TECHNICAL UNIVERSITY OF MUNICH

TUM Data Innovation Lab

Development of an artificial conversation entity for continuous learning and adaption to user's preferences and behavior

Harshita Agarwala
Robin Becker
Mehnoor Fatima
Lucian Riediger

Advisors:
Andrei Belitski
Olena Schüssler
Laure Vuaille

February 2019

Abstract

Customer interaction is one of the most important aspects of a service environment. Artificial conversational entities provide the opportunity to automate such interactions and to streamline many business processes. Such a system should be able to operate with a minimal amount of data and should be easily expandable, since service environments change often. Additionally, the system should adapt to individual preferences, in order to maximize customer satisfaction. We implemented an artificial conversational entity for a general resort environment that is capable of understanding natural language, generating realistic responses and taking actions according to the customer's intent while remembering the context of the conversation. We propose a method of probabilistically generating arbitrarily large amounts of data with minimal user input, in order to train our model and to make it easily extendable. Our system for natural language understanding produces machine readable output by disambiguating sentences and thus providing a way of performing computationally driven business optimization. Furthermore, we propose a general framework for learning customer preference from the interactions with our system and use it to personalize the customer experience.

Contents

1 Introduction

1.1 Motivation

Artificial conversation entities (ACE) in the form of chatbots or voice assistants have developed significantly over the last decade. The main reason behind this is the progress of technologies in the field of natural language processing, speech recognition, parsing (i.e. the analysis of natural language) and machine-learning.
Many businesses have already realized the advantages of ACEs, especially in the fields of customer service, convenience and streamlining processes. Two very popular applications of ACEs are voice assistants on mobile phones and for domestic use, which enable the user to execute tasks like playing music, taking notes, setting reminders and browsing the internet with voice input. For single commands ACEs have a high precision in detecting the users input correctly. However, a current limitation to a lot of ACEs is the context of a conversation. Detecting if given input is independent from previous input or if it is implicitly related is crucial when holding a conversation. Yet, automating this is very challenging, especially if the variety of input it has to understand is very broad.

1.2 Problem Statement

We aim to develop an ACE which is tailored to a hotel environment. The product is supposed to process hotel services like ordering food and drinks. Hence, our ACE should only be able to detect a limited number of commands which simplifies development and allows to overcome challenges in the field of detecting the context of input. As a result, longer and connected conversations between the customer and the ACE are enabled. Additionally, customer satisfaction shall be increased by learning customer preferences which results in recommendations.
In order to focus on these features, we assume all input to be given in text. The output will also be in text-form only. But since voice-to-text and text-to-voice components are independent components and openly available, they can be added afterwards with relatively little effort.

2 Research

Our implementation was divided into three components. The first part was the Natural Language Understanding. We wanted to create a model that comprehends the input. For example for the input "I would like to drink a coffee", the application should be able to understand that the user wants to place an order for a coffee. The second part is the dialog management component wherein the application should be able to respond to the above request by an output like: "Yes, your order has been placed". Along with these two components, the third includes recommendations and history which is integrated with the second

component.
Natural language understanding (NLU) is a branch of artificial intelligence that uses computer software to understand input in the form of sentences in text or speech format. NLU directly enables human-computer interaction. NLU's ability to comprehend natural human languages enables computers to understand commands without the formalized syntax of computer languages and also for computers to communicate back to humans in natural languages. NLU is tasked with communicating with untrained individuals and understanding their intent, that is, NLU goes beyond understanding words and interprets meaning. NLU is even programmed with the ability to understand meaning in spite of common human errors like mispronunciations or transposed letters or words.[1]

2.1 Existing NLP-Frameworks

2.1.1 Parsers

Parsing, syntax analysis or syntactic analysis is the process of analyzing a string of symbols either in natural language, computer languages or data structures conforming to the rules of a formal grammar.[2] In computer programming, a parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree. [3]
There are many functions in a parser that help in analyzing different structures in a sentence. A few of the important ones:

- Strings of characters are grouped as tokens which represent words
- Part-of-speech (POS) tagging assigns word type and function of word in a sentence (e.g. John buys milk → (John, noun); (buys, verb); (milk, noun))
- NER (Named Entity Recognition) tagging locates and classifies entities in unstructured data to pre-defined categories like 'person', 'organization', 'time'. (e.g. John bought milk on Tuesday → (John, PERSON), (Tuesday, TIME))
- Dependency Parsing assigns a word type to each word and mentions the other word in the sentence it is connected to. (e.g. John buys cold milk → (adj, cold, milk))
- Then semantic and sentiment analyses follow, where words are looked up in a database (e.g. WordNet) for meaning and possibly sentiment (e.g. 'fat' → voluminous, negative)

The most common Natural Language Parsers are Stanford NLP, spaCy and NLTK. The figure below shows a brief comparison between these parsers.
It can be seen that although spaCy is much faster compared to others, Stanford

Figure 1: Comparison between Parsers [4]

Speed: Key Functionalities			
Package	**Tokenizer**	**Tagging**	**Parsing**
SpaCy	0.2ms	1ms	19ms
CoreNLP	2ms	10ms	49ms
NLTK	4ms	443ms	-

Accuracy: Entity Extraction			
Package	**Precision**	**Recall**	**F-Score**
SpaCy	0.72	0.65	0.69
CoreNLP	0.79	0.73	0.76
NLTK	0.51	0.65	0.58

NLP gives better results. Stanford NLP is a Java implementation package but there are open source Python wrappers(packages) available that help in running the same Core-NLP package on Python. Initially we experimented with Stanford's Open IE annotator and dependency parsers. The Open IE was initially thought of as a potential candidate in the pipeline of our Natural Language Processing which could help in splitting complex sentences and identifying multiple intents. Open information extraction (Open IE) refers to the extraction of relation tuples, typically binary relations, from plain text. The main difference from other information extraction is that the schema for these relations does not need to be specified in advance. The system first splits each sentence into a set of entailed clauses. Each clause is then maximally shortened, producing a set of entailed shorter sentence fragments. These fragments are then segmented into OpenIE triples, and output by the system. An OpenIE triplet consists of three parts: subject - relation - object. For example, 'Barack Obama was born in Hawaii' would create a triple `(Barack Obama; was born in; Hawai)` [5]. This annotator, although useful did not result in comprehensive results for the case of an ACE. Being a general purpose annotator and depending heavily on the grammatical structure of the sentence, it gave ambiguous results for sentences that were not grammatically correct or well-formed. Encountering such sentences are highly possible in an ACE as it is a form of natural conversation or chatting. Another important annotator is the Dependency Parser which is also used in the final pipeline. It provides a naive external approach to extract the main entity from the sentence.
The annotators or functions included under the parser packages are very powerful in analyzing the sentence structure. However they have been made in such a way that they work for all scenarios. Depending solely on these annotators without using machine learning would give very generalized results. Therefore, these parser packages have been used along with machine learning models in our application to produce comprehensive results.

2.1.2 Machine Learning Models

Parsing provides a very easy way to extract relevant information from sentences. However, they are general purpose applications and not specific to the task at hand. As the business model is based on a discreet service provision,hence it is

important to adopt machine learning models so that the application is trainable for the given purpose. All most all components of the above mentioned parsers are trainable. However there are other commercial as well as open source models that are easier to train and have multiple functionalities combined together to form robust pipelines. We looked into both open source and commercial models for the purpose of developing chat bots, more specifically "goal-oriented chat bots".
Bot Development framework is a set of predefined functions and classes which developers use for faster development. It provides a set of tools that helps in writing the code better and faster. They can be used by developers and coders to build bots from scratch using programming language. All conversational bots require some fundamental features. They should be able to handle basic input and output messages. They must have natural language and conversational skills. A bot must be responsive and scalable. Most importantly, they must be able to extend an as human as possible conversation experience to the user.[6]
A few of the common commercially available models available are:

- **Facebook Bot Engine**:
  In April 2016, Facebook released Facebook Bot Engine based on Wit.ai technology. Wit.ai runs from its own server in the cloud, the Bot Engine is a wrapper built to deploy the bots in Facebook Messenger platform. The Facebook Bot Engine relies on Machine Learning. The Bot Framework is trained on sample conversations and can handle many different variations of the same questions. The potential is quite big as developers could improve their chatbots over time. Wit.ai offers certain options:
  - It can extract certain predefined entities like time, date, etc.
  - Extract user's intent
  - It extracts the sentiments
  - It defines and extracts own entities.

- **Microsoft Bot Framework**:
  Just like Wit.ai, Microsoft's SDK (Software Development Kit) can be viewed as 2 components, which are independent of each other.
  1. Bot Connector, the integration Framework
  2. LUIS.ai, the natural language understanding component
  The Bot Connector is highly impressive. It can be integrated with Slack, Facebook Messenger, Telegram, Webchat, GroupMe, SMS, email and Skype. There is additionally a PaaS option on Azure, just for Bots. The Microsoft Bot Development Framework consists of a number of other components as well like the Bot Builder SDK, Developer Portal and Bot Directory.

- **API.ai/Dialogflow**:
  API.ai (Dialogflow) is another web-based bot development framework. It provides a huge set of domains of intents and entities. Some of the SDKs and libraries that API.ai provides for bot development are Android, iOS, Webkit HTML5, JavaScript, Node.js, Python, etc. The concepts built on API.ai is built on the following concepts:

1. Agents: Agents corresponds to applications. Once trained and tested, an agent can be integrated with an app or device.
2. Entities: Entities represent concepts/objects, often specific to a domain, as a way of mapping NLP (Natural Language Processing) phrases to approved phrases that catch their meaning.
3. Intents: Intents represents a mapping between what a user says and what action should the software take.
4. Actions: Actions correspond to the steps the application will take when specific intents are triggered by user inputs.
5. Contexts: Contexts are strings that represent the current context of the user expression. This is useful for differentiating phrases which might be ambiguous and have different meaning depending on what was spoken previously.
API.ai can be integrated with many popular messaging, IoT and virtual assistant's platforms. Some of them are Actions on Google, Slack, Facebook Messenger, Skype, Kik, Line, Telegram, Amazon Alexa, Twilio SMS, Twitter, etc.

A simple problem that arises using commercial frameworks are that they are not free of charge and they save the data in their cloud. Hence, for business models this can not prove to be a lucrative solution. Therefore, we used an open source framework for this purpose that provided a basic skeleton for NLU. It also provides the flexibility to modify the pipeline and deploy the entire chatbot from personal servers. We found Rasa Stack to be the most promising Python framework that also provids an additional Dialog Management framework.
Rasa Stack is an open-source bot building framework. It comprises of two modules: Rasa NLU and Rasa Core, both of which can be used separately.
Rasa NLU is responsible for the Natural Language processing. It recognizes intent and entities from user's input data based on previous training data and returns them in a structured format. It combines different annotators from the spaCy parser to interpret the input data .

- Intent classification: Interpreting meaning based on predefined intents (Example: Please send the confirmation to amy@example.com is a `"provide_email"` intent with 93% confidence
- Entity extraction: Recognizing structured data (Example: amy@example.com is an `"email"`)

Rasa Core takes structured input in the form of intents and entities (output of Rasa NLU or any other intent classification tool), and chooses which action the bot should take using a probabilistic model (LSTM NN implemented in Keras). It also takes into account the conversation history and training data. It assigns a probability to all the possible actions and performs the one with highest probability.[7, 8] The Rasa Core Pipeline is explained in more details in the following sections.

2.2 Databases

For a collection of drinks and food items, a database with relevant groupings was needed. We looked into WordNet [9], which is one of the largest lexical databases for English words. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations.[9] A comprehensive list of drinks was not available in the database, hence for training purposes, the list of drinks was adopted from the database of Dialogflow. However, the powerful relations of these synsets was utilized in a later module (see section 4.3).

3 Data Generation

As a result of our research, we choose to work with the Rasa NLP framework which is based on a machine learning model. In order to train the model, we need a data set of potential sentences which customers would use in a conversation with our ACE. Sentences of this data set not only need to be structured realistically but they also need to include entities such as drinks, food, amount, temperature, size etc. of an order. Since the client of our project partner has no ACE in place yet, there is no real customer data. Hence, we need to create our own data set on possible sentences of a conversation. To create this data efficiently, reduce bias in formulations of sentences and have the flexibility to increase or decrease the data set efficiently, we decide to write a data generation script.
At a later stage, we extend the script in order to incorporate noise. Noise are parts of the input sentences of an ACE which do not belong to the sentences and can for example appear if the ACE misunderstands words or picks up parts from other, unrelated conversations.
Furthermore, the data generation script takes the likelihood of combinations of entities into account. All these features aim to make the generated data set as realistic as possible.

3.1 Approach

The general idea of our data generation approach is to randomly combine gathered raw data to sentences which customers would actually use while communicating with the ACE. This raw data can be split into two groups: Entities and bare-bone sentences.
Entities are characteristic values of an input sentence of a customer. In order to generate realistic data we include eight entities: DRINKS, FOOD, AMOUNT, SIZE, LOCATION, TEMPERATURE, TOPPINGS and SUGAR AMOUNT. Each of these entities has multiple values, such as over 800 standard soft and alcoholic drinks in DRINKS, almost 300 international dishes in FOOD, and roughly 50 different locations of a typical hotel / cruise ship in LOCATIONS.

These values of entities were partially brainstormed and added manually or extracted from lists of drinks and food online.
Since natural language is more elaborate than only listing values, we also need to create sentences which can be filled with such values. Hence, we create bare-bone sentences which have slots to be filled with entity values. Each bare-bone sentence is assigned to one intent, which is the basic message of the sentence. In total we generate three major intents: ORDER DRINKS, ORDER FOOD and CHANGE ORDER. Let us have a look into the ORDER DRINKS intent to illustrate how the bare-bone sentences are created. First, we gather standard phrases of how to order a drink, again by brainstorming and online research. One example is "I would like to have". We can now either add the slot which is to be filled directly and obtain "I would like to have a ENTITY". Alternatively, we can add additional parts to the bare-bone sentence such as "please" or other fillers which do not change the intent of the sentence. We then obtain "I would like to have a ENTITY please".
Now that we created our raw data we consider the actual data generation. The script first loads all entity values and bare-bone sentences. For each slot of a bare-bone sentence which can be filled with entity values we create a list of values that fit into the slot. Taking the previous example sentence of the ORDER DRINKS intent "I would like to have a ENTITY please", we can find two slots which can be filled: "a" and "ENTITY". The first slot "a" can be replaced by any number or other entity values which refer to the amount of an order such as "a" itself, "a round of" or "a dozen". The second slot "ENTITY" can be substituted by a number of entity values. Besides the specific DRINK, additional entities such as SIZE, TEMPERATURE, LOCATION, TOPPINGS and SUGAR AMOUNT can be filled in at the second slot. This substitution happens in an arbitrary and nested way. First an arbitrary sub-list of the list of additional entities is generated. Let us assume the arbitrary sub-list contains SIZE, LOCATION and TOPPINGS. The nested substitution starts with SIZE and replaced the slot "ENTITY" with "SIZE ENTITY". Afterwards the word "ENTITY" is again replaced by "ENTITY to LOCATION". And finally, the word "ENTITY" is replaced by "ENTITY with TOPPING". At a last step, all we fill in real values for all variables ENTITY (e.g. coke), SIZE (e.g. large), TOPPING (e.g. lemon) and LOCATION (e.g. my room). Again taking the bare-bone sentence we used before, we then obtain "I would like to have two large cokes with lemon to my room please". In this way, we can generate as many sentences as we need for each of the three major intents.
Besides the three major intents, there are also sentences of minor intents such as RECOMMEND, GOOD, BAD, HOW_ARE_YOU, THANKS, HELLO, GOODBYE, CONFIRM_POSITIVE, CONFIRM_NEGATIVE and CANCEL ORDER. In contrast to the three major intents, the minor intents usually either have all slots already filled because they are taken from other ACE databases or the sentences are of such simple structure that they contain no slots (such as "Cancel my order please"). Hence, the data generation script simply adds these sentences to the training data for the machine learning model which is the outcome of the data generation script.

Additionally, Rasa provides sample sentences on different intents which are often used for ACEs such as BOOK_RESTAURANT, PLAY_MUSIC or GET_WEATHER. These sentences are also added to our training data. Without these additional intents, we could not tell if the intent detection of our machine-learning model actually works properly, or if the right intent of an input was only detected because there are simply no other intents to choose from. So far we only considered the case that our ACE fully understands every part of the input sentence given by the customer. Since this is not always the case in reality due to multiple reasons, we also add noise to the sentences of the training data.

3.2 Noisy Data

Making a machine understanding natural language from speech is very prone to errors. Although technology made great progress in understanding natural language from speech over the last years, there are still many factors which can lead to partially incorrect understanding such as background noise, accents and dialects or simply lacking knowledge of how to pronounce certain words such as foreign dishes.
Since we want to develop an ACE which is also robust in terms of noise, we want to incorporate noise to our data generation script such that some of the sentences of the training data contain noise. Our first approach is to add a random string of characters of random length somewhere between two words of a sentence. Since not all sentences contain noise, we decide on adding one random string of characters to 10% of the sentences. Such a sentence could look like this: "I would like to sfhsaf have two large cokes with lemon to my room please".
However, our first approach ignores that nowadays speech-to-text software usually transforms the part which was not understood correctly in actual words which are similar to the noise. For this reason, we adapted the part of the data generation script that adds the noise. Instead of a random string of characters, we now pick an actual but random word from a large database of words which is called corpus, a package from the nltk library, a popular python library for natural language processing. Now the same noisy sentence from above looks like the following: "I would like to house have two large cokes with lemon to my room please".
Adding noise to our generated sentences leads to more realistic training data. Another aspect of realistic training data is the combination of entities, which will be discussed in the next section.

3.3 Correlated Data

So far we combined entities like DRINK and TEMPERATURE randomly, which might leads to unrealistic training data such as "I would like to order a hot coke". Since we generate the training data ourselves, our goal is to generate it as realistic as possible. Hence, we introduce interrelations between entities which

can be combined such as DRINKS and TOPPINGS, DRINKS and TEMPERATURE, DRINKS and SIZE and DRINKS and DRINKS. These interrelations indicate how likely two values are combined in reality. The two values "cold" and "coke" are more likely to be combined than "hot" and "coke" and therefore obtain a higher value. These interrelations are stored in a table for each possible combination of entities. If a bare-bone sentence contains such entities, such as "I would like to order a TEMPERATURE ENTITY please", the slots are filled according to the probability of the DRINK-TEMPERATURE table.
These tables do not only result in more realistic training data, they also help suggesting recommendations. If for example an espresso is ordered, the ACE then knows that the espresso often ordered with a glass of water and can ask the customer. We cover the topic of recommendations and the tables of combined entities in more detail in chapter 6.

3.4 Generating Training and Test Data

To be able to measure the performance of our data, we need to train our model with training data which we do not use for performance testing. Hence, we adapt the data generation script such that it outputs training data and test data which have no overlapping bare-bone sentences and entities. In order to do that, each table of entities and bare-bone sentences is shuffled such that we do not only train on very similar bare-bone sentences or entities. Afterwards 80% of each table is used to generate the training data. With the remaining 20% the test data is generated. In this way, we can ensure that no bare-bone sentence or entity of the test data has been trained on the model.

4 Natural Language Understanding

Natural language processing consists of two parts: Natural language understanding (NLU) and natural language generation (NLG). We have to understand what customers are communicating to the ACE (i.e. NLU) based on which we take actions and generate output to communicate back to the customer (i.e. NLG). Or we could also ask further questions to the customer if clarification is required. This chapter covers the first part of the NLP pipeline: NLU.
Rasa offers different pipelines for NLU. These pipelines determine the order and type of actions executed in order to understand the language input properly. Since Rasa is an open source framework, one can make changes to the pipeline, for example by adding steps to it.
In the first section of this chapter we present the pre-defined pipeline we chose from Rasa. Since we do not want to restrict customers of this ACE to follow specific guidelines of how to interact with the ACE, we had to adapt the pipeline partially. The adaption is discussed in the second and third section of this chapter. In the section of "Stanford CoreNLP", we deal with language input that contains multiple intents which Rasa's NLU pipelines cannot deal with by default. It also provides a naive approach to extract entities as a form

of 'double-checking' the output from Rasa's pipeline. In the third section, we introduce synonym detection which makes input by customers more machine-readable. This allows customers to give natural input and enables the ACE to assign entities of this input to known values.

4.1 Rasa NLU Pipeline

Rasa itself recommends two different pre-defined pipelines to use: spaCy and Tensorflow. The main difference between the pipelines is the use of pre-trained word vectors. spaCy uses pre-trained word vectors to analyze the input. In order to understand what the advantage of spaCy is, let us assume that our training data contains wine as a drink, but not champagne. If the customer's input is "I would like to order champagne", spaCy recognizes that wine and champagne are similar words (since both are drinks made from grapes). This allows Rasa to increase the confidence that champagne is a drink (and not a dish) which then results in detecting the intent ORDER DRINKS.[7]
On the other hand, Tensorflow does not use any pre-trained word vectors but the word vectors are trained by our own training data. An example which shows the advantage of Tensorflow are the words "balance" and "cash". Usually the word "balance" is closely related to words like "symmetry", but not to cash. However, having the specific use case of accounting in mind, "balance" and "cash" are definitely related. If our goal is to differentiate financial reports from other reports and our training data uses the words "balance" and "cash" often together, the Tensorflow pipeline would increase the confidence of Rasa detecting intents of sentences of financial reports which contain the word "balance" but not "cash". [7]
As a rule of thumb, Rasa suggests to use the spaCy pipeline if the training data contains less than 1000 labelled examples for each intent and Tensorflow otherwise. We decided to use the spaCy pipeline for this project. Although we could create as much training data as we needed with our data generation script, there was a limited number of bare-bone sentences for each intent which, usually between 100 and 500. This specifically applies to minor intents which have no slot to fill and hence no variation in the sentences. Hence, using the Tensorflow pipeline would result in difficulties understanding the intent of a sentence with a different wording than in the training data. Therefore, the spaCy pipeline is the better pipeline to choose, since expressions which are similar to our training data are more likely to be identified as the correct intent.
The pre-defined spaCy pipeline consists of the following parts:

- nlp_spaCy
- tokenizer_spaCy
- intent_entity_featurizer_regex
- intent_featurizer_spaCy
- ner_crf

- ner_synonyms
- intent_classifier_sklearn

Throughout this section we would also discuss how they were customized in the pipeline. But for now we have a look at what each component is doing.

4.1.1 NLP spaCy

This component is the language initializer of spaCy. Hence, all following spaCy components build on this part which is the reason why it is the first component of the pipeline. NLP spaCy initializes spaCy structures on the input, such as loading the language model (e.g. english) or deciding on case sensitivity.

4.1.2 Tokenizer spaCy

The tokenizer groups character strings (i.e. words) to tokens. These tokens are used later used to vectorize words and extract entities.

4.1.3 Intent Entity Featurizer Regex

Regex stands for regular expressions. During the training of the machine learning model, this component creates a list of regular expressions in the training data. For a given input, the component then compares it to all regular expressions for similarity. This information will then later be fed into the intent and entity classification.

4.1.4 Intent Featurizer spaCy

This spaCy featurizer serves as input for the last component of the pipeline: Intent Classifier Sklearn.

4.1.5 NER CRF

NER CRF is short for "named entity recognition - conditional random fields". The method of CRF is used to extract entities of the input. By analyzing the position and features of words (such as capitalization), probabilities on entities are calculated.

4.1.6 NER Synonyms

This component maps extracted entities to known synonyms. This step simplifies further actions in the NLG part later. Examples for synonyms can be same words with different capitalization (if not removed earlier) or abbreviations such as "NYC" and "New York City".

4.1.7 Intent Classifier Sklearn

The intent classifier uses outputs of previous components in order to classify the intent of an input with a support vector machine. Additionally, we do not only get the confidence of the most likely intent but of all trained intents.

This pre-defined pipeline already equips us with well working intent and entity extractors, which is the basis of our NLU model. However, for some cases of input we need to take additional actions. Since Rasa is an open source framework we can customize such pipelines by adding components to it.

4.2 Stanford CoreNLP

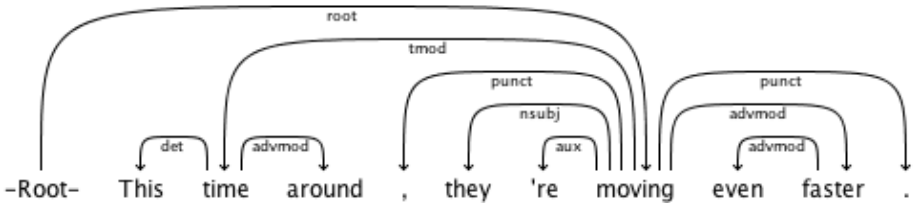
4.2.1 Motivation

As introduced before, we use the Dependency Parsing annotator from the Stanford CoreNLP package in the final pipeline. The main motivation behind this idea is to double check the main entities extracted from Rasa NLU. In a later version of the ACE model, this could be combined with Rasa NLU output to increase the confidence interval of entities. Moreover, the approach provides an elementary way to split sentences that can help to identify multiple intents and their corresponding entities. Another possible way to utilize this parser is to make it a 'backup'. This implies that in case Rasa is not able to identify intent or entities, the solution from Stanford Parser could be directly used to place an order.

4.2.2 Approach

A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between "head" words and words which modify those heads. The figure below shows a dependency parse of a short sentence. The arrow from the word `moving` to the word `faster` indicates that `faster` modifies `moving`, and the label `advmod` assigned to the arrow describes the exact nature of the dependency. The tag `advmod` denotes an adverb. The parser is powered by a

Figure 2: Dependency Parser Example [10]



neural network which accepts word embedding inputs. It is trained using adaptive gradient descent (AdaGrad) with hidden unit dropout. [10] The parser is used to exploit the grammatical structure of the input sentences and extract

the main entity (like DRINK) with all the words that 'modify' the entity (like TOPPING, AMOUNT or SIZE). It is based on the assumption that the main entity will either be the object in a sentence or the subject. For example:
"I want to order a coffee" → (coffee is the object)
"A coffee for drinking will be great" → (coffee is the subject)
The function that was created based on the dependency parser first checks for a direct object in the sentence and saves it as the main entity. It then searches for all the words that 'modify' the main entity and saves them as first-trier entities. It then also looks for the words that modify the first-trier entities. This was done as it was noticed that most frequently stopping words (like 'a' or 'the') were the direct ones connected to the main entity. If it can not find a direct object, it picks up the subject of the sentence as the main entity and follows the same procedure. However, if neither an object nor subject is found, it simply returns the input text as output. Moreover, we added a naive approach to split sentences with multiple intents or multiple main entities. We decided to split the sentence at 'and' and pass each part to the above function with the Dependency Parser. Following are the outputs for a few example sentences.

- "I want two coffees and new towels please" → ['two', 'coffees'] ['new', 'towels']
- "Get me a cold lemon ice tea in glass for afternoon" → ['a', 'cold', 'lemon', 'ice', 'tea', 'in', 'glass']
- "I want to order bananas and chocolate milk and get them soon" → ['bananas'], ['chocolate', 'milk'], ['them']
- "A cafe creme will work best for today" → ['A', 'cafe', 'creme']
- "A cafe creme and fresh orange juice will work best for today" → ['A', 'cafe', 'creme'] , ['fresh', 'orange', 'juice']
- "A cafe creme will work best for today and my wife wants a cappuccino" → ['A', 'cafe', 'creme'], ['a', 'cappuccino']

4.3 Synonyms

4.3.1 Motivation

As was previously established, our NLU model parses a sentence by classifying its intent and extracting entities. One of the goals of this project was to provide the resort with a way to optimize their processes by analyzing the data collected through this automatic parsing. This requires that the output of the NLU model be machine-readable to facilitate such analysis.
In practice this means that every intent and entity value should have a real counterpart in the resort, e.g. for a sentence expressing the "order drink" intent containing a "drink" entity with the value "cappuccino", there should be a real mechanism in place, which fulfills the function of ordering a drink as well as an

item corresponding to "cappuccino" to be delivered to the customer. Defining these counterparts for all possible intents is trivial, since their number is finite and expected to be small. However, there is an unknown and potentially large number of names referring to the same item. In other words, by identifying a drink in a sentence, we do not always know what particular item in the resort's range of drinks it corresponds to.
In order to avoid the need to manually map every possible entity value to its counterpart in the resort, the NLU model should ideally only output one possible entity value for each distinct item. We approach this problem by attempting to automatically identify sets of synonyms or "synsets" across all observed entity values. It is then assumed that each of these synsets only corresponds to a single item in the resort.

4.3.2 Approach

WordNet [9] is one of the largest resources for synonyms. Each synset contained in WordNet expresses a distinct concept, however, one word or phrase can be part of multiple synsets, since it may express different concepts. For example the word "second" in the sentence "i will be there in a second" has the meaning "a short time frame", whereas in the sentence "i will be there in 60 seconds" refers to an actual precise measurement of time. The task of identifying the correct concept of a word or phrase in a given context is called word sense disambiguation (WSD).
Multiple algorithms have been proposed for this task including dictionary based, graph based, supervised and similarity based methods [11]. Popular algorithms include Lesk algorithm [12], maximizing path similarity [13] and maximum entropy [14]. Since we do not have a ground truth for word sense in our example sentences, supervised methods do not apply to our problem. We also need the WSD algorithm to be computationally inexpensive, since typically there are a lot of example sentences to consider at training time and the prediction needs to be fast at inference time. This, in our experience, excludes graph based and similarity based methods.
Adapted Lesk [12] is a version of the dictionary based Lesk alorithm, which has been adapted to the dictionary definitions and synsets of wordnet. We use this algorithm to predict a synset for each of the entity values in all training sentences. To minimize erroneously assigning synsets, the final prediction for each entity value is the synset which was most often predicted across all sentences containing this value. Since we expect the word senses to be disjoint across entities, we consider two entity values of different entities to be different entity values, even if they are the same string.
Entity values are then considered synonyms if they are members of the same entity and synset. The output of the NLU model for an entity value is the lexicographically smallest member of its synset. This selection is arbitrary, though it ensures that the output for all entity values of a synset is always the same. If at inference time, an unseen entity value is detected, the synset can be found by applying adapted Lesk on the newly observed sentence. We implemented

our approach for synonym detection as a custom component of the rasa nlu framework, so that it may be trained end to end.

4.3.3 Results

Across 25584 example sentences containing 14189 distinct entity values our approach detected 78 synsets, which contain more than one entity value. As a result, 80 entity values could be replaced by their synonym. This relatively low number of detected synonyms might be attributed to the fact that this data was artificially generated, which excludes a lot of grammatical as well as dialect and cultural variation that would be found in natural language data.
Of the 80 detected synonyms, 2 were found to be falsely detected. The algorithm equated the genres "punk" and "punk rock" as well as the drinks "arak" and "arrack". This can be attributed to mistakes in WordNet's definitions, since these two instances are actually common confusions.
Among the correctly identified synonyms 72 are abbreviations, such as the initials referring to the individual states of the USA or the numeral representations of written out numbers, as well as grammatical conjugations such as plurals. This alone justifies the use of our approach, since the need of any additional corpora is eliminated.

5 Recommender System

5.1 Motivation

The advent of big data has spawned a trend of adapting interactive systems to users' preferences. This can partly be attributed to the availability of user specific data, as well as the need of the customer to filter through an overwhelming range of products. Examples include Netflix and Amazon attempting to filter for movies or products, which the customer is most likely to interact with.
This personalization is particularly relevant to a hotel resort, since the goal is for the customer to feel as comfortable as possible in this environment. Hence, it is necessary to tailor our ACE system to the customer's individual preferences. One approach to this problem is to estimate the customers preferences to all relevant items, based on his past interactions and other users' preferences. This approach corresponds to a recommender system. In our framework, entity values represent all items which are available to the customer through the resort. Thus, interactions of the customer with the ACE system provide a data source for recommendations.

5.2 Approach

A resort is a constantly changing environment. Since the needs of the resort may differ depending on the situation, a recommender system should be easily customizable. Additionally, customers might stay only a short while and only interact very little with the system. Thus, one wants to utilize all available

information about the customer at any time. This includes previous knowledge about the customer such as gender, age, and nationality as well as situational information such as the time of the interaction. Recommender systems which include such information are called context-aware recommender systems [15]. Since, we do not have rating data but only interaction data, we are interested in the probability that a customer will select a given item in a given context $P(item|context)$. Viewing X as a random variable over all items of a given entity, one obtains a probability distribution of X, with $\sum_{items} P(X = item|context) = 1$. The context can take the form of any prior information and can consist of multiple different contexts, such as age, nationality and gender with $context = context1, context2, context3, ...$ and thus $P(item|context1, context2, context3, ...)$. There exist three different paradigms for context-aware recommender systems [15].
Contextual modeling describes the attempt to model $P(item|context)$ directly. The context however can be high-dimensional resulting in sparse data. Additionally, when incorporating new contexts into the model, the whole model has to be retrained.
Contextual pre-filtering is the approach of dividing the data by its context and modeling each context separately. However the same problem in contextual modeling of sparsity arises.
In contextual post-filtering the probability $P(item)$ is modeled independently of context and post-modified according to the context. We use this paradigm by modeling $P(item)$ as well as $P(item|context_1)$, $P(item|context_2, ...)$ and then combining $P(item)$, $P(item|context_1)$, $P(item|context_2), ...$ in a weighted sum:

$$P(item|context) = a * P(item) + \sum_{i=1}^{\#contexts} b_i * P(item|context_i)$$

with $a, b \in \mathbb{R}_+$ and $a + \sum_{i=1}^{\#contexts} b_i = 1$.
$P(item)$ can be post-filtered through other methods than a weighted sum, though this approach allows to customize how much each context weighs into the final recommendation.
Usually $P(item)$ implicitly refers to the probability of a customer interacting with an item, given his past interaction. We choose to view the past interactions as yet another context and model $P(item)$ by simply considering the relative frequency of the item in all interactions.

$$P(item) = \frac{\text{\# of interaction with item}}{\text{\# of interactions with all items}}$$

The reasoning behind viewing a customer's past interactions as a context is given in section 5.3. The following subsections explain the different contexts and the approaches of modeling $P(item|context_i)$.

5.2.1 Meta-data as context

With meta-data, we refer to any context, which can take finitely many discrete values. Such as gender with values male and female. Continuous contexts

can be transformed into meta-data by binning the values. For example, in the context of time, the values get binned into morning, daytime and evening. One obtains the data matrix $M \in \mathbb{N}^{c \times e}$ for a context C with c values and an entity E with e values. The entries of the matrix are the number of observed interactions containing the specific entity value and meta-data value. We obtain the probability distribution for $P(E|C)$ by dividing the columns of M by their sum and then dividing the rows by their sum. Normalizing the columns in this way has the effect of calculating the relative frequency of all contexts for all entity values. Normalizing the rows transforms each row into a probability distribution over the entity conditioned by a given context. Hence:

$$P(E = entity_i | C = context_j) = M_{i,j}, i \in [1, e], j \in [1, c]$$

Since we were not provided with any data, we have no means of validating this approach with a validation data set. However, we regard our the method of using relative frequency as simple enough as not to require further validation.
In order to generate a first rough estimate of recommendations for different meta-data, we manually curated lists of drinks relevant to different values of the contexts time, age and nutritional value. We then generated context specific data with our approach for data generation and used our algorithm to recover the probability distributions.

5.2.2 Entities as context

Entities can also be considered as context. This can be interpreted as the correlation of different entity values. We distinguish between two cases.
In the first case, the context entity is different from the recommended entity. Here we utilize the same method as for a meta-data context. The entries of M are the number of interactions, in which the two entity values occur together.
In the second case, the context entity is the same as as the recommended entity. This is equivalent to a common task of collaborative filtering called item-item recommendation. For this, we use an algorithm of the popular slope one family. G. Linden et. al [16] suggested the approach of computing the cosine similarity between all column vectors in the customer-item matrix. This matrix contains the amount of interactions of each customer with each item.
As in section 5.2.1, we regard this algorithm as simple enough as not to require further validation. However, since we do not have a way of generating data with meaningfully correlated entities, we were not able to use the algorithms as a first estimate for entity contexts.
We instead used the popular web search Bing [17] to obtain numbers of web documents in which the two entity values occur together. Here we make the assumption that this roughly correlates with the number of interactions they would typically occur together. In order to justify this assumption, we performed dimensionality reduction with T-SNE algorithm [18] so that the obtained similarities can be visualized and evaluated. In this visualization, we observe clusters of a significant number of items, which can be regarded as similar. Figure 3 shows two of those clusters for drink-drink similarity.

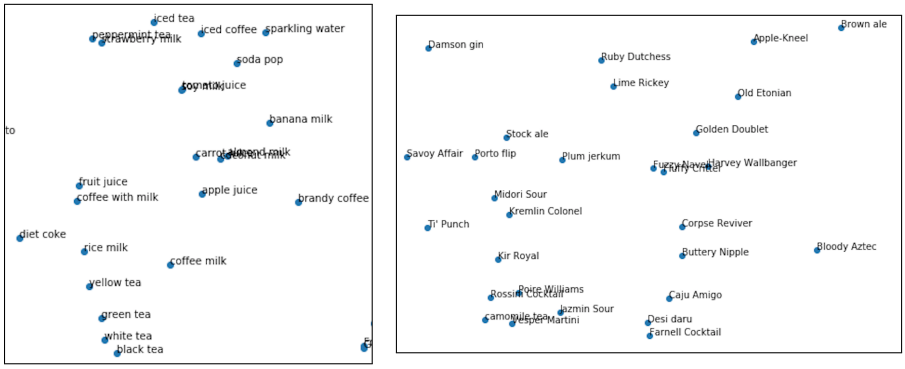


Figure 3: Clusters of drinks observed in a T-SNE embedding of the drink-drink similarity matrix estimated with web search. Figure a) shows a cluster of alcoholic beverages, while figure b) shows a cluster of non-alcoholic beverages.

We conclude that our method of estimating entity-similarity through web search does indeed produce meaningful results.

5.2.3 Past interactions as context

When conditioning the probability of picking an item on all past interactions, one hopes to estimate a specific customer's preference by learning from other customers' preferences. This is a known problem in collaborative filtering called user-user similarity. One of the most popular ways of approaching this problem is to factorize the customer-item-matrix. This represents customers as well as items in a latent space, where a customer's preference for an item is simply the product of his and the item's latent vector.
However, our data does not contain explicit preferences like ratings, but only implicit preference in the form of interactions. Conventional matrix factorization methods are not applicable to implicit data, since it does not contain any negative feedback. Y. Hu et al. [19] proposed a method for factorizing implicit feedback data by weighting each observation by a factor of confidence. We use this weighted matrix factorization to estimate the probabilities $P(item|customer, interactions)$.
To obtain a first estimate of customer interactions, we generated data with pseudo preferences. To obtain these preferences, we make the assumption that one customers preference consists of multiple clusters of similar drinks. We use our drink-drink similarities estimated by web search to select such clusters and probabilistically combine them into 400 pseudo customers. After fitting the model to this data, we observed that it reliably recovers the entire clusters from just a few items that are in those clusters.

5.3 System integration

All calculated recommendations are saved to a database and can be retrieved at runtime. The final recommendation for a set of context are obtained by calculating their weighted sum. Which contexts are relevant and how they are weighted depends on the situation and the resorts' needs.
Including the customer's past orderings as a context when only few orderings have been made is to be be avoided, due to the cold start problem. Relying on more general contexts will yield more general, though more robust recommendations. As a baseline, all available contexts are used and weighted equally.
In order to update the recommendations, we use two approaches. The first is to simply recalculate all recommendations including the newly observed data. This however, is too slow to apply after each new data point. Since the system should respond quickly to new data, the recommendations for an observed entity are simply increased by a fixed value. Higher increase values result in more recent orders being weighted more heavily.
Once the recommendations are obtained, they can be used to personalize the system in different ways. The most straightforward way is to allow the customer to ask for recommendations for a specific entity, or to automatically make recommendations once a sufficiently high confidence is reached. Additionally, if the customer asks e.g. 'What items are available?', only the most likely recommendations are given instead of an exhaustive list.
The recommendations can also be used to improve the performance of the ACE itself. If there exists a mandatory entity for a given intent, one can fill this slot with the most likely recommendation if the confidence is high enough, instead of the default value. Additionally, when the NLU model gives ambiguous predictions, one can select the most likely recommendation.

6 Dialog Management

6.1 Motivation

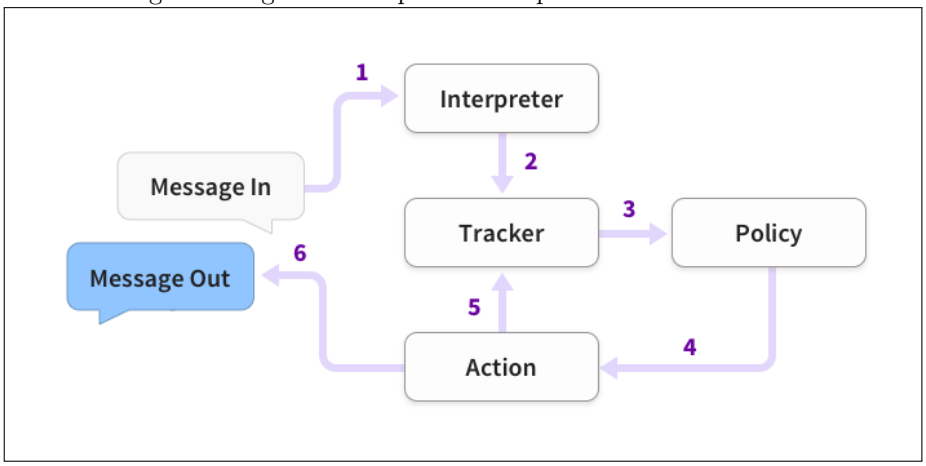
To consolidate outputs of all the components (RASA NLU, Customer History, Recommendations) mentioned earlier, and to curate a conversation between the user and the bot, a dialog management framework was required.
It was required to opt for a framework that is open source, keeps the context of the conversation and can be developed with zero or less training data.
Considering the above-mentioned constraints, we decided to go forward with RASA Core for the development of dialogue management component.

6.2 RASA Core

Rasa Core is an open source machine learning framework used for dialog management in the development of conversational softwares like chatbots.

6.2.1 High-Level Architecture

Following diagram gives an overview of high-level architecture of Rasa Core and how it manages messages as an input and output.



1. The input message is received and passed to an Interpreter, which converts it into a dictionary including the original text, the intent, and any entities that were found. RASA Core cannot interpret the input messages from user itself. Therefore, it requires a NLU model to parse the input. In our project, Interpreter is the RASA NLU model which parses the input message and identifies the intents and entities.

2. The Tracker is the object which keeps track of conversation state with a single user. It receives the information that a new message has come in. The tracker stores and maintains the state of the dialogue with a single user. It is stored in a tracker store, retrieved when incoming messages for the conversation are received and updated after actions have been executed. In our project, Dialog State Tracker is InMemoryTrackerStore, which is the default tracker store. It is used if no other tracker store is configured and stores the conversation history in memory.

3. The policy receives the current state of the tracker.

4. The policy decides about the next action. A Policy decides what action to take at every step in a dialog. In our project, following four policies were configured.

Policy	***Objective***
Keras Policy	*To configure Keras machine learning library to train the model*
Fallback Policy	*To configure fallback action*
Memorization Policy	*To configure how much dialogue history the model looks at to decide which action to take next*
Form Policy	*To activate/deactivate Forms*

5. The chosen action is logged by the tracker.

6. A response is sent to the user.

The entire process is handled by the Agent. Agents allows the developer to train, load, and use the model. It is a simple API that provides most accessibility to Rasa Core's functionality.

6.3 Elements of RASA Core

6.3.1 Domain

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

1. Intents: As mentioned earlier in the documents, Intents are the things expected the user would say during the conversation. These Intents are identified by RASA NLU Model. All intents that can be identified should be included in the Domain file. A list of intents curated for ACE project can be found in RASA NLU section.

2. Entities: Entities are described as the pieces of information extracted from the messages. Like Intents, all entities are extracted by the RASA NLU Model and should be mentioned in the Domain file of RASA Core. A list of entities curated for ACE project can be found in RASA NLU section.

3. Slots: Any information that is needed to be tracked of during a conversation can be stored as Slots in key-value format. Slots can be defined as the bot's memory. The value of slots can influence the flow of dialog. (More information on this in Stories Section)

Slot Name	Slot Type	Objective
*DRINK***	*Unfeaturized**	*To keep track of DRINK ordered*
*AMOUNT****	*Unfeaturized*	*To keep track of AMOUNT of the entity (DRINK) ordered*
*LOCATION****	*Unfeaturized*	*To keep track of LOCATION for the order delivery*
*SIZE****	*Unfeaturized*	*To keep track of SIZE of the order*
*TEMPERATURE****	*Unfeaturized*	*To keep track of TEMPERATURE of the order*
*TOPPING****	*Unfeaturized*	*To keep track of TOPPING with the order*
time	*Unfeaturized*	*To keep track of TIME of each message*
time_diff	*Unfeaturized*	*To keep track of the TIME DIFFERENCE between two messages*
main_entity	*Boolean*	*To keep track if the main entity** is present(true)/absent(false)*
previous_intent	*Unfeaturized*	*To keep track of the previous intent of the user where the respective main entity was present in the message*

* Unfeautrized: Data you want to store which shouldn't influence the dialogue flow
* Main Entity: For each intent, there is a Main Entity defined and stored in the database. This Main Entity is required for that Intent. For example, order_drink intent must have DRINK as entity in the message.

Slot Filling :
The next step after defining the slots, is to set how and from where each of these slots will take value.
a. Slot Filling via Form Action
It is recommended to create a Form Action for slot filling, if multiple pieces of information are required to be collected in a row. It is a single action which contains the logic to loop over the required slots and ask the user for this information. To add forms to the domain file, it is needed to reference their name under forms: section in the domain file. In our project, we used Form Action to fill the slots*** that are required to order drink. More details on it will be explained in the Actions section later.
b. Slot Filling via SlotSet Event
Every slot has a name and a value. The SlotSet event can be used to set a value for a slot on a conversation. In our project, the remaining slots were set in various actions using SlotSet event. More details on it will be explained in the Actions section later.

4. Templates: Utterance templates are messages the bot will send back to the user. There are two ways to use these templates:
If the name of the template starts with utter_, the utterance can directly be used like an action. It is required to add the utterance template to the domain. In our project, utter_templates for all the utter_actions mentioned

above are added to the domain file
Templates can be used to generate response messages from custom actions using the dispatcher e.g. dispatcher.utter_template("utter_greet"). This allows to separate the logic of generating the messages from the actual copy. Adding templates to the domain file is optional.

5. Actions: Actions are defined as things bot say or do in response to the input. All actions should be included in the Domain file under action: section. There are three kinds of actions:

Default Actions: Some default actions defined in RASA Core are as follows:

Action Name	***Objective***
action_listen	*Stops predicting more actions and waits for the user input*
action_restart	*Resets the whole conversation. It is triggered by using /restart*
action_default_fallback	*Executed if the intent recognition has a confidence below* ***nlu_threshold*** *or if none of the dialogue policies predict an action with confidence higher than* ***core_threshold****. Undoes the last user message as it was never sent*

For our project, we changed the default_fallback_action to our custom utter action (utter_fallback). Respective parameters were set in the Fallback Policy.

Utter Actions: These actions send messages to the user as per the templates defined in the domain file. Utter Actions always start with utter_. These actions look for text of the messages in templates defined in domain file. Templates for utter actions must start with "utter_"

For our project, the utter actions used are as follows:

Utter Action Name	Objective
utter_AddToPlaylist	Response for add to playlist intent
utter_BookRestaurant	Response for book restaurant intent
utter_GetWeather	Response for get weather intent
utter_PlayMusic	Response for play music intent
utter_RateBook	Response for rate book intent
utter_SearchCreativeWork	Response for Search Creative Work intent
utter_SearchScreeningEvent	Response for Search Screening Event intent
utter_cancel_order	Response for cancel order intent
utter_change_order	Response for change order intent
utter_order food	Response for order food intent
utter_ask_AMOUNT	Asks user for amount of order
utter_ask_DRINK	Asks user for drink of order
utter_ask_LOCATION	Asks user for location of order
utter_ask_SIZE	Asks user for size of order
utter_ask_TEMPERATURE	Asks user for the temperature of order
utter_ask_TOPPING	Asks user for the topping of order
utter_drink_submit	States the complete drink order
utter_wrong_DRINK	Notifies if invalid drink
utter_wrong_AMOUNT	Notifies if invalid amount
utter_wrong_LOCATION	Notifies if invalid location
utter_wrong_SIZE	Notifies if invalid size
utter_wrong_TEMPERATURE	Notifies if invalid temperature
utter_wrong_TOPPING	Notifies if invalid topping
utter_submit	Notifies with affirmation
utter_time	Prints out time slot
utter_fallback	Asks the user to be more specific (Fallback Action)
utter_good	Response when given good feedback from user
utter_bad	Response when given bad feedback from user
utter_how_are_you	Response when asked how the bot is doing
utter_thanks	Response when user thanks the bot
utter_hello	Response when user greets the bot
utter_goodbye	Response when user leaves the conversation with goodbye intent
utter_confirm_positive	Response for confirm_positive intent
utter_confirm_negative	Response for confirm_negative intent

Custom Actions: Custom Actions can be defined to add customized response from the bot, apart from utter and default actions. A custom action can run any code and can do anything from turning on lights, adding events in schedule, checking a user's bank balance, to anything else

Rasa Core calls an endpoint specified in endpoints.yml, when a custom action is predicted. This endpoint should be a webserver that reacts to this call, runs the code and optionally returns information to modify the dialog state.

Rasa provides rasa_core_sdk to define custom actions in python

Custom Action Name	*Objective*
Order_Drink_Form	*Form Action to collect information for drink order*
Save_Time	*To calculate time difference between two user inputs*
Coreferencing	*To keep the context of conversation*
Order_db	*To store orders in the database*
Recommendations	*To make recommendations to the user*

In our project following custom actions were defined in Action.py file. Let us investigate the custom actions in detail.

1. Order_Drink_Form:

The FormAction will only requests slots which haven't already been set. Order_drink_form has three methods:

- name: the name of the action
- required_slots: a list of slots that need to be filled for the submit method to work.
- submit: what to do at the end of the form, when all the slots have been filled.

Every time this form action gets called, it will ask the user for the next slot in required_slots which is not already set. It does this by looking for a template called utter_ask_slot_name, which need to defined in domain file for each required slot. Once all the slots are filled, the submit() method is called, where you can use the information you have collected to do something. After the submit method is called, the form is deactivated, and other policies in your Core model will be used to predict the next action. Some additional methods that defined in the Form Action are:

slot_mappings: defines how to extract slot values from user responses.The predefined functions work as follows:

- self.from_entity(entity=entity_name, intent=intent_name) will look for an entity called entity_name to fill a slot slot_name regardless of user intent if intent_name is None else only if the users intent is intent_name.
- self.from_intent(intent=intent_name, value=value) will fill slot slot_name with value if user intent is intent_name.
- self.from_text(intent=intent_name) will use the next user utterance to fill the text slot slot_name regardless of user intent if intent_name is None else only if user intent is intent_name.

To allow a combination of these, it is required to provide them as a list.
validate: After extracting a slot value from user input, the form validates the value of the slot. By default, it only checks if the requested slot was extracted. In our project, custom validation is added, which checks the value against the database.

2. Save_Time:
This action calculates the time difference between the previous and current user input. It sets the slot Time and Time_Diff

3. Coreferencing:
This action calculates and then used the context of the conversation to set some slot values since these slot values influence the flow of the dialogue. It first extracts current intents and entities from the latest message and check if the main entity of the respective intent is present or missing. If the main entity is present, it:

- sets the slot for that entity to its value
- sets the Boolean slot, main_entity, to true
- sets the slot of previous intent to the current intent as the main entity of the intent is present

. If the main entity is missing, it:

- takes the value of previous intent from the slots
- checks if the current entities detected are in the list of entities for that intent
- checks for the confidence of the value if detected as any of the entities in the list
- if the confidence is more than a certain value and the time difference is not major then it treats this sentence as one of the previous intents
- sets the slot of main_entity as false
- sets the slot previous_intent as the previous intent detected
- sets the slot of that entity detected to the value identified
- all this slot setting then changes the flow of dialogue in the story

4. Order_db:
Once the order is completed, this action stores the values of the message and slots (extracted from the tracker) into the database

5. Recommendations:
This action calls the recommendation components of the framework, giving in the context and entity you want to recommend as the parameters

6.3.2 Stories

A training example for the Rasa Core dialogue system is called a story. RASA Core learns from these example conversations provided. A story starts with a name preceded by two hashes ##story03248462. A story can be called anything, but it can be very useful for debugging to give them descriptive names.The end of a story is denoted by a newline, and then a new story starts again with ##. Messages sent by the user are shown as lines starting with * in the format intent"entity1": "value", "entity2": "value".
Actions executed by the bot are shown as lines starting with - and contain the name of the action. Events returned by an action are on lines immediately after that action. For example, if an action returns a SlotSet event, this is shown as the line - slot"slot_name": "value"
In our project, we have multiple stories for each intent. We have specially added the stories where the main entity is detected or missing. These stories can included various scenarios as per the kind of conversations expected between the user and the bot.

6.3.3 Training Dialogue Model using RASA Core

Rasa Core works by creating training data from stories and training a model on that data.

1. Using Command Line:
   Dialog model using RASA Core can be trained from the command line like:
   python -m rasa_core.train -d domain.yml -s data/stories.md -o models/current/dialogue -c config.yml

2. Using Agent:
   Dialog model using RASA Core can be trained creating an agent and running the train method:

```
from rasa_core.agent import Agent

agent = Agent()

data = agent.load_data("stories.md")

agent.train(data)
```

3. Using Interactive Learning:
   Another powerful way to train the Rasa Core is through Interactive Learning which is a form of Reinforcement Learning. Here, the model trains through continuous feedback. It learns the stories and dialog flow by interacting with the user and corrects itself in case of mistakes. Initially, a few stories have to be designed which acts like a starting point for the model. The code used is:

python -m rasa_core.train interactive –core models/dialogue –nlu models/current/nlu –endpoints endpoints.yml
In interactive mode, the bot asks user to confirm every prediction made by NLU and Core before proceeding. For example:

```
Bot loaded. Type a message and press enter (use '/stop' to exit).

? Next user input:  hello

? Is the NLU classification for 'hello' with intent 'hello' correct?  Yes

------
Chat History

 #    Bot                       You
──────────────────────────────────────────
 1    action_listen
──────────────────────────────────────────
 2                                   hello
                          intent: hello 1.00
------

? The bot wants to run 'utter_greet', correct?  (Y/n)
```

In the above scenario, on typing 'n' the bot then asks the user to select the right action from a list of possible pre-defined actions. On continuing like this, the bot stores the results and creates new stories for Rasa Core to incorporate. At the end it appends them to the stories.md file. The model has to retrain again using either the 1st or 2nd methods.

7 Results and Discussion

Now that we know how our customized NLU model works, the question of how good it works is obvious. It is the concept of machine-learning models that one tries to learn the model with input such that it then can later handle new input which might have not been seen before. To ensure that our ACE is able to deal with unknown input, we conduct performance measuring in a way that we split our generated data in training and test data. The model is then trained with the training data and we test how well the model detects intents and entities from the test data. This approach does not only indicate how well our ACE works, it also helps us to find out how we should generate the data in order to train a model which works well under real-life conditions.
In this chapter, we first explain how we generate training and test data with our data generation script. Afterwards we look into how the performance measuring works and which metrics are relevant. We then present the multiple variations of our generated data and conclude which one performs the best. The best performing model will then become our baseline model we use to run the ACE.

7.1 Models

We trained eight different models to measure the performance and choose the model with the most promising results. The models are as follows:

0. Baseline:
   The Baseline model, as the name suggests is the most basic model that we trained with the sample data that we generated. The pipeline used was the pre-defined spaCy pipeline. The breakup of training and test was 80% - 20%. Number of sentences in the training set were 2400 and test set had 600 sentences for every intent.

1. Baseline with Noise:
   This model was an extension of the Baseline model to which we added noise to make the model more robust. One random string of alphabetical characters of random length (between 1 to 10 characters) was added between any two words of the input sentences. Since not all sentences contain noise, we decided on adding one random string of characters to 10% of the sample data. Here is an example of such a sentence: "I would like to `sfhsaf` have two large cokes with lemon to my room please". Like before, the same number of samples were used for training and testing.

2. Baseline, Noise with Lookup-Table:
   We introduced a lookup-table as a list of drinks in the training data which becomes part of the spaCy pipeline. Upon loading the training data, this list is used to generate case-insensitive regex patterns that are added to the regex features. [20] On adding a list of pre-specified drinks, it becomes easier to extract the entity. This model is also an extension of the second model. It contains the baseline model with noise as well. In this model too, we used the same number of training and test data.

3. Baseline, Noise with Synonyms:
   In this model, we specified a synonym list which is also part of the training data. For example: {"value": "New York City", "synonyms": ["NYC", "nyc", "the big apple"]}. Here if the model extracts any from the list ["NYC", "nyc", "the big apple"] and maps it to the fixed value "New York City". However, this only happens after the entities have been extracted, so the training samples must contain the "synonyms" so that Rasa can learn to pick them up. [20] The synonym list was generated using the script mentioned in the Rasa NLU Pipeline

4. Baseline, Noise, Lookups and Synonyms:
   This model combined both Synonyms and a Lookup Table.

5. Baseline with One Real-Word Noise:
   Our second model ignored the impact of speech-to-text softwares. They transform every part of the noise to an actual word. So all words that were noise would be mapped to a real word similar to the noise. For this reason,

we adapted the part of the data generation script that introduced noise. Instead of a random string of characters, we picked an actual but random word from a large database of words which is called `corpus`, a package from the nltk library, a popular python library for natural language processing. Now the same noisy sentence from above looks like the following: "I would like to house have two large cokes with lemon to my room please". As before, the word is introduced randomly between any two words in the sentence and is added to only 10% of the data.

6. Baseline with Two Real-Words Noise:
   This model adds two random real words from the nltk library anywhere in the sentence.

7. Half Amount of Bare-Bone Sentences:
   As explained in the data generation script, we used a list of bare-bone sentences (e.g. I would like to order a ENTITY with TOPPING) that were combined with different entities (e.g. juice and ice) in various orders to give final sentences (e.g. I would like to order a juice with ice). The motivation behind this was to analyze performance with reduced variability in the model. More number of bare-bone sentences introduces higher richness in the model. As the training-test split till now was 80% - 20%. For this we used a 40% - 60% split. Therefore working with only half the amount of bare-bone sentences as well as list of all entities.

7.2 Evaluation

We chose Rasa NLU's extensive evaluate module for the performance evaluation. The module focuses on the three most important metrics: Precision, Recall and F1 Score. Correctly predicted observations (True Positives in binary case) is the number of observations that were predicted correct for the class. They belonged to the class and the model classified them correct. Let $k_1, k_2, ..., k_m$ be the number of correct classifications for the m classes and $r_1, r_2, ..., r_m$ be all the number of predictions made for the m classes. Also, let $n_1, n_2, ..., n_m$ be the total number of actual points belonging to the m class. Then,

- Precision: It is the ratio of correctly predicted observations of a class to the total predicted observations of the class. The precision for the model is calculated as the average Precision for all classes.

$$Precision = \frac{1}{m} \sum_{i=1}^{m} \frac{k_i}{r_i}$$

- Recall: Also called Sensitivity is the ratio of correctly predicted observations of a class to the all observations in the actual class. Recall for a model is then the average of all recall values for m classes.

$$Recall = \frac{1}{m} \sum_{i=1}^{m} \frac{k_i}{n_i}$$

- F-Score: It is a harmonic mean of Precision and Recall.

$$F\ Score = \frac{(1+\beta^2)(Precision * Recall)}{\beta^2(Precision + Recall)}$$

where β is commonly 0.5, 1, or 2. Therefore the F1 Score is with $\beta = 1$

$$F1\ Score = \frac{2(Precision * Recall)}{(Precision + Recall)}$$

It provides Precision, Recall and F1 Score for both intents and entities. The following two tables summarise the metrics for all the nine models for intent recognition and entity recognition respectively.

Figure 4: Average Performance Measure for Intent Recognition

	Models	Precision	Recall	F1-Score
0	Baseline (B)	0.98	0.98	0.98
1	B+Noise (BN)	0.98	0.98	0.98
2	BN + Lookup Table (BNL)	0.98	0.98	0.98
3	BN + Synonym (BNS)	0.98	0.98	0.98
4	BN + Lookup + Synonym (BNLS)	0.98	0.98	0.98
5	B+One Real Word Noise (BO)	0.99	0.99	0.99
6	B+Two Real Word Noise (BT)	0.99	0.99	0.99
7	Half Amount Barebone (HBB)	0.83	0.83	0.83

Figure 5: Average Performance Measure for Entity Recognition

	Models	Precision	Recall	F1-Score
0	Baseline (B)	0.96	0.96	0.96
1	B+Noise (BN)	0.96	0.96	0.96
2	BN + Lookup Table (BNL)	0.98	0.98	0.97
3	BN + Synonym (BNS)	0.98	0.98	0.98
4	BN + Lookup + Synonym (BNLS)	0.98	0.98	0.97
5	B+One Real Word Noise (BO)	0.99	0.99	0.99
6	B+Two Real Word Noise (BT)	0.99	0.99	0.99
7	Half Amount Barebone (HBB)	0.86	0.87	0.85

We can see from the summary table that all models perform extremely well when it comes to intent recognition. The Lookup tables and Synonyms only

impact the entity recognition and hence improve results for the same when compared to the Baseline models. This follows from the Rasa NLU pipeline. The intents and entities are extracted separately, independent of each other. An interesting observation is that the model with noise (BN) performed equally well as the Baseline models. Moreover the BO and BT models with real word noise performed better compared to all other models. The main reasoning behind this is the fact that the model is now more "careful" to classify words as entities or use them to identify intents. Hence, the possibility of wrong classifications reduces in these models and improves performance. The last model HBB which reduces the variability in the model has to encounter newer words in the testing environment. Hence, the miss classifications increase and the performance drops.
The Rasa Evaluate module also generates a confusion matrix for the intent classification and a histogram showing the average Confidence of extracted Intents on x-axis with the sample size of training data on the y-axis. Additionally a breakup of entities for more comprehensive in-depth analysis of metrics for all models is included in the Appendix. Following this extensive analysis of different models, we decided on using a model with the best performance which would also be sensitive to unseen data. The BO and BT models fit this criteria well. Additionally, we wanted to be more realistic with our model and avoid unnecessary complications. Hence we decided on incorporating the One-Real Word Noise with Rasa Core, the Dialog Management framework.

8 Conclusion and Future Work

The goal of this project was to develop and implement an artificial conversational entity framework for a resort environment. We implemented such a framework with the Rasa open-source natural language processing tool and expanded it to meet the needs of the specific domain. Our additions include an approach for synthesizing data, a method to automatically detect synonyms of ambiguous item names and a recommender system to learn customer preference. We designed our work in such a way that it can be easily adapted to a specific domain and that it operates on only a small amount of data.
The system can be used to automate customer interactions and can be integrated into a business's specific processes in order to automatically perform actions which are usually labor intensive. This is facilitated by our framework producing machine interpretable output, which additionally allows for computational optimization.
Our system operates on the general concepts of intents and entities and has been shown to function well in a diverse domain. This justifies that our system can be adapted to the domain of any service environment like for example that of hotels, spas or cruise ships.
Other possible extensions include connecting the system to a speech recognition model and learning the dialogue responses with reinforcement learning.

References

[1] "Natural language understanding." https://searchenterpriseai.techtarget.com/definition/natural-language-understanding-NLU. Accessed: 2018-12-01.

[2] "Parsing." https://en.wikipedia.org/wiki/Parsing. Accessed: 2018-12-01.

[3] "Parser technopedia." https://www.techopedia.com/definition/3854/parser. Accessed: 2018-12-01.

[4] "Natural language processing made easy – using spacy (in python)." https://www.analyticsvidhya.com/blog/2017/04/natural-language-processing-made-easy-using-spacy-in-python/. Accessed: 2018-12-01.

[5] "Stanford open information extraction." https://nlp.stanford.edu/software/openie.html. Accessed: 2018-12-01.

[6] "Complete guide on bot frameworks." https://www.marutitech.com/complete-guide-bot-frameworks/. Accessed: 2018-12-01.

[7] "Rasa documentation." https://rasa.com/docs/nlu/. Accessed: 2018-12-01.

[8] "Conversational ai chatbot using rasa nlu rasa core." https://medium.com/@BhashkarKunal/conversational-ai-chatbot-using-rasa-nlu-rasa-core-how-dialogue-handling-with-rasa-core-can-use-331e7024f733. Accessed: 2018-12-01.

[9] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, pp. 39–41, Nov. 1995.

[10] "Neural network dependency parser." https://nlp.stanford.edu/software/nndep.html. Accessed: 2018-12-01.

[11] R. Navigli, "Word sense disambiguation: a survey," *ACM COMPUTING SURVEYS*, vol. 41, no. 2, pp. 1–69, 2009.

[12] S. Banerjee and T. Pedersen, "An adapted lesk algorithm for word sense disambiguation using wordnet," in *International Conference on Intelligent Text Processing and Computational Linguistics*, pp. 136–145, Springer, 2002.

[13] T. Pedersen, S. Patwardhan, and J. Michelizzi, "Wordnet::similarity," pp. 38–41, 01 2004.

[14] A. L. Berger, V. J. D. Pietra, and S. A. D. Pietra, "A maximum entropy approach to natural language processing," *Comput. Linguist.*, vol. 22, pp. 39–71, Mar. 1996.

[15] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender Systems Handbook*. Berlin, Heidelberg: Springer-Verlag, 1st ed., 2010.

[16] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *Internet Computing, IEEE*, vol. 7, pp. 76–80, 01 2003.

[17] "Bing web search engine." https://www.bing.com/. Accessed: 2018-12-01.

[18] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[19] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, (Washington, DC, USA), pp. 263–272, IEEE Computer Society, 2008.

[20] "Rasa documentation - training data format." https://rasa.com/docs/nlu/dataformat/. Accessed: 2018-12-01.

Appendix: Results from Performance Evaluation

The models are coded as:
1. Baseline → B
2. Baseline and Noise → BN
3. Baseline, Noise and Lookup Table → BNL
4. Baseline, Noise and Synonym → BNS
5. Baseline, Noise, Lookup Table and Synonym → BNLS
6. Baseline and One Real Word Noise → BO
7. Baseline and Two Real Word Noise → BT

The histograms generated by RASA's evaluate function are in the figures below. Following are the tables, summarizing Precision, Recall and F1 score for each entity in all the different models.

Figure 6: Baseline Model

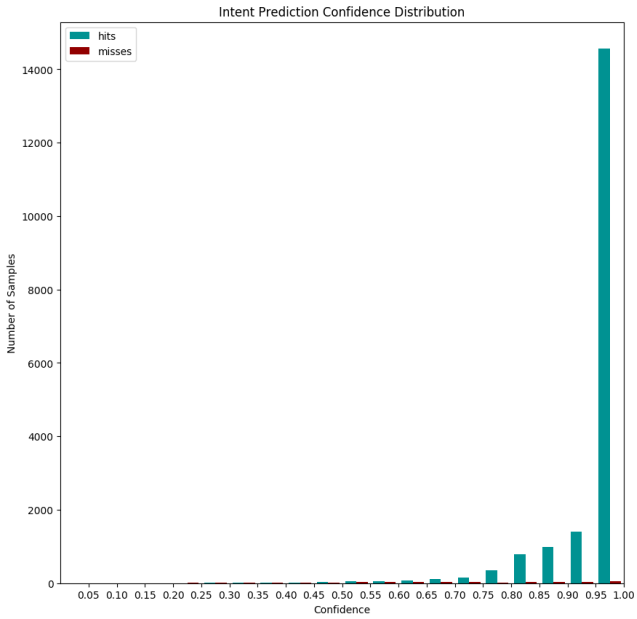


Figure 7: Baseline, Noise, Lookup Table and Synonym Model

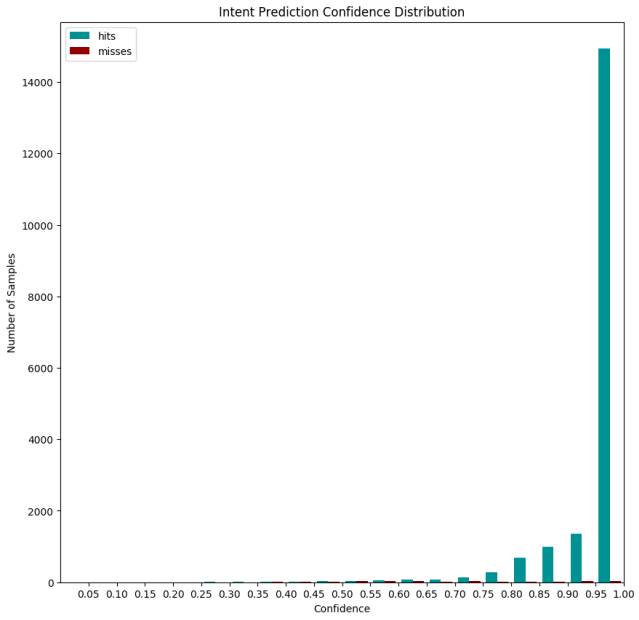


Figure 8: Baseline and One Real Word Noise Model

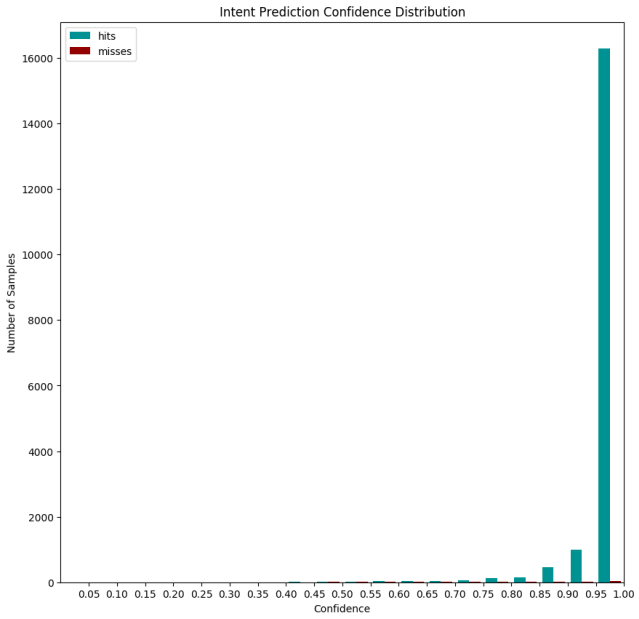


Table 1: Precision

Entities	B	BN	BNL	BNS	BNLS	BO	BT
AMOUNT	0.96	0.94	0.91	0.9	0.91	1	0.99
DRINK	0.75	0.75	0.88	0.84	0.88	1	0.99
FOOD	0.91	0.89	0.95	0.93	0.95	0.99	0.99
LOCATION	0.92	0.9	0.87	0.86	0.87	1	0.99
SIZE	0.99	0.94	0.97	0.97	0.97	1	1
TEMPERATURE	0	0	1	0.99	1	1	1
TOPPING	0	0	0.93	0.92	0.93	1	1
album	1	1	1	1	1	1	1
artist	0.99	0.99	1	1	1	1	1
best_rating	0.98	0.98	0.98	0.98	0.98	0.98	0.98
city	1	0.99	0.99	0.99	0.99	1	1
condition_description	1	1	0.99	1	0.99	1	1
condition_temperature	0.92	0.92	0.99	0.99	0.99	0.99	0.99
country	0.99	0.99	0.99	0.99	0.99	0.99	0.99
cuisine	0.98	0.98	0.98	0.99	0.98	0.99	1
current_location	0.99	0.99	0.99	0.99	0.99	0.99	0.99
entity_name	1	0.99	1	1	1	1	1
facility	1	1	1	1	1	1	1
genre	0.96	0.94	0.95	0.97	0.95	0.98	0.99
geographic_poi	1	1	1	1	1	1	1
location_name	1	1	1	1	1	1	1
movie_name	0.99	0.99	1	1	1	1	0.99
movie_type	1	1	1	1	1	1	1
music_item	0.99	0.99	0.99	0.99	0.99	0.99	0.99
no_entity	0.96	0.96	0.97	0.97	0.97	0.98	0.98
object_location_type	1	1	1	1	1	1	1
object_name	0.98	0.97	0.99	0.99	0.99	1	1
object_part_of_series_type	0.99	0.99	0.98	0.99	0.98	0.99	0.99
object_select	0.97	0.97	0.97	0.98	0.97	0.97	0.98
object_type	0.99	0.99	0.99	0.99	0.99	0.99	0.99
party_size_description	0.99	0.99	0.99	0.99	0.99	1	1
party_size_number	0.97	0.96	0.98	0.97	0.98	0.98	0.99
playlist	0.99	0.99	1	1	1	1	1
playlist_owner	0.99	0.98	0.98	0.99	0.98	0.99	0.99
poi	1	1	1	1	1	1	1
rating_unit	1	1	1	1	1	1	1
rating_value	0.97	0.97	0.98	0.98	0.98	0.98	0.98
restaurant_name	1	0.99	0.99	1	0.99	1	1
restaurant_type	0.99	0.98	0.99	0.99	0.99	0.99	0.99
served_dish	0.86	0.88	0.88	0.83	0.88	0.99	1
service	1	1	1	1	1	1	1
sort	0.99	0.98	0.99	0.99	0.99	0.99	0.99

spatial_relation	1	0.99	1	1	1	1	1
state	0.98	0.98	0.99	0.99	0.99	0.99	0.99
timeRange	0.99	0.99	0.99	0.99	0.99	0.99	0.99
track	1	0.99	0.99	1	0.99	1	1
year	0.88	0.9	0.91	0.9	0.91	1	1

Table 2: Recall

Entities	B	BN	BNL	BNS	BNLS	BO	BT
AMOUNT	0.31	0.27	0.79	0.75	0.79	0.96	0.98
DRINK	0.74	0.71	0.94	0.92	0.94	1	1
FOOD	0.38	0.34	0.39	0.44	0.39	1	1
LOCATION	0.39	0.41	0.78	0.82	0.78	1	1
SIZE	0.41	0.44	1	1	1	1	1
TEMPERATURE	0	0	1	1	1	0.99	0.98
TOPPING	0	0	1	1	1	1	0.99
album	0.98	0.94	0.95	0.98	0.95	0.99	0.98
artist	0.99	0.99	0.99	0.99	0.99	0.99	0.99
best_rating	1	1	1	1	1	1	1
city	0.97	0.97	0.96	0.96	0.96	0.96	0.96
condition_description	1	1	1	0.99	1	1	1
condition_temperature	1	1	1	1	1	1	1
country	0.97	0.97	0.96	0.97	0.96	0.97	0.97
cuisine	0.97	0.98	0.98	0.98	0.98	0.97	1
current_location	0.99	0.99	0.99	0.99	0.99	0.99	0.99
entity_name	0.99	0.99	0.99	0.99	0.99	0.99	1
facility	0.98	0.98	0.98	0.98	0.98	0.99	0.99
genre	0.99	0.96	0.99	0.99	0.99	0.98	0.99
geographic_poi	0.94	0.97	0.95	0.95	0.95	0.94	0.95
location_name	1	1	0.99	0.99	0.99	0.99	1
movie_name	0.98	0.97	0.97	0.97	0.97	0.97	0.98
movie_type	1	1	1	1	1	1	1
music_item	0.99	0.99	0.99	0.99	0.99	0.99	1
no_entity	0.99	0.99	0.99	0.99	0.99	1	1
object_location_type	1	1	1	1	1	1	1
object_name	0.96	0.98	0.97	0.97	0.97	0.97	0.97
object_part_of_series_type	0.96	0.96	0.96	0.96	0.96	0.96	0.96
object_select	0.98	0.97	0.98	0.98	0.98	0.98	0.98
object_type	0.99	0.99	0.99	0.99	0.99	0.99	0.99
party_size_description	0.99	1	0.99	0.99	0.99	0.99	1
party_size_number	0.96	0.97	0.97	0.96	0.97	0.96	0.96
playlist	0.99	0.99	0.99	1	0.99	1	1
playlist_owner	0.99	0.98	0.99	0.99	0.99	1	1
poi	0.97	0.98	0.94	0.95	0.94	0.95	0.97

rating_unit	1	1	1	1	1	1	1
rating_value	0.99	0.99	0.99	0.99	0.99	0.99	0.99
restaurant_name	0.95	0.95	0.94	0.95	0.94	0.94	0.96
restaurant_type	0.99	0.99	0.99	0.99	0.99	0.99	0.99
served_dish	0.96	0.93	0.95	0.94	0.95	0.97	0.97
service	1	1	1	1	1	1	1
sort	1	0.99	0.99	1	0.99	1	1
spatial_relation	0.99	0.99	0.99	0.99	0.99	0.99	0.99
state	0.98	0.99	0.98	0.98	0.98	0.98	0.99
timeRange	0.86	0.87	0.85	0.85	0.85	0.85	0.86
track	1	0.96	0.99	0.99	0.99	0.99	1
year	1	1	1	1	1	1	1

Table 3: F1 Score

Entities	B	BN	BNL	BNS	BNLS	BO	BT
AMOUNT	0.46	0.42	0.84	0.82	0.84	0.98	0.98
DRINK	0.74	0.73	0.91	0.88	0.91	1	0.99
FOOD	0.54	0.49	0.56	0.6	0.56	1	0.99
LOCATION	0.55	0.57	0.82	0.84	0.82	1	1
SIZE	0.58	0.59	0.98	0.98	0.98	1	1
TEMPERATURE	0	0	1	1	1	1	0.99
TOPPING	0	0	0.96	0.96	0.96	1	0.99
album	0.99	0.97	0.97	0.99	0.97	0.99	0.99
artist	0.99	0.99	0.99	0.99	0.99	0.99	0.99
best_rating	0.99	0.99	0.99	0.99	0.99	0.99	0.99
city	0.98	0.98	0.98	0.98	0.98	0.98	0.98
condition_description	1	1	1	1	1	1	1
condition_temperature	0.96	0.96	1	1	1	1	1
country	0.98	0.98	0.98	0.98	0.98	0.98	0.98
cuisine	0.98	0.98	0.98	0.99	0.98	0.98	1
current_location	0.99	0.99	0.99	0.99	0.99	0.99	0.99
entity_name	0.99	0.99	1	0.99	1	0.99	1
facility	0.99	0.99	0.99	0.99	0.99	1	0.99
genre	0.97	0.95	0.97	0.98	0.97	0.98	0.99
geographic_poi	0.97	0.98	0.97	0.97	0.97	0.97	0.97
location_name	1	1	0.99	1	0.99	0.99	1
movie_name	0.99	0.98	0.98	0.98	0.98	0.99	0.99
movie_type	1	1	1	1	1	1	1
music_item	0.99	0.99	0.99	0.99	0.99	0.99	0.99
no_entity	0.97	0.97	0.98	0.98	0.98	0.99	0.99
object_location_type	1	1	1	1	1	1	1
object_name	0.97	0.97	0.98	0.98	0.98	0.98	0.98
object_part_of_series_type	0.98	0.98	0.97	0.98	0.97	0.98	0.98

object_select	0.97	0.97	0.97	0.98	0.97	0.98	0.98
object_type	0.99	0.99	0.99	0.99	0.99	0.99	0.99
party_size_description	0.99	1	0.99	0.99	0.99	1	1
party_size_number	0.97	0.96	0.97	0.97	0.97	0.97	0.97
playlist	0.99	0.99	1	1	1	1	1
playlist_owner	0.99	0.98	0.98	0.99	0.98	0.99	0.99
poi	0.98	0.99	0.97	0.97	0.97	0.97	0.98
rating_unit	1	1	1	1	1	1	1
rating_value	0.98	0.98	0.99	0.99	0.99	0.99	0.99
restaurant_name	0.97	0.97	0.97	0.97	0.97	0.97	0.98
restaurant_type	0.99	0.99	0.99	0.99	0.99	0.99	0.99
served_dish	0.9	0.9	0.91	0.88	0.91	0.98	0.99
service	1	1	1	1	1	1	1
sort	0.99	0.99	0.99	0.99	0.99	0.99	0.99
spatial_relation	0.99	0.99	0.99	0.99	0.99	0.99	0.99
state	0.98	0.98	0.98	0.98	0.98	0.98	0.99
timeRange	0.92	0.92	0.92	0.92	0.92	0.92	0.92
track	1	0.98	0.99	0.99	0.99	0.99	1
year	0.94	0.95	0.95	0.94	0.95	1	1