Minecraft Question Answering System

*David Schulz*

CS4980 - Natural Language Processing

Sean Jones

May 10, 2022

Project Summary 3

Background 3

Data 6

The System 8

Performance 10

Future Work 12

Lessons Learned 12

# Project Summary

This system takes a question about Minecraft gameplay mechanics and extracts an answer from a given knowledge base. The system will prompt a user to enter one of five categories, blocks, items, mobs, places, and gameplay, and then it will prompt the user to enter a question about the category they chose. The knowledge base is split into the same categories, and tf-idf tables are made for each category from extracted keywords in the data. The top three files are chosen by scoring the keywords from the question with the tf-idf table of the user-chosen category. The question and all sentences from the three files are then embedded into vectors, and every sentence is compared with the question using cosine similarity. The sentence with the highest similarity is chosen as the answer. This system was created in hopes of centralizing the information about Minecraft gameplay mechanics into one simple question answering model, removing the need to search the web and potentially read several articles to find the answer. I hope to eventually implement this model directly into the chat of a Minecraft server so that a player doesn’t even need to leave the game window to find an answer.

# Background

Both the knowledge database and questions get tokenized and lemmatized to perform keyword extraction. Part-of-speech tagging and syntactic parsing can then be used to determine the answer type. Many words can often represent more than one type, so other words in the question need to be considered. A lexical dictionary such as WordNet can be used for understanding the context.

Once the question type has been identified, an information retrieval system is used to find a set of documents containing the correct keywords. A tagger and NP/verb group chunker can be used to verify whether the correct entities and relationships are mentioned in the found documents. Selecting documents with tf-idf tables based on unigrams and bigrams is also a possible approach. Only the relevant paragraphs are selected for ranking.

A vector space model can then be used for classifying the potential answers. Transformer models are used to convert paragraphs, sentences, and words into vectors. It will check if the answer is of the correct type as determined in the question type analysis stage. An inference technique can also be used to validate the potential answers. A score is then given to each of these according to the number of question words it contains and how close these words are to the question. The more and the closer the better. The answer is then translated into a compact and meaningful representation by parsing.

NLTK’s functions can split both the knowledge database and questions into tokens, remove stop words, and then stem all remaining tokens. nltk.word\_tokenize() can tokenize the sentences. nltk.corpus has a stopwords module that contains a list of stop words that should be removed. nltk.stem has a LancasterStemmer, PorterStemmer, and SnowballStemmer that can stem the tokens. The LancasterStemmer is the most aggressive stemmer while the SnowballStemmer is a slightly more aggressive version of the PorterStemmer. Various transformer frameworks and models, such as word2vec, BERT, GloVe, and Universal Sentence Encoder, can be used to convert sentences and tokens into vectors. Some modules, such as scipy and sklearn, have cosine similarity functions that can be used to compare the question’s vector with vectors from the knowledge database.

A paper titled “Closed Domain Question Answering System Using NLP Techniques” by Mukesh Raghuwanshi presents implementation methods and experimental results with analysis for a closed-domain QA system which handles documents related to the education system. Although Mukesh doesn’t name specific tools and frameworks he used to create his system, he describes the techniques and architecture well enough to figure out what he might have used. His system’s architecture begins with a corpus of questions and a knowledge database of domain-related documents. Both are tokenized, have stop words removed, part-of-speech tagged, and stemmed. The resulting lists of tokens from both are considered the extracted keywords. The keywords from the knowledge database are compiled into an “index term dictionary” which is a table containing two columns: word and file names. Word is an extracted keyword and file names is a list of the files which contain that keyword. He then performs document retrieval by matching the keywords from the question with the keywords in the index term dictionary. Only documents which contain all of the question’s keywords will be selected to potentially contain the answer. He then reranks the selected files using the Jaccard similarity function. To perform the final answer extraction, he uses the POS tagging from before and finds the string that matches the question’s POS tags the closest.

A paper titled “Closed Domain Question Answering for Cultural Heritage” by Bernardo Cuteri is about a novel architecture for closed-domain question answering and a possible application in the cultural heritage context. Bernardo also doesn’t name the specific tools and frameworks used, but he describes the techniques and architecture. The system is split into five tasks: question processing, template matching, query expansion and contextualization, query execution, and answer creation. First, the question is tokenized and tagged with part-of-speech (POS) tags. Then a natural language parser extracts grammatical relations (typed dependencies) from the text. Questions are then classified and transformed into formal queries by template matching. In this context, templates represent the structure of typical questions. If a certain template is matched, things can be inferred about the question type. Every question template is accompanied with a formal query in which some slots are empty and are filled with terms extracted from the question that matches the template. Sometimes, to be effective, the query must be expanded with context information and/or word semantic information. His solution is to expand the query by using synonyms, hypernyms, and other word semantic relations. In his model, the query is executed against a structured knowledge base. Query results can then be used to build a natural language answer with a mechanism like template matching, but in the inverse direction.

A paper titled “AQUA: A Closed-Domain Question Answering System” by Maria Vargas-Vera and Miltiadis Lytras describes AQUA, an experimental question answering system. The authors also don’t name the specific tools and frameworks they used, but they describe the techniques and architecture very well. AQUA combines NLP, ontologies, logic, and information retrieval technologies in a uniform framework. The architecture is split into four parts: user interaction, question processing, document processing, and answer extraction. During user interaction, the user inputs the question and validates the answer. Question processing is performed in order to understand the question asked by the user. This “understanding” of the question requires several steps such as parsing the question, representation of the question, and classification. For document processing, a set of documents are selected, and a set of paragraphs are extracted. This relies on the identification of the focus of the question. In the answer processing phase, answers are extracted from passages and given a score. The answer with the highest score is returned to the user.

For the sake of time and simplicity, my system will be based off of the first paper, “Closed Domain Question Answering System Using NLP Techniques”, the most. Given more time, I would have taken a more complex approach with an ontology like AQUA. Like the first paper’s approach, my system will tokenize and stem the knowledge database, but instead of creating an index term dictionary, it will create a tf-idf table, where the rows are all stemmed tokens that exist in any of the files and the columns are the files. The question will also be tokenized and stemmed, and then the top three files that are most likely to contain the answer will be chosen by getting all rows in the tf-idf table where their token is in the question and calculating the sum of those rows by column. The top three columns are the three files. Once the files are chosen, they will all be read and compiled into a list of sentences. Instead of matching the question with the best answer by POS tags like the first paper, those sentences and the question will be embedded into vectors using a sentence transformer. Then the question vector and each sentence vector will be compared using a cosine similarity function. The sentence with the highest cosine similarity will be chosen as the answer.

# Data

Since this project requires two different types of data, the knowledge database and the corpus of possible questions, data had to be collected from different types of sources. The knowledge database was captured from the following websites:

* [www.minecraft.fandom.com/wiki/Minecraft\_Wiki](http://www.minecraft.fandom.com/wiki/Minecraft_Wiki)
* [www.game.guide/minecraft](http://www.game.guide/minecraft)

The corpus of possible questions was captured from the following websites:

* [www.quizlet.com](http://www.quizlet.com)
* [www.allthetests.com](http://www.allthetests.com)
* [www.gamefaqs.gamespot.com/pc/606524-minecraft/answers](http://www.gamefaqs.gamespot.com/pc/606524-minecraft/answers)

None of the data from the sources, both the question corpus and knowledge base, has one specific author because everything has been created and edited by many people over the years. However, during my initial search for the data, I stumbled upon a GitHub repository that did the web scraping for me, created by someone named Corentin Dumont. Their repository also contains the skeleton of an ontology, which will be nice for future work, but I won’t be using it for the scope of this project.

Corentin’s repository has the knowledge database and question corpus in two separate folders. The knowledge database folder contains three folders, one for each of the original website sources. There are three instead of two because the data was scraped back when the Minecraft Fandom Wiki was split between two sites: gamepedia.com and wikia.com. In each folder, there are more folders that split the information into topic categories from the game (blocks, items, mobs, building, etc.). Those folders may also have more folders that split the information further (natural, nether, ores, mechanisms, etc. in the blocks folder). The lowest level of the folder tree contains text files for each specific entity or topic. In total, there are 1384 files in the knowledge database. The formatting of the text data is the same between all 3 source folders. Each text file contains raw sentences of information from their corresponding sources. Each line in a file is a single continuous idea, so the sentences on one line are related, but sentences on separate lines are not as closely related. The following is an example of a sentence in the knowledge database:

“Diamonds are one of the rarest materials in Minecraft.”

As for the question corpus, there are three files in the folder, but only the csv file is relevant to this project. Its columns are different topics in Minecraft (recipe, blocks, mobs, items, etc.) and the rows are questions about those topics, so some columns contain more questions than others, which means that columns in many of the rows are empty. In total, there are 1030 questions in the file. The following is an example of a question in the blocks category:

“What can I use to mine a Diamond?”

Before reading the data, the knowledge database text files that contain information about the same topic from different sources will be put together into one file, since the distinction between sources is irrelevant. Both types of data will be read with the pandas read\_csv() function, even though the knowledge database is made up of text files. Having everything in DataFrames makes it easier and more efficient to work with. After the data is read, the knowledge database will be pre-processed through tokenizing, stemming, and stop word removal so that the relevant data can be put into tf-idf tables, where each keyword maps to a tf-idf score for each file in the category. The question corpus will be pre-processed through tokenizing, stemming, and stop word removal to extract the same kind of keywords from each question. The following are the two previous examples after pre-processing:

['diamond', 'one', 'rarest', 'materi', 'minecraft']

['use', 'mine', 'diamond']

As a result of training, the final model should be able to read a question input by the user, pre-process it, use keyword extraction to match it with documents from the knowledge database, use the correct tf-idf table to determine which document has the best answer, embed the question and all sentences in the files as vectors, and extract the correct answer using cosine similarity between the vectors of the sentences and the vector of the question.

# The System

For the “training” phase, the system creates tf-idf tables as pandas DataFrames for each category. These tables will be used to select three files that are the most likely to contain the correct answer by scoring the keywords that appear in the question for each file. In these tables, the rows are all tokens, stop-word-removed and stemmed with NLTK, that exist in any of the files under its respective category, and the columns are the files. The tf-idf scores reflect how important a keyword is to a document in a corpus. They are calculated by multiplying the keyword’s term frequency score by its inverse document frequency score. The term frequency is how often a keyword appears in one document, while the inverse document frequency is the inverse of how often a keyword appears at least once in every document. The tf and idf scores are calculated with the help of various pandas functions that make calculations with large amounts of data simple and efficient. These tf-idf tables, represented as DataFrames, are saved to csv files so that they can be used repeatedly without having to be reconstructed every time. The files must be reconstructed, however, whenever the knowledge base is modified. The following is a small portion of the tf-idf table for the Mobs category:

Graphical user interface, text, chat or text message

Description automatically generated

Graphical user interface, text

Description automatically generated

After the tf-idf tables have been created or retrieved from the csv files, the system prompts the user to enter one of five possible categories: blocks, items, mobs, places, and gameplay. This will narrow down what their question will be asking about. This is why there are five different tf-idf tables, one for each category. Only the tf-idf table of the selected category is then used for document selection. This is beneficial over using a single tf-idf table for a few reasons. The first reason is that it’s less memory intensive; there are fewer rows and columns in the table. The second is that it removes some potential error when performing document selection; certain keywords might have higher tf-idf scores for files from different categories for different reasons. For example, the word “nether” could be referring to nether brick (block), nether quartz (item), or nether portal (place).



Then the system will prompt the user to enter their question. The question will also be tokenized, stop-word-removed, and stemmed with NLTK, and then the top three files that are most likely to contain the answer will be chosen by getting all rows in the tf-idf table where their token is in the question and calculating the sum of those rows by column. The top three columns are the three files. Once the files are chosen, they will all be read using pandas read\_csv() function and compiled into a list of sentences using a module called TextBlob, which takes a string and splits it into a list of strings that it believes to be the sentences. The question and those sentences will be embedded into vectors using a BERT-based sentence transformer from the sentence\_transformers module. Then the question vector and each sentence vector will be compared using the cosine\_similarity function from sklearn. The sentence with the highest cosine similarity will be chosen as the answer.



# Performance

As far as the performance of the training process goes, it takes about 45 seconds for the system to create the tf-idf tables.

Once I got my model in a state where it could give an answer, correct or not, obtained from the knowledge database, I ran it through the 3 categories of questions from the QuestionsCorpus that shared categories with the knowledge database: blocks, mobs, and items. I went through every Q/A combination and manually determined if each answer was correct and fully answered the question. As expected, the performance was not great. Out of 460 questions, only about 57 were fully correct, giving an accuracy of about 12%. Some of the correct answers are shown below.

Correct Answers:

* Q: Can I mine a Diamond without a Pickaxe?
  + A: Blocks of diamond can only be mined using an iron or diamond pickaxe.
* Q: What is the most rare Ore in Minecraft?
  + A: Emerald Ore is the rarest block in Minecraft.
* Q: What has the highest blast resistance?
  + A: End Stone has the highest blast resistance of any block that can be blown up easily.
* Q: How do Pressure Plates help?
  + A: A Pressure Plate is an item that can be used to activate certain features when it is stepped on.
* Q: What else destroys Cobwebs besides Swords?
  + A: Water, lava, and pistons will destroy the cobweb as well.

This was expected because the way the model works is it chooses 3 files that are most likely to have the answer based on the tf-idf tables for each category. It then reads every line of all 3 files and parses out all of the sentences. It then converts the question and every sentence from the 3 files to vectors and finds the cosine similarity between the question and every sentence. The most similar sentence is chosen as the answer. This approach creates a very naïve model.

As a result of this performance, I realized that the knowledge database still needed severe cleaning and standardized formatting. Some sentences weren’t information at all, but just scraped labels of images that were on the websites, so they were just a few words instead of complete sentences. I decided to manually clean the files about chickens and ask my own questions about them to see if that improved performance. Since all text in those files were now complete sentences with pure information, I could word the questions somewhat similarly to how they were worded to get full, correct answers more consistently. Some of those examples are shown below.

Chicken Questions:

* Q: How can chickens be bred?
  + A: Chickens can be bred using seeds.
* Q: What do chickens drop when they are killed?
  + A: Chickens drop 0-2 feathers and 1 raw chicken when they’re killed.
* Q: What is the chance of a baby chicken to spawn from an egg?
  + A: There’s a 1/8 chance of a baby chicken hatching from an egg.
* Q: How often does a chicken lay one egg?
  + A: A chicken will lay one egg every 5 to 10 minutes, unless it’s part of a chicken jockey.

The current performance of the model is not close to my vision statement’s desired performance at all. Very few questions actually get answered fully and correctly. With more dedicated time and some better resources, other approaches would create a much better model, like using an ontology for example.

# Future Work

Although some answers can be correct when the knowledge base is cleaned and formatted well and the question is worded just right, the full and correct answer rate in general is very low. This is an inherent effect of the architecture of this system, since finding the answer relies almost completely on just the question’s cosine similarity to the sentences in the knowledge base. In other words, the answer will only be correct if the question is worded like the sentence that contains the correct answer in the knowledge base. For future work on this, which I do intend to do, the system architecture will be completely redesigned to work around an ontology, which maps entities together by their relationships. Creating the ontology for something at this scale requires repetitive, time-consuming, manual entry, which is why I was unable to use it in the final system this quarter. Although I plan to redo the system completely, creating the current system wasn’t necessarily a waste of time, since it still allowed me to learn a lot about NLP techniques.

# Lessons Learned

Going into the project, I’ll admit my expectations were higher than they should have been. I was hoping to have a system with an extremely high correct answer rate. This was partially because I thought I would have enough time to implement a more robust method with an ontology. After a few days of research and starting the ontology, I realized it required too much time and repetitive manual construction to be completed by the end of the quarter. Luckily, I did some more generic research and found approaches that were a lot less accurate, but simpler and easier to implement. The approach I did end up taking still allowed me to learn much about NLP tools, frameworks, and techniques, such as NLTK’s tools with tokens, sentence embedding models, and the cosine similarity algorithm.