Implementing a Restaurant Recommendation System Using an LSTM Network and a State Transition Function - INFOMAIR Part 1

TEAM 19
ISA APALLIUS DE VOS, 5268664
JESSE VAN REMMERDEN, 6936601
DIEGO LIGTENBERG, 4323246
PAUL DUNCKER, 6183522

In this paper, we describe the implementation of a chatbot to help users find a restaurant. Firstly, we explain how the training data is structured and how the state transition system for the chatbot works. Before we implemented this function, we first needed to classify the kind of dialog act for a given user's input. To do this, we implemented two baseline systems and three machine learning algorithms, and compared them to each other. We found that an LSTM network had the best results. In our chatbot implementation, we first classified the sentences and the transition to the correct state. Depending on the state, we retrieved information from the sentence. We used the Jaro Winkler distance to retrieve this information by comparing words in user sentences with unique items from a dataset. With the information retrieved, the system tries to find a restaurant. If multiple restaurants are found, the system will ask the user more information to narrow its search. When no restaurant is found, the system will suggest another restaurant, based on membership set rules. We also implemented an implication system. This system uses rule-based logic to infer if a user's extra requirements hold given the current restaurant. For improvements, we suggest that the output sentences of the system should be generated more dynamically. This way, using our program would give a more human-like experience.

1 INTRODUCTION

Restaurant recommendation is one of the most interesting recommendation problems because of its high practicality, high demand, and rich context [4]. Both industry and academia have been researching user preference, different kinds of restaurant-related attributes, and social or demographic behaviors, in order to build optimal restaurant recommendation systems [12]. This paper will focus on how user preferences, expressed via natural language, can be optimally used to recommend the right restaurants in the city of Cambridge. In order to do this, a computer program needs to be built that understands human conversation. Conversation or dialog is one of the most important forms of communication between society members in almost any part of the world. Teaching machines to understand and generate human-like dialog has been a relevant but challenging task for decades [8]. Dialog systems are invented to let computer-built programs understand such dialogs. Common challenges for building such programs are the consistency of the program's responses, the relevancy of the system's generated output, remembering past utterances, and recognizing when to transition to a new subject. These subtle aspects of human conversations are important reasons why building dialog systems that converse in natural language is a difficult task [8]. Therefore it is no surprise that to this day, many of these challenges are active areas of research [13].

Task-oriented dialog systems such as Siri, Alexa, or Google are a subset of the (general) dialog systems built to solve some of these challenging problems. Instead of trying to understand and produce general human-like conversations, task-oriented dialog systems focus on a specific task that requires some form of human-like conversation (i.e., making a phone call, giving directions). Because these systems do not have to generalize, it is relatively easy to handcraft some sort of dialog state manager that deals with expected user input. This paper aims to implement an effective corpus-based restaurant recommendation system with a task-oriented dialog system for the city of Cambridge. This system consists of four major components: a dialog act recognizer, a dialog state tracker, a dialog policy, and a text generator.

A **dialog act recognizer** recognizes the type of dialog act that belongs to the user's input utterance. By using two different baseline systems and three different machine learning techniques, namely a decision tree, a logistic regression model, and an LSTM neural network, we attempt to match each potential user utterance to its corresponding dialog act. This allows us to build a **dialog state tracker**. This tracker will, at runtime, store the current phase of the program based on the current dialog act, previous inputs, and previous outputs, by respectively the user and the system. The dialog state tracker can store only relevant user data (such as food or location preference) by using frames. A frame is a high-level concept; it contains a set of important user information that belongs to a single subject (i.e. a user food-type frame that contains information about the user's food preferences). Each frame consists of several slots, which can be filled with user input. An example slot could be "specific user food preference". Once a user mentions their food preference in an utterance that also matches a certain dialog act, the slot gets filled with a value corresponding with the user utterance. The current state of the dialog state tracker then gets repeatedly updated with new current states. Given a current state, the **dialog policy** can then decide, via a set of inference based rules, what sort of response the system is allowed to give to the user. The decision for what type of response the system can give is based on the current dialog act, the inference rules, previous user input, and previous system output. After the dialog policy has decided what type of response is appropriate, it is the **text generator** that produces the system's output that matches the current state of the conversation. We have completely hard-coded the text generator that produces these system responses for the scope of this project. Iteratively applying these steps produces a human-like conversation between the user and the system, where the system attempts to give the user a good restaurant recommendation.

In order to build a restaurant recommendation system, we will thus be using three different machine learning techniques: a decision tree, logistic regression, and an LSTM neural network. Secondly, we conduct a corpus analysis and use common natural language preprocessing methods to clean up the data set. Then we build our dialog state manager consisting of a dialog act recognizer, dialog state tracker, dialog policy, and text generator. Thirdly, we implement a state transition manager (Figure 2) that, on a high level, determines what the task-oriented dialog system will do, and finally, we run an error analysis, review our models, and make cautious conclusions based on these analyses.

2 DATA

Two main data files have been used in this project: a file containing information about the restaurants that the system can offer, and a file with labeled dialog acts from the second Dialog State Tracking Challenge to train the classifier to recognize the user's speech acts. The following section will summarize the restaurant database, give more information on the details and use of the dialog act data set, and give an overview of the state transition diagram which was based on the given user-system dialogs.

2.1 Description of the Datasets

The restaurant database contains just over a hundred different restaurants, with each restaurant having nine possible properties. These properties are restaurant name, price range, area, food type, phone number, address, postcode, food quality, and spice level. The first seven properties were given, while the last two (food quality and spice level) were added later. This will be explained in the Reasoning section. Although most of the information about the restaurants is complete, some of them miss some properties such as the address, phone number or the postcode.

The dialog acts dataset contains over 25000 sentences from 3235 dialogs in the restaurant domain. These sentences consist of interactions between an user who wants to find a restaurant and a program that needs to recommend a restaurant that meets the user's requirements. The dialog system uses a turn-based approach, where each utterance by the user is followed by a response from the program. For each of the sentences in the data, a label from one of fifteen possible dialog act labels is given, such as the **inform** label for user sentences that inform the program about a preference, the **request** label for sentences where the user asks for certain information about a restaurant, or the **bye** label for sentences that indicate the end of a conversation. Figure 1 shows the distribution of the labels in the dialog dataset, while Table 1 highlights some statistics of the database.

Table 1. An overview of the dialog database

Shortest sentence	1
Longest sentence	23
Mean length sentence	3.72
Median length sentence	3
Standard Deviation length sentence	2.94
Unique words in corpus	784

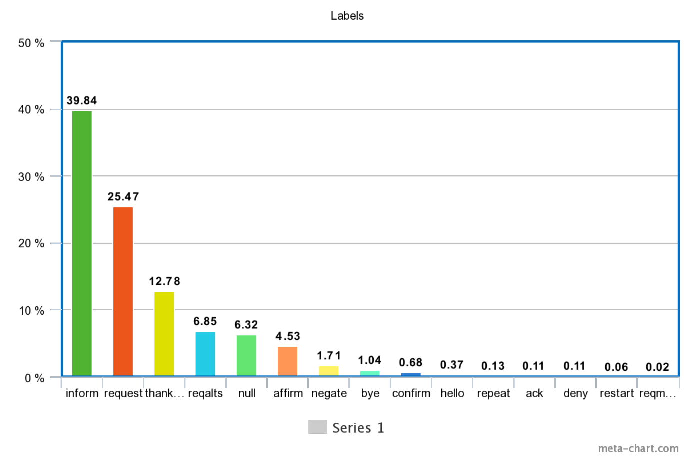

Fig. 1. Distribution of label occurrences in the dataset in percentile

The figure shows that a large amount of the data (40%) is labeled with the **inform** label, followed by the **request** label (25%) and the **thankyou** label (13%). The dataset sentences are generated using automatic speech generation, making the dataset more prone to errors and noise than with written text. The dialogs' language used is quite informal as the dialogs are based on spoken conversations, with a big part of the dataset consisting of single-word sentences and concise interactions between the system and the user. This can be seen in table 1, which shows that the average sentence length is around three to four words. The grammar used by the users generally seems correct, although some sentences in the data appear to be cut off or wrong because of problems with the voice recordings or the written text generation. The clear, simple, and specific content of the conversations results in an easier task for the classification algorithm. Short and common sentences that appear in a specified domain are less difficult to classify than longer and more complicated sentences without any domain restriction.

2.2 The Dialog State Transition Diagram

As was described before, this dialog act dataset was used to train a classifier to recognize new user input for the program made in this project. To model the idea behind our implementation of the user-program interaction, a dialog state transition diagram was made.

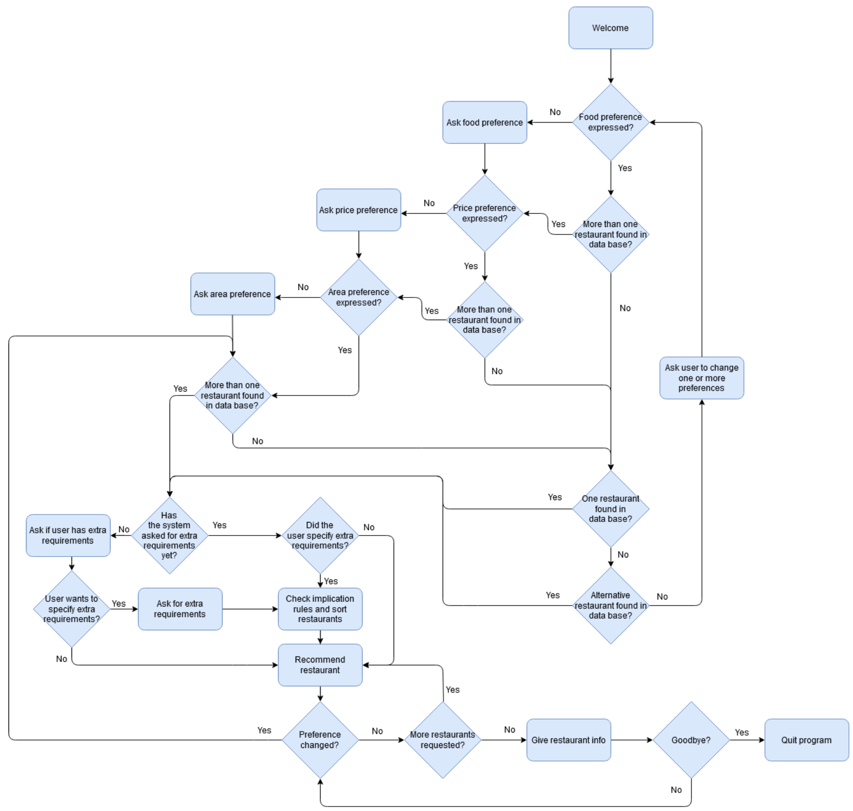

Fig. 2. A dialog state transition diagram of the program

This diagram, shown in Figure 2, shows how the program will react to the user input and in what order it will handle the restaurant recommendation task. The states in the system are based on the user input classification, meaning that when an input sentence is classified as, for example, an **inform** statement, the system's state will be a reaction to that input. This current state will then handle the user input content depending on the current knowledge the system has about the user's preferences and other related variables. We will explain the state transition diagram by walking through the steps the system takes when handling user input. In the initial state, the system welcomes the user and asks them to give their restaurant preferences. If the user has informed the system of one or more preferences, the system checks if there are any restaurants available and asks for more preferences if not all are given, and multiple restaurants are found in the dataset. If the user has not given any preferences, the system should ask about the first unknown preference (in this case, the food type) in order to fill that slot. The questions will continue until all slots are filled or until the system has found that either no restaurants meet the current requirements or exactly one restaurant meets the requirements. The system will then ask once if the user has extra requirements that are not found in the database, and depending on the answer, the system will ask follow-up questions or immediately start recommending a restaurant. Whenever the user indicates that they want to change a preference, the system will check again if restaurants match those new preferences, and if not, it will check if there are similar restaurants. If the user requests information about the restaurant, the system will look it up and give that information. Finally, if the user gives an input that appears to be a goodbye, the system will shut down.

The idea behind the model is based on the given example conversations between users and a recommendation system. Most of these conversations generally had their first part about the user's preferences and the second part about finding the right restaurant, and giving information about that restaurant. Because of this, we modeled our state transitions system in a similar order.

2.3 State Examples and Dialog Snippets

To explain the idea behind the state transition model, we will give a couple of examples with different states. For instance, when the program is started, it will be in the initial state. The system will introduce itself and then ask the user to specify what kind of food they want, in what price range, and in which part of town. After the user has given some input, the system will classify the input sentence and decide on a state depending on this classification. In this example, the input sentence is most likely an **inform** action because the user will tell the system one or more of their preferences. If it is an **inform** action, the system will go into the state for extracting preferences and give an output based on the information that is or is not known.

```
System: Hello, welcome to the Cambridge restaurant system? You can ask for
    restaurants by area, price range or food type. How may I help you?
User: I would like an Italian restaurant in the centre part of town.
System: Would you like something in the cheap, moderate, or expensive price
    range?
```

This example snippet captures the top half of the state diagram because it shows that the program has recognized the user's input as an inform sentence, has gone from the start state into the implicit state for extracting preferences, extracted the type of food and area from the first user sentence, and went to the explicit state for asking missing information, which in this case is the price range. Another example shows how the system handles requests from the user about the currently offered restaurant.

```
System: restaurant one seven is a restaurant that serves british food and in
    the north part of town
User: can i have the phone number and address
System: The phone number of restaurant one seven is 01223 337766
System: restaurant one seven is on de vere university arms regent street city
    centre
User: thank you
System: You're welcome
User: bye
System: Goodbye
```

The bottom part of the state model shows that the system should explicitly request information about the current restaurant if the user asks for it. In representing the state model, it is assumed that the user asks for information about the current restaurant if they have not yet specified a different preference or asked for another restaurant after receiving a restaurant recommendation. Whenever the user requests information about the current restaurant, the system will go into the state that handles requests from the user and find the right data from the dataset to present to the user. The example also shows the thanking state, which is triggered when the user thanks the system. This state is not explicitly represented in the model because it does not really change the conversation state. Finally, the goodbye state is shown, which is triggered when the user says goodbye. After reaching the goodbye state, the program is shut down.

3 MACHINE LEARNING

For our state transition system, we needed to classify the user's input to go into the correct state. To achieve each sentence's classification, we have implemented two baselines algorithms and three machine learning algorithms. In this section, we will discuss how each algorithm functions and compare them. In this comparison, we want to see how each algorithm holds up and which one should be the default one for our implementation.

3.1 Baseline Systems

3.1.1 Most Common. The first baseline system is the most simplistic. The algorithm will always retrieve the most common label in the data set. The retrieved label will always be **inform** because it is the most common label in the dataset. When we tested the system, it scored incredibly awful. The reason is quite simple, because it labeled everything as **inform**. The only good score is the precision of **inform**, being 1.00.

	precision	recall	f1-score	support
ack	0.00	0.00	0.00	0
affirm	0.00	0.00	0.00	0
bye	0.00	0.00	0.00	0
confirm	0.00	0.00	0.00	0
deny	0.00	0.00	0.00	0
hello	0.00	0.00	0.00	0
inform	1.00	0.40	0.57	3826
negate	0.00	0.00	0.00	0
null	0.00	0.00	0.00	0
repeat	0.00	0.00	0.00	0
reqalts	0.00	0.00	0.00	0
reqmore	0.00	0.00	0.00	0
request	0.00	0.00	0.00	0
restart	0.00	0.00	0.00	0
thankyou	0.00	0.00	0.00	0
accuracy			0.40	3826
macro avg	0.07	0.03	0.04	3826
weighted avg	1.00	0.40	0.57	3826

3.1.2 Keyword Matching. The second baseline system we implemented was a keyword matching system — the algorithm functions by looking at every sentence in the dataset and comparing it to the input sentence. The score is then calculated

(1) by dividing the number of words they have in common with the length of the sentence in the dataset. With this, you get a score between 0 and 1. The system then returns the label of the sentence with the highest score.

$$score = \frac{num\_same\_words}{length\_sentence} \tag{1}$$

To test the system, we split the data in a train (85%) and test set (15%). We then calculated the precision, recall, f1-score, and support for each label. We found that labels like **affirm** and **request** scored high in the tests. The reason for this is that each instance of **affirm** and **request** are quite similar. Another interesting label score happens with **reqalts**, with the precision being significantly lower than the recall score. We see this happening with other labels, but the probable reason is low support with those labels. This means that with **reqalts**, it is easy to find them all, but at the cost of a lot of false positives. In conclusion, the keyword matching system is still not an adequate classification system because the average recall and the average precision over the labels are not high enough.

	precision	recall	f1-score	support
ack	0.00	0.00	0.00	2
affirm	0.97	0.99	0.98	182
bye	0.71	0.93	0.81	29
confirm	0.00	0.00	0.00	0
deny	0.50	1.00	0.67	2
hello	0.47	1.00	0.67	7
inform	0.97	0.89	0.93	1670
negate	0.90	0.97	0.94	68
null	0.95	0.73	0.83	310
repeat	0.25	0.50	0.33	2
reqalts	0.39	0.96	0.56	102
reqmore	1.00	1.00	1.00	1
request	0.95	0.98	0.97	941
restart	1.00	0.50	0.67	2
thankyou	0.99	0.95	0.97	508
accuracy			0.92	3826
macro avg	0.67	0.76	0.69	3826
weighted avg	0.95	0.92	0.93	3826

3.2 Preprocessing

We needed to preprocess the machine learning algorithm data because the algorithms can not have string sentences as input. This meant we had to process the string to vectors of numerical values and the labels to integers, without losing information about the sentence in the process. For the Decision Tree (3.3) and Logistic Regression (3.4), we used the same preprocessing algorithms. For the sentences, we used Count Vectors [2]. With this function, you first fit the corpus on the Count Vector, and with each new input, a new set of vectors is returned of which words are present, which are also in the corpus, and how many times they are seen. For the labels, we use one-hot encoding[2]. One-hot encoding transforms each label into an integer. This transformation is a one-on-one transformation, which means you can also do the inverse by transforming the integer label to the string label.

The previous preprocessing is enough for the Decision Tree (3.3) and Logistic Regression (3.4), but for the LSTM network, we needed to use other algorithms. We first use one-hot encoding for the labels and then transform each label to a vector with a length of 15, an index for each of the possible labels. Each element in this vector will be zero, except at the index for which it was previously labeled. We used Keras [3] to handle this. We also used Keras for processing the string sentences. For this, we used the Tokenizer function within Keras to map each word to an encoding. The input for the LSTM needs to be the same size each time, so to do this, we pad each sentence with zeros at the front to ensure that each sentence is the same size as the longest sentence in the dataset.

3.3 Decision Tree

We implemented sklearn's decision tree package for this part of the project[10]. Our decision tree's input variable is a sparse document matrix $\mathbf{X}$, which contains a one-hot encoding representation of the training utterances. Each column $\mathbf{X_j}$ in the matrix represents an independent feature (word), which counts how many times the word occurs in each row of training utterances $\mathbf{X_i}$. The dependent target variable $\mathbf{Y}$, a dialogue act, is a tag that represents the intention of the utterance. These tags are converted (via a LabelEncoder) to integers ranging from 0 to 14. The goal of the decision tree is to predict the dialogue act $\mathbf{Y}$ (an Integer value between 0 and 14) given a new user utterance (a row in our input matrix $\mathbf{X_i}$).

A decision tree is a tree-like graph system that can be used to learn a classification function [7]. The algorithm predicts the value of a dependent variable based on the independent input variables. Hunt gives the following explanation about the structure of a decision tree, and Wiley [6] in one of the earliest papers about decision trees: Let D_t be the set of training records that reaches a node t. If D_t contains records that belong to the same class, let that class be called Y_t, then t is a leaf node labeled as Y_t. If D_t is an empty set or the attribute values are the same, then t is a leaf node labeled by the default class Y_d (the majority class for the parent node D_t). If D_t contains records that belong to more than one class, use an attribute test (this differs for different data types) to split the data into smaller subsets. Recursively apply this procedure to each subset.

For a decision tree algorithm, a major task is to decide which attribute $\mathbf{X_j}$ to split next. Our decision tree uses entropy (entropy measures the homogeneity of a node) to make this decision. Formula 2 shows how entropy is calculated.

$$Entropy(t) = \sum_j p(j|t) log_2 p(j|t) \tag{2}$$

Here p(j|t) is the relative frequency of class j at node t. Entropy ranges from 0.0 (where all tuples belong to one class, which implies the most information gain) to 1.0 (the tuples are split evenly between the classes). Once the Entropy for every leaf coming from a parent node is calculated, the average Entropy gain for splitting a certain parent attribute into x amount of leaves can be calculated with formula 3:

$$GAIN_{\text{split}}(t) = Entropy(p) - \sum_i^k \frac{n_i}{n} Entropy(i) \tag{3}$$

Here p is a parent node (that is split into k partitions). $\frac{n_i}{n}$ is the number of tuples in partition i, divided by the total amount of tuples that reached the parent node. Preferably, for each feature $\mathbf{X_j}$ the entropy is calculated, and then compared to all other features. In the training corpus, however, appear 784 different words. This means that for each layer in the decision tree, all of these 784 impurity calculations would have to be computed, and subsequently, the lowest impurity attribute should be selected to find a global optimum. Because this requires such a significant amount of computational power, our decision tree uses a greedy algorithm, which at each layer, only compares a random subset of all the features $\mathbf{X_j}$ for their impurity scores. After the impurity scores are calculated for each node, all the necessary information is known, and the decision tree can be built. The next figure shows the performance of the model.

	precision	recall	f1-score	support
ack	0.33	0.25	0.29	4
affirm	0.99	0.99	0.99	158
bye	0.95	0.97	0.96	39
confirm	0.90	0.87	0.88	30
deny	0.80	0.67	0.73	6
hello	1.00	0.93	0.96	14
inform	0.99	0.97	0.98	1526
negate	0.95	1.00	0.98	62
null	0.85	0.98	0.91	262
repeat	1.00	1.00	1.00	7
reqalts	0.98	0.92	0.95	252
reqmore	1.00	1.00	1.00	1
request	0.99	0.99	0.99	960
restart	1.00	1.00	1.00	4
thankyou	1.00	1.00	1.00	501
accuracy			0.98	3826
macro avg	0.92	0.90	0.91	3826
weighted avg	0.98	0.98	0.98	3826

To test the performance of the Decision tree, we looked at the Precision, Recall, and F1-score of the model. For almost all labels,the F1-score was higher than 0.95, except for **ack**, **deny**, **confirm** and **null**. An explanation of the relatively low score on these dialogue acts is their low frequency. The features **ack**, **deny** and **confirm** only occur is less than 0.01% of our training data. Because the decision tree is pruned to prevent over-fitting, such low prevalent features are difficult to predict correctly. The **null** feature could have a low F1-score because, by definition, it contains all sorts of "errors" the system does not recognize. These utterances can differ substantially, making it harder to predict with a bag of words representation of the data set.

The label **confirm** is one of the labels that did not score well on the tests, especially with the Recall. For example, the sentence: "*and is this a moderately priced restaurant*" is labeled as **reqalts**. The reason for this error is that the data set contains significantly more sentences labeled as **reqalts**, and quite a lot take the form of this sentence.

3.4 Logistic Regression

The second classification algorithm is logistic regression [14], and to be specific multinomial logistic regression from sklearn [10]. The input, the x variable, is a vector. The vector is a bag-in-words representation of the sentence. The output, the y variable, is not binary as in normal 'plain' logistic regression but is a vector with the classification probabilities of all the possible dialog acts. The input vector of each data instance has n dimensions, where n is the number of words in the training set. The output is a set *J - 1* variables, where J is the number of categories in the dialog act. The following function is weighting the input:

$$z = \sum_{i}^{n+1} (w_{ij} x_i) + b_{ij} \tag{4}$$

Weight w and bias b are first randomly initialized and trained using backpropagation. The logistic regression function maps z to a vector with probabilities between 0 and 1 for each category, the sum of all vectors is equal to 1. The J-th alternative is the baseline choice. The model is specified as:

$$Pr(y_1 = j|x_i) = \frac{e^z}{1 + \sum_{k=1}^{J-1} e^z}, j = 1, ..., J - 1 \tag{5}$$

$$Pr(y_1 = J|x_i) = \frac{1}{1 + \sum_{k=1}^{J-1} e^z} \tag{6}$$

The category with the highest assigned probability will be the category the input is classified as.

	precision	recall	f1-score	support
ack	0.00	0.00	0.00	6
affirm	1.00	0.98	0.99	190
bye	0.95	0.93	0.94	41
confirm	0.88	0.75	0.81	28
deny	1.00	0.67	0.80	6
hello	0.95	0.95	0.95	19
inform	0.97	0.99	0.98	1516
negate	0.99	0.99	0.99	67
null	0.94	0.87	0.91	254
repeat	1.00	0.75	0.86	4
reqalts	0.92	0.97	0.94	245
reqmore	0.00	0.00	0.00	2
request	0.99	0.99	0.99	954
restart	1.00	1.00	1.00	2
thankyou	1.00	1.00	1.00	492
accuracy			0.98	3826
macro avg	0.84	0.79	0.81	3826
weighted avg	0.97	0.98	0.98	3826

The classifier performs quite similarly compared to the decision tree classifier. The F1 score ranges between 0.8 and 1 except for **ack** and **reqmore**, where both precision and recall have a value of 0. This implies that the classifier does not classify a true positive in the test data. A possible explanation is that there were very few **ack** and **reqmore** data instances in the training data, so the algorithm can not generalize well.

3.5 LSTM

For one of the classification algorithms, we chose to implement an LSTM network (Long short-term memory network)[5]. An LSTM network is a recurrent neural network (RNN), which means that the LSTM layer nodes are connected. LSTM functions on the principle that the inputs are not independent of each other and this information should be given to its neighboring node. This is incredibly important with sentence classification because words can have a different definition depending on their use and context.

An LSTM has two inputs and outputs. The first input is from the previous layer or the input itself, while the second input is from the previous node in the same layer. From these two, the outputs are calculated. The first output is the input for the next layer's nodes, and the second output is the information given to the next node in the same layer. Within each LSTM, multiple calculations are done with both inputs, making it computationally heavier than a normal, fully connected neural network. Both the inputs have separate weights, which both are learned during training.

For our architecture, we used an LSTM network [9], because of the simplicity of the data, with most sentences having around four words and the longest sentence containing 23. Using more complex networks, like ELMO [11], would be unnecessary because these networks were mostly designed for more complex datasets. Our model starts with an Embedding layer. This layer turns the words in our sentence to word embedding vectors. After that, the layer is a 1D spatial dropout, which does a 0.3 dropout on the Embedding vectors. The layer after that is the LSTM layer, which consists of 256 nodes. In this layer, a dropout occurs on both the recurrent inputs and the normal inputs of 0.3. The next layers are a normal, fully connected layer of 256 nodes, with a ReLU activation function. After this, a 0.3 dropout happens. The output layer consists of 15 nodes, one node for each label, and uses a softmax function for finding the label. We used the dropout because we discovered that the model did not generalize as well without it, and it improved the performance of the model. We trained this model for 10 epoch and used a batch size of 128. The validation split was 0.2.

	precision	recall	f1-score	support
ack	1.00	0.50	0.67	2
affirm	1.00	0.99	0.99	156
bye	1.00	0.81	0.98	42
confirm	0.97	0.94	0.95	31
deny	1.00	0.75	0.86	4
hello	0.89	1.00	0.94	16
inform	1.00	0.99	1.00	1538
negate	0.97	1.00	0.98	61
null	0.99	1.00	0.98	238
repeat	1.00	1.00	1.00	6
reqalts	0.96	0.99	0.98	263
request	1.00	1.00	1.00	969
restart	1.00	1.00	1.00	1
thankyou	0.98	1.00	0.99	499
accuracy			0.99	3826
macro avg	0.98	0.92	0.95	3826
weighted avg	0.99	0.99	0.99	3826

To test the performance of the LSTM model, we looked at the Precision, Recall and F1-score of the model. We found that the model scored the best compared to the Decision Tree model (3.3) and the Logistic Regression model (3.4). With most of the labels, the F1-score was higher than 0.95, except for **ack**, **deny** and **hello**. The reason why these scores are low is that the support for these labels is low compared to most other labels. The model did not get enough training examples to adequately learn these labels.

Checking the classification on some of these labels, we found that it could classify the input *okay show me the first* correctly as **ack**, but it would classify *okay show me the least* as **null**. The probable reason for that is that word *least* only occurs twice in the corpus. The first time for the wrongly classified sentence as the second time for the sentence: *give me the least*, which has the label **null**. What has probably happened is that first set was in the test data and the second sentence in the train data. The model then reasoned that if the word *least* is in the sentence, it should be labeled as **null**. With the label **bye** we see something similar happen. The sentence *thank you good bye thank you good bye* is classified as **thankyou**, while it should be **bye**. The reason for this is that the sentence *thank you good bye* has the label **thankyou** and the sentence *thank you good bye thank you good bye* is this sentence, but then repeated. It should be noted here that the dataset we use to train the classifiers is a simplified version of the original dataset, because the original dataset included multiple labels per sentence while the used dataset only has one label per sentence. This causes a sentence such as *thank you good bye*, which occurs very frequently in the dataset, to have only one label (**thankyou**) while it originally had two (**thankyou** and **goodbye**).

In the scores for each label, we can see that precision for each label is almost perfect with the exception being the label **hello**. The reason is that some sentences with the label **inform** are labeled as **hello**. One of these sentences is *hello chinese type of food*. The model puts to much emphasis on the first word, which is logical within an uni-directed LSTM. The info of *hello* is given to the rest of the sentence and that is why it is classified as **hello**. This is again also caused by the fact that the dataset only uses one label per sentence instead of multiple labels, resulting in sentences with a **hello** label when they originally had both **hello** and **inform**.

4 DIALOG MANAGER

4.1 State Transition Function

To process the user's input, extract its meaning, and return the right output, we implemented a chatbot class in our program. This class contains the current information about the preferences of the user and other variables and multiple functions to interact with the user. It has a main function for handling user input and classifying which type of input it is, as well as state functions for each of the user input types (such as **inform** and **request**) and supporting functions for asking questions to the user, finding or suggesting restaurants, outputting restaurants, and logical inference. The current state of the system is changed depending on the type of user input, meaning that if the user states that they want to eat European food, this is classified as an **inform** statement, and the system will then extract the preference from the input and ask about a specific unknown preference. Similarly, if the system has classified the input as a request, it will go into the state for informing the user about the current restaurant. For every main state, chosen depending on the user input classification, the program checks the current known information relevant for that state and decides what to output next. For instance, if the user has specified a preference, the system extracts the input sentence's preference, checks whether all the preferences are known, and decides what to do. Similarly, if the user requests another restaurant, the system checks if all the preferences are known and how many restaurants were found before. It then decides if it will recommend another restaurant, state that there is only one restaurant matching the current preferences, or ask about unknown preferences first because the user has not specified them yet. A change of states can be seen in the following conversation:

```
System: Hello, welcome to the Cambridge restaurant system? You can ask for
    restaurants by area, price range or food type. How may I help you?
User: I want an expensive African restaurant in the centre part of town
System: Do you have more extra requirements for the restaurant? Answer 'yes'
    for follow-up questions
User: No
System: bedouin is a restaurant that serves african food, is in the expensive
    price range, and in the centre part of town
```

In the example, the system is in the initial state, after the user informs the system of their preferences, the system goes into the state for extracting preferences, and because all the preferences are known, the system asks for additional requirements. When the user's input is labeled as **negate**, the system checks what the last question to the user was and changes its information about the user. It is now known that the user does not have extra requirements, which means that the system can go into the state for recommending a restaurant. This example holds up to the state diagram (figure 2), where the system checks which preferences are known and then recommends a restaurant after asking about extra requirements. Another situation could be that the user has specified their preferences and now wants to change one of their preferences or request another restaurant.

```
System: la margherita is a restaurant that serves italian food, is in the cheap
     price range, and in the south part of town
```

```
User: i want chinese food instead
System: the missing sock is a restaurant that serves chinese food, is in the
    cheap price range, and in the south part of town
User: can i have more restaurants
System: the lucky star is a restaurant that serves chinese food, is in the
    cheap price range, and in the south part of town
```

In this example, the user has specified earlier that they want cheap Italian food in the south part of town, but now they ask for Chinese food instead. The state model shows that if a restaurant is presented and preference is changed, the system will extract the new preference, search in the database and present a new restaurant if one is found, which can be seen in the example. The system recognizes that the user is informing about their new preference, then goes into the state for extracting preferences, and finally goes into the state for presenting a restaurant that now matches the new preferences. It is assumed here that the earlier specified preferences for price range and area are still the same because the user has not explicitly stated that they wanted to change those as well. If this assumption is incorrect, the user can always explicitly state that they also want to change the other preferences. For requesting more restaurants, the system goes into the state for recommending more restaurants, checks if there are any more restaurants, and then goes into the state for presenting a restaurant, which in this case is a different restaurant with the same set of preference properties.

When handling user input that contains important preference information or request information, we use the Jaro Winkler distance (similar to the Levenshtein distance) to check if specific words in the input are misspelled. This is checked for the terms: food types, price ranges, areas, address, phone number, and postcode. When the edit distance between two words is low, meaning that a word in the input is very similar to a word in the database, it is assumed that it is the same word, and the information is extracted from the input. This makes the system more flexible because it does not completely rely on the spelling of the words in the database when recognizing new user input.

4.2 System Utterance Templates

In our system, we have implemented multiple utterance templates. These templates are sentences in which certain words can be changed or removed depending on the user's input. In this subsection, we will explain how we designed these templates and how they function.

If a user asks for a kind of food that does not exist in the database, the system will output the sentence: *"I'm sorry but there is no restaurant serving* ***@FOOD_ITEM*** *food"*, with ***@FOOD_ITEM*** being the kind of food asked for. We did the same if the user asked for an area that does not exist in the data. The output sentence for this is *"I'm sorry but there is no restaurant in the* ***@AREA_ITEM*** *area"*, with ***@AREA_ITEM*** being the input of the user.

If one or multiple restaurants are found, the system will output this to the user in the sentence: *"****@RESTAURANT_NAME*** *is a restaurant that serves* ***@FOOD_ITEM*** *food, is in the* ***@PRICERANGE*** *price range, and in the* ***@AREA_ITEM*** *part of town"*. This sentence can be shortened, so if the user stated they did not care about ***@FOOD_ITEM***, ***@AREA_ITEM*** or ***@PRICERANGE***, it will not output that part of the sentence. For instance, if ***@AREA_ITEM*** and ***@PRICERANGE*** are set to **DONTCARE**, the output sentence will be *"****@RESTAURANT_NAME*** *is a restaurant that serves* ***@FOOD_ITEM*** *food."*. We implemented this because we wanted the system to give a more realistic output, which should only include the properties of the restaurant that are important for the user.

The output for a request message can take multiple forms, but in essence, it comes down to two templates. One which replies with the requested information and the other telling the user that the requested information is not in the database. For example, if the user requests the phone number and it is available, the system will output *"The phone number of **@RESTAURANT_NAME** is **@PHONE_NUMBER**"*. If the phone number is not available, the system will output *"The phone number of **@RESTAURANT_NAME** is not known"*.

In this section, we discussed the most important system utterance templates. Our code includes a couple more of them, but most of them are similar to the discussed templates.

5 REASONING AND CONFIGURABILITY

5.1 Reasoning

Two reasoning components are added to the system to avoid unnecessary dialog steps and improve the system's usability. The first component helps to avoid unnecessary dialog steps, while the second component requests new additional preferences to make the user's choice more precise.

The first component scans through the restaurant database every time the user states a preference. The system will look up how many restaurants fit the preference description. If multiple restaurants fit the current preferences, the system keeps asking for more preferences to narrow down the query. If only one restaurant fits the description, this restaurant is immediately suggested, and the system asks no further questions. If no restaurant in the database fits the preference description, the system suggests alternative restaurants that are similar in cuisine style, price class, and location. Several predefined sets define this relation. It looks up all the sets that the preference is a member of and adds all the set members as a preference. If a preference is not a member of any set, no suggestions are being given. The sets are predefined as follows:

Cuisine styles:

- {thai, chinese, korean, vietnamese, asian oriental}
- {mediterranean, spanish, portuguese, italian, romanian, tuscan, catalan}
- {french, european, bistro, swiss, gastropub, traditional}
- {north american, steakhouse, british}
- {lebanese, turkish, persian}
- {international, modern european, fusion}

Locations:

- {centre, north, west}
- {centre, north, east}
- {centre, south, west}
- {centre, south, east}

Prices:

- {cheap, moderate}
- {moderate, expensive}

As was explained, the system immediately suggests a restaurant when a preference or combination of preferences only occurs once in the data base:

```
System: Hello, welcome to the Cambridge restaurant system? You can ask for
    restaurants by area, price range or food type. How may I help you?
User: I want a Cuban restaurant
System: Do you have more extra requirements for the restaurant? Answer 'yes'
    for follow—up questions
User: no
System: jinling noodle bar is a restaurant that serves cuban food
```

This saves time because the user does not have to give any more preferences and the system does not have to extract more preferences and search the database multiple times.

Another example shows the system's output when the user has specific preferences that do not match any restaurant in the database:

```
User: I would like a korean restaurant in the south part of town
System: No restaurant found with these specific preferences, our systems
    suggests the following restaurant:
System: little seoul is a restaurant that serves korean food, is in the
    expensive price range, and is in the centre part of town
```

The database apparently does not contain Korean restaurants in the south part of town, but there is at least one Korean restaurant in the center part of town, and because **center** and **south** are in the same location set, the system recommends this alternative restaurant.

When there are no restaurants meeting the current preferences and no alternatives are found, the system will ask the user to change one or more of their preferences:

```
User: I want Turkish food in the cheap price range and in the south part of town
System: No restaurant found, please restate one or more of your preferences.
```

The second additional component adds functionality to the system. It uses inference rules to see whether the user's additional requirements are met. When the user has stated their additional preferences to the system, the system infers if a restaurant meets these preferences based on the inference rules applied to the already available information of the current restaurant. When the system has checked all the inference rules, it will show the user which additional preferences are met and which are not.

The database inference rules and properties are divided into three different groups. The first group consists of rules with antecedents that are restaurant properties that are already in the database. These restaurant properties are food type, area, price range, and two manually added properties: food quality (bad, average, or good) and spice level (mild or spicy). The food quality is randomly assigned to the restaurants in the database, while the spice level is assigned depending on the restaurant's food type, meaning that Indian food is assumed to be spicy, while British food is not. The rules that rely on knowledge already in the database are considered level 1 rules. Rules which require the newly inferred level 1 properties as antecedents are considered level 2 rules with level 2 consequences. The last group of inference rules uses these level 2 properties as their antecedent. Their consequents are level 3 properties. These are the inference rules used in our database:

Level 1:

- If a restaurant is cheap and has good food it is busy.
  $(cheap(x) \land goodfood(x)) \implies busy(x)$
- If a restaurant is Spanish you have to wait a long time.
  $spanish(x) \implies longtime(x)$
- If a restaurant is Italian they have pizza
  $italian(x) \implies pizza(x)$
- If a restaurant serves spicy food it is not recommended for children
  $spicy(x) \implies \neg children(x)$

Level 2:

- If a restaurant is busy you have to wait a long time
  $busy(x) \implies longtime(x)$
- If you have to wait a long time the restaurant is not recommended for children
  $longtime(x) \implies \neg children(x)$
- If a restaurant is busy it is not romantic
  $busy(x) \implies \neg romantic(x)$
- If you have to wait a long time the restaurant is romantic
  $longtime(x) \implies romantic(x)$
- If the restaurant is not recommended for children it is romantic and recommended for serious business
  $\neg children(x) \implies (romantic(x) \land seriousbusiness(x))$
- If a restaurant is moderately or expensively priced and you have to wait a long time it is not recommended for students
  $((expensive(x) \lor moderate(x)) \land longtime(x)) \implies \neg students(x)$
- If a restaurant is not expensive and they serve pizza it is recommended for students
  $(\neg expensive(x) \land pizza(x)) \implies students(x)$

Level 3:

- If a restaurant is recommended for students it is not recommended for serious business
  $students(x) \implies \neg seriousbusiness(x)$

The implication function is implemented in such a way that the user is asked once, after extracting all the necessary preferences and before the restaurant recommendation, if they want to specify extra requirements. If the answer to the question is yes, the system will start asking a sequence of questions about the kind of restaurant the user is looking for. The restaurant is then printed, followed up by the inference sequence for each extra requirement and the final number of requirements that is met. The user can then decide if they want the current restaurant or a different one:

```
User: I want chinese food
System: Would you like something in the cheap, moderate, or expensive price
    range?
User: I dont care
```

```
System: What part of town do you have in mind?
User: I dont care
System: Do you have more extra requirements for the restaurant? Answer 'yes'
    for follow-up questions
User: yes
System: Would you like to go to a busy restaurant?
User: yes
System: Would you like to go to a restaurant where you have to wait for a long
    time?
User: yes
System: Would you like to take your children to the restaurant?
User: no
System: Would you like to go to a restaurant with a romantic atmosphere?
User: no
System: Would you like a place that serves pizzas?
User: yes
System: Would you like to go to a student-friendly restaurant?
User: no
System: Do you want to have a serious talk at the restaurant?
User: no
System: rice house is a restaurant that serves chinese food

System: This restaurant is recommended because:
System: Level 1, Rule 1: [cheap, good food] > busy = True
System: This restaurant is recommended because:
System: Level 2, Rule 5: [busy] > long time = True
System: This restaurant probably doesn't serve pizza
System: This restaurant satisfies 2 out of the 3 extra requirements
```

In this case, the user wants a busy restaurant, they want to wait for a long time, and they are looking for a place serving pizza. For the first two requirements, it is found that the properties of the current restaurant cause it to be busy, which then also means that you have to wait for a long time. The system has not found any specific rules about pizza, so it states that they probably are not served at this restaurant.

There is a possibility that a logical contradiction occurs. If a restaurant is in the moderate price range and you have to wait a long time for your food, that would mean that the restaurant is not recommended for students, based on rule 6 of level 2. However, if that same restaurant serves pizza, it would imply that it is recommended for students, based on rule 7 of level 2. This is a contradiction. The system solves this problem by only deriving the consequent of the first rule. This means that the order of the rules has an impact on the truth values of the properties.

5.2 Configurability

To prepare the system for further research, a couple of configurations have been implemented. These configurations make it possible to change five different settings to test whether changes in these settings impact the user. These five settings are:

- **Jaro Winkler Edit Distance** - this setting allows us to change the recognition threshold for important words that are misspelled. The setting can be changed to a float between zero and one, where a low value means that two words do not need to be very similar in order for the system to see them as the same, and a high value means that the words need to be very similar in order for the system to recognize them as the same. This configuration can show how users deal with how a system easily understands their input or not when written differently.
- **Output in All Caps** - this setting allows us to change whether the system's output is in all caps or not. This configuration might impact how the users would rate the interaction between them and the system, as all caps answers might be seen as unfriendly or not human-like.
- **Delay Answers** - this setting allows us to change the time it takes for the system to print its output. This delay can be zero or any float above zero. This configuration is implemented to see how users might react to a system that immediately answers compared to a system that takes some time to answer. The delay might influence how human or computer-like the user would rate the system.
- **Restart Dialogue** - this setting allows us to change whether we want to give the user the possibility of restarting their conversation with the system. If turned on, the setting will reset the conversation when the user asks to, while it returns an output informing the user that the restart is not possible when it is turned off. This configuration might impact how the user rates the system when they are stuck in a conversation and cannot restart it.
- **Text to Speech** - this setting allows us to change the system's written output to spoken output. When the speech setting is turned off, the user can see the system's output written in the terminal. When the setting is turned on, the text will not appear in the terminal, and instead, the program will say the output to the user. This configuration can influence how the user experiences their conversation with the system because a speaking system can seem more human than a system that prints output on a screen.

6 DISCUSSION

6.1 Discussing Dialog System Approach

6.1.1 Limitations. A downside of a task-oriented dialogue system is that it only works for a certain task. In this case; user utterances that request restaurant information in a very specific manner. For example, utterances of greater length, utterances with conflicting premises, utterances with both useful and noisy premises, or one-liners; are currently handled quite poorly by the system. Because of this limitation, the application we've build is not generalizeable. The following section will explain how a more sophisticated system could be built.

6.1.2 Possible Improvements and Alternative Approaches. In our implementation, the system's text generation part was generated via a rule-based system. This makes the system's responses to the user input quite stale. As we described in the introduction, current research often uses machine learning methods to generate text. Having such a text generator would make our system; more prone to change, less time consuming to make, and easier to update. Due to both knowledge, and time constraints we weren't able to implement this. Future research could take inspiration from Brownman's paper where he describes a method to implement a text generation system that uses machine learning [1].

Another improvement future research could make is to use less explicit system responses to user utterances. Currently our task-oriented dialogue system asks quite some explicit questions, because our frame-based system has a high percentage of mandatory slots to fill. Because our program only runs if these slots are filled, the system starts asking explicit questions fairly quickly, in order to increase the speed, and decrease the difficulty of getting through the program. Unfortunately this decreases the human-like experience the user has. Future research could attempt to have a better mix of both implicit and explicit questions.

6.2 Discussing Dialog Act Classification

6.2.1 Limitations. The main limitations for the classification of dialog acts were caused by two factors: the size of the dataset and the fact that each sentence in the data only had one label.

Firstly, the distribution of the number of occurrences of each label in the data, shown before in figure 1, was quite uneven, with the two most common labels taking up almost two-thirds of the dataset. This imbalance makes it hard for the classifiers to be flexible when recognizing less frequent dialog acts, meaning that it has such little variance in the data for less frequent dialog acts that it will not be able to recognize new ways of saying the same sentence. It also means that if these dialog acts do not appear in the training data, the classifier will not recognize them at all during testing. Although the performance of the best classifier, the LSTM network, was very good, it could probably be even better with a larger dataset or at least a dataset that contains more instances of the less frequent dialog acts.

Secondly, the dataset we used for training and testing our model was limited compared to the original dataset, because the original dataset contained multiple dialog act labels per sentence. This limitation can be quite problematic when users start their sentence with a word or group of words associated with a specific label and then continue their sentence with other information. Two examples of such sentences are *hi i want spanish food* and *thank you good bye*. The first sentence originally had two labels, namely **hello** and **inform**, but only had the **hello** label in the data we used for our classifiers. The second sentence had both **thankyou** and **goodbye** as labels originally, but only had **thankyou** as a label for our dataset. This is problematic, because in these cases the second label would actually have been more useful for our classifiers. Information about a user's preferences holds more value than a simple 'hello', and a 'goodbye' can inform the system that the conversation is over while a 'thank you' does not really add anything to the conversation. Although the inclusion of multiple labels per sentence could have been confusing for the classification of the sentences, the use of only a single label limited the system.

6.2.2 Possible Improvements. In our implementation, the dialogue act classified user utterances based on training data specifically focused on restaurant recommendation requests that were directly asking for restaurant information. Because of this, the system is not generalized for broader user input. If user input got outside the training data scope, the classifier classified the input as **null**, which is not ideal. Future research could combine different sorts of training data to make a better system at generalizing user input.

6.2.3 Algorithm Choice and Alternative Approaches. Although the initially implemented machine learning algorithms were the logistic regression model and the decision tree model, which both did an all right job at classifying most dialog acts, an LSTM network was added later to improve the new user input classification. This network is more complicated and needs more time to train, but it clearly performs better. The LSTM network was chosen because it considers that the words in a sentence are related and should not be seen as individuals. The context around a word impacts its meaning, which is why the LSTM network seemed to be the right fit for this project. Alternative machine learning algorithms could have been variations of the LSTM, for example, the multiplicative LSTM (mLSTM) or GRU, or a different approach such as the 1D Convolutional Neural Network.

REFERENCES

[1] Samuel R. Bowman et al. "Generating Sentences from a Continuous Space". In: *CoRR* abs/1511.06349 (2015). arXiv: 1511.06349. URL: http://arxiv.org/abs/1511.06349.

[2] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] Wei-Ta Chu and Ya-Lun Tsai. "A hybrid recommendation system considering visual information for predicting favorite restaurants". In: *World Wide Web* 20.6 (Jan. 2017), pp. 1313–1331. DOI: 10.1007/s11280-017-0437-1. URL: https://doi.org/10.1007%2Fs11280-017-0437-1.

[5] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[6] Earl B. Hunt. "The nature of concept learning." In: *Concept learning: An information processing problem.* John Wiley & Sons, Inc., 1962, pp. 1–10. DOI: 10.1037/13135-001. URL: https://doi.org/10.1037%2F13135-001.

[7] "Index". In: *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, 2011, pp. 607–629. DOI: 10.1016/b978-0-12-374856-0.00024-9. URL: https://doi.org/10.1016%2Fb978-0-12-374856-0.00024-9.

[8] Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Upper Saddle River, N.J.: Pearson Prentice Hall, 2009. ISBN: 9780131873216 0131873210. URL: http://www.amazon.com/Speech-Language-Processing-2nd-Edition/dp/0131873210/ref=pd_bxgy_b_img_y.

[9] Cory Maklin. *LSTM Recurrent Neural Network Keras Example*. July 2019. URL: https://towardsdatascience.com/machine-learning-recurrent-neural-networks-and-long-short-term-memory-lstm-python-keras-example-86001ceaaebc.

[10] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[11] Matthew E. Peters et al. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL].

[12] Ekaterina Pronoza, Elena Yagunova, and Svetlana Volskaya. "Corpus-Based Information Extraction and Opinion Mining for the Restaurant Recommendation System". In: *Statistical Language and Speech Processing*. Springer International Publishing, 2014, pp. 272–284. DOI: 10.1007/978-3-319-11397-5_21. URL: https://doi.org/10.1007%2F978-3-319-11397-5_21.

[13] Vaishak Suresh, Syeda Roohi, and Magdalini Eirinaki. "Aspect-based opinion mining and recommendationsystem for restaurant reviews". In: *Proceedings of the 8th ACM Conference on Recommender systems - RecSys '14*. ACM Press, 2014. DOI: 10.1145/2645710.2645716. URL: https://doi.org/10.1145%2F2645710.2645716.

[14] Juliana Tolles and William J. Meurer. "Logistic Regression". In: *JAMA* 316.5 (Aug. 2016), p. 533. DOI: 10.1001/jama.2016.7653. URL: https://doi.org/10.1001%2Fjama.2016.7653.

A TEAM MEMBER CONTRIBUTIONS

A.1 Reflection

During the first two weeks of the project, we ran into some troubles with planning and creating an even workload. This is because Jesse's programming skills were at a way higher level than the rest of the group's skills. The first week everyone tried to create the classifiers and get them to work, but only Jesse succeeded in completing the assignment. With lab 1b, most of the group again had troubles with programming, and because Jesse had already started with it, it became harder to understand the code and harder to help. After lab 1b, we discussed this because Jesse felt he had to do all the work. After this discussion, we decided that Jesse did not have to work on the implementation of lab 1c and would only help with debugging and if someone got stuck on the assignment. We did this because for lab 1c you had to understand the previously made code. For the second lab, we want to avoid this and do better planning to balance everyone's workload.

Assignment	Task	Person	Time
Lab 1a	Preprocessing	Jesse van Remmerden	0.2 hour
	Baseline	Jesse van Remmerden	0.5 hour
	Machine Learning	Jesse van Remmerden	4 hours
	Evaluation	Jesse van Remmerden	1.5 hours
	User Input Function	Isa Apallius de Vos	1 hour
Lab 1b	State Transition Diagram - Original Drawing	Diego Ligtenberg	0.5 hour
	State Transition Diagram - Final Version	Isa Apallius de Vos	2 hours
	State Transition Function - Main Part	Jesse van Remmerden	5 hours
	State Transition Function - Additions	Diego Ligtenberg	1 hour
	State Transition Function - Additions	Isa Apallius de Vos	0.5 hour
	Identifying User Preference	Jesse van Remmerden	6 hours
	CSV Lookup Function	Jesse van Remmerden	0.5 hour
Lab 1c	Database Satisfiability and Alternative Suggestions	Paul Duncker	8 hours
	Implication Rules	Isa Apallius de Vos	12 hours
	Configurability	Diego Ligtenberg	4 hours
	Support	Jesse van Remmerden	3 hours
Report	Abstract	Jesse van Remmerden	0.5 hour
	Introduction	Diego Ligtenberg	3 hours
	Data - Description of Dataset	Isa Apallius de Vos	1 hour
	Data - State Transition Diagram	Isa Apallius de Vos	2 hours
	Data - State Examples and Dialog Snippets	Isa Apallius de Vos	2 hours
	Machine Learning - Baseline Systems	Jesse van Remmerden	1 hour
	Machine Learning - Preprocessing	Jesse van Remmerden	1.5 hours
	Machine Learning - Decision Tree	Diego Ligtenberg	2 hours
	Machine Learning - Logistic Regression	Paul Duncker	2 hours
	Machine Learning - LSTM	Jesse van Remmerden	2.5 hours
	Machine Learning - Score Tables	Jesse van Remmerden	1.5 hours
	Dialog Manager - State Transition Function	Isa Apallius de Vos	2 hours
	Dialog Manager - System Utterance Templates	Jesse van Remmerden	2 hours
	Reasoning and Configurability - Reasoning	Paul Duncker	3 hours
	Reasoning and Configurability - Reasoning Example Snippets	Isa Apallius de Vos	1 hour
	Reasoning and Configurability - Configurability	Isa Apallius de Vos	1 hour
	Discussion - Dialog System Approach	Paul Duncker	2 hour
	Discussion - Possible Improvements	Diego Ligtenberg	1 hour
	Discussion - Alternative Approaches	Diego Ligtenberg	1 hour
	Discussion - Dialog Act Classification	Isa Apallius de Vos	1 hour
	Team Member Contributions - Table	Isa Apallius de Vos	1 hour
	Team Member Contributions - Content	Everyone	0.25 hour
Other	Debugging Code	Everyone	So many hours
	Lab 1a - trying to implement baseline and ML algorithms	Isa Apallius de Vos	4 hours
	Lab 1a - trying to implement baseline and ML algorithms	Diego Ligtenberg	4 hours