CNIT 519 - Project 2

Group 1: Melissa Kronenberger, Vinicius Lima, Aidan Sommers, Adrienne Zhang

Professor: Julia Rayz

Heterographic Pun Identification

1. Introduction

1.1. Project Objectives

In this project, our objective is to introduce a new dimension to our text analysis. Unlike our previous endeavor, where we focused on identifying words or phrases with multiple meanings that could be applied to different parts of the text, our current challenge revolves around words or phrases that can be pronounced differently. The aim is to uncover variations in pronunciation that can align with distinct segments of the text, potentially introducing humor through phonetic wordplay.

For this project, our primary task is to input a sentence into our system. The system's role is to identify a word or phrase within that sentence that has the potential to serve as the source of a joke. This source word or phrase will be manipulated to find alternatives that sound similar and create a pun target. Our mission is to identify phrases that exhibit phonetic similarities to the source, and these candidates are known as targets. It's important to note that we may have multiple options to explore. Ideally, the differences in pronunciation between the source and target should be subtle. Moreover, the source's meaning should align with one part of the text, while the target's meaning should correspond to another segment of the text. Success in finding these interpretations will signify the presence of humor in our text.

To illustrate with an example, consider the pun sentence: "Why won't the melon get married? Because they cantaloupe!" In this instance, the heart of the pun resides in the word "cantaloupe," which we technically define as the source word. The humor in this joke stems from the phonetic similarity between "cantaloupe" and the term "can't elope," both of which serve as our target words. The original text, in its pure form, may not make complete sense within its context, aside from the intrinsic humor. In other words, "cantaloupe" doesn't neatly integrate with the text's context in a conventional sense, although it is related to the word "melon." However, if we were to tweak the sentence to read "Why won't the melon get married? Because they can't elope!", the target words infuse a new layer of meaning into the text, now referring to the concept of "marriage." On the other hand, if we have "Why won't the melon get married?

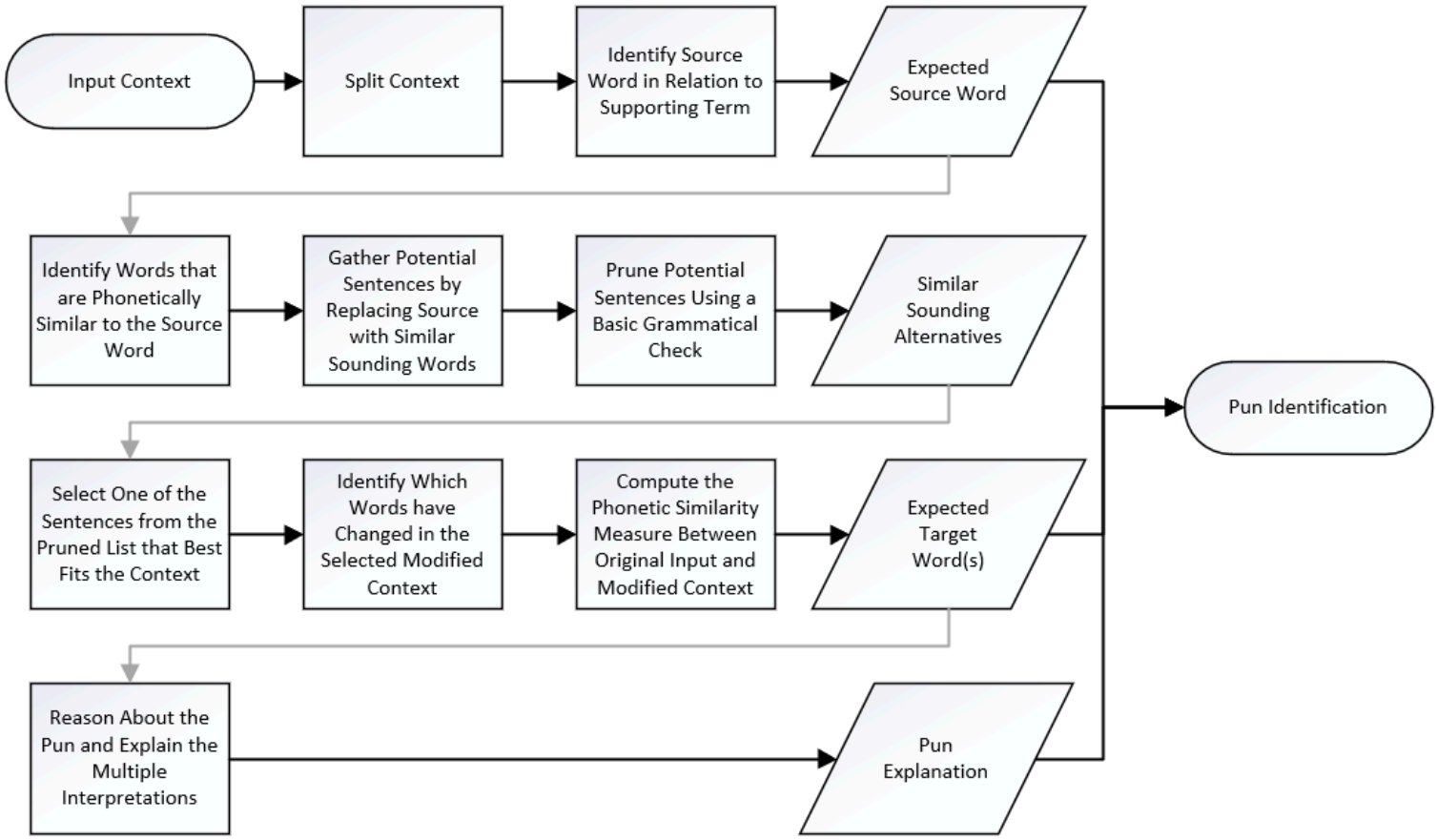
Because they **can't elope**!", the target words bring a new context to the text, referring now to the word "married".

In essence, our overarching objective is to pinpoint the pun word or words (referred to as the source word), pinpoint the corresponding target word or words that exhibit phonetic similarity to the source, and subsequently delve into the nuanced meanings derived from both the source and target words.

1.2. Program Architecture

Figure 1

Program Architecture Flowchart



Note. The general architecture for the program was based on an adaptation of the methodology for understanding puns described in “Identification of Homographic Pun Location for Pun Understanding” by Yu-Hsiang Huang, Hen-Hsen Huang, and Hsin-Hsi Chen (Huang et al., 2017). The adaptation in Figure 1 above is focused on heterographic puns and the

implementation of current technologies. Further explanation regarding the technologies used in the implementation of this architecture can be seen in section 2 below.

2. Methods

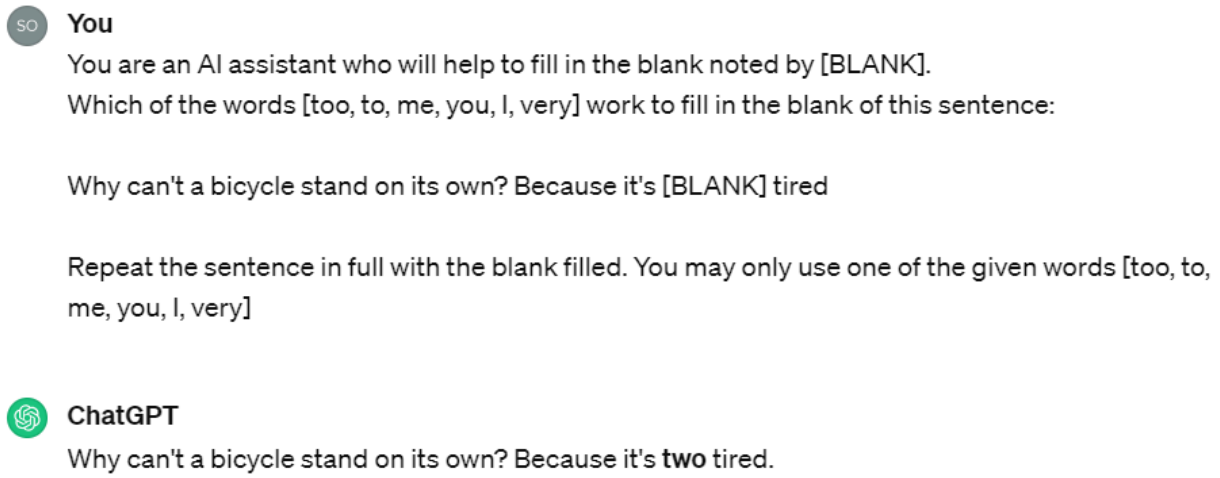
2.1. Large Language Model Usage

Throughout the implementation of the program architecture a large language model (LLM), OpenAssistant LLaMA 2 SFT v10 (70B), is used to perform the majority of tasks. This model was chosen as it is open source, has been tuned for using a system message, and was effective in our similar implementation for analyzing homographic puns. Further details regarding the strengths and weaknesses of this model in relation to this implementation have been explained in our first project report.

Building on feedback from our first project report, the methods outlined below have been intentionally guided with the goal of explicitly avoiding cases in which the LLM has memorized a given pun. Up until the very last step, we take special care to abstract out tasks into separate components, not directly related to humor, as to attempt to mitigate reliance on puns in the training dataset. In fact, when attempting to perform certain tasks using LLMs, the memorization of puns can be unhelpful for finding the alternative, less seen, structure. This is especially the case for heterographic puns, which are typically written to be recognizable, with the alternative wording being less common. An example of this can be seen in Figure 2 below.

Figure 2

Heterographic Pun Memorization with ChatGPT



Note. Critically, ChatGPT here is unhelpful as it has so closely memorized this pun that it ignores the given instructions. Despite being explicitly told that it may only use one of the given words from the list, it instead outputs the more commonly seen written form of this pun.

2.1.1 Splitting the Context

The LLM is prompted to split the given context into two parts based on the different subjects, multiple sentences, and conjunction words. The LLM's strength in this task comes from its flexibility to handle many different contexts without being given specific direction as to how to handle every scenario.

2.1.2 Identify Source Word in Relation to Supporting Term

The LLM is prompted to find which word in the first part of the context is most closely related to a word in the second part of the context. The underlying reasoning for this task can be obscure and nuanced, so the LLM is given examples to assist in explaining the task. The idea behind this method is that puns typically contain a supporting term from an alternative part of the given context. Puns often play on the linkage of the pun word to this supporting term in order to create their humor. By identifying which words are most closely related to each other from the different contexts, we look to find this underlying setup for the humor.

2.1.3 Grammatical Check

After phonetically similar words are identified, as explained in section 2.2 below, we prune the potential sentences using a basic grammatical check. For our usage demonstrated in class and in the testing of our architecture, we use the language-tool-python package. This is a wrapper for LanguageTool, which is an open-source grammar tool developed by Daniel Naber and Marcin Miłkowski (Naber & Miłkowski, 2005). It does not explicitly check for grammatical correctness, instead identifying typical errors based on general grammatical rules. For example, simple rules such as not repeating an article three times in a row. This is important to note, as it is still simple to create ungrammatical sentences that LanguageTool will not flag as incorrect. Such usage was still sufficient for our project, as our goal for this step is simply to prune nonsensical sentences. We also did develop an LLM implementation to detect grammatical errors, though this

was incredibly computationally expensive. As such, using LanguageTool proved much faster and more reliable for this step.

2.1.4 Select the Sentence that Best Fits the Given Context

From the pruned list of potential sentences, the LLM is prompted to select which sentence is grammatically correct and makes the most contextual sense. This results in a single selection containing a target word(s) for the alternative interpretation of the pun. Similarly to the case of identifying the source word and supporting term, this task can be somewhat nuanced. As such, the LLM is given examples of how to make this determination.

2.1.5 Reason About the Pun and Explain the Multiple Interpretations

Before the final explanation task for the LLM, the phonetic similarity measure is computed again. This is to detect any issues in which the final identified target word is not phonetically similar to the identified source word. The threshold we use to identify this is if our phonetic similarity measure for these words/phrases is less than 0.284. This threshold was identified as it is the resultant 1st percentile when computing our phonetic similarity measure on the heterographic puns contained in the SemEval 2017 Task 7 dataset. As such, 0.284 gives us a theoretical representation of how similar words must sound, according to our measure, in order to be considered heterographic puns. This method for identifying a threshold is very limited, though is intended to approximate a reasonable lower bound and in practice was not needed for the source and target pairs that reached this step throughout testing. This threshold serves as a final fail-safe in case the LLM has not properly selected from the provided list of similar-sounding targets.

The last task in our architecture is to explain the multiple interpretations relating to the source and target for a pun and to make a final determination as to if this satisfies the criteria for a pun. The given criteria is that if the source is compatible with one part of the text while the meaning of the target is compatible with another (different) part of the text, the input is a pun. Many prompting methodologies are used in this step to guide the model in generating a useful and thorough output. Full examples of both input and output can be seen in the Google Colab file, linked to in the appendix.

2.2. Determining Similarity

We opted not to employ Large Language Models (LLMs) for assessing sound similarity. LLMs are not typically trained on this task from a phonological perspective and are prone to hallucinations. Their ability to associate similar-sounding words is likely limited to what associations are made in the training data. Instead, we relied on more traditional methods of approaching this task. We employed a pronunciation dictionary to render words as a string of phonemes and then made calculations of similarity based on the string edit distance.

The dictionary used for this project was the CMU Pronunciation Dictionary, or CMUDict. CMUDict relies on the ARPHAbet phonetic transcription system and utilizes the two-character variant of the system (Kevin Lenzo, n.d.). The ARPHAbet is a set of phonetic transcription codes, similar to the IPA, for representing American English using only ASCII symbols, for use in computer applications (Daniel Jurafsky & James H. Martin, 2019). The ARPHAbet is not the only ASCII encoding of the IPA, with others including SAMPA, Wordbet, Usenet IPA, and Praat. However, the availability of a pronunciation dictionary in ARPHAbet made it more suited for our purposes. An example of a word run through the CMUDict is 'cheese,' which is phonetically rendered in ARPHAbet as 'CH IY Z.'

In calculating phonological similarity, we first calculated edit distances. An edit distance specifies the amount of 'operations' that must be performed on a string, in this case a string of phonemes, in order to transform it into a second, different string. These operations may be insertions, deletions, substitutions, or transpositions. There are a variety of string metrics for quantifying edit distance, and not all metrics make use of all four types of operations. The Jaro distance only allows for transposition, but heterographic puns are not restricted to only words that contain the same characters, so this metric is unsuitable. The Hamming distance can only be used to transform between strings of identical lengths. Heterographic puns do not necessarily obey this restriction, making the Hamming distance unsuitable. We developed our approach to edit distances in two stages, a minimum edit distance stage, and a weighted stage.

When we began brainstorming for the project, we hoped to take each query, mask each word in the query, and ask a language model to predict what word should go in the blank. We then hoped to test the original word against each of these replacement words and to assess both phonological similarity and similarity of word meaning. A word with high phonological

similarity but low semantic similarity would, we hypothesized, make a good candidate for the pun word.

This approach was stymied when we began to struggle with generating usable lists of replacement words. As the issues with word generation were being refined, we began tackling the issue of sound similarity from the opposite approach: Finding all the words in the dictionary with similar pronunciations. Because this required us to compare each word in the query to over a hundred thousand dictionary entries, speed was of the essence. Furthermore, heterographic pun words do not necessarily have a one-to-one relationship with the 'hidden' or 'target' meaning of the pun. The phonemes of two or more pun words may combine into a target word; conversely, the phonemes of a singular pun word may be divided into two target words. The task of searching the dictionary for not just each word in the query, but for each possible division of phonemes, exponentially increases the number of strings that need to be compared to the entire dictionary. Optimization of this process became vital.

Once our program could trawl through the dictionary and identify all potential target words for each subdivision of words in the query, we then calculated a similarity score between the source word and all existing words in the CMU dictionary. We then gathered all possible matches that exceeded a specific threshold of phonetic similarity. Ideally, the target word or words would be among this set of candidates. We then reintroduced the target words into the original pun sentence, substituting the source words, to assess whether the resulting sentence remained coherent. Ideally, the target word would produce the most coherent sentence, and an LLM might be able to select the most coherent sentence from among these options.

2.3. Minimum Edit Distance

When we first began experimenting with calculating similarity, we adopted a minimalistic calculation that only considered insertions and deletions, similar to the Least Common String approach. This was not meant to be our ultimate solution, but rather to produce halfway decent results for testing purposes while a better algorithm was being implemented. We defined similarity between two words as the ratio of shared phonemes between strings to the union of all phonemes in both strings.

$$Similarity = |[SourceWord] \cap [DictWord]| \div |[SourceWord] \cup [DictWord]|$$

where SourceWord is a set of all the phonemes in the original word
and DictWord is a set of all the phonemes in the test, candidate, or target word

Take, for example, the target word in the pun, "Why won't the melon get married? Because they cantaloupe!" The pun word is "cantaloupe" and the target words are "can't elope." The phonetics for these words are "K AE1 N T **AH0** L **OW2** P" and "K AE1 N T **IH0** L **OW1** P." The shared phonetics between these strings are 6, and the union of these phonetics totals 10. The result of dividing 6 by 10 is a similarity score of merely 0.6. As these are a pun and target words in one of our instructor-supplied puns, we had hoped for a higher similarity score.

After reevaluating the phonetics, we realized the CMUDict was adding numerical annotations to syllables to indicate stress, and realized that we should be evaluating the difference between "K AE N T **AH** L OW P" and "K AE N T **IH** L OW P" instead, which gives us a similarity score of 7 divided by 9, or 0.77. This value still seemed very low. Given that the maximum possible number of shared phonemes was equal to the length of the longest string, we realized we could change our ratio to divide not by the magnitude of the union of the phonemes, but rather by the length of the longest phoneme string. This gave us a similarity score of 7 divided by 8, or 0.875.

$$Similarity = |[SourceWord] \cap [DictWord]| \div max(|[SourceWord]|, |[DictWord]|)$$

where $[SourceWord]$ and $[DictWord]$ are truncated to a two-character symbol.

Our next step was to deconstruct words, to separate them into parts, to make it possible to detect puns in situations where the pun word(s) and target word(s) did not have a one-to-one relationship. Given that "cantelope" is the pun word, and there are *two* target words, "can't" and "elope," it becomes necessary for us to compare only part of "cantaloupe" with different words in the dictionary. We shifted our focus towards breaking down any string of phonemes into two part, so that we could compare "-aloupe" with "elope," or "**AH** L OW P" and "**IH** L OW P" When comparing just these two inputs, the similarity score is reduced to 0.75. Even though there has been no change in the phonemes themselves, there are fewer matching phonemes in the bisected words, and a smaller overall length in the denominator of the final division of the calculation, resulting in a smaller overall similarity. This process, of normalizing the similarity

scores using the magnitude of the phonemes, comes with the drawback of computing lower similarity scores for smaller words.

For simplicity, and to prevent an exponential increase in computation time, we restricted our code to splitting the phonetics of any query into only two parts, limiting our ability to find more than two similar-sounding words. This was a limitation we ultimately accepted, given that puns involving three or more relationships appeared rare and were nonexistent in the SemEval 2017 Task 7 heterographic pun dataset (Miller et al., 2017), and in the puns provided for our analysis by the course instructor.

While this approach worked well enough early on in our design process, it came with a significant drawback: In order to catch even highly similar strings of phonemes, such as "-aloupe" and "elope," we needed to set low similarity thresholds, which ultimately result in generating an overwhelming number of potential candidates for pun words. This word list creates a bottleneck in our pipeline, particularly if we must ask an LLM to evaluate each resulting word combination for sentence coherence. To provide an illustration, when analyzing the word "cantaloupe," we can successfully extract "can't" and "elope" but at the cost of generating a staggering 2035 words as candidates. This sheer volume of candidates generated surpassed the token limit set by the LLM we employ.

Ultimately, our initial approach relied on a simplistic, minimum edit distance calculation that did not always correctly align two strings. It was limited in all the ways we would expect a minimum edit distance calculation to be limited. It was limited to additions and subtractions, and unable to ascertain that the vowels 'IH' and 'AH' might be fairly similar to one another, and that the cost of substituting one for the other should be less than the cost of deleting and adding consonants with wildly different features, such as "M" and "G." Although this approach served its purpose during development by allowing us to test and refine other aspects of our program, it eventually needed to be replaced by a method that was sensitive to sound-similarity between phonemes.

2.4. Weighted Edit Distance

One of the most common edit distance metrics for calculating a minimum edit distance is the Levenshtein edit distance, which can perform substitutions, deletions, and additions. The Levenshtein edit distance can be used to calculate a minimum edit distance in which every

operation has the same weight, similar to what we had already been doing, or it can be combined with a matrix of weights in order to allow phonemes with similar features, such as ‘B’ and ‘P’ to substituted more cheaply than very different phonemes, such as ‘AH’ and ‘G.’ Our goal was to utilize the weighted approach. There are a variety of available algorithms for calculating Levenshtein edit distance, including a simple recursive algorithm, Levenshtein automatons, and Hirschberg’s algorithm. We utilized the Wagner-Fischer algorithm because, while Hirschberg’s algorithm runs faster on longer strings, we are always comparing very short strings. The Levenshtein automatons approach was prohibitively complex and expensive to set up.

When we were searching for tools to incorporate into our workflow, the *strsimpy* Python package stood out for implementing multiple types of edit distance, for implementing Levenshtein edit distance via the Wagner-Fischer algorithm, and it allowed for weighting of the Levenshtein algorithm from two different directions (Luo, 2021). However, this package was designed for *string* edit distance and worked by adding, deleting, and substituting individual characters in a string. The ARPHAbet phonemes supplied by CMUDict were two-character transcriptions, and stored in an array. In order to translate between the ARPHAbet form supplied by CMUDict and a form the algorithm implementation could understand, we quickly created single-symbol representations of ARPHAbet phonemes. These were not important aside from how they allowed the strsimpy Python implementation to manipulate the phonemes and were not used for any other purpose.

Armed with the algorithms and tools necessary to calculate edit distance using weights, we then required a table or matrix representing the ‘cost’ of each substitution, insertion, and deletion. Many phonetic indexing systems, such as the Soundex system, diminish the role of or entirely eliminate vowels from their calculations, under the assumption that all vowels sound similar enough that they can be substituted for one another extremely cheaply. However, eliminating vowels has the potential consequence of removing information about syllables. Consider the words ‘iris’ and ‘rise’ Alone, without a vowel in front of ‘rise,’ these words most likely do not comprise a pun and target word. Yet if you eliminate their vowels, both are transcribed as “rs.” Based on our previous experimentation with the unweighted minimum edit distance, we knew we wanted to decrease the number of false positives submitted to the LLM for analysis, and it made sense to retain information about vowels.

Every time we set out to research pun detection, be it homographic or heterotrophic, we inevitably found ourselves looping back to the influential SemEval-2017 Task 7 (Miller et al., 2017). This task provided, as a resource, the cost matrices developed by Christian Hempelmann in his PhD thesis paper (Hempelmann, 2003). However, it needs to be noted that Christian Hempelmann worked for the SemEval-2017 Task 7 team, and there may have been some slight bias in selecting his work to use as a resource. Regardless, his matrices were developed back in 2003, and Hempelmann mentions they built heavily off the work of Sobkowiak et al (1991), potentially leaving them out of date.

There were other issues with using Hempelmann 's work. Although there are a number of standardized ASCI representations of IPA symbols, such as SAMPA, which Sobkowiak would later go on to cover (Wlodzimierz Sobkowiak, 1996), Hempelmann utilized the same ASCII system that Sobkowiak used in his 1991 book, which is non-standard. Sobkowiak's 1991 text was not available to us, and Hempelmann did not describe the system or its use. Ten of the vowels were represented by numbers, and others were represented by symbols such as the ampersand.

It ended up being simplest to reverse engineer the meaning of these symbols from Hempleman's Main Corpus of Target-Pun Pairs in Appendix E. These symbols did not closely correspond to CMUDict's ARPHABet phonetics, and had numerous additional vowels, primarily vowels that occurred directly before the consonant 'r.' There appears to be some debate in linguistics over whether or not 'r,' might in itself be a vowel, or whether vowels before 'r' should be considered different from vowels not before 'r,' or whether only 'er' is a vowel and the other combinations are not. On this issue, the ARPHABet and Sobkowiak's system simply did not agree.

In order to use Hempelmann's tables, we would have to toss out these additional vowels, as well as an unaspirated 't' consonant denoted by the question mark symbol, which would skew the remaining data, as the remaining rows and columns would no longer account for the likelihood of replacement in vowels that occur before the letter 'r.' Furthermore, despite Hempelmann's conjoined vowel-and-consonant table leaving the appearance that it might handle the few situations in which a vowel and consonant might be confused for one another, such as the consonant 'y' (which he denoted with 'j,' and which ARPHAbet denotes with 'Y') and the long 'i' vowel (which he denoted with 'I', and which ARPHAbet denotes with 'IY'), the matrix

has no relationships between any vowels or any consonants that are anything other than full cost, meaning that the vowel and consonant portions are effectively unrelated, and the cells where they intersect are padded-out filler.

Lastly, a pronunciation dictionary using Sobkowiak's system either does not exist or is not in use even within the papers that utilize it; Hempelmann's data often transcribed phonetics for identical words in wildly different ways. See "ammonia," which in Hempelmann's Main Corpus is transcribed as "0m5nj0" and as "1m5nI6" just lines away from one another on page 172, with no explanation given. Neither of these pronunciations is exactly wrong; they simply evidence a lack of standardization in accent. In addition, a review of Hempelmann's Main Corpus was a cause for concern, as the transcriptions of the words were often flipped, suggesting the database had not been carefully pruned for error.

While we would later go on to implement Hempelmann's tables, as an alternative cost matrix to our chosen cost matrix, we believe that there are considerable theoretical flaws with doing so. When we did implement Hempelmann's tables, we found that the costs for substitutions were very small in some instances, meaning that the similarity thresholds of 0.70 and 0.80 which we had grown accustomed to using for other systems and other tables were no longer valid, and that they needed to be increased in some cases to 0.98 to start producing reasonably small numbers of similar word matches.

After inspecting a more recent paper that revisited the SemEval-2017 Task 7 and focused exclusively on heterographic puns (Diao et al., 2019), our attention was drawn to matrices developed a year before the task, in a paper titled *Phonological Pun-derstanding* (Jaech et al., 2016). This paper developed separate consonant and vowel matrices, and in fact referenced Hempelmann's work. However, Jaech et al. build their cost matrices using the ARPHAbet, rendering it immediately more compatible for our intended use. In fact, Jaech et al. also does not include the same exact vowels which the CMUDict does not include from the official ARPHAbet, namely AX and UX, leaving us with no data we needed to excise or throw away.

Unlike Hemplemann's cost tables, Jaech et al. were looking at edit-probabilities; thus, every item on their matrix is a percentage, representing the likelihood that any phoneme in the target word would have originated from a given phoneme in the pun word. Spaces left blank on the matrix represent percentages smaller than 1%. Including all of these missing sub-1% percentage points, each column should sum to 100%. In order to transform these

edit-probabilities into edit-costs, we normalized each percentage by dividing it by the likelihood the phoneme would replace itself (ie, that it would be unchanged from the pun word to the target word), and then subtracted that value from one. This would give us a cost value bounded between 0 and 1, with 0 meaning that phonemes were interchangeable at no cost, and 1 meaning that phonemes were completely different and ought to be interchangeable only at full cost.

This approach to normalization ended up briefly backfiring when it turned out that two rare phonemes, the 'th' phonemes represented in ARPHAbet by 'DH' and 'TH,' were more likely to end up in a target word by replacing a more common phoneme, namely, 'D' or 'T,' than they were by remaining unchanged. From the standpoint of a native English speaker, this does make some sense; a common way of mispronouncing 'the' is with a 'd' instead of a 'th,' and mispronunciation is a key feature of heterographic puns. It does not seem strange that a large number of puns should rely on splitting off a 'da' or 'de' or even 'di' at the beginning of a word into the article 'the,' and it therefore seems plausible that, since the 'th' (ARPHABet's 'DH') sound is so uncommon in English, and the word 'the' is conversely *very* common in English, almost every single pun involving a 'DH' phoneme will end up transforming it. The same issue persisted with the other phoneme associated with 'th' (ARPHABET's 'TH') but was not as pronounced.

In fact, the likelihood a 'DH' phoneme in the target word was paired with a 'DH' phoneme in the pun word was only 6%, and the resulting division led to catastrophically out-of-bounds cost values that temporarily had our algorithm massively overinflating the similarity of words including 'DH' and 'TH' phonemes to literally every other word. Ultimately we ended up changing our normalization algorithm to divide by the highest value in a column, rather than the likelihood a phoneme replaced itself.

To compute similarity scores between two words, or two strings of phonemes, we ran our weighted Levenshtein algorithm utilizing a Wagner-Fischer implementation and our Pun-derstanding cost matrices. However, the result of this was a *distance* measurement, a measurement of *difference* rather than of *similarity*. To produce a similarity metric, we had to first normalize it and then subtract it from one. To normalize it, we divide by the larger of the two strings' phoneme lengths. This did mean that longer words with a single edit had higher similarity scores than smaller words with a single edit, which for some time worked against us, especially with very small, three-letter words, such as 'fur' and 'for.'

Yet the problem with ‘fur’ and ‘for’ was deeper than we initially understood. The CMUDict transcribes the phonemes for ‘fur’ using the vowel ‘ER,’ which led to a transcription of ‘F ER.’ Conversely, ‘for’ was transcribed using the consonant ‘R’ and the vowel ‘AO,’ yielding ‘F AO R.’ Our Levenshtein algorithm would compare AO and ER, as these were the only vowels and all substitutions between consonants and vowels are full cost, and find that they were almost nothing alike, leading to a very expensive substitution, and would then conclude that ‘R’ had no mate, leading to a costly insertion or deletion. The result was that ‘for’ and ‘fur’ had a similarity score just barely above one-third, or 0.33, as they shared only a single phoneme. This brought us back to the issue of ‘r,’ and its close relationship with vowels. To patch this, we decided to carve out special case exceptions for ARPHAbet consonant phonemes ‘R’ and for ‘Y,’ allowing them to match up with the vowels ‘ER’ and ‘IY’ at no cost. When used with our for/fur example, this mechanism still mandated a full-cost insertion/deletion of the ‘AO’ phoneme, but dramatically increased similarity to over 0.66. While we would like to fine-tune what these substitutions should actually be through testing (as opposed to just making them zero-cost), or perhaps treat situations in which vowel-’R’ couplets are processed uniquely, these optimizations ended up falling out of scope for our needs for the current project.

At this time, we also entertained the possibility that vowel substitutions ought to be weighted more mildly compared to consonant substitutions, based on our experiences reading about phonetic indexing; we argued that ‘far’ and ‘for’ ought to be seen as closer together than ‘fat’ and ‘fall.’ However, by this point, we were already achieving much better performance in generating similar words, and had not yet located a paper that could describe to us an exact percentage that could accurately weigh the importance of vowels when compared to consonants, and so we left further calibrations to future exercise.

2.5. Optimizations

In our pursuit of efficiency, the incorporation of cost-tables marked a significant shift. We transitioned from generating a multitude of similar words, as seen in our earlier methods, to a more refined output—typically less than a hundred words, especially in scenarios where achieving a higher similarity score sufficed, thereby minimizing the number of generated words. However, challenges emerged in less favorable situations, such as when the similarity score equaled or exceeded 0.75. In these cases, relying on the computational resources available,

particularly on Google Colab, led to system crashes. This was primarily due to the increased computational load required for additional calculations and phonetic comparisons within the extensive CMU dictionary.

Recalling that the target word should phonetically resemble the source word, a logical strategy emerged. It made sense to limit the comparison of the source word to those words in the CMU dictionary that were proximate in length. Our approach involved searching and evaluating the similarity score exclusively with words in the dictionary that shared the same length or differed by only one phoneme length. This strategic adjustment significantly narrowed the scope of our search, effectively resolving the crash issues.

In the end, although our refined system demonstrated a slower processing speed compared to previous methods, it proved to be substantially more precise. This precision resulted in a reduced pool of potential target candidate words, streamlining the input for subsequent steps in our analysis.

Figure 3 illustrates our performance for this step when testing with the word "cantaloupe". Observe that the system iteratively breaks down the phonetics into two sets - until it achieves the whole word - and searches for similar-sounding words that are close in length. We successfully extracted the target words "can't" and "elope" (in bold) within a pool of just 92 words.

Figure 3: Similar words generated for all possible divisions of 'cantaloupe' utilizing the weighted cost-table implementation, after optimization.

```
      Phonetics1: ['K'] --> Similar Sounds 1: []
      Phonetics2: ['AE1', 'N', 'T', 'AH0', 'L', 'OW2', 'P'] --> Similar
Sounds 2: ['antelope', 'cantaloupe']
      ---------------------
      Phonetics1: ['K', 'AE1'] --> Similar Sounds 1: []
      Phonetics2: ['N', 'T', 'AH0', 'L', 'OW2', 'P'] --> Similar Sounds 2:
['antelope']
      ---------------------
      Phonetics1: ['K', 'AE1', 'N'] --> Similar Sounds 1: ['caen', 'cahn',
'can', 'cann', 'kan', 'kang', 'kann', 'kanne', 'krahn']
      Phonetics2: ['T', 'AH0', 'L', 'OW2', 'P'] --> Similar Sounds 2:
['tullos']
```

```
--------------------
Phonetics1: ['K', 'AE1', 'N', 'T'] --> Similar Sounds 1: ['cannot',
"can't", 'cant', 'canter', 'cantey', 'canto', 'cantor', 'cantu', 'canty',
'kalthoff', 'kandt', 'kant', 'kanter', 'kantor', 'kantz', 'kath', 'quant',
'scant']
Phonetics2: ['AH0', 'L', 'OW2', 'P'] --> Similar Sounds 2: ['elope']
--------------------
Phonetics1: ['K', 'AE1', 'N', 'T', 'AH0'] --> Similar Sounds 1:
['banta', 'caltha', 'cana', "can't", 'cant', 'canter', 'cantey', 'cantle',
'canto', 'canton', 'cantor', 'cantu', 'canty', 'casta', 'hanta', 'kana',
'kanda', 'kandt', 'kanka', 'kant', 'kanter', 'kantle', 'kantor', 'katha',
'khanna', 'manta', 'qantas', 'ranta', 'santa', "santa's", 'sante']
Phonetics2: ['L', 'OW2', 'P'] --> Similar Sounds 2: ['lope']
--------------------
Phonetics1: ['K', 'AE1', 'N', 'T', 'AH0', 'L'] --> Similar Sounds 1:
['antal', 'antle', 'bantle', 'cancel', 'candle', 'cannell', 'cantel',
'cantle', 'canton', 'cantrall', 'cathell', 'cattle', 'chantal', 'fantle',
'kandel', 'kantle', 'kastl', 'mantel', 'mantle', 'pantle', 'qantas']
Phonetics2: ['OW2', 'P'] --> Similar Sounds 2: ['hope']
--------------------
Phonetics1: ['K', 'AE1', 'N', 'T', 'AH0', 'L', 'OW2'] --> Similar
Sounds 1: ['cantaloupe', 'cantle', 'kantle']
Phonetics2: ['P'] --> Similar Sounds 2: []
--------------------
Phonetics1: ['K', 'AE1', 'N', 'T', 'AH0', 'L', 'OW2', 'P'] -->
Similar Sounds 1: ['antelope', 'cantaloupe', 'cantaloupes']
Phonetics2: [] --> Similar Sounds 2: []
--------------------
Total # words: 92
```

3. Results and Discussion

Due to the stochastic nature of LLMs, the complexity of interweaving these several methods, and the limited time constraints for this project, the dataset used to explicitly evaluate our full architecture was relatively small. Using heterographic puns from the SemEval Task 7

dataset, thirty interesting examples were chosen based on our team's preferences for humor and providing a reasonable challenge to the system. We acknowledge that this is a limitation that may have induced bias into the results, though we assessed that this is a worthwhile tradeoff to appropriately assess the subjective nature of humor. Doing so also made the manual labeling of these results more entertaining and subsequently less tedious.

After running the 30 example puns through the system, 26 were fully successful and 4 failed. Ten examples that are not puns were also evaluated. The number evaluated was limited as the system was performing very well at classifying these as not puns, correctly labeling all ten as not puns. Of the four failures, 3 were due to not finding the correct target word in the sound similarity task and 1 was due to not identifying the appropriate source word. Amongst the failed attempts relating to sound similarity, "bovine" and "divine" have a computed sound similarity measure of 0.609, "cue" and "clue" have a computed sound similarity measure of 0.667, and "cotton" and "caught on" have a computed sound similarity measure of 0.833. The first two here illustrate that by simply reducing our aforementioned threshold, we may have been able to arrive at these results. However, as we reduce this threshold, our computation time increases drastically and we risk overloading the LLMs context length. For "cotton" to "caught on," we were able to get outputs of "caught 'n" and "caught un," but the second word "on" ('AA1', 'N') continually hid from our system. Notably, 'AH0', 'N' for the "on" in cotton, and 'AA1', 'N' for the word "on" have a similarity measure of 0.583. Cases such as this illustrate that more work could be done to iteratively improve our similarity measure if given more time. Ideas regarding such work are briefly explored in section 4.

4. Conclusions and Future Work

Overall, this architecture contained many parts and was analyzing puns while explicitly attempting to avoid the mention of humor. This proved to be a very difficult challenge, though our system was able to perform well while successfully mitigating concerns of memorization. Especially once putting together all of the different parts, our system is able to give a thorough and detailed explanation as to why it has determined that something is a pun. With this, we look forward to building on top of this work.

4.1 Sound Similarity Improvements

As mentioned previously, our algorithm encountered difficulties when consonants sounded similar to vowels or vice versa. As phonetic indexing and transcription algorithms frequently seem to overlook vowels as unimportant, perhaps this issue has been masked in the papers we relied upon to support our work; however, it seems to be common for researchers building cost-matrices to consider vowels as entirely separate from consonants, and not to build cost relationships between them. We believe that this is unfortunate when it comes to vowel-consonant pairs like ARPHAbet's 'ER' and 'R' as well as 'IY' and 'Y.' We would like to look further into this issue to see if anyone has built a cost table roughly estimating the relationships between these sounds.

Furthermore, because 'R' sometimes takes the shape of a vowel (ie 'ER,') there are some difficulties when calculating the similarity between words like 'far,' which a distinct vowel before an 'r,' and words where the vowel and the 'r' have merged into a singular sound (ie 'ER'). In a Levenshtein algorithm, there is no mechanism for facilitating one-to-two or two-to-one substitutions; it isn't possible to indicate that vowels, when followed by 'R,' can merge together into a unit that can be cheaply replaced with 'ER.' Furthermore, if we delete 'ER' from our tables and try to separate every 'ER' we receive from CMUDict into 'EH' and 'R,' we are deleting valuable data and potentially skewing what remains. Yet without splitting them, our algorithm will always be left to delete or insert one phoneme, which is always expensive. We know that we can soften the problem by weighting vowels lighter than we weight consonants, which we can potentially experiment with, but it would also be interesting to further investigate the issue and to see if other constants are also affected by a mushy relationship with vowels. For example, 'W' with its relationship to 'UW,' or 'L' with its ability to consume vowels that come before it (such as in 'battle,' which pronounced formally is 'B AE T AH L' but which is often colloquially pronounced 'B AE T - L,' such that 'L' becomes a vowel of its own, not unlike 'ER.').

4.1.1 Letter Removal for Leftover Words

One method in which we could improve our ability to produce similar-sounding target words is by identifying which part of a word a particular sound comes from, fully removing those letters, and keeping the resulting letters as a potential word. Currently, we are able to do

similar splits, though we evaluate for potential words using the phonetic similarity threshold. We could add to the potential words found by also including the letters that are left over from this split as potential words. For example, in the case of "cotton" above, we were able to get "caught" from "cott" and would have been left with "on." If simply taking "on" as a potential word, we would have found the correct target in "caught on." This method may be particularly advantageous for such examples in which we are going from a source word to a target phrase made up of multiple words.

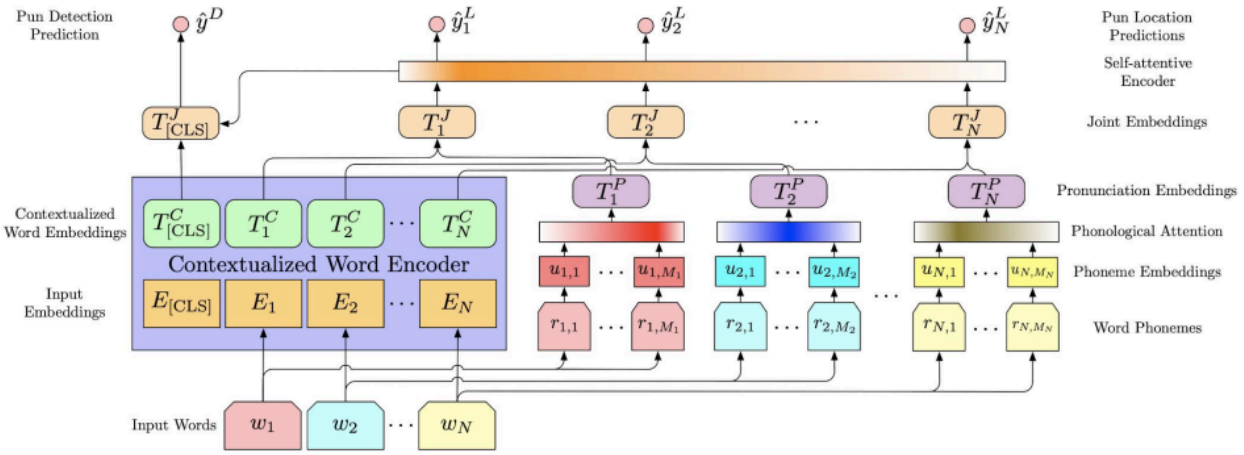
4.2 LLM Fine-tuning with Full Sound Similarity Search Capability

As large language models become more advanced and less expensive to utilize, their integration with larger iterative systems is incredibly interesting. It is possible that an LLM could be tuned and prompted to iteratively use external tools while in pursuit of an answer. Thus, it may be possible to give an LLM access to our phonetic similarity measure and allow it to comb through numerous words, unsupervised, in pursuit of an answer. Currently, this would be incredibly expensive, difficult to test, and may not give significantly better results. Still, as this technology matures, such opportunities are very exciting.

4.3. Pronunciation-attentive Contextualized Pun Recognition

An alternate approach we explored was a novel idea called 'Pronunciation-attentive Contextualized Pun Recognition,' from the 2020 paper entitled, '"The Boating Store Had Its Best Sail Ever": Pronunciation-attentive Contextualized Pun Recognition," which took a unique approach to homographic and heterographic pun detection and location. For the scope of our project, we decided to focus on exploring their approach to heterographic puns. The overall idea was to detect and locate puns utilizing contextualized word embeddings and pronunciation embeddings rather than calculating similarity scores from the pronunciation of detected pun words, as had been the strategy of previous algorithms. The figure below is a visual depiction of the framework used in the paper:

Figure 4: Visual depiction of attention algorithm, (Diao et al., 2019).



To explain this figure and the methodologies represented, the algorithm takes an input vector of words, which is the pun statement. Then, two subsequent vectors are derived from the input, one being the contextualized word embedding, and the other being the pronunciation embedding. To derive the contextual word embeddings, we pass the input vector into BERT's multi-layer bidirectional word encoder (the pre-trained model we used was BERT-Base, Cased), which derives the semantics of each word as well as the semantics of the overall input. To derive the pronunciation embeddings, we break the input vector's words into their respective phonemes, and each phoneme is projected to an embedding space as a trainable vector representing its phonological properties. Subsequently, the attention mechanism is applied to identify important phonemes (which estimates the importance vector of each word) and derive the pronunciation embedding, which is a combination of the importance vector and the previously created vector representing the phonological properties for each phoneme, aggregated to the word each phoneme was derived from. The contextual word embeddings and pronunciation embeddings of each word are then concatenated to create a joint embedding that models each word's semantics and phonological properties. From the joint embeddings, the self-attention mechanism is applied, which estimates the importance of each word in the context of the input statement. From the self-attentive embedding and the overall semantic representation, we can derive the pronunciation-attentive contextualized representation, which is used in pun detection through the training of a Linear Classifier for binary classification (which utilizes a softmax function on a fully connected neural network layer, then deriving the logits), optimized using cross-entropy loss. The self-attentive joint embedding is also used for pun location through binary classification and cross-entropy loss optimization as used for pun detection.

We attempted to compare this method with our own, but we ran into a variety of technical issues and further work was deemed too costly. The paper claims that each of the precision. recall, and F1-scores are higher compared to other benchmark methods, namely rule-based machine learning classifiers (Duluth, JU_CSE_NLP, PunFields, UWAV, Fermi, UWaterloo), recurrent neural networks (Sense), linguistic feature capturing (CRF), a combination of recurrent neural networks and linguistic feature capturing (Joint), and contextualized pun recognition without phoneme consideration (CPR), which suggests that considering phonemes gives this methodology an edge over others which utilize static word embeddings or external knowledge bases as a means of categorizing and locating puns. If given the opportunity to partner with other groups who are interested in this implementation, it may be insightful to see how this method performs.

References

Huang, Y.-H., Huang, H.-H., & Chen, H.-H. (2017, April). Identification of Homographic Pun Location for Pun Understanding. *WWW '17 Companion: Proceedings of the 26th International Conference on World Wide Web Companion*, 797–798. 10.1145/3041021.3054257

Naber, D., & Miłkowski, M. (2005, August 15). *languagetool: Style and grammar checker for 25+ languages*. GitHub. https://github.com/languagetool-org/languagetool

Zhou, Y., Jiang, J.-Y., Zhao, J., Chang, K.-W., & Wang, W. (2020). "The boating store had its best sail ever": Pronunciation-attentive contextualized pun recognition. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.acl-main.75

Daniel Jurafsky & James H. Martin. (2019). Chapter 27—Phonetics. In *Speech and Language PRocessing*. https://web.stanford.edu/~jurafsky/slp3/old_oct19/27.pdf

Diao, Y., Lin, H., Yang, L., Fan, X., Wu, D., Zhang, D., & Xu, K. (2019). Heterographic Pun Recognition via Pronunciation and Spelling Understanding Gated Attention Network. *The World Wide Web Conference*, 363–371. https://doi.org/10.1145/3308558.3313505

Hempelmann, C. (2003). *Paronomasic Puns: Target Recoverability Towards Automatic Generation*. Purdue University.

Jaech, A., Koncel-Kedziorski, R., & Ostendorf, M. (2016). Phonological Pun-derstanding. *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 654–663. https://doi.org/10.18653/v1/N16-1079

Kevin Lenzo. (n.d.). *The CMU Pronouncing Dictionary*. The Carnegie Mellon University. Retrieved November 8, 2023, from http://www.speech.cs.cmu.edu/cgi-bin/cmudict

Luo, Z. (2021). *strsimpy: A library implementing different string similarity and distance measures* (0.2.1) [Python; OS Independent]. https://github.com/luozhouyang/python-string-similarity

Miller, T., Hempelmann, C., & Gurevych, I. (2017). SemEval-2017 Task 7: Detection and Interpretation of English Puns. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, 58–68. https://doi.org/10.18653/v1/S17-2005

Sobkowiak, W. (1991). *Metaphonology of English Paronomasic Puns*. P. Lang.

Wlodzimierz Sobkowiak,. (1996). Phonetic Transcription in Machine-Readable Dictionaries. *PART 1 - Computational Lexicology and Lexicography*, 181–188.

Appendix

Input context

```
this girl said she recognized me from the vegetarian club, but I've never met herbivore
```

Split context

```
First context: this girl said she recognized me from the vegetarian club,
Second context: but I've never met herbivore
```

Identify source in relation to supporting term

```
Using the given context, the base context word vegetarian is most related to the input context word herbivore.
```

Identify words that are phonetically similar to the source word

```
[nltk_data] Downloading package cmudict to /root/nltk_data...
[nltk_data]   Package cmudict is already up-to-date!
Phonetics1: ['HH'] --> Similar Sounds 1: []
Phonetics2: ['ER1', 'B', 'IH0', 'V', 'AO2', 'R'] --> Similar Sounds 2: ['herbivore']
--------------------
Phonetics1: ['HH', 'ER1'] --> Similar Sounds 1: ['her', 'hur']
Phonetics2: ['B', 'IH0', 'V', 'AO2', 'R'] --> Similar Sounds 2: ['before', 'devor', 'livor']
--------------------
Phonetics1: ['HH', 'ER1', 'B'] --> Similar Sounds 1: []
Phonetics2: ['IH0', 'V', 'AO2', 'R'] --> Similar Sounds 2: ['devor', 'livor']
--------------------
Phonetics1: ['HH', 'ER1', 'B', 'IH0'] --> Similar Sounds 1: ['herbig', 'herbin']
Phonetics2: ['V', 'AO2', 'R'] --> Similar Sounds 2: ['vore', 'war', 'warr', 'woehr', 'wor', 'wore']
--------------------
Phonetics1: ['HH', 'ER1', 'B', 'IH0', 'V'] --> Similar Sounds 1: ['herbig', 'herbin']
Phonetics2: ['AO2', 'R'] --> Similar Sounds 2: ['hoar', 'hoare', 'hoerr', 'horr', 'oar', 'ohr', 'or', 'ore', 'orr', 'whore', 'yore', 'your']
--------------------
Phonetics1: ['HH', 'ER1', 'B', 'IH0', 'V', 'AO2'] --> Similar Sounds 1: ['herbivore']
Phonetics2: ['R'] --> Similar Sounds 2: ['er', 'eure', 'her', 'hur', 'ur', 'yer']
--------------------
Phonetics1: ['HH', 'ER1', 'B', 'IH0', 'V', 'AO2', 'R'] --> Similar Sounds 1: ['herbivore']
Phonetics2: [] --> Similar Sounds 2: []
--------------------
Total # words: 38
```

Gather potential sentences by replacing the source with the potential targets

A basic grammatical check is used to prune the potential targets

```
["This girl said she recognized me from the vegetarian club, but I've never met her before.",
 "This girl said she recognized me from the vegetarian club, but I've never met herbivore her.",
 "This girl said she recognized me from the vegetarian club, but I've never met herbivore yer."]
```

Select one of the sentences from the pruned list that best fits the context

```
This girl said she recognized me from the vegetarian club, but I've never met her before.
```

Identify what words have changed in the modied context

```
Enter the first string: girl said she recognized me from the vegetarian club, but I've never met herbivore
Enter the second string: girl said she recognized me from the vegetarian club, but I've never met her before
Differences in the first string: herbivore
Differences in the second string: her before
herbivore
her before
HH ER1 B IH0 V AO2 R
HH ER0 B IH0 F AO1 R
```

Compute the phonetic similarity measure between original input and modified context

```
0.8690476190476191
```

This score is used as a final check for if the source and predicted target sound similar
-> if they do not sound similar (similarity measure < 0.284) we will not consider this a heterographic pun

Reason about the pun and explain the multiple interpretations

Let's analyze the context step by step:

1. Both source and target contexts have the same sentence structure: 'This girl said she recognized me from the vegetarian club, but I've never met ___.'
2. The source word is 'herbivore,' which refers to an animal that eats only plants and is often associated with vegetarianism.
3. The target word is 'her before,' which is a part of the phrase 'I've never met her before,' indicating that the speaker has not previously encountered the girl.

Now let's compare the compatibility of the source and target words with the context:

1. The source word 'herbivore' is compatible with the first part of the sentence, 'This girl said she recognized me from the vegetarian club,' as it refers to a concept related to vegetarianism.
2. However, the source word 'herbivore' is not compatible with the second part of the sentence, 'I've never met ___,' as it does not fit grammatically or semantically with the rest of the sentence.
3. On the other hand, the target word 'her before' is compatible with both parts of the sentence. 'I've never met her before' is a common phrase that means the speaker has not previously encountered the girl, and it fits grammatically and semantically with the rest of the sentence.

Based on this analysis, the meaning of the source word 'herbivore' is compatible with one part of the text, while the meaning of the target word 'her before' is compatible with another part of the text. Therefore, this is a pun.