Bailey Johnson, Vicky Steger, Joe Antaki, Mike Gardner

Marcus Verhagen

LING 131A - Intro to Natural Language Processing

19 December 2019

Murder Mystery Spoilers

1. What We Did and Why

For our final project, we decided to create a "Murder Mystery Spolier". The idea was that we could import a file which contained a novel or TV script about a murder, and return the killer and perhaps other information, such as who the victim(s) were, how they died, and who was the detective or main character. We chose this project because we had done data processing for the in-class assignments, and knew that there was plenty of murder mysteries in media out there for us to test it on. We decided to use short murder-mystery stories taken from Project Gutenberg as our source material.

2. How We Implemented It

The first thing we needed to do was import our data. There were some minor problems with that which will be expanded on below, but after we they were sorted out, we decided to isolate the sentences that mentioned "murdery" words (such as kill, died, poison, murder, etc) as well as the two on either side (in case the sentence was something like "I killed him", and the "I" refered to someone mentioned in a previous sentence). After that, we went through the file and tagged all of the proper nouns using spaCy, filtered the sentences gained from the "murdery words", and only took those that had a proper noun in them. We tried to find a package that could tag theta-roles for us (in order to find the difference between the murderer vs the murdered), but were unable to find anything that would run without importing libraries of machine learned data. So instead we tackled the problem using linguistic intuitions about sentence structure. We began by splitting sentences to pre and post target word. Unless we found the passive form of the verb in the sentence, the entities before the word would be the murderers, and the entities after the

word would be the murdered individual. Then it was just a matter of tweaking how we parse for entities and pronouns through the sentence fragments, since pronouns need to access the larger context of the sentence, and returning out likeliest results.

3. What Went Right and What Went Wrong

As mentioned above, there was some trouble originally finding the data for this project. Our first thought was to use plays or tv show/movie scripts, as we thought that we would be more likely to find a clear "confession" sentence either in the dialogue ("The butler did it!") or in the stage direction ([Rachel STABS Sarah]). However, we soon found that there were very few well-annotated transcripts available for free on the internet. There were a few episodes of Law & Order, but not in an easily-readable (for Python) form. We also considered using some Scooby-Doo transcripts we found (as there would be a clear reveal of the culprit, even if there wasn't necessarily a murder), but upon closer inspection, found that there were varying levels of quality, and only about 12 episodes in total that were even close to being fully complete.

So we turned to project Gutenburg, and chose several murder mystery stories. After we had those, it was fairly easy to isolate the sentences that had the murdery words in them. There were also some hiccups with getting spaCy to correctly recognize certain names that look like other words or names that had "Mr." and "Mrs." in front of a last name, but we were able to overcome that with adjustments to our code. Once that was done, we could isolate the sentences that had both murdery words and proper nouns in them.

The real difficulty was with the theta roles. When we started out, wehoped that we could find a program that went through the text file and tag the theta-roles of each word - similar to the POS-tagging that was included in nltk. During our initial research for the project, we thought that we had found that in VerbNet. But as we got deeper into the project, we began to realize that VerbNet isn't quite what we wanted it to be. Rather than tagging each word in a sentence, it instead acted more like WordNet or SentiWordNet and returned all of the different theta-roles that a word *could* be.

Our conclusion was that if we wanted the theta-roles tagged, we would have had to write the program ourselves, which seemed like it would be too large a project for this small goal. So we looked at alternate ways to narrow down our character lists to the people most likely to be the murderer. We settled on the linguistic intuition that killers will precede murderous words (acting as agents), and victims will commonly succede murderous words (acting as themes). We did not want just one word prior to a murderous words and just after, so we had to construct and decompose contexts surrounding target words. We could then filter by conditions such as passive construction. Ultimately returning 2 sets, one of theme related entities, and one of agent related entities. For every story we ran through all the target sentence contexts this way, and returned the entities most commonly in those supposed roles for our target words; which could lead to an accurate generalization some of the time. But most-common turned out to be an overgeneralization, as often the most common named entity is the main character, or we were unable to find the referent of the pronoun or name fragment used in the murder sentence, even with the extra context. We captured some pieces of the intended goal, but our simplistic model isn't a strong enough predictor to weave through the intricate variations of different stories and different authors.

In the end, we were able to get a program that consistently worked better than random chance at finding out who the killer was, as shown in the two runs below, though its results were not as strong as we had hoped.

1st run:

Killer prediction accuracy: 0.2

Random chance accuracy: 0.05

Victim prediction accuracy: 0.35

Random chance accuracy: 0.25

2nd run:

Killer prediction accuracy: 0.2

Random chance accuracy: 0.1

Victim prediction accuracy: 0.35

Random chance accuracy: 0.0

4. Who Developed What

Preprocessing and General Group Management (Vicky)

I created two methods that divide a text into sentences and either stem or lemmatize the important words in each sentence. I wanted to give us a way to catch words like "murder" no matter what form they came in (for example "murdering", "murdered", and maybe even "murderous"). I experimented with two different stemmers first and decided on PorterStemmer for simplicity's sake. I finished that quickly so that the other group members would have something to work with while I developed the more in-depth lemmatizer method. This method required knowing the parts of speech of every word that was submitted to it. This originally was an issue, because the text wasn't tagged and the NLTK POS tagger didn't use the same tags as WordNet's lemmatizer, so I found a way to map from the NLTK letters to the WordNet letters.

I also created two methods to produce a list of murder-related words in the text. The first just returned a list of words that I felt were likely to produce results when used on our initial testing text. This was submitted to allow other group members to continue working while I looked for a cleverer solution. I had hoped to use the list of troponyms of "kill" that I could see in the online version of WordNet, but there is no NLTK method to access the troponym lists, and I felt copying the list by hand wouldn't be appropriate for a final project. I explored the available methods and decided that the best bet would be to use the shortest_path_distance to score how close various words in the text were to a noun, verb, and adjective that we wanted. After some experimentation with different target words, I picked "murder" as a noun, "murder" as a verb, and "fatal" as an adjective. I then tagged all of the words in the text (because WordNet requires knowing part of speech to get appropriate results for this), got their WordNet synset, and cycled

through the synset to find the shortest path to those target words from the word being checked. I initially had trouble because the default for shortest_path_distance is None instead of an int, which made using min_num to compare values more complex, but I was able to overcome it by setting my own default values. After experimenting to figure out what values still seemed relevant to our purposes, I decided to return any words with a shortest_path_distance of 3 or less. This meant that even if a text had a very creative way of killing the victim, like tomahawking (to take an example from the kill troponym list), we still had a chance of finding the appropriate sentence and the killer.

Finally, I made a method that searched each sentence in the text for any of the selected murder words. It returns a list with the murder word found, the sentence with the word, and the two sentences on either side of it. The extra sentences were included for context, to avoid problems from sentences with pronouns ("I am the killer" in isolation wouldn't help us, for example). Originally, I wanted to filter out any sentences with modals to avoid hypotheticals like "I could kill you, I'm so angry"). However, I wasn't able to find a way that could do that quickly without also filtering out sentences where the hypothetical wasn't attached to the murder word, such as "I did what I could, murdering the victim was harder than I expected." Since this method would be run before we really started analyzing the sentences, I didn't want it to add too much time onto the overall process.

Outside of coding, I helped organize the group to ensure we met regularly and had a plan to divide up the project into manageable chunks.

Named Entity Recognition (Joe)

This module picks out all the human characters from the sentences that were deemed relevant by preprocessing.murder_sents. It then tabulates the mentions of each character in order to reveal the most important actors, and to indicate potential murderers and victims.

preprocessing.murder_sents is run to get a list of each sentences (and its surrounding sentences, for context) that contain words that indicate murder. Each of these sentences is put through spaCy’s linguistic analyzer. From the analyzed sentence, named entities with the category of

"PERSON" are extracted and added to a list of human characters. If the named entity was preceded by a title or honorific, the title or honorific is included in the list entry. The final list of human characters is fed to an instance of NLTK's FreqDist class to obtain the occurrence count of each character. In general, it seems that the most frequently mentioned character is the murder victim, and the second most frequently mentioned character is a detective/investigator in the crime.

Originally I looked into NLTK's named entity recognition capabilities. The format of those results was a complicated tree structure, and items were often inaccurately tagged or missed as entities. To avoid extra processing work and model retraining, I turned instead to spaCy, which allowed me to instead submit a string to an analyzer and access all the named entities (and their positions in the original string) from a list.

At this point, the spaCy NER code was working, but I noticed that it did not count titles before names, such as "Mr." or "Mrs.", as part of the entity. An initial solution was to add underscores to such names in our original text files, so that e.g. "Mrs. Inglethorp" would appear as "Mrs._Inglethorp." The entire token was then recognized as a single entity, but an unwanted side effect was that the same names without the title, "Inglethorp" in this case, were no longer recognized. To remedy all this, the next solution was to remove the underscores and to use the entity's starting index in the string to look for a preceding title. If one was found, it was manually appended to the entity before adding the entity to the list of human characters. This solution allowed spaCy to recognize all instances of a name like "Inglethorp" as it normally would, and the extra consideration for titles taken after the fact enabled all variations of the name to be added to the character list. (As it turned out, this practice was not helpful in the end, as the other modules that analyze the text perform well enough simply with single names for entities, which is what they do.)

Testing Data and Routine (Joe)

In order to evaluate the effectiveness of our program, I created 'answers-story,killer,victim.txt', which listed the true murderer and victim for each story in our dataset in a comma-separated

format, as laid out in the filename. This data was gathered from the plot summaries contained in each story's Wikipedia article.

I then edited and added to Bailey's 'trial.py' script to perform the actual testing. Our murder/victim prediction routine was run on each file in our 'data/' directory, and its results were compared to the correct answers extracted from 'answers-story,killer,victim.txt'. The characters under consideration for the roles of murderer/victim are displayed for each story, as are the routine's final selections and the true answers, so that the human tester can make a visual comparison. The accuracies for both killer and victim prediction are then calculated and displayed at the end, as well as the accuracies that result from randomly selecting the murderer and victim from each story's list of characters under consideration. We have found that our prediction routine usually produces better results than the process of random character selection.

Data Sanitizing, Verbnet and Filtering Sentence Results (Bailey)

The first thing I did was sanitizing files that we were going to use for our project. This involved figuring out which of the Sherlock Holmes stories were actually murder mysteries and which ones weren't, as well as cleaning up the .txt files, because the ones that we got directly off of Project Gutenburg had characters that Python was unable to parse.

Once that was done, I took the methods that Vicky and Joe had created (preprocessing and ner, respectively) and combined the two to filter the data down a little. This involved turning the list of things tagged "Person" into a list of all of the people mentioned in the sentence. To do this, I turned each tagged object into a string, put it into a new list, and then turned the list into a set and back into a list. I then tokenized the sentences obtained with Vicky's method and checked to see if they contained any of the proper nouns in the finalized character list. If they did, I kept them, and if not, I discarded them. (This is the "clues.py" file.)

After that, I started to play around with VerbNet to see if I could get it working. I found out that there was a package of it included with nltk, which made things easier. I was able to get it working when given a lemma of a word - it printed the members, subclass, and theme roles of the lemma. There were several other methods that I couldn't figure out how to work, but by that

time I had come to see that this didn't have the functionality needed for our purposes, so I didn't pursue this any further. (This is the "verbs.py" file.)

I created a basic "trial.py" to test the accuracy of our results, but Joe took that over pretty early in development. I also created "alternate.py", which contained some alternate of finding an answer, to see how they did compared to our semantic role analysis - we didn't end up using these, as we found that randomly selecting a character was sufficient.

I also did the general write-up for this paper, although everyone wrote their individual sections.

Semantic role associations (Mike)

The thematic roles analysis takes in the named entities, the target "murdery" words, and a set of context where those words were used. The goal is to identify from as few sentences as possible, the killer and victim of the murdery words. SRLs, Semantic Role labellers do exist as packaged functions, (Senna as an example) but for our purposes, without installing a machine learned library or dataset, they weren't going to be accessible. Analysis using Verbnet and Propbank, packages in nltk, was useful, but only to the extent that they confirmed our intuition that our violent verbs tend to be preceded by an Agent, the killer, and followed by a Theme, the victim - verbnet does not offer labelling functionality, only descriptions of the theta roles associated with verbs. So, we needed to build a way to guess at the murderers and victims.

An obvious starting point for victims is taking a leaf out of the sentiment analysis understanding; our murdery words are biased to negativity, so I implemented ways to decide in context who is the most commonly associated with our violent words. A further task was to break up the sentences, let's take the first named entity before a murdery verb and the first named entity after a murdery verb, average out over all murdery sentence contexts and sometimes we're right. But the process then became paring down the over generalizations - not to mention, attempting to work with pronouns as anaphora. The struggle is that it's inconsistent from one sentence to another, one story to another, and one author to another. A story might have exactly one line where the killer is revealed and if we miss that, our generalization is wrong. The name most commonly associated with murder-verbs could be the main character or detective solving

murders. The thing I came to recognize the most was how powerful a machine learned or classifying algorithm would be for this task - linguistic intuition alone is not necessarily enough to parse the infinite variations of sentence structure.