Spring 2018

Elephant 2000: A Programming Language for Remembering the Past and Building on It

Kerry J. Holmes
The University of Akron, kjh80@zips.uakron.edu

Please take a moment to share how this work helps you through this survey. Your feedback will be important as we plan further development of our repository.

Follow this and additional works at: http://ideaexchange.uakron.edu/honors_research_projects

Part of the Artificial Intelligence and Robotics Commons, and the Programming Languages and Compilers Commons

Recommended Citation
http://ideaexchange.uakron.edu/honors_research_projects/622

This Honors Research Project is brought to you for free and open access by The Dr. Gary B. and Pamela S. Williams Honors College at IdeaExchange@UAkron, the institutional repository of The University of Akron in Akron, Ohio, USA. It has been accepted for inclusion in Honors Research Projects by an authorized administrator of IdeaExchange@UAkron. For more information, please contact mjon@uakron.edu, uapress@uakron.edu.
Elephant 2000: A Programming Language for Remembering the Past and Building on It

Kerry J. Holmes

Department of Computer Science

Honors Research Project

Submitted to

The Honors College
Abstract— Elephant 2000 is a programming language to specify programs that accept user speech as text inputs and outputs speech text. The inputs and outputs are based on Dialogue Act theory which describes several forms of speech outputs, such as requests, questions, and answers. The language also relies on Named Entity Recognition to determine what types of objects a user references. These entities include persons, locations, times and so on. Using these attributes of user speech, a program is able to perform simple rule matching and pattern recognition to respond to input. The result is a programming language with English like syntax that allows a programmer to create a chat bot. The system is backed with a machine learning implementation of a moderately complex chat bot that leverages Sequence to Sequence and Long Short-Term Memory techniques. This allows programmers to have the system respond on its own if none of their rules are matched. The idea of this is to avoid system responses like “I do not know how to help you with that” and “I do not know.” The language was implemented using a scannerless lexer and compiler called parglare in python version 3.6.

I. INTRODUCTION

In recent years there has been an increased interest in applications that are backed by Artificial Intelligences and can communicate with users via speech or text. Some popular versions of this are Apple’s Siri, Amazon’s Alexa, Microsoft’s Cortana and Google’s Voice Assistant [1]. These systems process user speech or text to respond to questions or perform actions. A common complaint with these systems is that they do not respond well to continuations of input. Essentially, these systems do not seem to maintain a context of conversations, and do not accurately refer to the past. John McCarthy suggested a programming language he called Elephant 2000 which can be used to create systems that can refer directly to
the past [2]. According to him the language should have been ready for the year 2000. He later revised his prediction to 2005 and then again to 2015; however, there has yet to be an implementation of it.

This project focuses on a minimal implementation of his language. The implementation is created mainly in python version 3.6 leveraging several freely available packages. The most import among these being parglare [3], a scannerless parser, and the python Natural Language Toolkit (NLTK) [4]. The syntax of the language has been modified to appear more English like. After this project, I believe a truly meaningful implementation of this language could be completed in two years. Thus, I would revise the name to Elephant 2020, two years after this writing.

II. BACKGROUND
A. NAMED ENTITY RECOGNITION

Named Entity Recognition (NER) refers to matching defined entities in text to named types. This text is a sequence of individual words, which are each labeled to any name that they match. For example, a sequence like “John has a flight number 435 at 6 am tomorrow” would produce a sequence with names similar to, “[John, PERSON], [435, NUMBER], [6 am, TIME] [tomorrow, TIME].” Systems for NER are typically implemented uses some form of machine learning or neural network. The system is trained given using a large dataset of text followed by the classification for that text. These systems perform better with more training and larger datasets. It is theoretically possible to train a NER system to classify in any domain, however the systems do not transfer well. Meaning that a system trained to recognize something like army
ranks, would not then be able to recognize dates and times, unless it was extensively trained to do so. [5]

In this project, the Stanford NLP NER model was used with the NLTK python wrapper. The Stanford model was trained by researchers to classify entities into one of seven categories: Location, Person, Organization, Money, Percent, Date and Time. It is also possible to extend this model to recognize other types named entities, but this requires a large amount of data to perform accurately. [4]

B. SPEECH ACTS

Speech acts are a philosophical classification of speech. They are meant to describe the function of speech. This branch of philosophy is attributed to J.L. Austin most notably in his book “How to Do Things with Words” [6] in which he explores the ideas and types of speech acts. He introduces the concepts of Locutionary, illocutionary, and perlocutionary acts. The Locutionary act is what was actually said by the speaker. The illocutionary act is what the speaker actually meant, which can be separate from what the audience understands or believes from the Locutionary act. Both of these actions can be said to have occurred after the speaker finishes speaking. The third type of action is more complex. Perlocutionary acts refer to the result of the user’s speech. This action is only said to occur after the audience of the speech does something in response to speech. For instance, if a speaker asks for food, the perlocutionary event would be that the speaker was offered food by someone who heard them. In this project, I mainly consider the illocutionary act as McCarthy did. This act is expanded to classify speech into several categories, including requests, answers, and questions.

The NLTK distribution provides a chat corpus that has statements labeled with their appropriate speech acts and also provides implementations of classifiers that can be trained with
this data. Using these two elements, it is possible to create a classifier that can somewhat accurately classify speech as one of the several speech acts or dialogue acts, which are essentially a more intensive version of speech acts. [7] [6]

C. ELEPHANT 2000

John McCarthy originally proposed this program in a research paper he wrote in 1998. In it, he states that other proposed programming languages for 2000 were not ambitious enough, and that programming languages should have more features of human language. The most notable specification he gives for that language is that the inputs and outputs are speech acts. This means that it can take in speech, albeit only in a written text format, and output speech that should be responsive to the input. In speech act theory a statement is considered responsive if after the output is heard, the listener will know the correct answer to their input. For example, if someone asks what the address of Amazon headquarters is, telling them that it is at the address of Amazon’s Headquarters is true but unresponsive. A responsive answer would be “410 Terry Ave. North Seattle, WA” which is their current address. There are several methods to ensure responsiveness, such as specifying that the response to a location request is a location. The responsibility of this mainly lies on the programmer, by specifying what correct outputs should be. Another key aspect of Elephant is that the programs can refer directly to the past. The structure of the history record is unknown to the programmer, rather it is up to the compiler or interpreter how it remembers inputs and outputs. McCarthy suggests that a simple form of matching is sufficient for this purpose. This aspect becomes more interesting when coupled with McCarthy’s specification that programs are specified in sentences of logic. His go to example is with an airplane reservation. A person making a reservation is logged and then the existence of
their reservation can be determined by checking the history to see if a reservation was made and if it was subsequently cancelled. [2]

There have been some critiques of this approach who do not believe that this is how humans naturally process information; however, it is an acceptable way to represent events like this because this method can accurately describe the existence and state of abstract objects. McCarthy believes that objects should be as abstract as possible. In other words, the programs should only be aware of things that it actually needs to know. For the example of an airplane reservation, it may only need to know that these can be made, cancelled and checked. If the reservation program is for a single airline it may also need to know the passenger’s name and the flight number, but it would not need to know something like the name of the airline or the price of the ticket. In this work, McCarthy gives a few code fragments to go off, but he does not give a formal documentation of all language specifications. He has several sections which are marked with “(more to come)” but I have yet to find documentation where those sections are completed. As a consequence, some parts of the language have to be assumed based on common programming languages and requirements of example programs. [2]

D. SEQUENCE TO SEQUENCE AND LSTM

Sequence to Sequence is a well-known model for Natural Language Processing systems. This model is normally used to translate language from one language to another. The popular Google Translate application uses a sequence to sequence model. It can also be used to create responses to input sequences. The models are considered to be good at retaining context in a sequence and understanding sentence structure because it processes words as sequences and uses the previous words in a sequence to choose the next output. A sequence to sequence model is actually made up of 2 Recurrent Neural Networks (RNNs). Essentially, RNNs are a specialized
type of neural network which are adept at processing sequential data, such as video frames, and more importantly sentences. Basically, one of the RNNs serves as an encoder, while the other serves as a decoder. [8] So, the first RNN will process each word in the input and these will be transferred forward in the decoder, which will generate a response word by word. Figure 1 illustrates this.

![Figure 1](image)

Figure 1 [8]

Ideally, the input will be translated into a meaningful response sentence as in this example. This is not always the case, a large number of inputs will return gibberish unless they are similar to the training set. One method to improve the quality of these models is Long Short-Term Memory (LSTM). LSTM is a method that allows the model to forget certain training values or inputs which helps it to be more a general model, while maintaining an overarching long-term structure. The LSTM uses several gates to strategically add or remove information from the overall context to make a more performative model.

The Sequence to Sequence model and LSTM used in this project are taken from Tensor Flow library which is written in Python by Google and provides several machine learning algorithms and models useful to researchers. [9]

III. IMPLEMENTATION

The implementation of this project was entirely in Python, except for the Stanford NLP NER model which is written and trained using Java. That model was loaded using the python wrapper provided by NLTK. The models require a Java runtime to be installed as well as a local
copy of the entire Stanford NLP library. The model is loaded at runtime and called when necessary, passing in the user’s input. I added a wrapper around this that allows me to group similar adjacent tokens and returned matched entities as a list. For example, “Jane Doe” would normally appear as two separate entities “Jane” and “Doe” both recognized as a Person. This wrapper will group these two names correctly into “Jane Doe”. [10] [11]

NLTK also provides the implementation used for the dialogue act classifier. First the NLTK chat corpus is downloaded from their site, it is then loaded into the program and fed through a Naïve Bayes classifier. The classifier is trained to classify text as one of 15 possibilities. On the test data, the classifier is 68% accurate. This result can be improved with more data to train with, however this level of accuracy is sufficient for this implementation. For the chat bot, Tensor Flow was used to create the model. Suriyadeepan Ram provides a wrapper class for this model compatible with Tensor Flow version 1.4. I adapted this wrapper to be used with the latest version of Tensor Flow GPU, 1.6. The training data was taken from a corpus housing a large collection of twitter posts and responses. The bot was trained for 55000 epochs on a NVIDIA GTX 1070. The training lasted for around 7 hours, so it was left running over night. The model was rated based on loss which is a function defined in Tensor Flow that computes the weighted cross entropy of a sequence of tensor logits. The classifier is trained using the Adam Optimizer which attempts to minimize the loss function. After the training, the model achieved a mean peak loss of about 2.3. This is not a great score but is adequate for the chat bot’s purpose. The output of the chat bot is gibberish when the input is not similar to any of the inputs it has seen before. For this reason, it may be beneficial to allow users to provide their own training data, or even their own chat bot implementation. This is possible currently but requires the programmer to modify the source of the actions file, which is not ideal. [8]
The history of the application is stored in memory as a python dictionary. After every input, the input and output are added to this dictionary with their respective keys being the time step of the action. When matching events, the program can look at either the text of the event that occurred, or at the time stamp. This supports lookups of certain actions, such as making a flight reservation, as well as looking up the timing of any events.

The implementation of the Elephant language was completed using the parglare package in Python. [3] This package provides an implementation of a scannerless LR parser. LR parsers are non-recursive bottom up parsers that operate in shifts and reductions. They are considered to be the most efficient technique for syntax analysis. In this implementation, statements are executed from the bottom of the parse tree up to the top. This works well for simple expression statements, such as “8 + 7 * 2” but it can make implementing complex structures more difficult. One example of this is the if statement. In a common programming language, the expression after the if keyword will be evaluated and then the subsequent if block will be executed depending on that value. In a bottom up parse the if block will actually be executed before the expression or if are processed. In order to remedy this, I dynamically wrap functions inside of lambda statements. This is done using the lambda wrapper as seen in figure 2.

```
def lambda_wrapper(func):
    def wrap(context, nodes):
        return lambda context, _: func(context, nodes)
    return wrap
```

*Figure 2*

This takes the function it decorates and returns a lambda function that will call it with its arguments. Doing this allows the parser to call the function when necessary, instead of where it is first encountered.
The tokens that are recognized by the language are created using regular expressions in python. Figure 3 shows the declaration of Identifiers, which are essentially variable names, and numbers which are integers or floating-point numbers.

![Figure 3](image)

These tokens can be combined into productions of more complex entities as shown in Figure 4 which shows the declaration of a program and a for statement list. These declarations are very similar to the standard notation for BNR grammars. A program is an optional Statement List followed by an optional For Statement List terminated by an End keyword and a “.”. A For Statement List is a single For Statement, or a For Statement List Followed by a “,” and another For Statement.

![Figure 4](image)

After all of the tokens and productions are specified, actions to take when productions or tokens are found are added. These actions are added to a dictionary that the parser performs a lookup on at runtime. Figure 5 is an example of the action for an Identifier token. Here, the value of the token, which is already a string, is just returned back to the caller.

![Figure 5](image)
Similarly, Figure 6 shows the action created for a Program. It first separates out the statement list from the for-statement list and then calls two recursive functions that process and run each statement they contain.

![Figure 6](image)

Finally, Figure 7 is an example of a test Elephant program similar to McCarthy’s flight request example. The for statement is a construct that McCarthy does not mention but, seemed necessary to fulfill his requirements. This for statement is not like those found in standard programming languages. Instead of performing an iterative loop, it matches the speech acts and named entities in the input and executes the subsequent statement if all of the entities and acts it specifies are found. The Exists, Cancel, Commitment, Make, and Answer Query statements are

![Figure 7](image)

This implementation of Elephant supports most of the syntax as described by McCarthy. Its input and outputs are speech acts, and it determines its actions based on that. The user can define their responses in a way that makes them truly responsive. The implementation also provides if, while and let statements which perform similarly to their counterparts in the Swift language. In addition, this implementation provides support for creating abstract objects called concepts. The syntax for this is shown in Figure 8. The statement begins with the Concept
keyword followed by an identifier that names the concept. After that there is a colon, and the actions of the object are defined. The To keyword precedes the name of the method and the Call keyword precedes the name of the user defined function that should be called in response to this action. To statements can be defined in a list separated by commas. These concepts allow a user to specify their own actions to take while fulfilling McCarthy’s specification that the system should not have more information than absolutely necessary about abstract objects. Allowing users to use their own functions allows them to integrate into their existing system. It is also necessary as the language does not currently support function definitions.

```
CONCEPT_test : TO check call to_func, to fix call to_func
```

Figure 8

After the program is parsed, and the executable statements are generated. The program begins a while loop that runs until the user types quit. During every iteration, the user inputs a statement and then the input is processed by the program and responded too.

IV. Future Work

The next steps in this project would be to add more language features and improve the accuracy of the backend chat bot and the NER and dialogue act recognizers. Improvements in these areas could make the language more viable for use in personal and commercial products. One feature that I believe would be a marked improvement is to support a more complex form of history lookup. Simplistic matching is good for simple programs, but more sophisticated programs will require a more complex form of event and action matching. Another improvement would be to allow users to provide their own examples of NER classes and then recognize custom named entities. This provides more flexibility to the system. For example, if a company provided something like insurance claims, rather than recognize clams as numbers, certain
alphanumeric patterns could be classified as claims. Similarly, a system to be trained to recognize models of cars or brands of products. I believe there should also be a testing library to allow users to verify the output of the program, currently the best option would be to manually test the program or to create a processing script that would repeatedly input to the program. Finally, the programming language needs better documentation. I worked based off of McCarthy’s paper, however this is an incomplete specification of his language, and, beyond that, my implementation differs from his specification in several ways.
V. BIBLIOGRAPHY


