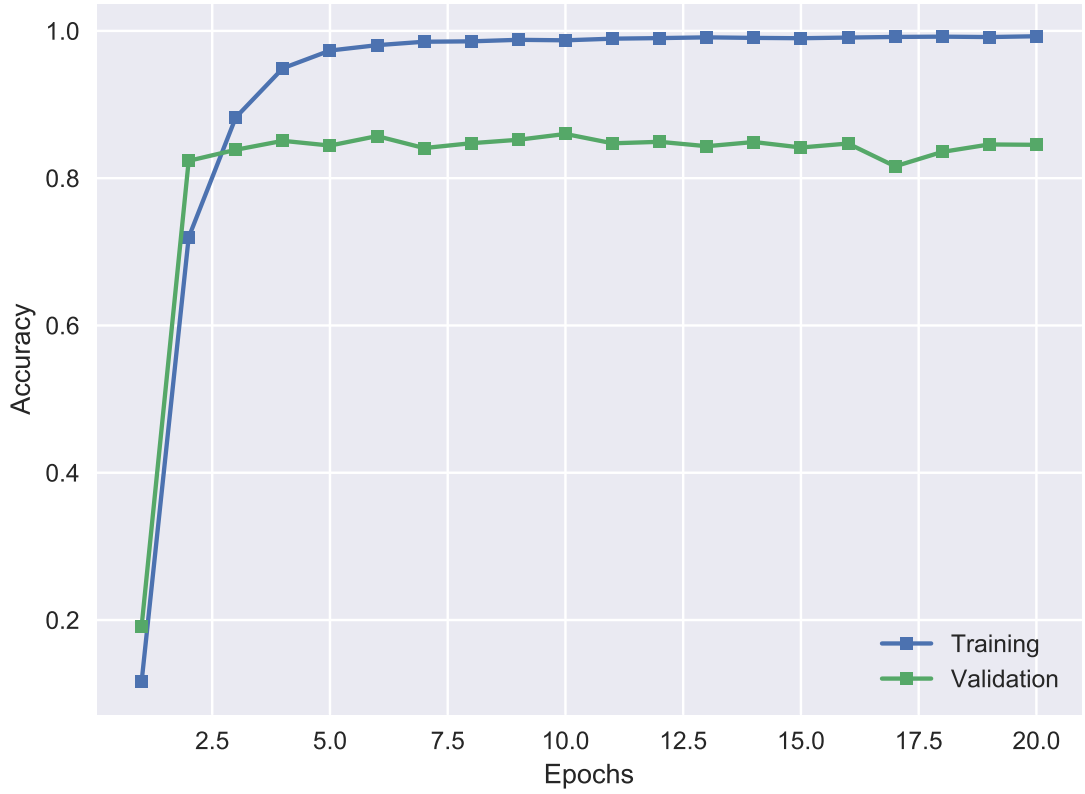


Figure 6.13: Accuracy of the final network on a bilingual dataset.

takes extra space in memory and thus come at a total memory footprint that is higher than a model trained on a mixed dataset as shown in Table 6.2.

Training set used	Size in memory [MB]
French dataset	233.5
Dutch dataset	99.2
Bilingual dataset	319.5

Table 6.2: Memory size of different intent classifier networks

Another drawback for using the mixed dataset approach is that the embedding will be the same for all languages. This could potentially hinder the performance due to the presence of homographs between languages (e.g. '*de*' in French and '*de*' in Dutch). The embedding layer is not capable of differentiating between the two and must produce only one vector for both words even though their semantics might be different.

The choice of the approach used only changes a few minor details which are closer to quality of service than fundamental differences. Nevertheless, for the sake of this project, the first one will be used since it provides flexibility and takes less time to train one model which is useful during the experimenting phase. However, functions are provided in the code to take the second approach if need be.

6.1.6 Epochs analysis

Hereafter, the influence of the number of epochs is analysed. In order to ensure that an optimum does not lie past the previous window of epochs of 20, the neural network is trained for 580 epochs. Figure 6.14 shows the training curve generated as a function of the number of epochs.

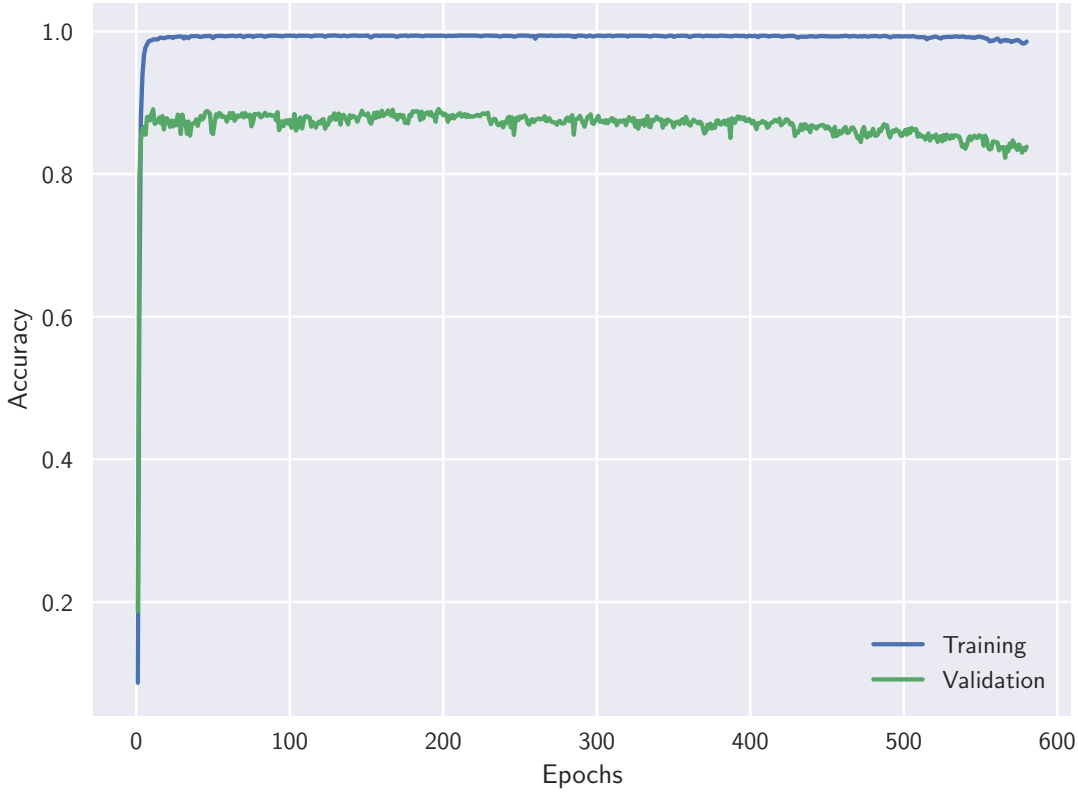


Figure 6.14: Network accuracy as a function of epochs.

As shown in this graph, the most useful epochs are the first ones since the validation accuracy worsens the more epochs the model is trained on. It is a clear sign of overfitting in this case. Thus, it can be deduced that as soon as the model starts to show significant signs of decline in accuracy or loss, there is no point in trying to force it to learn on more epochs as it has reached its maximum achievable performance.

6.1.7 Required number of samples

In order to guarantee efficient models, a minimum number of samples is required. This is useful since it can help quantify the amount of data needed in order to train a model that is good enough for deployment. This lower bound is somewhat dependent on the language, however that influence is insignificant in this case and can be safely ignored.

Figure 6.15 shows the training curves for a model trained on an English dataset consisting of 4,614 samples.

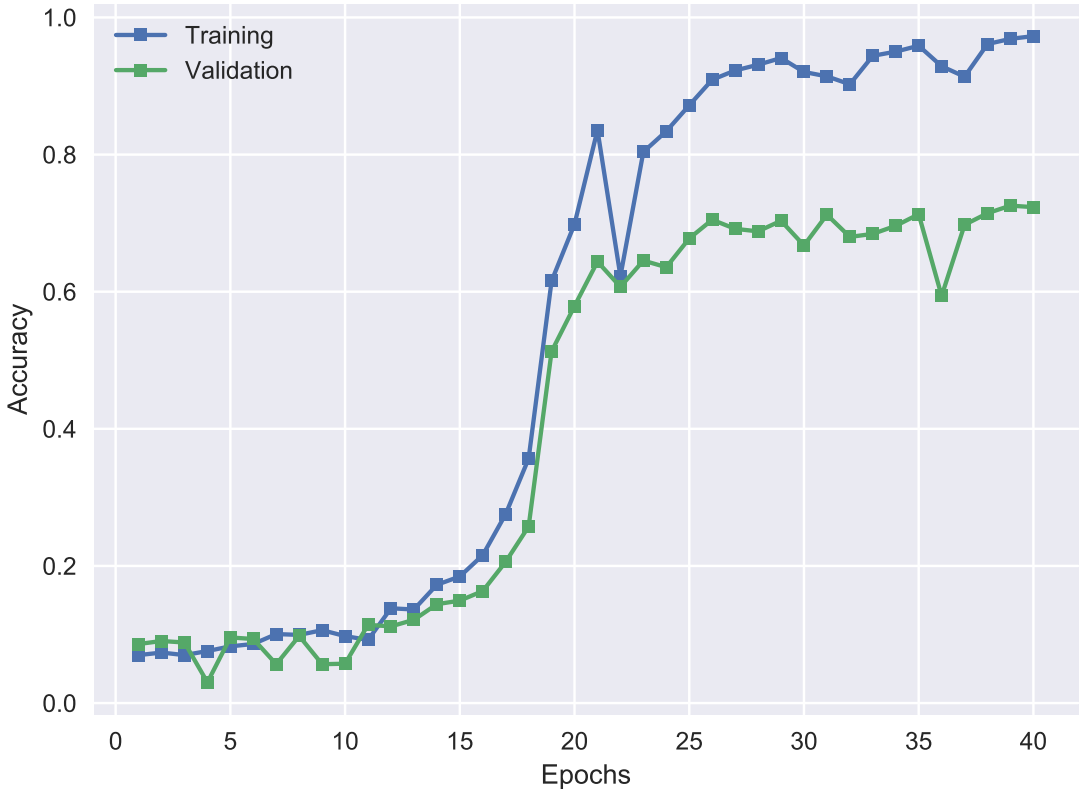


Figure 6.15: Network accuracy on an English dataset.

As can be seen, the model requires more epochs to reach the satisfying accuracy of 80% due to the relatively low number of samples contained in the dataset. In this particular case, the model needs to train on more than 40 epochs to achieve the required accuracy. However, despite the small amount of examples to train on, the model can still reach a satisfying validation accuracy if trained for long enough.

6.2 Human requester

This section will talk about the experiments conducted in order to analyse a potential model for a human assistance requester. This model should output a binary value which is 1 if the model wants to call a human for help, or a 0 if the chatbot thinks it should be able to handle the conversation. The model should therefore ideally saturate around those two values. This is translated directly in the model's architecture by using a tanh layer as last layer.

6.2.1 Dataset generation

The main idea is to take the output of the intent classifier (i.e. the vector of class probabilities) and use it as features to feed to this network. Therefore, the features are simply 7 floating point values $\in [0, 1]$. For the samples' label, the classes' labels from the dataset extracted in section 5.2 are simply converted by changing the

ones labelled `Other` into 1's and all other labels into 0's. Since the total amount of tickets is around 500,000 when including the `Other` class for French, a simple train-validation-test dataset division is used much like in section 6.1.

6.2.2 Training

Since the number of features is low for this classification problem, a basic neural network architecture is used. Figure 6.16 shows the simple structure adopted for the requester.

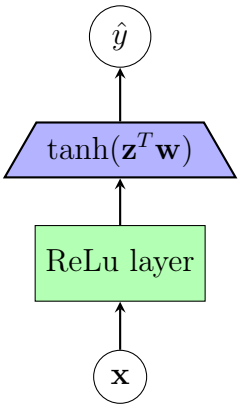


Figure 6.16: Requester's network structure.

As can be seen from the diagram, the features are sent through a layer of neurons with the ReLu activation function which is defined as :

$$ReLu(x) \triangleq max(0, x). \tag{6.4}$$

Then, this output is fed to a second layer of neurons using the tanh activation function.

The loss function used in this case is the binary cross-entropy which is a particular case of the categorical cross-entropy where $m = 2$. Taking into account the regularisation of the ReLu layer, the following loss function is optimised during training :

$$\mathcal{L}(\theta) \triangleq -\frac{1}{N}\sum_{j=1}^{N} y_j \log \hat{y}_j(\theta) + (1 - y_j)\log(1 - \hat{y}_j(\theta)) + \lambda \mathbf{w}^T\mathbf{w} \tag{6.5}$$

where N is the total number of samples, y_j is the true label of sample j, $\hat{y}_j$ is the predicted label for sample j by the network, and $\mathbf{w}$ is the weight matrix of the ReLu layer.

Once again, the training and analysis were performed only using a single language (in this case French since it contains the most tickets). Nevertheless, models were trained for other languages as well.

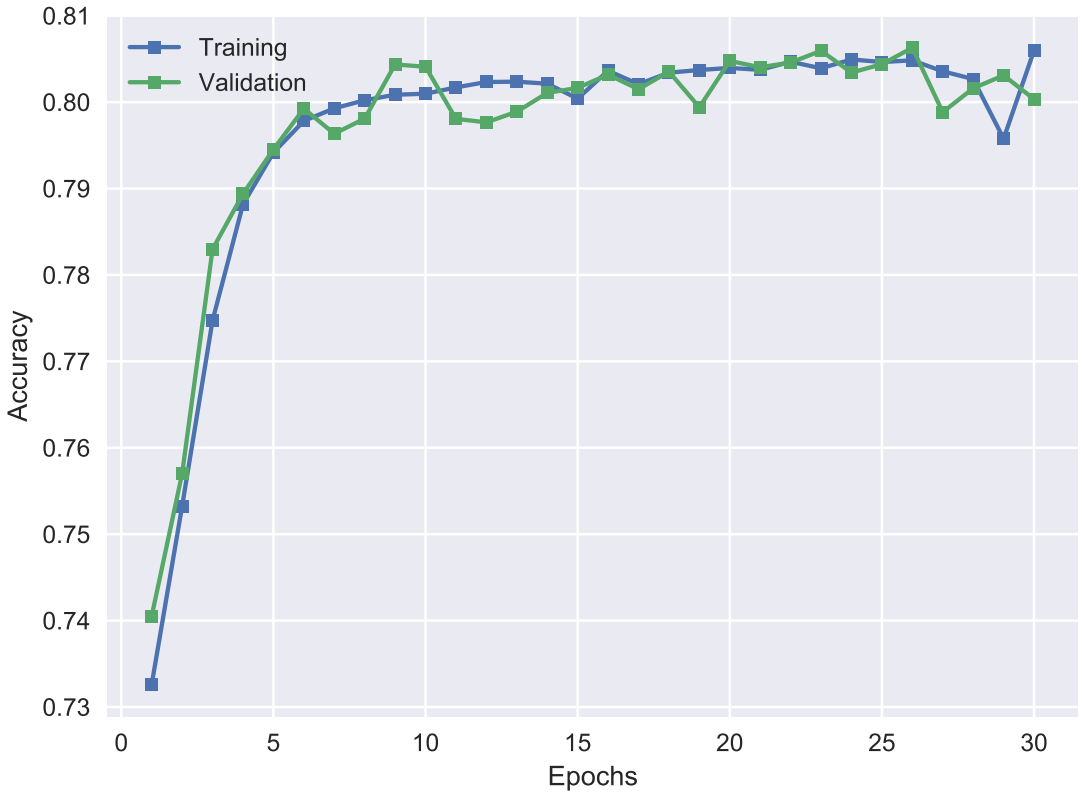


Figure 6.17: Requester accuracy for French.

As can be seen from Figure 6.17, the model reaches quickly its top accuracy of around 80.6%. The discrepancy between the training and validation loss is also relatively low as shown in Figure A.9 in Appendix A. A major positive observation is that due to the simple model architecture and the low number of features, the model takes a small amount of time to train : these curves only took 47 minutes to generate.

Overall the precision of the requester is sufficient. It is far from perfect but it might be due to the fact that some tickets are incorrectly labelled as `Other` . Indeed, older tickets did not have tags attached to them because the tags system was put in place after the first tickets were issued back in 2013. Therefore, it is possible that the model simply cannot reach a higher precision than 80%. Nevertheless, the learned requester will still be used for the rest of the developments.

6.3 Clarifying user intent

The particular task of clarifying user intent requires the chatbot to actively ask questions to the user or at least to give an opportunity to the user to clarify what he/she means. This creates a feedback from the chatbot to the user and conversely and enables a richer conversation.

6.3.1 Bayesian approach

The Bayesian approach consists in using a Bayesian update in order to infer the correct user intent. It is based on Bayes' rule :

$$P(c|m) = \frac{P(m|c)P(c)}{P(m)} \tag{6.6}$$

and in this context, c is a label of a user problem, and m is the message received from the user. Since the result of the update should be a vector of all class probabilities, the term $P(m)$ can simply be ignored as it only depends on m and serves only as a normalisation operation and is implemented as such. $P(c)$ is the prior probability and it is taken directly from the output of the intent classifier. It is assumed to be the first time the message is classified but that constraint is not strict. Finally, the term $P(m|c)$ is estimated using the intent classifier on the user's reply. The assumption is that the user when clarifying will tend to use a vocabulary similar to what other users might have said in their first message.

6.3.2 Comparison

Once the Bayesian update is implemented, a comparative analysis needs to be performed in order to show if it helps the chatbot determine the user's intent. 2 versions of the conversational agent were considered : a standard chatbot without Bayesian update and a chatbot with Bayesian update. Both are coupled with a user simulator that clarifies itself by sending a reply according to the sample's user intent. The user simulator attempts to clarify 3 times before giving up. If the chatbot has not understood the user by then, the sample is labelled as 'Failed to clarify'. The dataset used for this comparison is simply the test set obtained from Table 6.3 holds the distribution of tickets with the two variants.

Chatbot variant	Correctly identified w/out clarification	Identified after clarification	Failed to clarify	Incorrectly identified
Standard	16,508	0	0	4,899
Bayesian update	16,508	4,302	37	560

Table 6.3: Comparison of chatbot with and without Bayesian update.

As expected, the bayesian update helps the agent correctly address the problems that are unclear at first. As shown in the comparison, the Bayesian update almost clarifies all user problems in the test set. However the results should not be taken at face value since the clarifications generated by the user simulator are very optimistic and represent an ideal world where users are perfectly clear when clarifying their issues. If it were true, it means that the agent with a Bayesian update should be expected to achieve an accuracy of around 97% as shown here. This is of course

unrealistic but shows clearly that a Bayesian update is useful in determining a user intent.

Chapter 7

Results and performance

This chapter will assess the results of the completed system quantitatively and discuss its overall performance when running in a realistic environment.

7.1 Performance metrics

The following section is concerned about quantifying the quality of the chatbot's modules. Mainly, the intent classification subtask as well as the intervention requester subtask will be examined.

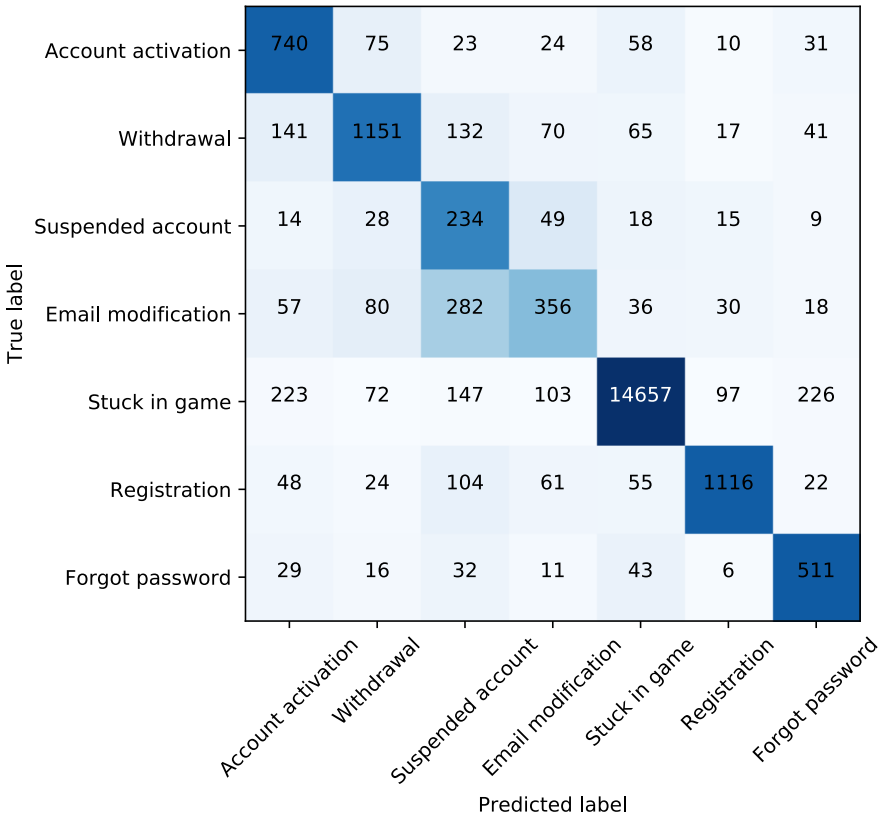


Figure 7.1: Confusion matrix for the French intent classifier.

7.1.1 Intent classification

For the intent classification, the test dataset was used to derive these final metrics. First, the confusion matrices of each intent classifier trained are derived. Figure 7.1 is the confusion matrix for French, Figure 7.2 for Dutch and Figure 7.3 for English.

The agent seems to correctly identify most French users' intents. However it seems to confuse some `Email modification` samples for the `Suspended account` class. This could be due to two things : either the information contained in the first message is not enough to distinguish accurately between the two classes, or either the samples in the dataset were not labelled correctly. This could be the case since some tags might overlap between the two classes for old enough tickets. Nevertheless, all the other classes are correctly predicted so the overall performance for French is satisfying.

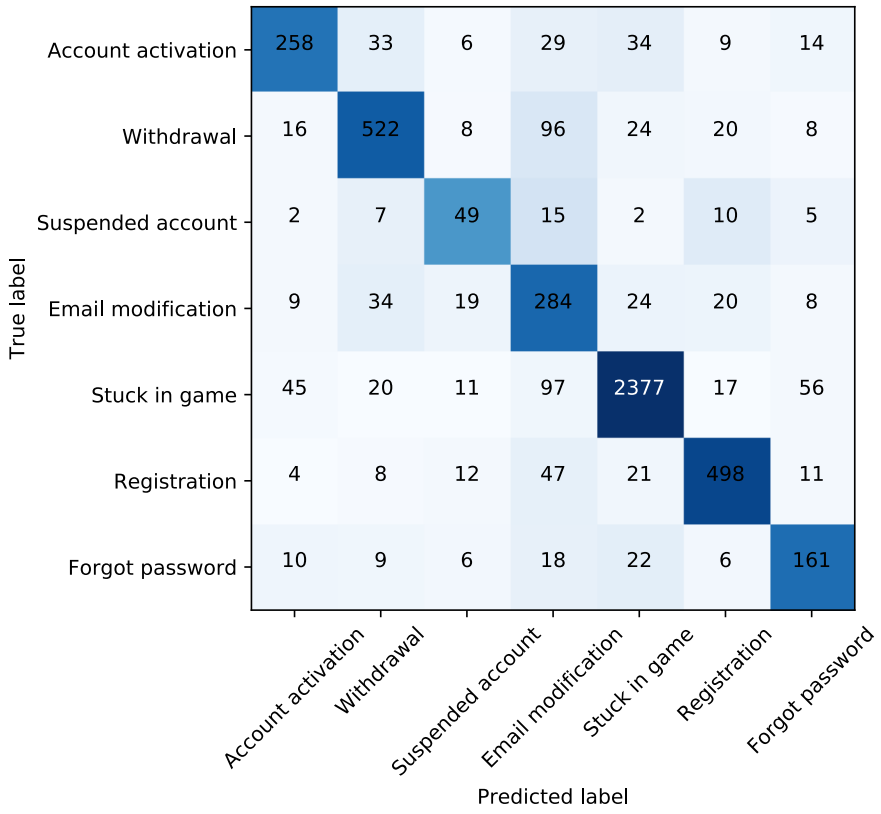


Figure 7.2: Confusion matrix for the Dutch intent classifier.

For Dutch, the majority of tickets are correctly classified. There are of course a small amount of outliers but the overall accuracy for each class is very satisfying.

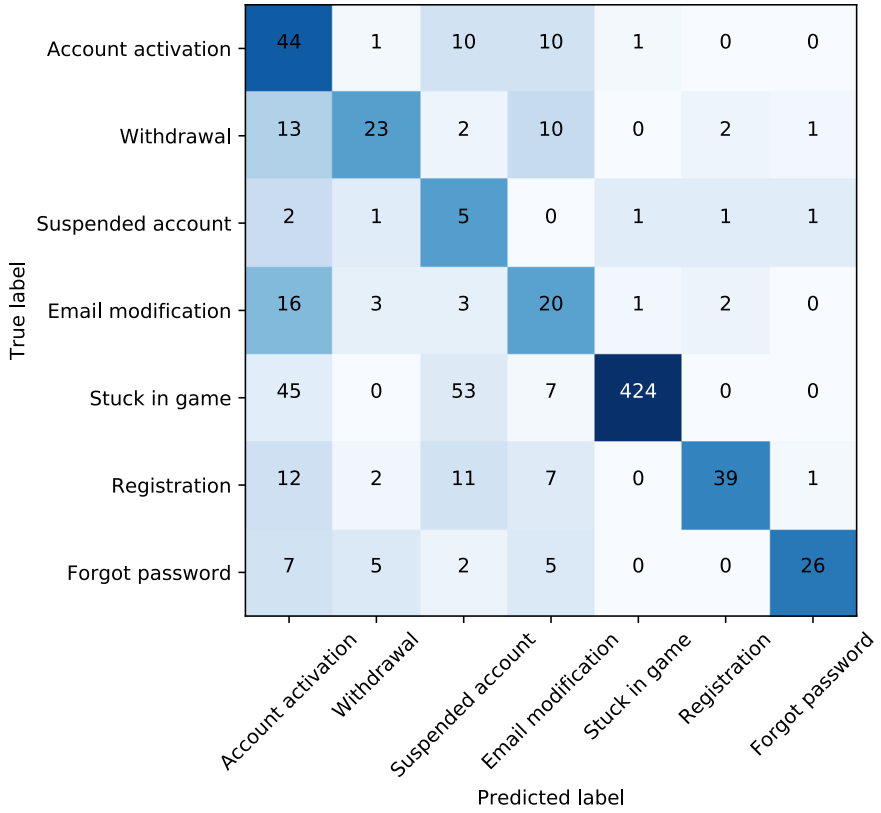


Figure 7.3: Confusion matrix for the English intent classifier.

For English, the results are less positive than the two other languages. There is a significant amount of confusion between the classes as can be seen from its confusion matrix. However it can be observed that the more samples there are in a class, the more likely it is to be correctly classified. Indeed, for instance the class `Stuck in game` is correctly predicted with an accuracy of 80.1% whereas the class `Suspended account` only reaches a prediction accuracy of 45.4%. All in all, this suggests that the English classifier is not robust enough to be used in a real customer service environment yet.

Finally, Table 7.1 summarises each classifier's final accuracy over the corresponding test dataset.

Language	Test accuracy
French	87.6%
Dutch	82.6%
English	70.9%

Table 7.1: Final accuracy on test set for each language

As expected, the English classifier shows a lower accuracy than the other classifiers. As a result, the English version of the chatbot might not be as efficient at determining a user's needs from a single message.

7.1.2 Intervention requester

Since the requester subtask is formulated as a binary classification problem, specific tools for assessing the performance of binary classifiers can be used. In this instance, the Receiver Operating Characteristic curve is generated for each language : Figure 7.4 represents French, Figure 7.5 Dutch, and finally Figure 7.6 represents English.

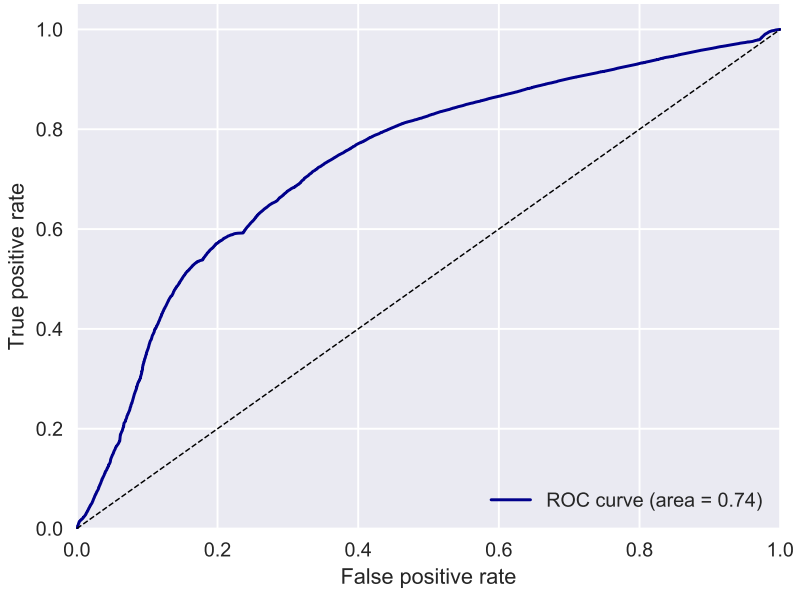


Figure 7.4: ROC curve of the final intervention requester for French.

For French, the ROC curve is promising and shows that the optimal threshold can be achieved with an expected true positive rate of approximately 80% and a corresponding false positive rate of 40%. The false positive rate can be even a bit higher since it just means that the model will be more cautious and request humans for more examples than necessary. Besides, it does not prevent the maintainers to add a failsafe feature that prevents the agent from requesting a human if the predicted class' probability is higher than a given threshold.

For Dutch, the ROC curve shows that the requester struggles more at determining whether or not the message does not belong to one of the problems the chatbot is expected to handle. Indeed, the AUC is slightly lower for Dutch than it is for French. Once again, the argument that this is not so much of a problem since the same heuristics of using a probability threshold for predictions can be made in this case.

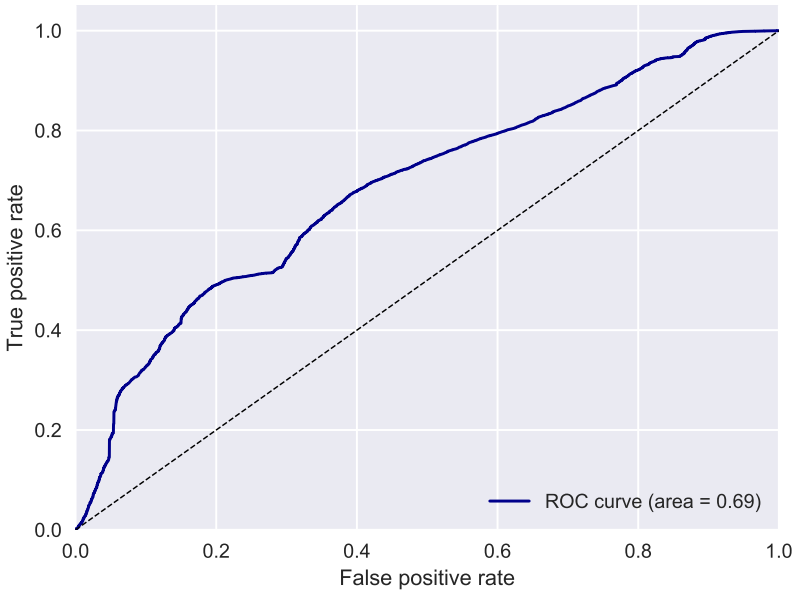


Figure 7.5: ROC curve of the final intervention requester for Dutch.

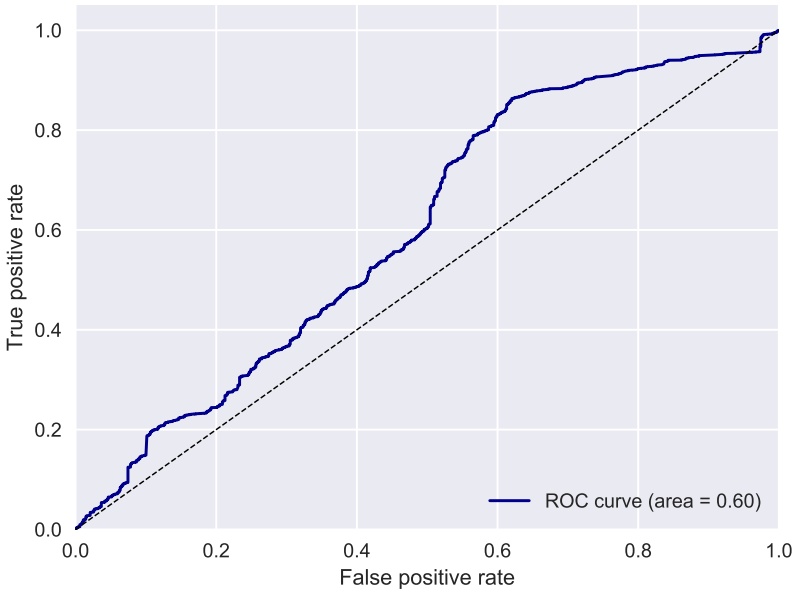


Figure 7.6: ROC curve of the final intervention requester for English.

Finally for English, the ROC curve is extremely unsatisfying. Indeed, the graph shows that in order to reach an acceptable true positive rate of 80%, the false positive rate needs to be as high as 60%. Furthermore, the AUC is only of 0.6, slightly higher than having a requester triggering at random. Therefore this suggests that the English requester be used only as last resort or that the chatbot should only reply to messages where its predicted user intent probability is very high.

7.2 Mockup environment test case

As stated previously, performance metrics do not tell the whole story when it comes to conversational agents [19]. Thus, in order to make sure the chatbot exhibits the right behaviour, a manual way of testing is devised from the ground up. The idea is to be able to input messages directly in a terminal or other keyboard input method much like a direct messaging app (be it a very primitive one in this instance). This test environment makes use of the interfaces introduced earlier in Figure 4.2 in Chapter 4. Specifically, the `BackendInterface` and `ActionExecutor` interfaces. Two mockup classes are developed for this test environment : `BackendSimulator` which inherits the `BackendInterface`, and `ConsoleExecutor` which inherits `ActionExecutor`. Each of these submodules will be explained in the following subsections and finally the manual test cases will be analysed as well.

7.2.1 Backend simulator

The simulator is simply implemented using a mockup database made of `.csv` files in order to allow quick and easy configuration for setting up particular user test cases. In this example, three particular tables are used :

- `users.csv`: contains the information concerning the users and their account's status.
- `games.csv`: contains information about the past and ongoing games.
- `withdrawals.csv`: contains information concerning withdrawals from games including bank account used for the transaction, the country where the bank account is hosted, etc...

7.2.2 Console executor

This derived action handler prints the planned action to the console but does not actively executes them in the backend. This ensures that test cases are reproducible. It also allows quick and easy debugging.

7.2.3 Manual testing

The following are transcripts of exchanges between the chatbot and the user in a mock environment. There is one example of successful interaction with the chatbot for each user problem. The first one is the `Account activation` in French.

USER : `Je n'ai pas reçu mon bonus d'activation.`

CHATBOT : `Bonjour User,`

`Votre compte n'est pas activé. Pour activer votre compte, veuillez`

cliquer sur le lien suivant : <activate account link>

Bien à vous,
Bot

The user states that they have not received their activation bonus. This is a common issue encountered for users when they have not activated their account as they need to do that before receiving their activation bonus. The bot simply replies by stating that the account is not activated yet and gives a link to activate the user's account.

Next is an example of interaction in Dutch about a very common issue : `Stuck in game`.

USER : Mijn credits zitten vast in het spel.

CHATBOT : (*Unblock user*) Goedendag User,

We hebben geprobeerd het probleem op te lossen. Controleer of het is opgelost aan uw kant.

Het bestje,
Bot

The user says that their credits are stuck in game. The bot immediately realises this is the case and in this scenario, it is assumed it is able to simply unblock the user. It then tells the user the problem has been solved and that they should check on their side if everything is solved as well.

Next example is about a French interaction for the `Withdrawal` user problem.

USER : Bonsoir, j'ai effectué une demande de retrait il y a quelques jours. Je n'ai toujours rien reçu, est-ce normal ? Merci et bonne soirée..

CHATBOT : Bonjour User,

Nous remarquons que votre retrait n'a pas été versé sur un compte Belge. Il est assez courant que le versement de retraits prenne plus de temps pour des comptes à l'étranger (De l'ordre de 2 à 3 jours). Ne vous inquiétez donc pas, le versement devrait être effectué d'ici peu.

Bien à vous,
Bot

In this scenario, the user simply says that they have issued a withdrawal a few days ago and they received no further confirmation since then. In this particular instance, the user logged in has a withdrawal pending because it is supposed to be withdrawn to a foreign account. Therefore, the chatbot replies stating that it is normal to observe a small delay for foreign accounts and reassures the user that it should be done shortly.

Another example is a Dutch interaction for the `Email modification` issue.

USER : `Hallo, ik wil mijn email veranderen alstublieft.`

CHATBOT : `Goedendag User,`

> `Geef ons een kopie van uw identiteitskaart en uw nieuwe e-mailadres om uw e-mailadres te wijzigen.`
>
> `Het bestje,`
> `Bot`

The user simply asks to change their e-mail. The chatbot then asks for a copy of the user's ID card and new e-mail address according to procedure, as expected.

Next up is the `Registration` problem addressed in French in this instance.

USER : `Je voulais m'inscrire sur votre site mais l'inscription est refusée. Merci !`

CHATBOT : `Bonjour User,`

> `Afin de résoudre votre problème d'inscription, veuillez nous fournir une copie de votre carte d'identité ainsi que vos coordonnées exactes (adresse e-mail en particulier).`
>
> `Bien à vous,`
> `Bot`

Here, the users tells the agent that their registration has been refused. The chatbot correctly identifies the problem and asks for a copy of their ID card as well their coordinates and especially their e-mail address.

The final example is about the `Forgot password` user intent and it is shown here in English.

USER : `I forgot my password and can't login.`

CHATBOT : `Hello User,`

```
Your password has been successfully reset, your new password is :
<password>

All the best,
Bot
```

As expected, the chatbot correctly identifies the user's issue and resets the user's password and generates a new one for him[1].

An interesting property to test is the robustness of the conversational agent to the way messages are written. Therefore, this last example is shown with abbreviations for *password* and *please* :

USER : `i forgot my pswd pls help.`

CHATBOT : `Hello User,`

```
We do not understand your request.  Could you try to be a bit clearer?

All the best,
Bot
```

USER : `idk which pswd i put i cant login.`

CHATBOT : `Your password has been successfully reset, your new password is :  <password>`

```
All the best,
Bot
```

As we can see, the message is correctly understood by the chatbot after a Bayesian update and clarification from the user even though most keywords are abbreviated.

Overall, despite the performance metrics not being as good as they should be, the chatbot is still able to correctly handle concrete user test cases in practice. It even displays a somewhat robustness towards spelling errors and abbreviations. These results are more than satisfying especially for the English version due to the small number of samples and the pessimistic performance metrics.

[1]This behaviour is not final and can be changed in the action planner.

Chapter 8

Conclusion

This chapter will summarise the content of all previous sections in order to recap what has been accomplished. It will give a global view of the completed system as well as provide pointers for future developments.

8.1 Summary

Starting from a subjective definition of the chatbot's goals, a set of user problems and constraints were laid down on paper in Chapter 3. Then in Chapter 4, a scalable software solution was designed and developed for training and deploying chatbots. This solution makes it possible to extract ticket data dynamically from an existing Zendesk environment and label them using tags attached to the tickets as shown in Chapter 5. Moreover, the chatbot concept was decomposed into several modules which solve a particular design problem. Of those modules, some of them make use of neural network models. These models' structure and property were analysed extensively in Chapter 6. A way for the chatbot to allow users to clarify themselves was also devised in that chapter, derived from Bayes' rule. Finally, the final results from the chatbot were shown in Chapter 7. In particular, adequate performance metrics were measured for each neural network generated beforehand and a manual way of testing the chatbot was presented.

Looking back at the accomplished work, it can be said that the solution developed is satisfying and allows for easy deployment and development if changes need to be performed. It can be ported on virtually any system provided it can host a Python environment. Furthermore, it leaves the door open for additional languages' support and user problems. It is not a perfect solution but considering the requirements, it works well in practice. There might be a few pathological cases where the chatbot might not behave in the best manner possible so the future system maintainers need to be aware of that.

8.2 Possible improvements

Since this solution is not a panacea, several improvements could be made in order to extend the realm of possibilities for the agent. The following list should be taken lightly however as no extensive studies have been conducted to confirm they are indeed feasible in this case. They are intended more as brainstorming than an actual roadmap to follow for the future of the chatbot.

- Additional languages could be trained in order to cover a wider user base.
- More challenging user problems could be investigated. In particular, computer vision algorithms could be trained to extract information from photos users send of their ID cards, credit cards, etc...
- A different dynamic version of the chatbot could be designed. This variant would need to be more reactive than the variant presented in this document by displaying more small-talk capabilities and creating more direct and customised interactions with customers. This variant could then be deployed to Facebook and other instant messaging services that support the use of bots.
- From time to time, users have multiple problems that they would like to address in a single message. This changes the formulation of the intent classification problem into a multi-label classification problem. Different methods could be considered to address this issue.
- If the user is unable to clarify itself, instead of directly requesting a human operator, the chatbot could suggest potential problems to the user in the form of a selection list with common problems. There could also be the option to request a human agent in this list to prevent users from getting frustrated with the bot.

8.3 Closing thoughts

Chatbots had a lot of weight put on their shoulders, labelled as "The New Apps". There has been a significant trend in 2016 with chatbots, a trend which has faded for now but is still lingering in the background. However there is still a long way to go for conversational agents before they can be used completely unsupervised. Nevertheless, the specific domain of customer service is very prone to chatbot use and is already very systematic as shown previously. The future holds promising results especially in the field of Natural Language Processing: new methods are born every day and older methods become even better. The growth of chatbots left a mark in History and in the minds of consumers. Besides, it might entirely be possible that, in a near future, the roles might be reversed between humans and bots [29]. To go further with that thought, we might see one day bots interacting with other bots using human language as means of communication to improve the quality of life for

mankind. As a matter of fact, these subjects already being investigated with mixed results [30]. However, I remain optimistic for these types of technology as they will shape the future of our societies. Maybe in a few years, these technologies will finally be undistinguishable from humans. We need therefore to be careful about the way researchers tackle these issues to make sure they do not end up in the wrong hands.

Appendix A

Experiments

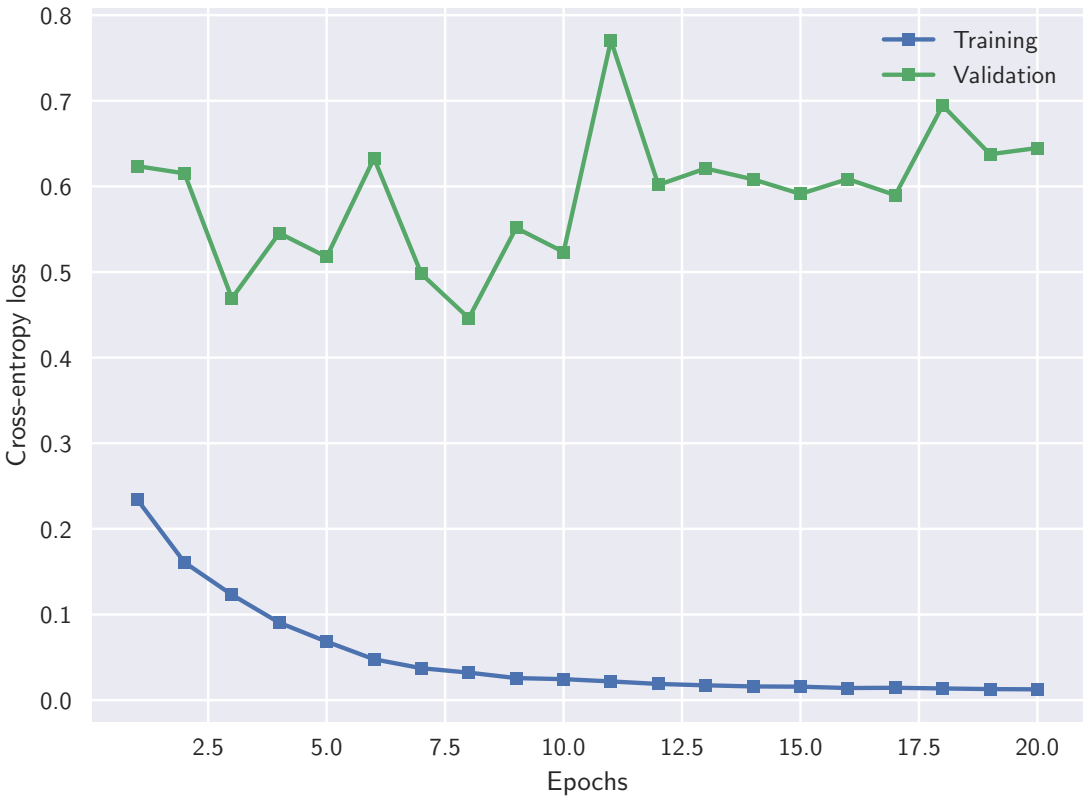


Figure A.1: Single LSTM layer network's loss.

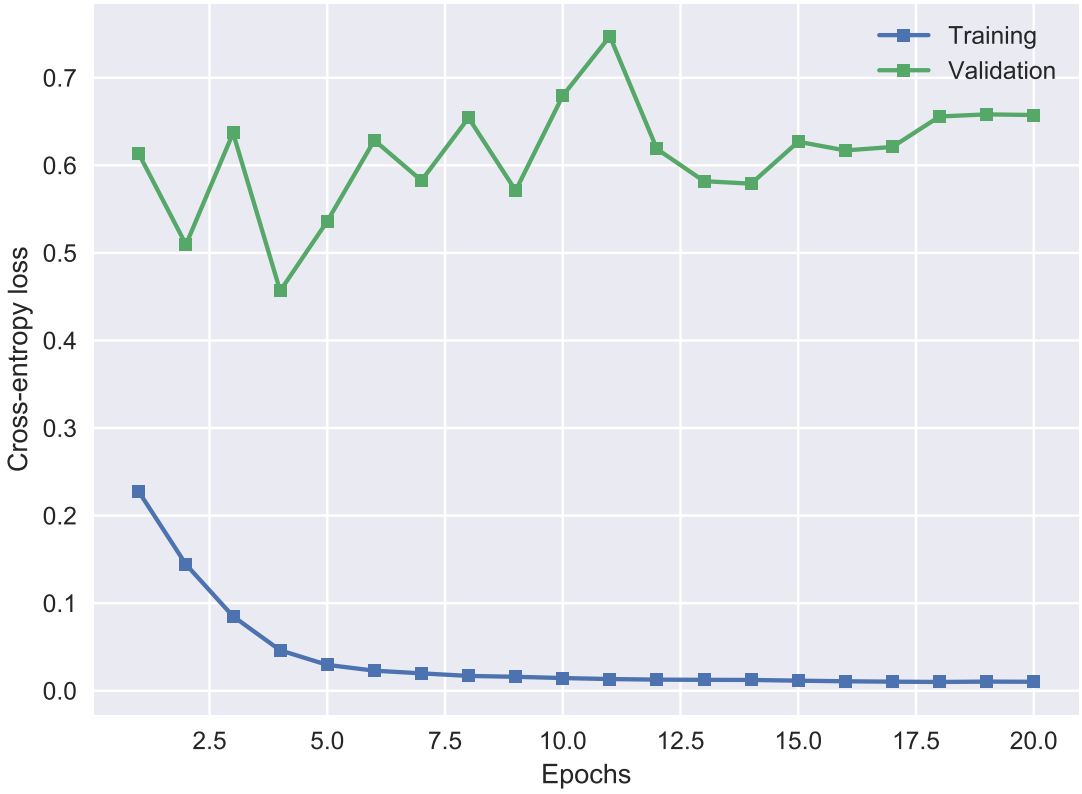


Figure A.2: Single GRU layer network's loss.

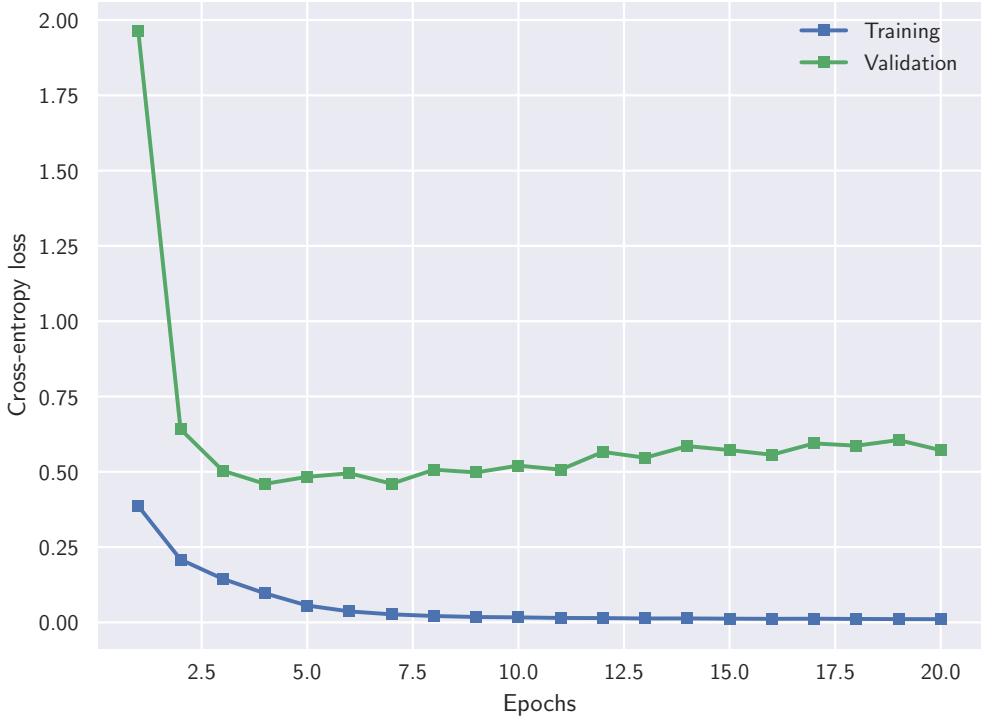


Figure A.3: Single GRU layer network with inverted input sequence's loss.

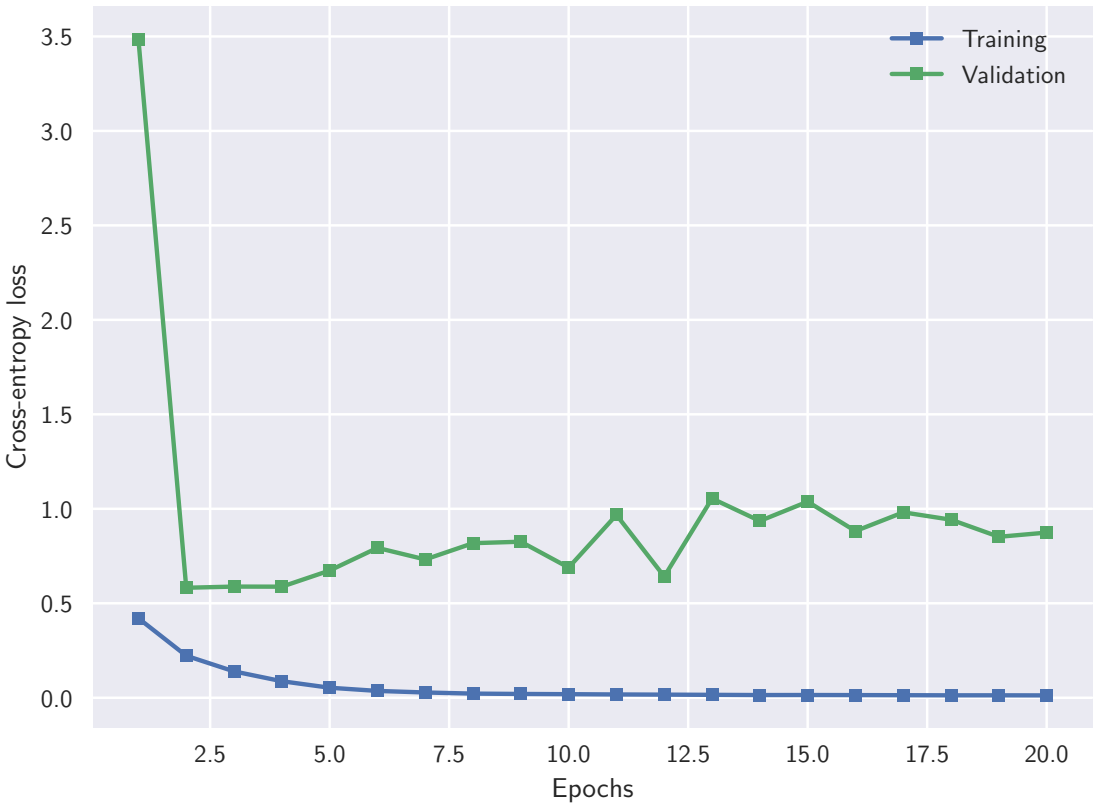


Figure A.4: Network with two GRU layers' loss.

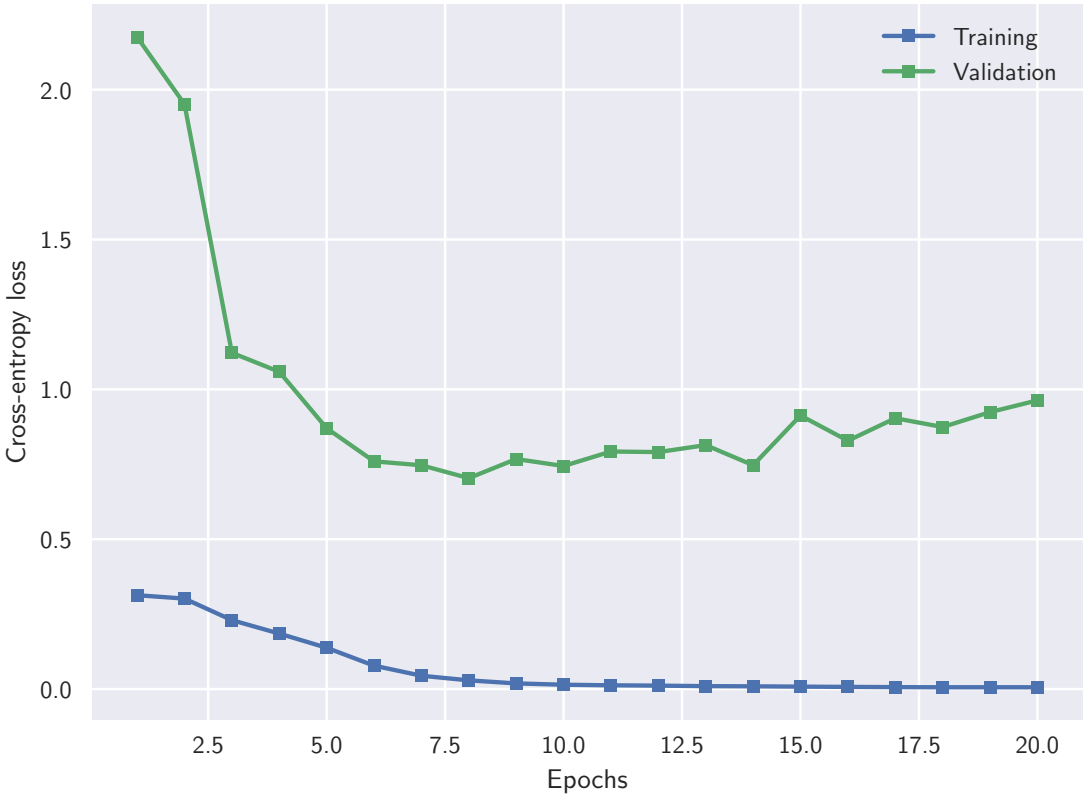


Figure A.5: Network loss for Dutch.

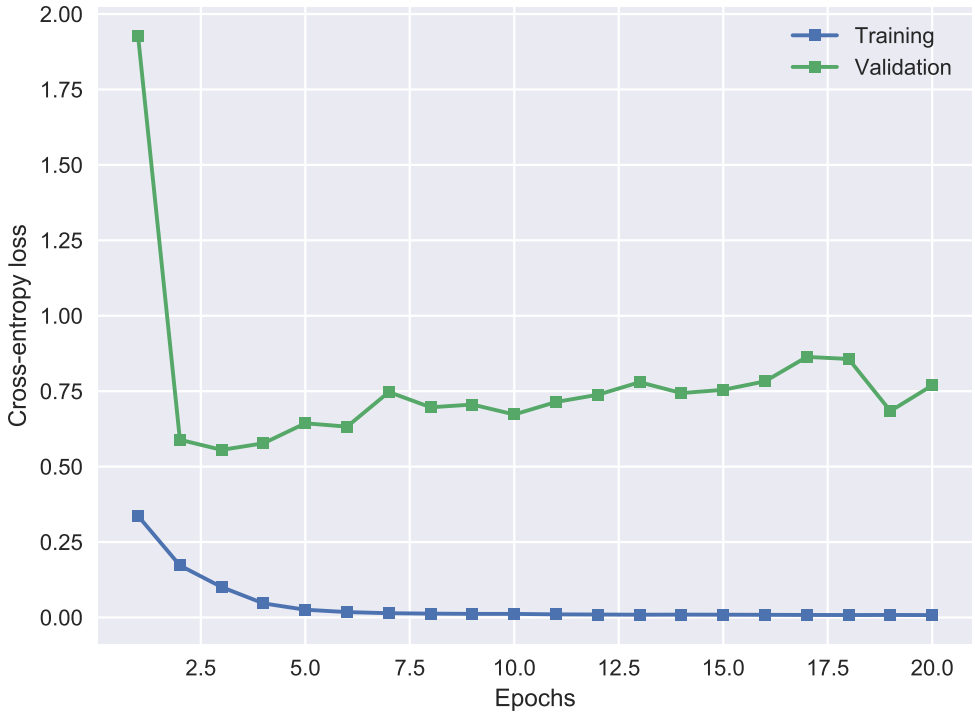


Figure A.6: Network loss for a bilingual dataset.

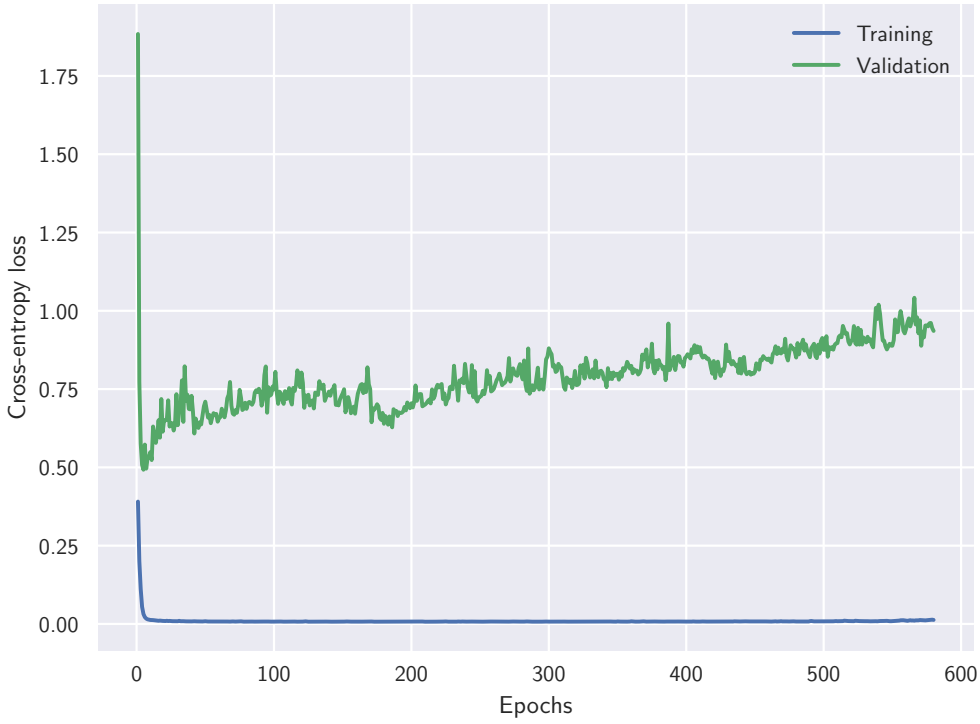


Figure A.7: Network loss as functions of epochs on a French dataset.

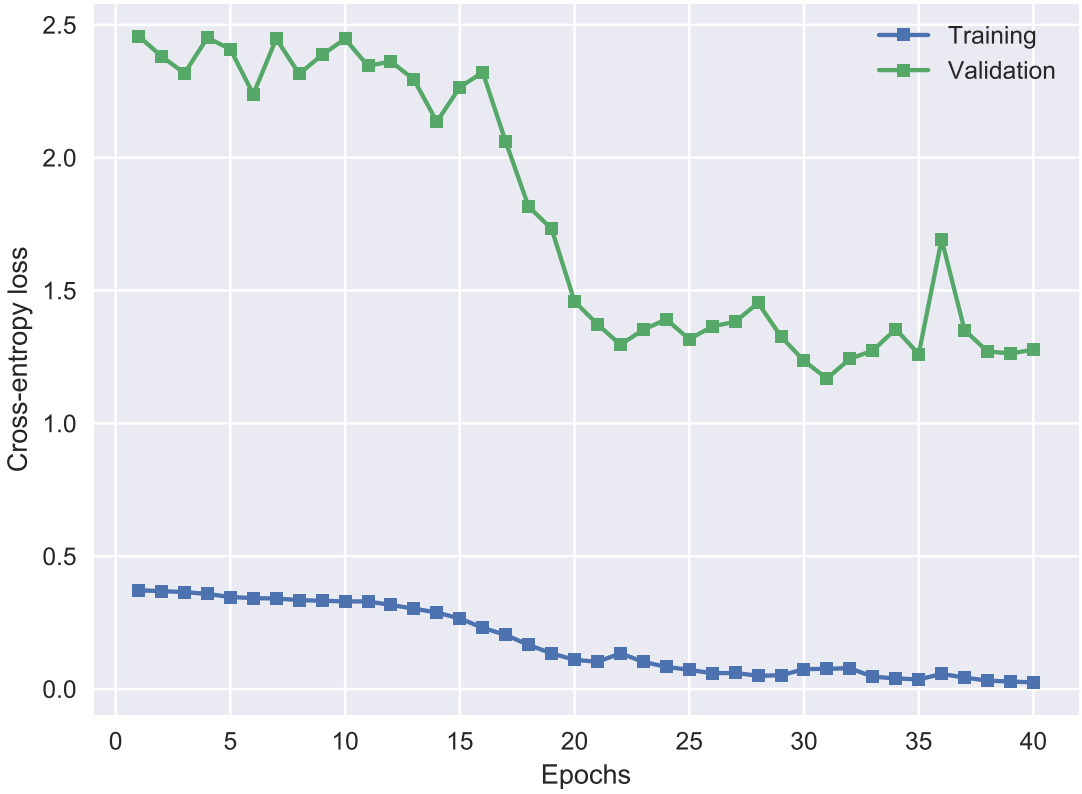


Figure A.8: Network loss on an English dataset.

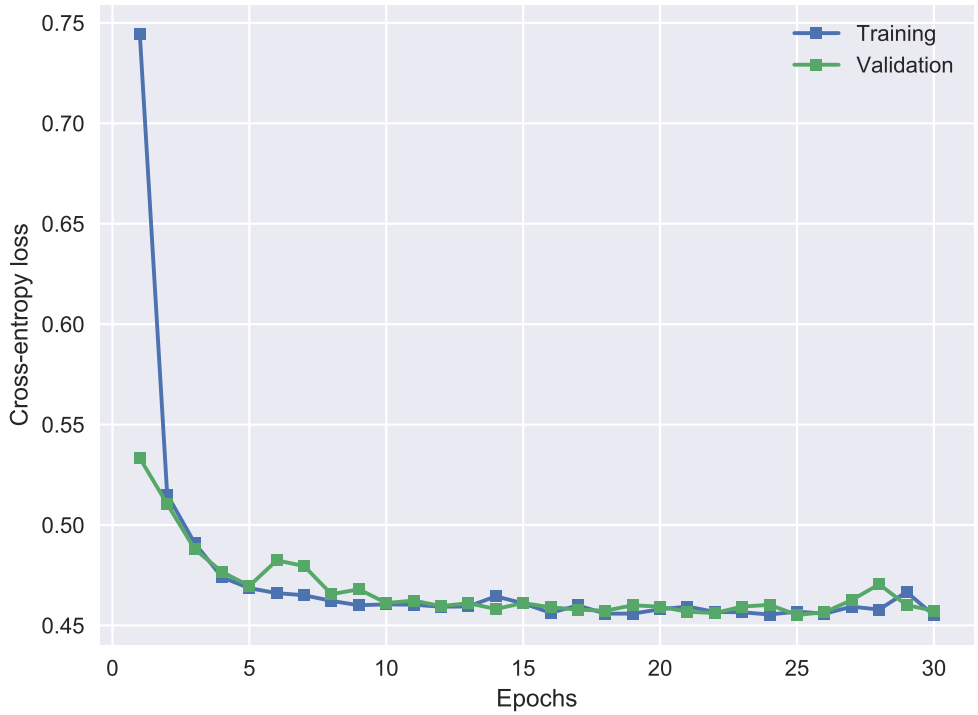


Figure A.9: Requester loss for French.

References

[1] A. M. Turing. "Computing Machinery and Intelligence". In: *Mind* 49 (1950), pp. 433–460.

[2] J. Weizenbaum. "ELIZA—a computer program for the study of natural language communication between man and machine". In: *Communications of the ACM* 9 (Jan. 1966), pp. 36–45.

[3] Drift. *The 2018 State of Chatbots Report*. Jan. 2018. URL: https://blog.drift.com/chatbots-report.

[4] Chatbots.org. *Consumers Say No to Chatbot Silos in US and UK Survey*. 2017. URL: https://www.chatbots.org/images/news/chatbot_survey_2018.pdf.

[5] T. Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *CoRR* abs/1301.3781 (2013). arXiv: 1301.3781. URL: http://arxiv.org/abs/1301.3781.

[6] J. Pennington, R. Socher, and C. D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[7] R. Pascanu, T. Mikolov, and Y. Bengio. "Understanding the exploding gradient problem". In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[8] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[9] I. Sutskever, O. Vinyals, and Q. V. Le. "Sequence to Sequence Learning with Neural Networks". In: *CoRR* abs/1409.3215 (2014). arXiv: 1409.3215. URL: http://arxiv.org/abs/1409.3215.

[10] O. Vinyals et al. "Show and Tell: A Neural Image Caption Generator". In: *CoRR* abs/1411.4555 (2014). arXiv: 1411.4555. URL: http://arxiv.org/abs/1411.4555.

[11] C. Olah. *Understanding LSTM Networks*. Aug. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[12] K. Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[13] J. Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555.

[14] M. Lui, J. H. Lau, and T. Baldwin. "Automatic Detection and Language Identification of Multilingual Documents". In: *Transactions of the Association for Computational Linguistics* 2 (2014), pp. 27–40. ISSN: 2307-387X. URL: https://transacl.org/ojs/index.php/tacl/article/view/86.

[15] L. Meng and M. Huang. "Dialogue Intent Classification with Long Short-Term Memory Networks". In: *Natural Language Processing and Chinese Computing*. Ed. by X. Huang et al. Cham: Springer International Publishing, 2018, pp. 42–50. ISBN: 978-3-319-73618-1.

[16] K. Lee et al. "Conversational Knowledge Teaching Agent that uses a Knowledge Base". In: (Jan. 2015), pp. 139–143.

[17] S. Perez. *Microsoft silences its new A.I. bot Tay, after Twitter users teach it racism*. 2016. URL: https://techcrunch.com/2016/03/24/microsoft-silences-its-new-a-i-bot-tay-after-twitter-users-teach-it-racism/.

[18] A. Tammewar et al. "Production Ready Chatbots: Generate if not Retrieve". In: *CoRR* abs/1711.09684 (2017). arXiv: 1711.09684. URL: http://arxiv.org/abs/1711.09684.

[19] C. Liu et al. "How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation". In: *CoRR* abs/1603.08023 (2016). arXiv: 1603.08023. URL: http://arxiv.org/abs/1603.08023.

[20] K. Papineni et al. "BLEU: a Method for Automatic Evaluation of Machine Translation". In: 2002, pp. 311–318.

[21] C.-Y. Lin. "ROUGE: A Package for Automatic Evaluation of summaries". In: *Proc. ACL workshop on Text Summarization Branches Out*. 2004, p. 10. URL: http://research.microsoft.com/~cyl/download/papers/WAS2004.pdf.

[22] M. Svane. *Zendesk*. URL: https://www.zendesk.com.

[23] F. Chollet et al. *Keras*. 2015. URL: https://keras.io.

[24] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[25] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[26] W. McKinney. *pandas: a Foundational Python Library for Data Analysis and Statistics*. URL: https://pandas.pydata.org.

[27] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[28] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[29] C. Welch. *Google just gave a stunning demo of Assistant making an actual phone call*. 2018. URL: https://www.theverge.com/2018/5/8/17332070/google-assistant-makes-phone-call-demo-duplex-io-2018.

[30] M. Field. *Facebook shuts down robots after they invent their own language*. 2017. URL: https://www.telegraph.co.uk/technology/2017/08/01/facebook-shuts-robots-invent-language/.