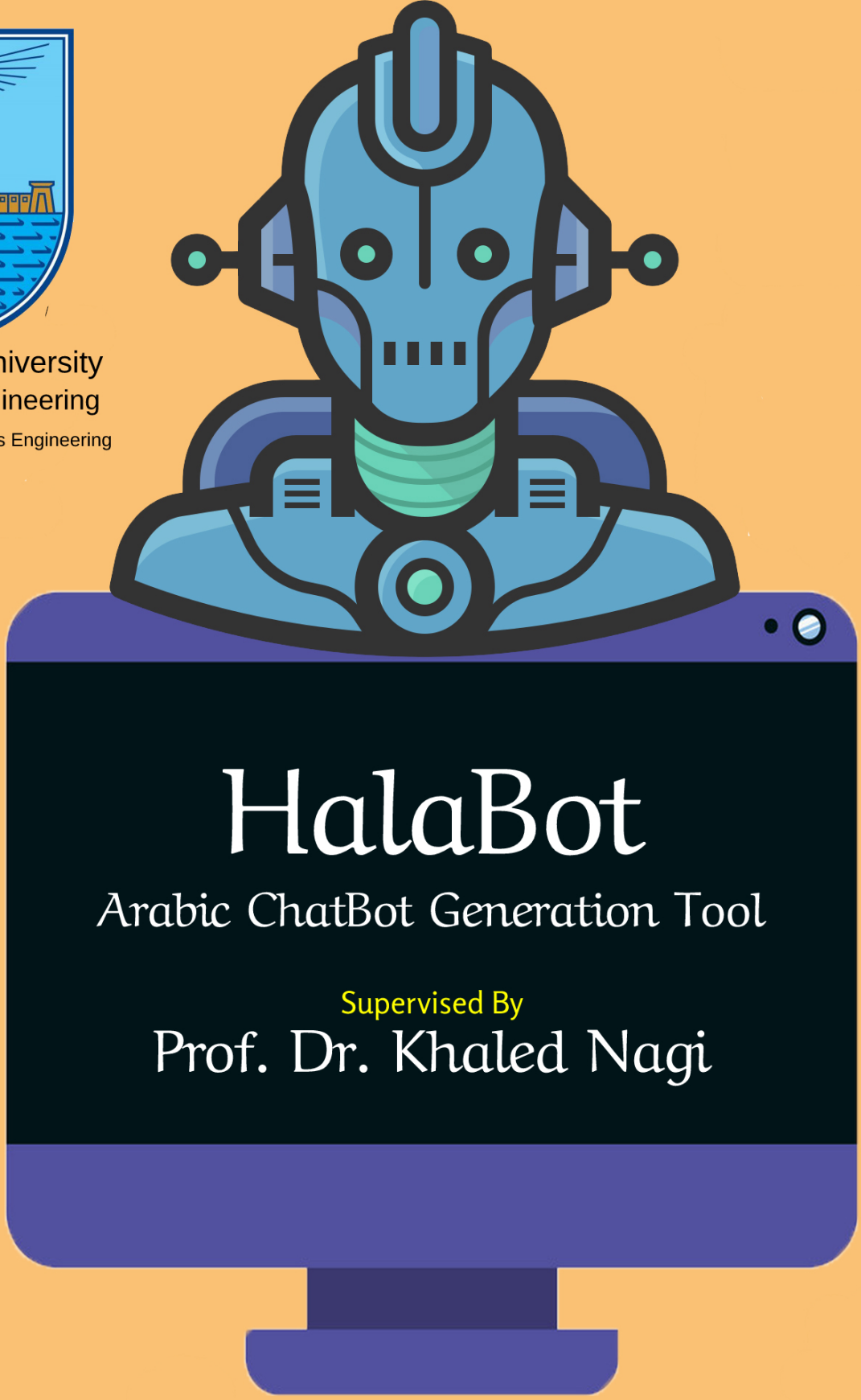
Alexandria University
Faculty Of Engineering
Computer And Systems Engineering

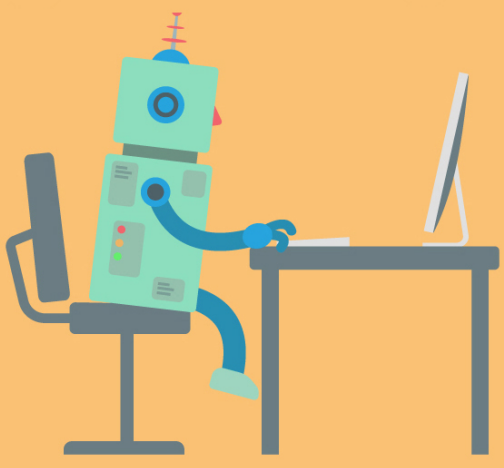


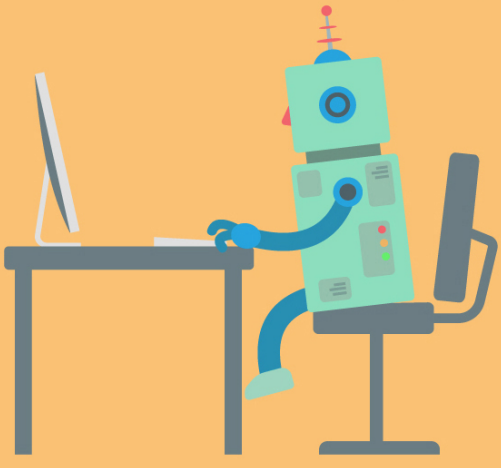
HalaBot

Arabic ChatBot Generation Tool

Supervised By

Prof. Dr. Khaled Nagi





Project Members

Fares Othman Taha Mehanna

Hesham Ibrahim Mahmoud El-Sawaf

Marwan Elsayed Abdelaal Tammam

Moustafa Mahmoud

Acknowledgements

Firstly, we would like to express our sincere gratitude to our advisor Prof. Dr. Khaled Nagi for his continuous support, patience, motivation, and immense knowledge. His guidance helped us in all phases of this project and writing of this report. We could not have imagined having a better advisor and mentor.

Moreover, we thank our colleagues for their stimulating discussions, the sleepless nights when working together before deadlines, and all the fun we have had in the last five years. We also thank all our friends, TAs and professors in the Faculty of Engineering.

Last but not the least, we would like to thank our families, our parents and our siblings for supporting us spiritually throughout this project and our lives in general.

We hope that this tool will offer a breakthrough in the field of replacing government departments with computers to ease citizens' extraction of official papers and official transactions.

Abstract

Chatbots, or conversational interfaces as they are also known, present a new way for people to interact with computer systems. Traditionally, to get a question answered by a software program involved using a search engine, or filling out a form. A chatbot allows a user to simply ask questions in the same way that they would address a human. The most well-known chatbots now are voice chatbots: Alexa and Siri. However, chatbots are being adopted at a high rate on computer chat platforms.

The technology at the core of the rise of the chatbot is natural language processing (NLP). Recent advances in machine learning have greatly improved the accuracy and effectiveness of natural language processing, making chatbots a viable option for many organizations. This improvement in NLP is firing a great deal of more research which should lead to continued improvement in the effectiveness of chatbots in the years to come.

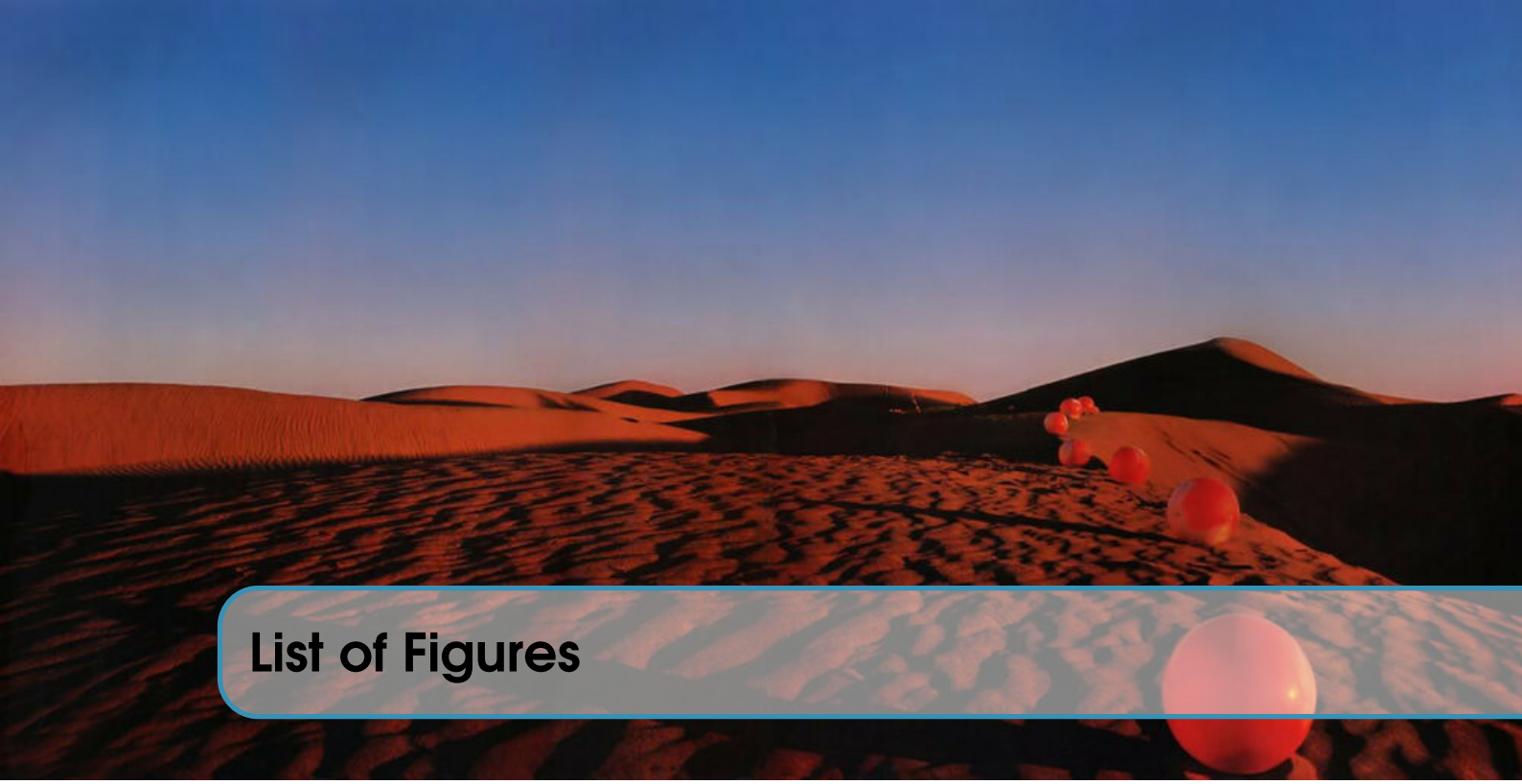
A simple chatbot can be created by loading a FAQ (frequently asked questions) into chatbot software. The functionality of the chatbot can be improved by integrating it into the organizations enterprise software, allowing more personal questions to be answered, likeWhat is my balance?or What is the status of my order?.

Most commercial chatbots are dependent on platforms created by the technology giants for their natural language processing. These include Amazon Lex, Microsoft Cognitive Services, Google Cloud Natural Language API, Facebook DeepText, and IBM Watson. Platforms where chatbots are deployed include Facebook Messenger, Skype, and Slack, among many others.

Using arabic language in chatbots is a challenge as we dont have enough arabic data to build good and stable models for arabic and most of the models currently used are based on transfer learning technique.

We aim to offer a reliable, robust and stable arabic chatbot that can be used in organizations to help users without involving humans in the process to reduce errors in replies, increase throughput and decrease time of waiting in queues.

Contents



List of Figures

List of Tables

Part One

1. Introduction

1.1 Problem Definition

The internet has taken part of everything around us. In developed countries you can order food, do bank transaction, buy online and more by internet. The internet also replaced traditional elections systems and even more complicated things.

Involving smartphones and IOT have widened the circle of usage. Using internet raises a problem for some users who cant deal with the internet and want to ask some questions. Like using the website to transfer amount of money from account to another account, the user here wants to ask some questions about the operation as it is something critical. Organizations using these websites have to assign custom services to help these people, but this is a temporary solution as some other problems raises like: affording more people working, delays in the queues to get help, help is not available 24/7, help should be provided in more than one language for multinational organizations and some ethical problems raises from these conversations.

We need more to computerize help process to avoid these problems so chatbots were provided to replace humans solving most of these problems. Using this technique for people who use Arabic is a challenge and our goal is to provide a robust chatbot that can be trained using FAQ (Frequently Asked Questions) and their related replies.

1.2 Solution

We want to introduce a stack of NLP pipelines to help build and deploy ready-to-use Arabic chatbots without th need to get involoved in the actual development process or have any prior knowledge in programming whatsoever.

We will use RASA [2] which is a framework used to build contextual AI assistants and chatbots in text and voice with open source machine learning framework alongside Farasa (which means "insight" in Arabic), a fast and accurate text processing toolkit for Arabic text.

The problem of understanding context and provide a suitable reply can be decomposed to a pattern recognition problem and RASA provide a solution to this problem using neural networks but RASA is used for english language only.

RASA can work with external pipelines for NLU part, so we can use word embeddings technique with a good lemmatizer to solve the problem.

The bot can be trained on a provided FAQ data and can be deployed for commercial applications. This work should be wrapped in a tool so that anyone can use it without bothering about how to train, launch and deploy it.

1.3 Importance

Using chatbots in replacement to humans will achieve the following:

1. Avoid outlay of custom services
2. Avoid delays in queues to get some help
3. Avoid ethical problems happening in chat (sexual harassment, racism, religious conversations, nudity, Etc.)
4. One server that can understand and reply in many languages
5. The global Chatbot market is expected to grow exponentially between 2016 - 2023
6. 85% of customer interactions will be managed without a human by 2020. - Gartner

1.4 Related Work - Intelligent Virtual Assistant

Intelligent Virtual Assistant (IVA) is an application that utilization information, for example, the users voiceand logical data to give help by noting inquiries in normal dialect, making suggestions and performing activities. Inside the writing the term IVAis utilized conversely with terms, for example, Conversational Agents, Virtual Personal Assistants, Personal Digital Assistants, Voice-Enabled Assistants or Voice Activated Personal Assistants, to give a few illustrations. IVAs join talk afrmation lingo understanding, trade organization, tongue age and talk association to respond to clients request and sales. Voice engaged IVAs like Siri, Google Assistant, Microsoft Cortana and Amazon Alexa are by and large open on cutting edge cell phones, and continuously in homes (e.g. Amazon Echo and Google Home) and automobiles (e.g. Google Assistant blend with Hyundai). The market for IPAs is foreseen to reach 4.61 billion by the mid 2020s.

The specialized foundations that empower IVAs have progressed quickly lately and have been the subject of broad research. Be that as it may, investigate concentrated on understanding the users experience of IVAs is more restricted. Obviously, from ourpoint of view, this has had suggestions for the selection and utilization of IVAs. For instance, in spite of their across the board and generally advanced consideration on cellphones individuals tend to utilize IVAs once in a while or not in the least. A current review x demonstrated that 98% of iPhone clients had utilized Siri previously. However just 30% used it routinely, with 70% utilizing it once in a while or just sporadically[20]

1.4.1 Google Assistant

Google Assistant[7] is an artificial intelligence-powered virtual assistant developed by Google that is primarily available on mobile and smart home devices. Google Assistant can engage in two-way conversations.

Assistant initially debuted in May 2016 as part of Google's messaging app Allo, and its voice activated speaker Google Home. It was released as a standalone app on the iOS operating system in May 2017. Alongside the announcement of a software development kit in April 2017, the Assistant has been, and is being, further extended to support a large variety of devices, including cars and third party smart home appliances. The functionality of the Assistant can also be enhanced by third-party developers.

Users primarily interact with Google Assistant through natural voice, though keyboard input is also supported. In the same nature and manner as Google Now, the Assistant is able to search the Internet, schedule events and alarms, adjust hardware settings on the user's device, and show

information from the user's Google account. Google has also announced that the Assistant will be able to identify objects and gather visual information through the device's camera, and support purchasing products and sending money, as well as identifying songs.

1.4.2 Cortana

Cortana[12] is a virtual assistant created by Microsoft for Windows 10, Windows 10 Mobile, Windows Phone 8.1 Invoke smart speaker, Microsoft Band, Surface Headphones, Xbox One, iOS, Android, Windows Mixed Reality, and Amazon Alexa.

Cortana can set reminders, recognize natural voice without the requirement for keyboard input, and answer questions using information from the Bing search engine.

Cortana is currently available in English, Portuguese, French, German, Italian, Spanish, Chinese, and Japanese language editions, depending on the software platform and region in which it is used.

Cortana was demonstrated for the first time at the Microsoft BUILD Developer Conference (April24, 2014) in San Francisco. It has been launched as a key ingredient of Microsoft's planned "makeover" of the future operating systems for Windows Phone and Windows.

1.4.3 Siri

Siri[5] is a virtual assistant that is part of Apple Inc.'s iOS, iPad OS, watch OS, mac OS, tv OS and audio OS operating systems.The assistant uses voice queries and a natural-language user interface to answer questions, make recommendations, and perform actions by delegating requests to a set of Internet services. The software adapts to users' individual language usages, searches, and preferences, with continuing use. Returned results are individualized.

Siri is a spin-off from a project originally developed by SRI International Artificial Intelligence Center. Its speech recognition engine was provided by Nuance Communications, and Siri uses advanced machine learning technologies to function. Its original American, British, and Australian voice actors recorded their respective voices around 2005, unaware of the recordings' eventual usage in Siri. The voice assistant was released as an app for iOS in February 2010, and it was acquired by Apple two months later. Siri was then integrated into the iPhone 4S at its release in October 2011. At that time, the separate app was also removed from the iOS App Store. Siri has since become an integral part of Apple's products, having been adapted into other hardware devices over the years, including newer iPhone models, as well as iPad, iPod Touch, Mac, AirPods, Apple TV and HomePod.

Siri supports a wide range of user commands, including performing phone actions, checking basic information, scheduling events and reminders, handling device settings, searching the

Internet, navigating areas, finding information on entertainment, and is able to engage with iOS-integrated apps. With the release of iOS 10 in 2016, Apple opened up limited third-party access to Siri, including third-party messaging apps, as well as payments, ride-sharing and Internet calling apps. With the release of iOS 11, Apple updated Siri's voices for more clear, human voices, started supporting follow-up questions and language translation and additional third-party actions.

Siri's original release on iPhone 4S in 2011 received mixed reviews. It received praise for its voice recognition and contextual knowledge of user information, including calendar appointments, but was criticized for requiring stiff user commands and having a lack of flexibility. It was also criticized for lacking information on certain nearby places, and for its inability to understand certain English accents. In 2016 and 2017, a number of media reports indicated that Siri is lacking in innovation, particularly against new competing voice assistants from other technology companies. The reports concerned Siri's limited set of features, "bad" voice recognition and undeveloped service integrations as causing trouble for Apple in the field of artificial intelligence and cloud-based services; the basis for the complaints reportedly due to stifled development, as caused by Apple's prioritization of user privacy and executive power struggles within the company.

1.4.4 Alexa

Amazon Alexa[11], known simply as Alexa, is a virtual assistant developed by Amazon, first used in the Amazon Echo and the Amazon Echo Dot smart speakers developed by Amazon Lab 126. It is capable of voice interaction, music playback, making to-do lists, setting alarms, streaming podcasts, playing audiobooks, and providing weather, traffic, sports, and other real-time information, such as news. Alexa can also control several smart devices using itself as a home automation system. Users are able to extend the Alexa capabilities by installing "skills" (additional functionality developed by third-party vendors, in other settings more commonly called apps such as weather programs and audio features).

Most devices with Alexa allow users to activate the device using a wake-word (such as Alexa); other devices (such as the Amazon mobile app on iOS or Android) require the user to push a button to activate Alexa's listening mode. Currently, interaction and communication with Alexa are available only in English, German, French, Italian, Spanish, and Japanese. In Canada, Alexa is available in English and in French (with the Quebec accent).

1.5 Related Work - Tools

There are some tools of course that are similar to ours, but by the end of this report, it shall be clear how different our tool is to others. Some of these tools we mention below.

1.5.1 Articulate

Articulate is a platform for building conversational interfaces with intelligent agents. It is built primarily on top of Rasa NLU, which internally uses Duckling, spaCy, and tensorFlow. It implements a custom dialogue management solution capable of deep complex dialog, but with a focus on simplicity in use.

This is different from our approach where we also rely on Rasa Core to provide dialogue solutions. Needless to mention that this solution has enabled us to integrate our pipeline much easier to provide consistency and performance to the end user through a full Rasa platform pipeline.

1.5.2 Wit.ai

Wit.ai takes a slightly different approach than the rest of these tools. Wit.ai makes it easy for developers to build applications and devices that one can talk or text to. Their vision is to empower developers with an open and extensible natural language platform. It also tries to learn human language from every interaction and leverages the community: what's learned is shared across developers.

The problem with Wit.ai is that its very much developer oriented in the sense that for one to use it, he has to have a proper programming knowledge and some programming languages as well, such as python and JS, and some frameworks such as Node.js. Also Wit.ai learns from its interactions, but that is not the case with Rasa stacks. This is not a disadvantage per se. Rasa can be considered more of a static stack where you choose when and how you would like your bot to learn more by providing more and more data to it, where on the other hand Wit.ai is more of a dynamic approach where the bot learns from its interactions with the user.

1.5.3 Arabot

Arabot is very different from all of the other tools in this list. For a starter, its the only one in the list that talks Arabic natively. But Arabot is not a tool by itself.

Its a bot-as-a-service tool where the complete bot framework and stack is provided as a service to the user. Of course its a paid service and a specialized service that is being built by an external

team. This may not be appropriate for many users especially startups where budget is a critical resource.

Arabot is not an open-source service, so we cannot make any judges other than that. They also do not provide any details on the frameworks or the methodologies used in their stack. So a more comprehensive analysis is not possible.

2. Background

2.1 Chatbot Core

Theres a Chatbot for almost every use case imaginable, and most are built one of two ways:

1. By using state-of-the-art platforms.
2. Independently built by the companies that use them.

There is, however, a third way that flies under the radar, and that is open-source chatbot. Open-source bots are a lot like modern web applications. They live on the interweb, use databases and APIs to send and receive messages, read and write files and perform regular tasks. Open-source bots usually consist of a few core components:

- A web server, in most cases one that is available on the public internet
- The Bot Builder SDK and Tool that provides an interface for developing bots.
- An intelligent algorithm service.
- Storage Service

The open source bots provide easy ways to customize your own bot without having to build it from scratch.

2.1.1 RASA

Rasa Stack is a platform that has seen some incredible growth. Only 2 years old, Rasa has over 300,000+ downloads. Thats almost a download every minute. The Rasa Stack is a set of open source machine learning tools. Developers can use these tools to create chatbots and assistants.

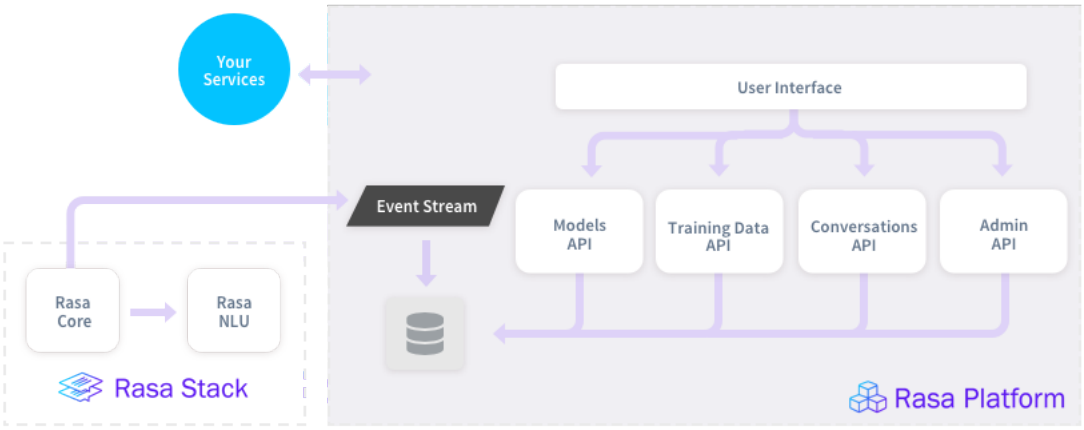


Figure 2.1: The RASA Core

Rasa Stack has two major components that are independent of each other: a core and NLU. Its fairly simple.

The NLU understands a users message based on predefined intent. The ML powered core decides what happens next. Rasa is an independent service i.e. all the data fed or received does not need to run through a third-party API. You can deploy it on-prem or in a private cloud. Its one of the only production-ready platforms delivering flexible and natural conversations that scale.

2.1.2 Microsoft Bot Framework

Possibly one of the most used tools in the business, Microsofts Bot Framework[6] has everything you need. Microsoft Bot Framework is one of the best open source chatbot platforms. In fact, over 41% of businesses in Mindbowsers study preferred MBF to industry alternatives. SMS, Skype, Slack, Email, Office 365, Twitter, Telegram are just some of the many platforms MBF covers. The framework consists of two major components, their Bot Builder SDK (that is open-source access on Github), and their NLU system called LUIS. The Bot Builder SDK supports .NET and Node.js. While with an automatic translation feature, LUIS provides support for over 30 languages.

Because of Microsofts vast array of resources, you can automate almost any type of conversation. You can use LUIS for natural language understanding, Cortana for voice, and the Bing APIs for search.MBF is a popular tool for a reason. It has various samples and templates that help devs build better bots, more quickly. Its also ideal for an omnichannel approach for businesses.

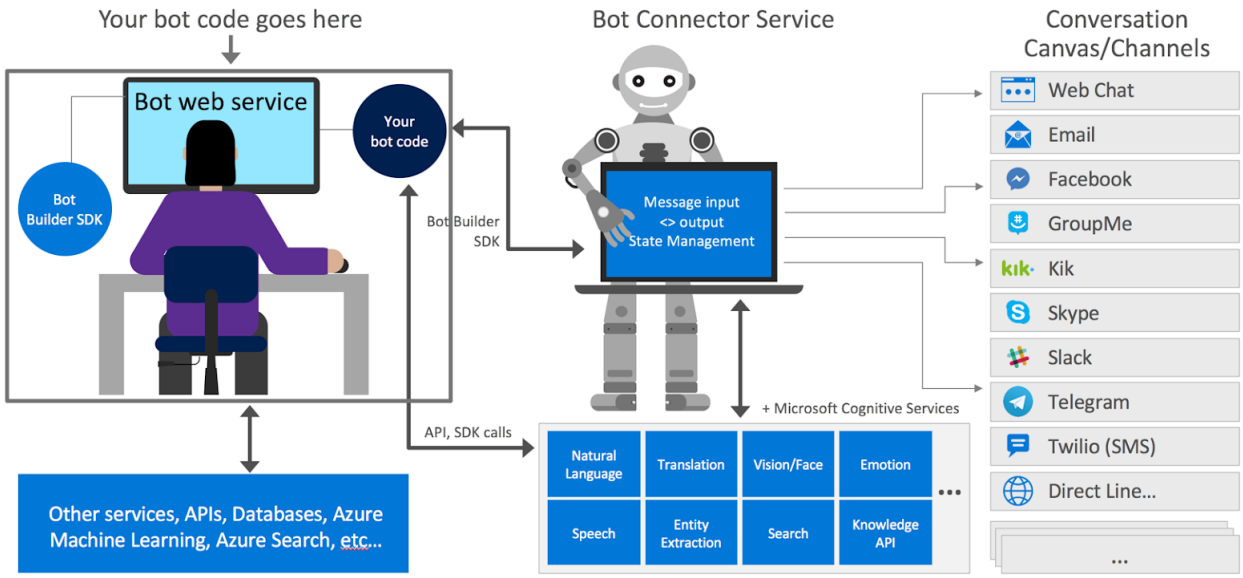


Figure 2.2: The MBF Core

2.1.3 Botpress

It sells itself as the WordPress of Chatbots i.e. an open-source bot building platform. Its built using a modular blueprint. You can snap pieces off and add new bits on an existing code frame. Botpress runs a three-stage installation process. Developers start building the bot, then deploy it

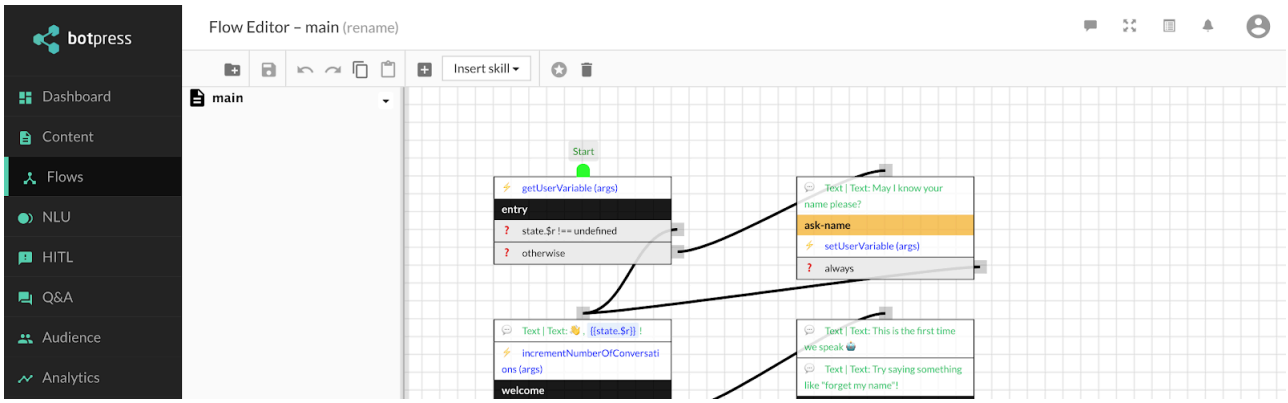


Figure 2.3: The Botpress Core

to their preferred platform and hand-off access so that it can be managed. Botpress built using a developer-friendly environment, has an intuitive dashboard, and is powered by flexible technology. It also comes with several pre-installed components:

- An NLU Engine.
- An administration dashboard.
- A visual flow editor.

- A chat emulator/debugger.
- Support for multiple messaging channels.

As with Rasa, BotPress runs on-prem, so you have full control over the data that comes in and out.

2.1.4 ANA.CHAT

Ana prides itself on being the Worlds First Open-source Chatbot Framework. Free for personal and commercial use, Ana can knock precious days off your chatbot development. Ana[1] comes

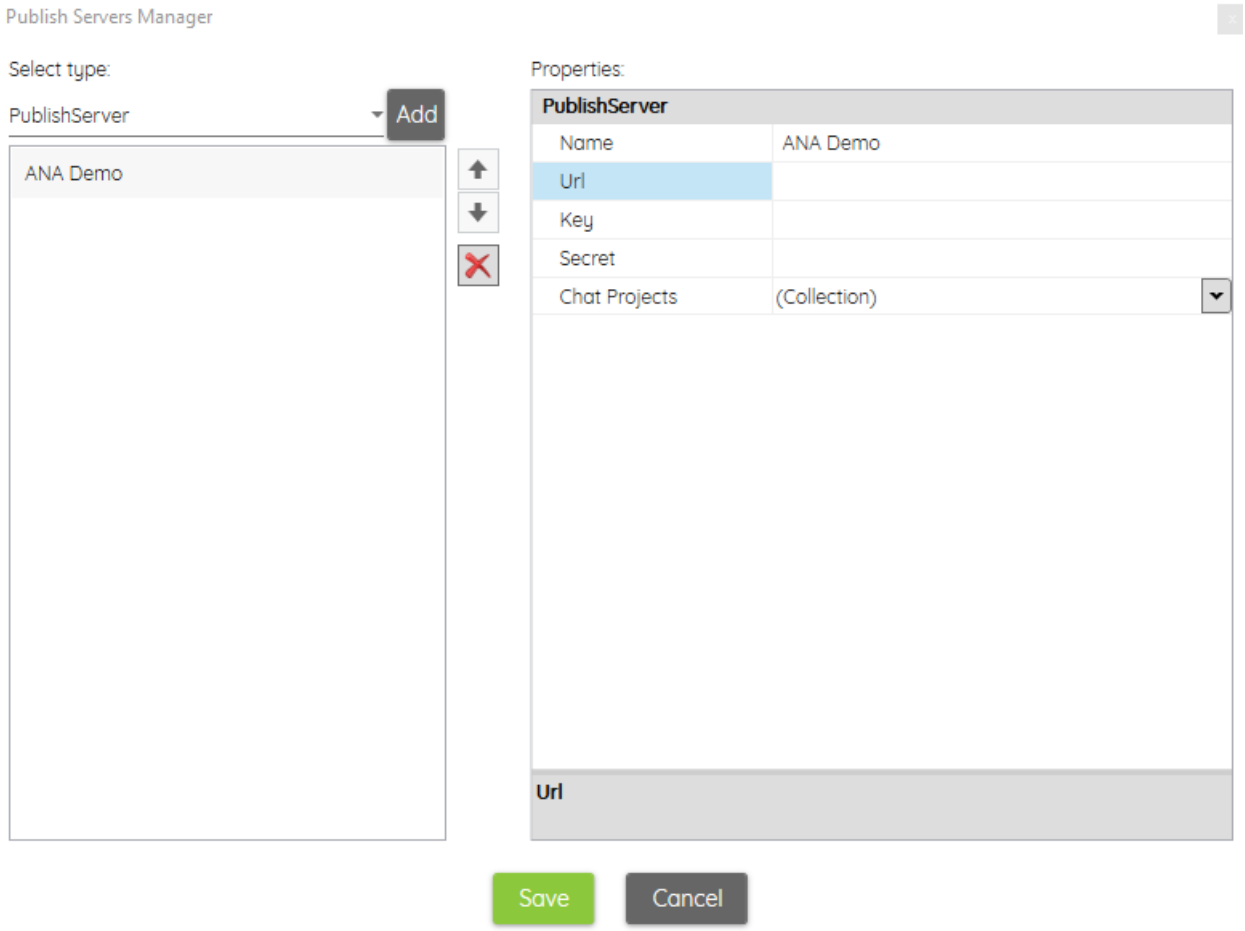


Figure 2.4: The ANA.CHAT Core

with a suite of inbuilt services, like; the Ana Studio, Server, Simulator and SDK. You can use the studio to create and edit text, buttons and input fields visually. The simulator allows you to control your bot experience with features like memory display. Servers allow you to distribute your bot to platforms without having to worry about scalability. Anas SDKs ensure that you can integrate Ana into your app in a matter of minutes.

2.1.5 DialogFlow

DialogFlow[4] (Formerly Api.ai, Speaktoit) is a Google-owned developer of humancomputer interaction technologies based on natural language conversations. The company is best known for creating the Assistant (by Speaktoit), a virtual buddy for Android, iOS, and Windows Phone smartphones that performs tasks and answers users questions in a natural language. Speaktoit has also created a natural language processing engine that incorporates conversation context like dialogue history, location and user preferences.

It supports 14+ languages including Brazilian Portuguese, Chinese, English, Dutch, French, German, Italian, Japanese, Korean, Portuguese, Russian, Spanish and Ukrainian. DialogFlow supports an array of services that are relevant to entertainment and hospitality industries. DialogFlow also includes an analytics tool that can measure the engagement or session metrics like usage patterns, latency issues, etc.

2.2 Morphological Analysis Of Arabic

Morphological analysis is the analysis of the morphology of Arabic language. It is considered the most important part in our bot as we dont have Arabic chatbot because we dont have any open source framework that is integrated with Arabic language processing pipeline. We will list best-known Arabic language analyzers.

2.2.1 Farasa

Farasa[18] (which means “insight” in Arabic), is a fast and accurate text processing toolkit for Arabic text. Farasa can do segmentation, lemmatization, POS tagging, Arabic diacritization, dependency parsing, constituency parsing, named-entity recognition, and spell-checking.

Farasa outperforms Stanford and Madamira while being more than one magnitude faster than both of them. Farasa toolkit is available as a RESTful API.

2.2.2 Stanford CoreNLP

Stanford core NLP[17] is by far the most battle-tested NLP library out there. In a way, it is the golden standard of NLP performance today. Among various other functionalities, named entity recognition (NER) is supported in the library. What this allows is to tag important entities in a piece of text like the name of a person, place, etc.

Core NLP NER tagger implements CRF (conditional random field) algorithm which is one of the best ways to solve NER problem in NLP. The algorithm is trained on a tagged dataset, and the output is a learned model. Basically, the model learns the information and structure in the training data and can use that to label an unseen text. CoreNLP comes with a few pre-trained models like English models trained to structured English text for detecting names, places, etc.

But if the text in your domain or use case doesnt overlap the domain for which the pre-trained models were built for then the pre-trained model may not work well for you. In such cases, you can choose to build your own training data and train a custom model just for your use case.

2.2.3 Madamira

MADAMIRA [19] is the combination and refinement of two valuable tools in Arabic Natural Language Processing (NLP): MADA and AMIRA. It also includes the functionality of MADA-ARZ, a version of MADA developed specifically for the Egyptian dialect.

MADA is a system for Morphological Analysis and Disambiguation for Arabic. The primary purpose of MADA is to, given raw Arabic text, derive as much linguistic information as possible about each word in the text, thereby reducing or eliminating any ambiguity surrounding the word.

MADA does this by using ALMOR (an Arabic lexeme-based morphology analyzer) to generate every possible interpretation (or analysis) of each input word. MADA then applies a number of language models to determine which analysis is the most probable for each word, given the words context. MADAMIRA also includes TOKAN, a general tokenizer for MADA-disambiguated text. TOKAN uses the information generated by the MADA component to tokenize each word according to a highly customizable scheme.

AMIRA is a system for tokenization, part-of-speech tagging, Base Phrase Chunking (BPC) and Named Entity Recognition (NER). Although not ready for this release, MADAMIRA will eventually incorporate the BPC and NER components of AMIRA, providing information that can be highly valuable for tasks such as Arabic parsing. MADAMIRA is designed with the goal of being a functional replacement for MADA and AMIRA, being platform-independent, and providing the ability to process Arabic text at a much faster rate than the older tools. In addition, MADAMIRA provides support for Server-client operation that will allow for processing of Arabic via HTTP. MADAMIRA is designed to process Modern Standard Arabic (MSA) and, in this version, most of its functionality is also available for the Egyptian (EGY) dialect as well.

2.3 Word Embedding

Word embedding is one of the most popular representations of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc. Word embeddings are vector representations of a particular word.

Word vectors are simply vectors of numbers that represent the meaning of a word. For now, thats not very clear but we'll come back to it in a bit. It is useful, first of all to consider why word vectors are considered such a leap forward from traditional representations of words.

Traditional approaches to NLP, such as one-hot encoding and bag-of-words models (i.e. using dummy variables to represent the presence or absence of a word in an observation (e.g. a sentence)), whilst useful for some machine learning (ML) tasks, do not capture information about a words meaning or context. This means that potential relationships, such as contextual closeness, are not captured across collections of words. For example, a one-hot encoding cannot capture simple relationships, such as determining that the words "dog" and "cat" both refer to animals that are often discussed in the context of household pets. Such encodings often provide sufficient baselines for simple NLP tasks (for example, email spam classifiers), but lack the sophistication for more complex tasks such as translation and speech recognition. In essence, traditional approaches to NLP, such as one-hot encodings, do not capture syntactic (structure) and semantics (meaning) relationships across collections of words and, therefore, represent language in a very naive way.

In contrast, word vectors represent words as multidimensional continuous floating point numbers where semantically similar words are mapped to proximate points in geometric space. In simpler terms, a word vector is a row of real-valued numbers (as opposed to dummy numbers) where each point captures a dimension of the words meaning and where semantically similar words have similar vectors. This means that words such as wheel and engine should have similar word vectors to the word car (because of the similarity of their meanings), whereas the word banana should be quite distant. Put differently, words that are used in a similar context will be mapped to a proximate vector space. The beauty of representing words as vectors is that they lend themselves to mathematical operators.

For example, we can add and subtract vectors the canonical example here is showing that by using word vectors we can determine that:

$$king - man + woman = queen$$

In other words, we can subtract one meaning from the word vector for king (i.e. maleness), add another meaning (femaleness), and show that this new word vector ($king - man + woman$) maps most closely to the word vector for queen.

The numbers in the word vector represent the words distributed weight across dimensions. In a simplified sense each dimension represents a meaning and the words numerical weight on that dimension captures the closeness of its association with and to that meaning. Thus, the semantics of the word is embedded across the dimensions of the vector.

2.3.1 Facebook FastText

FastText is a library for learning of word embeddings and text classification created by Facebook's AI Research (FAIR) lab. The model allows creating an unsupervised learning or supervised learning algorithm for obtaining vector representations for words. Facebook makes available pre trained models for 294 languages. FastText uses a neural network for word embedding.

2.3.2 AraVec

AraVec[16] is a pre-trained distributed word representation (word embedding) open source project which aims to provide the Arabic NLP research community with free to use and powerful word embedding models. The first version of AraVec provides six different word embedding models built on top of three different Arabic content domains; Tweets and Wikipedia.

The third version of AraVec provides 16 different word embedding models built on top of two different Arabic content domains. It produced two different types of models, unigrams and n-gram models and utilized a set of statistical techniques to generate the most common used n-grams of each data domain.

2.4 Arabic Data

Arabic data are the data collected and ready to be used to generate word vectors or to train other NLP models.

2.4.1 Wikipedia

This dataset is all Wikipedia Arabic articles from January 20, 2018, data dump (compressed) in Wikimedia format. Content is expected to be (mostly) in modern standard Arabic.

2.4.2 Twitter

Collected from tweets over twitter. Dont handle dialects properly but contain a powerful dataset that can be used for word embedding tasks as used in Aravec.

II Part Two

3. RASA

3.1 Architecture

This diagram shows the basic steps of how an assistant built with Rasa Stack responds to a message.Messages can be text typed by a human, or structured input like a button press. The steps

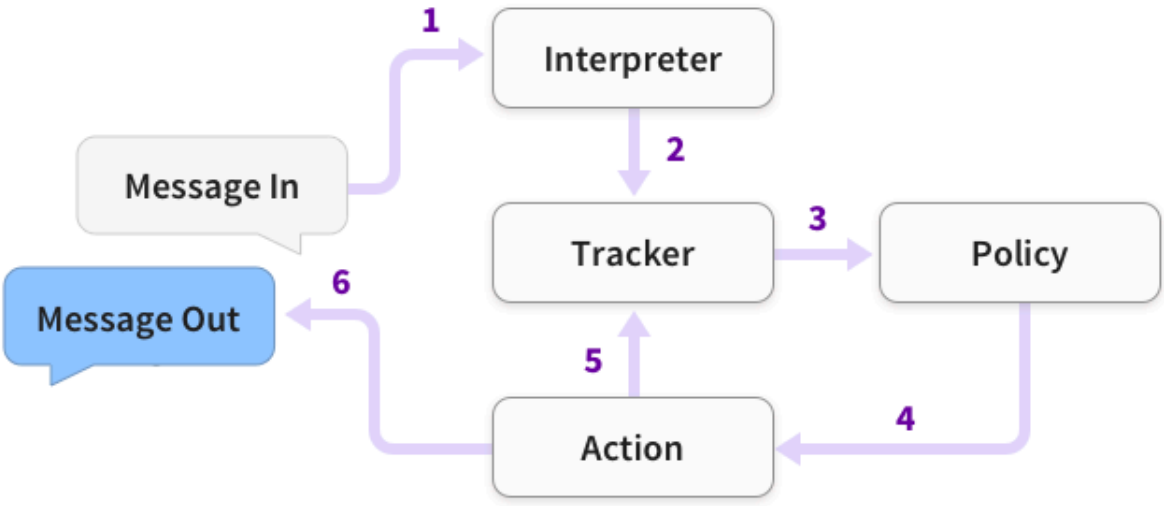


Figure 3.1: Main components of RASA

are:

- The message is received and passed to an Interpreter, which converts it into a dictionary including the original text, the intent, and any entities that were found. This part is handled by NLU.
- The Tracker is the object which keeps track of conversation state. It receives the info that a new message has come in.
- The policy receives the current state of the tracker.
- The policy chooses which action to take next.
- The chosen action is logged by the tracker.
- A response is sent to the user.

3.2 Interpreter

The interpreter invokes the NLU service to parse each message. The service endpoint URL is configurable via a configuration file and cannot be dynamically changed at runtime. The NLU endpoint receives the text of the message but does not receive the representation of the conversation.

3.2.1 Intent Classification

Rasa uses the concept of intents[8] to describe how user messages should be categorized. Rasa NLU will classify the user messages into one or multiple user intents. The two components between which you can choose are:

- Pre trained Embeddings
- Supervised Embeddings

Pretrained Embeddings: Intent Classifier Sklearn

This classifier uses the spaCy library to load pre trained language models which then are used to represent each word in the user message as word embedding. Word embeddings are vector representations of words, meaning each word is converted to a dense numeric vector. Word embeddings capture semantic and syntactic aspects of words.This means that similar words should be represented by similar vectors.Word embeddings are specific for the language they were trained on. Hence, you have to choose Arabic models. You can also use different word embeddings, e.g. Facebooks FastText embeddings.

Rasa NLU takes the average of all word embeddings within a message, and then performs a grid search to find the best parameters for the support vector classifier which classifies the averaged embeddings into the different intents. The grid search trains multiple support vector classifiers with different parameter configurations and then selects the best configuration based on the test results.

When You Should Use This Component?

When you are using pre trained word embeddings, you can benefit from the recent research advances in training more powerful and meaningful word embeddings. Since the embeddings are already trained, the SVM requires only little training to make confident intent predictions.This makes this classifier the perfect fit when you are starting contextual AI assistant project. Even if you have only small amounts of training data, which is usual at this point, you will get robust classification results. Since the training does not start from scratch, the training will also be blazing fast which gives you short iteration times.

Unfortunately good word embeddings are not available for all languages since they are mostly trained on public available datasets which are mostly English. Also, they do not cover a domain specific words, like product names or acronyms. In this case it would be better to train own word embeddings with the supervised embeddings classifier.

Supervised Embeddings: Intent Classifier TensorFlow Embedding

The intent classifier `intent_classifier_tensorflow_embedding` was developed by Rasa and is inspired by Facebooks starspace paper. Instead of using pre trained embeddings and training a classifier on top of that, it trains word embeddings from scratch. It is typically used with the `intent_featurizer_count_vectors` component which counts how often distinct words of training data appear in a message and provides that as input for the intent classifier. In the illustration below you can see how the count vectors would differ for the sentences My bot is the best bot and My bot is great, e.g. bot appears twice in My bot is the best bot. Instead of using word token counts, you can also use n-gram counts by changing the analyzer property of the `intent_featurizer_count_vectors` component to char.This makes the intent classification more robust against typos, but also increases the training time.

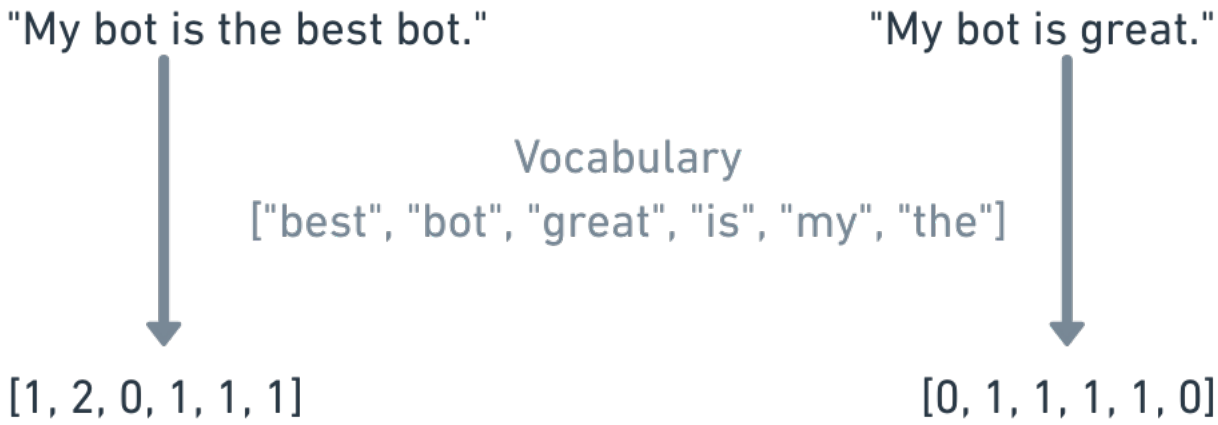


Figure 3.2: Supervised Embeddings Example

Furthermore, another count vector is created for the intent label. In contrast to the classifier with pre trained word embeddings the tensorflow embedding classifier also supports messages with multiple intents (e.g. if user says Hi, how is the weather? The message could have the intents greet and `ask_weather`) which means the count vector is not necessarily one-hot encoded. The classifier learns separate embeddings for features and intent vectors. Both embeddings have the

same dimensions, which makes it possible to measure the vector distance between the embedded features and the embedded intent labels using cosine similarity. During the training, the cosine similarity between user messages and associated intent labels is maximized.

When You Should Use This Component?

As this classifier trains word embeddings from scratch, it needs more training data than the classifier which uses pre trained embeddings to generalize well. However, as it is trained on training data, it adapts to domain specific messages as there are e.g. no missing word embeddings. Also, it is inherently language independent and you are not reliant on good word embeddings for a certain language. Another great feature of this classifier is that it supports messages with multiple intents as described above. In general, this makes it a very flexible classifier for advanced use cases.

Note that in some languages (e.g. Chinese) it is not possible to use the default approach of Rasa NLU to split sentences into words by using whitespace (spaces, blanks) as separator. In this case you have to use a different tokenizer component (e.g. Rasa provides the Jieba tokenizer for Chinese.).

The flowchart below to get a quick rule of thumb decision:

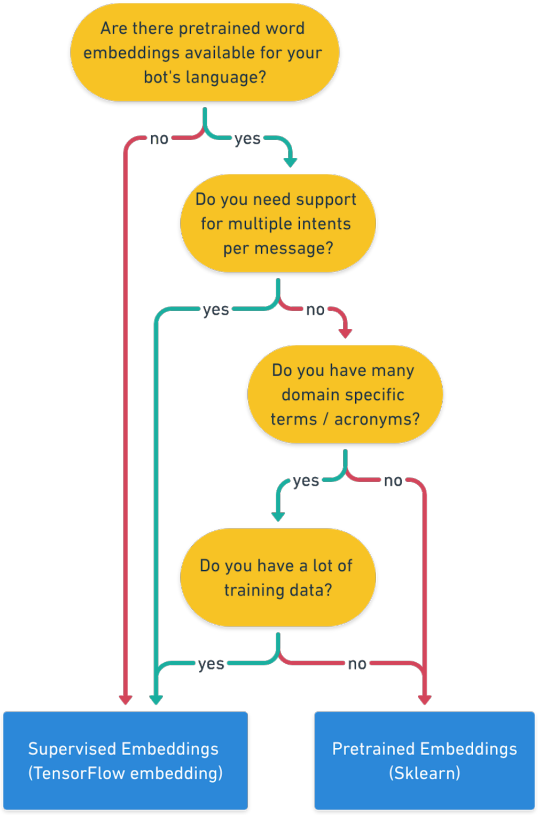


Figure 3.3: Pretrained Vs. Supervised

Common problems with rasa intent classification

1. Lack of training data

   When the bot is actually used by users, you will have plenty of conversational data to pick training examples from. However, especially in the beginning it is a common problem that you have little to none training data and the accuracies of intent classifications are low. An often used approach to overcome this issue is to use the data generation tool Chatino developed by Rodrigo Pimentel. Generating sentences out of predefined word blocks can give you a large dataset quickly. Avoid using data generation tools too much, as model may overfit on generated sentence structures. We strongly recommend you to use data from real users. Another option to get more training data is to crowdsource it, e.g. with Amazon Mechanical Turk (mturk). From our experience the interactive learning feature of Rasa Core is also very helpful to get new Core and NLU training data: when actually speaking to the bot, you automatically frame messages differently than when are thinking of potential examples in an isolated setting.

2. Out-of-Vocabulary Words

   Inevitably, users will use words which trained model has no word embeddings for e.g. by making typos or simply using words you have not thought of.In case you are using pre trained word embeddings, there is not much what you can do except trying language models which were trained on larger corpora. In case you are training the embeddings from scratch using the `intent_classifier_tensorflow_embedding` classifier you have two options: either include more training data or add examples which include the `OOV_token` (out-of-vocabulary tokens). You can do the latter by configuring the `OOV_token` parameter of the `intent_classifier_tensorflow_embedding` component, e.g. by setting it to. , and adding examples which include the token (My is Sara). By doing that the classifier learns to deal with messages which include unseen words.

3. Similar Intents

   When intents are very similar, it is harder to distinguish them. What sounds obvious, often forgotten when creating intents. Imagine the case where a user provides their name or gives you a date. Intuitively you might create an intent `provide_name` for the message It is Sara and an intent `provide_date` for the message It is on Monday.However, from an NLU perspective these messages are very similar except for their entities. For this reason it would be better to create an intent inform which unifies `provide_name` and `provide_date`. In Rasa Core stories you can then select the different story paths, depending on which entity

Rasa NLU extracted.

4. Skewed data

   You should always aim to maintain a rough balance of the number of examples per intent. However, sometimes intents (e.g. inform intent from the example above.) can outgrow the training examples of other intents. While in general more data helps to achieve better accuracies, a strong imbalance can lead to a biased classifier who in turn affects the accuracy negatively. Hyperparameter optimization can help you to cushion the negative effects, but by far the best solution is to reestablish a balanced dataset.

3.2.2 Entity Classification

Understanding the users intent is only part of the problem. It is equally important to extract relevant information from a users message, such as dates and addresses. This process of extracting the different required pieces of information is called entity recognition[9]. Depending on which entities you want to extract, our open-source framework Rasa NLU provides different components. Continuing our Rasa NLU in Depth series, this blog post will explain all the available options and best practices in detail, including:

- Which entity extraction component to use for which entity type
- How to tackle common problems: fuzzy entities, extracting addresses, and mapping of extracted entities

Extracting Entities

As open-source framework, Rasa NLU puts a special focus on full customizability. As result Rasa NLU provides you with several entity recognition components, which are able to target custom requirements:

- Entity recognition with SpaCy language models: `ner_spacy`
- Rule based entity recognition using Facebooks Duckling: `ner_http_duckling`
- Training an extractor for custom entities: `ner_crf`

1. SpaCy

   The spaCy library offers pre trained entity extractors. As with the word embeddings, only certain languages are supported. If language is supported, the component `ner_spacy` is the recommended option to recognise entities like organization names, peoples names or places.

2. Duckling

   Duckling is a rule-based entity extraction library developed by Facebook. If you want to extract any number related information, e.g. amounts of money, dates, distances or dura-

tions, it is the tool of choice. Duckling was implemented in Haskell and is not well supported by Python libraries. To communicate with Duckling, Rasa NLU uses the REST interface of Duckling. Therefore, you have to run a Duckling server when including the `ner_duckling_http` component into NLU pipeline. The easiest way to run the server, is to use our provided docker image rasa/`rasa_duckling` and run the server.

3. `NER_CRF`

   Neither `ner_spacy` nor `ner_duckling` require you to annotate any of training data, since they are either using pre trained classifiers (spaCy) or rule-based approaches (Duckling). The `ner_crf` component trains a conditional random field which is then used to tag entities in the user messages. Since this component is trained from scratch as part of the NLU pipeline you have to annotate training data yourself. Use `ner_crf` whenever you cannot use a rule-based or a pre trained component. Since this component is trained from scratch be careful how you annotate training data:

   - Provide enough examples (> 20) per entity so that the conditional random field can generalize and pick up the data.
   - Annotate the training examples everywhere in training data (even if the entity might not be relevant for the intent)

Regular Expressions / Lookup Tables

To support the entity extraction of the `ner_crf` component, you can also use regular expressions or lookup tables. Regular expressions match certain hardcoded patterns, e.g. [0-9]5 would match 5 digit zip codes. Lookup tables are useful when an entity has a predefined set of values. The entity country can for example has 195 different values. To use regular expressions and/or lookup tables add the `intent_entity_featurizer_regex` component before the `ner_crf` component in the pipeline. Then annotate training data as described in the documentation.

Regular expressions and lookup tables are adding additional features to `ner_crf` which mark whether a word was matched by a regular expression or lookup table entry. As it is one feature of many, the component `ner_crf` can still ignore an entity although it was matched, however in general `ner_crf` develops a bias for these features. Note that this can also stop the conditional random field from generalizing: if all entity examples in the training data are matched by a regular expression, the conditional random field will learn to focus on the regular expression feature and ignore the other features. If you then have a message with a certain entity which is not matched by the regular expression, `ner_crf` will probably not be able to detect it. Especially the use of lookup tables makes `ner_crf` prone for overfitting.

The flowchart below to get a quick rule of thumb decision:

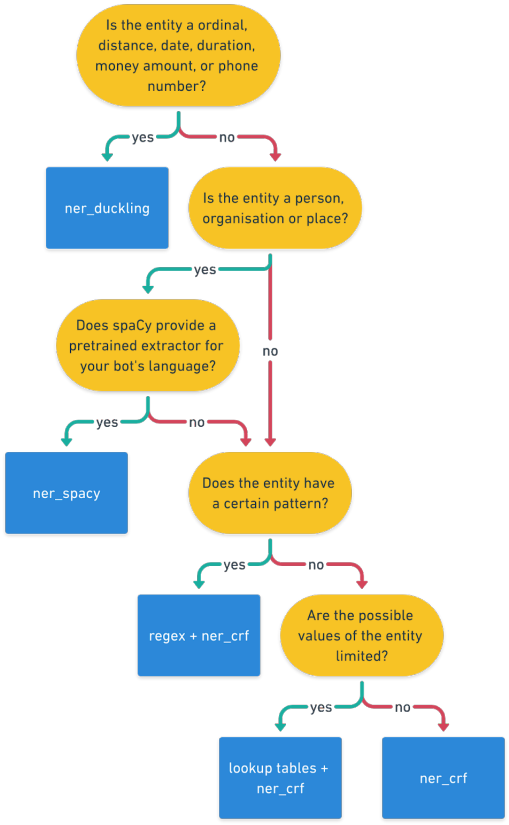


Figure 3.4: spaCy Vs. Duckling Vs. CRF Vs. lookup

Common Problems

1. Entities Are Not Generalizing

   If the extraction of entities does not generalize to unseen values of this entity, there can be two reasons: lack of training data and/or an overfitting `ner_crf` component. If you take heavy use of regular expressions or lookup table features, try training model without them to see if they make the model overfit. Otherwise, add more examples for entities from which model can learn.

2. Extracting Addresses

   If you want to extract addresses we recommend to use the `ner_crf` component with lookup tables. As described in our blog article on lookup tables you can generate lookup tables from sources such as openaddresses.io and use generated lists of cities and countries to support the entity extraction process of `ner_crf`.

3. Map Extracted Entity to Different Value

   Sometimes extracted entities have different representations for the same value. If you are

extracting countries for example, U.S., USA, United States of America, all refer to the same country. If you want to map them to one specific value, you can use the component `ner_synonyms` to map extracted entities to different values. In training data, you can specify synonyms.

3.2.3 Hyperparameter Tuning

This will be all about fine-tuning[10] the configuration of the Rasa NLU pipeline to get the maximum performance. We will cover:

- How to run hyperparameter optimization at scale with Rasa NLU.
- Which hyperparameters give the biggest boost when fine-tuning them.

Hyperparameter Optimization

We explained which NLU components were the best for individual use case and how to deal with potential problems. Choosing the right components is key to the success of contextual AI assistant. However, if you want to go the extra mile and get the best out of the components, you have to tweak the configuration parameters (also called hyperparameters) of the single components. Finding the best configuration is done by training different models with different parameter configurations, and evaluating them on a validation set. The hyperparameters which lead to the best evaluation score are the result of the hyperparameter search. As there are quite many parameters for the components and the model trainings are computational intense, we will show you how to utilize Docker containers so that you can conveniently spread out hyperparameter search to multiple machines.

Defining The Search Space

There are so many hyperparameters for the components where should I start? Since the `intent_classifier_sklearn` for pre trained word embeddings already performs a grid search during the training, the hyperparameter optimization will give you the additional benefit if you train own word embeddings using the `intent_classifier_tensorflow_embedding`. Important hyperparameters for these components are the one from the featurizer component, the classifier itself. For the component `intent_featurizer_count_vectors` we play around with `min_df`, `max_df`, and `max_features`.

The tensorflow classifier has quite a number of parameters. We start adjusting the dimensions of trained embeddings (`embed_dim`) and the number and size of the used hidden layers (`hidden_layers_sizes_a` and `hidden_layers_sizes_b`). For all three parameters, higher values should give you more accuracy, but also might lead to overfitting.

3.3 Tracker

If this is the first message of the conversation, rasa core will create a tracker object with the key `sender_id` which is the incoming identifier of the user and makes sure this id is unique to one user otherwise the prediction might not be coherent for a particular human being. Tracker object usually contains what is found out by the interpreter and/or if there are new slots that are set through an API call such as a users birthday.Tracker object is usually stored in a `tracker_store` which is by default is in Memory, however upon industrialisation you would like to scale your API and it is important that the `tracker_store` is externalised.

3.4 Policy

Since, we have trained different policies, when it comes to predicting the next action, each of these policies will provide a score for the particular action. Then there is a max taken from all scores given by every policy and whichever wins, will be the next action

3.4.1 Choose Next Action Policy

As we mentioned before we can use action with the maximum score to be next action Also, RASA provides a way which allows you to define a threshold that can be used to exclude all action with a score less than the threshold. This may lead to have no next action, but we can use this to identify new action which is "I didnt understand you" or "I didnt get it" which is used in case of ambiguity to ask the user for more clear messages.

3.4.2 Force Chatbot To Some Specific Replies By Training

Rasa provides a way to change your action depending on special training involving user experience. In this example we can ask the user for next best action that he/she expects to get from the bot

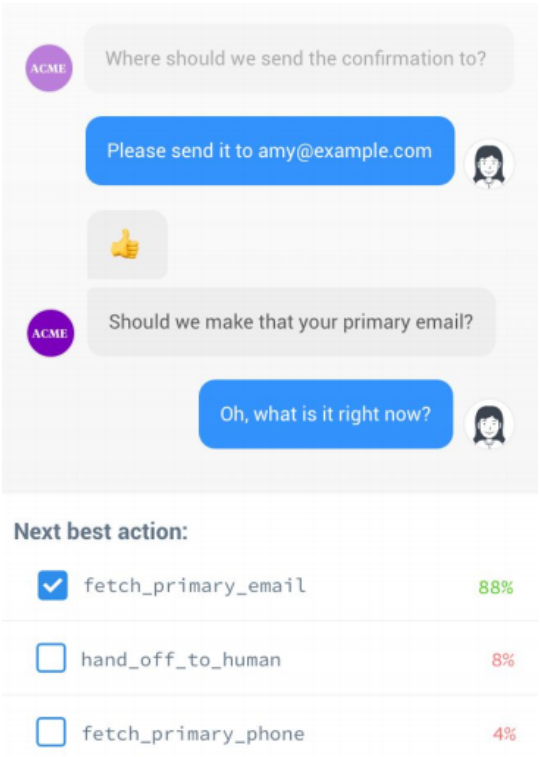


Figure 3.5: Making bot learn by asking user for the best reply

providing the most recommended action the bot would have replied with. Choosing different action will force the bot to change its action next time.

3.5 Actions

Actions are predefined replies for messages. It represents what should chatbot do in a certain state of conversation if we will move from a state to another We can add some actions not just a reply, but we can do something like calling API or changing some data in our database. Actions are determined by The Policy and passed to tracker to log it and move to the next state of the current conversation There are three kinds of actions in Rasa Core:

1. Default Actions

   Shown in table 3.1

action_listen	Stop predicting more actions and wait for user input.
action_restart	Reset the whole conversation. Can be triggered during a conversation by entering /restart if the Mapping Policy is included in the policy configuration.
action_default_fallback	Undo the last user message (as if the user did not send it and the bot did not react) and utter a message that the bot did not understand. See Fallback Actions.
action_deactivate_form	Deactivate the active form and reset the requested slot. See also Handling unhappy paths.
action_revert_fallback_events	Revert events that occurred during the TwoStageFallbackPolicy. See Fallback Actions.
action_default_ask_affirmation	Ask the user to affirm their intent. It is suggested to overwrite this default action with a custom action to have more meaningful prompts.
action_default_ask_rephrase	Ask the user to rephrase their intent.
action_back	Undo the last user message (as if the user did not send it and the bot did not react). Can be triggeredduring a conversation by entering /back if the MappingPolicy is included in the policy configuration.

Table 3.1: RASA default actions

2. Utter Actions

   Starting with `utter_`, which just sends a message to the user as a reply.

3. Custom Actions

   These actions can run arbitrary code An action can run any code you want. Custom actions can turn on the lights, add an event to a calendar, check a users bank balance, or anything else you can imagine. The core will call an endpoint you can specify, when a custom action is predicated. This endpoint should be a web server that reacts to this call, runs the code and optionally returns information to modify the dialogue state.

4. Farasa

In this chapter, we will introduce Farasa in details.

Word segmentation/tokenization is one of the most important preprocessing steps for many NLP task, particularly for a morphologically rich language such as Arabic. Arabic word segmentation involves breaking words into its constituent prefix(es), stem, and suffix(es).

For example, the word “wktAbnA” “وكتابنا” (gloss: “and our book”) is composed of the prefix “w” “و” (and), stem “ktAb” “كتاب” (book), and a possessive pronoun “nA” “نا” (our). The task of the tokenizer is to segment the word into “w+ ktAb +nA” “و + كتاب + نا”. Segmentation has been shown to have significant impact on NLP applications such as MT and IR. Many Arabic segmenters have been proposed in the past 20 years.

Statistical word segments are considered state-of-the-art with reported segmentation accuracy above 98%. The new segment is introduced, Farasa (“insight” in Arabic), an SVM-based segment that uses a variety of features and lexicons to rank possible segmentation of a word. The features include: likelihoods of stems, prefixes, suffixes, their combinations; presence in lexicons containing valid stems or named entities; and underlying stem templates. Extensive tests are carried out comparing Farasa with two state-of-the-art segments: MADAMIRA and the Stanford Arabic segmenter, on two standards.

4.1 Main Idea

Features: In this section, we introduce the features and lexicons that we used for segmentation. For any given word (out of context), all possible character-level segmentations are found and ones leading to a sequence of

$$prefix_1 + ... + prefix_n + stem + suffix_1 + ... + suffix_m$$

where: $prefix_1..n$ are valid prefixes; $suffix_1..m$ are valid suffixes; and prefix and suffix sequences are legal, are retained.

Valid prefixes are: f, w, l, b, k, Al, s. (س ال ك ب ل و ف).

Valid suffixes are: A, p, t, k, n, w, y, At, An,wn, wA, yn, kmA, km, kn, h, hA, hmA, hm, hn, nA, tmA, tm, and tn (تن تم تما نا هم هما ها ه كن كم كماين وا ون ان ات ي و ن ك ت ة ا).

Using these prefixes and suffixes, we generated a list of valid prefix and suffix sequences. For example, sequences where a coordinating conjunction (w or f) precedes a preposition (b, l, k), which in turn precedes a determiner (Al), is legal, for example, in the word fbAlktab فبالكتاب (gloss: "and in the book") which is segmented to (f+b+Al+ktAb ف + ب + ال + كتاب)

Conversely, a determiner is not allowed to precede any other prefix. We used the following features:

- Leading Prefixes: conditional probability that a leading character sequence is a prefix.
- Trailing Suffixes: conditional probability that a trailing character sequence is a suffix.
- LM Prob (Stem): unigram probability of stem based on a language model that we trained from a corpus containing over 12 years worth of articles of Aljazeera.net (from 2000 to 2011). The corpus is composed of 114,758 articles containing 94 million words. This feature helps the segmenter determine if a stem is a valid word.
- LM Prob: unigram probability of stem with first suffix. This would help capture cases where a particular stem most likely appears with a specific suffix such as "jdA" (جدا. Meaning: "much") where the proper segmentation is "jd+A" and the word "jd" appears rarely.
- PrefixSuffix: probability of prefix given suffix.
- SuffixPrefix: probability of suffix given prefix.
- Stem Template: whether a valid stem template can be obtained from the stem. Stem templates are patterns that transform an Arabic root into a stem. For example, apply the template CCAC on the root "ktb" "كتب" produces the stem "ktAb" "كتاب" (meaning: book). To find stem templates, we adopted QATARAs stem template module, and we modified it to im-

prove its coverage and accuracy. Character sequences leading to valid stem-templates are more likely to be stems.

- Stem Lexicon: whether the stem appears in a lexicon of automatically generated stems. This can help identify valid stems. This list is generated by placing roots into stem templates to generate a stem, which is retained if it appears in the aforementioned Aljazeera corpus.
- Gazetteer Lexicon: whether the stem that has no trailing suffixes appears in a gazetteer of person and location names. The gazetteer was extracted from Arabic Wikipedia. We retained just word unigrams.
- Function Words: whether the stem is a function word such as "ElY" "علي" (on) and "mn" "من" (from).
- AraComLex: whether the stem appears in the AraComLex2 Arabic lexicon, which contains 31,753 stems of which 24,976 are nouns and 6,777 are verbs.
- Buckwalter Lexicon: whether the stem appears in the Buckwalter lexicon as extracted from the AraMorph package.
- Length Difference: difference in length from the average stem length.

Learning: We constructed feature vectors for each possible segmentation and marked correct segmentation for each word. We then used SVMrank to learn the feature weights that would hopefully rank correct segmentations higher than incorrect ones. We used a linear kernel with a trade-off factor between training errors and margin (C) equal to 100, which is based on offline experiments done on a dev set. During testing, all possible segmentations with valid prefix-suffix combinations are generated, and the different segmentations are scored using the classifier. We had two varieties of Farasa. In the first, FarasaBase, the classifier is used to segment all words directly. It also uses a small lookup list of concatenated stop-words where the letter "n" "ن"is dropped such as"EmA" "عما" ("En+mA" "عن + ما"), and "mmA" "مما" ("mn+mA" "من + ما"). In the second, FarasaLookup, previously seen segmentations during training are cached, and classification is applied on words that were unseen during training. The cache includes words that have only one segmentation during training, or words appearing 5 or more times with one segmentation appearing more than 70

4.2 Segmentation

Segmentation is the process of dividing written text into meaningful units, such as words, sentences or topics. The term applies both to mental processes used by humans when reading text, and to artificial processes implemented in computers, which are the subject of natural language processing.

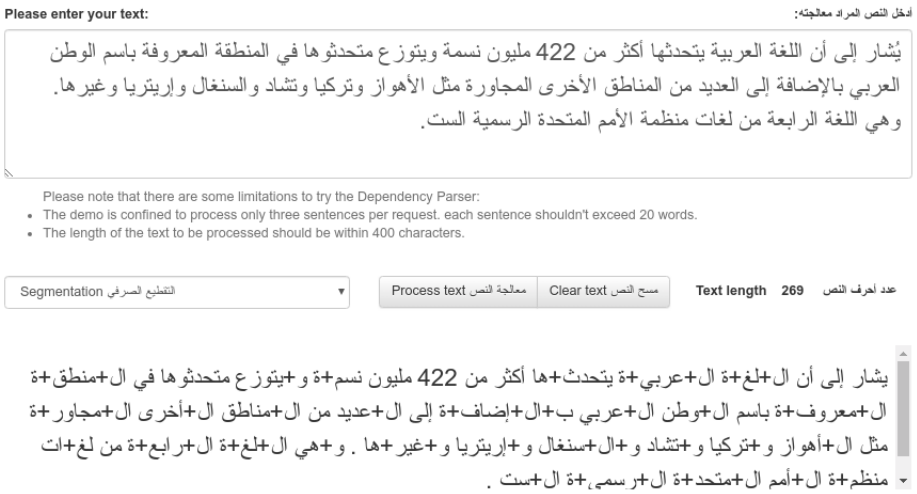


Figure 4.1: Farasa Segmentation

4.3 Spell Checker

Checker is a tool which determines the correctness of the spelling of a given word based on the language set being used.

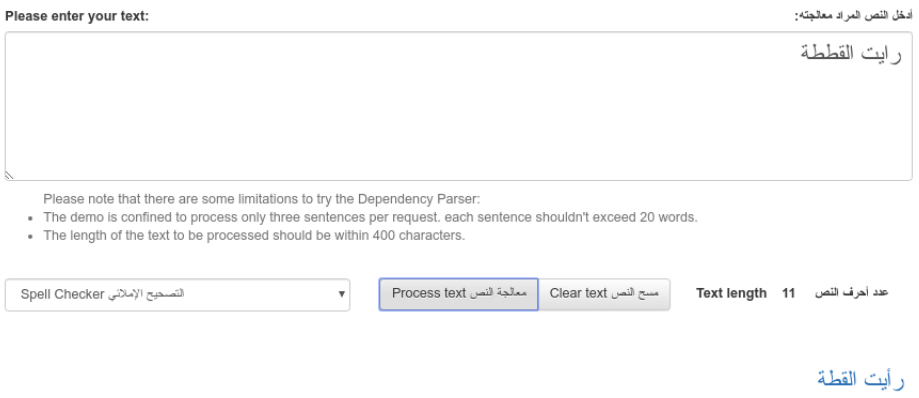


Figure 4.2: Farasa Spell Checker

4.4 Part Of Speech Tagging (POS)

POS tagging is a process of converting a sentence to forms list of words, list of tuples (where each tuple is having a form (word, tag)). The tag in case of is a part-of-speech tag, and signifies whether the word is a noun, adjective, verb, and so on.

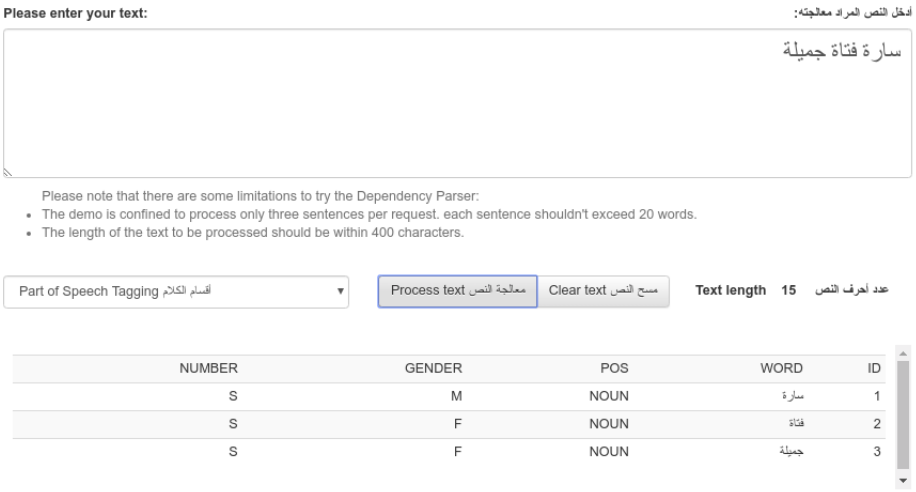


Figure 4.3: Farasa Part Of Speech

4.5 Lemmatization

Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma.

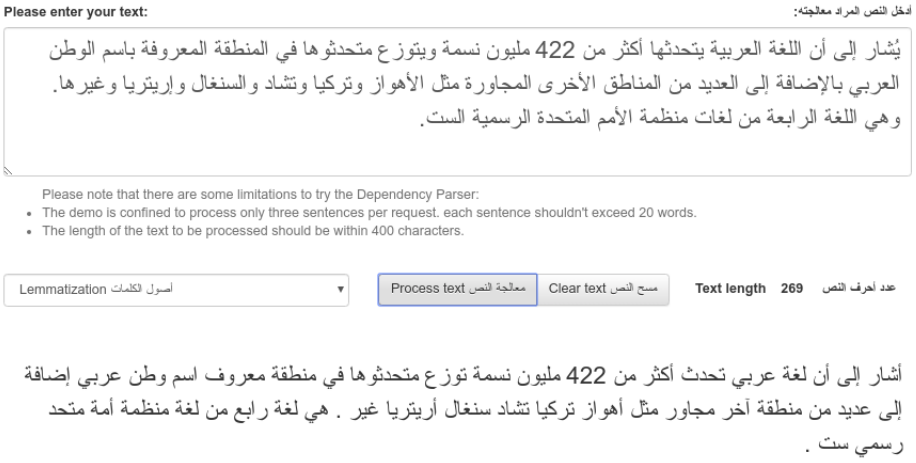


Figure 4.4: Farasa Lemmatization

Lemmatization was one part we relied on heavily in both our tool and the demo. We use Farasa lemmatizer to lemmatize every word in the queried sentence, just as in the example in the last figure. In theory, stemming is much similar and even simpler than lemmatization, so why not use it? The answer to that question is that while stemming tries to remove prefixes and suffixes from words that appear with inflections in free text, lemmatization tries to replace word suffixes with (typically) different suffixes to get its lemma. For languages having complex derivational and inflectional morphology, like Arabic, lemmatization needs more than just suffix replacement.

Arabic has a very rich morphology, both derivational and inflectional. Generally, Arabic words are derived from a root that uses three or more consonants to define a broad meaning or concept, and they follow some templatic morphological patterns (الموازين الصرفية). By adding vowels, prefixes and suffixes to the root word inflections are generated. For instance, the word "وسيفتحونها" (wsyftH-wnhA) "and they will open it" has the triliteral root "فتح" (ftH), which has the basic meaning of opening, has prefixes س + و (w+s) "and+will", suffixes ها + ون (wn+hA) "they...it", stem يفتح (yftH) "open", and lemma فتح (ftH) "the concept of opening."

Arabic verbs have the following grammatical categories: tense (past, present, imperative and future), number (singular, dual, and plural), person (first, second, and third), mood (indicative, subjunctive, and jussive for present verbs, given for past verbs and jussive for imperative verbs), gender (masculine and feminine) and voice (active and passive). Typically, lemmatization of a verb is achieved by obtaining its past tense without any prefixes or suffixes, singular number, third person, given mood, masculine gender, and active voice. Mapping between different grammatical values cannot be done in many cases by just stripping word from its prefixes and suffixes but by applying some complex morphological rules due to the derivational nature of Arabic morphology.

Arabic nouns and adjectives have the following grammatical categories: case (nominative, accusative, and genitive), number (singular, dual, and plural (proper or broken plurals جمع المذكر والمؤنث السالم والتكسير)), gender (masculine and feminine) and definiteness (definite and indefinite). Typically, lemmatization of a noun or an adjective is achieved by obtaining its nominative case without any prefixes or suffixes, singular number, masculine gender and indefinite form. Mapping between different values is not straightforward in many cases.

In addition, according to Arabic morphology and writing system, attaching pronouns to words in some cases changes their last letter. This adds an extra complexity when obtaining lemmas. For example, when nouns ending with Taa-Marbouta letter are attached to possessive pronouns, it will be changed to Taa letter as in حضارته <- ه + حضارة (hDArp+h, HDArth) "its civilization." Also, when verbs ending with Alif-Maqsoura letter is attached to some subject pronouns, it will

be changed to Yaa letter as in هدينا <- نا + هدى (hdY+nA, hdynA) “we guided” or even deleted in some cases as in هدوا <- وا + هدى (hdY+wA,hdwA) “they guided”, and when are attached to object pronouns, it will be changed to Alif letter as in هداها <- ها + هدى (hdY+hA, hdAhA) “guides her”, etc.

4.6 Diacritization

For Arabic, diacritizing written text is important for many NLP tasks. Arabic is generally written without the short vowels, which means that one written form can have several pronunciations, each pronunciation carrying its own meaning(s).

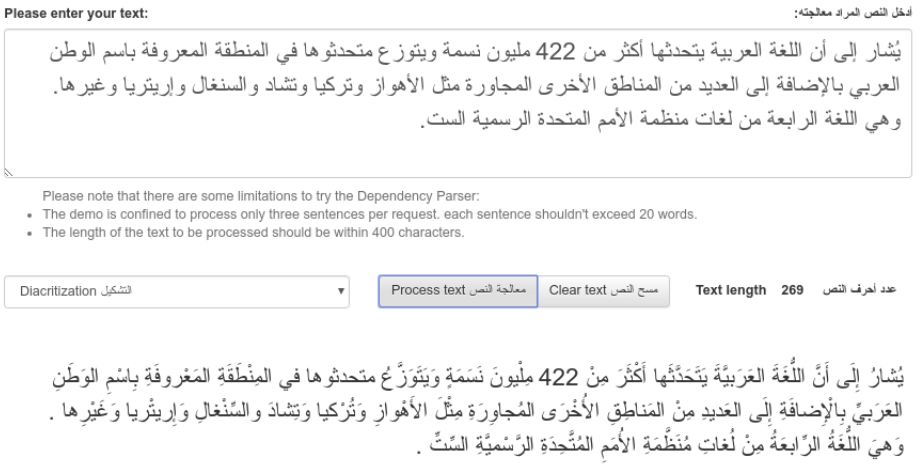


Figure 4.5: Farasa Diacritization

The mentioned cases are just few examples to show how complex the Arabic lemmatization is and reveal that many cases should be considered in addition to stripping words from prefixes and suffixes to get their proper lemmatization.

4.7 Named Entity Recognition (NER)

Named entity recognition (NER) , also known as entity chunking/extraction, is a popular technique used in information extraction to identify and segment the named entities and classify or categorize them under various predefined classes.

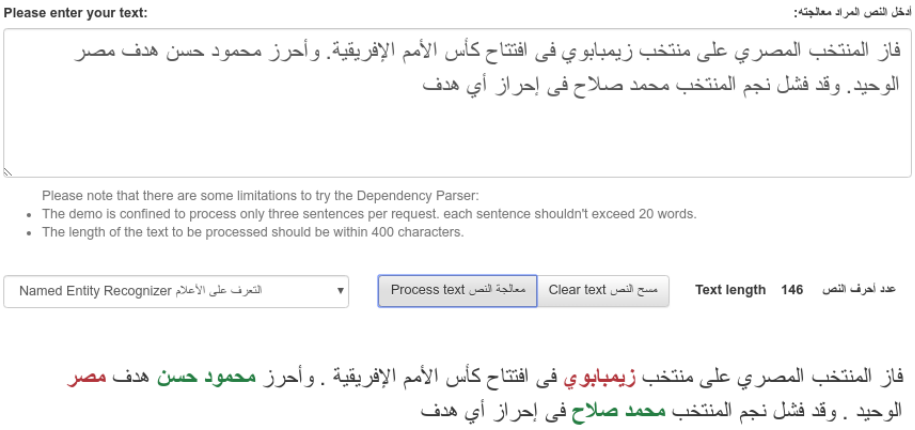


Figure 4.6: Farasa NER

5. FastText

Fasttext as a library for efficient learning of word representations and sentence classification. It is written in C++ and supports multiprocessing during training. FastText allows you to train supervised and unsupervised representations of words and sentences. These representations (embeddings) can be used for numerous applications from data compression, as features into additional models, for candidate selection, or as initializers for transfer learning.

FastText supports training continuous bag of words (CBOW) or Skip-gram models using negative sampling, softmax or hierarchical softmax loss functions. We have primarily used fastTexts pre-trained word vectors for arabic language.

5.1 Word Representations

A popular idea in modern machine learning is to represent words by vectors. These vectors capture hidden information about a language, like word analogies or semantic. It is also used to improve performance of text classifiers.

5.2 Skipgram versus cbow

FastText provides two models for computing word representations: skipgram and cbow ('continuous-bag-of-words').

The skipgram model learns to predict a target word thanks to a nearby word. On the other hand, the cbow model predicts the target word according to its context. The context is represented as a bag of the words contained in a fixed size window around the target word.

Let us illustrate this difference with an example: given the sentence 'Poets have been mysteriously silent on the subject of cheese' and the target word 'silent', a skipgram model tries to predict the target using a random close-by word, like 'subject' or 'mysteriously'. The cbow model takes all the words in a surrounding window, like been, mysteriously, on, the, and uses the sum of their vectors to predict the target. The figure below summarizes this difference with another example. FastText is able to achieve really good performance for word representations and sentence

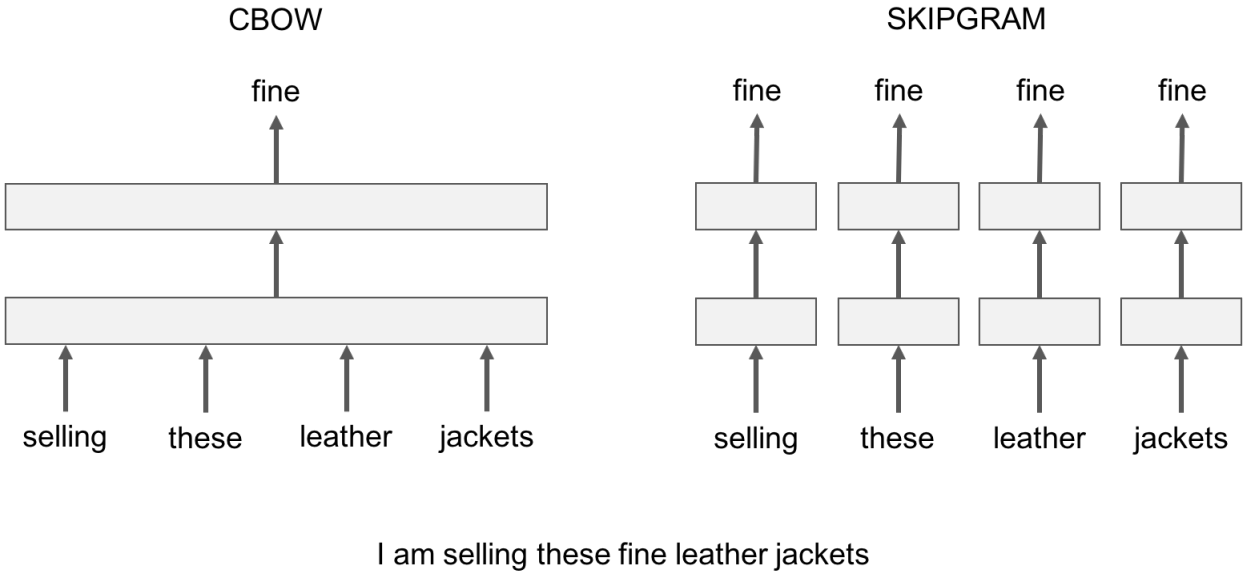


Figure 5.1: CBOW and Skipgram

classification, especially in the case of rare words by making use of character level information.

Each word is represented as a bag of character n-grams in addition to the word itself, so for

example, for the word matter, with n = 3, the fastText representations for the character n-grams is. < and > are added as boundary symbols to distinguish the ngram of a word from a word itself, so for example, if the word mat is part of the vocabulary, it is represented as . This helps preserve the meaning of shorter words that may show up as ngrams of other words. Inherently, this also allows you to capture meaning for suffixes/prefixes.

The model is considered to be a bag of words model[15] because aside of the sliding window of n-gram selection, there is no internal structure of a word that is taken into account for featurization, i.e as long as the characters fall under the window, the order of the character n-grams does not matter.

FastText initializes a couple of vectors to keep track of the input information, internally called `word2int_` and `words_`. `word2int_` is indexed on the hash of the word string, and stores a sequential int index to the `words_` array (std::vector) as its value. The `words_` array is incrementally keyed in the order that unique words appear when reading the input, and stores as its value the struct entry that encapsulates all the information about the word token. entry contains the following information:

Listing 5.1: Struct entry in the words array

```
struct entry {
std::string word;
int64_t count;
entry_type type;
std::vector<int32_t> subwords;
};
```

A few things to note here, word is the string representation of the word, count is the total count of the respective word in the input line, `entry_type` is one of word, label with label only being used for the supervised case. All input tokens, regardless of `entry_type` are stored in the same dictionary, which makes extending fastText to contain other types of entities a lot easier. Finally, subwords is a vector of all the word n-grams of a particular word. These are also created when the input data is read, and passed to the training step.

The `word2int_` vector is of size `MAX_VOCAB_SIZE` = 30000000; This number is hard-coded. This size can be limiting when training on a large corpus, and can effectively be increased while maintaining performance. The index for the `word2int_` array is the value of a string to int hash, and is unique number between 0 and `MAX_VOCAB_SIZE`. If there is a hash collision, and an entry

has already been added to the hash, the value is incremented till we find a unique id to assign to a word.

Because of this, performance can worsen considerably once the size of the vocabulary reaches `MAX_VOCAB_SIZE`. To prevent this, fastText prunes the vocabulary every time the size of the hash gets over 75% of `MAX_VOCAB_SIZE`. This is done by first incrementing the minimum count threshold for a word to qualify for being part of the vocabulary, and pruning the dictionary for all words that have a count less than this. The check for the 75% threshold happens when each new word is added, and so this automatic pruning can occur at any stage of the file reading process.

Facebook distribute pre-trained word vectors for 157 languages[14], trained on Common Crawl and Wikipedia using fastText. These models were trained using CBOW with position-weights, in dimension 300, with character n-grams of length 5, a window of size 5 and 10 negatives. We have used the arabic pre-trained word vector as the basis for RASAs (Spacy in specific) word representation[3].

6. Data Section

In this section we will try to break down the valuable pieces that we may call data in our tool. The tool handles all of these types in a very intuitive way so that the end user neednt worry about how to provide these formats of the specifics of a certain type of a data type, but nevertheless, the data in RASA stack can be thought of as follows.

6.1 NLU Data Format

Training data can be provided as Markdown or as JSON, as a single file or as a directory containing multiple files. Note that Markdown is usually easier to work with and for that our tool converts the data given in the fields by the user to Markdown format and thats why we will only talk about it here.

6.1.1 Markdown Format

Markdown is the easiest Rasa NLU format for humans to read and write. Examples are listed using the unordered list syntax, e.g. minus -, asterisk *, or plus +. Examples are grouped by intent, and entities are annotated as Markdown links, e.g. [entity](entity name).

```
## intent:greet
## intent:ask_to_cancel_licence
```

The training data for Rasa NLU is structured into different parts:

1. Examples
2. Synonyms
3. Regex Features
4. Lookup Tables

Synonyms will map extracted entities to the same name, for example mapping "حسابي الجاري" to simply "جاري". However, this only happens after the entities have been extracted, so we need to provide examples with the synonyms present so that Rasa can learn to pick them up.

Lookup tables may be specified either directly as lists or as txt files containing newline-separated words or phrases. Upon loading the training data, these files are used to generate case-insensitive regex patterns that are added to the regex features. For example, in this case a list of currency names is supplied so that it is easier to pick out this entity.

6.2 Core Data Format

The core is that part responsible for doing the text classification job. The user will most certainly never interact with this part. But its nevertheless very important to understand the building blocks of the core.

6.2.1 Stories

A training example for the Rasa Core dialogue system is called a story.

```
## story_07715946    <!-- name of the story - just for debugging -->
* greet
   - action_ask_howcanhelp
```

This is what we call a story.

- A story starts with a name preceded by two hashes ## `story_03248462`. The story can be called anything, but it can be very useful for debugging to give them descriptive names!
- The end of a story is denoted by a newline, and then a new story starts again with ##.
- Messages sent by the user are shown as lines starting with * in the format intent"entity1": "value", "entity2": "value".
- Actions executed by the bot are shown as lines starting with - and contain the name of the action.

6.2.2 Domains

The Domain defines the universe in which the assistant operates. It specifies the intents, entities, slots, and actions the bot should know about. Optionally, it can also include templates for the things the bot can say.

```
entities:
  - licenses_list
  - license_type
  - receipt_account_type
  - name
  - freezone_license_location
  - cancelled_licence_company_type

actions:
  - utter_greet
  - utter_did_that_help
  - utter_happy
  - utter_goodbye

templates:
  utter_greet:
```

What does this mean?

The NLU model will define the intents and entities that will be needed to include in the domain.

Slots hold information we want to keep track of during a conversation. A categorical slot called `risk_level` would be defined like this:

```
slots:
        risk_level:
        type: categorical
                values:
                - low
                - medium
                - high
```

Actions are the things the bot can actually do. For example, an action could:

1. respond to a user,
2. make an external API call,
3. query a database, or
4. just about anything!

6.3 Challenges In Arabic Data Generation

The Arabic is an extremely bent tongue, with unique sound, especially when pronounce the letters "ض" (d), "ظ" (Tha'a), and "غ" (Ghain). Arabic grammar has a rich morphology and intricate sentence structure and grammarians have described it as the language of d ("لغة الضاد"). Arabic makes use of many inflections because of the appendages, which incorporate relational words and pronouns. Arabic morphology is perplexing because there are about 10,000 roots that are the basis for nouns and verbs. There are 120 patterns in Arabic morphology[13].

The word order in Arabic is variant. We can have a free choice of the word we want to emphasize and put it at the head of a sentence. Generally, the syntactic analyzer parses the input tokens produced by the lexical analyzer and tries to identify the sentence structure using Arabic grammar rules. The relatively free word order in an Arabic sentence causes syntactic ambiguities which require investigating all the possible grammar rules as well as the agreement between constituents.

6.3.1 Inherent Ambiguity In Named Entities

Most Arabic proper nouns (NEs) are indistinguishable from forms that are common nouns and adjectives (non-NEs) which might cause ambiguity. For example, the noun "الجزيرة" (Aljazeera) can be recognized as an organization name or a noun corresponding to island. Nevertheless, Arabic names that are derived from adjectives are usually ambiguous, which presents a crucial challenge for Arabic Named Entity Recognition. As an example, consider the word "أمل" (Amal), which means "hope", and can be confused with the name of a person. In the following two sentences, the word "Amal" means two different senses:

- "الشباب هم أمل البلد" which means: the youth is the hope of the country
- "أمل بنت جميلة" which means: Amal is a beautiful girl

Remedies to resolve this type of ambiguity might not necessarily fix all problems. For example, consider the sentence "رأيت أمل" (I saw hope/Amal) which have either meaning.

6.3.2 Morphology Declension

Arabic is highly inflectional. The prefixes can be articles, relational words or conjunctions, though the suffixes are by and large protests or individual/possessive anaphora. Both prefixes and suffixes are permitted to be mixes, and along these lines a word can have zero or more affixes, I.e. $Word = Prefix(es) + Lemma + Suffix(es)$. Arabic verb morphology is central to the construction of an Arabic sentence because of its richness of form and meaning.

A more complicated example would be words that could represent an entire sentence in English such as "وسيحضرونها" (and they will bring it, wasayahdurunaha). This word can be written in this form: وسيحضرونها = و + سـ + يـ + حضر + ونـ + ها (wa+sa+ya+hdr+runa+ha, and+will+bring+they +it)

In this example, the Lemma "حضر" (hadr) accepts three prefixes: "و" (wa), "س" (sa), and "ي" (ya) and two suffixes: "ون" (wa noun), and "ها" (ha). Thereby, because of the complexity of the Arabic morphology, building an Arabic NLP system is a challenging task.

The early step in analyzing an Arabic text is to identify the words in the input sentence based on its type and properties, and outputs them as tokens. There might be a problem in segmentation where some word fragments that should be parts of the lemma of a word and were mistaken to be part of the prefix or suffix of the word; thus, were separated from the rest of the word as a result of tokenization.

This problem arises with Named Entities Recognition where the ending character n-grams of the Named Entity were mistaken for objects or personal/possessive anaphora, and were separated by tokenization. Moreover, the POS tagger used for the training and test data may have produced some incorrect tags, incrementing the noise factor even further.

6.3.3 Lack Of Uniformity In Writing Styles

The high level of ambiguity of the Arabic script poses special challenges to developers of NLP areas such as Morphological Analysis, Named Entity Extraction and Machine Translation.

These difficulties are exacerbated by the lack of comprehensive lexical resources, such as proper noun databases, and the multiplicity of ambiguous transcription schemes. The process of automatically transcribing a non-Arabic script into Arabic, is called Arabization.

For example, transcribing an NE such as the city of Washington into Arabic NE produces variants such as "واشنجطن", "وشنطن", "واشنغطن", "واشنطن". Arabizing is very difficult for many reasons; one is that Arabic has more speech sounds than Western European languages, which can ambiguously or erroneously lead to an NE having more variants.

One solution is to retain all versions of the name variants with a possibility of linking them together. Another solution is to normalize each occurrence of the variant to a canonical form; this requires a mechanism (such as string distance calculation) for name variant matching between a name variant and its normalized form.

6.3.4 Annexation

Another morphologic challenge in Arabic language is that we can compose a word to another by a conjunction of two words. This conjunction can be with nouns, verbs, or particles. Although it is not common in traditional Arabic language, it is used in Modern Standard Arabic.

Usually, the compound word is semantically transparent such that the meaning of the compound word is compositional in the sense that the meaning of the whole is equal to the meaning of parts put together. For example, the word "رأسمالية" (capitalism, rasimalia) comes from compound of two nouns "رأس المال" (capital, ras almal); the word "مادام" (as long as, madam) comes from the compound of a particle "ما" (ma) and a verb "دام" (dam), and the word "كيفما" (however) comes from the compound of two particles "كيف" (kayf) and "ما" (ma). The meaning of a compound word is important for understanding the Arabic text, which is a challenge to POS tagging and applications that require semantic processing.

Arabic language differs from other languages because of its complex and ambiguous structure that the computational system has to deal with at each linguistic level.

Therefore one has to carefully account for these challenges when both training and testing any arabic speaking bot. One simple approach that we have taken is to provide unambiguous dataset for the bot to train on and avoid the mentioned difficulties. And reask the user for his input in a simpler form.

Part Three

7. System Design

The first part of the system design is concerned with the chatbot itself which may be considered as a system of interacting systems as RASA, Farasa and FastText are interacting together to get a robust chatbot. Second part is concerned with the tool itself that builds and runs the chatbot.

7.1 Methodology

We used agile method while developing the tool.

Using plan-driven method for this tool is not a good idea at all as:

- It is expected to have more time for developing the tool; each phase will take more than a month and this is not suitable for the tool.
- Requirements change frequently and this cant be handled by a plan-driven method specially when new a new version of any component is released.
- The tool can be considered as a generic product that can be customized for a certain purpose as building two chatbots for two different organizations. Here plan-driven method will consider each bot as a product and this will bring high cost while agile method will consider customization steps as more iterations with the new requirements.
- The project is not a large project and can be considered as a medium size project.
- The number of programmers involved in the development process is small.

Extreme Programming method (XP) was used with the benefits of using pair programming.

7.2 Functional Requirements

- Provide a way to build Arabic chatbot manually with a good way to provide custom data either manually or from the file.
- Provide a way to test bot before deployment by being able to talk to it.
- Ability to add bot dependencies like intents, entities, stories and actions.

7.3 Non-functional Requirements

- Ease of use: users adapt to GUI in few times.
- Speed: being able to use all these functionalities with maximum delay 2 seconds.
- Size: the desktop is believed to run in the memory of the user and should use as few resources as possible.
- Reliability: tool always able to launch bot whenever it is correct, mean time failure is as less as possible.
- Portability: tool can run on different OSs
- Adding needed validations to check data before building.

7.4 Use Case Diagram

Use case diagram to illustrate the interaction between system actors.

The user interacts with validator when adding entity, intent, action, story and slot. Validator checks if it is valid or not and report it to the user.

The user interacts with the launcher when model is ready to launch and launcher start building the bot and make it ready to use and make a new actor the bot.

The user interacts with the new actor by sending message and getting proper reply.

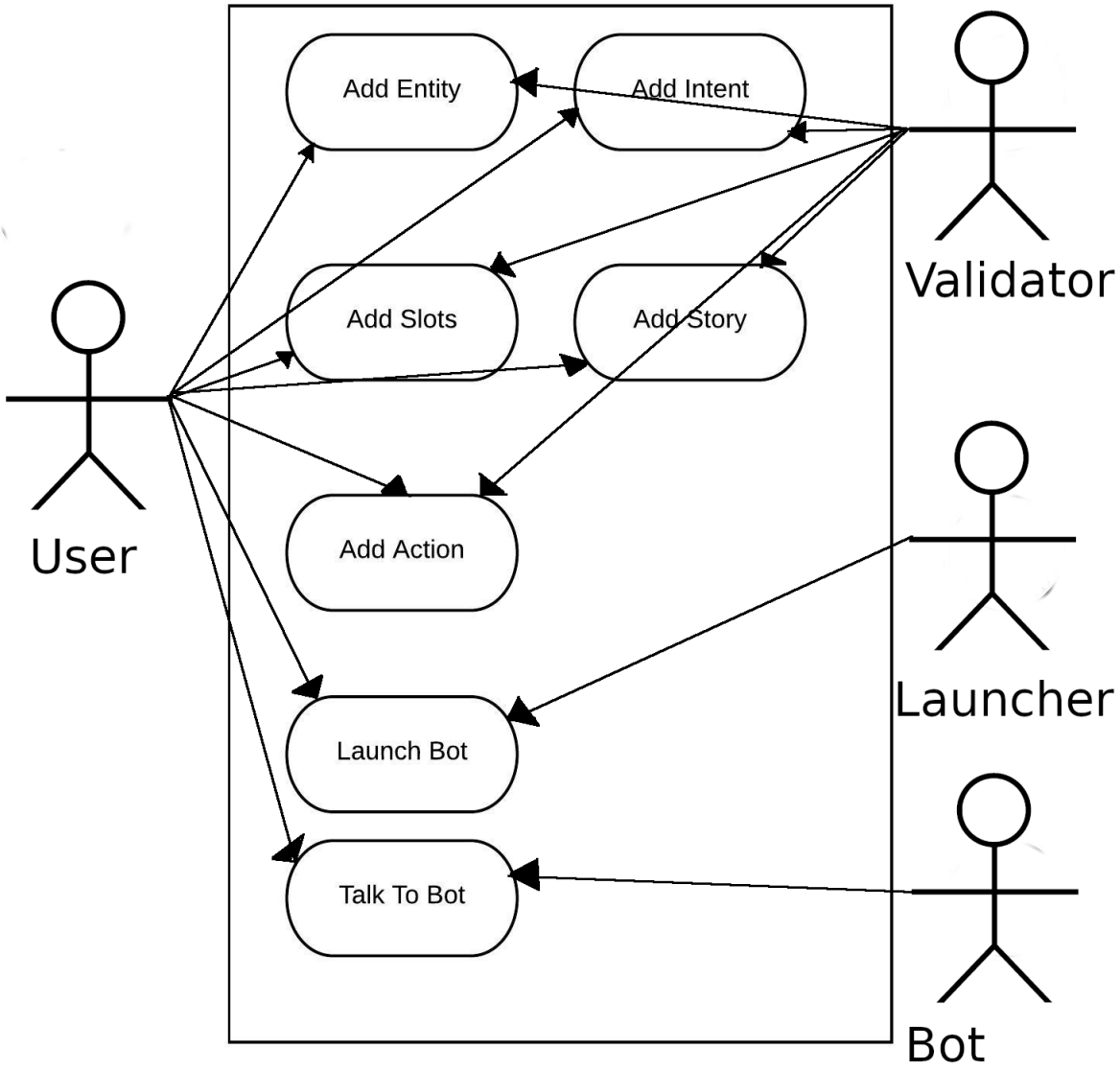


Figure 7.1: Use Case diagram of the system

7.5 Sequence Diagram

Sequence diagram showing user adding chatbot dependencies (entity, intent, ..) with validator checkzing it and then updating database if it is valid.

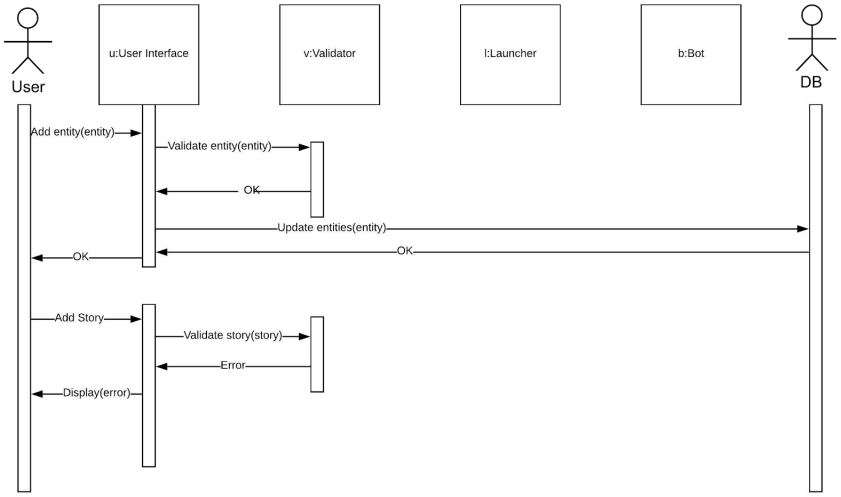


Figure 7.2: Adding attributes Sequence Diagram

Sequence diagram showing user launching demo model or his custom model. Interaction with local database is illustrated. Also, the validation part is shown for custom bots.

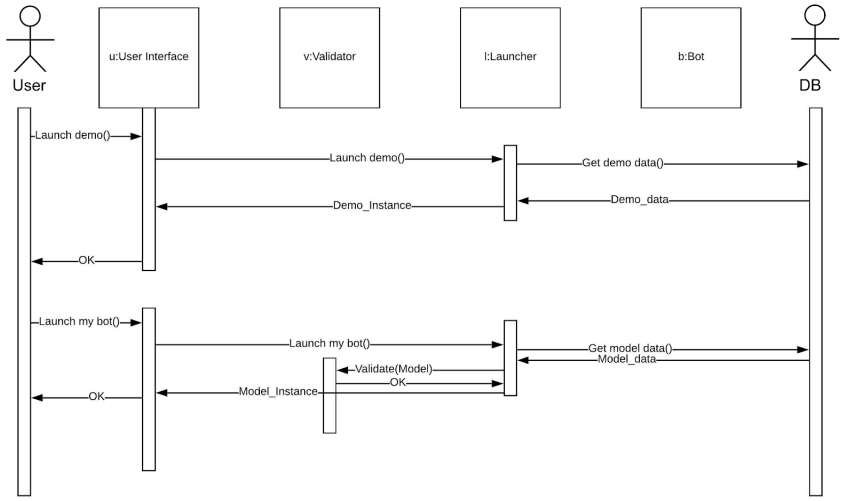


Figure 7.3: Launching Sequence Diagram

Sequence diagram showing how user can interact with bot and exchange messages

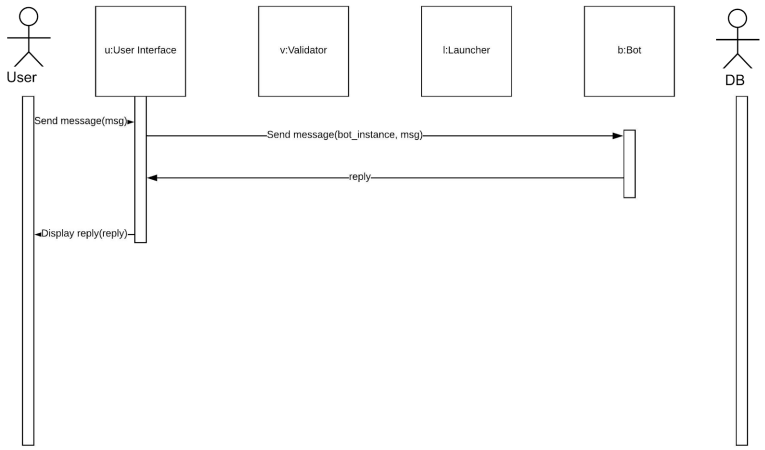


Figure 7.4: Talking to bot Sequence Diagram

7.6 Class Diagram

The generalization part in design and Main classes and Relationships between them.

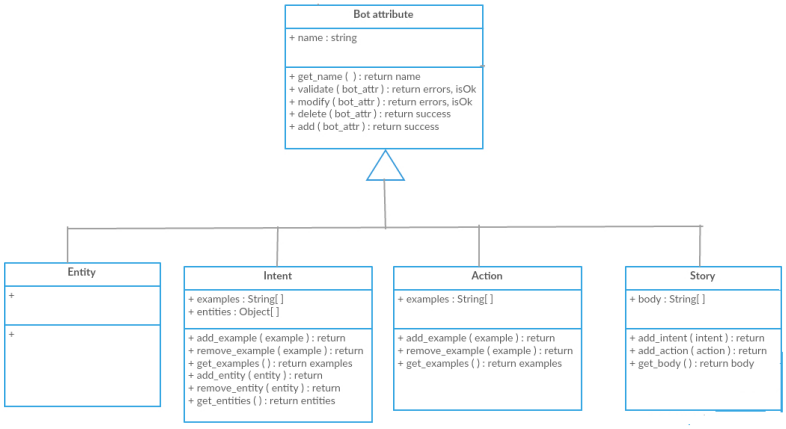


Figure 7.5: Generalization Diagram

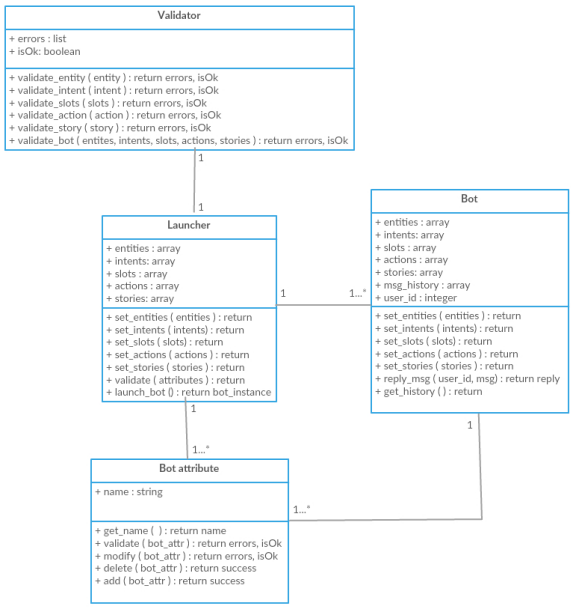


Figure 7.6: Class diagram of the system

7.7 Activity Diagram

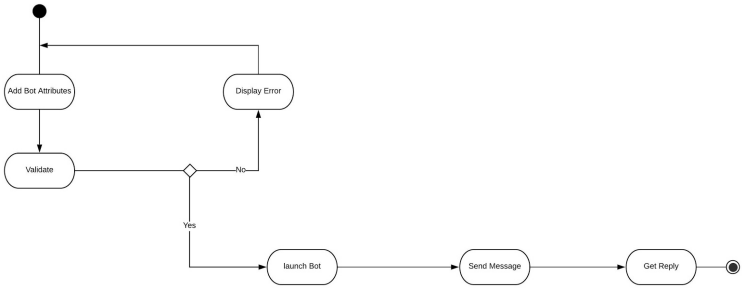


Figure 7.7: Activity diagram of the system

7.8 System Testing

Testing was performed throughout the project development.

Testing is mainly divided into two techniques:

- Automated testing.
- Human based testing.

The main parts in testing:

- Slots validation.
- Entities validation.
- Intents validation.
- Actions validation.
- Stories validation.
- Full validation.
- Backend converter.
- Backend generator.
- Frontend interactions.
- Frontend UI rendering.
- Frontend validations.
- Data integrity testing.
- Inter Process Communication testing.
- I/O performance testing.

Automated testing framework:

We have built and used our own simple framework for testing. It is quite simple and very easy to extend and add more tests throughout the development cycle.

We always prioritized simplicity in the design to keep the code understandable and easy to maintain and the testing code was no exception.

Testing run as a tree where the parent run the child nodes and the child run their child nodes and so on, so extending the testing is as easy as adding a node in the correct place. With the help of logging the tree organization it is quite easy to trace errors and find out where the fail occur to fix it.

Every component have over 10 tests with every one having around 10 or more test cases.

Tests were designed to show that the system is working as intended as well as some extreme test cases to make sure the system will not crash under invalid input.

There was no need to stress the application in simultaneous users at the same time since the

application will be shipped with electron module, so it is only intended to be used by a single user at a time.

But it was essential to test that the I/O performance is good with the needed abstraction to add a cache layer later if needed, as well as the data integrity is quite important in our case, the data MUST always be in a valid state.

From these needs we designed our tests to reflect our needs, With combination of static analysis and automated test case. In performance part we mainly applied static and run time analysis with no automated testing to validate that our I/O operations are efficient and fast with small/medium datasets.

We also implemented with our simple framework several automated tests consisting of tens of test cases to test our validator. Validator is considered the gate to our database backend. So it was very important to make sure it is working flawlessly and no change will reach the database unless the data is perfectly fine. With that said, if a corruption happens to a part of the files, the software will not crash and will alert the user that the Bot has mistakes, but it wont́ be able to recover corrupted data.

Running the automated tests is quite simple, you will just run the root testing file.

```
Assertion succeeded
Assertion succeeded
SLOTS: Test 7 - valid types.
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
SLOTS: Test 8 - float errors.
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
Assertion succeeded
```

Figure 7.8: Part of the automated test run

8. Tool

8.1 How To Add Entity

To add entity you need only to add entity name. Each entity must have a unique name. You have to choose a meaningful name that well-describe what is this entity for.

As you can see in figure 8.1 you can validate that entity name is valid and unique and you can press add directly which validates it before adding it to the local database. You can also load entities from comma-separated file (CSV). When you add invalid value an error message will appear as in figure 8.2

After adding entity it will appear in the table below as figure 8.3, You can delete it or modify it.

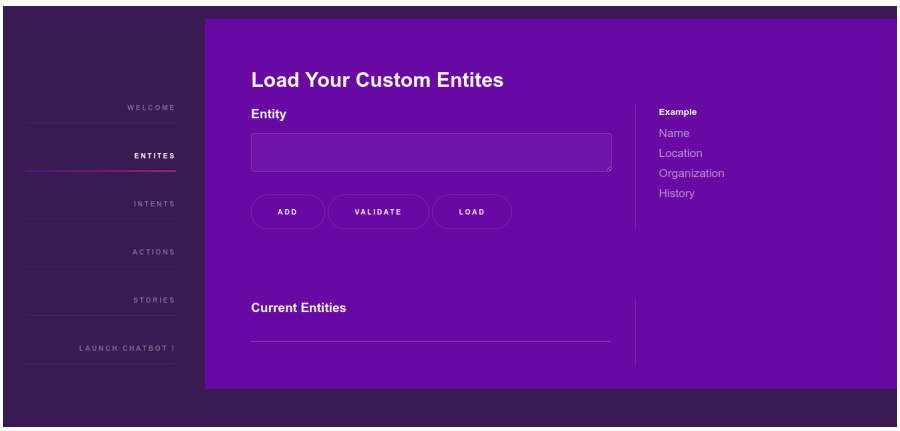


Figure 8.1: Add new entity

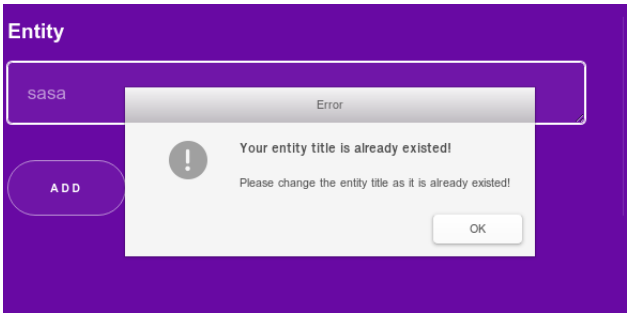


Figure 8.2: Error in entity validation

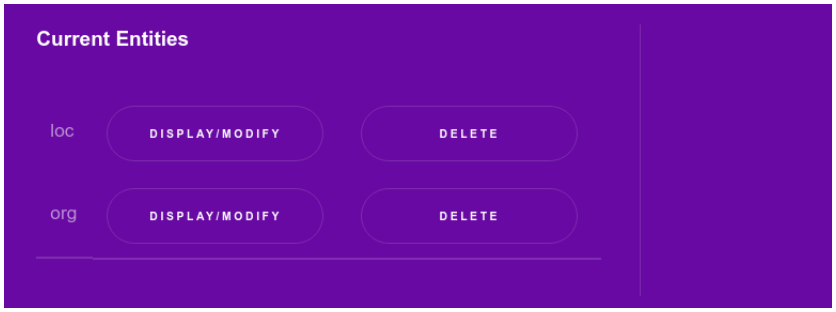


Figure 8.3: Entities display

8.2 How To Add Intent

To add entity you need to define a unique name for it. You must also define at least one example for it. Examples are different ways the bot should identify this intent with. Providing more than ten examples is recommended.

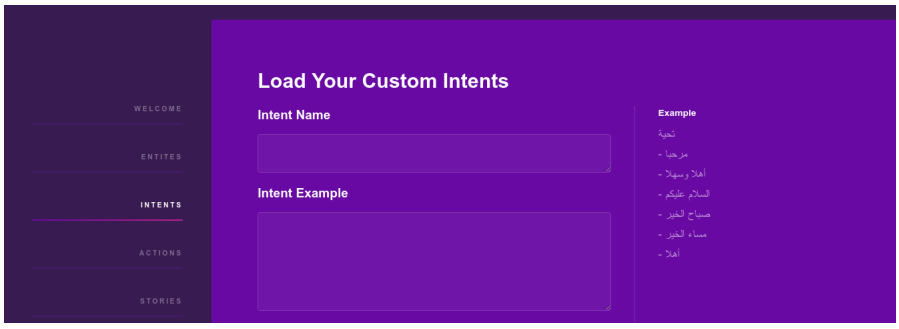


Figure 8.4: Add new intent

You can also define some entities in the examples. You can hover the words that represent the entity and choose entity name from your predefined entities and add it. You can also remove entities if you entered something wrong.

In figure 8.5 the word bad would be identified as loc entity. You can validate entity before you add it or add it directly and it will be automatically validated.

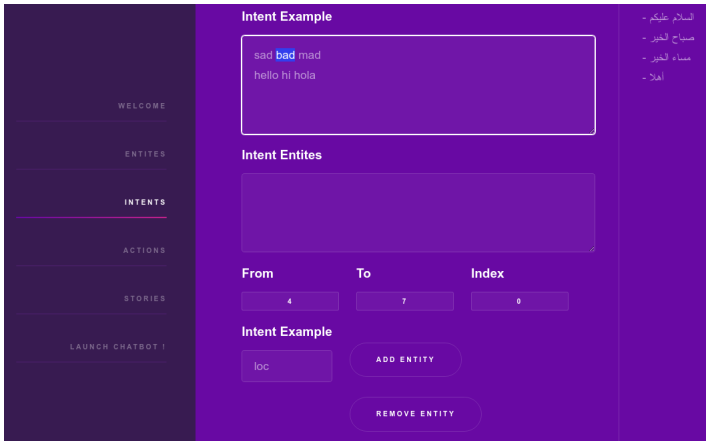


Figure 8.5: Define entites of intent

Any errors will appear as figure 8.6, And current intents are shown in the table as entities. You are able to change any intent or delete it completely

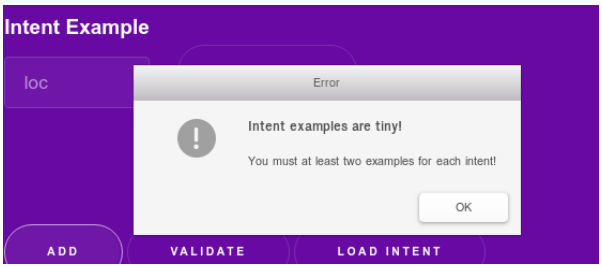


Figure 8.6: Error in intent validation

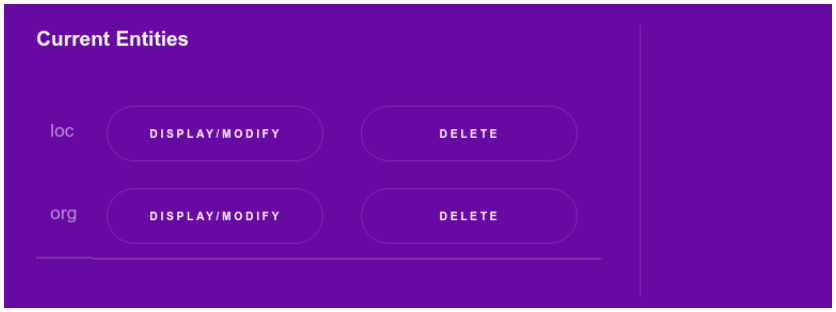


Figure 8.7: Intents display

8.3 How To Add Slots

To add a slot, you need to provide a unique slot name as well as a type. Available slot types are:

- Text
- Categorical
- Bool
- Float
- List
- Unfeaturized

For categorical slots, you also need to define the categories. For slots of type float, you can optionally specify min/max values. For all other types, no more information is needed.

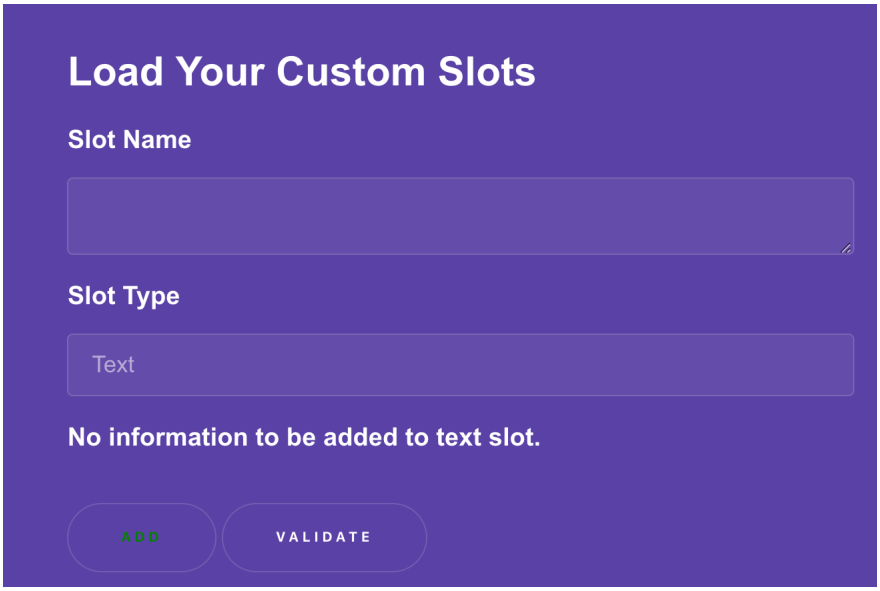


Figure 8.8: Add new Slot

8.4 How To Add Action

To add action you need to define unique name and action text. Name is recommended to start with the prefix `utter_` (e.g `utter_greet`). Action text is a set of predefined replies that the bot will reply with one of them randomly if it detects the related intents. You can also include a predefined slot inside the action text (e.g “hey there {name}”). This can be done by first selecting a predefined slot and then clicking Add. This will append {<slot_name>} to the action text, which shall be filled by the slot value when the bot replies using this action.

You also have the option to display buttons instead (or in addition to) text for an action. In that case you present the bots response in a form of multiple choice question for the user and he could

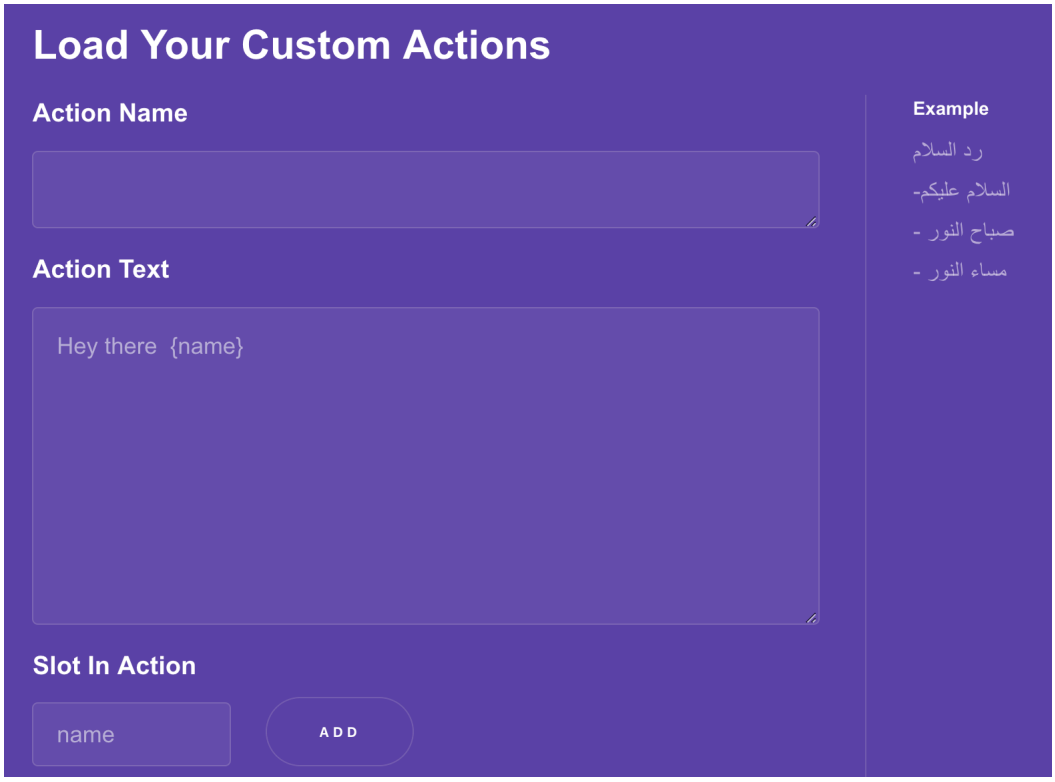


Figure 8.9: Add new Action

enter his response by clicking on one of the buttons or by writing it in text, provided enough NLU data for that response of course (generally if the bot displays buttons, then it‘s better to use them rather than writing a free text reply).

To add buttons, you need to specify three things per button. The slot which is associated with this button (the one thats going to be filled when the user clicks that button), the text to display on the button and finally the slot value that will be saved when the user clicks on the button. In case the slot type is categorical, you will only be able to choose one of the valid categories of that slot as slot value.

You can validate before adding or add it and it will be validated. You can also use CSV files.

You can see actions in the table. You can delete or modify any of the actions. Any errors will be displayed.

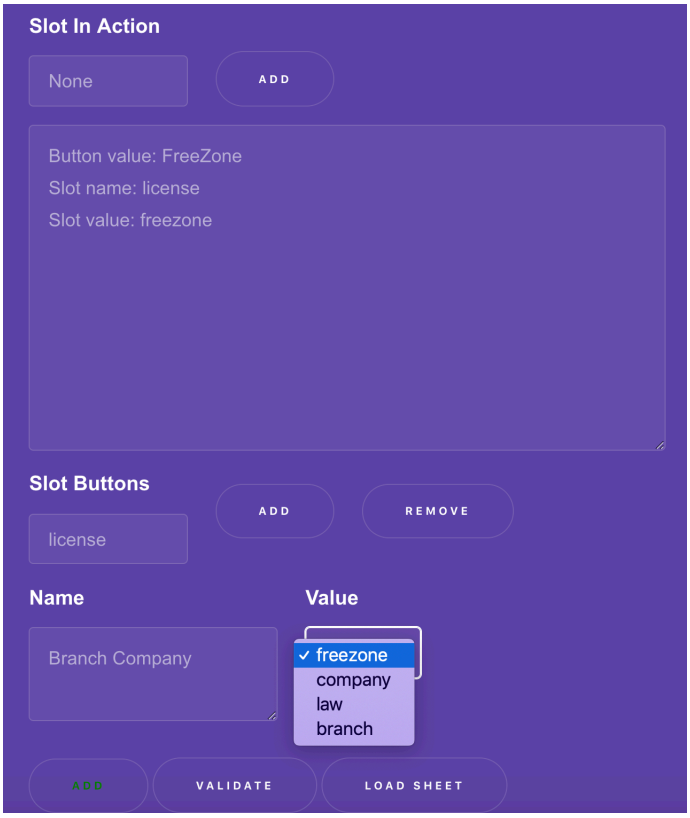


Figure 8.10: Add slots/buttons for actions

8.5 How To Add Story

To add a story you have to define unique name for it. Then you have to add a sequence of expected intents, slots and associated actions -replies- for these intents.

A list of all added intents is shown and a list of all actions also shown so you can add from them the intents and actions. For slots you need to provide a default value first before adding it to the story. For example if you have a slot 'name' of type text. You can provide the default value 'Ali'. If the slot is of type categorical, you will be presented with a dropdown list where you can only choose a default value from valid categories of the slot.

You can remove action or intent. You can validate the whole story or you can add it directly and it will be validated. You can use CSV file.

All added stories are shown in the table with the ability of modifying it as follows :

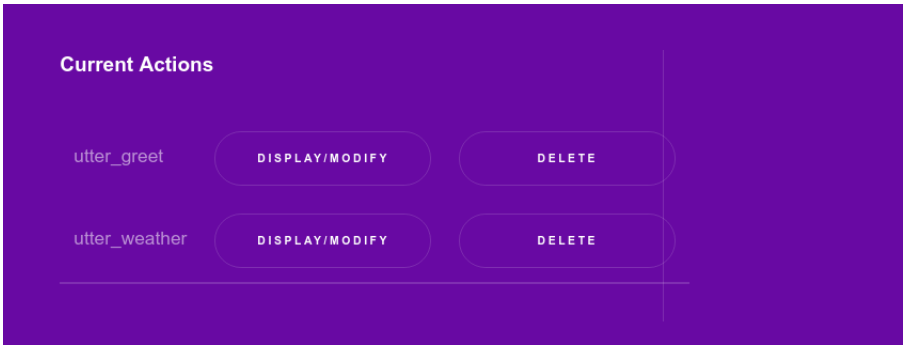


Figure 8.11: Actions display

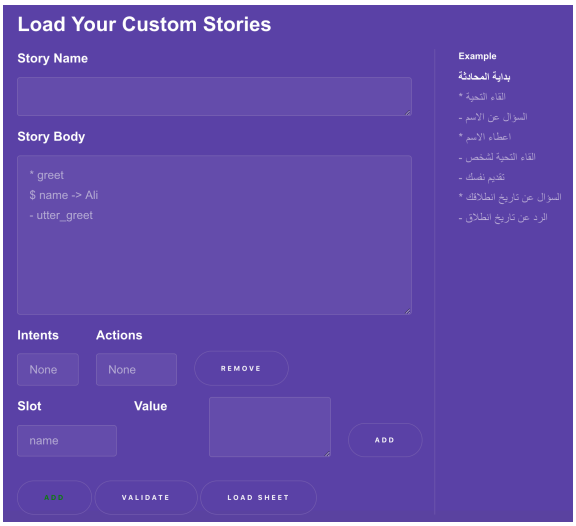


Figure 8.12: Add new Story

8.6 How To Launch

To launch model you have to start local server and build bot to get bot instance.

You can validate model before launching it to avoid any errors like deleting entity without deleting an intent using it or any other logical error.

Once you know your model has no error you can start server. The red offline will turn to green online indicating that you can build your model and start training.

In the backend the model will be trained over your custom data and an instance of the bot will be passed. You then can talk to chat until you stop server.

You can try trakheesak in the tool by starting its server and the pretrained bot will be passed to you and you can talk to it while the server is running.

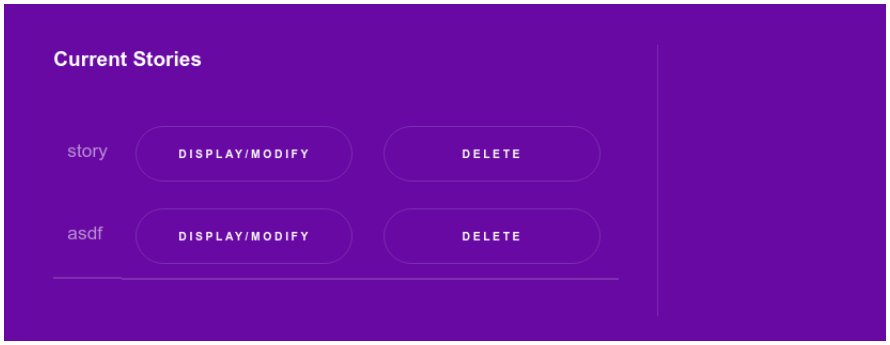


Figure 8.13: Stories display

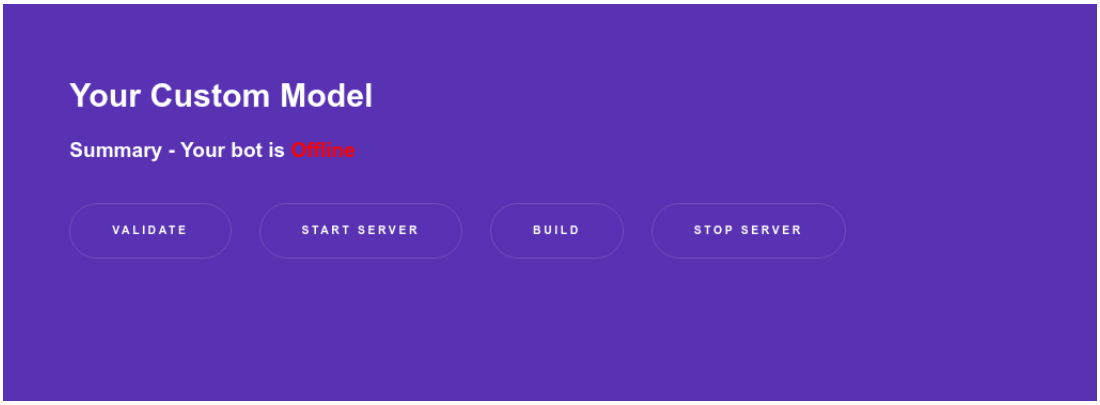


Figure 8.14: Launch custom model

8.7 How To Talk To Bot

You can talk to bot in the launch tab once you start server. You can only start one server and talk to one bot.

You can send messages to bot like in figure 8.14

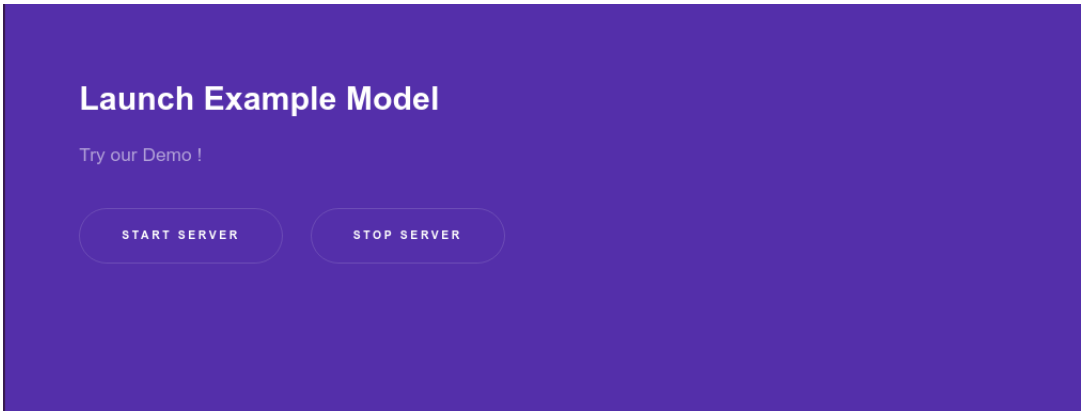


Figure 8.15: Launch Trakheesak

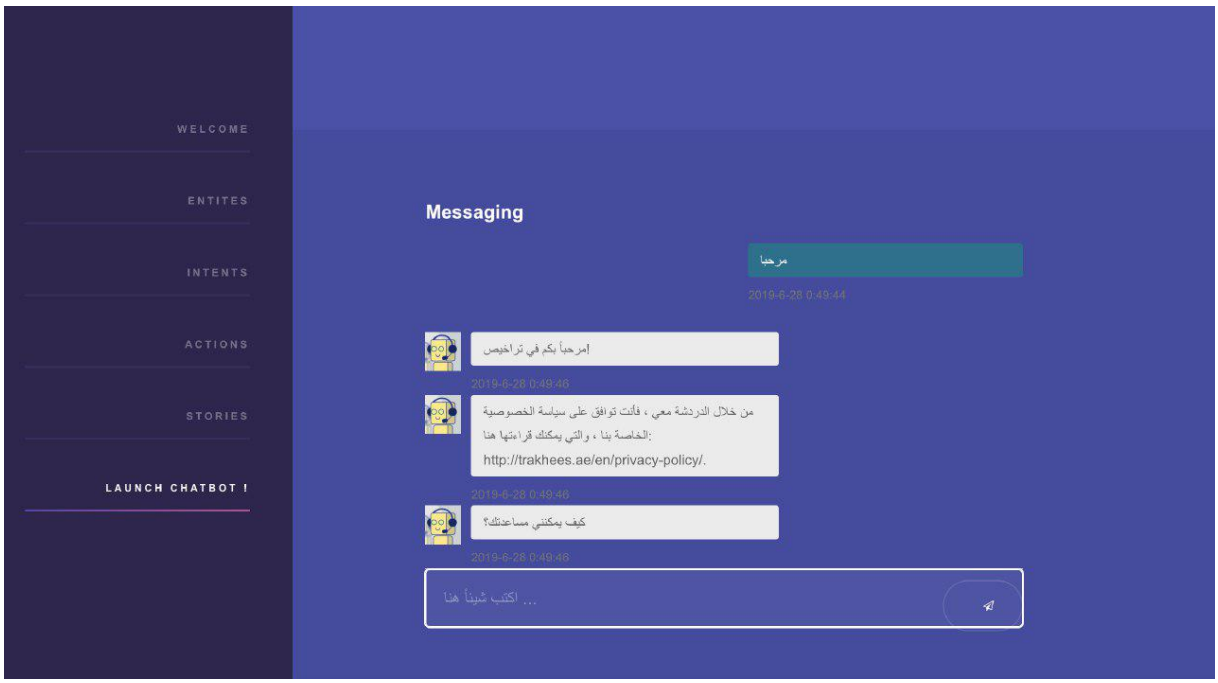


Figure 8.16: How to chat with the bot

IV Part Four

9. Application

The purpose of this chapter is to showcase a contextual AI assistant built with the tool. We will try to demonstrate all the aspects of the demo: Its strengths and its weaknesses.

9.1 Trakheesak

Trakheesak is an alpha version and lives in our docs, helping users getting started with the process of extracting licences in the UAE . It supports the following user goals:

- Understanding the trakhees framework
- Answering some FAQs around the trakhees process
- Applying for a visa
- Inquire about the status of an application
- Inquire whether a user can extract a licence from some location
- Requesting a call to the trakhees team
- Handling basic chitchat

Trakheesak can be divided into three types of questions:

1. Simple: where the user asks a question in one or more form and the answer will always be the same from the bot.

   In the simple type of questions the bot does not try to extract any data from the query, i.e. any entity in the query is not that valuable because the bot itself wont extract the entity.

   This type is most suitable for generic queries. Or in other words, queries that its answer

wont differ when asked by different users.

2. Intermediate: where the user may ask the question in one or more forms, the answer to these kind of questions usually depends on the context and the entities provided by the user in the query itself (or even a previous one).

   Sometimes these questions may be asked without specifying an entity though in which case it is treated as the simple type.

   In intermediate questions, entities are optional, but if provided it may affect how the bot handles the question, for example, the answer to this question might be to list all the possible places a user can extract a licence from.

   Whereas the answer to this question might be to check if the location exists in a predetermined list of possible locations (or possibly to send the location through an api to validate the location in some external database

3. Complex: where the user must provide the necessary entities for the bot to be able to answer. The bot can go a step further when its not able to extract the entity from the users query and asks the user directly for that entity or maybe make the user select some value from a predetermined list.

The first 2 types does not require any coding experience from the user and will be automatically created using the tool when the user provides the training set. This is not true though in complex questions as some programming experience is required to handle the entered entities and the missing ones. Luckily, only a very basic knowledge in python programming is more than sufficient to implement these types of questions.

Q type	**First**	**Second**	**Third**
Entities	Not required	Optional	Required
Slots	Not required	Required	Required
Accuracy	Usually high	Moderate to high	Very high
Difficulty	Easy	Moderate	High
Programming Knowledge	Not required	Optional	Required
Support in the tool	Automatic	Supported	-

Table 9.1: Questions types table

9.1.1 Weaknesses

Its not all roses, though. Trakhees suffers from very subtle weaknesses that makes its responses and reactions sometimes seem very weird. The good news is that all the weaknesses are training set related which means that given enough good dataset one can expect the bot to perform very well.

- Training set weaknesses

  When designing trakhees we had to write the training set ourselves and hence the test set too. In the world of data science that may be a mistake because we are normally biased. Moreover the quality of the training set tends to degrade as more and more examples are added. We went through two different iterations when designing trakhees.

  The first iteration was the worst as both the quantity and the quality of the training set were very low. The quantity varied from 4 to 6 examples per question which was very low and the quality was very low as some deliberate typos were usually introduced in the nlu data.

  The reason behind this was to test how dependable the system was to the training set. Of course a user should provide a large dataset if it were to expect a quality bot to be produced but its useful to know how large this dataset should be.

  In the second iteration we added much more data to the training set. The average was 25 to 30 examples/Question and for some questions it was even double of that. Along with some artificial data augmentation where entities from different examples are interleaved.
- Farasa weaknesses Even though Farasa is the best overall Arabic NLP toolkit, but it still suffers from very subtle weaknesses in its lemmatizer which can fail to correctly lemmatize a given word from the first time. And sometimes it even fails entirely to correctly lemmatize the word.
- Trakhees domain Many of the questions share the same word in the domain of trakhees such as the word “رخصة” or the word “فيزا”. These words occur very frequently that sometimes (Although very few), the intent classifier fails to distinguish the intents from each other. Luckily the solution is very straightforward, providing more training data shall eliminate the problem entirely.

10. Performance Evaluation

10.1 Metrics

- Response time for message
- Percentage of correctly classified entities and throughput
- Error rate of misclassified entities and probability of misclassifying
- Events of entity not classified at all and event time and probability
- Build time for chat tool
- Probability of wrong input data to the tool and not detected
- Event of tool failing and time and probability of that event

%———————————————————

10.2 System Parameters

- Speed of the user CPU.
- Speed of the server CPU.
- Speed of the network.
- Operating system overhead for interfacing with the channels.
- Operating system overhead for interfacing with the networks.
- Reliability of the network affecting the number of retransmissions required.

%———————————————————

10.3 Workload Parameters

- Time between successive messages sent.
- Message length and number of corresponding packets.
- Number and length of the replies and its corresponding packets.
- Type of channel.
- Other loads on the local and server CPUs.
- Other loads on the network between user and server.

10.4 Factors

- Number of epochs of training the chat bot
- The threshold for confidence of intents and entities
- Average number of training examples per intent/entity
- Max history size
- Different pipelines (tensorflow, spacy, mitie)
- Different combination of Farasa components (lemmatizer, spellchecker, tokenizer,)

10.5 Evaluation Technique

- Measurements for the system performance
- Analytical model for validation

10.6 Experimental Design

We tried different models in our experimental design hoping to achieve the best possible accuracy. We trained different nlu pipelines, gamma values and different kernels. Training was done with the Farasas lemmatizer included in the pipeline and without, using N-Gram featurizer with max number of n-grams of 10 and without the N-Gram, Gamma values of (0.1, 0.01, 0.001, 0.5, 0.05, 0.005), C values of (1, 2, 5, 10, 20, 100) and 4 different kernels (linear, polynomial, RBF and sigmoid). We designed full experiment with $2 * 2 * 6 * 6 * 4 = 576$ experiments We found out that the factors that affect the system the most are the existence of the Farasa lemmatization, the gamma values and the kernel.

10.7 Data Presentation

We try here to present to you the bone of our analysis and all the metrics we relied on when opting to choose the best set of hyperparameters.

- **The test set**

The test set used to test all the models that were generated has the following specification:

1. Splitted from the original training set which had a total of 684 intent examples (29 distinct intents) and 147 entity examples (7 distinct entities) with a ratio of 4:1.
2. The resulting test set had a total of 147 intent examples (29 distinct intents) and 36 entity examples (7 distinct entities).
3. Classes (Intents and entities) were carefully balanced in accounting for ill-defined metrics.
4. The NLU confidence threshold sets at 0.3 and it is the threshold at which the NLU decides if it's confident enough that the predicted intent is the correct enough or if it needs to reask the user to rephrase.

As mentioned in the experimental design section, we found out that the hyperparameters that affected the model the most were the existence of the Farasa lemmatization, the gamma values and the kernel. And so, we designed a new experiment with a target of finding the best possible set of parameters using a grid search. The search space itself was for the farasa lemmatizer {0, 1}, the gamma values {0.1, 0.01} and the different kernels {linear, polynomial, rbf and sigmoid). A total of 16 models were trained and tested.

Of course listing the full analysis for the 16 models is very infeasible, so we will only list the scores of all the models here (F-Score, precision and recall) and we will only talk about 3 models we find interesting.

Gamma	Farasa lemmatizer	Linear	Polynomial	RBF	Sigmoid
0.01	Using Farasa	0.818	0.028	0.767	0.700
	Without Farasa	0.765	0.028	0.683	0.593
0.1	Using Farasa	0.824	0.141	0.877	0.798
	Without Farasa	0.751	0.100	0.765	0.809

Table 10.1: Precision

Gamma	Farasa lemmatizer	Linear	Polynomial	RBF	Sigmoid
0.01	Using Farasa	0.782	0.170	0.734	0.687
	Without Farasa	0.708	0.170	0.646	0.585
0.1	Using Farasa	0.782	0.251	0.842	0.775
	Without Farasa	0.708	0.224	0.741	0.768

Table 10.2: Recall

Gamma	Farasa lemmatizer	Linear	Polynomial	RBF	Sigmoid
0.01	Using Farasa	0.775	0.049	0.722	0.670
	Without Farasa	0.697	0.049	0.622	0.561
0.1	Using Farasa	0.777	0.14	0.835	0.761
	Without Farasa	0.700	0.114	0.717	0.759

Table 10.3: F-Score

- **The best model**

We conclude from table 3 that the best model had an F-Score of 0.835 using the RBF kernel, Farasa lemmatization (as expected) and a gamma value of 0.1. In general, the performance of the Linear kernel and the RBF kernel was closely comparable. When comparing the different kernels is the same row, we observe that -except for the polynomial kernel- we cannot say that the 3 different models are statistically significant from each other in each case, but when introducing the farasa lemmatizer, there was a real difference. The best model had an F-score of 0.717 without using any lemmatization. But after introducing the lemmatization in the pipeline the model performed 12% better and that is a significant change.

The figure shows the intent prediction confidence distribution of the test set on the best model. From the histogram, we observe that the model misclassified a total of 25 intents out of 147 intents with and error rate of 0.17. But setting a NLU threshold of 0.3, the actual NLU misclassification drops to only 4 misclassifications: One with a 0.45 confidence and another of 0.65.

Now let us observe the 2 misclassification in the right spectra and try to understand why the model misclassified those 2 with a relatively high confidence.

1. The first misclassification was for the text "occ كارت طلب" for the intent "occ_health_card_req".

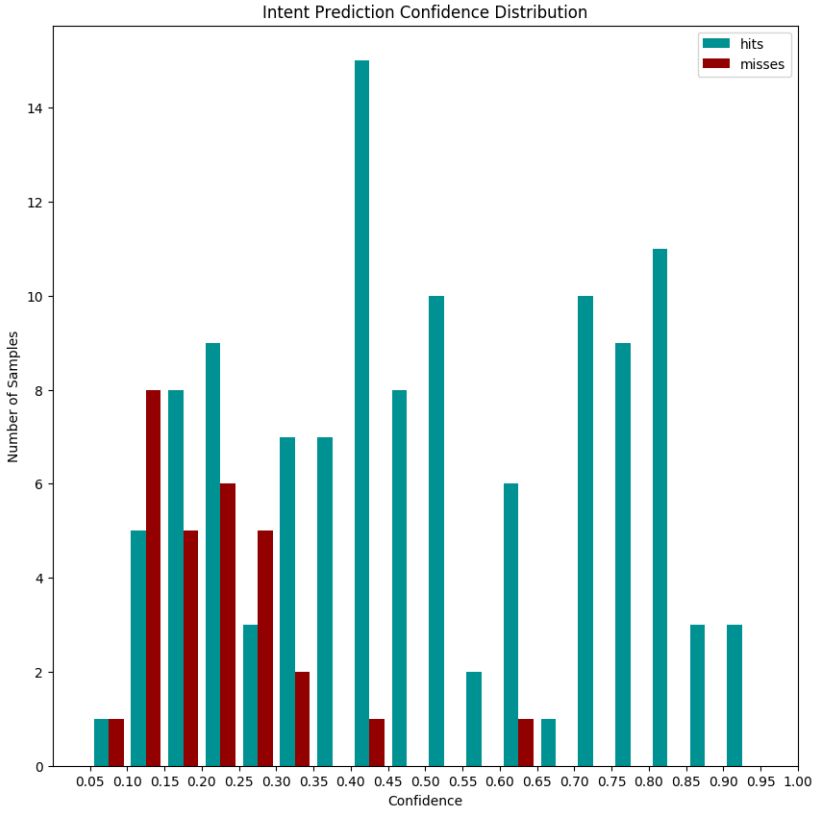


Figure 10.1: RBF kernel's Intent prediction confidence distribution

The model misclassified this query as the intent "ask_where_to_get_a_pro_card" with a confidence of 0.4218. After observing the training set we found out that the word "كارت" was repeated a total of 9 times in the "ask_where_to_get_a_pro_card" whereas it was repeated a total of 4 times only, but even more interestingly, the word "طلب" had 0 occurrences in the "occ_health_card_req" but appeared 4 times in the "ask_where_to_get_a_pro_card" intent. Moreover the word "كارت" is not an Arabic word and therefore lemmatizing the word will not help us very much here, so even if we were to rely on the feature representation of the 2 words "كارت" and "بطاقة" to be closely related to each other, that might not even work.

2. The second misclassification was for the text "بلا شك" for the intent "affirm". The model misclassified the query as the intent "deny" with a confidence of 0.635. This one is much more easier to analyze, as after examining the lemmatizer output, it was the sentence "لا شك". Having a very short count of words (2 words) one of which is "لا", a word that was repeated over 20 times in the "deny" intent. The problem here is related to the semantics of the Arabic itself, as appending the word "لا" to the word "شك" created a compound sentence to give an affirmation meaning.

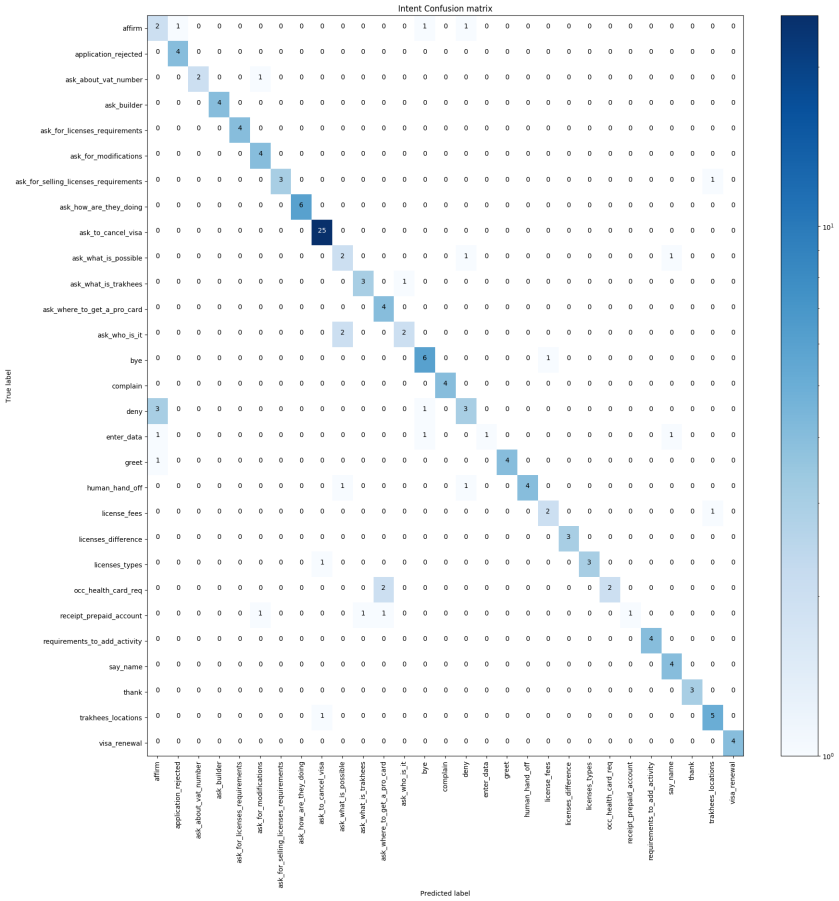


Figure 10.2: RBF kernel's Confusion matrix

The confusion matrix verifies our analysis as we can observe that the most misclassified intents are the 2 we analysed.

- **The worst model**

The polynomial kernel was the worst to perform in all the different sets of parameters. With a lowest confidence of 0.049. The reason behind this is that the data space is very sparse; hence a polynomial model can be very much complicated for this simple dataset (or any) and that might have resulted in overfitting the dataset. That explains why introducing the lemmatization step to the pipeline did not help at all as the space itself was highly overfitted.

The confidence distribution show that the model never predicts with confidence more than 0.25.

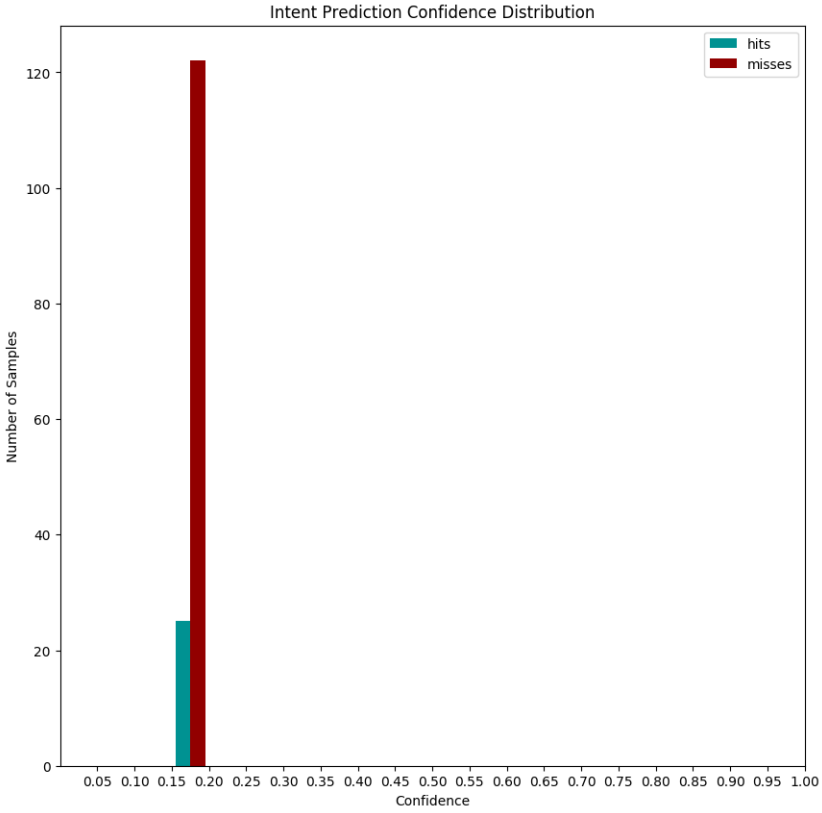


Figure 10.3: Polynomial kernel's Intent prediction confidence distribution

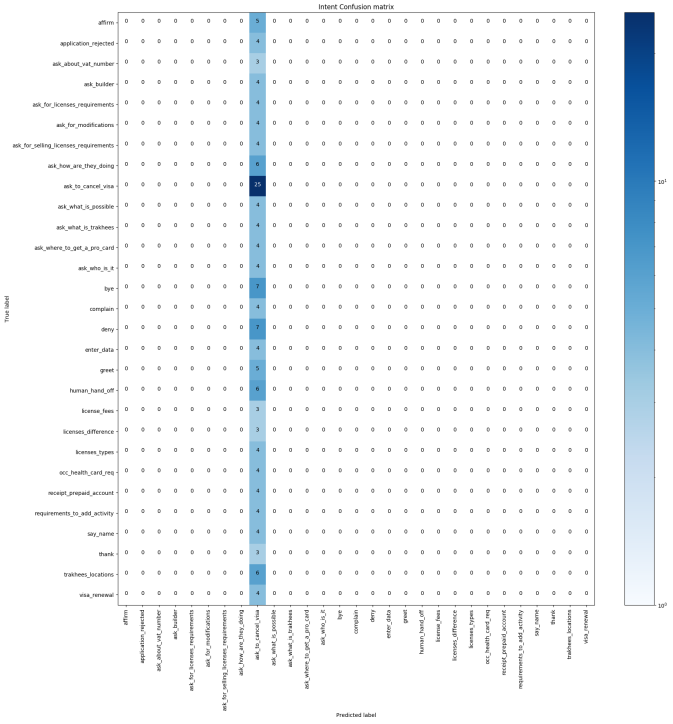


Figure 10.4: Polynomial kernel's Confusion matrix

From the confusion map we can see that the model always predicts the “ask_to_cancel_visa” intent. Looking at the training set we find the answer, the “ask_to_cancel_visa” intent is the most complicated form of question we have designed in the demo and hence we have provided a total of 97 training example for this intent alone. Being the most complicated intent (in terms of the number of training example relative to other intents), the model always infers any example to that intent as it has the most dominant term.

10.8 Performance Of The Tool

Regarding the tool we found out that tool is mostly affected by the size of bot data and bot attributes. The tool is believed to work properly if it was extended to be a web application instead of desktop application as tool boundaries are local database , local memory and local cpu which will be avoided if it turns to be a web application.

The tool was tested on different OSs (linux, windows and mac os) and on different machines. It was tested with different sizes of the bot data and it showed that it works well while bot data are less than 250 MB.

11. Conclusion

Through three months of research and trial and error testing and three other months of implementation and testing. We started our work with understanding the core components of our stack and the whats, whys and hows of that stack, we developed some english chatbots using RASA ourselves to gain more insights about RASA and the bits and pieces of how it works.

Having acquired some knowledge in RASA, we started developing many iterations of an Arabic version of the same english bot we developed earlier. Going through Farasa and FastText, we developed our first iteration after which we started working in parallel on our tool and improving the first iteration many times. And now we claim that we have developed a fully functioning desktop application that is supposed to let any end user (especially the ones with no prior programming experience) develop and deploy a state-of-the-art Arabic speaking chatbot very easily.

11.1 Future Work

Having implemented a rapid prototype, we believe the current system still has room for improvement on several ends, the following are our main suggestions:

- Improve tool by providing more fancy GUI that make it easier to build your own bot to generalize the bot to be used not just by commercial organizations but also by individuals
- Provide a way to launch chatbot over telegram, messenger and other applications to socialize the bot.
- Make bot answer more general questions like getting the weather if asked or making banking

transactions.

- Make bot accept voice chatting and reply to it
- Build web version of the tool.
- Build ML model to augment data and generate FAQ given one question For example if the FAQ is : what is my balance? This model should be able to generate similar questions like : Can you show me my balance? Is balance available for me to see it ? can you display the balance of account number `xxxx_xxxx` ? how much money do i have in my account ? This is believed to make chatbot more robust and will help reducing time to build the bot

V

Part Five

Bibliography

Articles

[Ana17] Ana. “Getting Started With ANA”. In: *Medium* (2017) (cited on page 26).

[Boc+17] Tom Bocklisch et al. “Rasa: Open Source Language Understanding and Dialogue Management”. In: (Dec. 2017) (cited on page 16).

[Boj+17] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pages 135–146. DOI: 10.1162/tacl_a_00051. URL: https://www.aclweb.org/anthology/Q17-1010 (cited on page 60).

[Che18] Hao Chen. “A brief introduction to Chatbots with Dialogflow”. In: *MAGRO* (2018) (cited on page 27).

[OBo19] Britta O’Boyle. “What is Siri and how does Siri work?” In: *Pocket-lint* (2019) (cited on page 18).

[Sto16] Lee Stott. “What is Microsoft Bot Framework Overview”. In: *Microsoft Developer* (2016) (cited on page 24).

[TO19] Maggie Tillman and Britta O’Boyle. “What is Google Assistant and what can it do?” In: *Pocket-lint* (2019) (cited on page 17).

[Woc19a] Tobias Wochinger. “"Rasa NLU in Depth: Part 1 Intent Classification"”. In: *RASA Blog* (2019) (cited on page 37).

[Woc19b] Tobias Wochinger. “"Rasa NLU in Depth: Part 2 Entity Recognition"”. In: *RASA Blog* (2019) (cited on page 41).

[Woc19c] Tobias Wochinger. “"Rasa NLU in Depth: Part 3 Hyperparameter Tuning"”. In: *RASA Blog* (2019) (cited on page 44).

[WG19] Megan Wollerton and Andrew Gebhart. “What is Alexa?” In: *CNET* (2019) (cited on page 19).

[Wol19] Sascha Wolter. “Cortana the simple way”. In: *Medium* (2019) (cited on page 18).

Inbooks

[Sha+18] Khaled Shaalan et al. “Challenges in Arabic Natural Language Processing”. In: Nov. 2018, pages 59–83. ISBN: 978-981-322-938-9. DOI: 10.1142/9789813229396_0003 (cited on page 65).

Inproceedings

[Gra+18] Edouard Grave et al. “Learning Word Vectors for 157 Languages”. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*. Miyazaki, Japan: European Languages Resources Association (ELRA), May 2018. URL: https://www.aclweb.org/anthology/L18-1550 (cited on page 60).

[Jou+17] Armand Joulin et al. “Bag of Tricks for Efficient Text Classification”. In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Valencia, Spain: Association for Computational Linguistics, Apr. 2017, pages 427–431. URL: https://www.aclweb.org/anthology/E17-2068 (cited on page 59).

Papers

[Abu17] Samhaa R. El-Beltagy" "Abu Bakr Soliman Mohammad Kareem Eissa. *"AraVec: A set of Arabic Word Embedding Models for use in Arabic NLP"*. 2017 (cited on page 31).

[Man+14] "Christopher D. Manning et al. *"The Stanford CoreNLP Natural Language Processing Toolkit"*. 2014 (cited on page 28).

[Mub16] "Ahmed Abdelali Kareem Darwish Nadir Durrani Hamdy Mubarak". *"Farasa: A Fast and Furious Segmenter for Arabic"*. 2016 (cited on page 28).

[Pas+14] "Arfath Pasha et al. *"MADAMIRA: A Fast, Comprehensive Tool for Morphological Analysis and Disambiguation of Arabic"*. 2014 (cited on page 28).

[Tul19] Amrita Sunil Tulshan. *Survey on Virtual Assistant: Google Assistant, Siri, Cortana, Alexa*. 2019 (cited on page 17).

Index

I

L

M

N

P

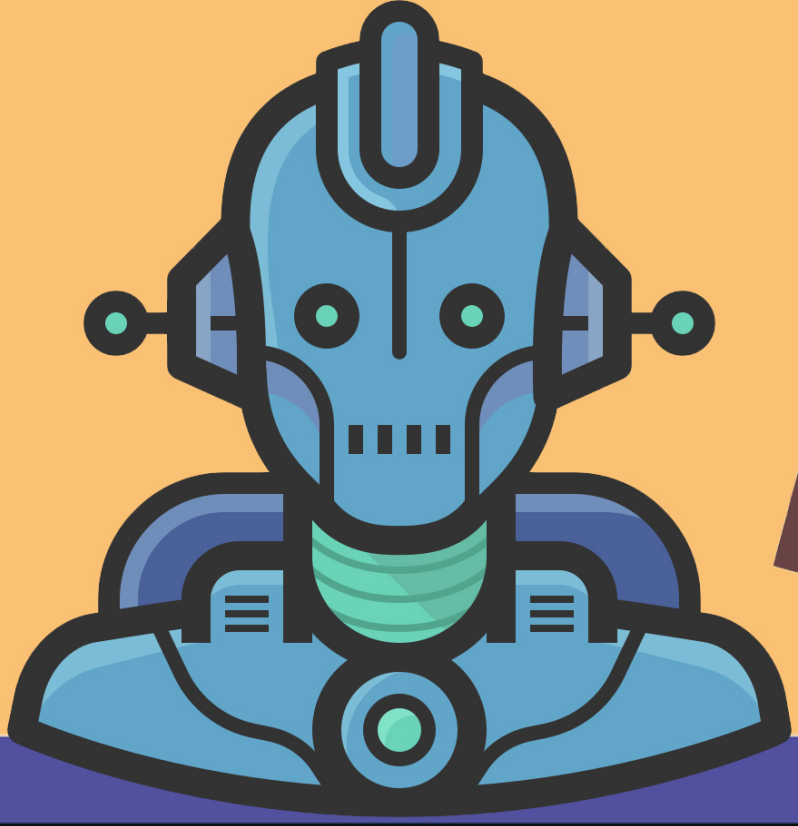
R

S

T

U

W



Graduation Project

Computer And Systems Engineering Department

2019