University of Trento

Department of Information Engineering and Computer Science

Masters Degree in
Computer Science

Final Dissertation

Bootstrapping Dialog data, annotations and models

Supervisor

Prof. Giuseppe Riccardi

Student

Marco Mancini

Co-Supervisor

Giuliano Tortoreto

Giuseppe Di Fabbrizio

Academic Year 2018/2019

Abstract

Background: Dialogue systems are complex systems carrying many research challenges and their production cycle is not standardized yet. In recent years virtual assistants are becoming much popular, with more and more people starting to use their voice for daily tasks [22][21][8][1] and forecasts [20] indicating an expansion of this market in any domain, from smart homes and cars to call centers and office works. Moreover, an important property for task-based virtual assistants is the error-handling capability since, in order to maximize the user satisfaction, an interaction must be as error-free as possible and guide the user to the completion of the task. So, this capability must be kept into account from the first stages of designing of a dialogue system. We can find projects [80] [95] [68] [79] trying to automate part or the entire process of designing and prototyping a dialogue agent, but very few start in a scenario where no dataset is available and the error-handling capability is not always kept as a requirement of the system.

Problem: Given the high demand of these systems, bootstrapping a dialogue system in absence of dialogue data is becoming much more important. Moreover, since the development of a task-based system can be a very hard task and it can be done in several different ways, a framework for facilitate the automatic prototyping of these system can be very helpful. Since these systems should maximize the satisfaction of the user, their error-handling capability should be kept into account from the firsts stages of prototyping.

Contribution: For these reasons we introduce CookieCutter, a new framework which aims to build end-to-end dialogue system in an automatic way, starting from a catalogue and leveraging Machine-To-Machine interactions, similarly to what done in [68]. Additionally, inspired by several works on the usage of feedback in building dialogue systems, we also design two experiments to collect explicit feedbacks about errors in real conversations. These feedbacks are then used from the framework to automatically detect errors during a conversation with an agent generated through the framework itself.

Method: The creation of the whole framework consists of the following phases:

1. Rapid prototyping of dialogue systems
   (a) Data generation
   (b) Data ingestion
2. Automatic error discovery
   (a) Data collection
   (b) Automatic error discovery tool development

Data generation (1a): Inspired by [54] and [68], we leverage M2M interaction to generate dialogue data. A multi-agenda-based user simulator, similar to [51], together with domain-agnostic rule-based dialogue manager interact in order to reach a specific goal (e.g book a movie ticket). Both the components exploit an underlying catalog in order to understand which information are needed to complete the task and which is their order. Data ingestion (1b): The generated data are then fed into an extended version of RASA [14] which allows us to train all the main components of a

data-driven dialogue system (NLU module, Dialogue Manager). This process produces a dialogue system.

Data collection (2a): In order to gather data for the automatic error detection tool we designed two different scenarios. In both the scenarios the user is asked to interact with the agent to reach a specific goal and he also needs to provide a feedback about errors commited during the interaction.

Automatic Error Discovery tool development (2b): The collected data in 2a have been analyzed and used to build data-driven models for error detection. We approached the error detection problem in two cases, a multi-class classification task, where we defined the error ontology, and a binary task where we collapsed the error ontology in a binary case.

Results: We report results for each of the two main contributions.

Rapid prototyping of dialogue systems: In this thesis we evaluated just the performances of the Dialogue Manager produced through the framework, since bootstrapping of the Natural Language Understanding module has been carried on by Federico Giannoni in its Master Thesis. In order to evaluate such performances we created a test-set of M2M dialogues and we evaluated the data-driven system in two domains and three different data-generation settings, varying featurization methodologies and underlying data-driven models. The provided baseline achieved a 10.22% and a 16.31% Action Error Rate (AER) in the two domains, respectively. The best data-driven DM produced through the framework outperforms the baseline and achieves the 2.92% and 4.82% AER in the two domains, respectively.

Automatic Error Discovery - Data Collection: To evaluate the goodness of the two experimental scenarios, after the testers have labelled the data, we also created a golden set and compared it with the ones from the tester, using the Cohen's Kappa Coefficient as metric. We discovered that people are not good in discriminate errors between 3 categories (NLU, DM, BOTH), with an inter-annotator agreement of 0.363 in the best case. Then we evaluated the agreement in a binary setting, collapsing the errors in two categories (NO_ERROR, ERROR), and in the best case scenario we reached a 0.862 agreement score, meaning that the binary dataset is much less noisy and can be used for Machine Learning experiments.

Automatic Error Discovery - Experiments: The performances of the data-driven models have been evaluated on the binary dataset, varying featurization methodologies. Then we compared data-driven models to three baseline models: random classifier, threshold classifier and majority class. A simpler model like the Logistic Regression was able to beat the best baseline model (threshold classifier) in the 65% of the tests with a top F1 of 72.37% (+17% F1-score w.r.t baseline). Complex neural models like the Neural Detector were not able to compete with the Logit, but the best score reached by a simpler Multi Layer Perceptron was able to reach a 70.22% F1-score, suggesting that with more data neural models could reach good performances.

Conclusions: We developed a new framework for the rapid prototyping of dialogue systems in absence of dialogue data. In order to build such framework we leveraged M2M interactions, similar to what done in [68], offering three different data-generation settings. We evaluated the DM performances in two different domains, varying data-generation settings, featurization and data-driven models and the most-performing system is able to deal with cooperative and non-cooperative simulated users, beating the baseline model. Then we designed and implemented two data-gathering experiments for errors in dialogue systems generated through the framework and we evaluated the goodness of the gathered data. We discovered that the context helps in identifying the errors and that people are better to discriminate errors in a binary case (ERROR, NO_ERROR) but they are not that good in identifying errors in a multi-class case (NLU, DM, BOTH). Different models and featurization methodologies have been tried on the binary gathered dataset, with Linear models performing better that Neural ones, probably because the low amount of data. However we argue that, with more data, also more complex models like Neural detectors and Recurrent Networks could work better than linear ones.

Acknowledgments

Developing this thesis work has been an amazing learning experience, full of difficult paths but always surrounded by wonderful and helpful people. A special thanks goes to my friend and colleague Federico Giannoni, which collaborated with me in order to build the first instance of the framework and with which I've faced and solved several hard challenges. I also want to thank VUI,Inc, the company which sponsored our internship in Boston and permitted us to work with their technologies and support. All of them have been extremely helpful and gave me the possibility to spend the best month of my life in the best city I've seen so far. I want to acknowledge Giuliano Tortoreto, Evgeny Stepanov, Alessandra Cervone, Pino Di Fabbrizio and my professor Giuseppe Riccardi for all the suggestions and support they give to me, especially in the numerous stressful and hard situations I've encountered. Finally, a huge thanks goes to my family and my friends which have been able to keep me up during a lot of bad days.

Contents

1 Introduction

In this chapter, following an overview of architectures of dialogue systems we will show some recent tool for the prototyping of dialogue systems. Then a review of the most common errors and issues in those systems is presented. Finally, the objective of the thesis work are discussed and the structure of the document is described.

1.1 Dialogue Systems ontology and architectures

According to [39] "Spoken dialogue systems have been defined as computer systems with which humans interact on a turn-by-turn basis and in which spoken natural language plays an important part in the communication". The main purpose of these systems is to provide an interface between an user and a computer-based application and, with the advances in various signal-processing technologies, these interfaces are becoming even more multimodal [67][85], looking beyond the language.

Dialogue systems can be roughly categorized in two main classes:

1. Task-oriented systems

2. Non-task-oriented dialogue systems (e.g chit-chat).

For the purpose of this thesis we will mainly focus on task-oriented dialogue systems, only giving a fast overview of chit-chat dialogue systems.

1.1.1 Task-oriented Dialogue Systems

Task-oriented dialogue systems have as main objective to assist the user to complete certain tasks such as booking a movie ticket, finding products, booking a restaurant reservation, and so on. These type of systems aim to have short conversations, intended to collect all the information needed to complete the task but they should also deal with uncooperative users.

Typically, the architecture of a task-based dialogue system is composed by many components, which are shown in Figure 1.1

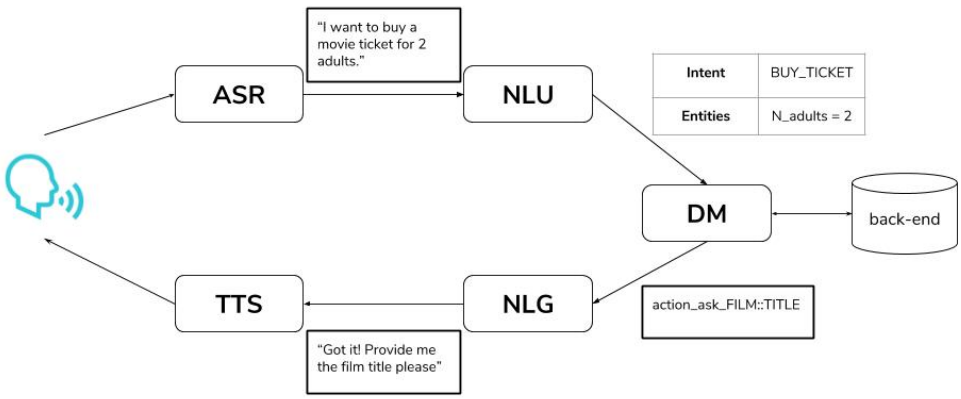


Figure 1.1: Standard pipeline architecture of a Spoken Dialogue System

The tasks each component faces are the following:

1. **Automatic Speech Recognition**. This module takes as input a voice signal and enables the recognition and translation of such signal into text. The text is then passed to the Natural Language Understanding component.

   Speech recognition has been researched since late 1950s and it faces several challenges [70] [70], from being able to work in real-world noisy scenarios to all the problems related to the speaker variability (e.g accents, speed of the speech,..). Gaussian Mixture Models (GMM) [52], together with Hidden Markov Models (HMM) have been the most performing models for this tasks for several years, but new deep learning techniques[46] [107] have been shown to outperform the GMM-HMMs on a large variety of datasets.

2. **Natural Language Understanding**. Given an utterance as a text, this module usually maps it into two dimensions.

   The first is the intentional dimension, which expresses the user intention (e.g "Request", "Inform", "Book a movie restaurant") in a predefined intent space. As second sub-task this module performs slot-filling. Slot-filling is a challenging task, defined as a sequence labelling problem where the words in the analyzed sentences are categorized with semantic labels (e.g in 1.1 the "2" is tagged as entity N_ADULTS). For this task, the input is a sequence of tokens and the output is a sequence of slots, one for each word. Both intent and slots are passed as input to the dialogue manager.

   Both the tasks of this module are very challenging. Firsts ideas for both the sub-tasks were purely rule-based but the advent of Support Vector Machines [34] [50] and CRFs [77] [61] pushed the state of the art ahead. In recent years, deep learning techniques have been successfully applied to both intent classification and slot-filling [105] [63] [97], beating the previous techniques.

3. **Dialogue Manager**. This module is usually composed by two sub-modules:

   - **Dialogue State Tracker**. This sub-module maintains a representation of the state of the dialog at each turn. A standard approach to represent the semantic dimension of the state of the dialogue is the so called semantic frame. According to [27], a semantic frame represents an event or scenario, and is composed by frame elements participating in the event. The most common representation for dialogue state tracking is the one composed by all the slot-values recognized so far by the system, and continuously updated through the interaction. Traditional methods adopt hand-crafted rules to select the most probable result [62] but, with the advent of challenges such as Dialogue State Tracking [102], much more effort has been put in statistical methods able to cope with noisy conditions and ambiguity. Henderson et al. in [45] introduced deep learning in belief tracking showing transfer learning capabilities to new domains. Others [65] [64] tried multi-domain Recurrent Neural Networks and Neural Belief Trackers.
   - **Action Selection Policy**. Conditioned on the representation of the state provided by the DST, this module generates the next available system action. A system action is an intermediate representation before the final system utterance. It could either trigger an operation on the back-end (e.g select an item from the database, insert an item,...) or being a simpler action (e.g acknowledge what the user just said). This action is then passed to the Natural Language Generation module.

     Also in this case traditional methods adopt hand-crafted rules but, in the last decade, data-driven models have been found to be very effective. Researchers used either supervised learning [73] or reinforcement learning [78] [89] to optimize this task, but also hybrid methods have been tried [44] [43]. An interesting direction is the usage of user simulators [51] [54] to train and fine-tune these models.

4. **Natural Language Generation**. This component takes the abstract system action into natural language surface utterances, which are passed as input to the last component.

   As stated in [90], a good NLG module should keep into account factors as adequacy, fluency, readability and variation. Traditional approaches makes use of sentence planning [92] [91],

mapping the utterances in intermediary forms with a tree-like structure and then converts the structure into a final response. More recent works make use of neural networks such as RNNs to tract this problem [100] [99], reaching good results compared to rule-based techniques.

5. **Text to Speech**. This module converts the utterances coming from the NLG component into voice signals.

This architecture has been used and studied since '90s, but it presents some issue.

A major problem, which will be discussed in depth later, is the error propagation. Since each component can produce errors (e.g misinterpretation of the voice input by ASR, slot-values not recognized by NLU, wrong action selection by DM,..) and their output is the input of the subsequent module, the pipeline is subject to an increasing overall error, while the information are processed though it. Another problem of this approach is the fact that each component requires a significant amount of manual effort, from the data collection to the fine-tuning of all the models in the pipeline. Moreover, extending systems with such architectures with new functionalities can easily end-up with the re-training of the whole architecture, which is expensive.

These reasons, together with an increasing availability of high volumes of data and new deep learning techniques, inspired the birth of the end-to-end architecture. Instead of the traditional pipeline, this architecture uses a Deep Neural Network (DNN) trained in an end-to-end setting and interacting with an external knowledge-base. This architecture avoid the major problem of error-propagation, since all the components of a standard pipeline architecture are collapsed into a single module. Moreover, DNNs can be fine-tuned, allowing developers to add new functionalities without re-training the whole system, even if they must face the problem of catastrophic forgetting [75].

Neural networks lack interpretability and cannot be easily constrained as others models and techniques, so they are mostly used for chit-chat dialogue systems. However, we can find works such as [28] [101] introducing network-based ent-to-end trainable task-oriented dialogue systems, using encoder-decoder [33] architectures for their DNN models and Supervised Learning as training technique. Li et al. in [55] tried to model the end-to-end learning problem in a Reinforcement Learning setting, leveraging an agenda-based user simulator [51] and also simulating a noisy environment. Since the lack of constrainability, some other work tried to constraint the output of end-to-end dialogue systems either featurizing domain knowledge contained in external knowledge bases [110] or post-processing the final output according to masking rules [103].

1.1.2 Non Task-oriented Dialogue Systems

Non task-oriented dialogue systems, sometimes reported as chit-chat dialogue systems, focus on conversing with human on open domains, without pursuing a specific goal.

Generally these kind of bots are implemented either by retrieval-based methods or generative methods. Retrieval-based methods usually select a proper response from a repository, leveraging response selection algorithms and they are characterized by informative and fluent responses. On the other hand, generative models are able to generate responses which could be never seen in the training corpus, so having a better generalization capability but carrying problems like the fluency or the coherency of the generated responses. Since the scope of this thesis are task-oriented dialogue systems, in the following we will just give a brief overview for both the above-mentioned methods.

Neural Generative Models

In recent years the huge amount of interactions available in social media websites like Facebook and Twitter gave birth to data-driven models able to ingest such data, with the objective of reproduce the behaviour of a real user. One of the first studies in this direction [74] tract the problem as a Machine Translation (MT) problem, using phrase-based Statistical Machine Translation to face the task [111]. However, even if the translation is a very difficult task, researchers found the response generation a much difficult problem and, with the advent of Neural Networks and Deep Learning, sequence-to-sequence models started to be the new research direction.

Formally a sequence-to-sequence model tries to solve the following problem. *"Given a source sequence (message) X = (x1, x2, ..., xT) consisting of T words and a target sequence (response) Y =*

(y1, y2, ..., yT) of length T, the model maximizes the generation probability of Y conditioned on X: p(y1, ..., yT—x1, ..., xT). [32].

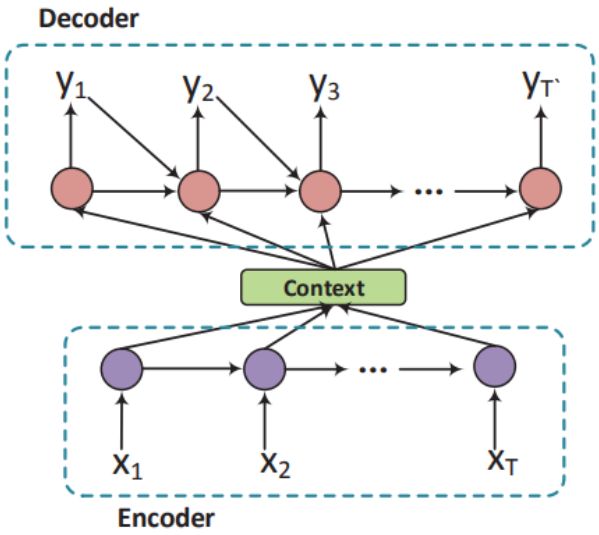


Figure 1.2: Encoder-decoder architecture.

The most common Neural architecture for this task is the encoder-decoder, shown in Figure 1.2. An encoder network is used to ingest a sequence X of words and the hidden representation of this network is used to represent the utterance itself. This representation is then used as feature from a decoder network, which reconstructs Y word by word. Usually for both the encoder and decoder a Recurrent Neural Network is used [93], but we can find other works trying to use CNNs [108]. Very good results have been reached using an attention mechanism, presented by [26], which conditions the output of the decoder network using an additional context vector indicating, for each output word, which are the most relevant input words.

The main challenges these systems face are the following:

- **Integrate dialogue context.** The ability to generate responses keeping into account the previous utterances is the key to build dialogue systems able to engage real interactions. In this direction, very interesting results have been reached using hierarchical neural networks [82] and continuous representation of words and phrases such as word2vec [41], glove [41] and ELMo [69].

- **Response diversity.** Since sequence-to-sequence models tend to output not meaning-full and trivial responses for most of the cases, the response diversity is an hard problem in such systems. Some work tries to leverage the beam search, an approximate inference algorithm to decode output sequences for neural sequence models, augmenting it with dissimilarity terms [98] or changing the standard approach in a stochastic beam procedure [84]. Others focused on the introduction of latent variables conditioning the decoder network [30][83].

- **Topic and Personality.** In human-human interactions, the generation of the response is usually related to the topic of the conversation and to the personality of the users. Some work shows how embedding topic information using techniques such as LDA [104] improve the performances. Other tries to embed different emotional information during the generation of the response [112] [25] in the decoder network.

The above-mentioned challenges are the main ones, but others like the integration of outside knowledge bases and the evaluation of such systems are important too.

Retrieval-based methods

These methods were born before the Neural Generative Models and they choose a response from a candidate set of responses so, in contrast to sequence-to-sequence models, they have no generalization

capability but are characterized by more informative and fluent responses. The most of the effort, in these systems, is put in the message-response matching algorithms, which can be subdivided in two main classes:

1. **Single-turn Response Matching**. The very first studies of these systems mainly focused on the response generation looking just at the last single message provided by the user, meaning that no context is kept into account. What typically happens is that both the user message and the response are encoded as vectors and a matching function is used to compute a matching score between those vectors, then a ranking strategy is applied and the higher-rank response is selected. Lu et al. in [57] proposed a Deep Neural Network model for modelling the matching function, combining the localness and the intrinsic hierarchy of the utterance structure. In [48] they show how is it possible to learn the representation of message and response using a Deep CNN and then they compute the matching score using a MLP neural network.

2. **Multi-turn Response Matching.** Recently much more attention has been put into the embedding of the context of a conversation for the selection of the most appropriate response. In these models the response selection is based on the current message and the previous utterances. In [56] they use RNNs to encode both the context and the candidate response into two vectors and then computed the matching degree score based on these vectors. Others [106] applied different strategies to select the most appropriate context and combined it with the current message to form a reformulated context.

Hybrid Methods

Neural generative models and retrieval based models can be combined, trying to exploit the generalization capability of the firsts while keeping fluent and informative responses. For example [88] and [72] tried to combine both the methods. With their approach the retrieved candidate, in addition to the original query, is fed to an RNN-based reply generator, so that the neural model is aware of more information. The generated reply is then fed back as a new candidate for post-reranking. in [81] they created an ensemble of natural language generation and retrieval models, including template-based models, bag-of-words models, sequence-to-sequence neural network and latent variable neural network models. By applying reinforcement learning to crowdsourced data and real-world user interactions, the system has been trained to select an appropriate response from the models in its ensemble.

1.2 Trends and technologies for building task-oriented Dialogue Systems

In recent years consumers have been offered to engage with a company through their virtual assistants (e.g Amazon, Google, Apple). As different studies noticed [10] [22] the global Intelligent Virtual Assistant (IVA) market size is growing very fast. Just to give an insight, in Figure 1.3 it is possible to see how the IVA market size, in 2014, was around USD 181 million, in just two years it raised up to 1005 million dollars and it is expected to grow more and more, with an increment in all the families of usage, from individual users to small and medium enterprises. Others sources [21] [1] claim that 39 million American people, in January 2018, own a smart speaker, with an increment of 128% from January 2017 and Amazon's echo speakers being the leader in this market. A recent report [1] shows how people are starting to use Virtual Assistants more and more, replacing old habits such as FM radios, smartphones, televisions and so on (Figure 1.4).

But the increment of VAs usage is not limited to daily activities, in fact other studies show how they are becoming more and more used also in office works. In [15] they claim that Robots will eliminate 6% of all US jobs by 2021, starting with customer service representatives and eventually truck and taxi drivers. As stated in [13] the main factors that contribute this market to growth are:

- **Instant gratification**: Providing instant, personalized and intelligent answers to any question the consumers have is one of the main objective of any company.

- **Mobile phone proliferation**: The usage of mobile phones has increased incredibly in recent years, and people use VAs on their smartphones for everything.
- **Research advances**: Recent advances in AI and NLP fields allow companies to build high-performant production systems.
- **Social media**: The increasing popularity of social medias give an easy access to the crowd.
- **Operating costs**: Large contact centers costs a lot and according to Forrester researches, a transaction done through a live agent costs $12 per call. Adopting VAs can cut these costs significantly.

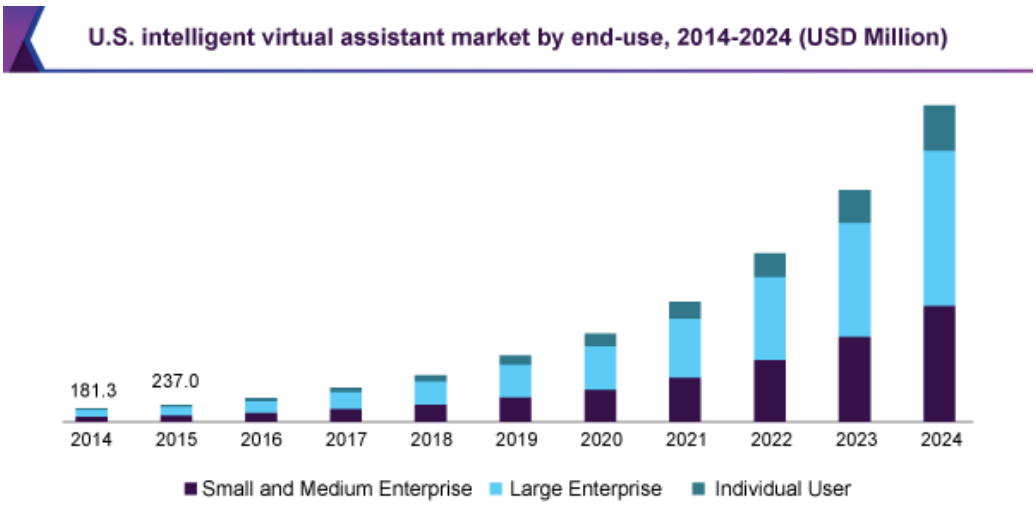


Figure 1.3: Intelligent Virtual Assistant market size from 2014 to 2024. From [10]

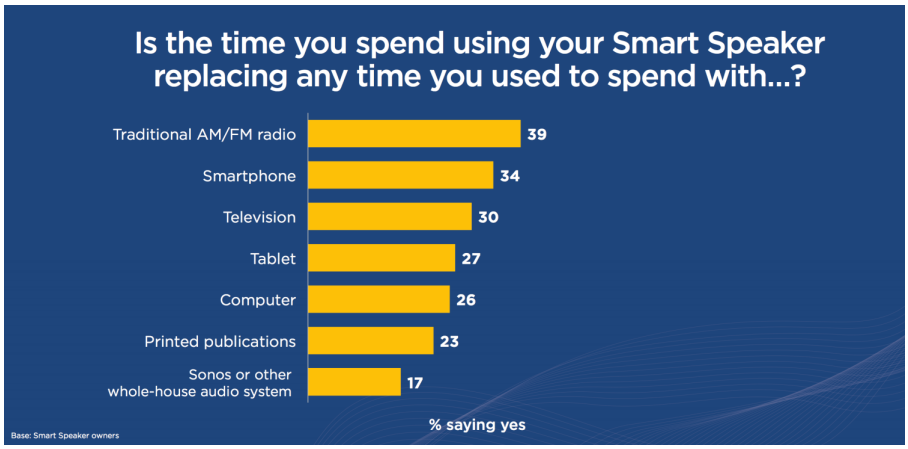


Figure 1.4: From [1]

This ever-growing paradigm demand for online self-service, self-reliance and rapid solutions for production systems.

However, these systems, in order to have in-production performances and being able to efficiently interact with an end user require a large amount of high-quality data, which is hard and expensive to collect. Moreover, as stated in 1.1.1, the designing and implementation of such systems requires a lot of manual effort and expertise.

For this reason, in recent years, more and more platforms for rapidly prototyping dialogue systems appeared online. Since we are focusing on task-oriented dialogue systems and, in the thesis work, we will make use of a specific tool for building these systems, in the following we are going to list and describe the major ones.

DialogFlow

This platform [6] was acquired by Google in 2016, connects with user on Google assistant and other devices and messaging apps and it can be used for free. By default, it offers an ASR module capable of recognizing 15 languages and customizable with additional data provided by the developer. Regarding the NLU module, they have built-in knowledge on topic like weather, widsom and casual talks, meaning that the developer shouldn't bother about those tasks. Instead they give the developer the possibility to build their NLU annotation schema (intent + entities) and feed to the tool data labelled with such schema so that, AI and ML models on the back-end can be trained. For what concerns the Dialogue Management, they allows the developers to define the actions which can be either only textual or more complex ones (e.g query to DB, webhook,...) and each one of these actions is triggered by an intent. As policy management they offer a frame-filling capability, working on frames defined by the developer and composed by the required entities. Moreover they also have a mean for the context, giving the developer the possibility to define them based on intents and to relate them in order to allow multiple-context dialogues. Concerning the TTS and NLG modules, DialogFlow allows both pronunciation and voice customization through the google TTS API and the developer defines the prompts.

Wit.ai

This platform [23] was acquired by Facebook and it can be used by developers for free. Their default ASR module doesn't offer customization capabilities, but they claim to be able to recognize 39 languages [24]. For the NLU module, as Dialogflow, they offer pre-trained models for some topic, allowing an extension of the NLU annotation schema with intents and entities defined by the developer. They give the developer the possibility to label and pass labelled examples for the new intents and entities defined. The DM, as the TTS and NLG module, are completely missing. In fact this platform does not handle actions, instead they only send a response including extracted entities. They only give the possibility to add some type of context (date/time, location, state names and entities) to api calls, but they do not offer a mean to manage it.

Amazon Lex

This [2] is a service developed by Amazon and released in 2017. The default ASR module is not customizable and currently supports only English. They allow the developer to define their NLU annotation schema, following a specific format and they follow a full data-driven approach, using pre-trained models and transfer learning to bootrstrap a NLU module requiring some additional examples provided by the user. The dialogue manager policy is full rule-based, defined through functions of the type if-then-else and the context is tracted as the full set of entities with current values, sent back and forth through request and responses. Concerning TTS and NLG module they offer voice and pronunciation customization, but they only support English. In contrast to the previous tools, it offers a 1 year trial for free, then they ask for money at each query.

IBM Watson

It [9] was born in 2011 to answer questions to the quiz show *Jeopardy* and, in 2013 IBM announced the first commercial application based on Watson. Starting from the ASR, they offer a good customization capability giving the possibility to customize the module with additional data. For what concerns the NLU module, they offer default pre-trained models for different topics and, as others, give the possibility to add NLU classifier based on additional data to be provided, tagged with both entities and intents. Regarding the DM, they use a rule-based one defined by the developer through a User Interface. This UI allows the developer to create a network of possible flows of conversations, based on recognized intents. Moreover they facilitate the creation of flows asking for frames to be filled during the conversations, and the tool automatically creates the logic for filling such a frame so that, the developer, just need to define what to do when the frame is completed. They also make use of context, defined as the set of entities recognized so far and updated through the interaction. As many

others, they offer voice and pronunciation customization, supporting 7 languages. The Lite version of the tool is free, but constrained by 100 queries per month, additionally a Standard or Premium account can be paid.

Nuance Nina

Nuance is a software technology company which provides speech recognition technologies and artificial intelligent based solution for virtual assistants. They offer Nina, an automated virtual assistant used for customer self service and allowing task-based interactions. Their ASR technology is one of the most efficient, providing a set of 86 predefined languages and customization capability for the customer. Concerning the NLU, they offer a central tool called Nuance Experience Studio, which gives the possibility to build both grammar-based models and data-driven models, providing either rules or data and they provide pre-trained models and features for more than 75 languages and dialects. Regarding the DM strategy, they offer several pre-defined editable dialogue modules which encapsulate the behaviours of a DM in order to collect pieces of information from the user. Finally, they embed a TTS module supporting 53 languages and 119 voices with a pronunciation customization.

RASA

This last framework [14] is an open-source tool allowing developers to build contextual chatbots. They don not offer an ASR module, but their framework is composed by two main components, RASA NLU and RASA Core.

RASA NLU is a library for natural language understanding with intent classification and entity extraction. As all the previous ones they offer a way to train data-driven models with labelled data. The developer needs to define its NLU schema and have labelled data. Differently to others, they do not offer any pre-trained model for well-known topics, but they offer a good set of extendable ML techniques for NLU tasks.

RASA Core is a chatbot framework with machine learning-based dialogue management. As a DM, its objective is to decide which action to trigger at each turn, based on the NLU output and the context of the conversation. Differently from all the previous frameworks, both the action definition and DM policy shaping is done in a programmatic way, offering the developer an high customization capability and giving him the possibility to develop either rule-based dialogue managers or data-driven ones with a set of different ML models already provided by the framework and easily extendable. The context is totally defined and managed by the developer in a programmatic way.

With respect to the other frameworks, RASA doesn't offer any TTS module, and the NLG is intrinsically defined through the programmatic definition of actions.

1.3 Errors in task-based dialogue systems

As stated in section 1.1.1 each component of a task-oriented dialogue systems faces several challenges. One of the most important challenges for the overall system is to deal with uncertainty and errors. As stated by Teigen in [96], uncertainty partly comes from the ambiguity of natural language. Additionally, in dialogue systems, a great deal of uncertainty and errors come from the different modules composing the whole system. If we think about standard task-based dialogue systems, the pipeline architecture doesn't facilitate the reduction of errors. In fact each component can produce errors and, since its output is the input of the subsequent one, the system accumulate errors as the information passes through it, increasing the probability of replying to the user with a non-satisfactory response.

However, how pointed out by Brown in [29], apparently satisfactory human-human interaction may often take place without the listener having a full interpretation of the words used. This because the same information may be conveyed in different ways within the same utterance or may be repeated by the speakers to ensure understanding. Another cause of this behaviour is the context-dependence of language use, meaning that the context may be used to reduce uncertainty and fill in the gaps. Additionally, as pointed out from Clark in [42], during human-human interactions there is a collaborative process of avoiding and recovering from miscommunication that often goes unnoticed.

Due to the nature of a dialogue system, it can never know for certain what the user is saying, instead each component of the system produces hypotheses. According to [87] errors in spoken dialogue systems may be classified into two broad categories: under-generation and over-generation of interpretation hypothese and are directly correlated with miscommunications in human-human interactions. In particular the under-generation is directly related to the notion of non-understanding, which means that the listener fails to obtain any interpretation at all, or is not confident enough to choose a specific interpretation. The second category instead is related to the notion of misunderstanding, meaning that the listener obtained an interpretation not in line with the spearker's intention. In contrast to misunderstanding, non-understandings are noticed immediately by the listener, while misunderstandings could never be detected at all, however both the problems may be present in the same utterance.

In the process of creating a dialogue system error handling should be kept into account as the development of any other module, and it should be considered in all parts of the system in order to handle the consequences of under-generation and over-generation. Usually, the process of error handling is subdivided in error-detection and error-recovery. Error-detection means that the dialogue system is able to understand, at any time, if an error occurred. With the error-recovery we mean some kind of strategy which, based on the error commited, guide the dialogue system through a recovery of the error.

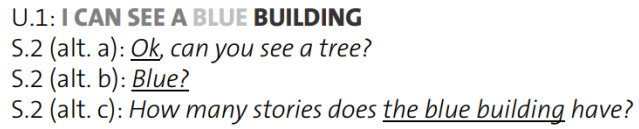


Figure 1.5: Example of error recovery strategy. U refers to the user and S to the system. The number indicates the turn number. The grayscale capital letters represent the speech recognition word confidence scores. Took from [87]

.

Just to give an idea of possible error-recovery strategies, in Figure 1.5 it is possible to see three possible responses that the agent will provide. In the first alternative (alt.a) the system assume complete understanding of the utterance and proceed with asking additional information but, with the second alternative (alt.b) it suspects that some part of the utterance is not correct and asks for a clarification. With the third and last alternative (alt.c) the system assume understanding but provide a verbatim repetition as part of the response. Obviously, this is just an example, and more works about error-detection will be showed in the next chapter.

1.4 Objective and challenges

Given the ever-growing paradigm of the usage of VAs showed in 1.2 and the importance of error-handling explained in 1.3, we want to provide:

1. A framework to rapidly bootstrap a task-oriented dialogue system. We want this framework being able to work in any domain and bootstrap a dialogue agent in absence of dialogue data. Instead, the framework will start from a catalogue (e.g MySQL database) and will guide the developer through the bootstrapping of all the components of a data-driven dialogue system.

2. A tool for error-detection in bootstrapped agents. This tool will use ML and AI techniques to detect errors during an interaction with an agent bootstrapped by the above-mentioned framework. The creation of this tool includes the designing of two experimental scenarios for data gathering about errors happening in human-machine conversations.

Challenges

The challenges we will face to create the framework are the following:

- **Reach in-production performances**. This means that we want high performances in each of the component of the task-based dialogue system.
- **Easy-to-use**. We want to cut as much as possible the effort the developer will put in bootstrapping the dialogue agent. Meaning that we do not want him to write lines of code or being involved in long designing processes.

Regarding the error-discovery tool, we will face the following challenges:

- **Tract uncertainty of natural language**. As stated by Teigen [96], uncertainty comes from the ambiguity of natural language and it is a source of errors during interactions. We need to keep into account that uncertainty when developing ML and AI models to detect errors.
- **Deal with error-propagation**. As explained in section 1.3, a great deal of uncertainty and errors come from the different modules composing the whole system and we need to keep into account all these signals to build an high-performing error-detection tool.

1.5 Structure

The thesis is structured as follows, in the next chapter we present the related work, both to rapid prototyping of dialogue systems techniques and error detection in dialogue systems. In the third chapter we describe our framework, mainly focusing on the modules developed by the author of the thesis, since the whole framewok is the product of a collaborative effort. Then, in the fourth chapter, we show how we evaluated the first instance of this framework under different settings and scenarios. In the 5th chapter we describe how we tackled the problem of error discovery in dialogue systems produced through the framework, describing the data-gathering process and the Machine Learning task we face. In the sixth chapter an evaluation of all the models for the automatic error detection is provided. Then, conclusions are depicted and, in the last chapter, acknowledgments are provided. At the end, appendix and bibliography are provided.

2 Related Work

In this chapter we describe the related work of each of the two man contribution of the thesis. First we focus on the rapid prototyping of dialogue system and the usage of M2M dialogues for such task. In the second part we describe the previous studies on the automatic detection of anomalies, mainly focusing on errors in dialogue systems.

2.1 Rapid Prototyping of Dialogue Systems

Building dialogue systems is an hard task, composed by many different sub-tasks. We can find different studies [38] [40] [113] exposing a lot of challenges from robust speech recognition to natural response generation. Several works, since from '90s, are trying to ease the approach to these challenges, building tools to speed up the prototyping of dialogue systems. Sutton et al., in his work [95], built a toolkit which supports rapid-prototyping and evaluation of dialogue systems, supported by a Graphical User Interface and an iterative procedure which facilitates the whole design process. Similar to what done by Sutton et al., also Klemmer et al. [80] developed a tool with the same purpose of easily prototyping Speech User interfaces but, since these are tools from late 90s, the research was much more focused on automatic speech recognition techniques and a unique finite state machine, graphically designed, was the core of the dialogue strategies.

2.1.1 Rapid Prototyping of data-driven Dialogue Systems

In recent years the idea of automatically generating dialogue systems has been carried on, with a strong interest towards the usage of data and data-driven models [109][66][54], moving the technology ahead in each component of the standard pipeline architecture.

An interesting work from Scheffler et al. [54] shows the first idea of using Machine-to-Machine interactions to automate part of the designing of policy strategies, using a three component environment composed by an user simulator, a noisy channel and a dialogue manager. As many others [78] [89][76], they train the dialogue manager in a Reinforcement Learning setting but they make use of a user simulator to automatically train a data-driven dialogue manager for frame-filling tasks. The user simulator they built is based on a goal and works both at intentional and sentence level, but needs to be learnt from data labelled especially for it also requiring expertise for the designing of the specific label-sets. The problem with this approach is that, even if the developer doesn't need to handcraft rules for the dialogue system, he needs to engineer a state representation for the RL task. Moreover this approach can be taken only in case we have a corpus to start with, since the user simulation is fully data-driven.

Our task As Scheffler et al. [54] our frameworks makes use of machine-to-machine interactions for a fast bootstrapping of a dialogue agent. We let an user simulator able to work both at intentional and sentence level communicate with a domain agnostic dialogue manager, for completing a frame-filling task. But the main differences are:

- We start without requiring a corpus but just a catalogue. They must have a corpus to start with.
- Our simulator is a multi agenda-based user simulator similar to [51]. It is rule-based and domain-agnostic, characterized by a set of indicators which determine its behaviour. Their user simulator is completely data-driven and require knowledge of the underlying domain.
- We generate all the data needed by the models of the dialogue system exploiting the interaction. They leverage the interaction to directly train a dialogue manager in a RL setting.
- We don't model the noisy environment, they do it.

Another interesting work in this direction is the one done by Pararth Shah et al. in [68]. They propose a framework combining automation and crowdsourcing to rapidly bootstrap end-to-end dialogue agents for goal-oriented dialogues in arbitrary domains. In this framework the environment is composed by two main components, an agenda based user simulator [51] and a finite-state machine based dialogue agent. They focus on database querying applications involving a relational database which contains entities that the user would like to browse and select through a natural language dialogue. Differently from [54], they don't require a dataset to start from, instead the input to the framework is a task specification obtained by the dialogue developer. The task specification is composed by a schema of the relational database and an API client to query in order to get lists of matching candidate entities for valid combinations of slot values. This task specification is fed to the framework and:

1. A set of scenarios is generated as goals and profiles for the agenda-based user simulator.
2. Based on the scenarios the interaction between the user simulator and the domain-agnostic rule-based dialogue manager starts, producing dialogue outlines.
3. A template utterance generator maps each turn of the dialogue outlines to templates.
4. The generated templates are fed to Amazon Mechanical Turk in order to be paraphrased by turkers.

The whole process end up with a dataset of dialogues tailored to the entities appearing in the schema of the relational database passed as input. According to them, the datasets produced by the

framework can be used to train any component of the standard pipeline architecture of a Dialogue System, or end-to-end models but no system is trained and evaluated with those data.

Our Task. We built a similar framework with respect to what did in [68]. Our framework also leverages M2M interaction using the same two components to generate high quality data for training the components of the entire pipeline of a dialogue system. The input to the framework is still a catalogue, but in our case we also keep into account the relations between entities during the generation of the data, instead they collapse everything in a flat representation of the catalogue. Additionally to the data, we also automatically produce the python code needed by RASA [14] (e.g Custom Action code for back-end call) so that, at the end of the process, we have a bootstrapped dialogue system and not only data to train model on.

2.2 Error detection in Dialogue Systems

The error detection in dialogue system is a well known problem. In the literature we can find several works [58] [49] studying how errors happen both in human-human interaction and human-computer interactions and designing different error-recovery strategies [37].
Typically, spoken dialogue systems, rely on confidence information from the ASR module to assess the reliability of their inputs, however, as stated by [86], there is not a simple one-to-one relation between low confidence scores and correct recognitions. However *"there are also other variables (such as consequence of task failure) that must be taken into consideration, in order to get a robust and efficient dialogue that the user does not perceive as burdened with errors and confirmations."* For example, we can find several works [53] [59] in the literature trying to address the problem of error-detection using prosodic features such as intonation, tone, stress and so on, but all of these works mainly focus on ASR errors, without leaving space for the other components of the pipeline.

An interesting work done by E.Krahmer et al. [36] makes use of user's cues (positive and negatives) to classify a turn of an interaction either as error or as correct. After having studied the process of error detection in human-human communication and compared it with human-machine interactions, they extracted a set of features indicating positive and negative cues of the user at each turn, such as lenght of user utterance, whether or not the user gave an answer to a system question, the presence of confirmation markers, and so on. These features are then used to label a dataset on which a memory-based ML model run to perform a binary classification of the turn. They showed that these features help in classifying the errors, beating a majority class baseline model of 30% accuracy.

Our Task. As E.krahmer et al. [36] we approach the problem in a binary setting, but we also performs experiments in a multi-class scenario where we want to discriminate between NLU and DM errors. Inspired by them, we makes use of some user cue feature such as lenght of utterances and the number of slots already verified by the system. We don't use all of the features used by them since the labelling task would require an expert to label each turn, instead we want a lightweight labelling procedure, in order to gather a good amount of data without much effort and costs.

Another interesting work done in this direction is the study done by Walker et al. in [60]. Instead of using prosodic and cues features, they focus on the detection of SLU errors testing different feature sets coming from all the components of the pipeline, such as ASR, NLU and DM. In particular they show how all the features extracted from the all the components greatly improves the error detection, giving a +23% of accuracy with respect to a majority class baseline.

Our Task. Inspired by Walker et al. [60] we want to exploit feature sets created with information from all the components of the dialogue system, but we will not engineer features from the ASR module, since we do not have it. Differently by them, we do not classify only SLU errors, instead we will work both on a binary case and a multi-class task where the errors to discriminate are between NLU and DM.

3 CookieCutter: A catalog2dialogue framework

Disclaimer

The first instance of this framework is the result of a collaborative effort with Federico Giannoni[1], and it has been designed and partly developed during an internship at VUI,inc.

In this chapter we will describe the framework and the various approaches applied to create it. In the firsts sections we will start re-capping the objectives and we'll give an high level description of the whole framework. Since our framework leverages RASA, a deeper analysis of the RASA Core framework will be presented. Then, since this work is the result of a collaborative effort, more focus will be put on the components developed by the author of the thesis. Finally we will show how the components interact in order to reach the objective.

3.1 Objective

Re-capping, the objective is to rapidly bootstrap a task-oriented dialogue system in a specific domain and for specific tasks the developer wants its virtual assistant to be able to carry on. We want this framework being able to work in any domain and bootstrap a dialogue agent in absence of dialogue data. Instead, the framework will start from a catalogue, representing the domain of interest, (e.g MySQL database) and will guide the developer through the bootstrapping of all the components of a data-driven dialogue system, with a minimum manual effort.

Technology used

As programming language for the codebase we decided to use Python 3.6, since it offers a large set of ML and AI libraries and is also the programming language mainly used by the company we were working with. Our framework is built on the top of RASA. We decided to use RASA mainly because it is the tool used by the company which hosted us during the internship. With respect to the other frameworks presented during the introduction 1.2, RASA is open-source and offers an easy extendibility.

For this reason, in the next section we are going present and describe the main components, needed to completely understand Cookie-Cutter.

3.2 RASA Core Structure

According to RASA [14], RASA Core is a chatbot framework with machine learning-based dialogue management. So, as a dialogue manager, at each turn its objective is to decide which action to trigger, based on the NLU module output and the context of the conversation.

In figure 3.1 it is possible to see an high-level view of the overall architecture. The diagram shows the steps of how a RASA Core app responds to a message, they are:

1. The message is fed to an interpreter, which extracts the original text, the intent and any entities found.

2. Text, intent and entities are passed to the tracker, which updates the state of the conversation at each step.

[1] Federico Giannoni - federico.giannoni@studenti.unitn.it

3. The policy is fed with the current state of the tracker.
4. The policy decide which action to take next.
5. The action is logged by the tracker
6. A response is sent to the user.

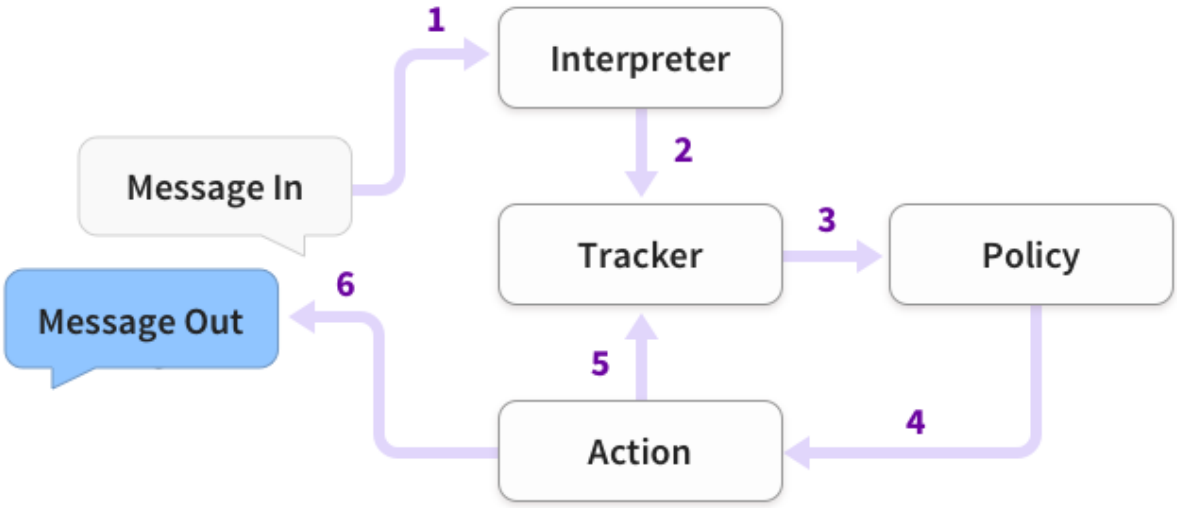


Figure 3.1: RASA Core high-level architecture. From [14]

3.2.1 Domain File

The very first component which RASA Core needs to build a dialogue manager is the domain file. This is a YAML file, defining the universe in which the bot operates and must contain the following information:

- **Intents**. As already explained, an intent represents the intention of the user (e.g BookMovieTicket = the user wants to buy a movie ticket). The set of intents recognizable by the bot interpreter must be included in this file.
- **Entities**. The set of entities the interpreter can recognize (e.g movie title, movie director, screening date,..).
- **Actions**. As explained during the introduction, actions are the output of a dialogue manager, and can be either simple actions (e.g Welcome utterance) or they can be more complicated ones (e.g trigger back-end calls). In RASA, "simple" actions are called Utter Actions and they only cause an utterance template to be triggered. More complex actions in RASA are defined through extendible Python classes called Custom Actions. Each action is identified by a name, which must be included in this Action list.
- **Slots**. We will better describe them later, but slots can be seen as key-value pairs which can be used to store some information gathered about outside world (e.g query results) as well as information provided by the user (e.g their home city).
- **Templates**. These are templates for the Utter Actions previously described.

```yaml
# all hashtags are comments :)
intents:
 - greet
 - default
 - goodbye
 - affirm
 - thank_you
 - change_bank_details
 - simple
 - hello
 - why
 - next_intent

entities:
 - name

slots:
  name:
    type: text

templates:
  utter_greet:
    - "hey there {name}!"  # {name} will be filled by slot (same name) or by custom action code
  utter_goodbye:
    - "goodbye 😔"
    - "bye bye 😔"   # multiple templates - bot will randomly pick one of them
  utter_default:
    - "default message"

actions:
  - utter_default
  - utter_greet
  - utter_goodbye
```

Figure 3.2: RASA domain format. From [14]

An example of domain is provided in Figure 3.2.

Since intents and entities are mostly related to RASA NLU, we will not describe how they are defined and managed. Instead we'll proceed giving insights on the remaining components of the domain.

3.2.2 Actions

As previously stated, in RASA we have two main families of actions: Utter Actions and Custom Actions.

Utter Actions are simple actions which objective is to provide a simple utterance, without causing any additional event. In order to define them, a developer just need to add an utterance template to the domain file, as it is possible to see in Figure 3.3.

```yaml
templates:
  utter_my_message:
    - "this is what I want my action to say!"
```

Figure 3.3: Example of utter action. From [14]

Custom Actions instead are more complex actions which can cause any programmatic event the developer wants (e.g back-end call, turn-on lights, add an event to a calendar,...). These actions are defined through a Python class called Action which must be extended by the specific Custom Action. The class is characterized by two methods:

- **Name**. This method just need to return a string indicating the action name indicated in the domain file.

- **Run**. This method is the core of the Custom Action and here the developer defines its behaviour. It has access to a dispatcher used to send messages back to the user, to the tracker containing the state of the conversation and to the bot domain.

An example of Custom Action is provided in Figure 3.4.

```python
from rasa_core_sdk import Action
from rasa_core_sdk.events import SlotSet

class ActionCheckRestaurants(Action):
   def name(self):
      # type: () -> Text
      return "action_check_restaurants"

   def run(self, dispatcher, tracker, domain):
      # type: (CollectingDispatcher, Tracker, Dict[Text, Any]) -> List[Dict[Text, Any]]

      cuisine = tracker.get_slot('cuisine')
      q = "select * from restaurants where cuisine='{0}' limit 1".format(cuisine)
      result = db.query(q)

      return [SlotSet("matches", result if result is not None else [])]
```

Figure 3.4: Example of custom. From [14]

3.2.3 Slots

In RASA Core, slots are defined as the bot's memory and they are key-value pairs which can be used to store and retrieve any type of information during the interaction. Slots have different predefined types, such as text, boolean, categorical, float, list and unfeaturized. Additionally, a Slot base class is provided, so that the user can define custom types of slots.

During the designing of the domain file, the developer can provide as many slot keys as he wants. Their relative value, during the interaction, can be managed by the NLU module and by the Custom Actions.

In general, if the NLU model picks up an entity and the domain contains a slot with the same name, the slot will be set automatically. Additionally, also Custom Actions can get and set slot values through the run method previously explained and ad-hoc functions (SlotSet and get_slot) provided by RASA.

How we will see in the next subsection, these slots will be featurized and used by the data-driven DM policies, playing an important role for the final behaviour of the system.

3.2.4 Policies

As it is possible to see in Figure 3.1 the policy decides which action to take at every step of the conversation, starting from the current state of the conversation encoded in the tracker.

As all the others components, a Policy is an extendible Python class which can be trained with data. By default, RASA offers the 3 following policies:

1. **Memoization Policy:** It memorizes the conversations in the training data, predicting the next action with confidence 1.0 if that exact conversation exists in the training data.
2. **Keras Policy:** It uses an LSTM implemented in Keras [12] (More on LSTM in the Appendix).
3. **Sklearn Policy:** It uses a Logistic regression implemented in sklearn to predict the next action.

Additionally, the developer can create its own rule-based or data-driven policies and he can also create ensembles.

Featurization

Any policy receives as input a featurized representation of the state encoded in the tracker. In fact at each turn, the tracker provides a bag of active features, which are:

- **Intent + entities**. The output of the NLU module in that turn.
- **Slots**. These features represent the slot values in that turn.
- **Previous Action**. These features represent the last action of the system.

These features are then converted into vectors and, in order to do that, RASA offers two featurizers:

- **BinaryFeaturizer**. Creates a binary one-hot encoding of the tracker state. (e.g [0 0 1 0 1])
- **LabelTokenizer**. Creates a vector based on the feature label. All active feature labels (e.g. prev_action_listen) are split into tokens and represented as a bag-of-words. For example, actions utter_explain_details_hotel and utter_explain_details_restaurant will have 3 features in common, and differ by a single feature indicating a domain

Additionally, in order to add context to the feature vectors, RASA offers a MaxHistory featurizer which, based on a *max_history* parameter, creates a feature vector for the state including the previous *max_history* turns. A Full Dialogue state featurizer is also provided, which keeps into account all the previous turns.

Training data format

RASA Core, in order to train a dialogue manager, needs training examples. These samples are called stories and must follow a specific format.

```
## story_07715946    <!-- name of the story - just for debugging -->
* greet
   - action_ask_howcanhelp
* inform{"location": "rome", "price": "cheap"}  <!-- user utterance, in format intent{entities} -->
   - action_on_it
   - action_ask_cuisine
* inform{"cuisine": "spanish"}
   - action_ask_numpeople        <!-- action that the bot should execute -->
* inform{"people": "six"}
   - action_ack_dosearch
```

Figure 3.5: Example of RASA story. From [14]

As it is possible to see in Figure 3.5 story always starts with a name preceded by two hashes and ends with a newline. Messages in a story are expressed at intentional level, meaning that messages sent by the user are shown as lines starting with "*" and following the format INTENT{"ENTITY1": "VALUE", "ENTITY2": "VALUE"}. Actions executed by the bot are shown as lines starting with "-"

3.3 High-level architecture

How it is possible to see in Figure 3.6, the input to the framework are a catalogue (e.g MySQL database) and a tasks specification file. The complete structure of this file will be discussed later but, for now, imagine it as a set of tasks, each one describing which are the information contained in the catalogue (e.g which are the column values the bot will ask for) needed to perform an operation on the database (e.g Insertion on a table).

Once the framework is fed with these inputs, a multi-agenda based user-simulator similar to [51] and a domain-agnostic rule-based Dialogue Manager, supervised by an interaction manager, interact and produce the following outputs:

- **RASA code**: How we have discussed in during the introduction 1.2, RASA is an open-source framework for building dialogue systems and, in contrast to many others, it offers a full programmatic way to do it. How we have previously seen, RASA in order to build the final system, needs a domain configuration file and python code for the Custom Actions. With Cookie-Cutter

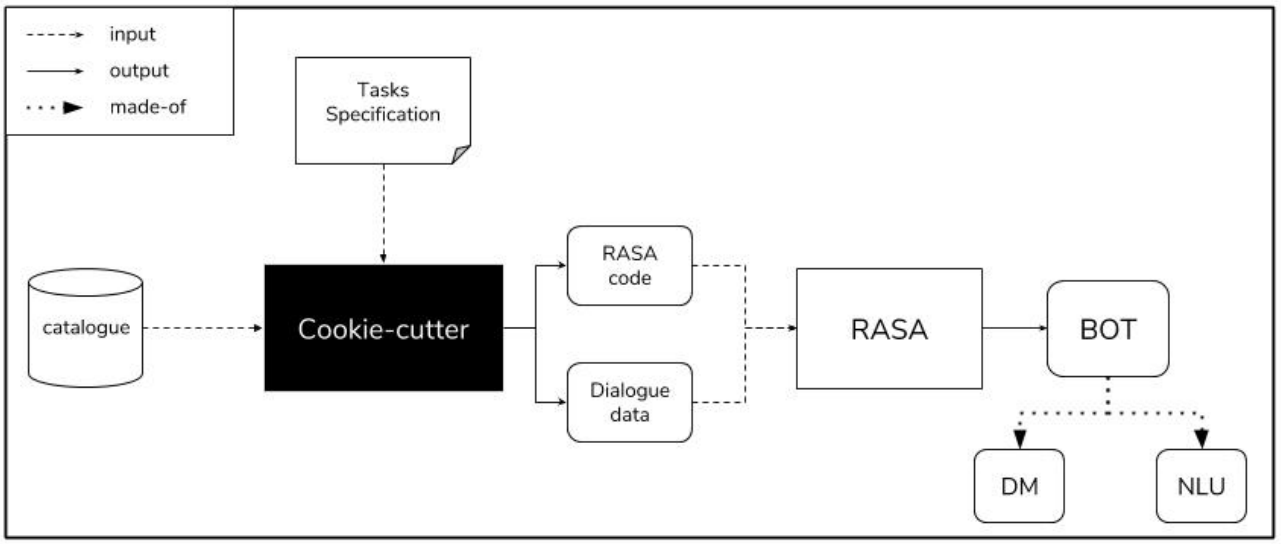


Figure 3.6: Cookie-Cutter high-level overview

we automatically produce and feed these files to RASA, so that the developer doesn't need to hand-craft them. Eventually he can decide to modify them in order to add or change the functionalities of the bot (e.g change the utterance templates, add/remove system actions).

- **Dialogue data**: Since RASA, by default, offers data-driven models for both the NLU and DM modules, the M2M interaction will produce data for bootstrap both the components, in the format required by RASA.

Both the code and the data for the models are then feed to RASA, the RASA training procedures for both the NLU and DM modules are called and a task-based dialogue system, tailored to the ingested catalogue and the defined tasks, is the outcome of the overall process.

Ar first glance it would seem that the NLG module is missing but, how we have discussed previously, RASA embeds template utterances directly in the system action code, which we are automatically producing. What this bot misses is an ASR and a TTS module but there exists several easy-to-install services offering such functionalities [11] [4] [5] [3].

Disclaimer

Concerning Cookie-Cutter, the scope of this thesis work is limited to the bootstrapping of the DM module, which has been designed and developed as first instance of the framework. The second instance of the framework, which keeps into account the bootstrapping of the NLU module, has been designed and developed by Federico Giannoni and it is explained in its thesis work.

3.4 Cookie-Cutter components

In the following section I will give a brief description of all the framework components, then a deeper analysis of the modules developed by the author of this thesis will be presented.

3.4.1 Core overview

The core of the framework is composed by the components showed in Figure 3.7.

Every component has been modelled as Python classes and a main script manage how the flow of information passes through them. These modules are:

- **DB Metadata Extractor.** This module takes as input a populated catalogue (e.g MySQL database) and outputs a metadata structure containing all the information related to entities and relations contained inside the catalogue (e.g table names, columns, default values, data types, references,...).

core-architecture.jpg

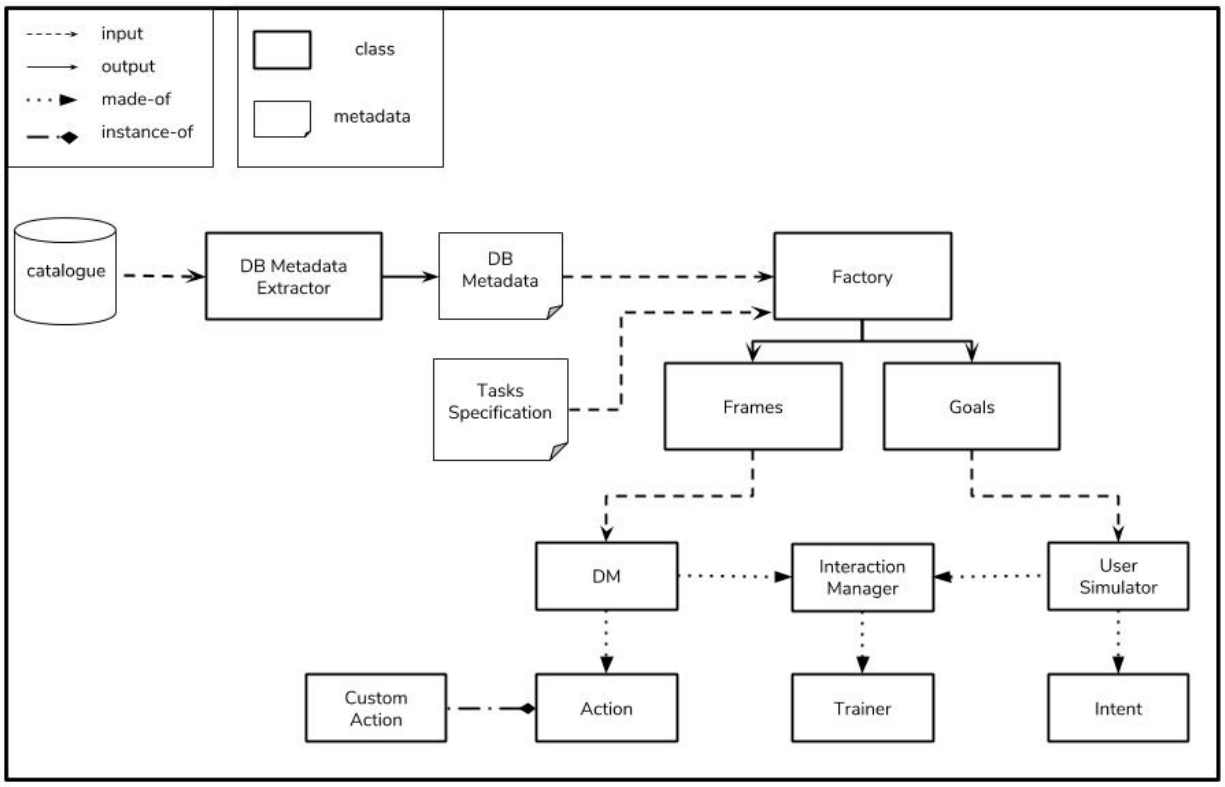


Figure 3.7: Cookie-Cutter core architecture overview

- **Factory.** This component, starting from the DB metadata and the Task specification file, outputs both a set of semantic frames for the DM and a set of goals for the agenda-based user simulator.

- **Frames.** We will describe later the full concept of Frame in cookie-cutter but, for now, imagine it as a set of slot values to ask to the user in order to complete a specific task defined by the frame.

- **Goals.** A goal is defined as a set of slots with specific values the user wants. Additionally we also enrich the concept of goal with requestable slots, which are slot for which the user can ask value to the system.

- **DM.** This component is a rule-based domain-agnostic Dialogue Manager. In order to work it requires frames as input and based on the information enclosed in the Frames, it leverages an Interaction Manager (IM) instance to communicate with the User Simulator. Through the IM it is able to send Actions or Custom Actions and to receive Intents.

- **Actions & Custom Actions.** The former are what RASA identify as Utter Actions, the latter are the Custom Actions already explained in the RASA section. Cookie-cutter already provides all the Actions needed for every category of task it offers, which are going to be explained later.

- **User Simulator.** This is a multi-agenda based User Simulator similar to [51]. It is characterized by a profile vector indicating attributes such as cooperation, flexibility, experience, indecision and ambiguity. Once it has been fed with the Goals, it uses the IM to send Intents and read Actions or Custom Actions sent by the DM.

- **Intent.** This class models the intents used by the user simulator. As for the Actions, Cookie-Cutter offers by default all the intents needed for every category of task.

- **Interaction Manager.** This module deals with the whole communication between the user simulator and the dialogue manager, offering directives such as *send_intent*, *read_intent*, *send_action*, *read_action*.

- **Trainer.** This component is the one producing all the files needed by the RASA framework in order to build the final bot. It outputs:

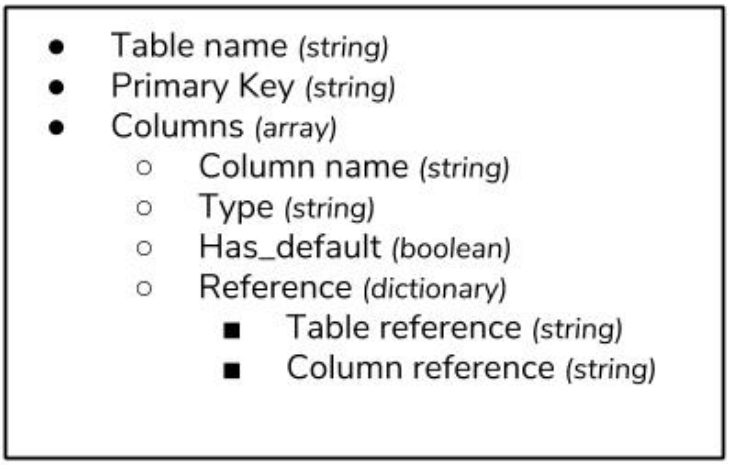
  - The domain file.
  - A Python file containing all the code for the RASA Custom Actions.
  - Two story files for the training of the data-driven DM. One is the validation set, the other is the training set.

Additionally to the modules just briefly described, a Graphical User Interface (GUI) helps the developer in uploading a catalogue dump, defining the task specification file and start the whole process to build the dialogue system.

3.4.2 DB Metadata Extractor

This component objective is to extract a metadata structure containing all the information about entities, their attributes and relations in the ingested catalogue.

We decided to model the metadata structure as an array of tables contained in the database, each table described by a the structure showed in Figure 3.8.



metadata struct.jpg

Figure 3.8: Metadata structure extracted by the module.

For each table we store:

- **Table Name.** A string containing the table name

- **Primary Key.** A string containing the column name of the primary key. As simplifying assumption the first instance of the framework only works on primary keys defined on a single column value since it simplify both the user simulator and the DM handling logic that we will see later.

- **Columns.** An array where each element describes a column of the table, with the following structure:

  - **Column Name.** A string defining the column name
  - **Type.** A string defining the type
  - **Has Default.** A boolean value defining if the column has a default value or not.

- **Reference.** A dictionary expressing which are the table and the column referenced by this column. This dictionary follows the structure:
  - **Table reference.** Table referenced by the column.
  - **Column reference.** Which column of the referenced table this column references.

This structure, together with the Task specification which we are going to discuss later, we'll be used to build all the set of information needed by both DM and User Simulator in order to generate data.

Additionally to the DB metadata, this module also offers directives to perform all the queries needed (insert, update, delete, select, projection).

Finally, a procedure for slot values normalization is also provided. This is needed since, when we need to perform some query with attribute values for some entity, it could happen that the attribute values recognized are not exactly the same contained in the database. So, this method checks each word in the attribute value passed as input against all the values in the DB for such attribute, returning the closest result to the source.

3.4.3 Frames, Slots and Tasks

The underlying assumption on which Cookie-Cutter is based is the following:

All task-based dialogues can be modeled as a sequence of frame-filling tasks that may culminate with one or more operations involving a knowledge-base

Meaning that, in task-based interactions, we want to gather all the information needed for the task, here represented as sequences of frames, which are then used to perform some type of operation in the knowledge-base. So, before to understand the concept of task we need to understand its base component, the frame.

A frame is a predicate-argument structure where the predicate is a database operation (select, insert, delete, update) and the argument is a database table, which can be tracted as an entity with attributes and relations to other entities. It is composed by different sets of slots, defined as the subsets of the entity attributes for which the system needs to gather information about, before to proceed with the final DB operation. Each of these slots have a priority value, defining in which order the system will asks for their values. Some of these slots can be defined as requestable, indicating to Cookie-Cutter that the user can ask values to the system for such information.

Additionally, we also give the possibility to define dependencies between slots. Formally, we say that a slot S1 depends on a slot S2 when a new user preference for S2, may imply a new preference for S1. Finally, a frame can have children with the same structure defined so far and has a priority value defining its order inside the task, which is going to be more clear at the end of this subsection.

In Cookie-Cutter we offer 4 instances of frames, one for each main database operation. We'll see later how the DM handling strategy changes according to the type of Frame but, in general, the only difference is the operation triggered on the knowledge-base once the frame has been filled. The general behaviour triggered by the 4 types of frames is here described:

- **Select Frame:** The DM collects all the values for the slots defined by the frame, checking at each time that a valid configuration exists in the database. Once the frame is filled, the DM proposes the configurations to the user which selects a suitable item from the list.

- **Delete Frame:** Collects and checks as above. Once the frame is filled the DM deletes the selected entry.

- **Insert Frame:** Collects without checking for the validity of the configuration, since we are inserting a new item. Then the DM insert the collected entry.

- **Update Frame:** As for select and delete, this frame triggers a collect and check procedure of the information. Once the item to be updated has been selected, the DM starts gathering the new values to update since the user explicitly communicates that he is done with updates.

Additionally, the framework also offers reference resolution in the case some frame depends on some other. This is exactly the case of foreign keys (FK) in the database, where some attribute of an entity is a reference to another entity and, in order to have its value, the DM should need either the direct FK value or a procedure to select it. Specifically, Cookie-Cutter automatically allocate a child Select Frame for each foreign-keys entity in these two scenarios:

- The developer wants to allocate an insert frame. Since, in order to insert an entry into a database, we need to have all the foreign-keys values, the DM will need these information to be gathered.
- The developer selects a slot for a frame which is a foreign key.

Once the foreign-key value has been gathered through the allocated Select Frame, its value will be passed to the parent Frame, which now has all the information needed to proceed. Multi-reference cases are also took into account and, for this reason, we tracted a **task** as a tree of frames having as root the final frame to be filled and a cascade of Select children frames defining all the slot values needed to complete the information-gathering process. In addition to the hierarchy imposed by this structure, we constraint the priority value of each children frame to be lower with respect to the priority value of the parent. This is needed since, how we'll see later, the DM needs all the frame slot values to be filled before to proceed with the final operation, so the children frames need to be filled before their parents.

To clarify, let's take a look at the graphical example (Figure 3.9), showing the definition of a task through a GUI. In this example the developer wants to add a task for **order registration**, having as input its database of shoes and orders for shoes.

The steps he follows are the following:

1. Add a new task for **insert** order. Since the only way to register a new order is an insert on the order table.
2. **All columns** are added to the frame, since it is an insert frame and, as stated before, all the information are needed.
3. Cookie-Cutter identifies that product_id refers to a separate entity (size_chart) and a new **select** frame is automatically allocated for that entity.
4. The developer now specifies all the information needed for the new frame.
5. Between all the column values he also chooses shoe_id as slot and, as before, Cookie-Cutter allocates a new select frame which the developer needs to fill.

This procedure will generate a tree of frames (Figure 3.10), each one with a filling priority (lower number = higher priority). This tree will be used by the Factory class showed in Figure 3.7 in order to build Frames and Goals used by the DM and the User Simulator to interact and generate data.

3.4.4 Dialogue Manager, Actions and Custom Actions

This component is a domain-agnostic rule-based Dialogue Manager and it interact with the User Simulator at an intentional level, receiving intents and sending actions.

As previously stated, the DM leverages the tasks of frames generated by the developer in order to handle the whole rule-based logic. From an high-level perspective its overall objective is to fill all the frames composing the task, running the frame completion routine (insert, update, delete, select) once the frame is filled, from the lower-priority frame to the higher-priority one (the root of the tree). Once the root frame has been filled and its completion routine has been run, the task is considered completed.

The core of this module is a **run** function which uses the Interaction Manager to read intents and entities sent by the user simulator. Based on both these elements and the state of the dialogue

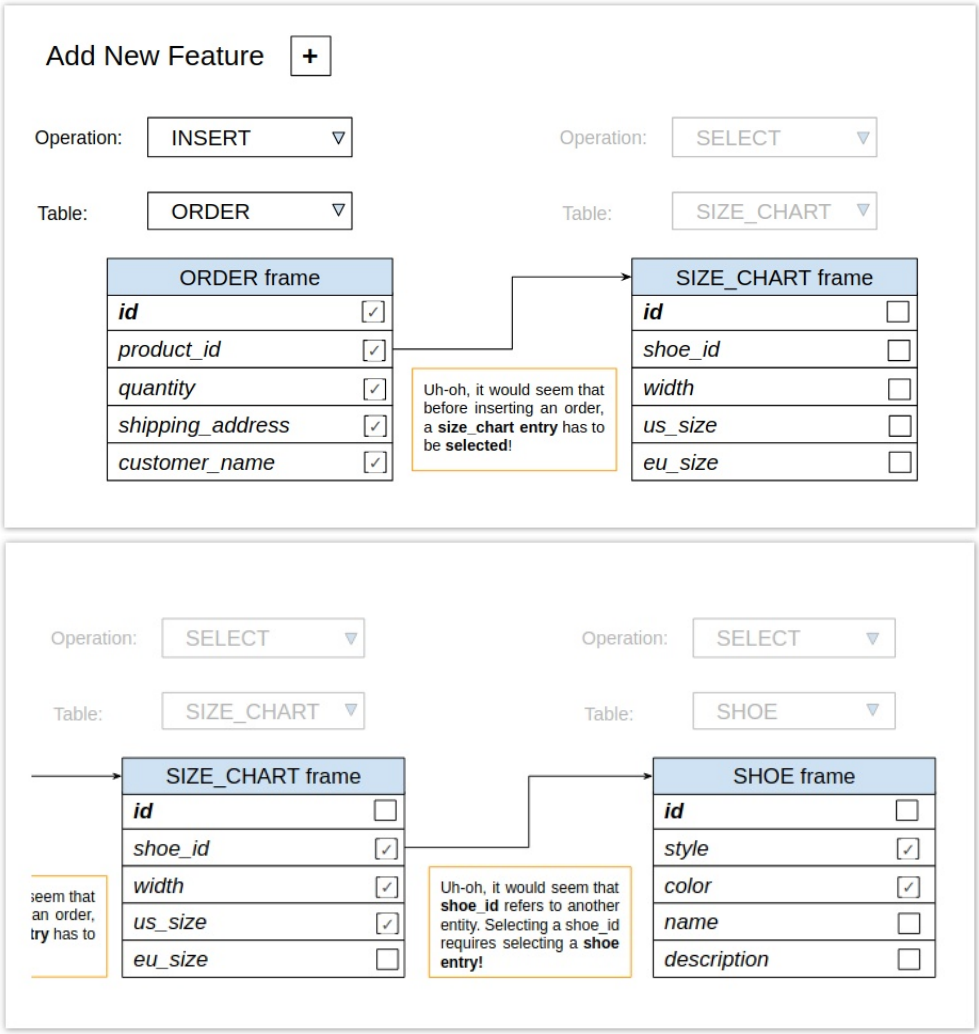


Figure 3.9: Example of task definition in Cookie-Cutter

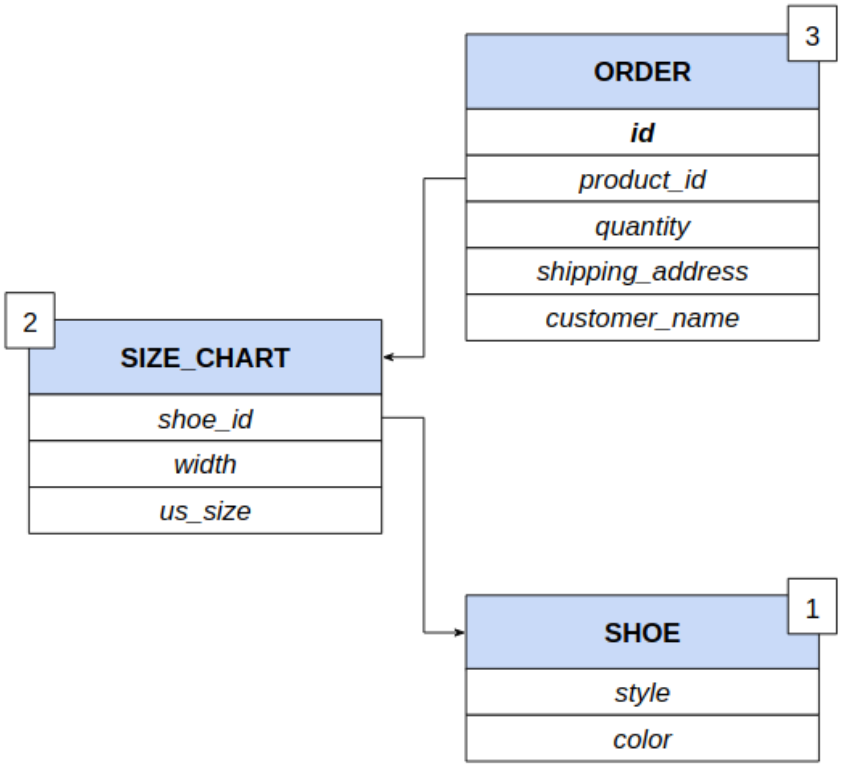


Figure 3.10: Task tree produced by the task definition procedure showed in Figure 3.9

(current slot values in each frame), the method chooses an action and sends it through the IM. In order to understand its behaviour and how it uses Actions and Custom Actions to reply to the user, in the following we are going to describe the handling logic of each rule we wrote.

Begin Task Rule: This rule is triggered if the history of actions and intents is empty. As result, this rule just initialize the internal knowledge-base used to perform the queries.

Repeat Rule: This rule is triggered by a Repeat intent which indicates an user asking to the system the repetition of the last prompt. The DM handles this logic picking up the last action triggered from its history and sending it.

Give Up Rule: If the user simulator sends a GiveUp intent or a ThatsIt intent, this rule triggers a Bye action, which is just a simple action uttering the goodbye.

Inform Rule: Once the User Simulator uses the Inform intent to communicate some slot value preference, the DM triggers this rule. Since the User Simulator, as previously stated, can vary its behaviour and change preferences during the interaction, the DM firstly checks if the slot values have been updated with respect to already communicated values. If the values have been changed and the related frame has been already completed, a RestorePreference custom action is called and it restores the frame slot values according to the new communicated configuration, keeping only the valid slot values and discarding the old ones. Instead, if the user didn't update its preference, the DM just updates its belief without restoring any preference. Then it runs a Lookup custom action to check if the configuration exists in the DB and replies accordingly to the result-set.

Request Rule: The user simulator can use a Request intent in order to request the available values for some slot. In this case the rule triggers a MultiChoice custom action which communicates only the slot values, for the requested slot, bringing to a valid DB configuration with respect to the current slot values in the whole set of frames.

DontCare Rule: This rule is triggered when a DontCare intent is received by the DM. This intent can be communicated from the user simulator when the DM asks for some slot value and the user does not have preferences for the slot. At this point, the DM triggers a RelaxLastRequest custom action which simply store a DONTCARE as slot value for the communicated slot. This special value will be then used by the database queries to indicate ANY value as constraint for the slot.

Frame Filling Rule: This rule is triggered at any time the DM has not yet completed all the frames. The rule look for the lower-priority slot not yet filled, starting from the lower-priority frame not yet completed. Once it finds the slot value to ask for, he sends a Request action to the user.

Frame Completion Rule: At each turn, after having correctly updated the dialogue state, the DM checks frame by frame if all the slot values have been filled, starting from the lower-priority ones. Every time a filled frame is found, depending on the type of frame, a completion routine is run. As previously stated, Cookie-Cutter offers 4 types of frames, one for each DB operation and their handling logic only differs inside this rule:

- **Insert Frame.** The completion routine of this type of frame is the easiest one. It just run an insert query on the table referenced by the frame, using as values the whole set of slot values communicated so far.

- **Delete Frame.** When completing this type of frames the DM firstly needs to show to the user the set of items satisfying all the slot values gathered so far and wait for a selection of an item, this is an **elicit user preference routine** which is going to be discussed next.

  Once the item has been selected by the user, the DM just send a Delete action which communicates to the user the deletion of the item and, on the back-end, triggers the deletion of the selected item from the knolwedge-base.

- **Select Frame.** In order to propose the available items and let the user choose one of them, the same routine just presented is run. After the selection of the item a SaveChoice custom action is triggered, communicating to the user the selection of the item and, immediately after, the DM send also a TransferPreference custom action which manages the transfer of the foreign key values to the parent frame.

- **Update Frame.** In line with the previous two frame completion routine, also this procedure makes use of the process to elicit the user preference. Once an item has been selected by the user, the DM sends a PollForUpdates action, which is just a verbal action asking to the user the values he wants to change. At this point the procedure wait for Inform intents carrying the new slot values for the selected item and at each new Inform received, it updates the frame and asks for new updates using the custom action MoreUpdates. Once the user simulator reply with a Deny intent, the routine triggers an Update custom action, producing a back-end call to update the item with all the slot values gathered.

Elicit User Preference routine: This routine starts sending a Propose custom action which, given all the slot values collected so far, retrieves and shows to the user simulator the firsts 3 results from the knowledge-base. Then it loops until the user either select some of the items or give up the communication. We offer 2 types of selection for the item, which are:

- **By index**. The user, using a specific slot "index" passed with an Inform intent, can select the item just indicating its position inside the proposed list (e.g the first, the second, ...).

- **By slot value.** The user, using an Inform intent, can select an item using some additional information not already passed as slot value(e.g "I want the shoe manufactured in Italy"). At this point the DM filtrates the result-set according to the new value and this elicit routine is run again till the user selects a single item.

In addition, this procedure also handles the navigation of the matches from the knowledge-base. Since we only show the firsts 3 results, we offer three custom actions (LoadMoreOptions, LoadPrevOptions and LoadHeadOptions) triggered by 3 intents (AskMoreOptions, AskPrevOptions and AskHeadOptions) in order to let the user being able to navigate all the items.

Task Completion Rule: Once all the frames have been filled and all the completion routines have been finished, this rule just erase all the frame contents, so that a new interaction can start.

All the actions listed so far have been modelled in order to embed also all the information needed by RASA. In particular, our Actions instances corresponds to RASA Utter Actions and, since both a name and template utterances are needed by the domain file, all the Actions instances (e.g Bye, Greet, Acknowledge, PollUpdates, MoreUpdates and Request) have two already defined attributes for such properties. Concerning the Custom Actions (Insert, Update, Delete, Select, MultiChoice, SaveChoice, Lookup, CompleteTask, BeginTask, ...) , in RASA we also need the python code so, in addition to the Actions, they are extended with general python code templates, which will be populated at instance time with all the specific information contained in the frames. These templates also embed general utterances which can be then modified by the developer in the python code file generated at the end of the process.

3.4.5 Trainer

This module handles the generation of all the files needed by RASA in order to build the data-driven dialogue agent. As previously explained, RASA needs:

- A domain file describing intents, entities, slots, actions and templates.
- A Python file containing all the Custom Action code.
- Story files for training and testing the data-driven DM.

Cookie-Cutter allows three different settings for training the data-driven DM. The first setting will set-up the framework, in particular this Trainer, to generate data for an **entity only** data-driven DM, meaning that we completely discard the concept of intents and we produce data to train a DM using only entities and slot to select the next action, without featurizing the intents. With the second approach, Cookie-Cutter will produce data for an **intent + entity** data-driven DM, keeping also into account the intents used by the user during the training of the system. The last setup is used to produce a **Sentence Embedding + entity** data-driven DM, giving to the intent a vectorial representation given by a sentence embedding algorithm (more on sentence embedding and algorithm used in the Appendix).

The Trainer instantiated for the **entity only** setting offers an initialization procedure in which all the files are initialized according to the RASA format and the domain file is populated with the following information:

- Only the **communicate** intent is added to the intent list.
- All the slots of all the frames, defined by the user, are added as entities and slots.
- We add 4 slots for keeping the information about the user navigation preferences on the items we are going to present to him (matches, index, result offsets, results displayed)
- All the utter actions names are logged in the action list.
- General templates for the utter actions are retrieved and logged in the domain file.

In addition to this initialization procedure, it also offers 2 main methods which are:

1. **log_intent** This routine is called any time an intent passes through the interaction manager and it logs the intent, for the current dialogue, in the story file, following the RASA format. Since we are in an **entity only** setting, even if the user simulator makes use of intents for the M2M interaction, we only uses the **communicate** intent in both the domain and the RASA stories.

2. **log_actions** This method is run by the IM everytime an action is sent from the DM. Likewise to log_intent, this method logs the actions in the story file, according to the RASA format. Moreover, since during the initialization of the trainer we do not have information about custom action which will be triggered during the interaction, this procedure also logs all the custom action Python code, leveraging the general templates we have previously explained.

The **intent + entity** Trainer instance is very similar to the one just explained. Its implementation just differs because:

- We add **all** the intents used by the user simulator to the domain file.
- The **log_intent** procedure logs the real intents in the story file, and not the **communicate** one.

The last Trainer has been developed after a first implementation of the second instance of the framework has been developed. In this second version, the framework was enhanced by Federico Giannoni to support also templates for intents, so that the simulator can also communicate at utterance level, producing sentences. The **Sentence Embedding + entity** instance instead makes use only of the **communicate** general intent but, in addition, it uses the Universal Sentence Encoder [2] algorithm to dump the sentence embeddings of the intents which will be used by RASA in order to train the data-driven DM.

In addition to these procedures all the trainer instances have routines to feed the data produced to the RASA framework, calling the appropriate training procedures offered by the framework.

3.5 Bot generation process

Cookie-Cutter, in order to build the final data-driven dialogue agent, guide the developer through these steps:

1. **Preparation**
   (a) The developer, through a GUI similar to the one showed in 3.9, uploads a dump of the database.
   (b) The developer defines the Task structure, populating all the involved frames with all the information described in the subsection 3.4.3.
   (c) The developer chooses the training setting of the data driven DM between the three options: **entities only**, **intents + entities**, **Sentence Embedding + entities**.
2. **Data Generation**
   (a) A main script run, starting to generate Goals and Tasks of Frames. Goals are fed to the user simulator and Frames to the DM.
   (b) A Trainer is instantiated, according to the settings indicated by the user, and is fed to a singleton instance of the Interaction Manager.
   (c) The just generated IM is shared between DM and User Simulator.
   (d) The M2M interaction begins in order to generate N training dialogues under the form of story files. A 30% of these dialogues will be kept as validation set, the remaining 70% as training set.
   (e) The Trainer instance, during the interaction, populates all the files needed by RASA as previously explained.
3. **Data Feeding**
   (a) Once the simulation finishes, a data folder is filled with the domain file, the custom actions python code and the story files. Additionally, also sentence embeddings can be provided.

[2] Python library: https://tfhub.dev/google/universal-sentence-encoder/2

(b) The Trainer calls specific routines to feed those data to RASA, and a data-driven DM is the output of the overall process.

In order to have diversity in the generated data, during the generation process Cookie-Cutter automatically manage the User Simulator profile, starting from a full collaborative user till a non-cooperative user [3]. The number N of generated dialogues has been fixed to 1000 dialogues since, after some test, we realized that it is enough to reach good performances and cover all the dialogues paths. Regarding the parameters of the data-driven models, we use the default ones already provided by the RASA framework. Concerning the RASA featurization methodology, for each data-generation setting offered by Cookie-Cutter, we chose the most performing one among the ones presented in the experiments chapter.

4 CookieCutter: Evaluation

In the following we'll show how we evaluated the performances of the first instance of Cookie-Cutter so, only the bootrstrapped DM will be evaluated under different settings and scenarios [1].

4.1 Domains and Tasks

In order to evaluate Cookie-Cutter in different scenarios we decided to work on two domains, the movie domain and the restaurant domain.

Since Cookie-Cutter need a catalogue to start from we decided to build our own DB schemas, wich are shown in Figure 4.1. Then we wrote a script build and populate the two MySQL databases, leveraging a python library [2] to populate the movie database and a json file [3] for the restaurant one.

Once both the database have been populated we also generated the task structures to be fed to the framework with the following objective for each domain:

1. **Movie domain:** Guide the user to the purchase of a ticket for a movie screening (**ticket purchase task**). Figure 4.2 .

2. **Restaurant domain:** Guide the user through the reservation of a table restaurant (**restaurant reservation task**). Figure 4.3.

As it is possible to see from the Figure 4.2, we decided to model the ticket purchase task as an INSERT frame on the ticket_purchase table. Since it is an INSERT frame, Cookie-Cutter forces all the attributes to be part of the frame and, since screening_id is a foreig key a SELECT frame for the screening entity is generated. This last SELECT frame, again, carries the selection of the foreign keys movie_id and theater_id, directly co-related to the SELECT frames for movie and theater. As it is possible to see we defined different set of slots for each frame and their priority values have been force to respect the hierarchy.

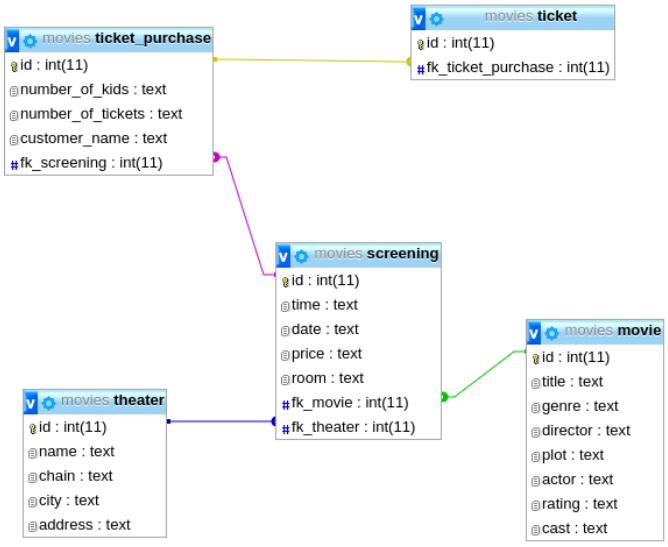
Concerning the restaurant reservation task (Figure 4.3) we modeled it as an INSERT frame defined on the reservation table. Like for the movie task, Cookie-Cutter automatically allocates frames for FK attributes which in this case are the entities restaurant and location. Again, different sets of slots have been defined for each frame and the relative priority values have been forced to respect the hierarchy.

[3] A detailed explanation of the whole User Simulator behaviour can be found in the Master Thesis of Federico Giannoni [federico.giannoni@studenti.unitn.it]

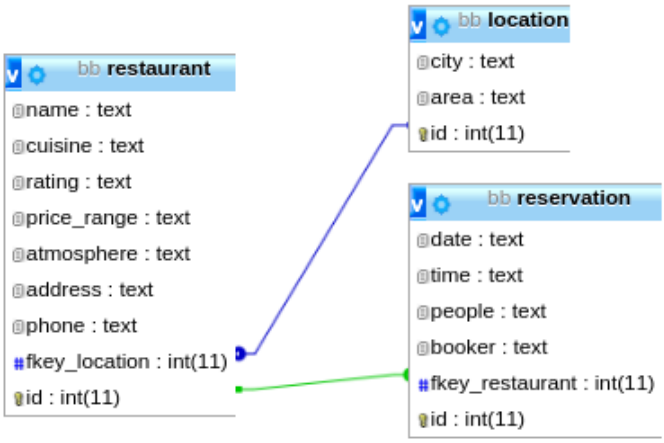
[1] An evaluation of the bootstrapped NLU module is given in the master thesis of Federico Giannoni

[2] https://imdbpy.sourceforge.io/

[3] https://github.com/MattZhao/cs61a-projects/blob/master/maps/data/restaurants.json



(a) DB schema movie domain



(b) DB schema restaurant domain

Figure 4.1

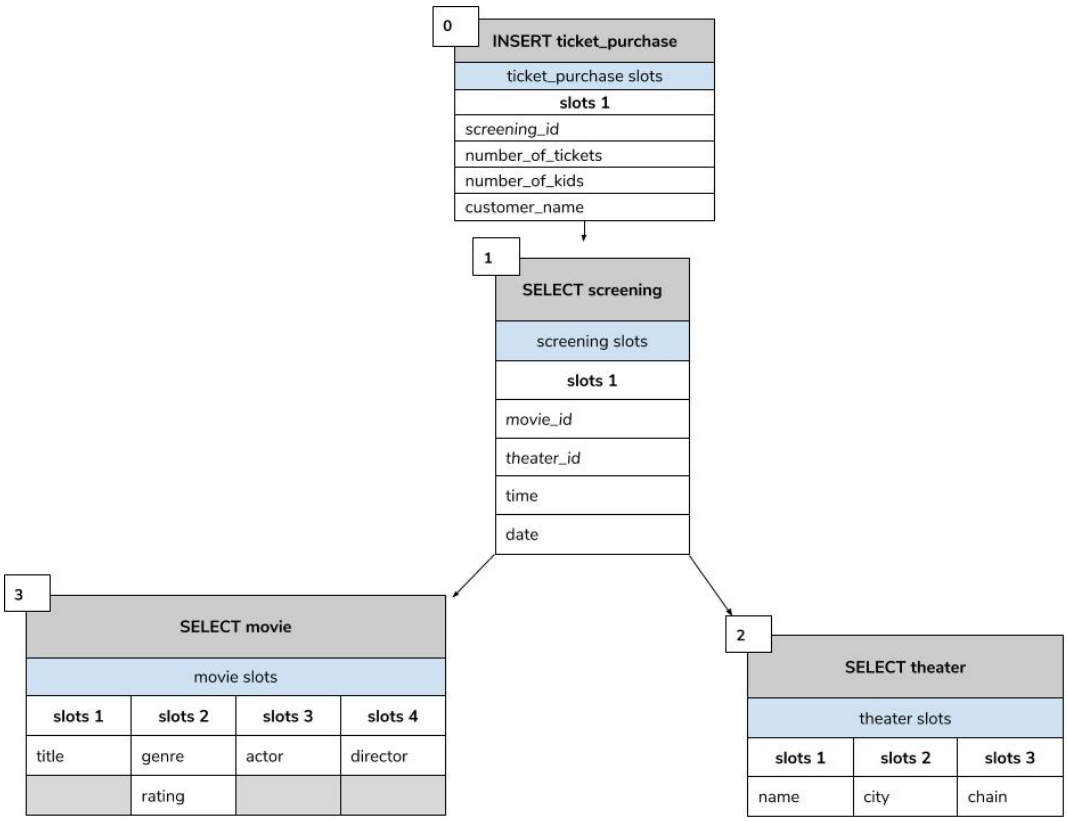


Figure 4.2: Task structure for ticket purchase task

4.2 Approach and Metrics

In order to evaluate the performances of the data-driven DM generated by Cookie-Cutter we need to define proper evaluation metrics.

The best approach would be to have an expert communicating at intentional level with the DM and covering all the dialogue paths that the DM should be able to handle. The expert, dialogue by dialogue, should annotate at each turn the set of correct actions and the ones predicted by the DM. Additionally, he should also annotate how many dialogues bring to a correct completion of the task. To perform such experiment we should need at least one expert and he should know very well how the DM strategy of the specific bot works. Moreover the number of possible dialogue paths, dependending on the task, can be very high, causing an high cost for the overall experiment.

Our Approach. Instead we opted for the standard approach, using a test set. The test-sets, for both the domains, have been generated through an additional data-generation round with Cookie-Cutter under the same conditions used for the generation of the DM in the two scenarios.

Evaluation Metrics. We choose two evaluation metrics:

1. **Action Error Rate (AER):** Defined as the percentage of correct actions A_i predicted by the DM, with respect to the total number of actions predicted A_j.

$$AER = A_i/A_j$$

$$A_i \subset A_j$$

2. **Error-free dialogue rate (EDR):** This metric is calculated as the number of dialogues D_i bringing to a completion of the task, over the total number of test dialogues D_j.

$$EDR : D_i/D_j$$

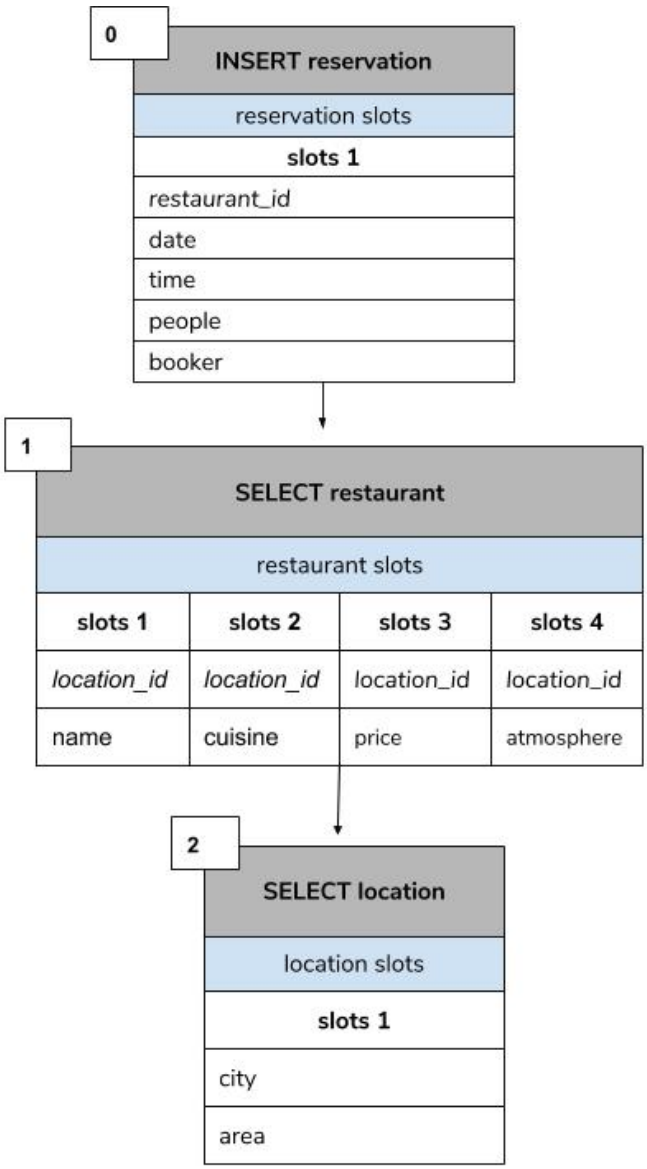


Figure 4.3: Task structure for restaurant reservation task

$$D_i \subset D_j$$

4.3 Training settings

For each of the three Cookie-Cutter setting described in the previous chapter, we've run tests varying:

- **RASA Featurization.** We tested both binary and label featurizer offered by the framework. Moreover we varied the *max_history* training parameter in order to embed from 0 to 5 previous turns as context for the feature vector passed to the models.

- **Policies.** We tried all the three RASA policies previously discussed. The parameters of the data-driven models (LSTM and Logistic Regression) are the default offered by Keras and SKLearn.

As baseline model we chose the DM produced by Cookie-Cutter under the **only-entity** setting.

For both the domains and for all the experiments we've just described we generated a total of 1000 dialogues to be used for the training, split as 70% training-set and 30% validation set. A separate set of 255 dialogues has been generated for the testing.

Policy	Features	Max History	AER	EDR
Keras	binary	5	10.22%	41.56%
Keras	binary	2	12.31%	39.21%
Keras	binary	1	18.50%	36.86%
Sklearn	binary	5	22.62%	34.90%
Sklearn	binary	3	26.70%	31.37%

Table 4.1: Restaurant domain

Policy	Features	Max History	AER	EDR
Keras	binary	5	16.31%	24.70%
Keras	binary	3	20.35%	23.13%
Keras	binary	1	22.27%	21.17%
Sklearn	binary	3	25.12%	19.60%
Sklearn	binary	1	27.20%	17.25%

Table 4.2: Movie Domain

Table 4.3: Performances of DM generated by Cookie-Cutter under the **only-entities** setting.

Policy	Features	Max History	AER	EDR
Keras	binary	5	2.92%	63.9%
Keras	binary	3	4.01%	61.56%
Keras	binary	1	5.80%	59.21%
Sklearn	binary	4	8.30%	55.29%
Sklearn	binary	3	10.21%	53.33%

Table 4.4: Restaurant domain

Policy	Features	Max History	AER	EDR
Keras	binary	5	4.82%	37.2%
Keras	binary	4	6.12%	36.07%
Keras	binary	2	7.35%	34.50%
Sklearn	binary	4	9.80%	31.37%
Sklearn	binary	2	11.14%	28.62%

Table 4.5: Movie Domain

Table 4.6: Performances of DM generated by Cookie-Cutter under the **intent + entity** setting.

4.4 Results

We report the top 5 results for each of the three Cookie-Cutter setting previously described, respectively in table **??**, 4.6 and 4.9. Additionally, a table with the overall top 3 results is shown (table 4.12.

Only-Entity setting. In this baseline setting, the best performances have been reached by the Keras Policy using binary features with a context of 5 previous turns, for both the domains. It is interesting to see how the LSTM used by the policy always beats the Logit implemented in the Sklearn policy. Moreover it is possible to see how the contextualization (max_history parameter) helps during the predictions.

Intent + Entity setting. In this setting the best policy for both the domains has been again the Keras one, with binary features and a max_history parameter set to 5. As in the previous case the LSTM beats the Logit and the contextualization helps.

Policy	Features	Max History	AER	EDR
Keras	binary	4	23.50%	19.60%
Keras	binary	3	26.72%	17.25%
Keras	binary	1	30.42%	15.68%
Sklearn	binary	3	34.60%	13.33%
Sklearn	binary	1	36.41%	11.37%

Table 4.7: Restaurant domain

Policy	Features	Max History	AER	EDR
Keras	binary	5	34.62%	15.68%
Keras	binary	3	37.10%	14.11%
Keras	binary	1	38.98%	11.76%
Sklearn	binary	4	40.21%	10.19%
Sklearn	binary	2	42.10%	8.23%

Table 4.8: Movie Domain

Table 4.9: Performances of DM generated by Cookie-Cutter under the **SentenceEmbedding + entity** setting.

SentenceEmbedding + Entities. Under this setting the best DM generated has been the one using a Keras policy with binary features. Likewise the two previous experiments, the LSTM and contextualization helps a lot for the performances.

Final Overview. How it is possible to see in table 4.12, the baseline DM (only-entity) has been outperformed by the **intent + entity** DM, in both domains but the sentence embedding DM performs very bad with respect to the others. This because a one-hot encoding of the intent used by the user helps a lot for the discrimination problem of action selection. Instead, a vectorial representation of the intent, given by the embedding algorithm used, seems to be not that helpful for the discrimination problem, probably because the high-dimension of the feature vector and the usage of template-based utterances which brings to a low vocabulary size.

Something interesting is that the binary features always give better performances with respect to the features given by the label featurizer offered by RASA, probably because the compactness of the vector representations provided by the binary featurizer. Finally, it is possible to see how the movie domain is harder with respect to the restaurant one. This can be easily motivated looking at the task structures presented in Figures 4.2 and 4.3. The movie ticket purchase task has an additional frame with respect to the restaurant task, carrying much more slots. So, in order to reach the root frame, a dialogue in the movie domain could require much more turns in order to gather all the information needed, so the complexity goes up.

5 Automatic Error Discovery: The tool

In this chapter we'll describe how we designed and implemented the tool for automatic error discovery during live interactions with a dialogue system produced by Cookie-Cutter. We'll start re-capping the objectives and showing our error ontology. Then we'll show how we designed two experiments in order to gather data about errors happening in real interactions. Finally we'll explain the two different settings of experiments we've decided to run and the data-driven models used.

Setting	Policy	Features	Max History	AER	EDR
Intent + Entity	Keras	binary	5	2.92%	63.9%
Only entity	Keras	binary	5	10.22%	41.56%
SentenceEmbedding + Entity	Keras	binary	4	23.50%	19.60%

Table 4.10: Restaurant domain

Setting	Policy	Features	Max History	AER	EDR
Intent + Entity	Keras	binary	5	4.82%	37.2%
Only entity	Keras	binary	5	16.31%	24.70%
SentenceEmbedding + Entity	Keras	binary	4	34.62%	15.68%

Table 4.11: Movie Domain

Table 4.12: TOP 3 overall performances for each setting

5.1 Objective

The objective of this tool is to perform automatic error-discovery during real interactions with the agent bootstrapped by Cookie-Cutter. As stated during the introduction, the main challenges for this task are: tract uncertainty of natural language and deal with the error-propagation.

5.2 Error Ontology

Since, at each turn, we want to understand if the agent made a mistake and the dialogue systems produced with Cookie-Cutters are composed by two main components, which are the NLU and the DM module, we modelled the error ontology with the following categories:

1. **NONE:** The agent didn't make a mistake in the turn.
2. **NLU:** The agent made a mistake in the turn and it was caused by the NLU module (e.g misunderstanding, non-understanding).
3. **DM:** The agent made a mistake in the turn and it was caused by the DM module (e.g wrong action prediction)
4. **BOTH:** The agent made a mistake in the turn and it was caused by both NLU and DM modules.

5.3 Data Gathering

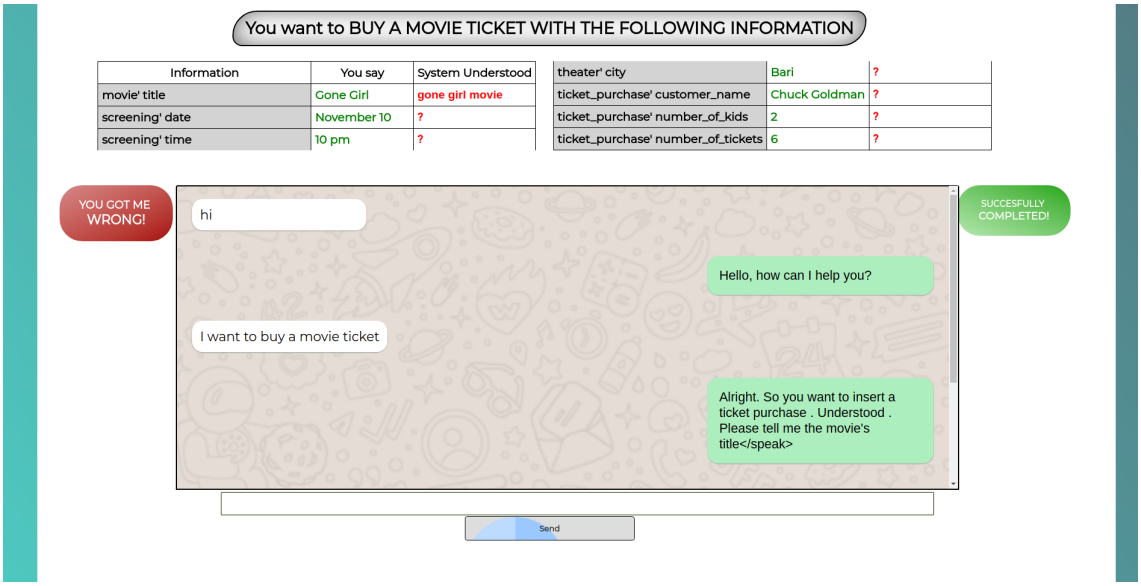
Since we'll use data-driven models to perform the task, we've designed experiments in order to gather labelled data with real testers. We decided to design two slightly different experimental scenarios, in order to understand which is the best setting to leverage for data-gathering, a baseline scenario and a contextualized one which are going to be discussed.

5.3.1 Baseline scenario

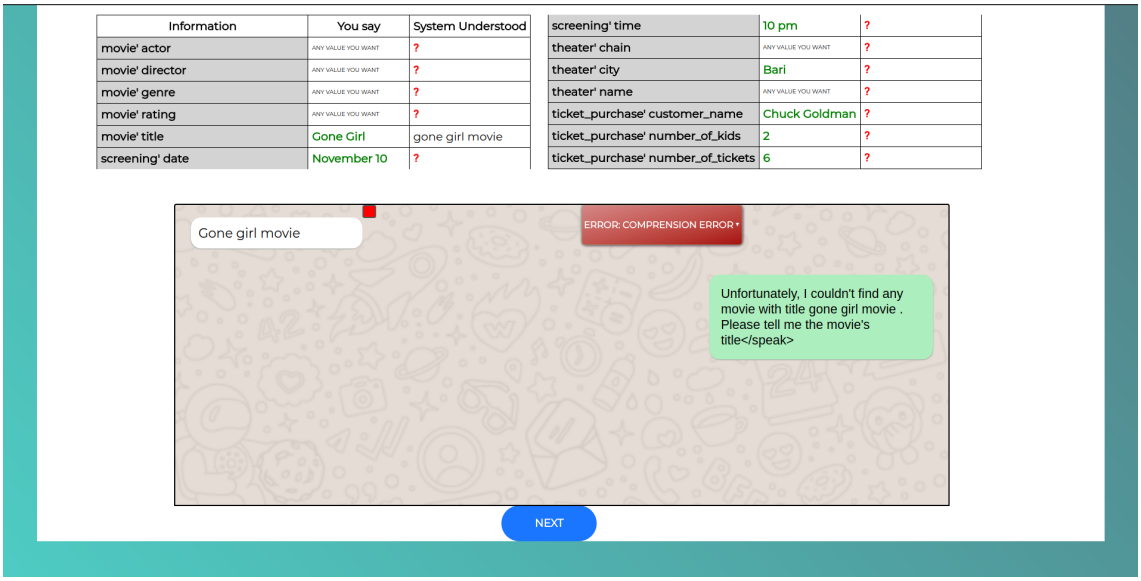
In this scenario the tester is asked to communicate with a dialogue agent, trying to reach a specific provided goal. The goal is characterized by an overall objective (e.g book a movie ticket) and a set of information about slot values to communicate (e.g movie title: The Godfather, screening date: November 08,..). Then the Graphic User Interface offers a chat to communicate with the bot through text and two buttons with the following meaning:

- **Red button:** For some reason, the system wasnt' able to assist the user completely, through the completion of the task.
- **Green button:** This means the system correctly guided the user till the completion of the task.

The tester can choose to press one of the two buttons at any time he wants and, if the green button is pressed the communication closes without asking the user any additional information. Instead, if the red button is pressed the GUI behaves following these steps:



(a) Goal presentation and chat in Baseline scenario



(b) Error selection in baseline scenario

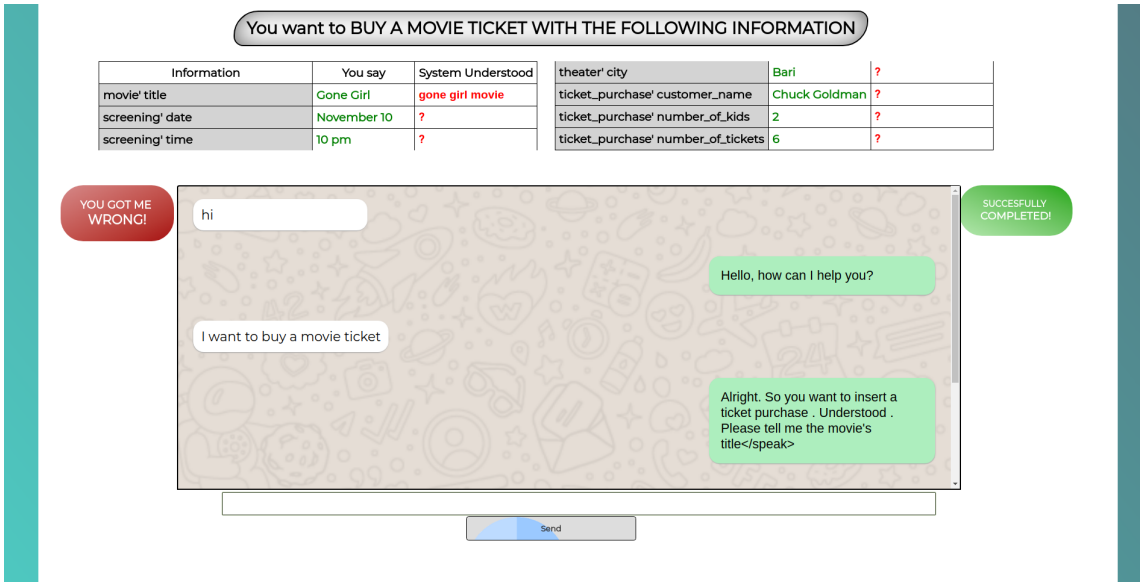
Figure 5.1: Baseline UI

1. Only the last turn is showed to the user.
2. The belief of the system in the last turn is shown. It is composed by all the slot name and values gathered so far.
3. The user is asked to provide an error category to the last turn. He can choose between:
   - COMPREHENSION ERROR: It corresponds to the NLU error category.
   - INTERACTION ERROR: Directly related to the DM error category.
   - BOTH: Corresponding to the BOTH category previously explained.
   - DON'T KNOW: If the user is not able to categorize the error. It does not correspond to any of the previously shown error categories.

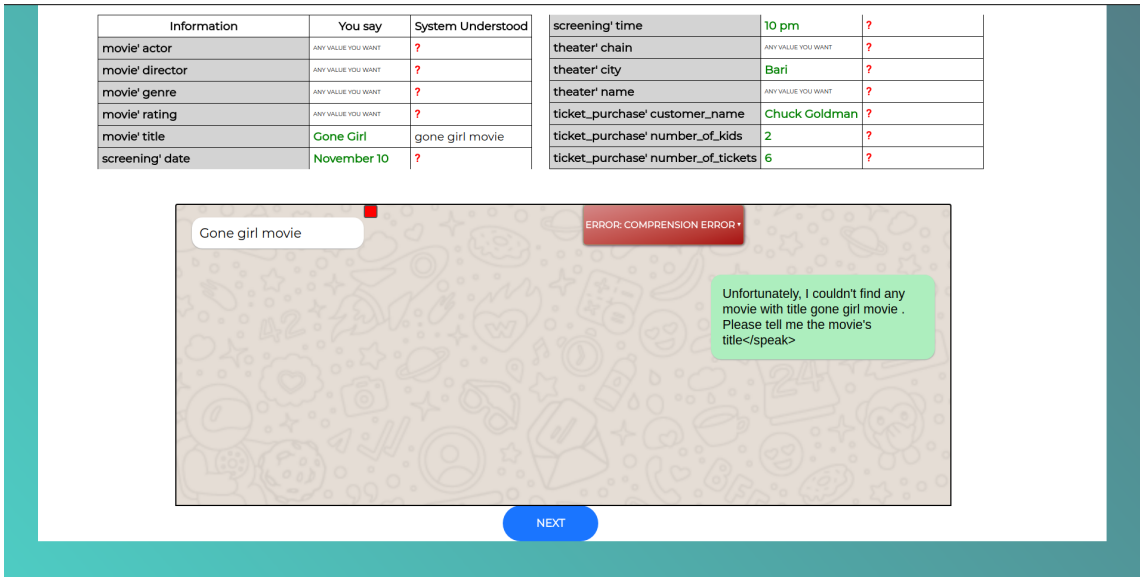
In figure 5.1 it is possible to see the GUI developed for this scenario.

5.3.2 Contextualized scenario

This scenario provides the goal, the chat and the two buttons exactly like the baseline scenario does. What changes with respect to the previous one is the error selection strategy we offer to the user. In



(a) Goal presentation and chat in Contextualized scenario



(b) Error selection in contextualized scenario

Figure 5.2: Contextualized UI

fact, if the red button is pressed we follow these steps:

1. All the turns are showed to the user, from the beginning of the chat.
2. The user can navigate the turns and the related system belief clicking on the turn. So he is able, for each turn, to read the text and what the system understood so far.
3. The user is asked to mark the first turn where the agent made a mistake and to provide the error category for that turn. The list of available errors doesn't change with respect to the previous scenario.

In figure 5.2 it is possible to see the User Interface provided for this scenario.

5.4 Data gathering: Experiments setup

We decided to run the experiments on the movie domain shown in the previous chapter.

Since we needed a complete dialogue agent for the testing, with the second instance of Cookie-Cutter [1] we produced a dialogue system with the most performing DM presented in the previous

[1]The second instance also bootstraps the NLU module. Done by Federico Giannoni in its thesis work.

Scenario	N Dialogues	N Turns	Success	NLU errors	DM errors	BOTH errors	IDK errors
Baseline	20	278	12	3	4	0	1
Contextualized	28	342	11	8	8	0	0

Table 5.1: Statistics about dialogues gathered during the experiments

Scenario	Turn samples	NONE error	NLU error	DM error	BOTH error
Baseline	277	270	3	4	0
Contextualized	294	278	8	8	0

Table 5.2: Statistics about turn samples gathered during the experiments

chapter. Concerning the NLU module, we used the most performing NLU module we were able to produce at the time, the one bootstrapped with template-based utterances. Since the bootstrap of the NLU module is not part of this thesis work, we refer the reader to the master thesis of Federico Giannoni in order to better understand how he designed and implemented such bootstrapping.

The goals to be used in such experiments have been generated through the Factory module of Cookie-Cutter shown two chapters ago. So, the goals are not forced to be completely valid, meaning that some slot value could bring to invalid DB configurations, forcing the user to change its goal, which is a realistic assumption in real task oriented conversations.

Since we'll need to featurize each turn of each dialogue, during the interaction with the dialogue agent, we store:

- The turn number
- The **entities** recognized in that turn
- The **belief** of the system, as the set of slot values recognized so far.
- The **text** typed by the user.
- The set of **actions** predicted by the system.

Regarding the label for each turn, when the user decides to close the interaction with the green button all the turns are labelled with the NONE error label. Instead, if the red button is used, all the turns previous to the one marked are labelled as NONE error, the one selected is labelled with the user label and all the subsequent ones are discarded, since we cannot know their true label.

Concerning the testers, we decided to pick 2 testers, each one having 1 hour to complete as many dialogues he can. Half of the hour on the baseline scenario and the remaining half on the contextualized one. In addition we subjected the two testers to a training session were we shown them how the GUI works in both scenarios and the meaning of the error categories. Finally, the 2 tests have been run in different days and the order of the scenarios presented to the user has been changed between an experiment and the other.

5.5 Data gathering: Experiments statistics

After have run the experiments and gathered the dialogues, we've calculated some statistics, which can be found in table 5.1.

How it is possible to see we were able to gather more data in the contextualized scenario, for a total of 28 dialogues against the 20 gathered through the baseline one. Out of those 28, 11 were succesfull dialogues bringing to a correct completion of the task, 8 NLU errors and 8 were DM erros. Concerning the baseline scenario, 12 dialogues were tagged as succesfull, 3 as NLU errors, 4 as DM errors and just 1 "I Don't Know" label.

Since we need to train and test data-driven models for turn-level error-detection, I extracted a turn-level dataset for each of the two scenarios. The statistics are shown in figure 5.2 and, how it is possible to see, the great majority of samples (97.47% for the baseline scenario and 94.56% for the contextualized) are assigned to the NONE error category. This because, for each dialogue, at most the tester can label 1 turn as erroneous but all the previous ones are kept as NONE error.

5.6 Data Inspection and Comparison

In order to understand the goodness of the two experimental scenarios we inspected the data, checking all the dialogues marked as erroneous. The steps we've followed to check these dialogues are:

1. Check turn by turn user text, dialogue state, entities recognized and actions predicted.
2. Annotated the tagged errors and possible causes.
3. Annotated true error

After have checked and annotated all the erroneous dialogues we analyzed them and, from an high level point of view, we realized that:

- Testers seems to be quite good in marking comprehension errors (NLU). The most of the errors are misunderstanding or non-understanding of slot values and we provide the system belief for each turn, so the tester has all the information he needs in the UI to take the correct decision.
- Testers seems to find it difficult to discriminate interaction errors (DM). The behaviour of the DM is very task-specific and even for an expert who designed the system is not easy to keep in mind all its possible trajectories.
- In the Baseline scenario the most of the errors were in the previous turn with respect to the one marked. This seems plausible since, in the baseline scenario, we just provide the last turn to be marked and, evidently, people doesn't stop the interaction at the first error but try to go ahead with the conversation.

In order to have a quantitative analysis we decided to create a golden set and compare it with the labelled one, computing an agreement score.

Cohen-K score. As agreement score we decided to use the Cohen-K score. This metric is an ad-hoc statistic to be computed for measuring the inter-rater agreement in the case of categorical items, and it also takes into account the possibility of agreement by chance. Given a confusion matrix as in Figure 5.3, we can calculate the K-Cohen score as:

$$K = (Pr(a) - Pr(e))/1 - Pr(e)$$

Where Pr(a) is calculated exactly as the accuracy metric (correct predictions / total predictions) and Pr(e) represents the probability of agreement by chance and it is calculated as:

$$Pr(e) = (P * P' + N * N')/T^2$$

where T is the total number of elements in the confusion matrix. The score can range within -1 and 1 and, depending on the value, we can have the following degrees of agreement:

- $\mathbf{k < 0}$: No agreement.
- $\mathbf{0 \leq k < 0.4}$: Scarce agreement.
- $\mathbf{0.4 \leq k < 0.6}$: Discrete agreement.
- $\mathbf{0.6 \leq k < 0.8}$: Good agreement.
- $\mathbf{0.8 \leq k \leq 1}$: Optimal agreement.

So, for each scenario we had:

1. Annotated all the erroneous dialogues
2. Annotated a random sample of dialogues marked as correct

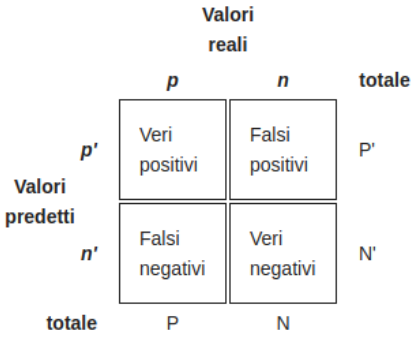


Figure 5.3: Confusion matrix

Scenario	Cohen-K score
Baseline	0.243
Contextualized	0.363

Table 5.3: Cohen-K score in both scenarios in a multi-class case

3. Compared the gold label with the one given by the testers, using as agreement metric che Cohen-K score previously explained

The scores are shown in table 5.3 and, how it is possible to see, in both the scenarios, we have a scarce agreement.

At this point, since the multi-class labelling case seemed to be an hard task for an annotator, we tried to collapse all the error categories into a single ERROR category, passing to a binary case where we have only the NONE and the ERROR categories. Then, using a python script, we re-mapped all the dialogues labelled by the testers to the new label-set, also recomputing the Cohen-K score. The results of this experiment are shown in table 5.4. In this binary setting, the baseline scenario produced again a scarce agreement score but the contextualized one gave us an optimal agreement. The low K score in the baseline scenario can be motivated by the fact that, as previously stated, the most of the errors were in the turn previous to the one marked but we only show the last turn to the user in order to provide a label. Instead, the optimum result reached with the contextualized scenario is given by the fact that most of the DM errors were tagged as NLU errors and, in a binary case, they are under the same class, raising up the overall agreement.

5.7 Error Detection experiments

Given the data analysis showed in the previous section we decided to run the experiments only on the data gathered with the contextualized scenario, in a binary setting, since these data are much less noisy with respect to all the others. So, the classification problem blows down to classify a turn as NONE or ERROR. In the following we are going to show and describe features and data-driven models used for such experiments.

5.7.1 Features

As previously stated, during the data gathering process, for each turn we store: turn number, entities recognized, system belief, user text and predicted actions. Out of these information we generate several feature vectors that will be fed to the data-driven models, these vectors are the following:

- **Intent vector:** This is a binary vector representing in a one-hot-encoding setting the intent

Scenario	Cohen-K score
Baseline	0.379
Contextualized	0.862

Table 5.4: Cohen-K score in both scenarios in a binary case

used by the user in such turn. The one-hot encoding forces only 1 bit of the vector to have value 1 (only the intent recognized), keeping all the others as 0.

- **Entities vector:** Binary vector indicating which entities have been recognized in the featurized turn.
- **System Belief vector:** Binary vector indicating all the slots recognized so far by the system.
- **Actions vector:** Binary vector indicating all the actions predicted by the system in that turn.
- **User text vector:** Vector representing the user text under the form of Bag of Words. It is a vector of size N, where N is the size of the vocabulary used by the user, where each element corresponds to a word in the vocabulary and it works as a counter of that word in the current turn.
- **Confidence vector:** This vector embeds the confidence scores of either the NLU or the DM module, additionally we also offer a setting in which both confidences are concatenated.
- **DB consistency vector:** This is a binary vector with the same length of the entities recognizable by the system. Each bit corresponds to an entity and if its value is 1 it means that the entity recognized by the system is available in the database, otherwise its value is 0.

Given these feature vectors, we want to try all the possible combinations with all the possible data-driven models we're going to show. In addition we also make use of contextualization, leveraging a dialogue history parameter which triggers a concatenation of the feature vectors of the previous N turns to the current one.

5.7.2 Models

Concerning the models used for this experiment I decided to implement 3 baseline models, 2 supervised learning models and two novelty detectors which we are now going to be briefly reviewed (a detailed explanation of all the models can be found in the Appendix)

Random classifier

This is the most stupid classifier and it classifies a turn randomly, according to an uniform probability distribution. So it doesn't need any training procedure to be called.

Threshold Classifier

Given a turn it checks the confidence levels of both NLU and DM modules against two thresholds T1 and T2. If at least one of the two confidence levels is under the related threhsold the turn is classified as ERRROR, otherwise it is classified as NONE. The thresholds T1 and T2 are learned on the training set by averaging the NLU and DM confidence scores.

Majority Model

This model classifies each turn with the majority class. The majority class is the class with the higher frequency in the training set.

Linear SVM

A Linear SVM is a particular instance of a Support Vector Machine. Introduced by Vapnik in 1963 [35], SVMs are Supervised Learning models which can be used to solve both classification and regression problems. As any other SL model it needs a labelled dataset to work but, differently to many other models, it tries to find an hyperplane which discriminates the data-points in the feature space, maximizing the margin.

In our project we used the LinearSVM implementation offered by sklearn [16]

Logistic Regression

The logistic regression is a SL model and one of the most used models appertaining to the class of generalized linear models. Differently from linear regression, the logistic regression predict probabilities, which can be compared to the probabilities of other models.

For our purposes we decided to use the sklearn implementation of the logistic regression [17]

Isolation Forest

This model is used as a novelty detector, meaning that its purpose is to identify novelties (or outliers), so patterns in the data that does not conform to the expected behaviour. In contrary to the previous models, this is an unsupervised model, meaning that we don't need label to train it, but just positive examples.

As for the previous data-driven models, we used the implementation offered by sklearn [18].

Multi-layer perceptron

Another SL model we tried is the Multi-layer perceptron, a Neural Network modelled with multiple layers.

As python implementation for the neural network, in our project we decided to use the implementation offered by sklearn [19]

Neural Anomaly detector

As last model we decided to exploit a particular instance of neural networks, the so called Neural Anomaly Detectors. As the Isolation Forest, this is an anomaly detector trying to identify outliers but they make use of Autoencoders as core of computation.

Regarding the implementation, we found an available GitHub [7] project implementing this detector.

6 Automatic Error Discovery: Evaluation

6.1 Methodology and metrics

In order to evaluate the models described so far on all the feature sets presented we adopted an evaluation methodology that we called **leave one dialogue out cross-validation**. As the well known k-cross validation metric we run multiple training-testing rounds on the full dataset, each time changing the training set and the test set. At each round a full dialogue is kept as test fold and the remaining ones are used as training set; the overall performances are calculated averaging the performances on all the folds.

Evaluation Metrics Concerning the evaluation metrics, we chose the following one:

- **Accuracy**: Given the confusion matrix shown in figure 6.1 we can calculate the accuracy score as:

$$Accuracy = (TP + TN)/(TP + TN + FP + FN)$$

  As evaluation metric it works well if there are equal number of samples belonging to each class, since it doesn't keep into account any balancing of the classes.

- **Precision**: It is calculated as:

$$Precision = TP/(TP + FP)$$

This metric tells you how many of the positive predictions did the machine correctly identify among all the positive predictions it made.

- **Recall**:

$$Recall = TP/(TP + FN)$$

Recall tells us how many of the positive predictions did the machine correctly identify among all the true positive predictions.

- **F1**: The F1 score is the harmonic mean between precision and recall. It tells us how precise is the classifier and how robust it is. It can be expressed as:

$$F1 = 2 * (1/((1/Precision) + (1/Recall)))$$

Given the high unbalancing of positive and negative classes in our dataset, we'll mainly focus on F1 score as summary metric.

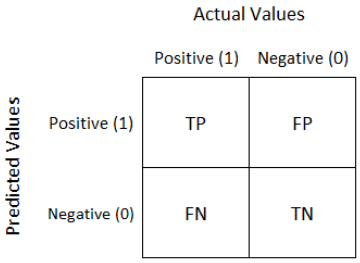


Figure 6.1: Confusion matrix

6.2 Results

In table 6.1 it is possible to see the top 10 results. Analyzing all the results we discovered that basically all the models were able to beat the Majority class in all the tests. The overall best performing model is the Logistic Regression, beating the remaining models in the 65% of the tests with a top F1 score of 72.37%. The second most performing model is the MLP, reaching a 70.44% F1-score and appearing the 9% of the time as best model, finally we have the Threshold classifier which was able to reach the 55.12% F1-score, covering the 34.50% of the rankings.

Analyzing the feature sets for the top 120 results ($\geq$ 65% F1) we discovered the following:

- The user text expressed as **BoW** is the most discriminant feature for all the models, with a 75% of appearances in the top 120 and 90% in the top 10.

- The features extracted from the NLU module (intent + entities) are equivalent in terms of impact on the performances. In fact they appear always together in the 60% of the cases. In the top 10 they appear for the 50% of the cases, meaning that they are helpful for the discrimination problem.

- The DB feature is equivalently important to the NLU module features reaching the same frequencies in both the top 10 and top 120.

- Confidence features seem to help, appearing the 77% of the times in the top 120 and 70% in the top 10. Between all the possible values this feature can obtain, the best ones resulted to be both at act level and act + entities level.

Context	Intent	Acts	Belief	Entity	BoW	DB	Confidence	RANDOM	THRESHOLD	MAJORITY	SVM	LOGIT	ISOLATION FOREST	MLP	NEURAL DET
1	True	False	False	True	True	True	none	0.36	0.55	0.43	0.43	0.72	0.49	0.56	0.42
1	True	False	False	True	True	True	act	0.46	0.55	0.43	0.43	0.72	0.45	0.56	0.42
1	False	False	True	False	True	True	act	0.39	0.55	0.43	0.43	0.72	0.46	0.65	0.42
1	True	False	False	True	True	False	act	0.50	0.55	0.43	0.43	0.72	0.43	0.58	0.42
1	True	False	False	True	True	False	none	0.48	0.55	0.43	0.43	0.72	0.43	0.54	0.42
0	False	False	True	False	True	True	act	0.39	0.55	0.43	0.43	0.71	0.46	0.54	0.42
0	False	False	True	False	True	True	act + entity	0.38	0.55	0.43	0.43	0.71	0.46	0.56	0.42
1	False	True	False	False	False	True	act + entity	0.37	0.55	0.43	0.43	0.61	0.50	0.70	0.46
1	True	False	True	True	True	True	act	0.50	0.55	0.43	0.43	0.70	0.43	0.58	0.42
1	False	False	True	False	True	True	none	0.36	0.55	0.43	0.43	0.70	0.46	0.58	0.42

Table 6.1: Top 10 performances. The features are represented with the yellow color, the caps lock names are the models and the green cells are the top results.

- Contextualization helps in discriminating errors. Specifically, a context window of 1 previous turn resulted to be in the 80% of the cases in the top 10 and 60% in top 120. Bigger contextualization windows, in this case, didn't help that much.
- The featurized system belief and acts (information from DM module) predicted by the system appear with a frequency of 50% in the top 120 results and with lower frequencies (35%) in the top 10, meaning that these are not too discriminant features for the models used.

Final Overview The results and insights presented so far showed us that a simpler model like the Logistic Regression was able to reach the best performances on more or less all the feature sets. Instead, more complex models like the neural detector and the MLP, weren't able to compete with the Logit, probably because these usually require a large amount of data to proper discriminate and generalize on new samples and, with our data-gathering experiments, we did not gather enough data but the 8th position reached by the MLP suggests that with more data it could reach very good performances. Concerning the feature sets, we discovered that the user text and DB features were the most discriminant features, indicating that the natural language and the related uncertainties are very helpful for tracting errors, as stated by Teigen in [96]. In addition we discovered that information coming from the NLU module are more important with respect to the ones coming from the DM module, probably because the NLU module commits much more errors with respect to the DM one. Finally, we also confirmed the fact that keeping into account previous turns helps a lot in the discrimation process.

7 Conclusions

We built Cookie-Cutter, a framework for the fast bootstrapping of data-driven task-oriented dialogue systems in absence of dialogue data. In order to build the framework we developed a data-generation process involving a multi-agenda based user simulator and a domain agnostic rule based dialogue manager which, starting from a task definition and a populated catalogue, generate high quality data for all the modules needed by a data-driven dialogue agent. Then we enriched the process through a data-ingestion procedure, feeding the generated data to the RASA framework which produces the final data-driven bot, tailored to both the task and the catalogue ingested. We tested the framework under two different scenarios and three settings, varying the featurization of the produced data and testing different RASA settings and we discovered that the best DM policy produced (intent+entities) is able to deal with cooperative and non-cooperative simulated users, reaching a 2.92% AER and 4.82% AER in the restaurant and movie domain respectively.

Additionally we also run experiments on error-detection on BOTs produced with Cookie-Cutter. We started building two data-gathering scenarios, asking to real users to communicate with the bot to reach a specific goal and, in case of an erroneous conversation, to give a specific error label among 4 different categories. Then we analyzed the gathered data in both scenarios, comparing them with a golden label dataset we generated and we discovered that this multi-class label setting seems to be quite hard for general users. In particular users are not enough experts to understand the complete behaviour of a DM, and they get confused in tagging those errors, instead they seem quite good in tagging NLU errors. For this reason we decided to run the ML experiments in a binary scenario, were the initial 4 error categories were collapsed in NO_ERROR and ERROR. Then we modelled the feature sets keeping into account all the information coming from the bot modules (NLU + DM) and from the external knowledge base, also leveraging contextualization. We tested several heuristic and data-driven models on all the possible combinations of feature sets and we discovered that the best performances were reached by the most simple data-driven model we had (Logistic Regression), probably because the low amount of data and the low correlation between the features which is an assumption of the model. Finally we discovered that the user text is the most discriminant feature and information coming from the NLU module, together with contextualization, help a lot the discrimination process.

Appendix A Sentence embedding

With the advance of deep learning techniques, in recent years word and sentence embedding algorithms have become very popular and used in many NLP tasks. These algorithms encode words and sentences in fixed-length dense vectors with the objective of having an high similarity between vectors obtained by words or sentences either with the same semantic meaning (e.g "boat" - "ship") or semantically related (e.g "boat" - "water"), depending on the algorithm used.

As a form of transfer learning [71], an huge ongoing trend is about Universal Embeddings, embeddings which are pre-trained on huge corpus and can be plugged in several downstream task models (intent classification, transaltion, sentimental analysis,...).

A.1 Universal Sentence Encoder

One of the best performing sentence encoder is the Universal Sentence Encoder developed by Google in 2018 [31]. It is modelled as a Transformer (encoder-decoder explained during the introduction), but only the encoder (figure A.1) is used in order to provide the embeddings. The encoder is composed by six identical layers, each one having two sub-layers. The first sublayer is a multi head self-attention mechanism which focuses on the most important slices of embeddings at each layer. The second sublayer is a simple fully connected feed-forward network. In addition, they also employ a residual connection around each of the two sub-layers, followed by a layer normalization. Since no recurrence or convolution is used in such model, the information about the relative and absolute position of the tokens in the sequence are given by an additional vector called "positional embeddings". The model is trained in an unsupervised setting and, like an autoencoder, it tries to reproduce the input as output, with an output vector of 512 dimensions. This architecture has been proven to be one of the most performing one in a lot of different NLP tasks, but it comes at the cost of computing time and memory usage scaling dramatically with sentence length.

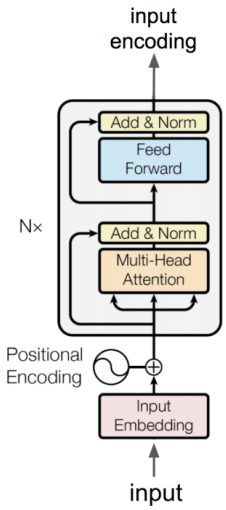


Figure A.1: Transformer encoder used is USE.

Appendix B Recurrent Neural Network

According to Sutskever [94], a Recurrent Neural Network (RNN) is a generalization of the standard feedforwards neural networks, applied to sequences. In general, it takes as input a sequence $(x_1, ..., x_t)$ and computes as output a sequence of elements $(y_1, ..., .y_t)$, based on the following equations:

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

$$y_t = W^{yh}h_t$$

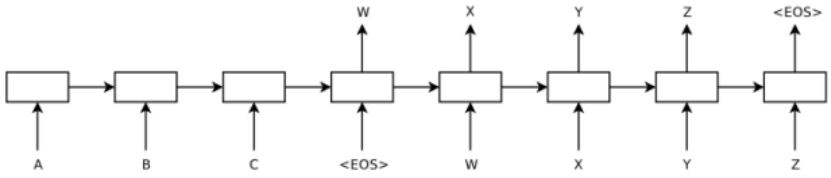


Figure B.1: Sequence-to-Sequence task

As it is possible to see in figure B.1, in a RNN the output of the network is also provided as input to the network in the next step. As stated by Sutskever [94], a Recurrent Neural Network estimates the conditional probability $p(y_1, .., y_t||x_1, .., x_t)$. So, the probability of the output sequence can be computed as:

$$p(y_1, .., y_t||x_1, .., x_t) = \prod_{t=1}^{T} p(y_t||v, y_1, .., y_{t-1})$$

where v is the representation of the input. Additionally, in 1997 Hocreiter [47] applied a more complex recurrent unit cell, called LSTM, which put more focus on learning long-term dependencies.

Appendix C Error detection models

In the following we'll describe in depth all the models used for the experiments on the error detection

C.1 SVM

A Linear SVM is a particular instance of a Support Vector Machine. Introduced by Vapnik in 1963 [35], SVMs are Supervised Learning models which can be used to solve both classification and regression problems. As any other SL model it needs a labelled dataset to work but, differently to many other models, it tries to find an hyperplane which discriminates the data-points in the feature space, maximizing the margin (more on Appendix). Maxizimize the margin means that the algorithm tries to find the hyperplane furthest away from any data point, which is considered the optimal hyperplane. This separating hyperplane can be expressed in terms of the data point that are closest to the boundary, also called **support vectors**. Just to give an idea of what a margin is, in figure C.1 it is possible to see a feature space populated with some data-point about women and men, featurized according to their weight and height. Given a particular hyperplane H, we can compute its margin as its distance between the closest data-point, doubled. It is basically a slice of the feature space in which no data-point will land. Obviously this is an hard constraint for a feature space, in fact these SVMs are called hard-margin SVMs and they can lead to no solution. For this reason, soft-margin SVMs have been introduced, permitting the minimal constraint violation for the hyperplane selection, making use of slack variables during its computation.

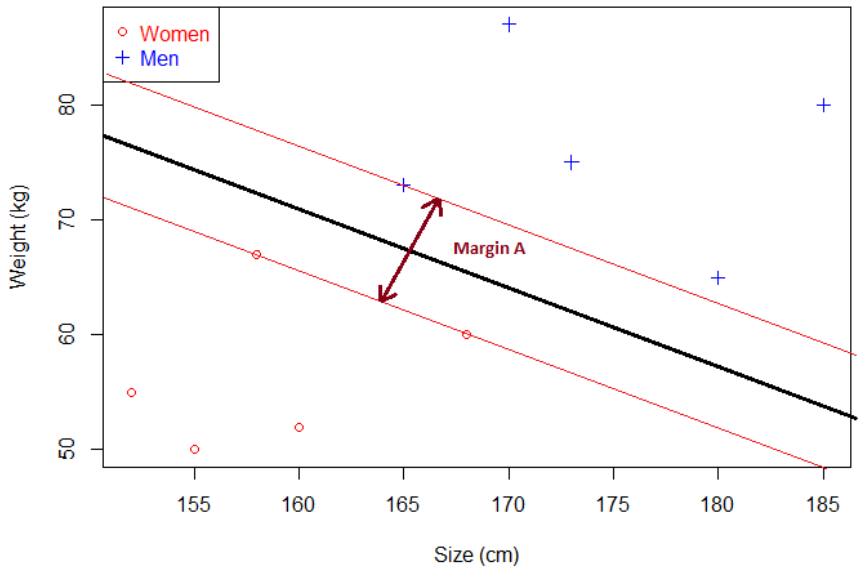


Figure C.1: SVM margin example

From a geometrical perspective an hyperplane is just a line in a multi-dimensional space and it

can be defined as:

$$w^t x = 0$$

A line, in a two-dimensional feature space, is defined as:

$$y = ax + b$$

which is the same thing as:

$$y - ax - b = 0$$

and, given two vectors **w** = (-b, -a, 1) and **x** = (1, x, y), we can write:

$$w^t * x = -b * (1) + (-a) * x + 1 * y$$

$$w^t x = y - ax - b$$

So, the two equations are just different ways of expressing the same thing.

As previously stated, SVMs selects as final solution an hyperplane maximizing the margin. Given an hyperplane H_0 separating the dataset and satisfying the equation

$$w * x + b = 0$$

we can always select two other hyperplanes H_1 and H_2 which also separate the data, with the following equations:

$$w * x + b = \delta$$

$$w * x + b = -\delta$$

so that H_0 is equidistant from H_1 and H_2, however since the δ variable is not necessary, we can set it to 1 in order to simplify the problem, having:

$$w * x + b = 1$$

$$w * x + b = -1$$

We want to be sure that there are no points in between these two hyperplanes, so we'll only select those who meet the following constraints for each vector x_i:

- w * x_i + b $\geq$ 1 for x_i having positive class.
- w * x_i + b $\leq$ -1 for x_i having negative class.

These constraints are shown in figure C.2. It can be shown that these two constraints can be re-written under a single constraint, with the form

$$y_i * (w * x_i + b) \geq 1$$

Under this constraint we can find several solutions for the hyperplane but, as previously stated, SVMs find the solution maximizing the margin. Looking at figure C.3, let:

- H_0 be the hyperplane having the euation w * x + b = - 1
- H_1 be the hyperplane with equation w*x + b = 1
- x_0 be a point in the hyperplane H_0

The margin m is the perpendicular distance from x_0 to the hyperplane H_1

In order to calculate the margin, the SVM leverages the perpendiculatity of the vector w to the hyperplane H_1, defining **u** as the unit vector of w:

$$u = w/||w||$$

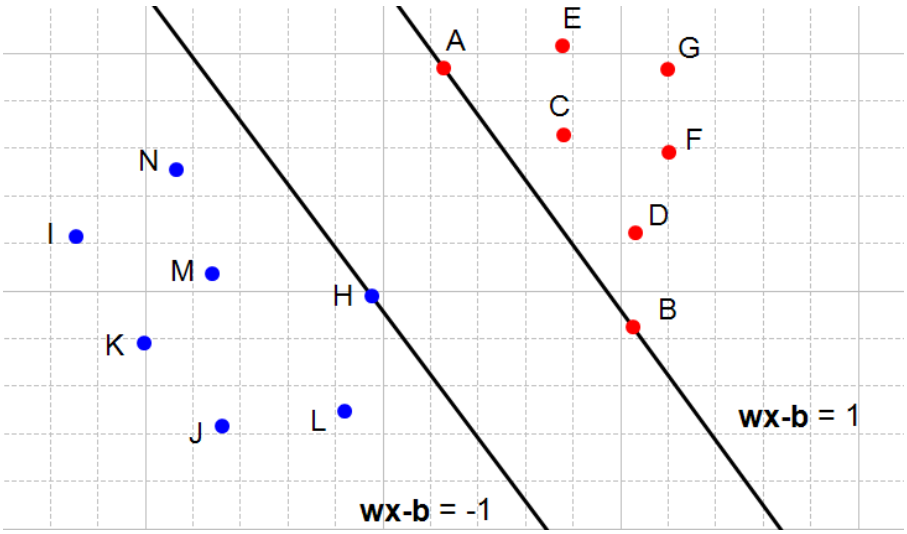


Figure C.2: SVM constraint example

where $||u|| = 1$ and it has the same direction as w, so it is perpendicular to the hyperplane. Multiplying **u** by m we get the vector $\mathbf{k} = m\mathbf{u}$, as showed in figure C.4.

It can be shown that adding this vector **k** to x_0 brings to:

$$m = 2/||w||$$

Since our goal is to maximize the margin m, it is equivalent to minimizing the norm of w. This give us the final optimization problem of an hard-margin SVM:

$$Minimizein(\mathbf{w}, b)$$

$$||w||$$

$$s.t y_i * (w * x_i + b) \geq 1$$

Solvers can be used to find the optimal solution for this optimization problem.

C.2 Logistic Regression

The logistic regression is a SL model and one of the most used models appertaining to the class of generalized linear models. Differently from linear regression, the logistic regression predict probabilities, which can be compared to the probabilities of other models. Differently from linear regression, this model assumes that the probabilities describing the possible outcomes of a single trial can be modeled with a logistic function (Figure C.5).

In particular it assumes that the log-odds of an observation y can be expressed as linear function of the K input variables:

$$(log(P(x)/1 - P(x)) = \sum_{j=0}^{K} b_j * x_j$$

If we take the exponent of both sides of the above equation we have:

$$(P(x)/(1 - P(x)) = exp(\sum_{j=0}^{K} b_j * x_j)$$

$$(P(x)/(1 - P(x)) = \prod_{j=0}^{K} exp(b_j * x_j)$$

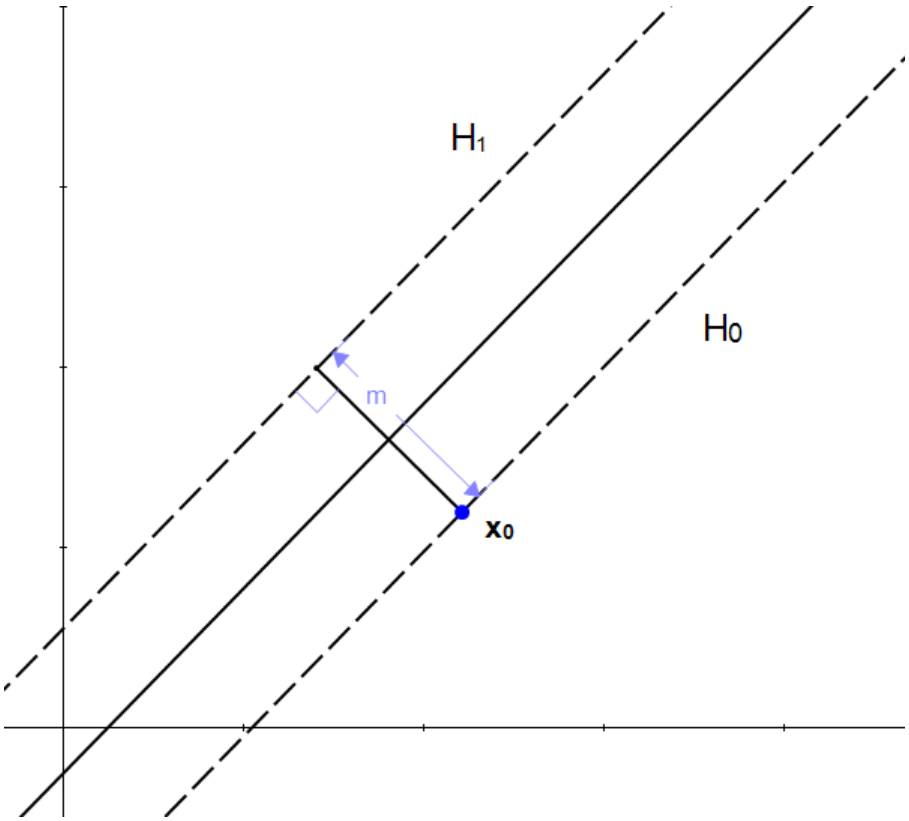


Figure C.3: SVM margin

From this last equation we can understand that logistic models are multiplicative in their inputs, giving a better way to interpret the coefficients.

Inverting the logit equation, we can get a new expression for P(x):

$$P(x) = exp(z)/(1 + exp(z))$$

$$z = \sum_{j=0}^{K} b_j * x_j$$

The right-hand side of this equation is the sigmoid of z, mapping the real lint to the interval (0,1).

So, the problem of finding the best logistic function blows down to find the optimal values for **b** In order to find such values diferrent approaches can be used, from Maximum Likelihood Estimation to the most used Newton's method.

For our purposes we decided to use the sklearn implementation of the logistic regression [17]

C.3 Isolation Forest

This is an ensemble model, so it is composed by many weak models which, in this case, are Decision Trees. The goal of a decision tree is to predict the value of a target variable by learning simple decision rules on the top of the data features, these rules can be interpreted as if-then-else statements and are arranged in the form of a tree. An example is shown in figure C.6, here a decision tree for deciding if playing Badminton is presented. The internal nodes (i.e non-leaves nodes) represent the features on which the game will be discriminated, on the edges we have the possible values for such feature and the leaves represent the final class.

In order to learn this tree, the learning Decision tree algorithm makes use of the information gain. Information gain is a statistical property measuring how well a given attribute (feature) separates the training examples according to the target classification value (the class). As shown in figure C.7, for a given attribute, an high information gain splits the data into groups with an uneven number of

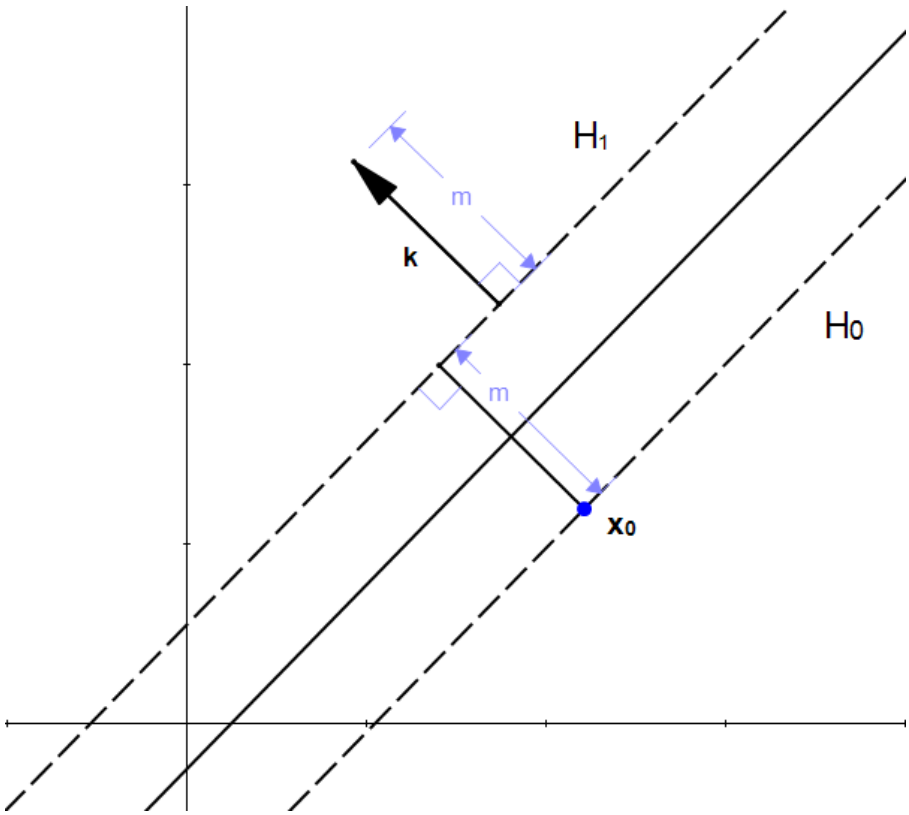


Figure C.4: K is a vector of lenght m perpendicular to H1

positives and negatives, helping to separate the two classes from the other. Instead, a low information gain produces an high class entropy in the groups generated. The Information Gain is based on the entropy:

$$Entropy : \sum_{i=1} -p * log_2(p_i)$$

$$p_i = Probabilityofclassi$$

The entropy is a measure of the impurity of a sample and it ranges from 0 to 1, the higher the value the more impure is the sample. Given this entropy value we can define the information gain (IG) as a measure of effectiveness of an attribute in classifying the training data.

$$IG(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} rac|S_v||S| \cdot Entropy(S_v)$$

In the above equation Values(A) are all the possible values for the attribute A and S_v is the subset of S having as attribute value for A the value v. The first term of the equation is the entropy calculated on the original samples S and the second term is the expected value of the entropy after S is partitioned using A as attribute (i.e the entropy of its children). This expected entropy is just the sum of entropies of each subset S_v, weighted by the fraction of examples $rac|S_v||S|$ belonging to S_v. So, the final IG is the expected reduction in entropy caused by knowing the value of attribute A. So, in order to build the final Decision tree, we start from attributes having the highest IG and we end-up when all the attributes have been processed. Pruning strategies can be used to avoid too much overfitting on the data and allow generalization.

An Isolation forest build N decision trees which are created by randomly selecting a feature and then selecting a random split value between the minimum and maximum value of the selected feature. The basic idea is that outliers are less frequent than regular observations and they differs from the regular ones in terms of attribute values. This is why using the random partitioning the outliers should be identified closer to the root of the tree, having a shorter average path length. So, once all the trees have been built a sample is classified as anomaly based on the average path length it takes in all the

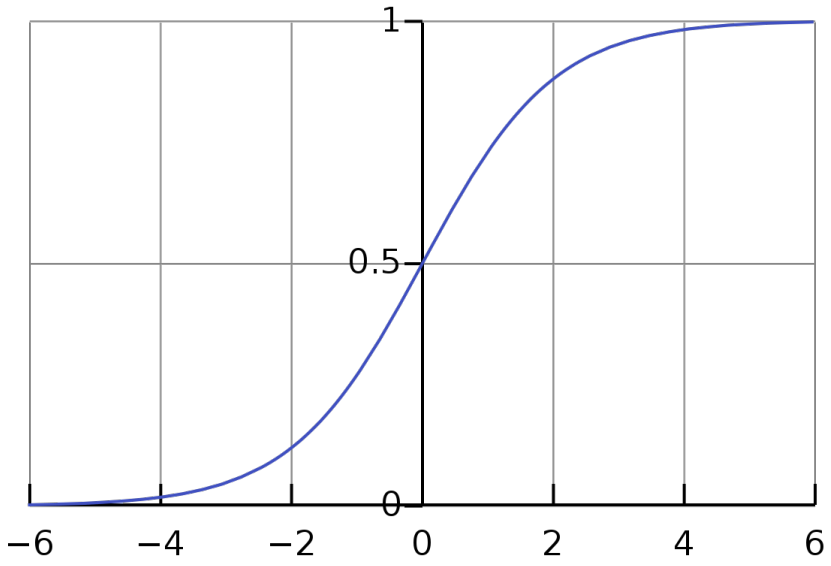


Figure C.5: Logistic function

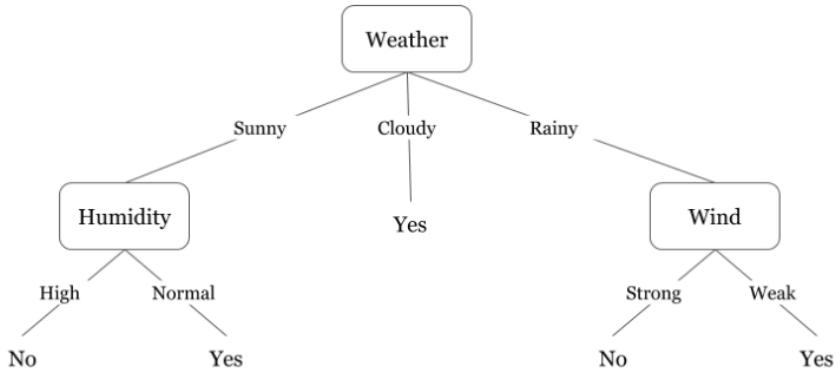


Figure C.6: Example of decision tree..

trees. The final score ranges from 0, meaning normal observation, to 1, meaning anomaly.

C.4 Multi-layer Perceptron

A neural network can be seen as an ensemble of neurons as the one shown in figure C.8. This figure shows how the neuron N_i, given an input, computes its output a_i, called activation. The input to this neuron can be the output of a precedent one, or a feature coming directly from a sample. Its output can be either the input for another neuron or one of the output of the neural network.

The computation of the output of a neuron is produced as follow:

1. The neuron N_i has a weight value w_i for each input and, as first step, it computes a weighted sum of all its input, called in_i:

$$in_i = \sum_{k \in Inputs(N_i)} w_{i,k} a_k$$

where the $Inputs(N_i)$ are all the inputs passed to the neuron.

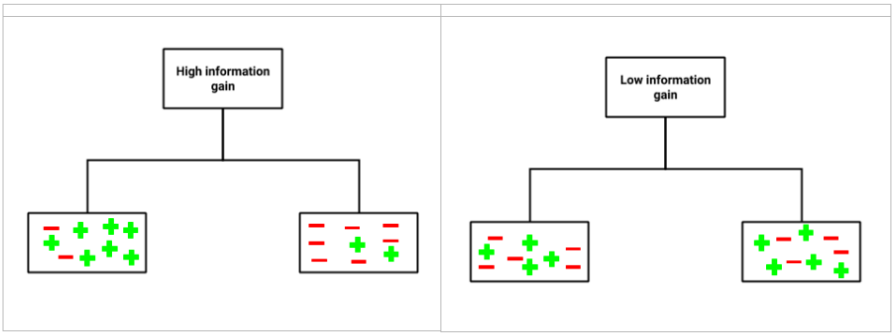


Figure C.7: Example information gain split

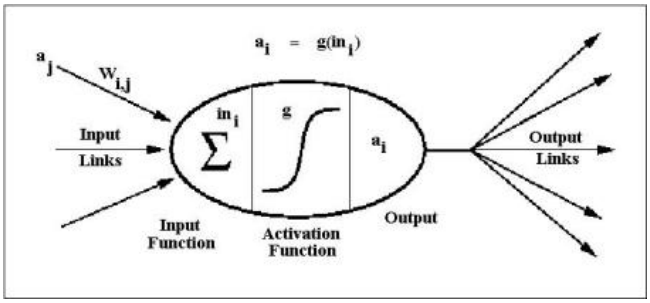


Figure C.8: Neuron example

2. The sum in_i is passed to an activation function, which is the output of the neuron:

$$a_i = g(in_i)$$

It is possible to use different activation functions, such as linear functions, rectified linear function, ...

So, given a neural network as shown in figure C.9, it is possible to compute its overall output starting from the inputs and computing the activation value of each neuron till the output layer.

Once a neural network has been designed (e.g choosing number of layers, number of neurons for each layer, activation functions, ...) the objective of the training process is to find the best values for all the neuron weights $w_{i,j}$ so that the whole network gives the right output for the given input.

In order to optimize these weights the **backpropagation** algorithm is used. This algorithm makes use of an error function between the true label and the predicted label of the network and leverages gradient descent in order to understand how to change properly the value of each weight, according to the error calculated. The gradient descent technique is a well-known technique for computing the partial derivative of the error function with respect to a specific weight value. It basically reduces the error function E by taking help of gradients with respect to individual weights $\frac{\partial E}{\partial w_{i,j}}$.

So, for a given training sample, after having computed a feed-forward pass on the network and computed the error E, we need to calculate $\frac{\partial E}{\partial w_{i,j}}$. Using the chain rule in Leibniz's form we have:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial in_i} \frac{\partial in_i}{\partial w_{i,j}}$$

Using basic differentiation we have:

$$\frac{\partial E}{\partial w_{i,j}} = a_j \frac{\partial E}{\partial in_i}$$

a_j can be computed since we know the value of the activation function, given the current weight

values and, using the chain rule we get:

$$\frac{\partial E}{\partial in_i} = \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial in_i}$$

If we use a sigmoid function g(x) it can be easily verified that:

$$alt = " \frac{dg}{dx} = g(x)(1 - g(x))$$

So, we have:

$$\frac{\partial E}{\partial in_i} = a_i(1 - a_i)\frac{\partial E}{\partial a_i}$$

Putting all together we have the final equation:

$$\frac{\partial E}{\partial w_{i,j}} = a_j a_i(1 - a_i)\frac{\partial E}{\partial a_i}$$

The last term can be expressed in two different cases:

- N_i is an output neuron and the term becomes:

  $$\frac{\partial E}{\partial a_i} = -(t_i - a_i)$$

  where t_i is the output of the neuron N_i

- N_i is an inner neuron. In this case we need to consider all the neurons N_k for whom N_i acts as an input. We can assume that we know the values for $\frac{\partial E}{\partial a_k}$ for all of them, since we are propagating backward. Using the simple chain rule. again, we have:

  $$\frac{\partial E}{\partial a_i} = \sum_k \frac{\partial in_k}{\partial a_i}\frac{\partial E}{\partial in_k}$$

  which becomes:

  $$\frac{\partial E}{\partial a_i} = \sum_k w_{k,i} a_k(1 - a_k)\frac{\partial E}{\partial a_k}$$

So, given a training sample, we feed-forward it to the network, we calculate the error E and we compute the gradient of each weight of the NN as shown above, starting from the output nodes till the input ones. Then we update the weight value according to the following rule:

$$\Delta w_{i,j}^t = -\epsilon\frac{\partial E}{\partial w_{i,j}} + \alpha\Delta w_{i,j}^{t-1}$$

Here the ϵ is the learning rate, it ranges from 0 to 1 and it is used to tune how much each training sample contributes to the weight update. The α parameter instead is called momentum and it is used to avoid local optimum solutions. It causes the current weight update to be dependent on the last weight update.

In order to speed up the training procedure we can train a NN in a batch mode, passing N samples for each weight update. In this mode the forward propagation is run with respect to all the samples in the batch and the backward phase is run once with respect to the accumulated results of the batch.

C.5 Autoencoders as anomaly detectors

Autoencoders are unsupervised neural network models, so they do not require labelled data. The objective of this model is to find a function reconstructing the input in the best way possible

$$f(x) = x$$

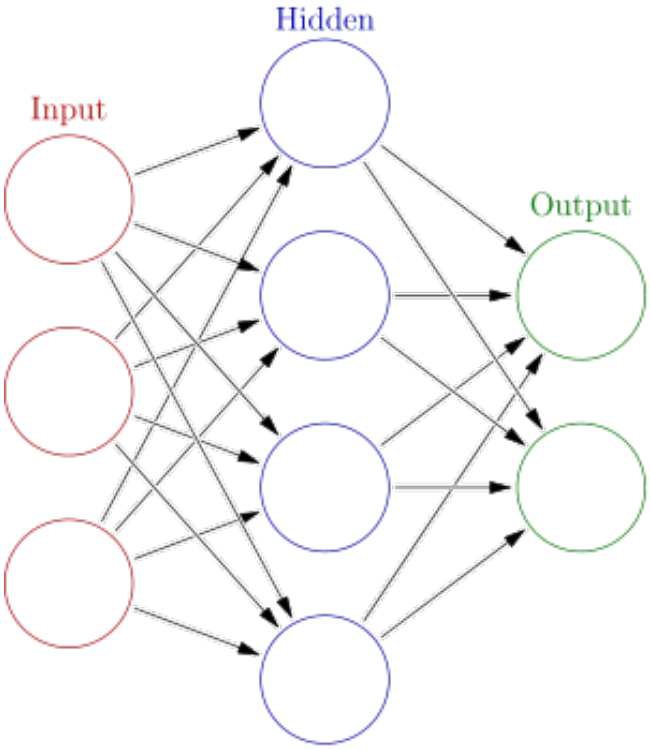


Figure C.9: NN example

So, it is a neural network having outputs equal to the inputs (figure C.10) and trying to minimize the so called reconstruction error (RE) $L(x, x')$, which is usually expressed as the mean squared distance between input and output:

$$L(x, x') \approx ||x - x'||^2$$

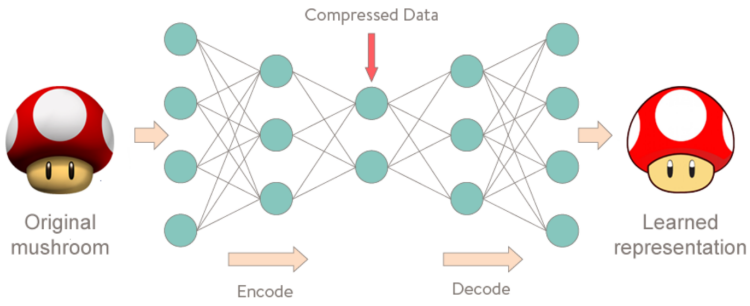


Figure C.10: Autoencoder example

So, given an unlabelled dataset X and a random initialization of the neural network weights, we can train an Autoencoder following these steps for each training sample x_i:

1. Do a feed-forward pass to obtain an output **x'**
2. Calculate the RE between x_i and **x**
3. Use the back-propagation previously presented to perform weight updates.

The core idea behind the usage of autoencoders as anomaly detector is that the reconstruction error of outliers is higher with respect to the reconstruction errors of a normal data-point. So, in order for it to being able to work as anomaly detector, after have trained an autoencoder on normal data-points we should set a threshold T on the RE to separate normal data points and outliers. An

easy way to calculate this threshold T is to use a development set containing labelled anomalies and compute an average threshold value as follows:

1. Calculate AVG_1 as the average RE value for normal data-points.
2. Calculate AVG_2 as the average RE value for abnormal data-points.
3. Calculate T as the average between AVG_1 and AVG_2

Bibliography

[1] 39 million americans now own a smart speaker, report claims. https://techcrunch.com/2018/01/12/39-million-americans-now-own-a-smart-speaker-report-claims/.

[2] Amazon lex. https://aws.amazon.com/it/lex/.

[3] Amazon polly. https://aws.amazon.com/it/polly/.

[4] Amazon transcribe. https://aws.amazon.com/it/transcribe/.

[5] Cloud text-to-speech. https://cloud.google.com/text-to-speech/.

[6] Dialogflow. https://dialogflow.com/.

[7] Github: Anomaly detector. https://github.com/yzhao062/pyod.

[8] Google home assistant stats. https://voicebot.ai/google-home-google-assistant-stats/.

[9] Ibm watson. https://www.ibm.com/watson/.

[10] Intelligent virtual assistant (iva) market size, share trends analysis report by type (speech recognition, text-to-speech recognition), by service, by application, by end-use, and segment forecasts, 2018 - 20248. https://www.grandviewresearch.com/industry-analysis/intelligent-virtual-assistant-industry.

[11] Kaldi. http://kaldi-asr.org/.

[12] Keras. https://keras.io/.

[13] The key market trends driving adoption of virtual assistants. https://whatsnext.nuance.com/customer-experience/virtual-assistants-adoption-rising/.

[14] Rasa, open source tools to build contextual ai assistants. https://rasa.com/.

[15] Robots will eliminate 6% of all us jobs by 2021, report says. https://www.theguardian.com/technology/2016/sep/13/artificial-intelligence-robots-threat-jobs-forrester-report.

[16] Sklearn: Linear svm. https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html.

[17] Sklearn: logistic regression. https://scikit-learn.org/stable/modules/generated/sklearn.linear$_m odel.LogisticReg$

[18] Sklearn: logistic regression. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

[19] Sklearn: Multi-layer perceptron. https://scikit-learn.org/stable/modules/neural$_n etworks_s upervised.html multi-layer-perceptron$.

[20] The state of smart speaker voice search in 2018. https://econsultancy.com/the-state-of-smart-speaker-voice-search-in-2018/.

[21] A survey of available corpora for building data-driven dialogue systems. https://breakend.github.io/DialogDatasets/.

[22] The rise of virtual digital assistants usage – statistics and trends. https://www.go-gulf.com/blog/virtual-digital-assistants/.

[23] wit.ai. https://wit.ai/.

[24] wit.ai claim 39 new languages. https://medium.com/wit-ai/new-languages-15fd1bf9e2ca.

[25] Nabiha Asghar, Pascal Poupart, Jesse Hoey, Xin Jiang, and Lili Mou. Affective neural response generation. In *European Conference on Information Retrieval*, pages 154–166. Springer, 2018.

[26] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[27] Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The berkeley framenet project. In *Proceedings of the 17th International Conference on Computational Linguistics - Volume 1*, COLING '98, pages 86–90, Stroudsburg, PA, USA, 1998. Association for Computational Linguistics.

[28] Antoine Bordes, Y-Lan Boureau, and Jason Weston. Learning end-to-end goal-oriented dialog. *arXiv preprint arXiv:1605.07683*, 2016.

[29] Gillian Brown. Speakers, listeners, and communication: Explorations in discourse analysis. 1997.

[30] Kris Cao and Stephen Clark. Latent variable dialogue models and their diversity. *arXiv preprint arXiv:1702.05962*, 2017.

[31] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *CoRR*, abs/1803.11175, 2018.

[32] Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. A survey on dialogue systems: Recent advances and new frontiers. *ACM SIGKDD Explorations Newsletter*, 19(2):25–35, 2017.

[33] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[34] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[35] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, Sep 1995.

[36] M.Theune E. Krahmer, M. Swerts and M.Weegels. Error detection in spoken human-machine interaction. 2001.

[37] Stephanie Seneff Edward Filisko. Error detection and recovery in spoken dialogue systems. 2004.

[38] Ron Cole et Al. The challenge of spoken language systems: Research directions for the nineties. 1995.

[39] Dafydd Gibbon, Roger Moore, and Richard Winski. *Handbook of standards and resources for spoken language systems*. Walter de Gruyter, 1997.

[40] James R. Glass. Challenges for spoken dialogue systems. 1999.

[41] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[42] Clark Herbert H. Using language. 1995.

[43] James Henderson, Oliver Lemon, and Kallirroi Georgila. Hybrid reinforcement/supervised learning for dialogue policies from communicator data. In *IJCAI workshop on knowledge and reasoning in practical dialogue systems*, pages 68–75. Citeseer, 2005.

[44] James Henderson, Oliver Lemon, and Kallirroi Georgila. Hybrid reinforcement/supervised learning of dialogue policies from fixed data sets. *Computational Linguistics*, 34(4):487–511, 2008.

[45] Matthew Henderson, Blaise Thomson, and Steve Young. Deep neural network approach for the dialog state tracking challenge. In *Proceedings of the SIGDIAL 2013 Conference*, pages 467–471, 2013.

[46] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[47] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[48] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *Advances in neural information processing systems*, pages 2042–2050, 2014.

[49] M.Ostendorfa J.Goldbergb I.Bulykoa, K.Kirchhoffa. Error-correction detection and response generation in a spoken dialogue system. 2005.

[50] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*, pages 137–142. Springer, 1998.

[51] Karl Weilhammer Hui Ye Jost Schatzmann, Blaise Thomson and Steve Young. Agenda-based user simulation for bootstrapping a pomdp dialogue system. 2007.

[52] Bing-Hwang Juang, S. Levinson, and M. Sondhi. Maximum likelihood estimation for multivariate mixture observations of markov chains (corresp.). *IEEE Transactions on Information Theory*, 32(2):307–309, March 1986.

[53] Marc Swerts Julia Hirschberg, Diane Litman. Prosodic cues to recognition errors. *citerseerx*, 1999.

[54] Steve Young Konrad Scheffler. Automatic learning of dialogue strategy using dialogue simulation and reinforcement. *ACL*, 2002.

[55] Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, and Asli Celikyilmaz. End-to-end task-completion neural dialogue systems. *arXiv preprint arXiv:1703.01008*, 2017.

[56] Ryan Lowe, Nissan Pow, Iulian Serban, and Joelle Pineau. The ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. *arXiv preprint arXiv:1506.08909*, 2015.

[57] Zhengdong Lu and Hang Li. A deep architecture for matching short texts. In *Advances in Neural Information Processing Systems*, pages 1367–1375, 2013.

[58] Christine Pao Lynette Hirschman. The cost of errors in a spoken dialogue system. *ISCA*, 1993.

[59] Julia Hirschberg Marc Swerts, Diane Litman. Corrections in spoken dialogue systems. *ISCA*, 2000.

[60] Irene Langkilde Marilyn Walker, Jerry Wright. Using natural language processing and discourse features to identify understanding errors in a spoken dialogue system. 2000.

[61] Andrew McCallum and Wei Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 188–191. Association for Computational Linguistics, 2003.

[62] Michael F McTear. Spoken dialogue technology: enabling the conversational user interface. *ACM Computing Surveys (CSUR)*, 34(1):90–169, 2002.

[63] Grégoire Mesnil, Xiaodong He, Li Deng, and Yoshua Bengio. Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding. In *Interspeech*, pages 3771–3775, 2013.

[64] Nikola Mrksic, Diarmuid Ó Séaghdha, Blaise Thomson, Milica Gasic, Pei-hao Su, David Vandyke, Tsung-Hsien Wen, and Steve J. Young. Multi-domain dialog state tracking using recurrent neural networks. *CoRR*, abs/1506.07190, 2015.

[65] Nikola Mrkšić, Diarmuid O Séaghdha, Tsung-Hsien Wen, Blaise Thomson, and Steve Young. Neural belief tracker: Data-driven dialogue state tracking. *arXiv preprint arXiv:1606.03777*, 2016.

[66] Olivier Pietquin Olivier Lemon. Machine learning for spoken dialogue systems. *HAL*, 2008.

[67] Sharon Oviatt. Multimodal interfaces. *The human-computer interaction handbook: Fundamentals, evolving technologies and emerging applications*, 14:286–304, 2003.

[68] Gokhan Tür Abhinav Rastogi Ankur Bapna Neha Nayak Larry Heck Pararth Shah, Dilek Hakkani-Tür. Building a conversational agent overnight with dialogue self-play. 2018.

[69] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018.

[70] Harshalata Petkar. A review of challenges in automatic speech recognition. 2016.

[71] Lorien Y Pratt. Discriminability-based transfer between neural networks. In *Advances in neural information processing systems*, pages 204–211, 1993.

[72] Minghui Qiu, Feng-Lin Li, Siyu Wang, Xing Gao, Yan Chen, Weipeng Zhao, Haiqing Chen, Jun Huang, and Wei Chu. Alime chat: A sequence to sequence and rerank based chatbot engine. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 498–503, 2017.

[73] Verena Rieser and Oliver Lemon. Learning effective multimodal dialogue strategies from wizard-of-oz data: Bootstrapping and evaluation. *Proceedings of ACL-08: HLT*, pages 638–646, 2008.

[74] Alan Ritter, Colin Cherry, and William B Dolan. Data-driven response generation in social media. In *Proceedings of the conference on empirical methods in natural language processing*, pages 583–593. Association for Computational Linguistics, 2011.

[75] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[76] M. Kearns M. Walker S. Singh, D. Litman. Optimizing dialogue management with reinforcement learning: Experiments with the njfun system. 2002.

[77] Sunita Sarawagi and William W Cohen. Semi-markov conditional random fields for information extraction. In *Advances in neural information processing systems*, pages 1185–1192, 2005.

[78] Diane Litman Marilyn Walker Satinder Singh, Michael Keams. Reinforcement learning for spoken dialogue systems. 2000.

[79] Konrad Scheffler. Automatic design of spoken dialogue systems. 12 2002.

[80] Jack Chen James A. Landay Nadeem Aboobaker Annie Wang Scott R. Klemmer, Anoop K. Sinha. A wizard of oz prototyping tool for speech user interfaces. *IEEE*, 2000.

[81] Iulian Vlad Serban, Chinnadhurai Sankar, Mathieu Germain, Saizheng Zhang, Zhouhan Lin, Sandeep Subramanian, Taesup Kim, Michael Pieper, Sarath Chandar, Nan Rosemary Ke, Sai Mudumba, Alexandre de Brébisson, Jose Sotelo, Dendi Suhubdy, Vincent Michalski, Alexandre Nguyen, Joelle Pineau, and Yoshua Bengio. A deep reinforcement learning chatbot. *CoRR*, abs/1709.02349, 2017.

[82] Iulian Vlad Serban, Alessandro Sordoni, Yoshua Bengio, Aaron C Courville, and Joelle Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. In *AAAI*, volume 16, pages 3776–3784, 2016.

[83] Iulian Vlad Serban, Alessandro Sordoni, Ryan Lowe, Laurent Charlin, Joelle Pineau, Aaron C Courville, and Yoshua Bengio. A hierarchical latent variable encoder-decoder model for generating dialogues. 2017.

[84] Louis Shao, Stephan Gouws, Denny Britz, Anna Goldie, Brian Strope, and Ray Kurzweil. Generating long and diverse responses with neural conversation models. 2016.

[85] Rajeev Sharma, Vladimir I Pavlović, and Thomas S Huang. Toward multimodal human–computer interface. In *Advances In Image Processing And Understanding: A Festschrift for Thomas S Huang*, pages 349–365. World Scientific, 2002.

[86] Gabriel Skantze. The use of speech recognition confidence scores in dialogue systems. 2003.

[87] Gabriel Skantze. *Error Handling in Spoken Dialogue Systems-Managing Uncertainty, Grounding and Miscommunication*. Gabriel Skantze, 2007.

[88] Yiping Song, Rui Yan, Xiang Li, Dongyan Zhao, and Ming Zhang. Two are better than one: An ensemble of retrieval-and generation-based dialog systems. *arXiv preprint arXiv:1610.07149*, 2016.

[89] Iñigo Casanueva Nikola Mrkšić Lina Rojas-Barahona Pei-Hao Su Tsung-Hsien Wen Milica Gašić Steve Young Stefan Ultes, Paweł Budzianowski. Reward-balancing for statistical spoken dialogue systems using multi-objective reinforcement learning. 2017.

[90] Amanda Stent, Matthew Marge, and Mohit Singhai. Evaluating evaluation methods for generation in the presence of variation. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 341–351. Springer, 2005.

[91] Amanda Stent, Rashmi Prasad, and Marilyn Walker. Trainable sentence planning for complex information presentation in spoken dialog systems. In *Proceedings of the 42nd annual meeting on association for computational linguistics*, page 79. Association for Computational Linguistics, 2004.

[92] Matthew Stone and Christine Doran. Sentence planning as description using tree adjoining grammar. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 198–205. Association for Computational Linguistics, 1997.

[93] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[94] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[95] S. Sutton. Building 10,000 spoken dialogue systems. *IEEE*, 1996.

[96] Karl Halvor Teigen. The language of uncertainty. *Acta Psychologica*, 68(1):27 – 38, 1988.

[97] Gokhan Tur, Li Deng, Dilek Hakkani-Tür, and Xiaodong He. Towards deeper understanding: Deep convex networks for semantic utterance classification. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 5045–5048. IEEE, 2012.

[98] Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*, 2016.

[99] Tsung-Hsien Wen, Milica Gasic, Dongho Kim, Nikola Mrksic, Pei-Hao Su, David Vandyke, and Steve Young. Stochastic language generation in dialogue using recurrent neural networks with convolutional sentence reranking. *arXiv preprint arXiv:1508.01755*, 2015.

[100] Tsung-Hsien Wen, Milica Gasic, Nikola Mrksic, Pei-hao Su, David Vandyke, and Steve J. Young. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. *CoRR*, abs/1508.01745, 2015.

[101] Tsung-Hsien Wen, David Vandyke, Nikola Mrksic, Milica Gasic, Lina M Rojas-Barahona, Pei-Hao Su, Stefan Ultes, and Steve Young. A network-based end-to-end trainable task-oriented dialogue system. *arXiv preprint arXiv:1604.04562*, 2016.

[102] Jason Williams, Antoine Raux, Deepak Ramachandran, and Alan Black. The dialog state tracking challenge. In *Proceedings of the SIGDIAL 2013 Conference*, pages 404–413, 2013.

[103] Jason D Williams, Kavosh Asadi, and Geoffrey Zweig. Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. *arXiv preprint arXiv:1702.03274*, 2017.

[104] Chen Xing, Wei Wu, Yu Wu, Jie Liu, Yalou Huang, Ming Zhou, and Wei-Ying Ma. Topic aware neural response generation. 2017.

[105] Puyang Xu and Ruhi Sarikaya. Convolutional neural network based triangular crf for joint intent detection and slot filling. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 78–83. IEEE, 2013.

[106] Rui Yan, Yiping Song, and Hua Wu. Learning to respond with deep neural networks for retrieval-based human-computer conversation system. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 55–64. ACM, 2016.

[107] Kaisheng Yao, Dong Yu, Frank Seide, Hang Su, Li Deng, and Yifan Gong. Adaptation of context-dependent deep neural networks for automatic speech recognition. pages 366–369, 2012.

[108] Robail Yasrab, Naijie Gu, and Xiaoci Zhang. An encoder-decoder based convolution neural network (cnn) for future advanced driver assistance system (adas). 2017.

[109] Yulan He ; S. Youn. A data-driven spoken language understanding system. *IEEE*, 2003.

[110] Tom Young, Erik Cambria, Iti Chaturvedi, Minlie Huang, Hao Zhou, and Subham Biswas. Augmenting end-to-end dialog systems with commonsense knowledge. *arXiv preprint arXiv:1709.05453*, 2017.

[111] Richard Zens, Franz Josef Och, and Hermann Ney. Phrase-based statistical machine translation. In *Annual Conference on Artificial Intelligence*, pages 18–32. Springer, 2002.

[112] Hao Zhou, Minlie Huang, Tianyang Zhang, Xiaoyan Zhu, and Bing Liu. Emotional chatting machine: Emotional conversation generation with internal and external memory. *arXiv preprint arXiv:1704.01074*, 2017.

[113] Victor Zue. Conversational interfaces: advances and challenges. 2000.