Mad Libs Generator: Report

Zeline Tricia Bartolome and Evalyn Berleant

December 2019

1 Introduction

Mad Libs is a game in which one person prompts others to give words that fit particular categories or parts of speech. Those words are then used to fill in the blanks of a passage, often leading to comical or nonsensical results.

We created a Mad Libs generator that analyzes a passage of text for parts of speech and randomly selects words to replace with user input. The text can be given by the user or selected from a list of pre-chosen passages. These options open up new possibilities for the game, and the random selection of words to replace allows players to reuse the same passage for Mad Libs repeatedly without knowing where the words they input will go.

We aimed for the project to be entertaining, as the original game is meant to be, while also encompassing different aspects of natural language processing in Python. This project involves a POS tagger that combines different taggers from NLTK. We also further classify verbs based on the context from the tags of the words following them.

2 Implementation

2.1 Choice of Passage

The file that runs the program and interacts with the user is main.py. The user can choose the passage from which the Mad Libs will be generated. The user can either paste the text of a passage, choose from the files in the passages folder, or let the program randomly choose a file from the passages folder. If the user wants to input their own passage, they must enter a line that says ‘END’ once they are done entering the passage. We had to implement this signalling for the end of the passage because normally only one line of input can be read at once, so pasting a passage with multiple lines would be read as separate inputs. If the user elects to choose a file from the passages folder, the name of each file in the folder is printed along with the first fifty characters in the file.

2.2 Tokenizing and Tagging

We created a part of speech tagger that combines NLTK’s trigram, bigram, and unigram taggers with a backoff algorithm. As a last resort, we use nltk.DefaultTagger(‘XX’) to tag unrecognized tokens with ‘XX’ instead of an actual tag because we do not want an inaccurate tag to ruin other parts of the program. For example, if we tagged non-nouns with ‘NN’ then it might end up being one of the words the user can replace, producing ungrammatical results. Our tagger is trained on NLTK’s Brown Corpus, which is pre-tagged using the Brown tagset. Because the Brown tagset is more detailed than necessary, we roughly convert the tags that we were planning to use into their corresponding Penn Treebank tags before training the tagger. The tagger is then pickled and stored in the file pos_tagger_brown.pkl.

After training the tagger, tokenizing and tagging are done in the MadLibs class found in text_processor.py. We started tokenizing by using nltk.word_tokenize() to split the raw string into tokens which are either words or punctuation. However, later, during the process of printing the completed output to the user, we realized that we needed a way to keep track of new lines in the original passage so that each line is still separate in the finished output. Therefore, we wrote another function for tokenizing that splits the raw string of a passage into lines first and performs nltk.word_tokenize() on each line before combining the lines with a separate token for ‘\n’.

In addition to loading the pickled tagger from pos_tagger_brown.pkl and using it to tag the tokens in the passage, we formed and applied a heuristic process for determining whether a verb is transitive or intransitive. The code for this process is in the determine_transitive() function of the MadLibs class. While iterating through the tagged tokens, the function uses a variable called verb_index to keep track of the last seen verb. If it is followed by a noun, pronoun, determiner, or article before any punctuation, conjunction, or preposition, then the verb is marked as transitive with ‘-T’ added to the end of its tag. Otherwise, it is marked as intransitive with ‘-IT’ added to the end of its tag.

2.3 Word Replacement

After tagging the words in the passage, we randomly selected words to be replaced with user input. We only wanted words with specific tags to be replaced because some words, such as pronouns and linking verbs, might make the passage too incomprehensible or grammatically incorrect if they are replaced. Therefore, only common nouns, adjectives, adverbs, interjections, and verbs are to be replaced. These are the specific tags we decided to include for replaceable words:

Tag	Definition
JJ	Adjective
JJR	Adjective, comparative
NN	Noun, singular or mass
NNS	Noun, plural
RB	Adverb
RBR	Adverb, comparative
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present

We did not want all words with these tags to be replaced because that would give the user too much work and remove nearly all the context from the original passage. Therefore, for each of the tokens with these tags, there is only a 40% chance of it being replaced by the user. If the word is to be replaced, then its (token, tag) pair is stored as a key in the dictionary instance variable word_replacements of the MadLibs class.

Next, for each word we want to replace, we prompt the user to give replacements that match the original word's part of speech. The program gives prompts using the tag descriptions from the Penn Treebank tagset. Since the tag descriptions use terms that may not be familiar to many people, we also have the option to enter 'h' for help. Then the program will print examples of the appropriate tag. The tag descriptions and examples were pulled from a help function in NLTK, which prints out a description and set of examples for a given tag. Since we wanted to extract this information and parse it into our own data structures for later use, rather than printing to standard output, we had to use the redirect_stdout() function from Python's contextlib module.

The user's word replacements are stored as values to the (token, tag) keys in the word_replacements dictionary. To replace the tokens in the original passage with the words given by the user, we iterate through the list of tagged tokens in the passage. For each (token, tag) pair, if it exists as a key in the word_replacements dictionary, then we append the value to that key to a list. We signify that it was a replaced word by surrounding it with asterisks. If the (token, tag) pair does not exist as a key in word_replacements, we just append the original word to that list. The list is then put into a readable format by the to_string() method and printed out to the user.

2.4 Printing the Output

As written above in 2.2 Tokenizing and Tagging, we left in tokens to signal where there should be a new line. This tokenization allows the program to print separate lines as they appeared in the original passage. However, we could not simply convert the tokens into a string by joining everything by a space. Then each new line would start with a space, and each punctuation token would be surrounded by spaces. We made sure to check for these things while iterating through the tokens to convert the list to a string. This process is done in the to_string() function of the MadLibs class in test_processor.py. Once the output of that function is printed, the program is done.

3 What Went Right and What Went Wrong

3.1 Problems Solved

One thing we had to account for is the fact that sometimes the repetition of a certain word within a passage is significant. For example, in the song "Shia LaBeouf" by Rob cantor, there is a part that says, "You can see there's blood on his face. My God, there's blood everywhere!" These lyrics would make much less sense if only one instance of blood is changed. Therefore, we decided to implement the program in a way that would replace every instance of a word in the passage instead of replacing just one token. We were able to do that by iterating through the original list of tokens and using the replacement if the original token is in the set of words to replace. Because certain words can function as multiple parts of speech (eg, *face* may be a noun or a verb), we only use the replacement if it matches both the word and the tag of the token that was originally replaced.

The Penn Treebank POS tagset takes many classifications into consideration, such as tense, number, and point of view. However, it does not have separate tags for transitive and intransitive verbs. As a result, we had to make our own function to determine whether a verb is transitive or intransitive, and the function does work in many cases.

3.2 Room for Improvement

Unfortunately, there are some situations in which the determine_transitive() function will not work. It searches for a noun after the verb, but sometimes the noun that the verb is acting upon occurs before the verb itself. For instance, in *The Princess Bride* by William Goldman, he writes, "Every shriek of every child at seeing your hideousness will be yours to cherish." In this case, "cherish" is theoretically performed on a "shriek," but since "shriek" occurs before the verb, the function marks "cherish" as intransitive. Another example involves passive voice. For example, Goldman also writes, "The next thing you will lose will be your left eye followed by your right." The function would mark the word "followed" as intransitive, but it would make more sense to replace it with a transitive verb because the action is being performed on the left eye.

Another case in which the determine_transitive() function fails is when there is a phrase that is used in place of a noun. For example, infinitives can function as objects, as in the sentence "I want to eat" where "want" is the verb and "to eat" is the object. One possible solution would be to mark a verb as transitive if it is followed by an infinitive, but not all transitive verbs can accept infinitives as objects. If we replace "want" with another verb, regardless of whether it is transitive or intransitive, "to" will mostly likely be interpreted to mean "in order to" rather than as an infinitive marker. The sentence may still make sense, but it will be very different semantically.

One other aspect of the program that might need a change is the probability that a word will be replaced. Replacing too many words may create too much work for the user and morph the original passage beyond recognition. On the other hand, replacing too few words may leave the passage to close to the original, which goes against the purpose of the program. We tried testing with different text files to find a good balance. However, it is difficult to find a probability that works well with a variety of passages because some passages may have a bigger proportion of words with tags among those we want to replace.

Another issue is the uneven distribution of replaced words. The program does not take into consideration whether the previous has just been replaced. As a result, multiple words in a row may be changed. Losing that much context in one part of the passage can make it harder to understand.

The help function also has room for improvement. Currently, it gives a short list of examples of the tag. However, for verbs, the examples given are not necessarily accurate in transitivity. Ideally, the list of examples could also be randomized so that they aren't the same every time.

Sometimes, words take different forms depending on their context. For example, the determiner 'a' switches to 'an' when the following noun begins with a vowel. In the future, we could try to account for more of these types of context dependencies. Additionally, if the same word is used in different forms in a passage, this may not be maintained in the final passage. For example, one passage may use the words 'walk', 'walked', and 'walking', but in the final passage they could be replaced with completely unrelated words, like 'assume', 'created', and 'running'. Similarly, some words that are different parts of speech might share roots or semantic relationships, such as 'blood' and 'bleed'. In this case, even if we realized this relationship, if 'blood' were replaced by 'laptop', there is no verb form of 'laptop' that matches the semantic relationship between 'blood' and 'bleed'.

4 Project Team

Our team members are Zeline Tricia Bartolome and Evalyn Berleant. Zeline set up the GitHub repository with the two .py files and the passages folder with two .txt files to start. In main.py, Zeline coded most of the user interaction and the reading of files. Evalyn coded the help function and the conversion from the tag name to the part of speech description that is used to prompt the user.

In text_processor.py, Evalyn coded training, dumping, and loading the tagger, as well as the tag conversion from the Brown tagset to the Penn Treebank tagset. She also coded much of the methods for processing, tagging, and replacing the words in the passage. Zeline created the method for determining whether a verb is transitive or intransitive. She also coded the tokenizing of words in a passage and much of the to_string() method, which puts the tokens back together into a readable passage.