DataBot: A Conversational System for Sourcing Data in CKAN

Ti Ern Ryan Tan

Supervisor: Prof. Philip Treleaven

Advisors: Michal Galas, Zeynep Engin

A dissertation submitted in partial fulfilment

of the requirements for the degree of

Bachelors of Science

of the

University College London

Department of Computer Science

University College London

30 April, 2018

Abstract

This dissertation describes the design, implementation and testing of a conversational system wrapped as a Comprehensive Knowledge Archive Network (CKAN) extension for sourcing data using natural language, similar to that of a chat bot, as part of the Urban Dynamics Lab (UDL) platform. The extension utilises Rasa, a conversational AI framework, to implement the Natural Language Understanding (NLU) and Dialogue models, which involves intent classification, entity extraction and classifying the next action based on current state. Rasa also exposes modules for parsing and processing input, model training, data representation and visualization and handling responses. CKAN, a data portal framework, is used for implementing the user interface and orchestrating data flow and interaction with the UDL Platform. The finished extension uses the NLU model to obtain search tags from user queries to perform data discovery on UDL data sources. This tool is important because it provides an alternative interface to search for data and is a prototype for the use of chat bots in data exploration, ultimately lowering the barrier to entry to data analysis for the non-technical community. The dissertation starts by describing the requirements gathered from UDL, the design, implementation and testing details of the system, an evaluation of the results, summary and future works of the project.

System Design

The system can be split into 5 components consisting of the User interface, Controller, Rasa Core Agent, Action Manager and Tracker Store.

- The User Interface is a minimalistic designed messaging interface for users to input messages and receive output, serving as the point of interaction with the system.

- The Controller acts as an interface to orchestrate the execution of the system. It is responsible for handling input and output channels, identifying the user, interfacing with the Action Manager and Rasa Core Agent, and formatting outputs as Data Objects, the data structure designed to for data transfer between client-server.

- The Rasa Core Agent which comprises of the NLU model and Dialogue model.

    - The NLU model comprises of 2 classifiers that performs pre-processing, intent classification and entity extraction on the input data through a processing pipeline. The system uses a pre-defined processing pipeline comprising of Conditional Random Fields for entity extraction and Support Vector Machine for intent classification and is trained over a dataset crafted for this problem domain.

    - The Dialogue model is a One Versus Rest Logistic Regression Multiclass classifier required to select the next executable action based on the current state and post processed user message. It is trained over a well defined user dialogues.

- The Action Manager is an interface that faciliates the actions that DataBot is able to execute.
- The Tracker Store stores Dialogue State objects which are Rasa abstractions for remembering the context of a conversation between the user and DataBot.

System Implementation

The system was implemented using a wide range of technologies and frameworks which are primarily Python related, with exception to the user interface:

- Users interact with the system through a messaging interface designed to be the terminal between users and DataBot. The user interface is implemented as a traditional web-based client that uses HTML, CSS, Javascript, Jinja2 and jQuery.
- The Controller resides on CKAN 2.6.3 web server which runs on Pylons. Modules were implemented between the Controller, Rasa Core Server and action methods which are exposed through an Action Manager
- The domain consists of 2 slots, 2 entities, 7 intents and 9 actions, with the core intents being sourceData, and requestHelp. The use of a Finite State Automata to design Stories greatly reduced the complexity of creating Story data and reasoning about Stories.
- The NLU model was implemented using a pre-configured pipeline, spacy_sklearn, provided by Rasa NLU. The main technologies in this pipeline are spaCy – text pre-processing library implemented in Cython, and scikit-learn (sklearn) –machine learning python library.
- The dialogue model was implemented using sklearn wrapped in a Policy. Policies have the advantage of loading and persisting the trained model as serialized data, removing the grunt interfacing work. It also exposes conversation state for debugging. The model was evaluate based on a “common sense” heuristic.
- A heuristic approach was used to create training data for Stories whereas templated sentences and tags found on UDL CKAN database were used to create training data for the language understanding model which consists of entity and intent classification data.

Testing and Results

Unit tests were prepared as a means to test correctness of individual methods and prevent regression as the system extends. Functional testing was carried out to ensure the dialogue responds according to the story scope. An evaluation of the Dialogue model was carried out through the use of a confusion matrix and appropriate precision, recall and f1-scores, critiquing its observed behaviour and identifying limitations. It was found that the both models over fit the training data which might be due to the lack of training data and the narrow dialogue scope. The outcome of the project is considered successful as it met all the “Must Have” requirements set out during the requirement gathering phase.

Contents

Chapter 1

Introduction

This chapter introduces the project's motivations, objectives, outcomes, approach and the report structure.

1.1 Motivation

Natural language has been the ultimate goal for query interfaces as humans ask questions about data in an intuitive way not ridden with rigid formalization. There has been initiative within the research community to develop natural language interfaces for interacting with data. Programming languages such SQL, Python and R provide rich expressive abilities for experts to analyse data based on their specific queries and ad hoc needs. With tools like Jupyter, users are able to interactively develop and test analysis methods. However, such conveniences and data analysis is constraint by technical proficiency and programming knowledge. Even a query language like SQL, which has a lot of similarity with English and has an expressive syntax suffers from rigid code formulation and knowledge about setting up the environment. It is prevalent that even in the scientific community, it is arguable that ease of use is the reason for Python's popularity, that though it severely lacks in execution efficiency as compared to counterparts like OCamel and C, its simplistic syntactic nature enables data scientists and enthusiasts to quickly extract the information they need and perform complex data analytics. However, this still remains a barrier for non-experts such as students, business users or policymakers who wants to make sense of their data without the need of wrestling with data analysis tools.

The feasibility of conversational agents as personal assistants such as Apple's Siri, Samsung's Bixby, Microsoft's Cortana and Amazon's Alexa has grown recently as software attempts to create personalized interactions that have access to the wealth of data available publicly as well as privately. These agents are so powerful because they allow users to interact with them in a human natural way with the use of all sorts of ambiguities that arise from slang words, technical terminology and contextual phrases. However, their aim to be generalizable enough to tackle a large domain space makes it difficult to gauge its performance as it can perform well in tasks like scheduling and reminders but may fail in conversations about the meaning of life. By reducing

the scope of the conversational system, we reduce the problem domain the system is expected to solve and have seen great results reflected in customer service conversational systems that aim to drive the conversation around the specific goal in an overall linear fashion. The technology is available to create powerful conversational systems given that the problem domain is carefully crafted.

In essence, DataBot, conversational system implemented in this project, hopes to be an expert system in the data analysis domain to guide users use the correct methods, fill in the right parameters, and ultimately return useful information about their data. It is targeted to improve accessibility with data for non-experts, introduced as an education tool for students and enthusiasts to focus solely on data analysis whilst deferring coding for more advanced levels and a convenient tool for data experts who want to quickly interact with their data.

1.2 Objective

The aims of this project is to design, implement and test a fully functional conversational system that would integrate into UCL Urban Dynamics Lab Data Portal as an extension and is to be to able source datasets from their existing database. Initial functional and non-functional requirements must be defined based upon the Urban Dynamics Lab requirements and use case analysis. System architectural decisions should be informed by current state of the art methods and tools. A personal objective is to increase my knowledge of Natural Language Understanding (NLU), the current state of the art methods and tools, and apply them into a data analysis product. NLU is concerned with representing human language in a machine useful way and creating seemingly intelligible machines that understand the functional and emotional semantics of natural language.

1.3 Outcome

The project outcomes are the fruition of the project objectives. The various components of the system are outline below:

1. The system needs to be a conversational system. As a result, the system would need to respond to user input through human relatable responses. Besides that, a conversational system will require that the decision made to select the next response must take into account the conversation topic and context and must have a mechanism to reasonably select that response.
2. The conversational system must resides on UCL Urban Dynamics Lab (UDL) Data Portal. UDL’s Data Portal is implemented using Comprehensive Knowledge Achieve

Network (CKAN), a popular data portal framework. Consequently, the conversational system will be implemented as a CKAN extension and must be completely open sourced.

3. The conversational system must be able to source datasets stored on UDL's Data Portal.
4. Any data used to model the conversational system must be relevant to the data portal's domain and thus, would not require a generalized approach.
5. The conversational system must be able to retain user context even once they have left the site. This may require the need for serializing user information into a database or data storage system.
6. The conversational system must be able to handle multiple users at once.
7. The conversational system would require a user interface that is intuitive and simple to use.

1.4 Project Approach

This section will discuss the different phases of the project and the selected approach.

1.4.1 Knowledge Discovery

Having had no prior knowledge to Natural Language Understanding or anything related, the first few weeks were spent learning the common techniques used by machines to understand natural language such as N-grams, Markov Chains, Word Embedding, Context Free Grammar and POS tagging mainly from online resources not limited to Standford University CS224n [1], MIT Open Courseware 6.034 [2], Stack Overflow [3] and Reddit's ELI5 [4]. It was also important that a clear definition of what constituted as a conversational system was formed.

1.4.2 Toolchain Research

It was important that the project applied state of the art methods yet the product would be accomplished in the limited time constraint. This involved looking towards frameworks and libraries that could speed up the development process, and still meet the non-functional requirements of the project.

1.4.3 Requirements Gathering

As this was a self-proposed project, the goals needed to be in line with UDL requirements, specifically for the UDL platform project. A few weeks were spent refining the requirements and narrowing the scope such that it achieved both UDL and personal goals.

1.4.4 Design and Implementation

An iterative approach to this project was taken, which involved moving back and forth between design and implementation. This was because of a coupling of certain project impediments that prevented me from easily acquiring all the information needed to properly scope the project such as how a conversational system fits into the UDL Platform, the current state of the UDL system and relevant data, and the time pressure to begin implementation.

1.4.5 Testing and Evaluation

Unit test were developed at the end project and were useful to identify bugs and edge cases which were not thought of beforehand. Functional tests for the correctness of Stories were designed to ensure the dialogue meets expected behaviour as well as resist against regression if the project were to be extended. Since that project is open for extension, it was vital that functional testing covered the specified domain, both the dialogue and, intent and entity classification so that quality of the conversations are maintained. Finally, the dialogue and language understanding models were evaluated using test data sets and confusion matrices.

1.5 Report Structure

Chapter 2 will cover the background and research undertaken to prior to starting and throughout the project, when knowledge gaps were identified. Information sources that were drawn upon would be outlined and referenced. Technologies and methods used would also be accredited. A literature review was carried out to identify 2 projects that developed Natural Language Interfaces which took on different approaches but were similar in nature.

Chapter 3 will detail the problem statement, requirements and use cases established that guided this project.

Chapter 4 will discuss the system design which includes system overview, user interface design, the domain definition and story scoping, the methodology behind modelling the data, and the Data Object.

Chapter 5 will present the system implementation which includes an architectural overview, implemented classes, core system interactions, and testing.

Chapter 6 will evaluate the Dialogue and Natural Language classifiers, project success, and discusses limitations found and assumptions made in the development of the system.

Finally, Chapter 7 provides a project summary and suggestions for future works.

Chapter 2

Background and Literature Review

This chapter presents the core background knowledge required to effectively carry out the project and make informed design decisions. A literature review that discusses two different implementations of Natural Language Interface systems was also carried out.

2.1 Background

2.1.1 Artificial Intelligence

Artificial intelligence (AI) is a well-studied field that has garnered a lot of attention since its inception in the 1950s [5]. AI is the field of study concerning intelligent computer agents which mimic the cognitive functions associated with the human mind such as learning and problem solving. An intelligent agent in general is any machine that maximizes its chances of success at achieving some goal. Thus, an intelligent agent which mimics cognitive functions is one that is goal-oriented in its ability to learn and problem solve outside of its programmer's scope. The long term goal of AI research is to conjure a machine with general intelligence that has the ability to perform any task a human can do successfully.

AI tasks are usually in some form of pattern recognition whether it is classification, clustering or regression [6]. There are many approaches in training and implementing intelligent agents such as reinforcement learning, evolution algorithms and convolutional neural networks [7]. In practice, these different techniques share in common the idea of representing the problem as some statistical model, popularly using weighted features and more commonly drawing inspiration from biological phenomena, to correctly model a classification or prediction task.

A big part of AI research is in Neural Networks and Deep Learning. The design of neural networks are inspired by how neurons interact and are organized in the brain, where the introduction of some form of perceptron to represent neurons and weighted connections between perceptrons to represent functionalities of dendrites, axons and synapses. These neural nets usually require large amounts of training data for the system to correctly respond as intended, a primary characteristic of the term "Deep Learning" [6]. Currently, state of the art AI has seen

some promising results such as its success in the ancient game of Go, complex multiplayer games such as Poker, Starcraft 2 and Dota 2 and text-to-speech systems like Tacotron 2 [7].

The learning process of an artificially intelligent system can be generally broken down into supervised and unsupervised. These terms are related to the method of training of the learning system such that supervised learning occurs where training is performed with "feedback" such as classified data sets and unsupervised learning occurs where training is performed without "feedback" such as on unclassified data and instead uses clustering techniques to classify groups of data into generic sets, where the data scientist would need to define what these sets represent. Examples of machine learning methods include support vector machines, logistic regression and k-nearest neighbours algorithm.

2.1.2 Natural Language Processing

Natural Language Processing (NLP) is the field of research that explores how machines can be used to understand natural language in the form of text or speech to perform useful tasks. It aims to manifest meaningful representations of human communication, particularly in linguistic form, to achieve some pre-defined objective or even objectives that have not yet been defined (as per generalized NLPs). This means that the ideal NLP is to convert natural languages like English and Mandarin that are ridden with contextual idiosyncrasies, slangs and even mistakes to a data structure that completely describes the meaning of the text, dismissed of any ambiguity, hence, allowing computers to execute actions based on them [8]. Clearly, this has yet to be achieved, and thus many NLP models are curated towards a specific domain with a finite number of tasks such as summarizing emails and documents [9] and personalized travel assistants [10], leaving the unrelated tasks undefined. NLP is derived from a collection of disciplines such as linguistics, computer and information science and artificial intelligence and has a plethora of applications in fields of speech recognition, translation and text processing to name a few. Progress in NLP can be attributed to 4 initiative groups– statistical and corpus-based methods; the use of a lexical reference systems such as WordNet that organize English components such as nouns and verbs into synonym sets that represent one underlying lexical concept; the use of Finite state automata and computationally lean methods; and collaborative projects to create large grammars [11].

2.1.3 Natural Language Understanding

Natural Language Understanding is a subset of NLP that deals with understanding the meaning of text. According to Chowdhury, the process of building computer programs that understand natural language involves three major problems: The first relates to thought processes, the second to the representation and meaning of the linguistic input, and the third to world knowledge. Thus, an NLP system may begin at the word level to determine the morphological structure and nature

(such as part-of-speech or meaning) of the word; and then may move on to the sentence level to determine the word order, grammar, and meaning of the entire sentence; and then to the context and the overall environment or domain. A given word or sentence may have a specific meaning or connotation in a given context or domain, and may be related to many other words and/or sentences in the given context [11]. This is true in the specific domain of data analytics where words such as mean and feature would synonymized to "average" and "attribute of the data set" rather than "to refer to something" and "to showcase". In its implementation, NLU objectives can be categorized into 3 parts; named entity recognition, intent classification and semantic analysis.

2.1.3.1 Named Entity Recognition

Named-entity recognition (NER), or entity extraction. is a subtask of information extraction that concerns itself with recognizing information units (entities) such as "Donald Trump", "UCL" and "United Kingdom" then classifying them into its respective categories, "person", "organization" and "location" from a piece of text. In other words, entity extraction has the objective of annotating entities in unstructured text in order for each token to be easily processed either on its own or as a whole sentence. Additionally, a more fine grained classification could be imposed on the previous examples being classified as "politician", "university" and "country" instead. Thus, the available categories are commonly those that have been pre-defined by the system with the exception of the use of a defined named entity hierarchy for ontologically classification.

Early studies of NER relied on rule-based systems which were rigid and invariant. However, recent NER systems use some sort of learning algorithm either through supervised, unsupervised or semi-supervised. The blended semi-supervised learning methods were introduced to counteract the cost of large amounts of annotate data required in supervised learning techniques. This section will focus on describing the semi-supervised and unsupervised method.

Semi-supervised require small amount of supervision, with an initial entity seed set, to begin the learning process. It tries to identify contextual clues that are common to the seed examples. The system then tries to identify entities similar to the seed set from using the texts' contextual clues. Iteratively, it applies the learning process to the new entities as to discover new relevant contexts.

Unsupervised learning utilizes lexical resources such as WordNet, lexical patterns and statistics computed on large unannotated corpora to obtain named entities from clustered groups based on contextual similarity [12].

2.1.3.2 Intent Classification

Intent classification aims to correctly classify the intention of an input query. There are 3 main hurdles for accurate intent classification. Firstly, there is the challenge of obtaining a semantic representation that can distinguish between intents. Traditional approaches make use of a set of seed queries that represent an intent which, in turn, requires large samples of labelled training data

for better representation. Secondly, intent classification faces the challenge of domain coverage. The intent classifier based on previous learning approaches relies heavily on labelled sample data and hence, would not be able to classify intents which are not represented in that data. A generalized (or seemingly generalized) intent classifier would then require large amounts of intent diverse data. Lastly, intent classifiers need to understand the semantic meaning of user input query. Currently, one of the most used approaches is the bag-of-words method [13].

2.1.3.3 Semantic Analysis

Semantic analysis is a field of linguistics that aims at analysing the meaning of words, fixed expressions, sentences and utterances, usually in context. In practice, this means converting the natural language into an intermediary metalanguage that can be decomposed into a suitable structure for extracting meaning. The processing pipeline usually involves a combination of parsing the utterance into a syntactic structure that is the grammatical role of each individual word by means such as Context Free Grammar and Top-Down Parsing and analysing the structure in context to select the most appropriate meaning of the utterance [14]. The extraction of meaning depends on the application of semantic analysis which may include information retrieval, information extraction, text summarization, data-mining, and machine translation and translation aids. [15] In the chat bot use case, intent classification and entity extraction are the desirable outcomes of the semantic analysis phase where an utterance corresponds to an intended action and within that utterance, specific components pertaining to a category or ontology which are specific and essential to the objective are extracted.

2.1.4 Methods

Key methods used during the design and implementation of DataBot are summarized in this section. The goal was to seek an intuition of these methods that would help understand how to better apply them in the project because they are implemented as APIs through the *sklearn* [16] library.

Conditional Random Fields (CRF) is a framework for building probabilistic models to segment and label sequence data [17]. Its aim is to maximize the conditional probability of sequences of variables and predict a label of a given sequence [18]. CRFs are commonly said to be an improvement over the Hidden Markov Model and Maximum Entropy Markov Model where its main advantage over other probabilistic models are it doesn't require conditional independence of the observations, providing an efficient and effective segmentation and labelling of data. [18]

Logistic Regression is a statistical method for binary classification of patterns. In the context of machine learning, it is a supervised method that relies on the use of a logit or sigmoid function to predict the probability that a dependent variable $\boldsymbol{y}$ belongs to a certain class $\boldsymbol{c}$ given a set of

independent variables $\boldsymbol{x}$, where independent variables with their respective coefficients are linearly combined. The use of a cost function $\boldsymbol{J(\theta)}$ [19] is to identify the set of coefficients $\boldsymbol{\theta}$ such that for some $\boldsymbol{y}$, $\boldsymbol{P(y = c \mid x)}$ is maximized ($\boldsymbol{J(\theta)}$ is minimized), and similarly $\boldsymbol{P(y = \bar{c} \mid x)}$ is minimized ($\boldsymbol{J(\theta)}$ is minimized). Multinomial logistic regression (MLG) [20] is a generalization of the logistic regression for multiclass (as opposed to binary) classification. The output of a $\boldsymbol{K}$-class MLG is a $\boldsymbol{K}$ dimensional vector where each feature in the vector is $\boldsymbol{P(y = K_i \mid x)}$, where $\boldsymbol{K_i}$ is the class at the $\boldsymbol{i^{th}}$ index, and is usually normalized so that the output sums to one. In a One Versus Rest MLG [21], a binary problem is fit for each class instead and the output is not normalized.

A **Support Vector Machine (SVM) Classifier** aims to classify patterns by using an optimal linearly separating hyperplane such that the margin between the classes' closest points is maximized. Points lying on the margin boundaries are called support vectors and the hyperplane in the middle of the margin is the optimal separating hyperplane. When the patterns cannot be classified linearly, the use of a kernel function projects the data points into a higher dimensional space [22], colloquially known as feature space [23] which can be linearly separable. When tuning the SVM, the 2 main hyper parameters to be chosen carefully are the regularization parameter which determines the trade-off between minimizing training error and minimizing model complexity and the kernel function [24].

Precision, Recall, F1-Score and Confusion Matrices are techniques used to evaluate the performance of classifiers. Precision is the ratio of correctly classified positive observations against all classified positive observations, in other words, its accuracy. Recall is the ratio of correctly classified positive observations against all observations in the class, in other words it resembles coverage. Precision and recall are usually inversely related, therefore, the F1-score takes this trade off into account and is the harmonic mean between precision and recall [17]. A confusion matrix is a matrix that is often used to test the performance of classifiers where the true values are known such that its axes denote the predicted and true labels, and cell values denote the frequency. Using a confusion matrix is a black-box method because only the outcome is known and the reasons affecting the outcome is not obvious.

2.1.5 Conversational Systems

Conversational system (chat bots) are a category of natural language interfaces that present an end-to-end solution –comprising a natural language understanding and dialog engine – to engage in goal-oriented conversations with human users. It receives natural language input and executes one or more related commands that gets it closer to its goal, optionally responding in natural language. For machine-human conversations to flow smoothly, the dialog engine needs to have a strategy when deciding the next response.

The 2 types of chat bots are rule based and Artificial Intelligence (AI) based. Rule based conversational bots utilize pattern matching for deterministic response, where each input is mapped to an output, and can get more complex when conversation state is tracked. AI bots use both supervised and unsupervised machine learning techniques such as Markov Decision process [25] to represent dialogue strategy (strategy to which the next response is selected) as an optimization problem and Sequence-to-Sequence based frameworks [26] which produces a conversational model that maps a sequence of tokens to another sequence of tokens, both of variable length.

There is a strong desire to test and measure the effectiveness of chat bots such as at the Annual Loebner Prize [27] which mixes in human judging in the Turing test, a commonly used measurement of a chat bot's ability to exhibit intelligent behaviour [28] . In some dialog systems, a chat bot can be modelled as an optimization problem to minimize the cost of the objective function which enables quantitative success metrics [25]. In Collobert's work, the performance of their general NLP model was benchmark against the state of the art specialized systems in Part-of-Speech Tagging, Sentence Chunking, Named Entity Recognition and Semantic Role Labelling, all of which are important tasks in effective NLP [8].

2.1.6 Rasa

Rasa is an open soured software framework for developing conversational systems by incorporating state-of-the-art machine learning research into developer tools. This essentially bridges the gap between natural language understanding research and its applications. Rasa decouples language understanding from dialogue management into separate components. In other words, natural language understanding, state tracking, dialogue strategy, and response generation are either a part of the language understanding or dialogue management component as opposed to end-to-end learning systems. This means Rasa is made up of 2 modules; Rasa Core and Rasa NLU. Consequently, responses are selected from a list pre-defined by the developer as it is noted to be more reliable in generating sentences which are grammatically and semantically correct [29]. Intentional decoupling is due to its modular architecture philosophy with which Rasa is built, to support integration with other conversational systems such as existing bot and web front ends.

One key benefit of Rasa is training models can be done offline and models are persisted in local storage by default, which reduces friction during development. Besides that, developers have full access to the modules' source code which enables customization to take place at the source code level. Models can be developed separately which may aid splitting work packages among teams and narrow down fine tuning. Since the system can be developed locally, no confidential data passes through third-party services. Rasa is under the Apache License 2.0 which allows for

modification, distribution and patent use, ideal for UDL's use case. A list of its solutions and use cases could be found here [30]

Rasa Core

Rasa Core is responsible for handling functionality related to dialogue management which includes the domain, tracker, actions, policies and utilities [31].

The Domain defines precisely the knowledge of the system in 5 different characteristics, each expressed as a list; Entities, Slots, Intents, Actions and Templates.

- Entities are pieces of information desired to be extracted from a user message which are commonly used as input parameters to an action.
- Slots are pieces of information desired to be tracked during the conversation duration. Slots types are defined as either text, float, categorical, bool, list, unfeaturized or as a custom type, where each type is intuitively understood with exception to unfeaturized which denotes that that piece of information will not be accounted for in the dialogue state.
- Intents are intention labels for user messages. It defines what intentions the system expects users to respond with.
- Actions are a list of available responses the system can perform.
- Templates are pre-defined textual responses.

By pre-defining a domain, the problem of classifying input reduces tremendously at the cost of generalization especially when the data sets are considerably small.

The Tracker keeps track of the current dialogue state of the conversation through a tracker object uniquely identified by the user id. Tracker objects store slots and logs all the events that had occurred up until the current state. By storing the events, the tracker can conveniently be restored to its current state (or any state in the conversation) by replaying the events in chronological order which is useful for debugging.

An Action can either be a simple utterance in response or an arbitrary function. The action is able to make use of dialogue state through the tracker with access to slots as well as intent confidence levels and a list of entities recognized. Effectively, the actions could also query databases, make an api call or update the state of the system its applied to.

Policies determine which action the bot should take next based on the current dialogue state. Policies makes use of the current dialogue state by instantiating a featurizer to create vector representations of its inputs as input to the dialogue model. The dialogue model used for dialogue strategy is wrapped in the Policy class during its definition.

Rasa Core makes available a host of Utilities to aid the development process. The 3 main utilities are:

1) A fleet of APIs through an Agent object to perform operational tasks such as loading in a serialized dialogue model and interpreter (parser and featurizer), training, and handling messages.
2) Training a group of policies at one time so that different models can be cross validated against test data and compared against each other for best performance.
3) A visualization tool which lets developer visualize the conversation flows of their training data as a Directed Graph of intents and actions.

Rasa NLU

Rasa NLU is the natural language understanding module provided by Rasa. Rasa's Language understanding represents an input of text as a list of intents and their respective confidence levels, and a list of entities. This component is replaceable with other language understanding components that classify intent and extract entities.

Rasa NLU has pre-defined processing pipelines which utilize robust and high quality text processing and language understanding python libraries such as spaCy, scikit-learn and mitie. The goal of a processing pipeline is to transform raw text from user input into the appropriate data structure consisting of intents and entities, which are exposed to the internals of Rasa Core. The pipeline can have any number of components depending on the complexity of text transformation as long as dependent components are defined sequentially. As of writing, Rasa does not support custom components and only pre-configured components are able to be included in the pipeline, but it is in Rasa NLU's roadmap [32]. The current list of components made available are intent featurizers, intent keyword classifiers, synonymizers, regex featurizer, tokenizers and entity extractors using different learning models.

2.1.7 Training Data

Rasa NLU requires NLU Data to train the language understanding model and Rasa Core requires Story data to train the dialogue model. The data for each model must be supplied in their respective training data formats [33, 34].

NLU data

NLU data is represented as a JSON object with 3 attributes; common examples, regex features and entity synonyms. Common examples is the main data segment that comprises of a sentence and associated intents, entities and optionally, entity synonyms. Regex features aids the classifier to recognize intents and entities by providing regular expression patterns. Entity synonyms are

used for synonyms to specific entity values. These synonyms will be used to augment the parsed data through the synonymizer if it were included in the processing pipeline.

Story Data

Stories are a sequential list of intents and actions to represent the interaction between a user and the conversational system where words making up the sentence is abstracted away and the system is only concerned with the intents from users and expected action in response. In essence, the Stories define the dialogue flow between the users and conversational system in a structured way.

2.1.8 Comprehensive Knowledge Archive Network (CKAN)

Comprehensive Knowledge Archive Network (CKAN) is an open sourced web framework used for making open data websites. On the backend, it provides admins a data management dashboard similar to that of Wordpress and many popular Content management systems for managing blog posts and pages. On the frontend, users can source data and preview it using built in visualization widgets such as maps, graphs and tables. On the technology side, CKAN is built with Python on the backend and Javascript on the frontend. It makes use of Pylons as the web framework, SQLAlchemy as the ORM, PostgreSQL as the database and Apache Solr as the search platform. One key advantage is its modular architecture which allows extensions to be developed to provide additional features such as customized visualization layers, automated archiving and OAuth support [35]. CKAN is the web platform used by UCL Urban Dynamic Labs in its UDL platform. In CKAN, packages are used to refer to a cluster of linked datasets and resource are used to refer to a single dataset, which will be the terminology when referring to datasets that reside specifically on CKAN.

2.1.9 UDL Platform

Figure 2.1 illustrate the proposed UDL platform architecture. As of writing, the UDL Platform is still under development by UCL Urban Dynamics Lab. The aim of UDL Platform is to automate data provisioning, auditing, and metadata generation from well-known online sources to provide them pre-processed and readily available on an open access data portal [36]. The data portal would be backed by Hadoop and Comprehensive Knowledge Archive Network (CKAN) and is to be a central avenue for data experts to source quality data.

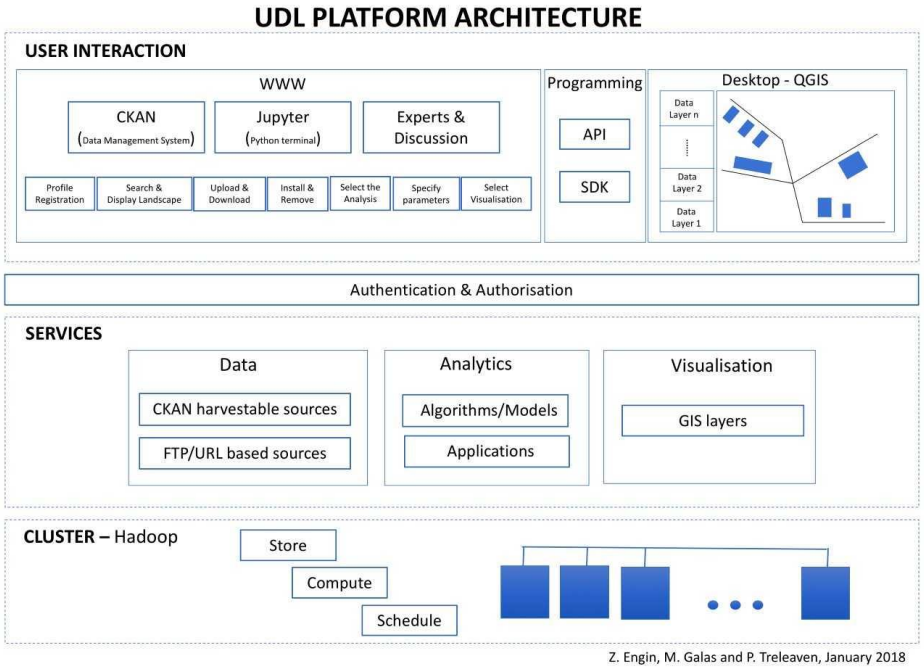

Figure 2.1: UDL Platform Architecture

2.2 Literature Review

2.2.1 Constructing an Interactive Natural Language Interface for Relational Databases

The work done by Li and Jagadish describes the architecture of an interactive natural language interface (NLI) to query relation data. The paper introduces a Natural Language Interface for Relational databases (NaLIR) to translate logically complex English language sentences into a correctly translated SQL query [37]. The author begins by making a comparison between natural language interfaces for databases (NLIDB) and keyword search engines to identify the most crucial pitfalls. The first of 2 key problems identified was that results from search engines are verifiable from reading the abstract while results returned from NLIDB cannot explain themselves. The second key problem was NLIDB queries are more sophisticated than search engine queries yet ridden with ambiguities. The overview strategy used by the developed system to address these problems was to explain the disambiguation process to the user when handling "hard" ambiguities which are organized in 3 steps.

The first is the use of a Dependency Parser which aims to understand the natural language query linguistically. It uses the Stanford Parser to generate a linguistic parse tree. Each node is a word or phrase specified by the user and each edge is a linguistic relationship between 2 nodes. The next step is to use a Parse Tree Node Mapper to identify nodes in the parse tree that can be mapped to SQL Components. In cases where nodes fail to map to any SQL component, the system warns the user and ignores it. In cases where there are multiple mappings of a parse tree, the system uses the Parse Tree Structure Adjustor described in the next step. Finally, the Parse Tree Structure Adjustor, aims to solve the problems of incorrect linguistic parse trees, out of sematic coverage of the system or the parse tree being ambiguous from the databases perspective. The Structure Adjustor solves this by first making the parse tree fall in linguistic coverage by reporting candidate mappings to the user for him/her to decide interactively, then inserts implicit nodes into the chosen parse tree to make it more semantically reasonable, which is also done under user supervision.

The system was measured by effectiveness and usability where effectiveness is defined as the quality of results returned and usability is defined as the ease of use for non-technical users. Effectiveness was evaluated by the percentage of queries that were perfectly answered by the system. The issues with this metric is that returned results which have high overlaps are still marked as wrong. It was also noted whether the user could recognize that the answers were wrong. Usability was measured using the time taken to complete correct queries and through a post-experiment questionnaire.

The experiment found that the system had a 90% effectiveness rate for a set of 98 queries, where 4 out of 10 of the wrong answers were accepted by the users. By the usability measure, the average level of satisfaction for easy, medium and hard were 5, 5, and 3.8 respectively on a scale from 1 to 5, where 5 denotes extremely easy to use.

2.2.2 Eviza: A Natural Language Interface for Visual Analysis

Setlur et al. discusses the design, implementation and evaluation of Eviza, a natural language interface (NLI) that facilitates interactive querying with an existing data visualization [38]. The problem the authors were tackling was the problem of enabling users to have interactive conversation to explore existing visualizations. The approach taken by the authors began by performing an initial user study to understand how users interact with various type of visualizations such as, but not limited to, scatter plots, bar charts, time series and heat maps. Eviza was developed as a web-based, traditional server-client model. The data flow process could be broken down into 5 modules; grammar and semantics, autocomplete, ambiguity handling, pragmatics, and analytics.

The system requires a method to obtain a structural description of the input query. This is performed using a probabilistic grammar model, LL(*) parsing, that is applied algorithmically. The production rules are augmented with syntactic and semantic predicates based on the visualization context. Augmented semantics from existing knowledge bases enables the system access to vast amounts of common sense and domain specific knowledge. These semantics are obtained from third party corpora such as WolframAlpha's unit taxonomy and Wordnet's synsets.

An identified problem with NLP systems is to communicate to the user the types of inputs which are supported. To handle this, Eviza uses a sophisticated autocomplete mechanism that parses typed input dynamically to generate candidate template queries as a drop-down below the query input box. The suggestions are informed using a combination of user history, input statistics and context free grammar.

Ambiguity arises due to discrepancies between the user's mental model and the system's model. The Eviza team quantized semantic and syntactic entropy using cosine similarity and Wu-Palmer similarity respectively. These metrics are used to inform the presence of an ambiguity widget which take different forms depending of the data type of the ambiguity.

Eviza provides support for a range of analytical functions to provide statistics and update the visualization. The analytical functions include general analytics (min, max, average, highlighting, colouring etc.) and special cases of spatial and temporal analytics, for example point-based

distance measure, point-in polygon relationships, fuzzy spatial prepositions, and temporal units, propositions and connectives.

The system was mostly evaluated qualitatively, comparing Eviza with Tableau – direct manipulation system. Tableau users found that using Eviza provided a more natural and overall positive experience. The experimenters observed that Eviza performed best where the task required many clicks, users didn't know how to do the task in Tableau, users didn't know where to find the control functions or users didn't know which category an item belonged to.

Chapter 3

Requirements

This chapter presents the requirements analysis performed which consists of a detailed problem statement, a list of project requirements and use cases.

3.1 Problem Statement

The UDL platform is sub divided into many components, one of which is Landscape for Data which aims to tackle the problem of sourcing useful datasets online, hence providing an avenue to list as many open access and licensed datasets divided by categories. As of writing, there are 20 recorded data source providers that are listed on their project page. Within Landscape for Data, there is a need for a user interface to search and display the available datasets that a user may access in CKAN.

The purpose of this project is to design, implement and test a fully functional conversational system that would integrate into UCL Urban Dynamics Lab Data Portal as a CKAN extension and is able to source datasets from their existing database. The project is experimental and aims to become grounds for a case study on the effectiveness and usefulness of a conversationally focused search engine, in hopes that it may become a useful data exploration tool. The conversational system will be called DataBot, a general name that is unrestrictive to only data sourcing since extensibility is taken into account because use cases such as in-chat data visualization, data analysis and uploading packages through the messaging bot are potentially viable. The project will use technology that is open sourced, and support customizable features that provide flexibility for further extension. DataBot will be integratable into the current UDL platform.

3.2 Requirements

The requirement gathering was curated by UDL requirements, system attributes, and feasibility within the time constraint. It was mainly an iterative process between the UDL advisors and I, meeting up with them once a week, to identify an intersection between their needs and the proposed project plan. The requirements were mainly communicated informally, providing me flexibility to implement functionality that best suited the situation. Deeper understanding of the current state of UDL platform moulded the requirements over time[1]. The finalized requirements were inferred from the communications with UDL clients and the system attributes required to tackle the stated problem. They were outlined using the MoSCoW method because it presents a prioritized view of requirements which assisted in developing an implementation roadmap. The finalized functional requirements and non-functional requirements are in Sections 3.2.1 and 3.2.2 below.

[1] The initial requirements included tagging packages based on topics. However, a closer look at UDL platform revealed that packages had tag metadata associated with them, given by the resource owners who contributed the packages.

3.2.1 Functional Requirements

(C) denotes a core function.

ID	Details	Priority (MoSCoW)
FR1(C)	DataBot will source packages from UDL CKAN as a list of results given a query crafted in natural language. The language used will be English.	Must
FR2	DataBot will search for relevant UDL packages using metadata associated with each package.	Must
FR3	DataBot will be able to persist trained models in some form of storage so that it can be reused in the future.	Must
FR4	DataBot will handle multiple users concurrently.	Must
FR5	DataBot will store user context to continue from previous conversation.	Must
FR6	DataBot will use user context to make decisions on its next actions.	Must
FR7	DataBot will have a front-end messaging user interface that will be the main terminal of interaction between DataBot and its users.	Must
FR8(C)	DataBot will have a mechanism to let the user know how it can assist the user to accomplish his or her goals.	Must
FR9	DataBot will check the correctness of it "understanding" what the user meant at appropriate times such that a balance between validating its understanding and frictionless conversation is maintained.	Should
FR10	The system will prompt users about appropriate methods that are useful for the task they are describing.	Should
FR11	DataBot will be able to handle spam and unrelated queries in an appropriate way.	Should
FR12	DataBot will use a model that is dynamically trainable as it spends more time interacting with the user.	Won't

Table 3.1: Finalized Functional Requirements

The details of a functional requirement in Table 3.1 relates the input, behaviour, and output of a specific task. This framework of defining a functional requirement was adhered to where possible. Core functions are functions that are used for DataBot objectives as detailed in Story Scoping (Section 4.3.2).

3.2.2 Non-Functional Requirements

ID	Details	Priority (MoSCoW)
NFR1	The system will use only free and open sourced technologies that will remain free and open sourced.	Must
NFR2	Full unit test coverage will be performed for all source code used to implement the system.	Must
NFR3	Appropriate functional tests will be carried out for all defined functionality to validate its correctness.	Must
NFR4	DataBot will integrate into the currently available UDL platform as part of a CKAN extension. As an implication, it will be installed on their production environment.	Must
NFR5	The system will not create noticeable negative impact on the other services currently hosted on UDL CKAN instance.	Must
NFR6	The system should have automated integrated testing scripts to be running on the production server.	Should
NFR7	DataBot will be usable by UDL as part of its platform at the end of the project.	Should
NFR8	The system will be extensible by design.	Should
NFR9	A full or partial documentation of the system will be prepared before project handover.	Should
NFR10	DataBot will use its own custom vocabulary and web embedding to utilize a search based on Bag of Words technique	Could

Table 3.2: Finalized Non-Functional Requirements

The details of non-functional requirement in Table 3.2 defines the specific standard for the operation of the system at an architectural level. However, the definition is extended to include external project constraints[2].

[2] NFR1, NFR7, and NFR9

3.3 Use Cases

The use cases were developed from analysing functional requirements. In the context of a conversational system, these represent the dialogues, turn-by-turn, from the start to the end, guided by some overarching objective. Note that not all functional requirements are meant for end users, in particular, FR3 is meant for future extensibility by other developers. As mentioned in Section 3.2, some uses cases, namely FR8 and FR9, were added in well into the implementation phase because it improved usability of the system. This was informally gathered through user feedback and evaluation. The main purpose of FR9's inclusions is to counteract the inconsistencies that comes with using a classification model to classify every sentence into simply 7 intentions. Besides that, the use case list is non-exhaustive. Instead, use cases listed are the main functionality closely related to a user or DataBot objective. These are further built upon in greater detail to develop Stories in Section 4.3.2. The list of use case descriptions can be found in Appendix C.

Chapter 4

System Design

This chapter describes the design of the developed system, and details the methodology used to decide key design choices which includes the user interface design, the domain definition and story scoping, methodology for data modelling, and the Data Object.

4.1 System Overview

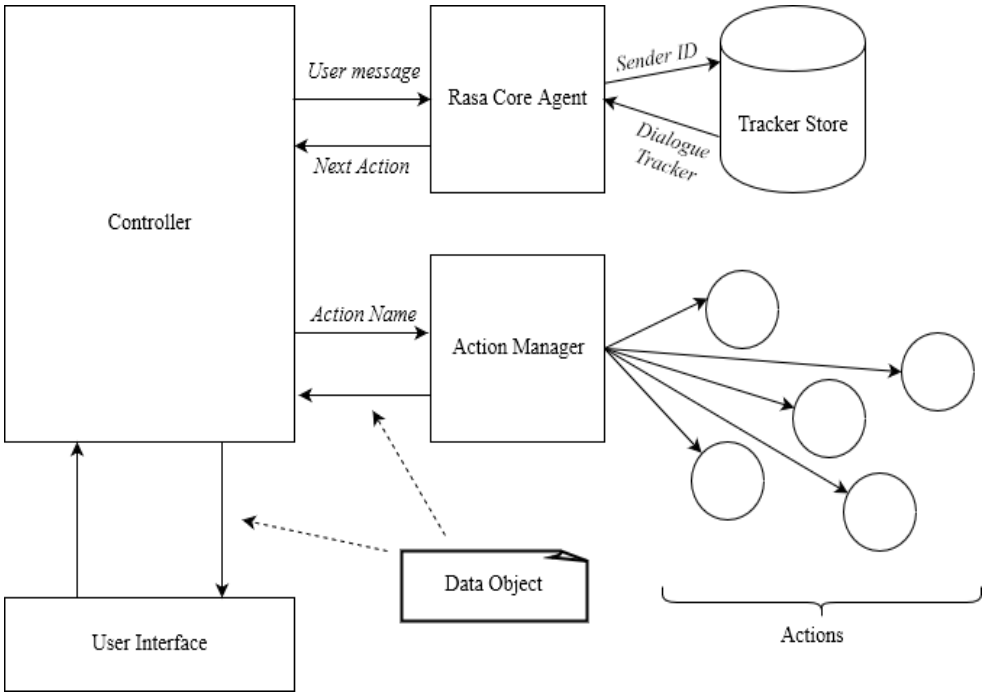

Figure 4.1: System overview

Figure 4.1 shows the overview of the system and data exchanges that occur between different components. The system comprises of 5 components, User Interface, Controller, Action Manager, Rasa Core Agent and Tracker Store.

The User Interface (UI) acts as the terminal for message transfer between the user and DataBot. The Controller is responsible for orchestrating data flow from user request to DataBot response which includes identifying the user, handling the action prediction loop and formatting responses into the Data Object. The Action Manager is responsible for executing the next actions which are predicted. The actions are those defined in the domain (Section 4.3.1). The Rasa Core Agent processes the input by extracting entities and classifying intents which are then used to predict the next action that should be taken. The Tracker Store is used to store the context and past history of the conversation.

4.2 User Interface

An analysis of existing user interfaces was carried out before designing the interface to highlight key features of an intuitive messaging service user interface. The analysis compares 2 existing messaging services that target different audiences, one for the general public and the other for a technically inclined one, so that overlapping components could be extracted from them. A wireframe of the proposed user interface was created and detailed.

4.2.1 User Interface Analysis

Implementation principles were gathered from popular messaging systems using heuristic evaluation [39] against the 10 principles of usability outlined in [40]. The popular messaging systems evaluated were WhatsApp and Gitter. These services were chosen because they targeted different audiences, where WhatsApp targets a general audience and is suitable for both formal and casual communication and Gitter targets a technically inclined audience and is suitable for formal communication. For relevancy, only their web clients were evaluated. The detailed evaluation can be found in Appendix A. The analysis will first describe what was notably similar, then what was notably different and what is relevant and irrelevant to DataBot.

A centred messaging wall was found on the landing page of both services, after login. This is trivially understood to be the most important aspect of a messaging system. The service incorporated social elements such as Rooms or Groups and having a list of friends and activate conversations. More subtle design choices include a search bar that either performed an omni or facetted search for messages, groups or people, input boxes without buttons that send message on the “Enter” key, distinct differentiators for sent and received message, timestamps on each message and latest messages always appeared at the bottom.

Where the systems differed were largely due to the difference in target audience. As would be expected, more advanced features were found in Gitter such as formatting using markdown and shortcut keys, and formatted code snippets. It was noted that Gitter used more text based

buttons compared to its counterpart, which mainly used icons, suggesting that some of its features can't be intuitively understood from an icon. A noticeable style difference is the layout of messages, where WhatsApp uses message bubbles that do not fill the entire width of the screen and Gitter uses card which fill the entire width of the screen.

It was noted that many features were to incorporate social elements which are irrelevant in the case of DataBot. Distinction between sent and received messages wouldn't need textual cues because there can only be 2 participants in the conversation. A centred messaging wall would capture the attention of the user immediately. A search function to quickly find word occurrences in the conversation would be useful but not required. Adopting from WhatsApp's intuitive styling will be relevant to DataBot moving forward because there is only one messaging related functionality at this point.

4.2.2 Wireframe

The proposed wireframe found in Figure 4.2 includes only essential components which are a message wall, text box, scroll bar, and heading, adopting an intuitive and minimalistic design. A simple heading describes DataBot in one sentence below the UDL platform page header. Message bubbles appear stacked atop of each other, with latest message below, as this would allow for richer media that fit into box-like components such as cards, maps, images or iframes. Since the conversation will only have 2 participants, it is not necessary to name the message owners. Instead, a dark coloured background is used for sent messages and no colour for received. A right scrollbar will automatically scroll to the latest message when messages overflow. A text box with a clear placeholder is added at the bottom of the frame that sends a message with the "Enter" key.

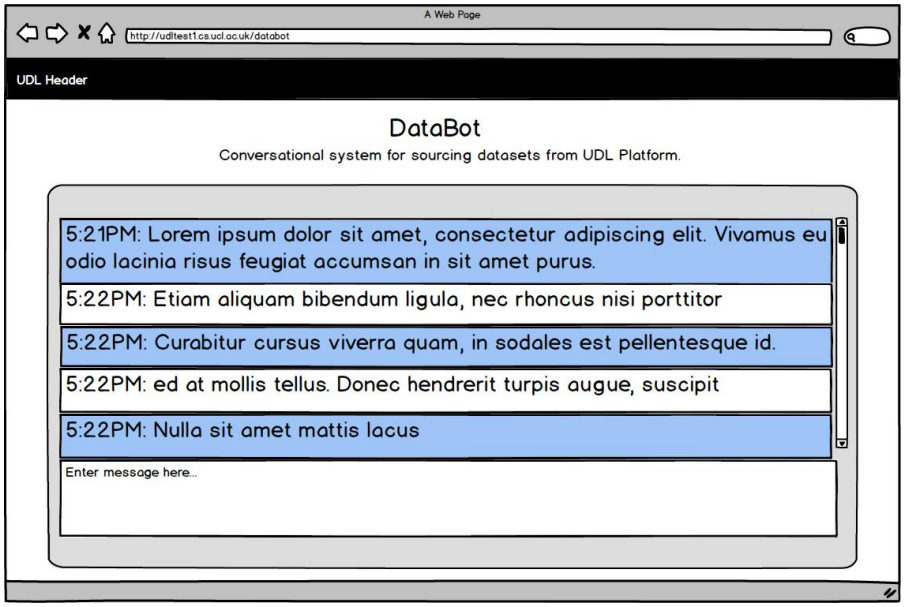

Figure 4.2: Proposed Wireframe

4.3 Domain Definition and Story Scoping

The Domain definition and Story scoping are required to define the "world" known to DataBot where the domain is a list of elements and Stories are list of sequences of actions and intents. Domain definition and Story scoping are tasks that inform each other and were completed iteratively between the two.

4.3.1 Domain Definition

Slots, Entities, Intents, Actions and Templates are listed in Tables 4.1, 4.2, 4.3, 4.4, and 4.5 respectively. Each table details the name of the element and describes its necessity in the Domain.

ID	Slot	Type	Description
S1	Tags	ExtendedListSlot	Tracks a list of query tags identified so far.
S2	Limit	Text	Tracks the number of results to return which is set by the user.

Table 4.1: List of defined Slots

ID	Entity	Description
E1	Tags	Identifies query tags from a user utterance if any is available.
E2	Limit	Identifies query result limit from a user utterance if any is available.

Table 4.2: List of defined Entities

ID	Intent	Description
I1	Greet	Detect that the user intended to greet DataBot
I2	Farewell	Detect that the user intended to greet farewell to DataBot
I3	Affirm	Detect that the user intended to affirm
I4	Deny	Detect that the user intended to deny
I5	requestHelp	Detect that the user intended to request for assistance in using DataBot
I6	sourceData	Detect that the user intended to source packages
I7	sourceDataProvideTags	Detect that the user intended to provide query tags to source packages

Table 4.3: List of defined Intents

ID	Action	Description
A1	action_greet	Returns a greeting to the user.
A2	action_farewell	Returns a farewell greeting to the user.
A3	action_offer_help	Returns an offering to assist the user.
A4	action_source_data	Retrieves results from server side API, formats results into a response, and returns the response.
A5	action_help	Returns a message that tells the user about DataBot's current functionality and how to use them.
A6	action_clarify_understanding	Returns a question that checks if understood intent and entities are classified correctly when confidence levels are below a certain threshold.
A7	action_reoffer_help	Returns a question that asks the user if he/she needs any more assistance
A8	action_source_data_prompt_tags	Returns a question that prompts the user to provide query tags
A9	action_reset_slots	Resets current filled slots
A10	action_give_up	Returns a statement that demands the user to rephrase his/her message when confidence levels are below a certain threshold.
A11	action_restart	Resets all events and slots.

Table 4.4: List of defined Actions

ID	Template	Description
T1	action_greet	Response template to greeting users
T2	action_farewell	Response template to greet farewell to user
T3	action_offer_help	Response template to offer assistance
T4	action_source_data_prompt_tags	Response template to prompt for query tags
T5	action_reoffer_help	Response template for reoffering help

Table 4.5: List of defined Templates

4.3.2 Story Scoping

In Rasa, Stories define the dialogue between users and DataBot by representing it as a list of events, where each event is either an intent from a user or an action by DataBot. Note that intents can be classified alongside their respective entities. A key problem with Stories is that users can have arbitrary responses, and therefore, the need to define exactly the Stories the conversational system can handle is important so that each Story has an objective and the system's conversational limits are clearly defined. Story scoping is a necessary step before creating Story data to train the dialogue model on.

Conversations between the user and DataBot were modelled linearly towards a particular objective, which is a function to be executed. These are core functions FR1 and FR8 in Section 3.2.1. The rest of the functions are required to minimize the distance between moving from what the user wants and the user's objective, and to improve usability. In general, a conversation between a user and DataBot would follow a path similar to:

1. DataBot asks user what they want.
2. If there is sufficient and correct arguments to fill parameters of that task, DataBot executes with correct arguments in those parameters.
3. Otherwise, ask specific question to fill unfilled parameters. Repeat from step 2.
4. Return to step 1.

The assumption here is that the user will always respond with a certain level of predictability given a question from DataBot which is certainly not true. This is further addressed in Chapter 6.

When DataBot is waiting for an input intent and entity, it can be said that it is in a particular state. Each next intent and entity detected represents an input which shifts DataBot into another state. When DataBot enters a state, it executes an action. DataBot will wait for a user message if there is no input paths to take, emitting an *action_listen* which has been omitted for clarity. The *setSlot* action has been omitted because the system assumes that every entity detected will automatically be registered as a slot in our use case.

Stories were represented as a 5-tuple *modified* Finite State Automata (FSA) $\boldsymbol{M} = (\boldsymbol{\Sigma}, \boldsymbol{S}, \boldsymbol{s_0}, \boldsymbol{\delta}, \boldsymbol{F})$ such that

$$\boldsymbol{\Sigma} = \{I1, I2, I3, I4, I5, I6, \{I6, E1\}, \{I6, E1, E2\}, \{I7, E1\}, \{I7, E1, E2\}, \{I7, E2\}\}$$
$$\boldsymbol{S} = \{A1, A2, \ldots, A11, \emptyset\}$$
$$\boldsymbol{s_0} = \emptyset$$
$$\boldsymbol{F} = \{A4, A5\}$$

with $\boldsymbol{\delta}$ defined in the state transition diagram in Figure 4.3, where $\emptyset$ is a state with no other actions except *action_listen*. Each state is associated with the actions it executes and inputs are the set of intents that is detected.

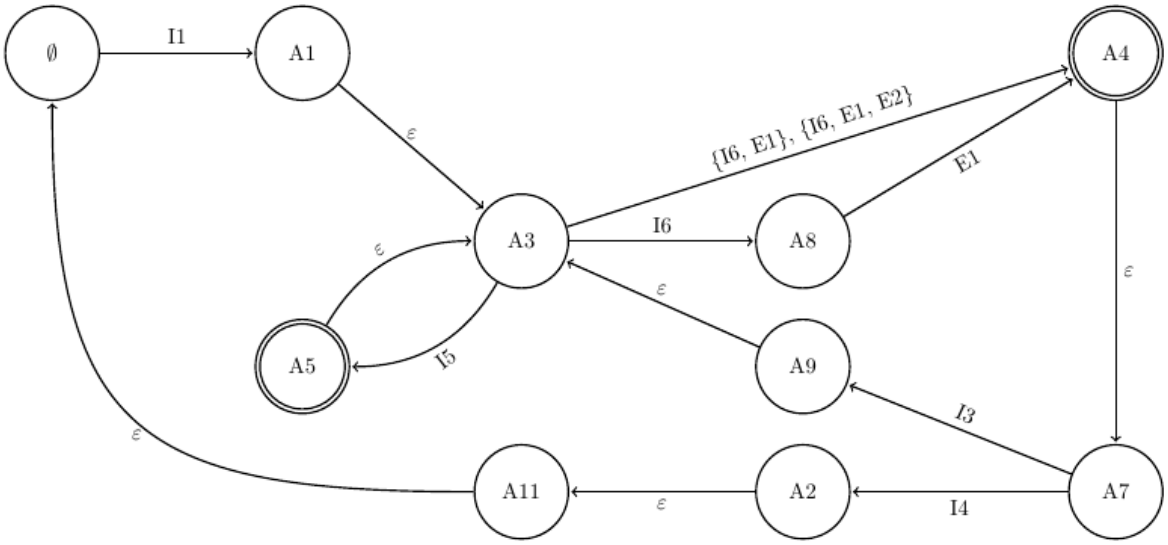

Figure 4.3: State transition diagram for $\boldsymbol{\delta}$-component in $\boldsymbol{M}$.

Actions *action_clarify_understanding* and *action_give_up* were not represented in the Stories because they do not alter the state of DataBot. These actions are used when confidence levels are below certain thresholds to check that DataBot has understood the user correctly or request the user to rephrase his/her message. However, due to last minute code refactoring, they were not included in the final product even though were thought of in the initial design.

An FSA was used to design Stories because of its characteristic of state transitioning and the well-defined nature. However, Stories do not inherit all characteristics of an FSA. For example, FSA's are used to determine whether an input is accepted or rejected. Our *modified* FSA in Figure

4.3 contradicts this in 2 characteristics. Firstly, the input string is possibly an infinite set, as conversations between the user and DataBot could potential run for a very long time. Consequently, amendments were made to handle infinitely long conversations so that the conversation will be able to progress without ending abruptly. Secondly, because of the aforementioned deviation, the *modified* FSA's objective is not to determine whether an input is acceptable, but is to move the dialogue from any initial state to a final state. Notice that after any move into a final state, a ε-transition moves the state back into an intermediary state, effectively restarting the path from a new initial state and never determining whether an input is acceptable or not. Nevertheless, the use of FSA's to model Stories has greatly reduced the complexity of Story data creation.

4.4 Data Creation

4.4.1 Story Data

The decoupling for the language understanding and dialogue model requires that 2 sets of training data be provided, one for each model in their respective Training Data Format (see Section 2.1.9). Created Story data follows the FSA model design in Section 4.2.2 closely except that when a *farewell* intent is identified, DataBot will always perform an *action_farewell* followed by *action_restart* as an exit option for the user so that the user is not trapped in a conversation state. To create the data, checkpoints were assigned for dialogue flows that commonly reoccur in each conversation and within the conversation itself. They are used to modularize chunks of the dialogue and create conditional branching.

Two heuristics for data creation were made from observing the FSA in Figure 4.3. Firstly, checkpoints should reflect that the use of a ε-transition on an edge between states A_X and A_Y implies the executions of A_X and A_Y will be chained without the need for user input in between. Secondly, checkpoints always await for a finite set of user input when it enters a state with conditional branches such as A_3 and A_7. However, it was found that the addition of A_7 as a checkpoint did not reduce the complexity of creating correct and diverse data. Checkpoints were also cautiously used because, as of writing, it is still under development and may contain unknown side effects [41]. There were a total of 61 Stories.

Story data was chosen to be created for 2 reason. Firstly, as far as I know, there is no publicly available dialogue datasets in the field of conversational data analytics. Secondly, Story data follows strict formatting with a mixture on intents, entities, and actions. Using real dialogues would need heavy automated pre-processing and a significant amount of manual correction.

4.4.2 NLU Data

NLU data requires a sentence classified with an intent and relevant entities (as those stated in Section 4.3.1) with their appropriate indices in that sentence. At the time of data creation, there were no automated tools to generate data in this format but as of writing there are services such as Tracy [42] and Chatito [43]. The approach I took to separate the data creation processes into 3 phases; obtaining entities, creating pre-filled sentences and generating classified sentences.

To obtain entities, I queried CKAN for all the available tags which amounted to 21,252. The data was cleaned by converting non-alphanumerical characters into whitespaces, and removing tags which contain non English words and tags that denoted IDs which reduced the dataset to 6,435 tags. The next step was to create pre-filled sentences which were labelled with their respective intents. These sentences used a placeholder where either a limit or tag entity was desired instead. The set of sentences were created subjectively without the use of heuristics or rules. Of course, this leaves it liable to a lack of coverage and biases. Unfortunately, this is one of the limitations currently faced when developing a conversational system for industries where previously there was no human-human conversation support. Sentences have strong biases depending on where the data was sourced from, and the quality of conversation data is usually erroneous and difficult to correct. Snips overcomes this by sourcing their data creation jobs to a pool of humans [44]. Lastly, classified sentences were generated by filling in the placeholders with their respective entities. The data was divided into training and testing with a 7 to 3 split.

4.5 Data Object

The Data Object is a custom data structure abstraction used as the standard data structure for data transfer between components that was designed to solve the need of having more expressive types where handlers only require a single method of processing transferred data. In essence, it forms a tree that when used in conjunction with a queue, is unpacked breath-first. This is further detailed in the client response handler (Section 5.5). There are 3 different types of Data Objects; *string, list* and *source_data_object*. A *string* and *list* Data Object has a *data* attribute which consists of either a string or a list of Data Objects respectively. A *source_data_object* contains the attributes *title, organization,* and *number of resources*.

Chapter 5

Implementation and Testing

This chapter describes the implementation details of the system, the classifier models, custom and extended modules, core functionality of the system and the testing performed which includes unit testing and testing of the Dialogue Model.

5.1 System Architecture

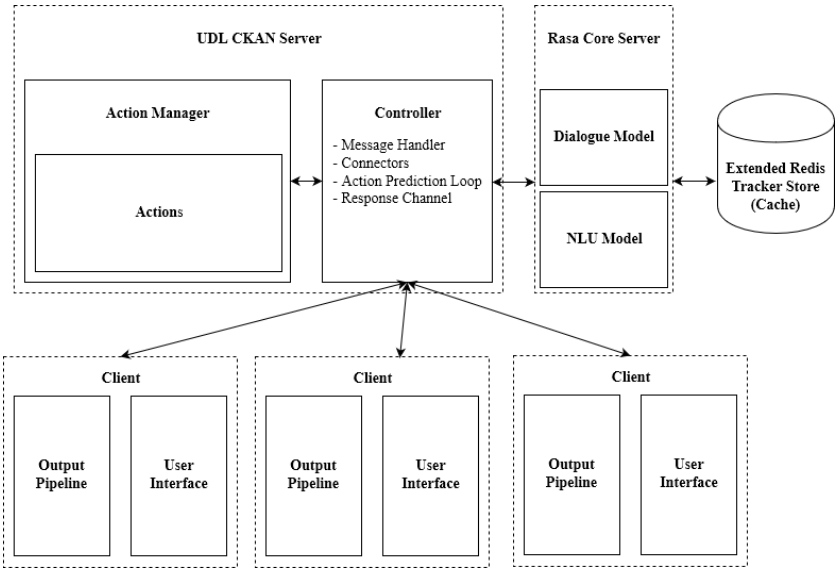

Figure 5.1: System Architecture of implemented components

Figure 5.1 shows the system architecture which is organized as a traditional web-based, client-server model. The expected Client-side consists of 2 main components which are the output pipeline and user interface implemented using HTML, CSS, Bootstrap 4, Javascript and jQuery. The output pipeline is responsible for formatting Data Objects into well-structured and correctly styled user messages and the user interface is the terminal for users to send and receive messages. The UDL CKAN Server hosts the service which is exposed through API endpoints from the Controller. The Controller performs message handling, instantiates connectors to interact with the

Rasa Core Server, manages the action prediction loop and collects responses in the Response Channel. Actions are exposed through an Action Manager, which is a static module that resides on UDL CKAN Server. It was tested that the Rasa Agent was blocking for 80-100 seconds during initialize when the production CKAN server were reloaded, which resulted in the decision to host the service separately on a Twisted server provided through the Rasa Core Framework [45]. The service is managed by a Supervisor script which is responsible for starting, stopping and reloading the server. Redis, an in-memory database, is used as the Tracker Store because it boasts low latency and is supported by CKAN.

As an overview, when a user sends a message, the client makes an AJAX request to the UDL CKAN Server which gets handled by the Controller. The Controller identifies the user, checks that the Rasa Core Server is running then passes its ID and message to be parsed by the Rasa Core Server. The server parses input, updates the Tracker Store and returns the next predicted action to the Controller which executes the action through the Action Manager. This continues until Rasa Core Server predicts an *action_listen.* The response returned from each action is collected into a Response Channel, which gets formatted and sent back to the client. The client processes the response, formats the output and appends it to the user interface message wall.

5.2 Classifier Models

The Dialogue classifier uses a logistic regression to classify the next action based on the current state of the system. In Rasa, this is wrapped as a Policy which handles the featurizing of input. The NLU model is 2 separate classifiers, an intent classifier and entity classifier. The intent classifier uses a Support Vector Machine to classify vectorised sentences where the sentences are tokenized using *spaCy* and vectorized using a statistical word embedding model, *en_core_web_md*, from *spaCy*. The entity classifier uses Conditional Random Fields to extract the entities. These models were provided by the Rasa Core and NLU framework respectively. Initially, logistic regression and the NLU models were intended to be used as the model for the Minimum Viable Product. However, due to time constraints, I could not complete a full implementation and cross validation of different models. However, this has not hindered the quality of the classification as it was observed that the most influential factor for both classifiers was the training data (Section 6.1).

5.3 Custom and Extended Modules

This section describes custom and extended modules that will be referred to in the remaining chapter. The custom modules defined are different types of interfacing layers. Interfacing layers are important because it provides a standardized way for different components to communicate between them. Connectors are interfacing layers between servers and the Action Manager is the interface between actions and Controller. Modifications were introduced into *rasa_core* modules to solve the problem of incompleteness due to the Rasa framework still being in unstable development phase and to meet specific needs of the system.

RasaCoreConnector

An abstract class to be inherited by Rasa Core Connectors. It implements generic *get* and *post* methods and handles their responses for interfacing between the Rasa Core Server and the CKAN Server. Besides that it also defines configuration information such as the port and base URL of the Rasa Core Server.

ParseConnector

A wrapper for the Rasa Core Server endpoint */conversations/{sender_id}/parse.* Responsible for querying the *parse* request from the Rasa Core Server. The *parse* POST request takes as input a JSON object with field *query*, where the value represents the input message of the user. The message is then parsed by the Rasa Core Server which returns a JSON object representing the dialogue state. The dialogue state includes the predicted next action to be executed and additional information regarding the parsed text such as confidence levels of other intents, detected entities, current slots, sender's id and message timestamp.

ContinueConnector

A wrapper for the Rasa Core Server endpoint */conversations/{sender_id}/continue.* Responsible for reporting that an action has been executed by making a POST request with a JSON consisting of fields executed action and events. Executed action is the action that had been executed and events is a list of Rasa Core event names that inform Rasa Core Server to update the user context with the new events. When the ContinueConnector is used, Rasa Core Server will return the same data structure as it returned to ParseConnector.

VersionConnector

A wrapper for the Rasa Core Server */version.* Responsible for checking if the Rasa Core Server is running. This is implemented by making a GET request to check for the version of rasa_core library currently in use.

UDLApiConnector

An interface between the Controller and UDL database. The class implements *search_package* which takes as input a list of tags and an integer limit, that makes a Solr query through either a REST API call if the system is in development or the *ckan.plugins.toolkit* interface if the system is in production. If the REST API is called, special characters in the target URL are replaced with percent escape characters.

ActionManager

A static module responsible for interfacing between the Controller and executable actions. Each predicted action needs to be executed by the Controller through the Action Manager. The actions are exposed through an *action_map* dictionary which is the only callable attribute of the module that maps action names to their executable functions. If an action is called but not defined, the Action Manager is intended to throw a fatal error.

ExtendedRedisTrackerStore

This extension extends the RedisTrackerStore defined in *rasa_core.tracker_store* to create DialogueState (tracker) objects that are deleted after an idle amount of time. The timeout value is updated on each interaction with the caching server.

ExtendedListSlot

Due to the Rasa framework still being in unstable development phase, its original *List* slot type actually holds a string instead of a list. Because of this, when multiple instances of the same entity type are detected in a single user utterance, the detected entities replaces the slot value instead of appending to it. Effectively, only the last detected entity fills the slot. This is not representative of all the tags in a user query. The ExtendedListSlot was implemented to solve this problem by initializing its *value*[3] attribute as a list and modifying *__setattr__*[4] to append the new value whenever there is an assignment to *value*.

ExtendedRasaCoreServer

[3] Attribute that holds the slot's value
[4] Responsible for defining how class instances handle assignments

This extension is responsible for enabling custom Tracker Stores to be used instead of the default *InMemoryTrackerStore*.

5.4 Handling Messages

Messages are handled by calling a POST request to one of the CKAN server endpoints. Within the server, the Controller will check messages for errors such as not being in JSON format, and if an empty message was sent. Next, the server will identify if this is the first message from the user. Because users are not required to login to use the service, the Controller will check if the session id has been saved before. If it has not been saved before, the Controller will save the current user's session id as a sender id. Next, the Controller will verify that the Rasa Core Server is running by querying it through a *VersionConnector*. If no response is received, the Controller exits early with an error response to the user. Otherwise, the message and sender id will be used as input to the Rasa message handler, which initiates the Action Prediction Loop, which will return a list of Data Objects. The returned value is serialized into JSON format and returned to the calling client.

5.5 Action Prediction Loop

When the Rasa message handler is called with a message and sender id, it initializes an empty List Data Object known as the response channel. The Rasa Core Server is queried using the *ParseConnector* with the input data to initialize the action prediction process which will return the next action to be executed. The action is then executed using the Action Manager. Next actions are continually predicted by querying the Rasa Core Server through the *ContinueConnector* in a loop until an *action_listen* is emitted. The *ContinueConnector* also indicates to Rasa Core Server to take into account any actions that were executed and events that had occurred. Each prediction is followed by executing the action through the Action Manager and collecting the results in the response channel. When an *action_listen* is received, the message handler returns the response channel to the caller.

5.6 Sourcing Packages

When the *action_source_data* is called with inputs *tags* and *limit* through the Action Manager, an API is executed that searches UDL database for relevant packages through the *UDLApiConnector w*here queries are structured in SolrQueryFormat. The results are sorted by relevancy in descending order. Each returned result goes through a processing pipeline that reduces the information into title, organization and number of resources and formatted in a *source_data_object* Data Object. The processing pipeline strips away non-printable ASCII

characters so that more primitive endpoints such as command line terminals are supported. The query returns a higher number of packages then requested by the *limit* parameter because it was found that some packages were not associated with any resources (data). Currently, the effective search limit is an additional 10 results on top of the input *limit.* The processing pipeline includes into the output set packages from the results set that have at least 1 resource until *limit* is reached or no more packages are in the result set.

5.7 Client Response Handler

A response handler in the calling client unpacks the response into a queue. When Data Objects are dequed, the 4 cases that are handled depends on whether the Data Object is of type *list*, *string*, *source_data_object*, or not a defined type. A *string* Data Object sent to the output pipeline. Data Objects of type *list* Data Objects are pushed into the queue. *Source_data_object* Data Objects are processed into grammatically acceptable sentences with the appropriate HTML formatting before being sent to the output pipeline. An error message informing the user that the data was undecipherable is sent to the user if the type is not defined. Effectively, this reflects a breath-first exploration of a tree starting from the root.

5.8 Output Pipeline

The output pipeline is responsible for appending new messages to the message wall and formatting the message bubbles. The message bubbles are unfilled if the message is received and filled blue if it were sent. Messages are formatted with <*li*> tags with class “human” or “bot” for their respective styling and appended to a designated <*ul*> location using jQuery. The message wall is styled using Boostrap 4 and CSS.

5.9 Testing

5.9.1 Unit Testing

The process of testing a unit of code is known as Unit Testing. This usually tests the output of a method that performs a single task and can be extended to test for statements and system state given some input. Its objective is to test that the system works as expected in it “lowest” level of code. The approach I took to unit testing tested the behaviour of each global function and class method for given correct input, and the identified edge cases. It was decided that private methods are included as a means of regression testing if the system were to evolve. The unit tests were performed using python’s built-in unit testing module.

5.9.2 Testing the Dialogue Model

The approach I took tested that the Dialogue model had been trained to exhibit the behaviour defined in the use cases (Section 3.3) by supplying it a mixture of stories that it had seen and not seen. The unseen stories differ in length and the number of entities detected which includes capturing up to 6 entities when testing *sourceData* and *sourceDataProvideTags*. The goal was to obtain a confusion matrix where frequencies are found only on the diagonal which can be found in Appendix B.

Chapter 6

Evaluation

This chapter evaluates the Dialogue and NLU models, the project results based on the requirements set out in Chapter 3 and the system limitations and assumptions made. The limitations are divided into key limitations and notable limitations.

6.1 Model Evaluations

Both models were evaluated by computing the precision and recall of their performance on a testing data set. The model has perfect accuracy in classifying the said criteria if the precision is 1.

6.1.1 Dialogue

To evaluate the Dialogue model, testing data was created using "common sense" as the heuristic. For example, consider the case in which the user begins the conversation by requesting to source data instead of greeting. A suitable response is for DataBot to respond with *action_source_data* (A4). However, this response short-circuited the state transitions that were set out in Section 4.2.2. One would then infer that a successful model would be able to always handle responses by short-circuiting the state transition. This is an incorrect conclusion because of the following counter example. Consider the case in which DataBot is in state *action_source_data_prompt_tags*. At this point, DataBot expects the user to provide tags to source on, but instead, the user replies with a greeting. DataBot's appropriate response is not immediately clear. From the state transitions, the correct response would be to treat the greeting as a tag to source data on, which may not have been the user's intention. In human to human interactions, this is unlikely to happen, however, it is not known how users will interact with DataBot and this is definitely a possible case. In either case, the absolute correct behaviour is uncertain.

The approach I took to evaluate the model was to use a "common sense" led evaluation. Precision and recall were noted and f1-score was computed for completeness but was not used.

The results were also measured using a confusion matrix similar to the approach used in testing the dialogue model (Section 5.9.2).

In the "common sense" testing data set, a spam story of length 16 for each intent was created. For example, intent I_1 should always be followed by actions $A_1$and A_3. The following precision, recall, f1-score and number of occurrences for each actions are detailed in Table 6.1 below. From the scores, we observe that the worst performers in precision are Farewell, followed by Greet then Offer Help and Restart. The following paragraphs will identify weakness of the model in relation to the "common sense" test data set. Reset slots is omitted because it had no occurrences. As it would be identified in the succeeding paragraphs, the reasons behind the lack of precision are not independent.

Action	Precision	Recall	f1-score	Number of occurrences
None	0.00	0.00	0.00	0
Farewell	0.00	0.00	0.00	16
Greet	0.32	0.50	0.39	16
Help	1.00	0.94	0.97	16
Listen	1.00	0.97	0.99	78
Offer Help	0.77	0.75	0.76	32
Reoffer Help	1.00	0.75	0.86	28
Reset Slots	0.00	0.00	0.00	0
Restart	0.67	1.00	0.80	16
Source Data	1.00	0.75	0.86	28

Table 6.1: Precision, recall, f1-score and number of occurrences for each action in the "common sense" test data set.

A farewell intent is always succeeded by a Farewell action which is always succeeded by a Restart action as noted in *modified* FSA. It can be seen from the confusion matrix in Appendix D that 16 out of 16 occurrences of Farewell actions are being classified as Greet actions and 16 out of 16 occurrences of Restart actions are being classified correctly. This indicates the model was able to generalize that a restart action should always succeed 2 steps after a *farewell* intent. However, the model miserably fails to predict a Farewell action after a *farewell* intent and instead

predicts a Greet action. This might be because Restart action shifts DataBot to state ∅, where the Greet action is always succeeding. This would also explain the poor precision in Greet action.

A *greet* intent is always succeeded by a Greet action which is always succeeded by an Offer Help action as noted in *modified* FSA. Greet has 8 occurrences classified as Farewell and 8 occurrences which supposed to be Offer Help was classified as Restart. It was found on every other occurrence of a *greet* intent, the model would predict a Farewell followed by Restart action. This is surprising since the Greet action only occurs after the greet intent in the training data. The occurrences of a Farewell followed by Restart action is partly due to enabling an exit option as discussed in Section 4.4. This would also explain Offer Help and Restart's lack of precision.

Without doing further testing, it would not be possible to identify the cause of Source Data's lack of recall because Source Data is called after 2 different intents, *sourceData* and *sourceDataProvideTags* and their succeeding actions are identical.

The model performed well in the Help and Listen actions. The behaviour of the Help action should be expected because it is always used in a closed loop with the *requestHelp* intent. Precise Listen actions is also expected since the test data set conformed to the *action_listen* mechanism set out in Section 4.3.2.

Overall, the model performed in extremes in different cases which suggests that it is over fitted for the training data. This is might be because of 2 problems which were identified late into the project development. Firstly, as previously mentioned, there is not absolute correct response to ambiguous cases. Instead, this needs to be informed from real user data or having controls implemented that prevents these edge cases from occurring. Secondly, the Story scope is too small for generalization to add value. In such a case, the use of a rule-based AI for this problem domain would be more suitable. However, as the system extends to more functionality and complex interactions, a rule-based AI would become obsolete and the need for a more intelligent agent would be recognized.

6.1.2 NLU

To evaluate the NLU Model, the precision and recall for the intent classification and entity extraction were noted and analysed separately, where precision is used as the primary metric and recall as the secondary metric similar to Section 6.1.2. F1-scores were computed for completeness but not used in the evaluation.

Intent	Precision	Recall	F1-score	Number of occurrences
affirm	0.67	0.5	0.57	4
deny	0.17	0.17	0.17	6
farewell	1	0.2	0.33	5
greet	1	0.17	0.29	6
requestHelp	0	0	0	3
sourceData	0.94	0.95	0.95	1068
sourceDataProvideTags	0.93	0.93	0.93	874

Table 6.2: Precision, recall, f1-score and number of occurrences for each intent in the test data set.

The results from Table 6.2 show that the most poorly classified intents were *requestHelp*, followed by *deny*, then *affirm*, as noted from their precision scores. The lack of training samples for each of these intents is most likely the biggest cause of its poor performance. Unlike *sourceData* and *sourceDataProvideTags*, the diversity of training samples were much greater in the smaller sets and little to no repeated words were found from sample to sample. However, it is also noted that *farewell* and *greet* intents have a similar training sample size but a perfect precision. The reason for this observation might be because of the nature of greet and farewell, where sentences of the two intents are more invariant and less prone to ambiguity. Small training samples would mean that experimenter biases has a larger influence.\

Attribute	Precision	Recall	F1-score	Number of occurrences
tags	0.99	0.97	0.98	7359
limit	1	1	1	468
number of entities	0.96	0.98	0.97	5319

Table 6.3: Precision, recall, f1-score and number of occurrences for each entity in the test data set.

The results from Table 6.3 infer that the entity extractor extracted *limit* and *tags* entities with near perfect precisions and it was almost able to identify the correct number of entities in each sentence. The *limit* and *tags* entities that were supplied are unique and have not been seen by the

entity extractor before. This meant that the model was able to generalize on the characteristics of the entities.

6.2 Success Criteria

The project is considered a success because all "Must Have" requirements set out in Section 3.2.1 and 3.2.2 have been met. Table 6.4 details how the system has fulfilled each "Must Have" requirement. Within the scope of UDL Platform's objectives, the system is able to search for datasets using tag relevancy and the Solr search engine.

ID	Details
FR1	DataBot is able to source packages through the *search_package* method of the *UDLApiConnector* exposed through the Action Manager when the user has intent *sourceData.* The *search_packages* method searches for datasets based on tag relevancy using the Solr search engine.
FR2	*search_package* method uses relevancy scores determined by Solr.
FR3	After training, the models are stored as binary *pickl* files offline.
FR4	As long as each user is on a separate host machine, DataBot stores the session ID to keep track of each conversation.
FR5	DataBot stores the context of each conversation in Redis exposed through the *ExtendedRedisTrackerStore.*
FR6	Policies featurize the current tracker state and user input as a vector that is used as input to the Dialogue model.
FR7	A web interface was implemented as a terminal of interaction between DataBot and its users.
FR8	Users can request help from DataBot which will inform the functionalities currently implemented in DataBot and how to utilize it.
NFR1	All libraries, packages and modules used are free and open sourced. No paid service was used in the development of this service.
NFR2	Each implemented method from classes and modules have associated unit tests.
NFR3	Functional tests were carried out on the Dialogue Model to ensure its correctness, prevent regression and identify limitations.
NFR4	DataBot can be accessed through the UDL CKAN website.
NFR5	The Rasa Agent is exposed as a separate web service to avoid blocking the CKAN web server on reload. Load and runtimes have no noticeable difference.

Table 6.4: Description of system functionality that fulfils the requirements.

6.3 Limitations and Assumptions

A key limitation included is that there is no support for performing any operations on the found packages as the system does not returns the resources themselves. Theoretically, the dialogue model cannot handle dialogues outside its story scope as the model seemed to over fit the training data. Without obtaining fresh user data from survey, questionnaire or experiments, unbiased alternatives for the domain of conversational language data interaction interfaces are lacking. Besides that, the system has primitive "natural" capabilities in the sense that it does not have any human-like characteristics such as identifying spelling mistakes, having reasonable delay in between responses, crafting its own response messages and having a "personality" it its approach. Furthermore, DataBot always expects the user to greet it first and can never initialize the conversation. This can be resolved using additional conditional control flow, but as of writing, Rasa Stories do not have support that enables this feature. Another key limitation is that the current version of DataBot classifies all utterances into one of seven intents which will cause return unexpected responses to the user.

A notable limitation to this project's approach is it did not compare the performance between multiple learning models for either dialogue prediction, intent classification or entity extraction. Additionally, it was also found that both the NLU models are strongly reliant on training data which it may or may not perform well on unseen data. When building the tags data set, it was found that the besides the aforementioned data "dirtiness", tags could come in single words, sentences, and descriptions. This is not ideal for training entities because the model would not be able to differentiate when a group of words are composite words or meant to be separated. For example, the tag "child population and health" could either mean "child population" and "child health" or "child population" and "health". Besides that, the system uses an intent featurizer based on word embedding that do not take into account the ordering of the words in the sentences which would provide incorrect intent classification for input such as "find information about sourcing data techniques". Furthermore, another limitation is that Rasa is a relatively young framework and still in its development stage (v0.8.6 as of writing). As the framework evovles, the affected parts of DataBot would need to be migrated. Currently, there is a bug where 2 packages[5] would always be returned which is unexpected since returned packages are based on search relevancy [46]. Regrettably due to time constraints, integration and performance tests were not able to be carried out, hence effects the system has on the UDL platform's CKAN service is not known. However, there hasn't been any observed noticeable

[5] "Agricultural Land Classification detailed Post 1988 survey ALCC09596u" and "Family lineage and landscape quality data for wild bumblebee colonies across an agricultural landscape in Buckinghamshire, U.K."

negative effects, with at least 3 users. For the same reason, there is a lack of evaluation for the usability and effectiveness of DataBot.

A key assumption made in this project is that users will respond to DataBot in a reasonable manner, without a “troll” attitude. This may be the case in human-to-human interaction but is unlikely in a human-machine interactions as experienced by Microsoft [47]. No software can be correctly built without first assuming that the underlying libraries are correct and then overcoming the issues that are found. Even though Rasa underlying uses well established python libraries, an included assumption is that their implemented classes and interfaces perform as documented which may cause severe issues such as Stories checkpoints were not being concatenated properly [48].

Despite the list of limitations and assumptions identified, I am positive that these could be addressed by future iterations of the project. Indeed, this would be inevitable given the wide scope the project entails, and the time constraint.

Chapter 7

Summary and Future Work

This chapter summarizes the work and evaluations that had been completed. It concludes with a final thoughts for developing the system further.

7.1 Summary

This dissertation focused on the design, implementation and testing of DataBot, a conversational agent for sourcing data sets. A set of MoSCoW requirements were inferred from communication with the project's clients and necessary system attributes iteratively throughout the project. The underlying conversational framework used to implement DataBot was Rasa, where the conversational domain is represented as intents, entities, actions, slots and templates. DataBot utilizes an SVM and CRF for the intent classifier and entity extractor respectively and uses Logistic Regression for the dialogue model. The data obtained to train these models were independently created from UDL sources, using heuristics and experimenter subjectivity. Users interact with DataBot through a web interface that was implemented and resides on the UDL platform. The main functionality of the system is sourcing data with search tags or without the search tags, in such a case DataBot would rectify the solution by asking the necessary questions. Unit tests were carried out on each method and functional tests were carried out for testing Story correctness. The techniques used to evaluate the models are precision, recall and confusion matrices, where appropriate.

It was noted that the dialogue model performs in extremes that is, it either performs perfectly or fails miserably which indicates that it is over fit for the training data. Similar observations were made with the NLU models, however, this is likely due to the lack of training data in some of the labels. The key limitations of the project are there is no support for performing operations on the found packages, the dialogue model is unable to handle dialogues outsides its training data, the system has primitive "natural" capabilities, DataBot always expects the user to initiate conversation, and its likely that many edge case have not been tested. These limitations could be addressed in future iterations of the project. Nevertheless, the project was considered a success because it met all "Must Have" requirements stated.

7.2 Future Work

There are many improvements that can be made to further build on top of the system, including additional functionality and improving the current ones, and those mentioned here are not exhaustive but are are thought of as important to the UDL platform and generally to improve the stability and quality of DataBot.

The most crucial next step is the creation of data reflective of end users to improve the quality of conversations between users and DataBot. Data needs to be gathered from real users using surveys, questionnaires or experiments both for the model and user interface, since all data created so far are based off theoretical principles and assumptions. We want to understand the user set so that input intent sentences would represent them linguistically. For example, domain experts may use much more accurate and technical terminology whereas average users might use more general language. In general, a usability and effectiveness analysis, such as that performed in [37] and in [28], should be carried out to understand how users would interact with a system like DataBot, its pros and cons. A feature that would be interesting to implement on top of the current system is to allow the users to focus onto a single package found by loading in resources from the package source and perform descriptive statistical analysis (min, max, average, frequency, etc.) on it. This would outlay grounds for more sophisticated data exploration and analysis techniques in the future. Besides that, features that are able to rectify a certain level of errors such as spell check-and-correction, and able to reduce the rate of errors such as autocomplete or query suggestion should be explored to improve user friendliness and lessen the friction of adopting DataBot. Lastly, to better understand and gain a more confident intent classification and entity extraction, it would be interesting for future works to augment DataBot's vocabulary with real-world semantics using third-party corpora as well as identify synonyms and resolve abbreviations frequently used to describe data sets found on the UDL Platform.

Since this project focused on getting a working prototype running, many support tools were ignored due to the time consuming activity of learning and setting them up of which would add great value to future extenders. Support tools such as data loggers, health monitors and performance evaluators should be incorporated as there would be significant value in automating the collection of real user data, and the performance evaluation and assessment of DataBot. Dialogues, tracker states, and user history are logged in the system but as data dumps. This should be logged in a meaningful way. Decision policies of how DataBot should respond to unexpected intentions would need to be explored. However, as DataBot collects more dialogue data, this could be used to develop a more generalized model capable of handling unseen patterns at an acceptable rate.

Appendix A

User Interface Analysis

The principles use are described in [40].

Principle	WhatsApp	Gitter
Simple and Natural	Core components are chat list side dock, search bar, center message wall, text bar, conversation name in a top banner, latest messages at the bottom, buttons for searching within the conversation, attaching files and other options. Text bar contains microphone button.	Core components are quick access toolbar that expands into a side dock, center message wall, right side dock to view people currently in the conversation, conversation name in a top banner, and latest messages at the bottom. Text bar contains buttons to switch to typeset mode and markdown info.
Speak the User's Language	Few words to aid the user. No domain specific terms.	Words and short sentences used for some buttons and as section dividers. Some words used are domain specific such as Github, markdown and "Room".
Minimize the User's Memory Load	Buttons such as a clip, magnifying glass, microphone and vertical dots intuitively represent their functions. Messages are arranged on the left if received and right if sent. Received messages are in a white bubble while sent messages are in a green bubble. Each message has a timestamp and blue banner denotes conversation wide events. Uses "Enter" for submit only.	Buttons such as hamburger button, chat bubble and globe are not instantly intuitive. To aid users, each button has a title when hovered over. Sent and received messages are directly stacked on top of each other. User cards can contain more formatted text and often has rich media such as code snippets. Each message has a timestamp. Uses "Enter" for submit only.
Be Consistent	N/A	Instead of omitting options that require high permissions from view, those options are shown but disabled instead.

Provide Feedback	Uses a clock, single tick, double tick and double blue tick to inform the user about the level of reception of his/her message. Blue banner denotes conversation wide events. Loading bar when chats are being loaded.	Informs the user of unread messages and mentions.
Provide Clearly Marked Exits	Exit by closing browser.	Exit by closing browser.
Provide Shortcuts	N/A	Able to format messages using special syntax.
Provide Good Error Messages	Clear error messages and resolution options when chats cannot load.	Non-encountered during evaluation.
Error Prevention	N/A.	N/A

Table A.1: User Interface Analysis based on the 10 principles of usability

Appendix B

Dialogue Confusion Matrix

The figure below shows the confusion matrix obtained after functional testing of each story.

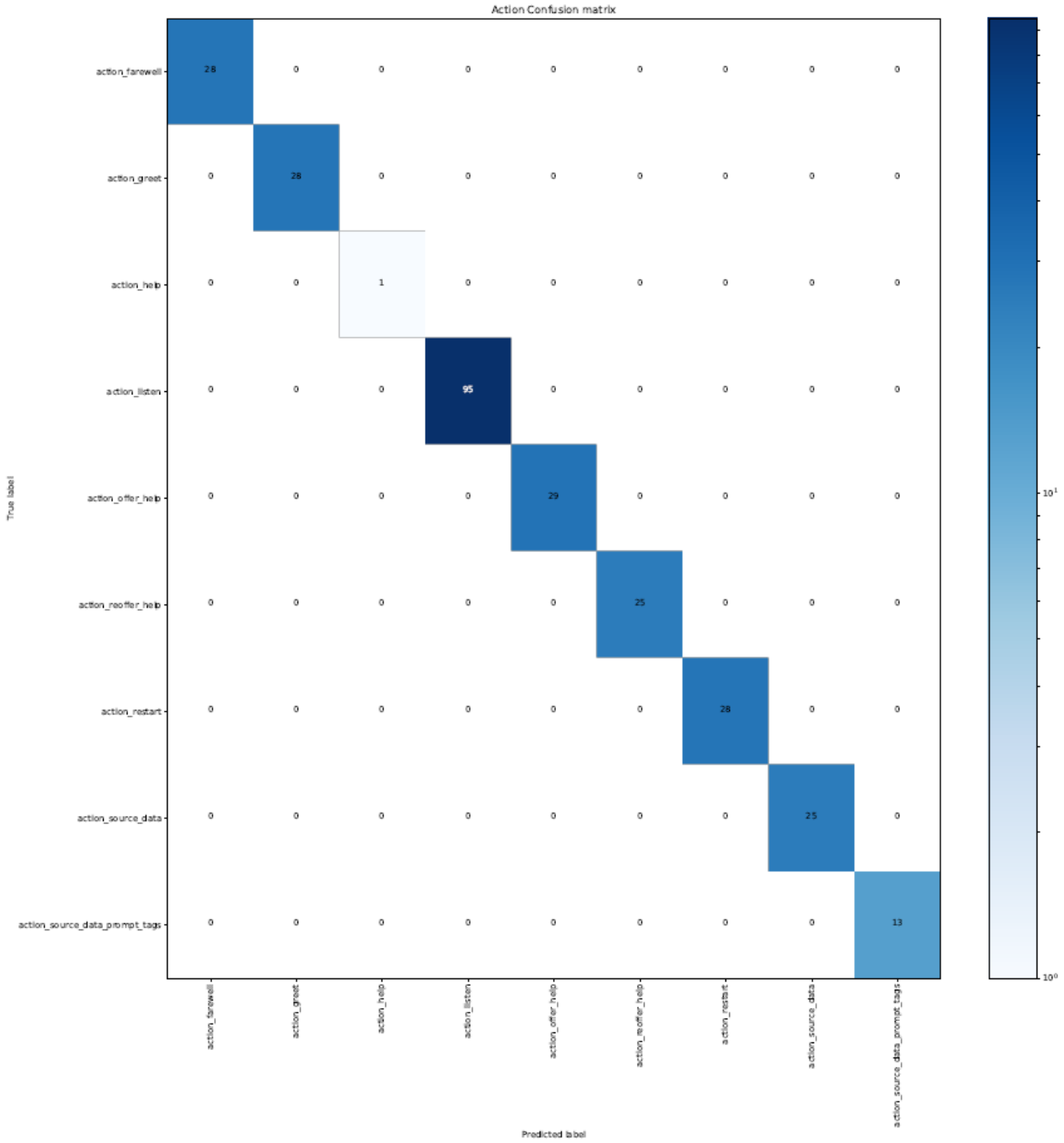

Figure B.1: Dialogue Model Confusion Matrix

Appendix C

Use Case Descriptions

USE CASE (OBJECTIVE)	Source data with Tags
ID	UC1
DESCRIPTION	User intents to source data with relevant search tags
PRIMARY ACTORS	User
SECONDARY ACTORS	None
PRECONDITIONS	Completed UC3
MAIN FLOW	1. User request to source data, providing search tags. 2. DataBot returns 5 results from UDL's database. 3. DataBot reoffers help.
POST CONDITIONS	Context tracker is updated.
ALTERNATIVE FLOWS	A, Starting from stage 1 of main flow. 1. User request to source data, providing search tags and limit. 2. DataBot returns results from UDL's database limited to specified limit or less than specified limit if there are insufficient results. 3. Continues main flow.

USE CASE (OBJECTIVE)	Source data without Tags
ID	UC2
DESCRIPTION	User intents to source data but initially did not provide relevant tags.
PRIMARY ACTORS	User
SECONDARY ACTORS	None
PRECONDITIONS	Completed UC3
MAIN FLOW	1. User request to source data, without providing search tags. 2. DataBot prompts user for search tags.

	3. User provides search tags 4. DataBot returns 5 results from UDL's database. 5. DataBot reoffers help.
POST CONDITIONS	Context tracker is updated.
ALTERNATIVE FLOWS	A. Starting from stage 4 of main flow. 3. User provides search tags and limit. 4. Bot returns results from UDL's database limited to specified limit or less than specified limit if there are insufficient results. 5. Continues main flow.

USE CASE (OBJECTIVE)	Greetings
ID	UC3
DESCRIPTION	User greets DataBot to begin session.
PRIMARY ACTORS	User
SECONDARY ACTORS	None
PRECONDITIONS	None
MAIN FLOW	1. User greets DataBot. 2. DataBot greets User. 3. DataBot offers help
POST CONDITIONS	Context tracker is created for user session ID.
ALTERNATIVE FLOWS	None

USE CASE (OBJECTIVE)	Query DataBot about its functionality
ID	UC4
DESCRIPTION	User intends is to find out about Databot's functionalities.
PRIMARY ACTORS	User
SECONDARY ACTORS	None
PRECONDITIONS	UC3
MAIN FLOW	1. User queries DataBot about its functionality. 2. DataBot lists its functionality, and describes how each one works. 3. DataBot offers help.
POST CONDITIONS	Context tracker is updated.

ALTERNATIVE FLOWS	None

USE CASE (OBJECTIVE)	Handling ambiguous phrases
ID	UC5
DESCRIPTION	DataBot handles user phrases where intents are not clearly "understood".
PRIMARY ACTORS	User
SECONDARY ACTORS	None
PRECONDITIONS	UC3
MAIN FLOW	1. User phrases some intent. 2. DataBot confirms the understood intention with user. 3. User affirms
POST CONDITIONS	None.
ALTERNATIVE FLOWS	A. Starting from stage 2 of main flow: 2. DataBot confirms the understood intention with the user. 3. User denies. 4. DataBot offers help. B. Starting from stage 2 of main flow: 2. DataBot confidence level is too low, and request user to rephrase sentence. 3. User rephrases sentence.

USE CASE (OBJECTIVE)	Farewell
ID	UC6
DESCRIPTION	User denies any more help.
PRIMARY ACTORS	User
SECONDARY ACTORS	None
PRECONDITIONS	Continuation from Bot reoffering help.
MAIN FLOW	1. User denies. 2. DataBot greets farewell.
POST CONDITIONS	Context tracker is reset, all previous user context data is destroyed.
ALTERNATIVE FLOWS	None

Appendix D

Dialogue Evaluation Confusion Matrix

True Action \ Predicted Action	None	Farewell	Greet	Help	Listen	Offer Help	Reoffer Help	Reset Slots	Restart	Source Data
None										
Farewell			16							
Greet		8	8							
Help			1	15						
Listen	2				76					
Offer Help						24			8	
Reoffer Help							21			
Reset Slots										
Restart									16	
Source Data								7		21

Table D.1: Dialogue Model Evaluation Confusion Matrix

Appendix E

Code Listing

base.html

This file is required to overwrite CKAN's base.html in order to create custom template views.

```
{% ckan_extends %}

{% block styles %}
  {{ super() }}
  {% resource 'rasa/bootstrap.css' %}
  {% resource 'rasa/bootstrap.js' %}
  {% resource 'rasa/jquery-3.3.1.js' %}
{% endblock %}
```

databot.html

```
{% extends 'page.html' %}

{% block styles %}
  {{ super() }}
  {% resource 'rasa/rasa_style.css' %}
  {% resource 'rasa/rasa_master.js' %}


{% endblock %}

{% block meta %}
  {{ super() }}
  <meta name="description" value="My website description" />
{% endblock %}

{% block main_content %}
    {% snippet "snippets/chat_interface.html", note=note %}
{% endblock %}
```

chat_interface.html

```
{# Messaging interface #}
<body class="DataBot">
  <h1 style="text-align: center; font-family: 'Times New Roman', Times, serif;
margin-bottom: 0;">DataBot</h1>
  <h3 style="text-align: center; font-family: 'Times New Roman', Times, serif;
margin-bottom: 3vh;margin-top: 0;">Conversational System for sourcing data
from UDL Platform.</h3>
  <div class="container" id="outer-wrapper">
    <div class="chatbox">
      {% if note %}
        <div class="alert alert-primary" role="alert">{{ note }}</div>
      {% endif %}
      <ul id="chat-start"></ul>
      <ul id="chat-loading"></ul>
    </div>
    <!-- Input Bar -->
    <form class="input-form" onsubmit="submit_query(this); return false"
autocomplete="off">
      <div class="input-group mb-3"  id="chat-input">
        <input type="text" class="form-control user-input" id="user-input"
placeholder="New message..." autofocus >
        <div class="input-group-append">
          <input class="btn btn-outline-dark" type="hidden">
        </div>
      </div>
    </form>
    <!-- End Input Bar -->
  </div>
</body>
```

rasa_master.js

```
var chatbox = $(".chatbox")
var chat_start = $("#chat-start")
$("#user-input").val('')
chatbox.animate({scrollTop: chat_start.height() }, "slow")
BASE_URL = document.location.href

function submit_query(form){
  /*Entry from DOM*/
  route = "/user/message"
  url = BASE_URL + route
  text = form.elements["user-input"].value

  if (!is_malicious_text(text)) {

    _append(text, true)
```

```
    chatbox.animate({scrollTop: chat_start.height() }, "slow")

    $(document).ajaxStart(function () {
      // Loader goes in here

    }).ajaxStop(function () {

    })

    $("#user-input").val('')
    // Make AJAX request. Check that not we are not waiting for AJAX request,
otherwise void input
    response = send_user_message(url, "POST", text).then(
      bot_response,
      handle_request_error
    )
  }
}
function is_malicious_text(text) {
  if (text.length < 2){
    _append("Message should be atleast 2 characters long. Plesae avoid using
abbreviations.")
    return true
  }

  return false
}



// Bot responses

function handle_request_error(response) {
  console.log(response)
  _append("We're really sorry, DataBot couldn't process your message. This
means the Rasa extension that hosts databot is down. Please report this to
system administrators.", false)
}

function bot_response(response) {
  /**
   * Handles the returned data recursively. Data object
   * can either be of type "string", "list", or "*_object".
   * Each custom "*_object" type needs its own handler
   *
   * response: Data object
   *
   */
  response = response["bot"]
  console.log(response)
  if ($.isEmptyObject(response)) {
    _append("No data received from CKAN server. Sorry about that!")
    return
  }
```

```
  handle_stack = [response]
  while (handle_stack.length > 0) {
    resp = handle_stack.pop()
    if (resp["type"] == "string") {
      _append(resp["data"], false)
    }
    else if (resp["type"] == "list") {
      for (i = resp["data"].length - 1; i > -1; i--) {
        resp["data"][i]["index"] = i
        handle_stack.push(resp["data"][i])
      }
    }
    else if (resp["type"] == "source_data_object") {
      message = format_source_data_object(resp)
      _append(message, false)
    }
    else {
      _append("CKAN controller returned a format that's undecipherable. Sorry
about that!", false)
    }
  }
}

function format_source_data_object(data) {
  /**
   * Converts a SourceData object into a string
   *
   * data: SourceData
   */
  index = data["index"]
  organization = "Organization: " + data["org"]
  num_of_resources = "No. of resources: " + data["num_of_resources"]
  title = "Title: " + data["title"]
  message = index + ". " + title + "<br>" + organization + "<br>" +
num_of_resources
  return message
}


function send_user_message(url, methodType, text) {
  /**
   * Sends an AJAX request to url with the method:methodType
   * and payload:text. AJAX uses Promises.
   *
   * url: Str
   * methodType: Str
   * text: Str
   */
  data = {
    "text" : text
  }
  console.log("Sending message to: " + url)
  return $.ajax({
    url: url,
```

```
    method: methodType,
    data: JSON.stringify(data),
    contentType: "application/json",
    dataType: "json",
    timeout: 5000
  })
}


// Format incoming messages
function create_message_object(text, is_human){
  /**
   * Creates Message object
   *
   * text: Str
   * is_human: bool
   */

  // Converts \n given in python to breakline suitable for HTML
  text = text.replace(/\n/g, "<br>")
  time = _get_time()
  return {
    payload: text,
    time: time,
    human: ((is_human) ? "human" : "bot"),

  }
}

function _get_time(){
  /**
   * Returns current time in HR:MM {AM,PM}
   */

  date = new Date()
  time = [date.getHours(), ":", date.getMinutes(), " "]
  // Hour correction
  if (time[0] > 12) {
    time[0] = time[0] - 12
    time.push("PM")
  }
  else {
    if (time[0] == "0") time[0] = "12"
    time.push("AM")
  }
  // Minute correction
  if (time[2] < 10) time[2] = "0".concat(time[2])
  return time.join("")

}

function _append(text, is_human) {
  /**
   * Appends an li onto the ul tag with id chat-start.
```

```
    * Text send to this method shouldn't require formatting anymore.
    *
    * text: Str
    * is_human: bool
    */
  message = create_message_object(text, is_human)
  li = construct_li(message)
  $("#chat-start").append(li)
}

function construct_li(message){
  /**
   * Constructs a simple li with from the resulting message object.
   * Sets the li to the appropriate id ("human"/"bot") for styling
   *
   * message: Message
   */
  a = '<li class="message" id="'
  b = '">'
  c = "</li>"
  message = a + message.human + b + message.time + ": " + message.payload +
"<br>" + c
  return message
}
```

rasa_style.css

```
body .DataBot{
  font-size: rem;
  font-family: 'Times New Roman', Times, serif;
}

.input-form{
  background-color: #f2f2f2;
  bottom: 0;
  left: 0;
  width: 100%;
  height: 8%;
  padding: 10px 10px 0px 10px;
  border-bottom-left-radius: 5px;
  border-bottom-right-radius: 5px;
}

.chatbox {
  margin-top: 1%;
  width: 100%;
  height: 88%;
  background-color: #ffffff;
  overflow-y: scroll;
  overflow-x: hidden;
}
```

```
ul#chat-start {
  padding: 0;
  margin: 0;
}

li {
}

.message {
  list-style-type: none;
  border-radius: 5px;
  padding: 0.3em 0 0.3em 0.5em;
  margin: 0.3em;
}

.message#human  {
  background-color: #80ccff;
}

.message#bot  {
  background-color: #f2f2f2;
}

.message > #timestamp{
  font-weight: 600;
}

#outer-wrapper{
  overflow: auto;
  border-radius: 5px;
  height: 600px;
  background-color: #cccccc;
}
```

plugin.py

```
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit
import logging
from ckan.lib.base import BaseController
from data_bot.main.main import dir_path
import os.path as path

class RasaPlugin(plugins.SingletonPlugin): # Inherits PLugin Singleton Class

    plugins.implements(plugins.IConfigurer)
    def update_config(self, config_):
        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        # 'templates' is
        toolkit.add_template_directory(config_, 'templates')
```

```
        # Register this plugin's fanstatic directory with CKAN.
        # Here, 'fanstatic' is the path to the fanstatic directory
        # (relative to this plugin.py file), and 'example_theme' is the name
        # that we'll use to refer to this fanstatic directory from CKAN
        # templates.
        toolkit.add_resource('fanstatic', 'rasa')

    plugins.implements(plugins.IRoutes)
    def after_map(self, map):
        map.connect(
            "databot_user_message", # name of path route
            "/databot/user/message", # url to map path to
            controller="ckanext.rasa.controller:RasaPluginController", #
controller in rasa/controller.py
            action="send_user_message"
        )

        map.connect(
            "data_bot", # name of path route
            "/databot", # url to map path to
            controller="ckanext.rasa.controller:RasaPluginController", #
controller in rasa/controller.py
            action="databot_index"
        )

        return map

    def before_map(self, map):
        return map
```

controller.py

This file contains the classes RasaPluginController (Controller), RasaCoreConnector, ParseConnector, ContinueConnector and VesionConnector and global function _update_slot_and_next_action.

```
import json
import logging
import urllib2
import ckan.plugins.toolkit as toolkit
from ckan.common import request, response, session
from ckan.lib.base import BaseController, render
from ckanext.rasa.action_manager import execute_next_action

logger = logging.getLogger(__name__)

def _update_slot_and_next_action(data, response):
    """
    Sets the necessary data attributes
    """
    next_action = response.get("next_action")
```

```
    slots =(response["tracker"]).get("slots") if response.get("tracker") else
{}
    data["next_action"] = next_action
    data["slots"] = slots
    if response.get("error"):
        data["error"] = response["error"]

class RasaPluginController(toolkit.BaseController):

    def send_user_message(self):
        body = {}
        try:
            request_body = json.loads(request.body)
        except Exception:
            # Didn't get appropriate JSON format
            bot_response = "Inappropriate body format - body must be
application/json"
            logger.info(bot_response)
            body["bot"] = bot_response
            body["error"] = True
            response.body = json.dumps(body)
            return
        if not session.get("sender_id"):
            session["sender_id"] = session.id
            session.save()
        sender_id = session["sender_id"]
        message = request_body["text"]
        version_connector = VersionConnector()
        if not version_connector.query_rasa():
            body["bot"] = ["Rasa server is down"]
            body["error"] = True
        else:
            bot_response = self.rasa_handle_message(message, sender_id) #
Returns a list of responses
            if not bot_response:
                body["error"] = True
                bot_response ={
                    "type": "string",
                    "data" : "DataBot didn't get any response. DataBot server
is probably down."
                }
            body["bot"] = bot_response
        response.body = json.dumps(body)
        return

    def databot_index(self):

        note = ""
        return render('databot.html', extra_vars={"note": note})


    def rasa_handle_message(self, message, sender_id):
        """
```

```
        Handles a message by executing a sequence of actions until an
action_listen is predicted
        """
        parse_connector = ParseConnector()
        continue_connector = ContinueConnector()
        response = parse_connector.query_rasa(sender_id, message)
        data = {}
        _update_slot_and_next_action(data, response)
        response_channel = {
            "type": "list",
            "data" : []
        }
        if data.get("next_action") is None:
            response_channel["data"].append({
                    "type" : "string",
                    "data": data.get("error")
            })
            return response_channel
        next_action = data["next_action"]
        while next_action != "action_listen":
            events = []
            ret = execute_next_action(next_action, data["slots"])
            if ret.get("type"): response_channel["data"].append(ret)
            if ret.get("events"): events = ret["events"]
            if ret.get("error"):
                return ["{}. The UDL team has been notified of the
error.".format(ret["error"])]
            response = continue_connector.query_rasa(sender_id, next_action,
events)
            _update_slot_and_next_action(data, response)
            next_action = data["next_action"]
        return response_channel


class RasaCoreConnector(object):

    port = "5005"
    base_url = "http://localhost:" + port + "/"

    def __init__(self):
        self.response_data = {}

    def _post(self, url, query):
        try:
            request = urllib2.Request(url, data=json.dumps(query),
headers={'Content-Type': 'application/json'})
            f = urllib2.urlopen(request)
            response = f.read()
            return json.loads(response)
        except urllib2.URLError as e:
            return {"error": "URLError: Failed to access Rasa Core server"}


    def _get(self, url):
```

```
        try:
            f = urllib2.urlopen(url)
            response = f.read()
            return json.loads(response)
        except urllib2.URLError as e:
            return {"error" : "URLError: Failed to access Rasa Core server"}

    def _set_response_data(self, response_data):
        self.response_data = response_data

    def query_rasa(self):
        raise NotImplementedError

class ParseConnector(RasaCoreConnector):

    def __init__(self):
        super(ParseConnector, self).__init__()
        self.endpoint = self.base_url + "conversations/{}/parse"

    def query_rasa(self, sender_id, message):
        """
        Query Rasa Core endpoint 'conversation/{sender_id}/parse'.
        """

        if not message:
            raise ValueError("No message passed into query rasa method of
ParseConnector")
        elif not sender_id:
            raise ValueError("No sender id passed into query rasa method of
ParseConnector")
        query = {
            "query" : message
        }
        url = self.endpoint.format(sender_id)
        response = self._post(url, query)
        self._set_response_data(response)
        return self.response_data

class ContinueConnector(RasaCoreConnector):

    def __init__(self):
        super(ContinueConnector, self).__init__()
        self.endpoint = self.base_url + "conversations/{}/continue"

    def query_rasa(self, sender_id, executed_action, events=None):
        if not executed_action:
            raise ValueError("No executed_action passed into query rasa method
of ContinueConnector")
        elif not sender_id:
            raise ValueError("No sender id passed into query rasa method of
ContinueConnector")
        if events == None:
            events = []
        query = {
```

```
            "executed_action": executed_action,
            "events" : events
        }
        url = self.endpoint.format(sender_id)
        response = self._post(url, query)
        self._set_response_data(response)
        return self.response_data

class VersionConnector(RasaCoreConnector):

    def __init__(self):
        super(VersionConnector, self).__init__()
        self.endpoint = self.base_url + "version"

    def query_rasa(self):
        response = self._get(self.endpoint)
        self._set_response_data(response)
        return self.response_data
```

action_manager.py

```
from random import randint
from ckanext.rasa.data_bot.main.main import UDLApiConnector
def greet(**kwargs):

    possible_responses = [
        "Hi, I'm DataBot! Currently, I can source data. Ask me what I can do
for a detailed explanation!\nPlease keep messages as unambiguous as possible -
I'm still under development by very smart geeks.",
        "Hello there, I'm DataBot! Currently, I can source data. Ask me what I
can do for a detailed explanation!\nPlease keep messages as unambiguous as
possible - I'm still under development by very smart geeks.",
        "I'm DataBot. Nice to make your acquaintance. Currently, I can source
data. Ask me what I can do for a detailed explanation!\nPlease keep messages
as unambiguous as possible - I'm still under development by very smart geeks."
    ]
    response = {
        "type": "string",
        "data": possible_responses[randint(0, len(possible_responses) - 1)]
    }
    return response

def farewell(**kwargs):

    possible_responses = [
        "Thanks for using DataBot. Goodbye!",
        "Thanks for using DataBot. See you soon."
    ]
    response = {
        "type": "string",
        "data": possible_responses[randint(0, len(possible_responses) - 1)]
```

```
    }
    return response

def offer_help(**kwargs):
    possible_responses = [
        "What can I do for you?",
        "How can I be of assistance?",
        "What would you like me to do?",
        "What shall we do today?"
    ]
    response = {
        "type": "string",
        "data": possible_responses[randint(0, len(possible_responses) - 1)]
    }
    return response

def source_data( ** kwargs):
    tags = kwargs.get("tags")
    limit = int(kwargs.get("limit")) if kwargs.get("limit") else 5
    udl_api_connector = UDLApiConnector()
    results = udl_api_connector.search_packages(tags, limit)
    response = {
        "type": "list",
        "data": [{
            "type": "string",
            "data": "Searching for datasets that relate to {}, limiting search
to {} results".format(" ".join(tags), limit)
        }]
    }
    response["data"] = response["data"] + results
    return response

def provide_help( ** kwargs):
    function_list = [
        "(1) Source data - Return datasets that are related to a given search
term. An optional limit can be provided to constraint that number of results
returned, defaults to 5. e.g.'Find data relating to population and pollution
2016', 'get child health care policy datasets limited to 20 results.'"
    ]
    f = ",".join(function_list)
    message = "Currently I can: \n{}.".format(f)
    response = {
        "type": "string",
        "data": message
    }
    return response


def reoffer_help(**kwargs):
    possible_responses = [
        "Is there anything else I can do for you?",
        "Do you need further assistance?"
    ]
    response = {
```

```
        "type": "string",
        "data": possible_responses[randint(0, len(possible_responses) - 1)]
    }
    return response

def prompt_tags(**kwargs):
    possible_responses = [
        "Great! What data would you like?",
        "Excellent! What data are you searching for?",
        "Wonderful! What kind of data are you looking for?",
        "Sure, what are the search terms?"
    ]
    response = {
        "type": "string",
        "data": possible_responses[randint(0, len(possible_responses) - 1)]
    }
    return response

def reset_slots(**kwargs):
    response = {
        "events": [{"event":"reset_slots"}]
    }
    return response

def restart( ** kwargs):
    response = {
        "events": [{"event":"restart"}]
    }
    return response

action_map = {
    "action_greet": greet,
    "action_farewell": farewell,
    "action_offer_help": offer_help,
    "action_source_data": source_data,
    "action_help": provide_help,
    "action_reoffer_help": reoffer_help,
    "action_source_data_prompt_tags": prompt_tags,
    "action_reset_slots": reset_slots,
    "action_restart" : restart
}

def execute_next_action(next_action, data):
    """
    Process to execute the next action
    next_action: String
    response_channel: List
    data: Dict
    """

    action = action_map.get(next_action)
    if action is None:
        return {"not_found": next_action}
```

```
    ret_val = action(**data) #Each action shoud return dict with optional keys
{message, events}
    return ret_val
```

main.py

This file contains the class UDLApiConnector and all global configurations used throughout the project.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import logging
import os
import urllib2
import json
import re
import unicodedata
import ckan.plugins.toolkit as toolkit

dir_path = os.path.dirname(os.path.realpath(__file__))
MODEL_PATH = os.path.join(dir_path, "models/dialogue")
INTEPRETER_PATH = os.path.join(dir_path, "models/nlu/default/current")
HOSTNAME = "http://udltest1.cs.ucl.ac.uk/" # config.get('ckan.site_url')
REDIS_HOST = "localhost"
PORT = 6379
DB = 2
QS = "databot"

logger = logging.getLogger(__name__)
all_chars = (unichr(i) for i in xrange(0x110000))
control_chars = ''.join(c for c in all_chars if unicodedata.category(c) ==
'Cc')
control_chars = ''.join(map(unichr, range(0, 32) + range(127, 160)))
control_char_re = re.compile('[%s]'% re.escape(control_chars))

class UDLApiConnector(object):

    hostname = "http://udltest1.cs.ucl.ac.uk/"
    prod = True # When False, uses REST API to search packages
    def __init__(self):
        if not self.prod:
            self.endpoints = {
                "package_search": 'api/action/package_search?q={}&rows={}'
            }
        else:
            self.endpoints = {
                 "package_search": toolkit.get_action("package_search")
            }
```

```
    def _get(self, resource, safe_chars=":/?=&"):
        try:
            url = self.hostname + resource
            url = urllib2.quote(url, safe=safe_chars) if safe_chars else url
            f = urllib2.urlopen(url)
            response = f.read()
            return json.loads(response)
        except urllib2.URLError as e:
            raise urllib2.URLError

    def search_packages(self, tags, limit=5):
        """
        tags: List[String]
        limit: Int
        return: List[Dict]
        """
        if not tags:
            data = [{
                "type": "string",
                "data": "No search tags were detected"
            }]
            return data
        resp = []
        search_limit = limit + 10
        try:
            if not self.prod:
                resource = self.endpoints["package_search"].format(tags,
search_limit)
                response = self._get(resource)
                results = response["result"]["results"]
            else:
                response = self.endpoints["package_search"](
                    data_dict = {
                        "q": tags,
                        "rows": search_limit
                    }
                )
                results = response["results"]
            results = self._filter_num_resources(results, limit)
            results = self._strip_data(results)
            if len(results) == 0:
                results.append({
                        "type": "string",
                        "data": "No results found. You can't access UDL CKAN
Server without being on UCL's network."
                    })
            return results
        except urllib2.URLError as e:
            return[{
                "type":"string",
                "data": "URLError: Failed to access UDL CKAN server"
                }]
```

```
    def _filter_num_resources(self, results, limit):
        # Filter results that have no resource
        res = []
        for i in range(len(results)):
            if results[i]["num_resources"] > 0:
                res.append(results[i])
                limit -= 1
            if limit == 0: return res
        return res

    def _strip_data(self, results):
        # Remove non-printable charcters
        def _remove_control_chars(s):
            return control_char_re.sub('', s)
        res = []
        for r in results:
            d = {}
            d["type"] = "source_data_object"
            d["title"] =  _remove_control_chars(r["title"])
            d["num_of_resources"]=  str(r["num_resources"])
            d["org"] =  _remove_control_chars(r["organization"]["title"])
            res.append(d)
        return res
```

extended.py

This file implements all extended classes including ExtendedAgent, StatisticalMessageProcessor and Statistics which were never used due to last minute code refactoring.

```
"""
Extends rasa_core clases to record data suitable for creating training
data(NLU and Stories),
and gather statistics
"""
from rasa_core.processor import MessageProcessor
from rasa_core.channels.direct import CollectingOutputChannel
from rasa_core.agent import Agent
from rasa_core.dispatcher import Dispatcher
from rasa_core.actions.action import ActionListen
from rasa_core.slots import DataSlot
from rasa_core.events import UserUttered
from rasa_core.tracker_store import RedisTrackerStore
from rasa_core.domain import TemplateDomain
from rasa_core.server import RasaCoreServer
import os.path as path
import json
import csv
import logging

DIR = path.dirname(path.realpath(__file__))
```

```
CONVO_STATES_PATH = path.join(DIR, "bot/states/")

logger = logging.getLogger(__name__)

class ExtendedRedisTrackerStore(RedisTrackerStore):
    def __init__(self, domain, mock=False, host='localhost',
                 port=6379, db=2, password=None, timeout=None):
        self.timeout = timeout
        super(ExtendedRedisTrackerStore, self).__init__(domain, mock, host,
port, db, password)

    def save(self, tracker, timeout=None):
        timeout = self.timeout
        serialised_tracker = RedisTrackerStore.serialise_tracker(tracker)
        self.red.set(tracker.sender_id, serialised_tracker, ex=timeout)

    def retrieve(self, sender_id):
        stored = self.red.get(sender_id)
        if stored is not None:
            self.red.expire(sender_id, self.timeout)
            return self.deserialise_tracker(sender_id, stored)
        else:
            return None

class ExtendedAgent(Agent):

    def _create_processor(self, preprocessor=None):
        # type: (Callable[[Text], Text]) -> MessageProcessor
        """Instantiates a processor based on the set state of the agent."""

        self._ensure_agent_is_prepared()
        return StatisticalMessageProcessor(
            self.interpreter, self.policy_ensemble, self.domain,
            self.tracker_store, message_preprocessor=preprocessor)


class StatisticalMessageProcessor(MessageProcessor):

    def __init__(self, *args, **kwargs):
        self.high_threshold = 0.5
        self.low_threshold = 0.3
        super(StatisticalMessageProcessor, self).__init__(*args, **kwargs)

    def handle_message(self, message):
        # type: (UserMessage) -> Optional[List[Text]]
        """Handle a single message with this processor."""

        # preprocess message if necessary
        if self.message_preprocessor is not None:
            message.text = self.message_preprocessor(message.text)
        tracker = self._get_tracker(message.sender_id)
        self._handle_message_with_tracker(message, tracker)

        # Ask user to repeat if confidence level too low
```

```
        if tracker.latest_message.parse_data["intent"]["confidence"] <
self.high_threshold:
            # Ask the user to repeat message
            tracker._paused = True
            self._set_and_execute_next_action("action_give_up", message,
tracker)
            tracker._paused = False

        else:
            # Predict next message like normal
            self._predict_and_execute_next_action(message, tracker)
            self._save_tracker(tracker)

        # Log tracker into file
        try:
            Statistics.log_state(CONVO_STATES_PATH, tracker)
        except IOError as e:
            logger.exception("Don't have permissions to write to file")

        finally:
            if isinstance(message.output_channel, CollectingOutputChannel):
                return [outgoing_message
                        for _, outgoing_message in
message.output_channel.messages]
            else:
                return None


    def _set_and_execute_next_action(self, action_name, message, tracker):
        """
        Sets the next action to action_name then execute
        :param action_name: String
        :param tracker: DialogueStateTracker
        :return: void
        """
        # this will actually send the response to the user

        dispatcher = Dispatcher(message.sender_id,
                                message.output_channel,
                                self.domain)

        action = self.domain.action_map[action_name][1]
        self._log_slots(tracker)
        self._run_action(action, tracker, dispatcher)
        self._run_action(ActionListen(), tracker, dispatcher)

class Statistics(object):

    @staticmethod
    def log_state(dir_path, tracker):
        filepath = path.join(dir_path, tracker.sender_id + ".log")
        with open(filepath, 'a') as f:
            string = json.dumps(tracker.current_state()) + "\n"
            f.write(string)
```

```
    def _convert_json_into_csv(self, input_filepath, output_filepath):
        data = []
        with open(input_filepath, 'r') as f:
            # convert each line into a csv string
            for line in f:
                line_data = json.loads(line)
                intent_ranking = line_data["latest_message"]["intent_ranking"]
                for intent in intent_ranking:
                    d = dict()
                    d[intent["name"]] = intent["confidence"]
                data.append(d)
        if len(data) > 0:
            keys = data[0].keys()
            with open(output_filepath, 'w') as f:
                dict_writer = csv.DictWriter(output_filepath, keys)
                dict_writer.writeheader()
                dict_writer.writerows(data)

class ExtendedListSlot(DataSlot):
    type_name = "extended_list"

    def __init__(self, name, initial_value=None, value_reset_delay=1):

        super(ExtendedListSlot, self).__init__(name, initial_value,
value_reset_delay)

    def __setattr__(self, name, val):
        if name == "value":
            if self.__dict__.get(name) is None:
                self.__dict__[name] = []
            if val is not None:
                self.__dict__[name].append(val)
        else:
            self.__dict__[name] = val

    def reset(self):
        while len(self.value) > 0:
            self.value.pop()

    def as_feature(self):
        try:
            if self.value is not None and len(self.value) > 0:
                return [1.0]
            else:
                return [0.0]
        except (TypeError, ValueError):
            # we couldn't convert the value to a list - using default value
            return [0.0]

def _configure_logging(loglevel, logfile):
    if logfile:
        fh = logging.FileHandler(logfile)
        fh.setLevel(loglevel)
```

```
        logging.getLogger('').addHandler(fh)
    logging.captureWarnings(True)

class ExtendedRasaCoreServer(RasaCoreServer):

    def __init__(self, model_directory,
                 interpreter=None,
                 loglevel="INFO",
                 logfile="rasa_core.log",
                 cors_origins=None,
                 action_factory=None,
                 auth_token=None,
                 tracker_store=None):

        _configure_logging(loglevel, logfile)

        self.config = {"cors_origins": cors_origins if cors_origins else [],
                       "token": auth_token}
        self.agent = self._create_agent(model_directory, interpreter,
                                        action_factory, tracker_store)


    @staticmethod
    def _create_agent(
        model_directory,
        interpreter,
        action_factory=None,
        tracker_store=None
        ):
                return Agent.load(model_directory, interpreter,
                          action_factory=action_factory,
tracker_store=tracker_store)
```

develop.py

This file implements the utility methods used to automate the model training workflow. The 3 methods defined are train_nlu, train_dialogue and run.

```
import os.path as path
import argparse
import warnings
import logging

from ckanext.rasa.data_bot.main.main import MODEL_PATH, dir_path,
INTEPRETER_PATH
from ckanext.rasa.data_bot.main.extended import ExtendedAgent
from rasa_core.policies.sklearn_policy import SklearnPolicy
from rasa_core.domain import TemplateDomain
from rasa_core.interpreter import RasaNLUInterpreter
from rasa_core.channels.console import ConsoleInputChannel
from rasa_nlu.model import Trainer
```

```
from rasa_nlu.config import RasaNLUConfig
from rasa_nlu.converters import load_data


import rasa_core.utils as utils

DOMAIN_FILE = path.join(dir_path, "domain.yml")
STORIES = path.join(dir_path, "data/stories.md")
NLU_DATA = path.join(dir_path, "data/enhanced-data.json")
CONFIG_PATH = path.join(dir_path, "nlu_config.json")

#INTEPRETER_PATH = path.join(dir_path, "bot/models/nlu/default/current")
logger = logging.getLogger()  # get the root logger


def train_nlu():
    training_data = load_data(NLU_DATA)
    trainer = Trainer(RasaNLUConfig(CONFIG_PATH))
    trainer.train(training_data)
    model_directory = trainer.persist('../models/nlu',
fixed_model_name="current")

def train_dialogue(domain_file=DOMAIN_FILE, model_path=MODEL_PATH,
training_data_file=STORIES):
    agent = ExtendedAgent(domain=TemplateDomain.load(domain_file),
policies=[SklearnPolicy()])
    logger.info("Begin dialogue training")
    agent.train(filename=training_data_file, max_history=3)
    agent.persist(model_path)

def run():

logger.info("======================================================================
======================================")
    logger.info("Rasa process starting")
    interpreter = RasaNLUInterpreter(INTEPRETER_PATH, lazy_init=False)
    agent = ExtendedAgent.load(MODEL_PATH, interpreter=interpreter)
    logger.info("Finished loading agent, starting input channel")
    agent.handle_channel(ConsoleInputChannel())
    dialogue =
agent.tracker_store.get_or_create_tracker("default").as_dialogue()
    utils.dump_obj_as_str_to_file("logs/conversation", dialogue)

logger.info("======================================================================
======================================")

if __name__ == "__main__":
    # Log warnings in a separate folder
    warning_fh = logging.FileHandler("logs/warnings.log")
    warning_logger = logging.getLogger("py.warning")
    logging.captureWarnings(True)
    warning_logger.addHandler(warning_fh)
```

```
    parser = argparse.ArgumentParser(
        description='starts the bot')

    parser.add_argument(
        'task',
        choices = ["nlu", "dialogue", "run"],
        help = "Runs the methods tran_nlu, train_dialogue or run eg. nlu,
dialogue, run"
        )

    task = parser.parse_args().task

    if task == "dialogue":
        train_dialogue()
    elif task == "nlu":
        train_nlu()
    elif task == "run":
        log = logging.getLogger('werkzeug')
        log.setLevel(logging.DEBUG)
        logging.basicConfig(filename='logs/rasa_core.log',level=logging.DEBUG)
        run()
    else:
        warnings.warn("Need to pass either 'nlu' or 'dialogue'")
```

nlu_config.json

```
{
        "pipeline" : [
                "nlp_spacy",
                "tokenizer_spacy",
                "intent_entity_featurizer_regex",
                "intent_featurizer_spacy",
                "ner_crf",
                "ner_synonyms",
                "intent_classifier_sklearn"
        ],
        "path" : "./models",
        "project" : "data-bot",
        "language" : "en",
        "num_threads": 1,
        "max_training_processes": 1,
        "data" : "./data/data.json",
        "log_file":"./log.txt",
        "spacy_model_name": "en_core_web_md"
}
```

domain.yml

```
slots:
  tags:
      type: ckanext.rasa.data_bot.main.extended.ExtendedListSlot
  limit:
      type: text

entities:
  - tags
  - limit

intents:
  - greet
  - farewell
  - affirm
  - deny
  - requestHelp
  - sourceData
  - sourceDataProvideTags

actions:
  - ckanext.rasa.data_bot.main.bot.actions.Greet
  - ckanext.rasa.data_bot.main.bot.actions.Farewell
  - ckanext.rasa.data_bot.main.bot.actions.OfferHelp
  - ckanext.rasa.data_bot.main.bot.actions.SourceData
  - ckanext.rasa.data_bot.main.bot.actions.Help
  - ckanext.rasa.data_bot.main.bot.actions.ClarifyUnderstanding
  - ckanext.rasa.data_bot.main.bot.actions.ReofferHelp
  - ckanext.rasa.data_bot.main.bot.actions.SourceDataPromptTags
  - ckanext.rasa.data_bot.main.bot.actions.ResetSlots
  - ckanext.rasa.data_bot.main.bot.actions.GiveUp

action_factory: remote

templates:
  action_greet:
    - "Hi, I'm DataBot! Currently, I can source data. Ask me what I can do for
a detailed explaination!\nPlease keep messages as unambiguous as possible -
I'm still under development by very smart geeks."
    - "Hello there, I'm DataBot! Currently, I can source data. Ask me what I
can do for a detailed explaination!\nPlease keep messages as unambiguous as
possible - I'm still under development by very smart geeks."
    - "I'm DataBot. Nice to make your acquaintance. Currently, I can source
data. Ask me what I can do for a detailed explaination!\nPlease keep messages
as unambiguous as possible - I'm still under development by very smart geeks."
  action_farewell:
    - "Thanks for using DataBot. Goodbye!"
    - "Thanks for using DataBot. See you soon."
  action_offer_help:
    - "What can I do for you?"
    - "How can I be of assistance?"
    - "What would you like me to do?"
```

```
    - "What shall we do today?"
  action_source_data_prompt_tags:
    - "Great! What data would you like?"
    - "Excellent! What data are you searching for?"
    - "Wonderful! What kind of data are you looking for?"
    - "Sure, what are the search terms?"
  action_reoffer_help:
    - "Is there anything else I can do for you?"
    - "Do you need further assistance?"
```

converter.py

This file implements the convert function, which converts 'raw' NLU data into formatted NLU data. 'raw' NLU data formatting was designed by myself, and was optimized for speeding up creation of NLU data.

```
from pprint import pprint
from json import dump
import argparse
def start_end(text, word):
        ltext = len(text)
        lword = len(word)
        i = 0
        while i < ltext:
                j = 0
                k = i
                while  j < lword and k < ltext and text[k] == word[j]:
                        k += 1
                        j += 1

                if j == lword:
                        return (k-j, k)

                i += 1

        return (0,0)


def convert(filename, output):
        rasa = {
                'rasa_nlu_data' : {
                        'common_examples' : [],
                        'regex_features' : [],
                        'entity_synonyms' : []
                }
        }
        data = []
        count = 0
        with open(filename, 'r') as f:
```

```
                line = f.readline()
                while line:
                        count += 1
                        line = line.lstrip('*').rstrip().split(sep='-')
                        entry = {}
                        print(line)
                        entry['text'] = line[0]
                        entry['intent'] = line[1]
                        entry['entities'] = []
                        for _ in range(int(line[2])):
                                line = f.readline().rstrip().split(sep='-')
                                synonyms = line[2].split(',') if len(line)
== 3 else []
                                tup = start_end(entry['text'].lower(),
line[1].lower())
                                if synonyms:
                                        for i in range(len(synonyms)):
                                                entity = {
                                                        'start' : tup[0],
                                                        'end': tup[1],
                                                        'entity' : line[0],
                                                        'value' :
synonyms[i]
                                                }

        entry['entities'].append(entity)
                                else:
                                        entity = {
                                                'start' : tup[0],
                                                'end': tup[1],
                                                'entity' : line[0],
                                                'value' : line[1]
                                        }
                                        entry['entities'].append(entity)
                        data.append(entry)
                        line = f.readline()
        rasa['rasa_nlu_data']['common_examples'] = data
        print(len(data))
        with open(output, 'w') as f:
                dump(rasa, f)

if __name__ == "__main__":
        parser = argparse.ArgumentParser()
        parser.add_argument("-i", "--input", help="Data file to parse from",
required=True)
        parser.add_argument("-o", "--output", help="Output file to write to.
hint: this will be your data.json", required=True)
        args = parser.parse_args()
        filename = args.input
        output = args.output
        convert(filename, output)
```

ckan_api.py

This file implements the functions used obtain and clean tags from UDL CKAN database. Finally, it produces an output file that resemble 'raw' data as mentioned previously.

```
entities_tags = list(cleaned_data.keys())

import urllib.request as req
import json
import re

HOSTNAME = "http://udltest1.cs.ucl.ac.uk/api/3/action/"

def get_data(url):
    # Returns a dictionary from URL
    read = req.urlopen(url).read()
    return json.loads(read)

def get_tags(limit):
    # Returns a result dictionary. Read ckan.logic.action.get.package_search
in CKAN API Docs
    data =
get_data("http://udltest1.cs.ucl.ac.uk/api/action/package_search?facet.field=[
%22tags%22]&facet.limit={}&rows=0".format(limit))
    return data

def clean_labels(data):
    # Converts non-alphanumericals to whitespace and removes data that contain
non-english words
    import nltk
    words = set(nltk.corpus.words.words())
    import re
    pattern = r"[\w']+"
    new_data = {}
    for k,v in data.items():
        new_key = re.findall(pattern, k) # get only alphanumericals
        incl = 1
        for w in new_key:
            if w not in words:
                incl = 0
                break
        if incl == 1:
            new_data[" ".join(new_key)] = v
    return new_data

def list_of_tags(data):
    # Converts tags into a set of words (Unused)
    for k in data.keys():
        words = k.split()
        s = set()
        for w in words:
          s.add(w)
```

```
    return list(s)


def create_data(entities_tags,
                train_test_split=0,
                input_file="data-raw",
                output_file="enhanced-data-raw",
                test_output_file="test-data-raw"):
    """
    Creates NLU data for sourceData and sourceDataProvideTags.
    Pre-filled sentences must be suitable for converter.py

    entities_tags: List <- list of tags
    """
    # Input file
    with open(input_file, "r") as f:

        l = []
        i, entities_tags_len = 0, len(entities_tags)
        import random

        # While there's still entities tags
        while i < entities_tags_len:

            for line in f:
                # iterate through each pre-filled sentence
                s = ""
                tags_count = line.count("{}")
                k = 0
                args = []
                ts = []

                # Replace each placeholder with a tag from the tags list
                while i < entities_tags_len and k < tags_count:
                    args.append(entities_tags[i])
                    ts.append("tags-" + entities_tags[i] + "\n")
                    i += 1
                    k += 1

                # If the tags list is empty, discard the latest created
sentence
                if i >= entities_tags_len:
                    break
                limit_count = line.count("{limit}")
                limit_dict = {}

                # Replace limit placeholder with a random int if its defined
                if limit_count == 1:
                    limit_dict["limit"] = random.randint(3, 50)
                    ts.append("limit-{}".format(limit_dict["limit"]) + "\n")
                s += line.format(*args, **limit_dict)
                s += "".join(ts)

                # Collect the output in a list
```

```
                l.append(s)
            f.seek(0)


    if train_test_split > 1 or train_test_split < 0:
        print("Can't have a train_test_split must be between 0 and 1")
        return
    elif train_test_split > 0 and train_test_split < 1:
        num_tags = int(train_test_split*len(l))
        train = l[:num_tags]
        test = l[num_tags:]
        train_output = "".join(train)
        test_output = "".join(test)

        # Output file
        with open(output_file, "w") as f:
            f.write(train_output)

        with open(test_output_file, "w") as f:
            f.write(test_output)
    else:
        output = "".join(l)
        # Output file
        with open(output_file, "w") as f:
            f.write(output)
    print("all ok")
```

sourceData templated NLU data

```
*find {} data-sourceData-1
*search for {} about {} {}-sourceData-3
*source {} data -sourceData-1
*source information about {} {}-sourceData-2
*I would like to source data regarding {} {}-sourceData-2
*looking for {} datasets-sourceData-1
*search database for {} data-sourceData-1
*do a database search on {} {}-sourceData-2
*look through the databases for {} {} {} {}-sourceData-4
*query packages relating to {} {} {} {} {}-sourceData-5
*retrieve data concerning {}-sourceData-1
*get statistics regarding{}-sourceData-1
*data about {}-sourceData-1
*search for {} {}-sourceData-2
*{} {} {} data-sourceData-3
*find {} data {} limited to {limit}-sourceData-3
*search for information {} {}  top {limit} results-sourceData-3
*source {} with {limit} results-sourceData-2
*source information on {} take first {} results-sourceData-2
*I would like to source data regarding {} and {} return {limit} results-
sourceData-3
*looking for top {} datasets on {} limited to {limit}-sourceData-3
```

```
*top {} results for database search on {} top {limit} results-sourceData-3
*query database for {limit} results on {}-sourceData-2
*can you get {} data? limit to {limit} -sourceData-2
*can u find {} data?-sourceData-1
*will you find data related to {}-sourceData-1
```

sourceData ‘raw’ data (first 50 lines)

```
*business development-sourceDataProvideTags-1
tags-business development
*county court-sourceDataProvideTags-1
tags-county court
*undeveloped coast-sourceDataProvideTags-1
tags-undeveloped coast
*final budget-sourceDataProvideTags-1
tags-final budget
*state pension age waste management leisure activity-sourceDataProvideTags-3
tags-state pension age
tags-waste management
tags-leisure activity
*traffic regulation electricity-sourceDataProvideTags-2
tags-traffic regulation
tags-electricity
*marine recorder cargo transport herbicide-sourceDataProvideTags-3
tags-marine recorder
tags-cargo transport
tags-herbicide
*cross border mobility, road traffic incident, regional, type specimen-
sourceDataProvideTags-4
tags-cross border mobility
tags-road traffic incident
tags-regional
tags-type specimen
*national institute for health and clinical excellence staff morale play
space-sourceDataProvideTags-3
tags-national institute for health and clinical excellence
tags-staff morale
tags-play space
*waste disposal-sourceDataProvideTags-1
tags-waste disposal
*monkfish south west conservation policy limited to 39-sourceDataProvideTags-4
tags-monkfish
tags-south west
tags-conservation policy
limit-39
*build end charter expanded with 7 results-sourceDataProvideTags-4
tags-build end
tags-charter
tags-expanded
limit-7
*35 results of clover ecosystem assessment-sourceDataProvideTags-3
```

```
tags-clover
tags-ecosystem assessment
limit-35
*bacon budget-sourceDataProvideTags-2
tags-bacon
tags-budget
*second local authority-sourceDataProvideTags-2
tags-second
tags-local authority
```

Bibliography

[1] Stanford University, "CS224n: Natural Language Processing with Deep Learning," Stanford University, [Online]. Available: http://web.stanford.edu/class/cs224n/syllabus.html.

[2] Massachusetts Institute of Technology, "Artificial Intelligence," Massachusetts Institute of Technology, [Online]. Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/.

[3] Stack Overflow, "Stack Overflow," [Online]. Available: https://stackoverflow.com/.

[4] Reddit, "Explain Like I'm Five," Reddit, [Online]. Available: https://www.reddit.com/r/explainlikeimfive/.

[5] Dartmouth College, "The Dartmouth Artificial Intelligence Conference: The Next Fifty Years," [Online]. Available: https://www.dartmouth.edu/~ai50/homepage.html. [Accessed 12 1 2108].

[6] S. James, "Is AI Riding a One-Trick Pony?," MIT Technology Reviews, 29 9 2017. [Online]. Available: https://www.technologyreview.com/s/608911/is-ai-riding-a-one-trick-pony/. [Accessed 12 1 2019].

[7] D. Britz, "AI and Deep Learning in 2017 - A Year in Review," WILDML, 31 12 2017. [Online]. Available: http://www.wildml.com/2017/12/ai-and-deep-learning-in-2017-a-year-in-review/. [Accessed 12 1 2018].

[8] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu and P. Kuksa, "Natural Language Processing (almost) from Scratch," 2009.

[9] J. Mannes, "Salesforce aims to save you time by summarizing emails and docs with machine intelligence," Techcrunch, 11 5 2017. [Online]. Available: https://techcrunch.com/2017/05/11/salesforce-aims-to-save-you-time-by-summarizing-emails-and-docs-with-machine-intelligence/. [Accessed 7 1 2018].

[10] L. Kolodny, "AI-powered virtual assistant, Mezi, pivots to focus on travel," Techcrunch, 15 11 2016. [Online]. Available: https://techcrunch.com/2016/11/15/ai-powered-virtual-assistant-mezi-pivots-to-focus-on-travel/. [Accessed 7 1 2018].

[11] G. G. Chowdhury, "Natural Language Processing," *Annual Review of Information Science and Technology,* pp. 51-89, 2003.

[12] D. Nadeau and S. Sekine, "A survey of named entity recognition and classification," *Lingvisticae Investigationes.,* 2007.

[13] J. Hu, G. Wang, F. Lochovsky, J.-T. Sun and Z. Chen, "Understanding User's Query Intent with Wikipedia," *WWW,* pp. 471-480, 2009.

[14] C. T. Tan, "Natural Language Processing... Understanding what you say," [Online]. Available: http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol1/ctt/. [Accessed 12 1 2018].

[15] N. Indurkhya and F. J. Damerau, "Semantic Analysis," in *Handbook of Natural Language Processing Second Edition*, CRC Press, pp. 93-95.

[16] Scikit Learn, [Online]. Available: http://scikit-learn.org/stable/. [Accessed 8 1 2018].

[17] J. Lafferty, A. McCallum and F. Pereira, "Conditional random fields: Probabilistic Models for Segmenting and Labeling," in *18th International Conference on Machine Learning 2001*, 2001.

[18] D. Y. Liliana and C. Basaruddin, "A Review on Conditional Random Fields as a Sequential Classifier in Machine Learning," in *International Conference on Electrical Engineering and Computer Science*, 2017.

[19] Computer Science Department, Stanford University, "Logistic Regression," [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/LogisticRegression/. [Accessed 29 12 2017].

[20] Computer Science Department, Stanford University, "Softmax Regression," [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/. [Accessed 29 12 2017].

[21] Scikit Learn, "sklearn.linear_model.LogisticRegression," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html. [Accessed 15 2 2017].

[22] D. Meyer, *Support Vector Machines,* 2017.

[23] M. A. Hearst, "Support Vector Machines," *IEEE Intelligent Systems.*

[24] K. Duan, S. S. Keerthi and A. N. Poo, "Evaluation of simple performance measures for tuning SVM hyperparameters".

[25] E. Levin, R. Pieraccini and W. Eckert, "Using Markov Decision Process for Learning Dialogue Strategies," *Acoustics, Speech and Signal Processing,* pp. 201-204, 1998.

[26] O. Vinyals and Q. V. Le, "A Neural Conversational Model," *ArXiv,* 2015.

[27] The Society for the Study of Artificial Intelligence and Simulation Behaviour, "Events: Loebner Prize," The Society for the Study of Artificial Intelligence and Simulation Behaviour, [Online]. Available: http://www.aisb.org.uk/events/loebner-prize. [Accessed 7 1 2018].

[28] N. Radziwill and M. Benton, "Evaluating Quality of Chatbots," *ArXiv e-prints,* 4 2017 .

[29] T. Bocklisch, J. Faulkner, N. Pawlowski and A. Nichol, "Rasa: Open Source Language Understanding and Dialogue Management," 2017.

[30] Rasa, "Rasa," [Online]. Available: http://rasa.com/. [Accessed 16 12 17].

[31] Rasa , "The Rasa dialogue engine," [Online]. Available: https://core.rasa.com/. [Accessed 2 12 2017].

[32] Rasa, "Rasa NLU," [Online]. Available: https://nlu.rasa.com/. [Accessed 16 12 2017].

[33] Rasa, "Training Data Format," [Online]. Available: rasa-nlu.readthedocs.io/en/latest/dataformat.html. [Accessed 12 4 2018].

[34] Rasa, "Stories - The Training Data," [Online]. Available: https://core.rasa.com/stories.html. [Accessed 12 4 2018].

[35] Comprehensive Knowledge Archive Network Association, "About CKAN," [Online]. Available: https://ckan.org/about/. [Accessed 1 1 2018].

[36] M. Galas, 2011. [Online]. Available: https://github.com/mgalas/OpenDataHub/wiki. [Accessed 11 4 2018].

[37] F. Li and H. V. Jagadish, "Contructing an Interactive Natural Language Interface for Relational Databases".

[38] V. Setlur, S. E. Battersby, M. Tory, R. Goddweiler and A. X. Chang, "Eviza: A Natural Language Interface for Visual Analysis," *29th Annual Symposium on User Interface Software and Technology ,* pp. 365-377, 2016.

[39] J. Nielson and R. Molich, "Heuristic Evaluation of User Interfaces," *CHI '90,* pp. 249-256, April 1990.

[40] R. Molich and J. Nilson, "Improving a Human-Computer Dialogue," *Computing Practices,* vol. 33, no. 3, pp. 338-348, 1990.

[41] A. Nichol, "Improve Common Pattern docs," [Online]. Available: https://github.com/RasaHQ/rasa_core/issues/196#issuecomment-367572071. [Accessed 13 4 2018].

[42] YuukanOO. [Online]. Available: https://yuukanoo.github.io/tracy/#/agents.

[43] "Chatito," [Online]. Available: https://rodrigopivi.github.io/Chatito/.

[44] "Chat Bot Magazine," [Online]. Available: https://chatbotsmagazine.com/snips-data-generation-as-a-service-fixing-the-cold-start-problem-for-natural-language-interfaces-4a855e5b3885.

[45] Rasa, [Online]. Available: https://core.rasa.com/http.html.

[46] Apache Solr, "Apache Solr Reference," [Online]. Available: http://lucene.apache.org/solr/guide/7_3/relevance.html.

[47] TechCrunch, [Online]. Available: https://techcrunch.com/2016/03/24/microsoft-silences-its-new-a-i-bot-tay-after-twitter-users-teach-it-racism/. [Accessed 2018 4 16].

[48] jctrouble, "Github," [Online]. Available: https://github.com/RasaHQ/rasa_core/issues/166.

[49] R. C. Schank, "Where's the AI," *AI magazine,* vol. 12, no. 4, p. 38, 1991.