Netids: Amit Rajesh (ar883)
Afnan Arshad (aa2439)
Terryn Jung (tj223)
Github Repository: https://github.com/amitrajesh/CrosswordCreator

Crossword Puzzle Generator Written Report

Description

The three main goals of this crossword puzzle generator project were 1.) finding a set of legal English words such that all crossword constraints (character overlaps between "Across" and "Down" entries) are satisfied; 2.) providing informative clues for each of these words; and 3.) Making sure each word from the chosen set of legal words fit reasonably well into a particular category, or theme. Beyond this, we had an additional aim of turning this solver into an interactive application, which required developing a Graphical User Interface, and making sure that our solver worked at an acceptable minimum speed.

Software Written

An example of a crossword puzzle layout that most people are familiar with would look like the following Figure 1. Throughout the report, we will refer to each individual white space as a *cell* and a series of cells that make up a placeholder for a word as a *slot*. We can see that the layout in Figure 1 needs seven different slots to be filled with proper words.

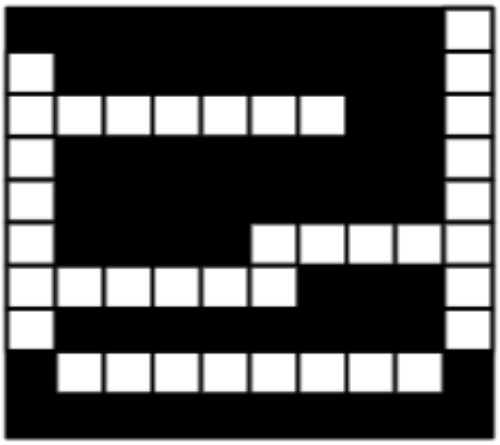


Figure 1: Empty crossword layout

The first step in creating the generator was coming up with a suitable way to translate a user-friendly crossword layout into a parseable, code-friendly crossword layout that a computer can work with. We found the cleanest possible way of achieving this was representing the crossword as a matrix of 1s and 0s. Figure 2 below is the software-friendly version of the user-friendly layout that was previously shown in Figure 1. We used a 2-dimensional list where the elements are either "0"s or "1"s. The "1"s indicate an empty cell to be filled with a character and the "0"s indicate blacks spaces in the layout that cannot be filled. Given a 2D matrix of 1s and 0s, the computer can identify where the "across" and "down" slots were with two passes of the matrix. The first time, we traverse horizontally; once a "1" was detected, we kept moving horizontally until a 0 was reached, and the resulting word was recorded as a word slot item. This process was repeated for the vertical direction for the "down" slots. Each of these word slot items contains the starting position of the slot in the matrix, the length, the orientation (across or down), and empty spaces for candidate words to be filled and their definitions. During the second scan of the matrix, we additionally keep track of *slot overlaps*, where different word slots share a letter. Every slot object has a list of overlaps, where each overlap is a 3-tuple containing the id of the slot that intersects with the current slot, and the row and column matrix location showing exactly where the overlap is. Importantly, overlaps can only happen between "across" and "down" slots, and so any position *(i,j)* in the matrix can be shared by 2 word slots at maximum.

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
 [1, 1, 1, 1, 1, 1, 1, 0, 0, 1],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
 [1, 0, 0, 0, 0, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 0, 0, 0, 1],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Figure 2: Software-friendly version of layout in Figure 1

With this information, the next task could be tried: filling in the crossword. At this time, no theme constraint was placed (any set of legal words could be used). The remaining constraints are formalized as follows: If a word slot is *n* cells long, the word that fills the slot must also have *n* characters. For each location *(i,j)* that is shared by word slot *m* and word slot *n*, each word corresponding to slots *m* and *n* must have the same character at location *(i,j)*. Finally, any word representing a word slot cannot simply be a random sequence of characters strung together, but rather a real, existing word that can be found in a standard English dictionary. A set of words that fulfill the three criterias mentioned above count as a legal solution for the given crossword layout.

To start, it was immediately clear that the choice of dictionary is critical. We initially used an API called Datamuse that returned a JSON of words that follow a certain format (e.g. 5 letter words with an 'e' for the third character) upon a specific URL request.

The next task was to find a set of words that would fit the constraints of the corresponding word slots on the crossword puzzle. In our initial, naive approach, we started with a shared 2D grid representing the solved crossword, exactly mirroring the layout matrix of 1s and 0s except that each element initialized to '?'. Then, we would iterate through each identified slot, retrieve its relevant information (layout matrix start location, direction, length, etc.) and read the shared matrix elements corresponding to that word slot to see which characters are already there to add as constraints ('?' serve as a wildcard, indicating any character works). Once the constraints were determined, we simply pinged the DataMuse API for all legal words that fit the constraints, randomly picked one, and wrote it to the shared 2D matrix (so later word slots can 'pick up' those letters added as new constraints). If the crossword generator comes across a slot where no suitable words existed, we simply backtracked to the last filled slot and picked another word to write to the slot. In order to save time, when a particular word slot pinged DataMuse for a list of words, the word slot "cached" about 10 random words immediately, so that backtracking can occur at that word slot 10 times without any extra API calls.

This framework became the baseline model for the crossword puzzle. And while it did work, we quickly realized that the algorithm runtime grew *exponentially* with the number of overlaps existing in the inputted layout. For example, even a 3x3 layout (all 9 cells are white) took upwards of 40 minutes to complete! Further research indicated that this constraint-optimization problem was NP-complete, and there was no clean algorithm we could

use to solve the NP-complete issue. Finally, it was clear that even in some of the solved crosswords, the words that were being used were extremely strange, with many having no definition at all in the Mariam-Webster dictionary.

It was clear that many changes had to be made. To begin, we switched out the dictionary: while DataMuse has a massive number of entries to choose from, some of the words were clearly not real English words. Additionally, API requests through the internet are slow; difficult crossword puzzles require thousands of words to be tested before a legal crossword will be found, and doing thousands of GET requests for a single crossword puzzle was simply too inefficient. To alleviate both of these problems, we instead turned to using a formal dictionary (Webster's Unabridged English Dictionary") that was saved locally. The conditional searching available in DataMuse was replaced by local python regular expression searches, immediately leading to improved word quality and fetch speed.

Despite all this, the overall time to complete the crossword actually went up, as gains in search/fetch speed became rather insignificant after considering how large the vocabulary reduction was from DataMuse to Webster's: DataMuse has access to over 500,000 words, while Webster has ~100,000 words, roughly 80% smaller. However, the solved puzzles were using more consistently legal words with Webser's, so we kept Webster permanently.

The process of switching, however, taught us that crossword creation time is not only dependent on the number of overlaps in the initial layout, but also the number of vocabulary words to choose from. Given that the problem is NP-complete, the only real way to achieve higher speeds was to introduce multiple heuristics so that the crossword can be filled in a much more intelligent manner.. Inspired by Ginsburg, M et al. "Search Lessons Learned from Crossword Puzzles" (1990), we decided to focus on three main heuristics:

The first was the "choice of variable". This particular heuristic dealt with the question of which world slot should be filled first. According to Ginsberg et al., it was observed that filling the word slot with the most constraints is the best heuristic. Thus, at each iteration, we made the crossword generator evaluate which word slot had the least amount of available words that fit its current letter constraints, and fill that word slot first. This heuristic is quite expensive, as it involves analyzing every possible word in every possible word slot at every iteration of the algorithm.

The second heuristic was “choice of instantiation”. Once we decide on a particular word slot, which word should we pick? There are in fact a host of ways to approach this question, but we focused on two. The first is essentially a “one step look ahead” approach: we take every candidate word for a word, place it on the shared matrix, and then see how many possible words are available for the *next* word slot (determined by the previous heuristic). The second approach is focused on overlaps: we take each candidate word, place it on the shared matrix, and for each word slot that *overlaps* with the current word slot, we calculate how many words are available for each one and multiply them all together. The word that maximizes this product is the word that is picked (we decided product instead of sum because even if one overlapping word slot has no options left after a particular word is put down, it is obvious that word should not be picked; using products means the “score” for that word will always be 0).

Finally, there is the backjumping heuristic. Naively, after encountering a word slot with no available words, we simply go back to the last picked word slot. It is clear, however, that this will not necessarily address the problem at all; if there exists a word slot for which no legal words exist for filling, then one of word slots that overlap with the word slot in question *must* change its word in order for the word slot in question to possibly have more words to pick from. Thus, the heuristic is simply backjumping instead of backtracking: after encountering a dead end, we find a previously filled, overlapping word slot that has alternative options to pick, and start from there again.

There are many combinations of heuristics possible, and further analysis about the effect of each one is seen in the Results section. Needless to say, these implementations drastically improved performance on crossword generation.

After creating a suitable set of words, the next milestone we aimed to achieve was to create a suitable, non-deterministic clue for each word in the crossword. Originally, our goal was to use natural language processing to condense and paraphrase the dictionary definitions to use as clues. However, after extensive research we learned that this was a very difficult task to accomplish within a short duration of time. Instead, we realized it would be much better to simply clean up the dictionary definitions provided by the JSON and manipulate them in the best way possible. “Webster’s Unabridged English Dictionary” provides nearly a paragraph of multiple definitions for many of the words. Additionally, the dictionary has a very clear format. Hence, we were able to use the consistency of the format to clean up the string and extract the

various definitions. Occasionally, there were times when the definitions would have the actual word itself. If this was the case, we used the Datamuse API from earlier to find synonyms and replace the word. The clues that we managed to generate ended up being slightly unreliable at times as there were infrequent cases of the software cleaning up the string too much such that there was nothing left. If given more time to work on the project, the clue making process would definitely be something we would work on to improve.

Our final major goal was to add themes; to make sure that our set of words that create a crossword are somehow related by a category. This is a famous problem in NLP (text classification) and while there are a myriad of ways to approach this problem using machine learning. We had serious debates over whether we wanted to create a supervised training model or an unsupervised training model. While unsupervised learning would not limit us on the number of different themes as it would generally cluster words together, it seemed unreliable and there was a need for huge amounts of data. There were also issues regarding how clusters would be linked to a certain theme. On the other hand, supervised learning would limit us on the number of different themes we could choose from as the training data would have to explicitly link strings to a specific output. However, it would work more reliably and it would allow us to have a word bank to work off of. Because having a word bank that is reliably categorized would be very convenient, we decided to take the supervised learning approach.

Our learning algorithm uses Naive Bayes to classify the words as various themes. As mentioned above, we used the 20 Newsgroup Dataset for training. The data set labels various newsgroup documents into 20 different categories. Additionally, we applied traditional NLP text cleaning methods, such as removing stop words and implementing the Porter stemmer algorithm, to our data in order to maximize precision. We wanted to have access to a word bank for each theme after training a single time rather than training every time before running the software as this would be much more efficient. We realized that because we had a limited supply of words, if we were to force the classifier to categorize words into all 20 different categories, we would have a serious limitation on the number of words we could use for each theme when solving the crossword puzzle. To avoid this problem, we decided to train with five different categories of the 20 such that each theme would have more words available. The themes that we chose were

“science: medicine” , “science: electronics”, “religion: miscellaneous”, “politics: miscellaneous” and “miscellaneous: for sale”.

Of course, the main concern in this process was that the Naive Bayes classifier trained by associating *documents* to certain themes, but in our crossword project we want the classifier to identify just *words* to certain themes. Originally, we thought that the paragraph-long definition for each word provided by the Webster dictionary would be long enough for a reliable classification. After training, however, we realized that there were some ridiculous classifications and clearly a paragraph was not enough information for our classifier to make informed decisions. To solve this problem, we decided to extract test data by web scraping Wikipedia articles that were relevant to the word in question. This document, if available, would be summarized, clearned, and given as input to the classifier. If non-existent, the Webster definition was used as test data. This leads to much better results. After one long session of training and classifying, we had a word bank which we could access anytime in the future for the crossword generator to work with.

Of course, the drawback is that while this method did a good job in grouping together similar words, it also further restricted our vocabulary: each word in the ~100,000 word dictionary got split into 5 themes, with one theme having as few as 15,000 words. This adds yet *another* slowing factor to take into account when judging the speed of our crossword creator, adding more importance to the major heuristics. In fact, there were situations where the crossword generator would struggle to find a suitable solution within a reasonable amount of time because the pool of words was too small. To solve these issues, we decided that after getting a layout and randomly picking one of five themes, we evaluate whether the given layout is feasible with the number of words that were classified into the theme using word length frequency distributions. If we find out that the current layout demands a certain number of words that the theme does not have, the theme is switched out; if none of the themes individually have safely enough words for the layout, then all the themes are pooled together, forming a 6th, “miscellaneous” theme that covers all words in the Webster dictionary.

After theme classification was accomplished, there was only one major thing left to do: make the application usable and friendly. To do this, we decided to create a GUI that would allow users to enter a layout in a similar format as Figure 1 and extract information from the user

input to data that the algorithm could work with. The GUI would have to be intelligent enough to notice when the user draws an illegal or impossible crossword layout immediately.

Within the GUI, we wanted to have users be able to enter their crossword layout without having to use “1”s and “0”s, but in a more intuitive way with black and white cells. The GUI was designed such that the user could click where they wanted a cell to be in their layout. Figure 3 shows an empty crossword GUI before any user input and Figure 4 shows what the crossword GUI would look like after user input.

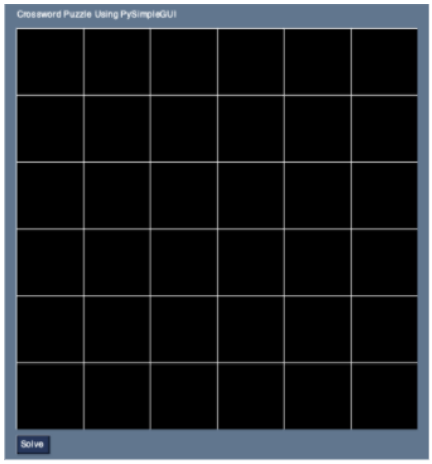


Figure 3: Empty crossword GUI

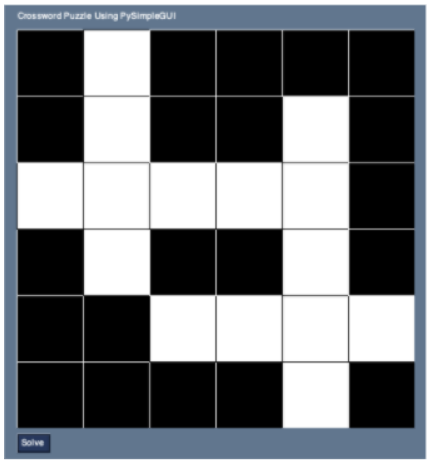


Figure 4: User-input filled crossword GUI

Afterwards, the user could click solve to start running the crossword generator software. The GUI input is turned into the familiar matrix of 1s and 0s, and the steps of theme picking, solving, and generating clues are followed until the crossword is solved. Once this is done, the GUI will take the relevant word slot data and return the same format as before, except with characters inside the cells. Figure 5, below, is an example of a filled-out, solved version of the user input provided in Figure 4. The key word here is an example as running the same layout could return different results each time as the entire software randomizes each step. It can also be seen that the clues for each word are at the bottom. These are the snippets of definitions from the clue-making process discussed earlier.

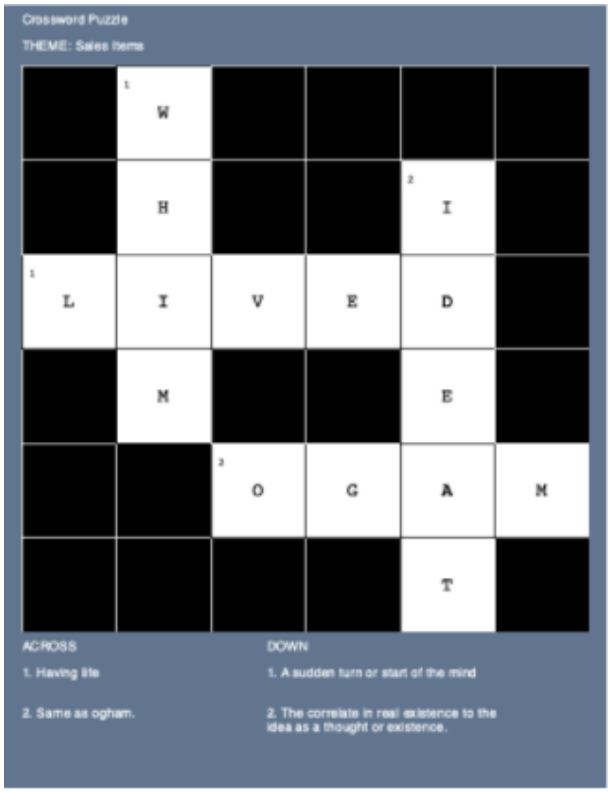


Figure 5: Solved crossword GUI

Results

While the crossword generator must be able to solve the layouts in an efficient manner, it must also be able to produce high-quality results. Because of this dual nature, we decided to evaluate our final product by objective and subjective factors separately.

In total, there are 3 classes of heuristics: for choice of variable, we have the naive method and the intelligent constrained-first approach; for instantiation, we have the naive method, the "one-step lookahead", and "overlapping product" approach; for backtracking, we have naive backtracking, and intelligent backjumping to actual problem areas. This leaves a total of 12 different combinations of heuristics to test. There are additionally a host of hyperparameters to consider. For example, there is the number of candidate words we "cache" upon searching the dictionary for a particular letter sequence, the minimum number of words present at a certain word length for a theme not to be "thrown out", etc. etc. To make matters worse, the ideal hyperparameters depend on what heuristics are being used; caching 10 word results is ideal when

using “one-step lookahead” or “overlapping product” for variable instantiation, but actually degrades performance otherwise.

In order to evaluate the performance of each heuristic as best as possible, we ran the crossword generator over 10 different combinations of heuristics while keeping the layout (a full 3x3 grid), vocabulary (constant theme of ‘sci.med’), and hyperparameters constant. Each run in the table below takes the form of a 3-tuple *(X,Y,Z).* X is the choice of variable option, and can be either 0 (naive) or 1 (most constrained first). Y is the variable instantiation option, can be 0 (naive), 1 (one-step lookahead) or 2 (overlapping products). Z is the choice of backtracking, and can be 0 (naive backtracking) or 1 (backjumping through overlaps). Because random sampling was used, leading to fluctuation in results, each run was done 5 times and the average time was reported. Also, the time limit for this study was 30 minutes; if 1 run is not completed within 30 minutes, the average time is simply reported as “30+ minutes”).

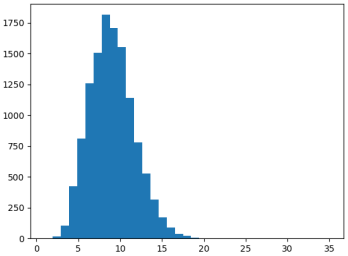
Heuristic Combination	Average time
(0,0,0)	30 + minutes
(1,0,0)	15.4107 seconds
(0,1,0)	30 + minutes
(0,2,0)	30 + minutes
(0,0,1)	30 + minutes
(1,2,0)	18.5940 seconds
(1,2,1)	22.2388 seconds
(1,0,1)	16.1382 seconds
(1,1,0)	44.8610 seconds
(1,1,1)	80.0266 seconds

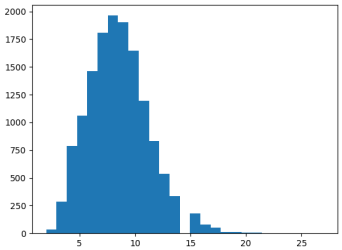
From this data, it is immediately clear that the “most-constrained variable first” heuristic is clearly the most important one. The crossword generator cannot solve a 3x3 grid layout at all

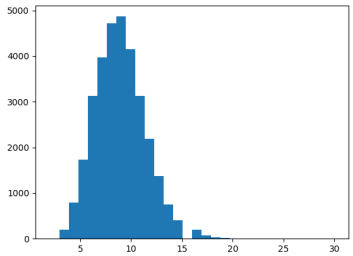
within 30 minutes without the heuristic, but with it can solve a 3x3 grid layout in as little as 15 seconds. In fact, looking at the table, it seems as though the “most-constrained variable first” heuristic is the *only* one that matters, as the fastest combination reported is simply (1,0,0). It is important to note, however, that different heuristics show their usefulness in different scenarios; due to the very high cost of “backjumping”, “lookahead” and “overlap product”, they will likely only save more time than they lose in large layouts with huge amounts of overlap.

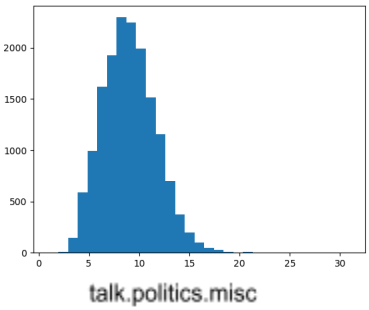
Another objective measure include puzzle accuracy (how many generated crossword solutions were legitimate) and clue accuracy (how many words had clues at all); after testing 50 various layouts (keeping heuristics, theme, and hyperparameters constant) it is safe to say that 100% of crossword solutions were legitimate, and 100% of them contained some form of clue.

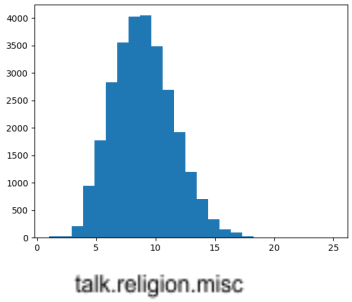
Finally, there is theme accuracy, which is how well each word set of a theme actually corresponds to that theme. This is mainly a subjective question that requires a human, but due to the large-scale nature of these theme sets it is clear that some form of objective criteria is needed to assess these word sets. So for each theme, a histogram analysis counting the number of words at a particular word length was done. Here are the 5 histograms:











Two major things are of note. Firstly, all 5 histograms are approximately normally distributed, meaning the ratio of different-length words in each theme is roughly the same, ensuring a certain degree of fairness (one theme will not dominate certain layouts over others). Secondly, the number of words in each theme is *not* distributed evenly; the medicine theme has close to 40,000 words, for example, while the "for sale" theme has a little over 15,000. This does have significant implications on crossword performance for different themes, as larger vocabulary means faster and more versatile crossword generation.

For the subjective factors, we decided to focus on three different criterias. The first is theme accuracy which is mainly concerned with whether the words generated follow a common theme and whether the theme is clear to see. We also wanted to evaluate the quality of the word which is about whether a typical user will be able to recognize the words that are picked. We did not want the words to be too obscure such that only experts could solve the crossword, but we also did not want the words to be too simple such that a typical user would not find the crossword entertaining. Finally, we wanted to evaluate the clue usability which is about whether the definition generated for a word is useful. Ideally, the quality of the clue would be in the middle of explicitly giving away the answer or being completely misleading.

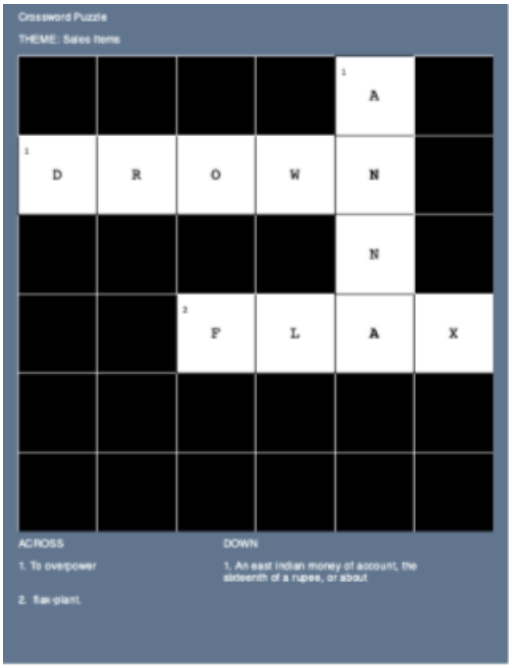


Figure 6: Solved crossword GUI 2

When trying to evaluate theme accuracy, we realized that words can often be interpreted in various ways to form links to a theme even if they don't seem remotely close at first glance. The best way to evaluate and explain this factor is to look at specific examples. Taking a look at Figure 6 on the left, it can be seen that the theme that the software chose was "Sales Items". The words that were chosen were "ANNA", "DROWN", and "FLAX". At first glance, it seems like "ANNA" is a good fit as it has the definition of money which can easily be observed within the context of sales. One can also come to the reason that "FLAX" is a sellable item and is therefore suitable for the theme. On top of that, we could easily argue that neither of these words would fit well for the other themes that are available such as medicine, electronics, religion, etc. The only word

that seems questionable is “DROWN”. One could argue that it fits if one uses the word in the context of “drowning” out sales competition, but it seems like too far of a stretch. Thus, we have evaluated our theme categorizing to be reasonably correct, but with occasional flaws. We are interested to see if a different training model (perhaps unsupervised) would have produced different results. We mainly believe that the occasional flaws arise from the fact that our Naive Bayes classifier was forced to categorize words in our local dictionary into five predefined themes. For future research, this could be improved by either including more themes or not having a set number of words that must be classified (i.e. words from a local dictionary), but rather creating a word bank on-the-go through web scraping.

For evaluating the quality of the word, we looked at various examples and judged whether the words were too simple or difficult. When checking that the words were not too simple, we made sure that words that a typical child would learn in kindergarten were not coming out frequently. Given the standard example in Figure 6 with words like “ANNA”, “FLAX”, and “DROWN”, we believe that our crossword generator gives sufficiently high-level words. When checking that the words were not too difficult, we made sure that the words were not obscure and not too acronym heavy. Previously, when using the Datamuse API, we realized that the dictionary was returning very strange acronyms that no group member had heard of. By changing our source of words to Webster’s dictionary, we were able to pick from a pool of viable words. Thus, we have evaluated our crossword generator to have words of good quality. Again, future research would involve improving the word bank through web scraping rather than having a hard-coded local dictionary. By using web scraping, there are possibilities of being able to use words that are more relevant to today which could increase the quality of our words.

Finally, for evaluating the clue usability, we once again looked at various examples and judged whether the clues were reasonable. Looking at the example in Figure 6, “ANNA” and “DROWN” have fairly well-written and descriptive clues that do not explicitly give the answers away. However, there are occasionally cases like the definition for “FLAX” - flax-plant. While we tried to stick to the format of the dictionary to efficiently clean up and divide definitions, there were too many variations and not all of them could be taken into account. Hence, there are times when there are flawed clues. As previously mentioned, a definitive improvement that could be made for future research would be using natural language processing to paraphrase the definitions. This would ensure that the answer isn’t explicitly stated in the clues.