Finally ‘someone’ who knows where to eat: A Decision Tree-Based Restaurant Recommender System

Andrea Scorza
a.scorza@students.uu.nl
6649173

Jos Hens
j.j.a.hens@students.uu.nl
5737222

Luka van der Plas
l.p.vanderplas@students.uu.nl
4119142

ABSTRACT

Goal-based dialogue systems are increasingly more in demand for their ability to facilitate the interaction between humans and digital databases. This paper presents the implementation of one such system, which helps users select a restaurant that matches their preferences. The implementation is based on the DSTC 2 [1] dataset, which contains logs from user conversations with a similar agent. Based on this dataset, a decision tree-based dialogue act classifier was trained, which is compared to both a rule-based and a stochastic baseline classifier. The classifier is used in the implementation of a dialogue system, which uses a limited number of states and a minimal model of dialogue history to manage conversations. Possible improvements of this system are discussed.

KEYWORDS

Dialogue systems; restaurant recommendation; artificial intelligence; decision trees.

INTRODUCTION

We live in an era where amounts of available knowledge are vast and digital archives typically surpass anything a human can process. It is not strange therefore, that part of the focus in artificial intelligence (AI) is on developing systems that facilitate the interaction between humans and digital knowledge. Such systems include dialogue systems.

Dialogue systems driven by AI – amongst which are chatbots – could play a pivotal role in human-computer interaction. This potential is sometimes referred to as the fourth industrial revolution (IR). This term, gaining popularity since 2015 [2], comprises the development of cyber-physical systems and the promise to change the world in a way similar to what the mechanical revolution (1st IR), electrical revolution (2nd IR) and the digital revolution (3rd IR) have meant for society.

This is an assignment that was made for the course ‘Methods in AI Research’ as part of the Master’s degree in Artificial Intelligence.

The goal of this project was to implement a dialogue system for a limited domain using a machine learning-based dialogue act classifier. The implemented system is intended to make restaurant recommendations to the user, based on their preferences.

In order to accomplish the implementation of machine learning, we had access to dialogue data from the second Dialogue State Tracking Challenge (DSTC 2) [1] and trained our system on these data. The system was now able to correctly qualify what was intended by a user in a restaurant recommendation setting. These qualifications will from now on be referred to as dialogue acts. For details of these user utterance data, see the Data section. For the machine learning, we used a technique from the class of decision trees. For details, see the Dialogue Act Classification section.

With the system being able to correctly classify dialogue acts, we wanted to extend its use to new, unseen user input. Then, the system is able to reason about the goal of the user and associate with this goal a knowledge state for this user. From this knowledge state the system knows what information to ask from and give to the user in order to give it a restaurant recommendation based on the user’s preferences. The association of user input and its dialogue act with an appropriate dialogue state was first modelled in a state transition diagram. This state transition model is implemented in code to derive the dialogue state and – closely related to that – the knowledge state from the user. For this dialogue state transition diagram, see the Data section.

The restaurant recommendation system thus consists of three main parts: 1) a machine learning component, which trains the system in classifying dialogue acts, 2) an implemented state transition function, such that the system can reason with obtained knowledge and 3) a simple user interface (UI) that lets a user communicate (giving input and receiving output) with the system in order to get a restaurant recommendation.

DATA

The DSTC 2 dataset

As mentioned earlier, the training dataset contains dialogs between users and a recommendation system from the

Dialogue State Tracking Challenge 2 [1]. This dataset contains 3235 dialogues between a restaurant recommendation system and its users. Dialogues are represented as a dictionary in a *.json* file, which contain transcriptions of user and system utterances (the conversations were originally spoken), alongside various metadata. This metadata included dialogue act classifications.

It stood out that user utterances tended to be short: the mean utterance length was only 3.8. Single word utterances were common, making up 23% of all user utterances. Unfortunately, this means that utterance contain little information to infer dialogue acts. Using these utterances as training data for a dialogue act classifier may lead to a very simplistic model which inflates the weight of certain single words in classifying an utterance. This could lead to the system having difficulties when interacting with more verbose speakers.

As mentioned, metadata about user utterances included the dialogue act. The database uses 15 dialogue act types to distinguish user utterances, including a *null* type which covers unintelligible input.

Initial state architecture

The implemented dialogue system is based on the system from the DSTC 2, which was used to create an initial state architecture. This initial model of the system states is presented in figure 1.

The model as displayed in figure 1 profited from new insights along the way, and constitutes a simpler version of the state flow in the final product. For instance, this model does not include certain utterances that immediately move the system to a particular state, such as 'start over' (which restarts the dialogue) or 'thank you bye' (which ends the conversation).

As can be seen in figure 1, some state transitions required the system to check its model of the conversation domain and the last user utterance. If the conversation model did not yet contain a type of food preference, the system would ask questions in order to get this information. If a user would open with 'I'm looking for Asian food', the system would update its model and no long ask about cuisine, but preferences about area remained unknown. Questions would be asked to the user, to obtain this information and so forth.

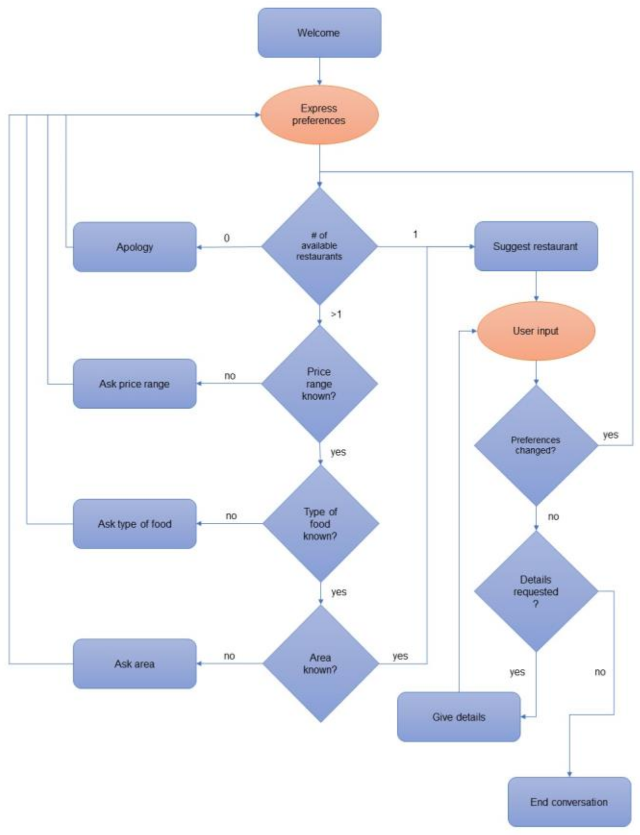

Figure 1: *initial state diagram of the dialogue agent.*

More complex situations arose when the system would know all the user's preferences. The system constructed a list with suggestions along the path of getting more preference information. When all preferences were known, the system would get into its `suggest` state and give a recommendation to the user. As long as the system was in this state and had no reason to go to an `end` state, it would go over its suggestion list again and again until new preferences became available or the user expressed satisfaction with the current recommendation. The latter would be the case in the event of an utterance like 'Thank you, can you give me the address'. The system would then reach a state in which it would reach for termination of the dialog.

DIALOGUE ACT CLASSIFICATION

The dialogue act classifier is fundamental to the program, as it allows the system to make sense of user input. This classifier is implements a decision tree trained on the DTSC 2 dataset. Two baseline systems were implemented to evaluate the final classifier: a keyword-matching function and a stochastic baseline.

The dataset from DTSC 2, as discussed above, was used as the basis for all three classifiers. For the sake of this classification, user utterance transcriptions with their associated dialog acts were extracted from the database. These utterances were then split into a training set of 13,269 sentences and a test set of 2,343 user utterances.

Keyword matching baseline

The first baseline method is an implementation of keyword matching. For this baseline, user utterance transcriptions with their associated labels were extracted from the database. These were manually analysed to identify common keywords that went with a certain dialogue act label.

To avoid making the classifier too specific for the training data, these expressions did not include specific types of restaurants that users mentioned, but focused on the surrounding sentence. For instance, a user saying 'im looking for italian food' would match with the patterns 'im looking for' and 'food', both of which suggest an 'inform' act, but there is no pattern match with 'italian'.

Key patterns, together with the associated act, were represented as regular expressions in a list. The classifier function is fairly simple, going through the keywords one by one and identifying the first one that is a match with the input. The act for that pattern is then presented as output.

Stochastic baseline

The second baseline assigned tags randomly, matching the distribution of tags in the database. The same representation of training data was used as for the keyword matching baseline, i.e. a file containing utterances and their corresponding acts. User utterances were not used for this classifier, however, and only a list of all the acts was used.

The absolute frequency of each act was counted and stored in an array. Frequencies were normalised by dividing them by the total utterance count. Tags were sorted in decreasing order of frequency. Frequencies were then converted to incremental values so that the value for a tag was equal to the sum of the frequencies for that tag and all those preceding it.

For every utterance inside the list, a random number between 0 and 1 was selected, and then matched with the first element in the list that had a larger threshold. The corresponding tag was used as the function's prediction.

Decision tree classifier

Based on initial evaluations of the keyword-based classifier (cf. the final evaluation below), it was decided to use a decision tree to classify dialogue acts, with a bag-of-words representation of user utterances constituting the input features for the decision tree. This approach was chosen because a decision tree evaluating a bag-of-words representation would rely largely on keywords, which are processed in hierarchy of relevance. Since the rule-based baselined functioned adequately, it was seen as positive that the decision tree would work in a similar manner.

To evaluate the classifier during development, the training set from the DTSC 2 data was further split into a training set and a dev set with a 4:1 ratio.

A vocabulary from the training data was established, which consisted of 597 unique words, 266 of which occurred only once.

To represent the user utterances in a format suitable for machine learning, utterances were converted to bag-of-words vectors. The vector of an utterance contained the token frequency within the utterance of each item in the training vocabulary. All words in the vocabulary were included, since the total size of the vocabulary is fairly low and the spare nature of the resulting vectors does not greatly increase computation time when making predictions with a decision tree. Word vectors did contain one more dimension storing the frequency of out-of-vocabulary words, though these did not actually occur in the training data.

The bag-of-word representation of each user utterance was matched with an index representing which dialogue act it was tagged with. These were used as the input and output to train a decision tree classifier using scikit-learn [3]. The resulting tree had a maximum depth of 135. Without restrictions on the tree, accuracy on the development set was 99%. It was therefore decided that no overfitting was happening, and no further restrictions were implemented.

Evaluation of classifiers

Evaluation of the decision tree classifier was performed using the test set that was set apart initially in development. The performance on this test set for the decision tree classifier and the two baseline classifiers is provided in table 1.

Table 1: *accuracy ratio of the decision tree classifier and the two baseline classifiers on the test set.*

Classifier	Accuracy
stochastic	0.256
keyword matching	0.705
decision tree	0.971

As can be seen, the stochastic classifier achieved the worst performance, with only 25.6% accuracy. Note that for the 15 dialogue act tags in the database, this accuracy is still substantially higher than the 7% accuracy that would be expected if the distribution of dialogue acts was uniform. However, the distribution of acts in the training data is much more skewed. *Inform* was the most common class with a probability of 38%, followed by *request* with 27% and *thankyou* with 13%. Other classes had much smaller percentages.

The keyword matching baseline performed adequately with an accuracy of 70.5%, though this is not sufficient for any implementation. Given the discussed prevalence of one-word user utterances in the DTSC 2 dataset, the low accuracy is not surprising. For instance, the following exchange is fairly typical:

SYSTEM:	*what kind of food are you looking for*
USER:	*italian*

Note that in the development of the keyword matching classifier, context-specific terms like ‘italian’ were not used as keywords. As such, many one-word responses from uses would not find a suitable match with the classifier’s list of patterns, and the classifier would return *null*.

The decision tree classifier achieved an accuracy of 97.1%, which is considered sufficient for implementation.

DIALOG MANAGER

The dialogue manager implements the described dialogue act classifier to run a dialogue with the user. In addition to the act classifier the manager also relies on a database of restaurants, which it uses to make suggestions to the user. The UI consists of a simple text exchange between the system and the user, which can be run from a terminal or could be used as the internal code for a graphical interface.

On the surface, the dialogue system has a limited conversation with the user, where it asks the user about their preferred cuisine, area and price range. Depending on parameter settings, users can express their preferences in one or multiple utterances, and can change earlier choices. Once enough information has been gathered, the preferences are used as filters to find a restaurant in the database, and the system makes a suggestion. Users can then ask for more suggestions with the same filters, go back to change their preferences, or ask for contact details about the suggested restaurant.

Internally, the dialogue agent consists of four main components: importing the database of restaurants, processing the user utterances, managing the state of the agent, and generating system utterances. These components are discussed in more detail below. In order to determine the new state and the output, the system models the history of the dialogue in a minimal number of variables, without referring back to earlier user utterances. As the last point of interest, the configurable parameters of the program will be discussed.

Database representation

Information about restaurants is stored in a *.csv* file. This is imported by the system, which stores the information as a dictionary of all the restaurants. The entry for each restaurant is again a dictionary, which contains the restaurant’s entry for each topic. The structure of these restaurant dictionaries is identical for all restaurants.

User utterance processing

The primary component in processing the user output is dialogue act classification. This is done with the decision tree classifier, as described above. User input is converted to a bag-of-words vector to provide appropriate input for the classifier. The resulting dialogue act is often sufficient to determine user intention, but in some cases more information needs to be extracted from the utterance. This concerns two types of information that may be expressed (both of which can be expressed in a variety of acts): restaurant preferences and requests for information.

The first case concerns cases where the user specifies some demand on the type of restaurant they are looking for. These may concern cuisine, area or price. To extract these, a simple keyword search is performed, which checks if words from the restaurant database occur in the user utterance. For instance, in the case of cuisine, the program will look for a match with every type of food that is listed for a restaurant in the database. If the relevant parameter is selected, the program will use Levenshtein edit distance to include near-matches between the database and the user utterance (allowing for a maximum edit distance of 2). In cases where the system just asked the user about a specific topic (such as which area they would like), it will also look for keywords expressing a lack of preference, such as “whatever” or “any”. The output of extracting preferences is a dictionary with the expressed specification for each of the three filters.

The second case of information extraction that may need to be performed, concerns users performing a *request* act, where they ask information about the restaurant suggested by the system. Requested topics are typically address, phone number or postcode, though the system will process requests about other topics (cuisine, area, type) identically. The requested topics are identified with a list of keywords, each of which is matched to the concerning field in the database. The system will output a list of requested topics, so users can ask about any number of details in the same utterance.

State transitions

The dialogue states from the core of the program, as they determine the flow of the dialogue. These are handled in a state transition function, which outputs the next state and updates the dialogue model. While some user acts (such as ‘restart’ or ‘bye’) map directly to the next state, most instances in the use of the program require a more complex mapping that takes into account not only the previous state and the user act, but also other expressed information in the user utterance and the current dialogue model.

The states of the program, which indicate different stages in the flow of the dialogue, can be loosely divided into three classes. First, there are general states, such as ‘welcome’, ‘ask to repeat’, ‘start over’, and ‘end’. With the exception of ‘welcome’ (which is the initial state and cannot be reached later in the dialogue), these can be triggered at any point in the dialogue, and the way they are handled is mostly independent of the dialogue model. For instance, ‘start over’ will be triggered whenever a user asks the system to start over (i.e. performs a ‘restart’ act), and immediately clears the global variables that make up the dialogue model.

The second class of states denote points in the dialogue where the system needs to obtain information from the user about their restaurant preferences. These include three states named ‘ask _ preference’, where the system will ask the user for their preference in cuisine, area or price. Additionally, there is an ‘ask confirmation’ state, where the system will ask the user to confirm some preferences they voiced earlier. This state will be triggered if the system stores anything in its *to_confirm* global variable and will either clear the variable or move the information to the final preferences when the user respectively denies or affirms their choice.

From any of the ‘ask _ preference’ states or the ‘ask confirmation’ state, the system can go into several new states. First, it can go to ‘ask to repeat’ if the user used an unexpected dialogue act or voiced no discernible preferences in an ‘inform’ act. Second, the system can ask for confirmation. The system will always require confirmation if the user overrides an earlier choice, or there was a nonzero edit distance between the preference the system inferred and the corresponding word in the user utterance. If no further confirmation is necessary, the system can either continue to ask for preferences or move on the suggestion phase (discussed below). Depending on parameter settings, the system will wait with making suggestions until preferences are given on all three topics, or it may make them earlier on.

If enough preferences are given, the system will move on to the suggestion phase, which is conducted entirely with the system in the ‘suggest’ state. The system will be in this state when it makes restaurant suggestions, but also when it provides information about its current suggestion.

Output generation

The output string of the system is based on the current state and the dialogue model. Some states map directly to an output string, others require checking one or more of the variables that make up the dialogue model.

The generation of the ‘suggest’ state is the most complex, since the system needs to determine if it has to make a new suggestion, repeat the current one (this is generally avoided) or provide information about the current suggestion. Because making suggestions can be triggered by several situations in the state transition, the output generation will calculate a suggestion and store it in the dialogue model. With this case as the exception, the output generation function does not affect the global dialogue model.

Output strings are generated using simple string concatenation, based on templates. Output is normally deterministic, with the exception of confirmation words that the system uses when asking the user about their various preferences (e.g. ‘Lovely! What kind of area are you looking for?’), which are chosen randomly from a list. The use of string concatenation is somewhat limited and occasionally results in minor errors in punctuation or the use of *a/an* as a determiner (e.g. ‘Are you looking for a italian restaurant?’). Nonetheless, it was found to work sufficiently for the limited dialogue that the system is designed for.

Dialogue model

To keep track of the conversation history and inform the state transition and output generation, the system uses a few global variables. It keeps records of the following pieces of information:

- The preferences of the user. This is the system’s conclusion on what preferences the user has expressed over the entire conversation (not just their last utterance). These can be changed by the user when they express new ones. The preferences are used as filters when making a restaurant suggestion.
- Preferences that the user has expressed, but has yet to confirm.
- The current suggestion of the system. This is stored when the system makes a suggestion, and is thus intended to denote a suggestion that the user and system are mutually aware of.
- A set of topics regarding the current restaurant suggestion on which the user has requested information. Note that this is only applicable if a suggestion is stored as well.

Program parameters

Several binary parameters are implemented in the system:

- *Always ask confirmation:* If this parameter is set to True, the system will ask for confirmation whenever it infers some preferences from a user utterance. Otherwise, it will only do so if it found a nonzero edit distance between the utterance and the database match, or if the user overrides an earlier choice.
- *Use Levenshtein distance:* If the user makes an ‘inform’ act and the system cannot find a direct match with anything in the restaurant database, this parameter states that the system can do another search, this time allowing an edit distance of 2 between a word in the utterance and a possible match in the database.
- *Start suggesting ASAP:* This parameter, when set to True, determines that when the system is updating user preferences, it will immediately check the database for the number of matches. If there is only one match for the current set of preferences, the system will make a suggestion, even if not all preference fields are filled in.
- *Start suggesting immediately:* If set to True, this parameter will override the previous one. The system will always make a suggestion after updating the current set of preferences, regardless off the number of options or the number of fields that are filled in.
- *Force one by one:* Turning on this parameter forces the user to express their preference on only one topic at a time. In practice, this means that the system will only process the user’s preference on a single topic, (namely the one the system asked about), regardless of what other details they may provide in the same utterance. If this parameter is turned off, the system will process whatever preferences it can infer from the user utterance, allowing users to express their preference on two or three topics at the same time.
- *Max utterances:* This parameter limits the length of the dialogue, restricting it to 5 user utterances. If the parameter is turned on, the system’s main loop will update and print a countdown of how many user utterances are allowed. If the counter reaches 0, the system will end the dialogue, regardless of its current state or the last user act.
- *Text to speech:* When this parameter is turned on, the system will use the pyttsx3 module to read utterances aloud, in addition to printing them.
- *Speech to text:* Similar to the *text to speech* parameter, this option allows speech-to-text processing of the user input.
- *Verbose:* This parameter is primarily intended for debugging. If it is turned on, the system will print the evaluation of several global variables after each system utterance.

Parameters are implemented as global variables in the program's main module.

DISCUSSION

The dialogue act classifier performed well in development, so different types of classifiers were not tested. Since the current classifier already achieved 99% accuracy on the development set, it was decided that any further improvements could not be seen as significant. However, actual use may show that user data are not always similar to the DSTC 2 dataset. In this case, it may be necessary to re-evaluate the classifier and train one that is more robust towards new data. For instance, it may be necessary to limit the depth of the tree.

Evaluating the dialogue system is more complex, since it has no easy measure of accuracy. As such, the system as presented is only intended as a prototype for user evaluation. At this stage, there are certain limitations to the program that would benefit from more development. For instance, as mentioned above, string concatenation to generate system utterances often results in minor punctuation and phonotactic errors. These do not inhibit the effectiveness of the system’s communication, but may affect the effect that the system invokes in the user.

The short memory of the system and its simple representation of conversation history simplify implementation, but make certain situations harder to deal with. In particular, cases where the system needs to ask the user for clarification or where users negate certain options can be hard to deal with, since the agent cannot backtrack through the dialogue to see what was said last turn.

Because every combination of state, input and model is handled through a series of if-statements, edge cases (such as a user using an unexpected dialogue act in the current context) are often handled poorly, and they are easily missed in development. Initial user testing may reveal points of improvement.

It should be pointed out, however, that accounting for these edge cases often increases the complexity of the code, while presumably accounting for a very small percentage of actual usage instances. This problem increases for broader domains, as the possible combinations of state, user input and dialogue model will increase exponentially with the size of the system. For systems with a broader function, a rule-based state transition function may not be viable.

Nonetheless, there are some possible extensions to the system that should work well with the given architecture. As a minor adaptation, the system might use multiple restaurant databases for different cities, and start by asking the user about what city they want to eat in. Similarly, the information on restaurants in the database could be expanded by adding more fields. For instance, it could be useful if the system could provide information on restaurant opening times.

Another expansion to the system is to allow spoken interaction with the user. Existing modules of both text-to-speech and speech-to-text software are already implemented, but require more user testing. Besides issues in software compatibility, using speech-to-text and text-to-speech software may alter the desired flow of the dialogue, requiring more confirmations from both the user and system.

DISTRIBUTION OF LABOUR

During this project, all members of the group took part in every decision. We approached every problem in a democratic way, and a consensus was reached on all major decisions.

A specification of all individual contributions can be found in table 2 in Appendix A.

It is a difficult task to divide the time and the work carried out during these past weeks since everybody participated and helped the other members in every part of the project. When one person was coding the others helped with documentation on the methods to use, or helped to debug and to test the code. Hence the list above mentioned has to be considered as an approximation or simplification of the division of the tasks performed by the group.

REFERENCES

1. Dialog State Tracking Challenge. dstc2_traindev.tar.gz and dstc2_test.tar.gz. Retrieved October 7, 2019 from http://camdial.org/~mh521/dstc/
2. K. Schwab. 2015. The Fourth Industrial Revolution. Retrieved October 7, 2019 from https://www.foreignaffairs.com/articles/2015-12-12/fourth-industrial-revolution
3. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.

APPENDIX A

Table 2: *Specification of the individual contributions of group members to the project.*

	Task	**Contributors:**	**Total time spent (summed over contributors):**
Part 1a	Writing a script for the extrapolation of data	Andrea	1.00h
	Writing a script for the cycling through folders	Luka	1.00h
	Putting the scripts togeter	Andrea (50%) Luka (50%)	2.00h
	Designing the state transition diagram	Jos (50%) Luka (50%)	4.00h
Part 1b	Implementation of the keyword matching baseline	Jos (30%) Luka (70%)	6.00h
	Implementation of the random assignment baseline	Andrea (90%) Luka (10%)	4.30h
	Implementation of the machine learning classifier	Luka	1.00h
Part 1c	Writing a lookup function for the retrieval of restaurants	Luka	1.00h
	Designing the program (theoretically) using our state transition function	Andrea (33%) Jos (33%) Luka (33%)	3.00h
	Implementation of a dialog system using our state transition function	Luka	4.00h
	Designing an algorithm for recognizing user preferences	Luka	3.00h
	Debugging the final dialog system	Andrea (33%) Jos (33%) Luka (33%)	9.00h
	Writing the report	Andrea (40%) Jos (40%) Luka (20%)	20.00h