APPLICATION OF MACHINE LEARNING IN CONTENT GENERATION FOR EDUCATIONAL VIDEO GAMES

An Undergraduate Research Scholars Thesis

by

LLOYD DONELAN, BRENTON LENZEN, and KISHAN PATEL

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisors: Professor Andre Thomas
Professor Hadeel Ramadan
Dr. Theodora Chaspari

May 2020

Major: Computer Science

TABLE OF CONTENTS

Page

ABSTRACT 1

NOMENCLATURE 3

CHAPTER

I. INTRODUCTION 4

Background Information and Research Motivations 4
Differences from Previous Studies 5
Project Expectations 6
Potential Future Tasks 7

II. METHODS 8

NLP Pipeline Conceptual Overview 8
First Iteration of the Text Parser 13
First Iteration of the Chatbot 14
Second Iteration of the Chatbot 14
Second Iteration of the Text Parser: The NLP Pipeline 15
First Attempt at Generating Knowledge Bundles 16
Second Attempt at Generating Knowledge Bundles 18

III. RESULTS 22

Analysis of Question Generation Using Knowledge Bundle Attempt #1 22
Question Generation Good Examples (Knowledge Bundle Attempt #1) 22
Question Generation Mixed Examples (Knowledge Bundle Attempt #1) 23
Question Generations Bad Examples (Knowledge Bundle Attempt #1) 23
Analysis of Knowledge Bundle Attempt #2 24

IV. CONCLUSION 27

REFERENCES 30

APPENDIX A: DICTIONARY-GEN ALGORITHM 32

APPENDIX B: QUESTION-GEN ALGORITHM 36

ABSTRACT

Applications of Machine Learning in Content Generation for Educational Video Games

Lloyd Donelan, Brenton Lenzen, and Kishan Patel
Department of Computer Science & Engineering
Texas A&M University

Research Advisors: Professor Andre Thomas and Professor Hadeel Ramadan
Department of Visualization
Texas A&M University

Research Advisor: Dr. Theodora Chaspari
Department of Computer Science & Engineering
Texas A&M University

Over the past few years, students have become increasingly unmotivated to read their assigned textbooks as an accompaniment to classroom lectures and activities. Reading the textbook is known to improve comprehension and overall student performance in classrooms. If reading the textbook was reformatted into a more engaging experience, perhaps it would improve student motivation and knowledge retention. Teaching students the importance of learning while also motivating them to do well in class will help them gain the knowledge and grades needed to land competitive jobs after they graduate college.

Game-Based Learning (GBL) is an emerging field of study that attempts to use video games to create interactive educational experiences. Game-Based Learning has been shown to have educational merit, being well-known for providing intrinsic motivation for students to learn (most often, as a supplement to traditional coursework). With GBL in mind, is it possible to generate interactive game content from textbooks using machine learning (ML) and artificial

intelligence (AI) that can replace or supplement the source material in terms of educational content in a traditional classroom setting?

Our team proposes to lay the groundwork for future research in Game-Based Learning and Machine Learning at the LIVE Lab undergraduate research lab (Texas A&M University, College of Architecture, Dept. of Visualization) by attempting to reformat school textbooks into interactive chatbot AIs with the assistance of knowledge compilation & fact-retrieval systems designed for generating educational video game content.

NOMENCLATURE

GBL	Game-Based Learning
AI	Artificial Intelligence
ML	Machine Learning
MLAI	Machine Learning and Artificial Intelligence
The Lab	LIVE Lab at Texas A&M University (College of Architecture, Department of Visualization). The undergraduate research lab that the authors belong to.
edutainment	Educational entertainment
API	Application programming interface; also called “framework”
NLU	Natural Language Understanding
NLP	Natural Language Processing
LSA	Latent Semantic Analysis
NLTK	A natural language processing framework in Python
spaCy	A natural language processing framework in Python
chatbot	An interactive AI that users can send messages to, and the chatbot will generate semi-intelligent messages to send back to the user
Rasa	A retrieval-based chatbot framework in Python
QG	Question Generation
QA	Question-Answer
KB	Knowledge Base

CHAPTER I

INTRODUCTION

Background Information and Research Motivations

Studies have shown that reading textbooks improves student comprehension of class material at multiple levels of education[1][2][3], but have also shown that students are becoming increasingly unmotivated to read textbooks on their own[2][3]. Some studies have researched methods for improving student motivation and interest when performing assigned readings, but such methods have been mostly limited to traditional exercises such as worksheets, study guides, in-class discussions, and quizzes[2][3].

The growing field of Game-Based Learning is known for utilizing the intrinsic motivation students have when playing video games to assist comprehension and to reinforce learning[4][5][6], yet there are few studies, if any, that attempt to provide students with an engaging educational experience that embodies the vast amount of information found in a textbook.

Rather than designing games that encompass a small subset of learning objectives, what if we could procedurally generate content for all of the information in a textbook, thus encompassing most (or all) of the learning objectives of a class? Creating an extensive, interactive, and well-structured game that students could actively interact with would likely be used more frequently by students than their textbooks, which only offer a passive reading experience[6].

Furthermore, if we could change the way students interact with their textbooks we could provide greater motivation for acquiring the knowledge present in the book without them needing to read it[3]. And, what better framework to use to motivate students to learn than Game-

Based Learning? In addition to motivational benefits, GBL also provides both active and passive learning opportunities as well as the potential to embody practice problems and to foster social interaction between students[6].

Machine learning and procedural game content generation have seen much progress in the past few years, and we believe that it is a highly capable vehicle for achieving our goals[7]. Thus, we propose using Machine Learning and AI (MLAI) to assist in procedural content generation for educational video games, to be employed in a traditional classroom setting.

Differences from Previous Studies

There are many studies within the realms of Game-Based Learning and procedural content generation using Machine Learning and AI (MLAI), but few, if any, attempt to merge the two together. First, our project is different than past research projects because it attempts to use MLAI to create content with the explicit purpose of being used in GBL. Multiple published research papers focus on generating high-quality procedural game content (in terms of fun factor, fairness, complexity, and player reception) for entertainment purposes, but not for educational purposes. Second, our project differs in its use of procedural content generation – rather than generating levels, enemies, or mechanics from a pre-determined set of game assets and developer-set constraints (as has been attempted by others in the past), we will be attempting to parse a secondary source of educational information (textbook, encyclopedia, database, etc) into a format that we can then generate interactive game content from. Lastly, our project differs from past research projects because we will not be testing several different methods of procedural content generation, nor will it be comparing the reception of these methods by players. Those tasks might be carried out in the future in a follow-up study.

Project Expectations

My research team hopes to create a textbook parser using natural language processing techniques, outputting educational content from a school textbook into an intelligent and interactive chatbot AI. This is intended to be a “first venture” into procedural content generation for Game-Based Learning for my research team, and while we don’t expect to make major progress towards our end goal of procedurally generating comprehensive interactive educational experiences, we do expect to lay the groundwork for future research. By the end of the year, we hope to have completed:

- Create a system for parsing a secondary source of educational information, such as a textbook, and outputting well-defined “knowledge bundles” containing all information pertaining to a particular concept or idea.
- An interface between the information parser and the chatbot AI that allows the AI to receive “knowledge bundles” as inputs.
- Create a prototype for an interactive chatbot AI that can provide information about a historical figure, creating dialogue from player inputs and relevant “knowledge bundles.”
- Document all completed work and create estimates for the feasibility of potential future tasks for our team.

The successful completion of these tasks will demonstrate the importance of procedural content generation in educational entertainment (edutainment) as well as the relevance of Game-Based Learning to cutting-edge research in the field of Computer Science. Additionally, our research could form the basis for renewed discussions on the topics of: abstraction of learning objectives, educational examination methods, methods of measuring student comprehension, teaching strategies, and the development of intrinsic motivation in interactive educational content.

Potential Future Tasks

Because our current research is intended to lay the groundwork for future tasks, it is important to keep potential future tasks in mind. Some tasks we might like to work in the future that will build off of our research are:

- Generating game assets from “knowledge bundles”
- Generating game mechanics from “knowledge bundles”
- Creating an intelligent learning assistant/tutor AI that will assist the player and create a personalized learning environment

CHAPTER II

METHODS

Development on the project began with DonBot (shorthand for "Donatello Robot"). DonBot was a conceptual prototype for a chatbot that would store information on the renaissance artist Donatello, which would then be used to construct intelligent and informative responses to user inquiries about the artist. A fully functional prototype would require the following components:

1. A text parser, to read in and format information about the subject (Donatello).
2. A chatbot, to interact with users to provide them with information about the subject (Donatello).
3. An interface between the text parser and the chatbot, containing the output of the text parser in a useful format for the chatbot and other tools/software that we might want to develop in the future for generating interactive educational content.

To complete the overall process, a significant amount of NLP work is required before providing the chatbot with the information extracted from the input corpuses. This will ensure that all extracted information is sterilized and well-formatted, making it easy to use the information for future procedural work. Thus, we chose to store extracted information in a format which we call a "Knowledge Bundle."

NLP Pipeline Conceptual Overview

In our theoretical design, Knowledge Bundles contain facts that are represented as tree-like structures (called Knowledge Trees), with the root node containing a list of entities where the first listed entity is the subject of the knowledge bundle (in our case, Donatello) and the rest of the entities are known synonyms for the subject. The second level in the tree contains nodes with a list of verbs, where the first verb is considered to be the “main verb” and the rest of the verbs are synonyms of the main verb. The third level in the tree contains nodes with prepositions or determiners, and the fourth level contains nodes with nouns (the subject of the verb in the second level of the tree). These first four levels of the tree comprise “core facts,” and the second, third, and fourth levels together comprise “knowledge statements.” Following one branch from the Entity (root level) to the fourth level of the tree builds a single factual statement, hence the term “core fact.” Additional details may be included in subsequent nodes after the fourth level of the tree, but are not necessary.

By iterating through a Knowledge Tree, all known facts about a specific entity can be examined. Additional properties can be added to tree nodes or Knowledge Bundles to support project-specific algorithms or data that might be useful for further NLP techniques such as question and answer generation.

Properly generating Knowledge Trees for use in Knowledge Bundles would require pipelining several modern NLP techniques and is thus a significant research challenge. For the syntactic approach described in this paper, these techniques may include:

1. Spellchecking and Tokenization
2. Entity Classification and/or Super-sense tagging, (intended to assist Co-reference Resolution and Clause Extraction)
3. Co-reference Resolution, Pronoun Resolution
4. Sentence Simplification and/or Clause Extraction
5. Question and Answer Generation from Simplified Sentences

First, the input corpus(es) need to be sterilized and properly tokenized. There is already a great deal of work in these areas, and so we will not discuss these steps further[8]. After that, sentences should be read through and the parts of speech labelled, with special interest in preprocessing techniques that will assist with Co-reference Resolution, Pronoun Resolution, Sentence Simplification, and Clause Extraction. Next, ambiguous entity references should be resolved and replaced with entity names. Any ambiguous non-pronoun references to some entity X should be recorded as "synonyms" in the knowledge bundle for that entity (pronouns are not unique to any specific entity, and thus should not be considered synonyms). After (3), all entity references should be un-ambiguous. Sentences can now be simplified, and/or clauses extracted, to generate short factual statements.

As an example, the sentences "Einstein died in 1955 at the Princeton Medical Center in New Jersey. He is well-known for his contributions to Physics," would (ideally) yield outputs at each step in the pipeline similar to the following:

- ⇨ (1, 2) "Einstein died in 1955 at the Princeton Medical Center in New Jersey. He is well-known for his contributions to Physics."
- ⇨ (3) "Einstein died in 1955 at the Princeton Medical Center in New Jersey. Einstein is well-known for his contributions to Physics."
- ⇨ (4) "Einstein died in 1955. Einstein died at the Princeton Medical Center. Einstein died in New Jersey. The Princeton Medical Center is in New Jersey. Einstein is well-known. Einstein has made contributions to Physics. Einstein is well-known because he has made contributions to Physics."

Each sentence generated by (4) can now be added to a Knowledge Tree. Most of the sentences can be added to a Knowledge Tree for a Knowledge Bundle dedicated to the entity "Einstein," but notice that there are two peculiar sentences in the output:

(a) "The Princeton Medical Center is in New Jersey."

(b) "Einstein is well known because he has made contributions to Physics."

In (a), there is no mention of Einstein. Rather, this is a fact about a different entity (the Princeton Medical Center). Thus, (a) should actually be added to the Knowledge Tree for a Knowledge Bundle dedicated to the entity "Princeton Medical Center." In (b), Einstein is mentioned but this sentence is actually just a more detailed version of a previous sentence: "Einstein is well-known." For this reason, the Knowledge Tree for Einstein should contain a branch with "Einstein is well-known" as the core fact and "because he has made contributions to Physics" as additional details.

It is worth noting that (3) and (4) won't always proceed so smoothly. For example, the sentence "North Korea and Cuba are the only places you can't buy Coca-Cola" would look something like this:

⇨ (1, 2, 3) "North Korea and Cuba are the only places you can't buy Coca-Cola."

⇨ (4) "You can't buy Coca-Cola in North Korea. You can't buy Coca-Cola in Cuba. You can buy Coca-Cola in every place except North Korea or Cuba."

There are a few peculiarities about the output here. Perhaps our original intuition tells us that some facts should've been generated about the entities North Korea and Cuba, yet the only facts generated were for the entity "you." Additionally, there is no straightforward way to generate anything resembling the third sentence, "You can buy Coca-Cola in every place except North Korea or Cuba," from the input text. Finally, notice that there are two potential pitfalls to avoid – generating either of the sentences “You can’t buy Coca-Cola” (from the input sentence) or “You can buy Coca-Cola in every place” (which is seems to be a less-detailed version of the last output sentence, but is actually a different sentence because it has a different meaning from the original!).

The final step in the Knowledge Bundle generation pipeline is to generate potential user inquiries, based on known facts about the entity referred to by a Knowledge Bundle. This enables a retrieval-based approach to user inquiries about facts in the Knowledge Database (the set of all Knowledge Bundles for a particular project). Generated inquiries can then be added to the Knowledge Database and given a reference to the fact that serves as their answer. These questions and answers can then be used as training data for a chatbot or other Q&A system, to allow the system to respond appropriately to user inquiries based on user intents.

The process of question generation comes with its own challenges and pitfalls. Simple “reading comprehension”-style questions are known to be created with decent results using linguistic transformations on preprocessed sentences. Heilman took a similar approach to our proposed NLP pipeline, but did not fully explore the benefits of sentence simplification and paraphrasing on QG[8]. We hypothesize that “reading comprehension”-style questions can be generated more accurately with a similar approach if the input sentences are reduced to simple clauses (concise, factual statements) and are sterilized to remove unambiguous pronoun references. Additionally, by reducing input sentences to simple clauses, we can then use the reduced clauses (perhaps with some minor post-processing) as the answers to generated questions. This is a significant difference from Heilman’s approach, which deals exclusively with QG and not QA[8]. In simpler terms, the NLP pipeline is intended to reduce an input corpus down to a series of concise factual statements, which would then serve as answers to questions that are generated from those statements using the statements themselves in combination with any contextual information that was extracted from the original input corpus.

In the following sections, we explain the initial research process. Then, we give a high-level overview of how each component in our prototype of the NLP pipeline. Finally, we describe future work to be done and potential improvements and applications of the project as a whole.

First Iteration of the Text Parser

The text parser needed to take in a large, domain-specific corpus of text and output facts about historical figures. Our input domain was restricted to educational works containing information about the artist Donatello or his work. Initially, we used the Britannica entry on Donatello as our (small) input corpus, with the intention of increasing the size & diversity of sources in the input corpus in the future. Britannica was chosen over alternatives such as Wikipedia because Britannica is curated by verified professionals, and does not provide an excessively large input corpus (smaller input size was preferred for initial prototyping).

For the first iteration of the text parser, we did not yet have a clear understanding of our conceptual NLP pipeline. Instead, we retrieved data from the input text through the following process (using NLTK and spaCy):

1. HTML tag scraping
2. Removal of irrelevant information
3. Creating Intent/Answer Pairs
4. Creating Training Data Files for DonBot

In (1), we extract the text of the web page from the HTML code of the web page. In (2), we remove irrelevant information such as information about Britannica and information about other topics that are indirectly related to Donatello, which are not currently relevant to our research.

Next, we performed some additional processing in NLTK to group sentences by intent (based on detected keywords). Finally, we automatically generated training data for the chatbot from our text using python scripts. Sadly, our algorithm did not succeed in generating acceptable-quality intent bins, and this is reflected in the generated training data.

The text parser would remain in this state until the second iteration of the chatbot.

First Iteration of the Chatbot

DonBot's first iteration was based off a generative chatbot created by github user Hugging Face. This model was chosen for two reasons:

1. The model uses a 'persona' to generate responses, where the persona is defined with a couple of sentences that provide the AI with knowledge about a historical figure.
2. The model uses transfer learning, which would (theoretically) reduce the amount of training data needed for the AI and improve the AI's understanding of user messages.

This chatbot did not work well, lacking enough training data related to Donatello while also having too much modern information from the conversation training data. The bot was able to state its name, Donatello, but was not able to provide users with reliable facts. In some tests, the chatbot went so far as to insist that Donatello was from the Dominican Republic and that he was a slave! This implementation was quickly abandoned.

Second Iteration of the Chatbot

The next iteration of DonBot, DaVinciBot, was created using the Rasa retrieval-based chatbot framework. This version of the chatbot requires three types of training data:

1. NLU Training: a collection of potential user inputs, each labeled with a particular intent. The bot will be able to recognize similar inputs as having the same intent.
2. Domain: a collection of actions that the bot can take, the user intents that it can recognize, and templates for the bot's responses to user inquiries.
3. Stories: a collection of dialogue paths, outlining the structure of user-bot interactions. Also defines behavior for unrecognized intents.

This iteration of the bot responds well to user inquiries, as long as the inquiries resemble something in the training data. Unlike the previous iteration, the bot will never respond with inaccurate information unless we provide some in the training data. If the bot is unable to understand the user's inquiry it will always respond in a predefined way, which ensures professionalism and some level of reliability in the bot's responses.

Second Iteration of the Text Parser: The NLP Pipeline

By this point, it was clear that we would need a more structured attempt to process our input text. This is when we came up with the NLP Pipeline, which could (theoretically) *reduce an input corpus to a series of concise factual statements, grouped by the Entity they refer to.* Implementing such a design would allow user inquiries to be interpreted as a simple search over the Knowledge Tree of the Entity in question; such an implementation would require Entity Recognition, Sentence Simplification and/or Clause Extraction.

Entity Recognition has some major issues to deal with. Simplifying a sentence with only ambiguous references to an entity (such as pronouns or monikers) would not generate any useful information – a new entity would be generated (such as "The First President", instead of "George Washington"), and the fact would be attributed to the new entity instead of the entity it truly applies for. This would result in facts for an entity X being distributed among up to n different knowledge bundles, where n is the number of ambiguous references to X in the input corpus. Additionally, pronouns such as he/she could be used multiple times in the input corpus, each time referring to a different entity. Facts for two separate entities X and Y could be attributed to another Knowledge Bundle for a single entity “He”/“She”! Solving these issues would require additional steps to be added to the pipeline.

Solving the first issue (which will be referred to as “the nickname problem”) would require some way to resolve common nouns and nicknames to their owner’s name, which could most likely be achieved by, or with the assistance of, a semantic approach using Entity Clustering or word vector analysis. We choose to leave this to future work.

To solve the second issue, we decided to implement Pronoun/Co-reference Resolution, substituting unclear pronouns with the real Entity’s name before performing Sentence Simplification. For our initial prototype, we used an existing spaCy module for Co-Reference Resolution. The module worked well for most sentences, but was less successful with long sentences. Perhaps some level of Sentence Simplification could be done as pre-processing to improve the success rate of Co-Reference Resolution, but it is important to note that too much pre-processing could have negative effects on the main Sentence Simplification process by introducing errors or removing important contextual information early on in the NLP pipeline. We leave improvements to Co-Reference Resolution to future work.

First Attempt at Generating Knowledge Bundles

Before implementing the Knowledge Tree structure that we originally planned for the NLP Pipeline, we needed a way to generate Knowledge Trees and Knowledge Bundles from the input data. Our first attempt at generation of Knowledge Bundles ended up as more of a Dictionary generator, using a series of dictionaries and hash ids that are generated from sentences to associate parts of speech with their original context. Each part of speech would reference their source sentence, and each subject would then hold reference to every hash id of a sentence describing information about them. For example, for the sentence “Donatello seemingly demanded a measure of artistic freedom,” we would get the following output information for the sentence:

1. Sentence hash id = -572009579105218340
2. Subject(-572009579105218340) = Donatello
3. Predicate(-572009579105218340) = seemingly demanded a measure of artistic freedom
4. Modifiers(-572009579105218340) = null

This populates data into four different dictionaries, each with the sentence hash id as the key and the content as the value. Additionally, we would also get the following output information in a *dictionary of subjects*, where the key is the subject name and the value is a list of hash ids of all sentences containing information about that subject:

1. Donatello = [-572009579105218340, …]
2. demanded = [-572009579105218340, …]
3. measure = [-572009579105218340, …]
4. artistic freedom = [-572009579105218340, …]

This algorithm is referred to as the DICTIONARY-GEN algorithm. Now, all already known information about a subject(s) is accessible in O(1) time if the subject and clause keys are known, and the total time to iterate over all information about all subjects is in O(n) of the input corpus size (thanks to the use of dictionaries). The QUESTION-GEN algorithm can then be run on the collection of Dictionaries to generate questions for the RASA chatbot training data. In the future, we would prefer to use true Knowledge Bundles generated by some BUNDLE-GEN algorithm (as opposed to the dictionary format we are currently using) to provide information to the question generation algorithm. See an overview of the DICTIONARY-GEN and QUESTION-GEN algorithms in Appendix A and Appendix B.

For very large input corpuses, the collection of dictionaries might not be *searchable* at runtime at acceptable speeds for real-time applications (video games). This isn't currently an issue for the small-time retrieval-based chatbot that we are currently using, which generates training data at compile time, but future projects such as generative chatbots would require this information to be accessible in something faster such as O(log(n)), which would require that we turn our collection of dictionaries into a balanced binary decision tree structure. Such a structure might lend itself to a very fast and robust database system, allowing queries of the hashed sentence data, and perhaps eliminating the need for individual knowledge bundles to be created at compile time. The entire database could be a Knowledge Tree comprising facts of the entire input corpus, and Knowledge Bundles for individual subjects could be generated at runtime for frequent access to information about particular subjects as needed. Although we leave the implementation of such systems to future work, we will now discuss the theory behind such an implementation.

Second Attempt at Generating Knowledge Bundles

Finally, after implementing some semblance of the aforementioned NLP Pipeline and having a greater understanding of the linguistic tools at our disposal, we were able to start thinking about a truer implementation of Knowledge Bundles. By this point in time the idea of a Knowledge Bundle was no longer a "stand-alone package" of information, but was instead considered to be an "extracted subgraph" from a much larger graph of all known information, referred to as the Knowledge Base (KB).

All of the facts known to the AI would be placed into a massive Knowledge Tree, which serves as the AI's Knowledge Base. At any point in the tree, a node could be selected as the root of a subtree and that subtree could be extracted and cached (in theory, anyways) – such a subtree could then be used as a Knowledge Tree for the subject of the tree's root node. For example, extracting a subtree where the root node's subject is "artwork" (which is a category of things, rather than a named entity) would cache the Knowledge Tree for all artwork in the Knowledge Base. Extracting a subtree where the root node's subject is "Da Vinci" would cache the Knowledge Tree for all known information about the artist Da Vinci (a specific named entity, rather than a category of things); this subtree is, itself, a subtree of the subtree where the root node's subject is "artists," which is in turn a subtree of the subtree where the root node's subject is "people."

Nodes with categorical information such as "artwork," "people," or other broad specifiers are called "decision nodes." Nodes with factual information such as a date, e.g. "1922," or a name, e.g. "Donatello," are called "data nodes." A depiction of a Knowledge Tree is shown in Figure 1.

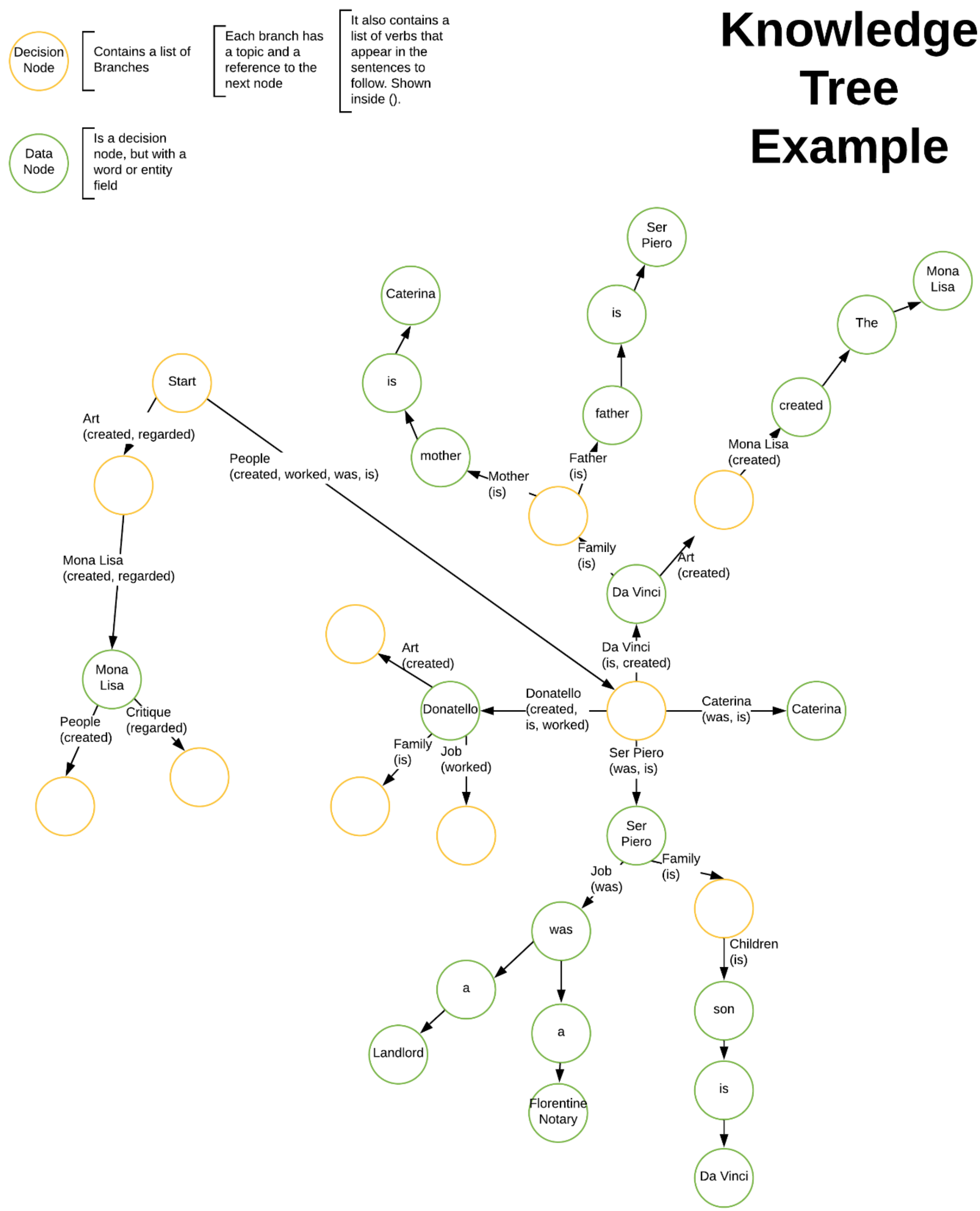

Figure 1: An example illustrating the structure of a Knowledge Tree.

Each Knowledge Tree, edges contain a list of verbs, where each verb is guaranteed to be found in the main clause of at least one fact in the subtree containing that edge. For example, if an edge E contains a verb V, and E is adjacent to some nodes N1 and N2, with N1 being higher level than N2, then V is guaranteed to be found in the main clause of at least one fact in a subtree originating at N1. In Figure 1, this can be illustrated by choosing the following values for E, V, N1, and N2:

- E := the edge from the decision node for "People" to the data node for "Da Vinci" or from the decision node for "People" to the data node for "Donatello"
- V := created
- N1 := the decision node for "People"
- N2 := the data node for "Da Vinci" or the data node for "Donatello," based on the choice of E
- With these definitions, the verb "created" is guaranteed to appear in at least one fact in a subtree starting at the decision node for "People." In this example, it appears in any subtree including the fact "Da Vinci created the Mona Lisa" or the fact "Donatello created art" (note that the training data for this tree should've been vetted to say "Donatello created *artworks*").

As demonstrated in the example, facts can be generated simply by starting at a data node that contains a subject, then traversing the Knowledge Tree down to a leaf node. This structure removes the need for dictionaries and is useful as both, an intermediate data format to change the input corpus into chatbot training data, and an efficient way to lookup facts (in terms of time complexity). Our Results section will discuss the time and space complexity of this format for Knowledge Bundles as well as implications of the system and ideas for future work.

CHAPTER III

RESULTS

Analysis of Question Generation Using Knowledge Bundle Attempt #1

The NLP Pipeline appears to be a fairly effective approach to generating facts for the chatbot's training data. Our current prototype shows mixed results, but can effectively identify user inquiries related to simple sentences from the input corpus or some extracted simple clauses. Although we are not currently looking up any information at runtime, runtime lookup of information in our collection of hash id dictionaries is O(1) with a known key, and O(n) for an unknown key. Some good and bad examples of user inquiries, and the chatbot's responses, will now be given. For these examples, the chatbot was trained on the output of question generation resulting from an input corpus comprised of the Britannica pages for Donatello and Leonardo Da Vinci.

Question Generation Good Examples (Knowledge Bundle Attempt #1)

(1) *Q:* What was Leonardo da Vinci's personality like?

A: Leonardo da Vinci was described as having a gracious but reserved personality and an elegant bearing.

(2) *Q:* Was Leonardo unmarried?

A: Leonardo never married, but Leonardo da Vinci had many close relationships with other arists and intellectuals as well as with Leonardo da Vinci's assistants.

(3) *Q:* Where his parents married?

A: Leonardo's parents were unmarried at the time of Leonardo da Vinci's birth.

Example (1) is straightforward. It demonstrates a good response to a simple question. Examples (2) and (3) both show an impressive response to seemingly simple questions that are essentially asking to retrieve the same factual information, but for different subjects (Leonardo versus his parents). Additionally, (2) uses "married" and (3) uses "unmarried," where "married" was in the training data but "unmarried" was not. In (3), "his" is easily ignored in favor of "Da Vinci" because, in this case, the chatbot was only trained on information about Da Vinci. Semantic information related to the context could be used to apply coreference resolution to user inquiries in the future.

Question Generation Mixed Examples (Knowledge Bundle Attempt #1)

(1) *Q:* Who was Leonardo da Vinci?

A: Leonardo da Vinci was an artist and engineer who is best known for Leonardo da Vinci's paintings, notable Mona Lisa Mona Lisa and the Last Supper.

(2) *Q:* Was Leonardo ever commissioned to make anything?

A: According to contemporary sources, Leonardo was commissioned to create three more pictures, but six works have since disappeared or were never done.

Example (1) demonstrates that issues in Coreference Resolution have caused strange syntax and duplication of the name "Mona Lisa." These issues were propagated down the NLP Pipeline. Example (2) confirms that Leonardo was commissioned to make something, but it does not tell the user what he was commissioned to make.

Question Generation Bad Examples (Knowledge Bundle Attempt #1)

(1) *Q:* When did Leonardo die?

A: In 1482 Leonardo moved to Milan to work in the service of the city's duke- a surprising step when one realizes that the 30-year-old artist had just received Leonardo's first substantial commissions from Leonardo's native city of Florence – the unfinished panel painting Adoration of the Magi for the monastery of San Donato a Scopeto and an altar painting for the St. Bernarn Chapel in the Palazzo della Signoria, which was never begun.

(2) *Q:* dasfsadfaserfewqrqwersdfsadfasefwaer

A: Leonardo da Vinci's father, Ser Piero, was a Florentine notary and landlord, and Leonardo da Vinci's mother, Caterina, was a young peasant woman who shortly thereafter married an artisan.

(3) *Q:* Did Leonardo take care of himself?

A: Leonardo da Vinci – Lady with an Ermine Lady with an Ermine, oil on panel by Leonardo da Vinci, c. 1489-91; in the National Museum, Krakow, Poland.

Example (1) is straightforward – the bot provides a very long and unrelated response to a seemingly simple question. Example (2) shows that the bot failed to detect an invalid input, and as a result chose a seemingly random response instead of using the fallback response of telling the user that it didn't understand the question. Example (3) shows the bot giving a completely unrelated response to a question that is very similar to some sentences in the training data.

Analysis of Knowledge Bundle Attempt #2

For the second formulation of Knowledge Bundles using a Knowledge Tree as the Knowledge Base, the worst-case search time for a fact and the space complexity of the system are both O(N+E), where N is proportional to the number of nodes and E is proportional to the number of edges in the tree to search through (either the entire KB, or a cached subtree of significantly smaller size). Although O(N+E) is an acceptable start, it is less than ideal for the massive Knowledge Base that would be generated from an input corpus comprised of multiple textbooks. To improve the space and time complexity of the current system to O(log(n)), it could be converted to a binary decision tree, which would require an efficient method of collapsing multiple nodes into "super nodes" or strongly connected components of the graph that could then be efficiently searched. We leave these optimizations to future work.

Efficient time complexity for such a KB requires loading the KB into memory which, for a large Knowledge Base, would incur a very large space complexity cost. For end-users this space complexity could be unfeasible due to low available memory. The space complexity becomes even larger if we also include references to cached Knowledge Trees, which would probably be stored in hash tables for constant time lookup, resulting in an additional O(n) space complexity where n is the maximum number of Knowledge Trees to keep cached in memory at a time. A potential solution to space complexity is to distribute data between a cloud server, which could host the entire Knowledge Base, and the end user's machine, which could keep only cached data in memory such as references to subtrees, or the subtrees themselves. Keeping more data in memory on the client's machine will result in faster fact retrieval times (because there is no travel time for packets being sent to / received by the server), but would also incur an increased local space complexity cost on the client. We leave the effective distribution of data across a cloud server and client machines to future work.

Named Entity Recognition (NER) accuracy is highly important to be able to generate such a Knowledge Base. The system hinges on being able to accurately label categories of entities to create decision nodes, and of course, being able to accurately label named entities for data nodes. During development it is imperative that training data contains accurately labelled entities to ensure that the performance of algorithms operating on Knowledge Trees is examined in optimal conditions. Testing the system with training data that is not labelled accurately (or, not labelled at all) is equally important – doing so will inform us of what kinds of additional training data are needed for the system to be reliable, and how to handle non-vetted training data.

Additionally, effective clause extraction is extremely important to be able to generate such a Knowledge Base. Each branch in the Knowledge Base is based on facts in the form of simple clauses that are generated by the clause extraction step of the NLP Pipeline. Our current clause extractor does not have sufficiently reliable performance to allow for Knowledge Trees to be generated without additional input processing. Introducing a heuristic to measure the accuracy of extracted clauses based on the original input sentence would improve the reliability of the system by providing an internal "vetting" mechanism, which could then be used to automatically switch from clause extraction to some other fallback mechanism if the measured accuracy of an extracted clause(s) does not meet some minimum threshold. We leave the improvement of the clause extractor to future work.

This format of Knowledge Trees lends itself extremely well to question generation (QG), allowing questions to be generated simply by specifying a data node S containing the subject of the question, and then traversing down a branch(es) to the leaf node(s) of a subtree(s) rooted at S. This is the primary motivator behind this Knowledge Base format.

CHAPTER IV

CONCLUSION

In the end, we were able to achieve most of our goals that we set out at the beginning of the project – we created a somewhat functional prototype, with mixed chatbot accuracy. The current chatbot is retrieval-based and thus does not require an efficient runtime implementation of knowledge trees (only compile time). Ideally, we will be able to create an effective data storage and retrieval system in the future that allows for fast search algorithms and lookup times for facts at runtime. This will allow for the project to shift its focus to a semantic-based approach, which has the potential to be more accurate and more flexible when responding to user inquiries.

The current description for the true Knowledge Tree format specifies a large Knowledge Base containing all the facts known by a particular instance of the AI, which can then be broken down into smaller subtrees of facts about particular subjects. Those subtrees of facts can then be used as a smaller knowledge base for retrieving information about the subject of the subtree (e.g. Donatello, Da Vinci). This scheme will provide a worst-case time and space complexity $O(N+E)$, where N is the number of nodes and E is the number of edges in the Knowledge Base and thus is proportional to the number of facts in the input corpus. Future implementations of such algorithms should include the use of a dictionary, hash table, or hash map data structure to cache references to subtrees for recently and frequently referenced subjects, allowing $O(1)$ lookup time after the initial $O(N+E)$ search.

As suggested by our faculty advisor, trees may not be the optimal data structure for fact storage and retrieval. A portion of future research endeavors should be dedicated to looking for other graph structures or improvements to the existing tree structure that might be better for our use case. Another avenue to explore is the use of fuzzy logic to assist searching through the knowledge base at runtime; fuzzy logic could potentially be implemented by providing some additional meta information to subject nodes in the graph, and then using that meta information in our search algorithm. We leave such tasks to future work.

There are also opportunities for future work in improving the NLP Pipeline. Coreference resolution was particularly problematic for us, and adding it to the pipeline tended to *reduce* the accuracy of other components in the pipeline and propagating errors, so we were unable to use it in our initial project showcase. Sentence simplification was useful, but not completely accurate and was not able to simplify larger, complex sentences. More recently, we discovered that translation to and from English to Chinese would produce a slightly different sentence than the original, allowing us to improve our training data. Similarly, what if we included information from before Co-reference resolution, and before Sentence Simplification and Clause Extraction, in our training data as well (but not in the knowledge base)? This would allow each sentence to be converted into tens of sentences for the training data, each being similar but demonstrating variations on the same semantic meaning and improving the breadth of the training data, potentially improving the AI's understanding of user inquiries which in turn would allow it to retrieve the correct contextually-relevant facts from the knowledge base when constructing or retrieving its responses in real-time.

One area for improvement that we didn't address with our current implementation is the additional processing of user inquiries, to allow the AI to better understand what the user is asking. We could run the NLP Pipeline on user inquiries at runtime, allowing users to ask slightly more vague questions or ask multiple questions in the same inquiry. Such goals also outline the necessity of a semantic-based approach to the AI, which would allow contextual information (such as information related to the *current* conversation) to be used when searching through the knowledge base and constructing responses. Other important contextual improvements such as the distinction between entities of different categories with similar names – for example, the Medici estate versus the Medici family – or differentiating between Queen Elizabeth I and Queen Elizabeth II in-context – should be considered as well, and might be particularly useful for a semantic approach. We leave the development of contextual analysis for the AI to future work.

In conclusion, our goal was to create a working prototype of the NLP Pipeline and interface it with a retrieval-based chatbot, with less emphasis on the accuracy of the chatbot's responses to user inquiries than on the design process itself, as long as the functional ideas behind the implementation were demonstrated clearly and in such a way that would allow improvements and support/outline further research. We believe that we have succeeded in reaching this goal, and hope that this research paper will serve as a valuable reference for future work on the project.

REFERENCES

[1] K. Baier, C. Hendricks, K. W. Gorden, J. E. Hendricks, L. Cochran, "College students' textbook reading, or not!" American Reading Forum Annual Yearbook, vol. 31, 2011. [Online]. Available: http://americanreadingforum.org/yearbook/11_yearbook/documents/BAIER%20ET%20AL%20PAPER.pdf. [Accessed Sep. 13, 2009].

[2] T. E. Ryan, "Motivating Novice Students to Read their Textbooks," Journal of Instructional Psychology, vol. 33, no. 2, pp. 136-140, Jun. 2006.

[3] E. Vandsburger, R. Duncan-Daston, "Evaluating the Study Guide as a Tool for Increasing Students' Accountability for Reading the Textbook," Journal of College Reading and Learning, vol. 42, no. 1, Fall 2011. [Online]. Available: https://files.eric.ed.gov/fulltext/EJ961149.pdf. [Accessed Sep. 13, 2019].

[4] M. Papastergiou, "Digital Game-Based Learning in high school Computer Science education: Impact on educational effectiveness and student motivation," Computers & Education, vol. 52, no.1, pp. 1-12, Jan. 2009. [Online]. Available:Science Direct, https://www.sciencedirect.com/science/article/pii/S0360131508000845. [Accessed Sep. 13, 2019].

[5] M. Ebner, A. Holzinger, "Successful implementation of user-centered game based learning in higher education: An example from civil engineering," Computers & Education, vol. 49, no.3, pp. 873-890, Nov. 2007. [Online]. Available:Science Direct, https://www.sciencedirect.com/science/article/pii/S0360131505001910. [Accessed Sep. 13, 2019].

[6] S. Rapini, "Beyond Textbooks and Lectures: Digital Game-Based Learning in STEM Subjects," Texas A&M Bush School of Government and Public Service, Texas A&M Univ., College Station, research paper. Summer 2012. [Online]. Available: https://www.esa.org/esa/wp-content/uploads/2012/12/digital-gamebased-learning.pdf. [Accessed Sep. 13, 2019].

[7] J. Roberts, K. Chen, "Learning-Based Procedural Content Generation," IEEE Transactions on Computational Intelligence and AI in Games, vol. 7, no. 1, pp. 88-101, Mar. 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6853332/authors#authors.[Accessed Sep. 13, 2019].

[8] Heilman, Michael, “Automatic Factual Question Generation from Text,” Ph.D. dissertation, Dept. Language and Dept. Information Technologies, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, 2011.

APPENDIX A

DICTIONARY-GEN ALGORITHM

function DICTIONARY-GEN(resolvedText, &sentences, &subjects, &predicates, &modifiers, &keywords, &inverseSubjects) **returns** null

```
#Dependency Tag Numbers (May Change in the Future)
prepositionTagID = 443 #443 = preposition
nominalSubjectTagID = 429 #429 = nominal subj

#Keyword Helpers
nounAndVerbAbreviations = ["NN", "NNP", "NNPS", "NNS", "PRP", "PRP$", "VB"
, "VBD", "VBG", "VBN", "VBP", "VBZ"]
nounAndVerbExceptions = ["is", "he", "him", "his", "she", "her", "hers", "
’", "have", "has", "be", "been", "being", "were"]

spacyDocument = nlp(resolvedText)
spacySentences = list(spacyDocument.sents)
for spacySentence in spacySentences:
  modifierStartIndex = -1
  modifierEndIndex = -1
  subjectStartIndex = -1
  subjectEndIndex = -1
  predicateStartIndex = -1
  predicateEndIndex = -1

  #Sentences
  sentences[hash(spacySentence)] = spacySentence

  startsWithPrep = False
  inMod = False
  predicate = []
  if spacySentence[0].dep == prepositionTagID:
    startsWithPrep = True
    inMod = True

  for token in spacySentence:
    #Keywords
    if (token.tag_ in nounAndVerbAbreviations and token.text.lower() not i
n nounAndVerbExceptions):
      if str(token) in keywords:
        if hash(spacySentence) in keywords[str(token)]:
```

```
        continue
      keywords[str(token)].append(hash(spacySentence))
    else:
      keywords[str(token)] = [hash(spacySentence)]

  # Modifiers - Starts with a preposition - Look for the separating comm
  a
  if(startsWithPrep and str(token) == ","):
    modifiers[hash(spacySentence)] = spacyDocument[spacySentence[0].i:to
  ken.i]
    inMod = False
    modifierStartIndex = spacySentence[0].i
    modifierEndIndex = token.i
    continue

  # Modififers - Ends with a preposition
  if(str(token) == "," and spacyDocument[token.i+1].dep == prepositionTa
  gID):
    modifiers[hash(spacySentence)] = spacyDocument[token.i+1:spacySenten
    ce[len(spacySentence)-1].i]
    modifierStartIndex = token.i+1
    modifierEndIndex = spacySentence[len(spacySentence)-1].i
    break

  #Subjects + Inverse Subjects
  if token.dep == nominalSubjectTagID and subjectStartIndex == -1:
    subjects[hash(spacySentence)] = token
    if str(token) in inverseSubjects:
      inverseSubjects[str(token)].append(hash(spacySentence))
    else:
      inverseSubjects[str(token)] = [hash(spacySentence)]
    subjectStartIndex = token.i
    subjectEndIndex = token.i

  #Predicates
  if not inMod and token.dep != nominalSubjectTagID:
    predicate.append(token.i)

if len(predicate) > 0:
  predicates[hash(spacySentence)] = spacyDocument[predicate[0]:predicate
  [-1]+1]
```

The first attempt of the knowledge bundler mainly involved the use of dictionaries. Dictionaries were preferred here because of their O(1) lookup time and simple implementation. This algorithm is used to fill those dictionaries with the relevant information.

The inputs contain a piece of text and 6 dictionaries. The input text refers to the text from a textbook, or alternate source, after it has gone through the first three steps of our NLP pipeline. The 6 dictionaries are as follows: sentences, subjects, predicates, modifiers, keywords, and inverse subjects. Sentences include the sentences extracted from the text. Subjects include the subjects of those sentences. Modifiers, in this case, refer to sentence level modifiers. “In 1452, …”, would be an example of this and would be parsed out as the modifier for the sentence. Predicates include the verb, the objects, and any extra part that isn’t a subject or modifier. These first four dictionaries have all have their key as the sentence id and the value as their respective data. The inverse subjects dictionary is the opposite of the subjects dictionary in the sense that the key is the subject and the value are lists of sentence ids. This is useful in cases where you want to get all the sentences of a given subject. The last dictionary is keywords. This dictionary is similar to inverse subjects since the key is the keyword and the value is a list of sentence ids. Similarly, this is useful to know information such as which sentences contain the word “painted.” Keywords are the nouns and verbs of a sentence with a few exceptions.

The algorithm starts by going through all the sentences in the input. For each sentence, it stores the sentence in the sentences dictionary. Once added, the algorithm looks to see if the first word in the sentence is a preposition. If it is, the sentence starts with a modifier, so the algorithm parses the text between the first word and the first comma, labelling that as a modifier for this sentence. Then for every token after, the algorithm checks to see if the token is a verb or noun and is not in the exceptions list; if the criteria is met then the word is added to keywords. After that, we check to see if the token is a comma that is followed by a preposition. If so, the sentence ends with a modifier, so the algorithm parses out the reminder of the sentence and labels it as a modifier for the sentence. Then the algorithm checks if the token is a subject. If so, the algorithm adds the subject to both the subjects list and the inverse subjects list. Any token that is not the subject or inside a modifier is then labeled as part of the predicate which is then appended together as the sentence's predicate.

This algorithm has a lot of holes in it but works well as a first prototype (for the most part), but it relies on well-formatted (e.g. human-vetted) input. For example, if a sentence with a starting preposition does not have a comma that closes the preposition, the algorithm will not be able to classify the sentence modifier. Similarly, if the sentence has an interrupting preposition, the algorithm will think everything including and past that preposition is a modifier.

APPENDIX B

QUESTION-GEN ALGORITHM

function QUESTION-GEN(sentences, subjects, predicates, modifiers, keywords, inverseSubjects, & `questionDict`) **returns** null

```
#dictionary containing all of the questions mapped to each intent that the
Rasa bot will recognize
#dictionary - {"intent_name" : ("Source Sentence", [questions])}
questionDict = {}

#generate questions for each sentence
#predicates is from DICTIONARY-GEN
for key in predicates.keys():
  sentence = "".join((token.text + " ") for token in predicates[key])

  subj = None
  subjType = None
  act = None
  obj = None
  objType = None

  sentDoc = nlp(sentence)

  #Get subject, action and object from sentence
  for token in sentDoc:
  #use earliest subject
    if (token.dep_ == "nsubj" or token.dep_ == "nsubjpass") and subj == No
    ne:
      subj = token
    #in most cases, the root is the main action
    elif token.dep_ == "ROOT":
      act = token
    #use earliest and most important object
    elif (token.dep_ == "pobj" or token.dep_ == "dobj") and ((obj == None
    or (obj.dep_ == "dobj" and token.dep_ == "pobj")) ):
      obj = token

  #Name intent for questions
  intentName = str(act) + "_" + str(obj)

  #get extended action, if possible
```

```
#adds on supplemental parts of verbs (like 'go to' instead of 'go')
tok = act
action = ""
action += tok.text
arry = [token for token in sentDoc]
index = act.i
while index+1 < len(arry):
  index += 1
  tok = arry[index]
  if(tok.dep_ == "aux" or tok.pos_ == "VERB"):
    action += " "+ tok.text
  else:
    break

#Get more specific object and object type, if possible
#searches entity list to see if there is an entity that includes the sel
ected object (to get its entire name)
for entity in sentDoc.ents:
 strtest = str(entity.text)
 if strtest.find(str(obj)) != -1:
   objType = entity.label_
   obj = entity.text
   break

# get more specific subject (ex. "Leonardo da Vinci" instead of "Vinci")
for entity in sentDoc.ents:
 strtest = str(entity.text)
 if strtest.find(str(subj)) != -1:
   subjType = entity.label_
   subj = entity.text
   break

questions = list()

#Generate basic questions based on object type and available information
if(subj != None and act != None and obj != None):
#checking for missing information removes about half of the possible int
ents
  if str(objType) == "GPE":
    questions.append(str("Where was " + str(subj) + " " + str(act) + "?"
  ))
  elif str(objType) == "DATE":
    questions.append(str("When did " + str(subj) + " " + str(act) + "?")
  )
```

```
elif str(objType) == "PERSON" or (not type(obj) == str and obj.dep_ ==
"pobj"):
  questions.append(str("Who " + str(act) + " " + str(subj) + "?"))
else:
  questions.append(str("What did " + str(subj) + " " + str(act) + "?")
 )

if str(subjType) == "GPE":
  questions.append(str("Where was " + str(obj) + " " + str(act) + "?")
)
elif str(subjType) == "DATE":
  questions.append(str("When did " + str(obj) + " " + str(act) + "?"))
elif str(subjType) == "PERSON":
  questions.append(str("Who " + str(act) + " " + str(obj) + "?"))
else:
  questions.append(str("What did " + str(obj) + " " + str(act) + "?"))

questions.append(str("What is " + str(obj) + " to " + str(subj) + "?")
 )
questions.append(str("What " + str(act) + " " + str(subj) + "?"))
questions.append(str("What " + str(obj) + " " + str(act) + " " + str(s
 ubj) + "?"))
questions.append(str("Why did " + str(subj) + " " + str(act) + " " + s
 tr(obj) + "?"))
questions.append(str("How did " + str(subj) + " " + str(act) + " " + s
 tr(obj) + "?"))
questions.append(str("How does " + str(subj) + " " + str(obj) + "?"))
questions.append(str(str(subj) + " " + str(act) + " " + str(obj) + "?"
 ))

translations = [SimplifyText(question) for question in questions]
 # '&quot' and '&#39' come up in some translations. Seem to denote pro
 per names (ex. 'Mona Lisa') or possessive
translations.append(SimplifyText(sentence).replace("&quot;","'").repla
 ce("&#39;","'"))
for translation in translations:
  questions.append(translation.replace("&quot;","'").replace("&#39;","
 '"))

#ensure intents aren't overwritten
while(intentName in questionDict.keys()):
  intentName += "_"
questionDict[intentName] = (sentence, questions)
```

The QUESTION-GEN algorithm is used to generate the questions that will be used to train the Rasa-bot's intent recognition. The input for this algorithm is the predicates formed by the DICTIONARY-GEN algorithm. Each predicate is reformed into a sentence string and fed into SpaCy NLP to tokenize each word. The sentence is then searched to find the tokens representing its subject, main verb, and object. This is done by checking the syntactic dependency tag (._dep) of the token. Subjects either have a tag of "nsubj" or "nsubjpass" (nominal subject or passive nominal subject). The main verb of the sentence usually has the "ROOT" tag, which denotes the token that connects multiple noun phrases (i.e. the subject and the object). The object of the sentence will either be tagged as "dobj" or "pobj" (direct object or object of preposition). There can be multiple objects in a sentence and it was found through testing that this algorithm was more effective if the first object of preposition is used, if there is one available. Note that separating the predicates with DICTIONARY-GEN already broke up most sentences that originally had multiple clauses and would therefore have multiple subjects, verbs, and/or objects. Since the subject and object found in this way would only be a single token, the list of entities in the sentence generated by SpaCy is searched to find the full name of the subject/object, if there is one available (ex. if "Leonardo" is found as the subject, it might be expanded to "Leonardo da Vinci"). Similarly, if the main verb is immediately followed by other verbs or auxiliary verbs, they are combined to form a more complete action (ex. "go to" instead of "go").

From there, the subject, action and object are then inserted into several question formats in order to generate questions. If the algorithm did not find a subject, verb or object in the sentence, no questions will be generated from the sentence and the previous process will start again with the next predicate from DICTIONARY-GEN. There are several general question formats that will be formed for every sentence such as “Why did [SUBJECT] [ACTION]?”, but there are also questions that will only be formed depending on the type of entity that SpaCy tagged the subject or object to be. For example, if the object is tagged as a location, the question generated will be “Where did [SUBJECT] [ACTION]?”. The sentences formed in this way may not be grammatically correct, but it was found that since the Rasa bot searches for keywords in input, it is trained better with quantity over quality. All questions generated in this way are then added to a list. To improve the training data, synonymous sentences are then generated by using the SimplifyText() function on each question, which translates the question into a foreign language and then translates it back into English, altering the structure of the question without changing the meaning. The retranslated sentences are then added to the list of questions.

Finally, the intent name that the Rasa bot will use to group these questions will be generated as “[ACTION]_[OBJECT]” with additional “_” characters added on if there already happens to be an intent with this name. An entry into a dictionary containing all of the intents/questions is then made with the intent name as the key mapped to a tuple containing the source sentence and the list of questions generated from that sentence. This process is then repeated for every predicate formed in DICTIONARY-GEN.