Spring 2024

Machine Learning and Natural Language Processing for Crossword Puzzles

Finn Brennan
Bard College

Machine Learning and Natural Language Processing for Crossword Puzzles

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Finn Brennan

Annandale-on-Hudson, New York
May, 2024

Contents

Abstract

Solving crossword puzzles (CWPs) is a fun pastime to some, while others might struggle to imagine something so dry. Regardless, the vast majority of people have some degree of familiarity with crossword puzzles at least as an abstraction, and many people elect to make a habit of solving them on a regular basis. Crossword puzzles require the people who solve them to think in unusual ways and demand that they be able to call on esoteric and highly specific information that might or might not be obvious upon said solver's first examining a clue. It is compelling to imagine what exactly would go into using standard Natural Language Processing (NLP) techniques to solve crossword puzzles.

1

Background

The structure and the rules of crossword puzzles impose a unique set of constraints on the problem of designing an system that has the ability to solve them. Crossword puzzles can be tricky. To the totally uninitiated and inexperienced human solver, they can seem next to impossible at first. Crossword puzzles demand that human solvers have a broad base of knowledge that they can draw on in the process of determining each particular word that they want to put down onto the puzzle board. Crossword clues often make use of wordplay, take advantage of double meanings, and sometimes make reference to the answers associated with other clues on the board, or even those clues themselves. Puzzle editors will occasionally go so far as to write clues that point to conceptual and heady ideas like the puzzle's overall theme, or for instance, the emotion that might be evoked in a certain type of human solver who has the right context and understands the clue's intended meaning if they were to combine the answers to various other clues in some unconventional or non-obvious way.

Figure 1.0.1: The digital version of a New York Times Crossword Puzzle. Image source is link #1 in Appendix D.

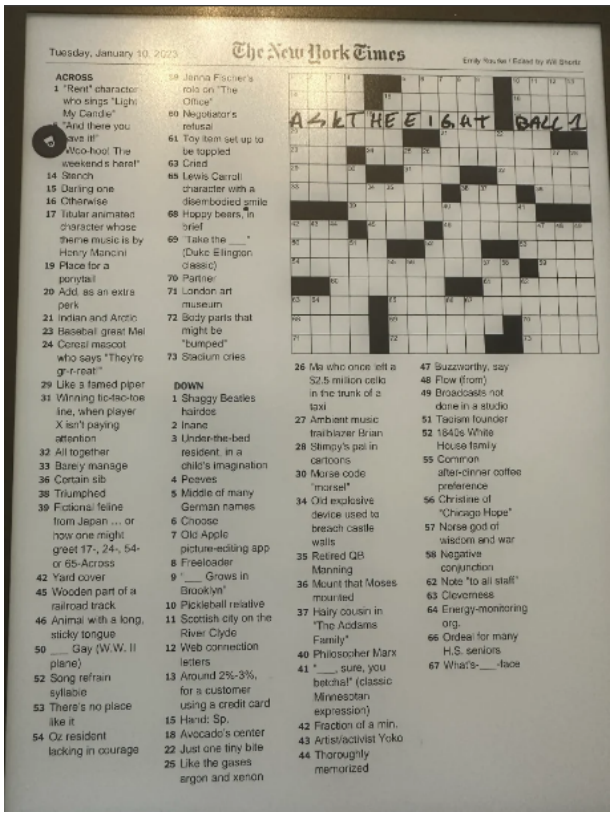

Figure 1.0.2: The classic version of a New York Times Crossword Puzzle. Image source is link #2 in Appendix D.

Beyond the layer of complexity presented by the clues themselves and the subtleties that go into actually making sense out of those clues, crossword puzzles also add depth to the puzzle solving experience in another way, which takes advantage of the thing that makes a crossword a crossword: the grid. The grid is the most iconic symbol of the crossword puzzle genre, and the layer of added complexity introduced by the grid is what distinguishes the crossword from any other sort of word puzzle one might encounter. The grid layout of crossword puzzles places further constraints on the process of selecting which potential answer word to play on a given clue spot, in addition to the semantic constraints given by the clue. From the perspective of a human solver, the grid layout of a given puzzle dictates which answers will have to avoid having any conflicting characters with the answers to the other clues which run perpendicular and intersect.

If you have ever done a crossword, you have perhaps experienced firsthand the fact that the puzzle becomes easier in one sense while at the same time becoming more difficult in another as the puzzle board gets filled in and thus more constraints are put on the solver's decision making process. Solving the puzzle gets easier as answers are played on the puzzle board in the sense that there is much less ambiguity in a mostly-solved puzzle grid than in an empty one. If most answer slots are filled, then many of them must cross other filled-in answer slots. If these intersections do not produce any conflicts and instead fit together, then a human solver is given a sense of confidence in their previous decisions. Similarly, a solver system should prefer potential answers that fit the constraints of the current puzzle board state and justify previously played words. Guessing a word that already has multiple characters filled in is a significantly easier problem than guessing a word that has no characters filled in. As such, each word guess that is played on the puzzle board narrows the possible solution space which a solver system or a human solver would have to explore in guessing at the answers whose spots on the puzzle-board intersect the initial answer candidate considerably, which will make it easier for either kind of solver to eliminate the fill candidates that do not fit the puzzle.

Note: throughout this project the terms "fills" and "answers" are used frequently. While their meaning are similar there is a key distinction between the two. This distinction is that for the purposes of this paper "answer" will be used to mean the correct answer that is associated with each clue in any particular crossword puzzle. The term "fills", on the other hand, will be taken to refer to possible answers that are generated by a crossword solving computer model.

At the same time, at least from the perspective of a human solver, and somewhat counter-intuitively, a puzzle might start to seem more difficult as the puzzle board becomes filled. One pretty typical solving strategy that is employed by human solvers would be to scan through the clue lists as well as the puzzle grid in order to determine which clues appear the easiest to solve. The puzzle solver would then used the information gained (the characters from solved clues which run into other, unfilled answer spots) to help them tackle those clues that at first seemed trickiest, since those words will have some number of characters filled and that will shorten the list of answers that could possibly fit in the clue spot. The problem that human solvers will often run into at this point stems from the fact that knowing a few characters of the word or phrase they are looking for will not necessarily give them enough information to come up with the correct answer. In other words, a partial solution, or even a collection of interconnected partial solutions to a given puzzle could very well not provide a human solver with enough data to overcome the information barrier presented by the current state of the puzzle in question. A computer program, on the other hand, would be able to totally sidestep this issue if it was trained on relevant data in adequate quantities. When faced with this type of situation a model should have all the information it needs to make some sort of inferences from the puzzle state (clue and constraints) that should at least be able to generate some kind of informed direction to its search. To that end, a computer based crossword solving system needs to be able to represent the meaning of clues in some way such that it's able to focus its search of the massive solution space in a manner that makes some sense and is able to

generate and select appropriate fills. There are a number of natural language processing techniques which could be of use in deriving meaning from some set of crossword clues, some of which will be explored in this project. But first we examine some previous work that has been published that involves solving crosswords with computers:

2

A Brief History of Automated Crossword Puzzle Solvers

Over the past few decades there have been many projects undertaken the application of computer models to the problem of solving crossword puzzles. A few of these projects have produced very insightful papers on the subject, some of which will be the concern of my discussion in this section.

The first of the papers that will be discussed here is also the first of them in terms of chronology. The paper is entitled *A Probabilistic Approach to Solving Crossword Puzzles*[2], and it was published in the year 2002. The solving approach taken by this paper is one that relies on a collection of separate specialized "expert" (Littman et al, 2002)[2] solver modules whose puzzle solutions are combined and coordinated by a "centralized solver" (Littman et al, 2002)[2]. The solver presented in this paper was able to achieve 95.3% accuracy in terms of target words correctly guessed by the solver. This is score

indicates that the solver is very effective on the dataset upon which it was tested, but it is still less effective overall than some of the others that have been created.

The second paper that seems worth mentioning here is called *Crossword Puzzle Resolution via Monte Carlo Tree Search*[1], and as its name alludes this paper describes a pretty successful approach to solving crossword puzzles with computers that centers around the Monte Carlo tree search. The paper was published in 2022, making it the most recent solver covered here. The solver outlined in the paper accomplishes that feat by re-imagining the problem of solving crossword puzzles as a Markov Decision Process or MDP. (Chen, 2022)[1] This solver is able to achieve an accuracy 97.04% across the puzzles that it was tested on as far as correct answers arrived at by the solver, making it approximately 2% more effective than *A Probabilistic Approach to Solving Crossword Puzzles*[2].

The third and final crossword puzzle solver paper that will be examined here is titled *DR.FILL: Crosswords and an Implemented Solver for Singly Weighted CSPs*[5] and it was published in 2014. This paper describes a crossword solver that first reinterprets the crossword solving problem as a weighted constraint-satisfaction problem. The solver then uses techniques, heuristics, and search algorithms that are specially crafted to suit the weighted constraint-satisfaction problem representation of crosswords to find solutions to the weighted constraint-satisfaction problem. (Ginsberg, 2014)[5] DR.FILL was able to achieve 95.8% word accuracy when tested, meaning that it performed a bit better than *A Probabilistic Approach to Solving Crossword Puzzles*[2], but worse than *Crossword Puzzle Resolution via Monte Carlo Tree Search*[1].

Of these three papers, none went into much detail about the data structures that were used to represent the puzzles in the process of solving, opting instead to focus their attention on the mathematical foundations of their respective solvers as well as the theoretical formulations that their solvers rely on. However, each of these papers did describe the dataset used to train their solvers. The datasets used in crafting the

solvers presented in *Crossword Puzzle Resolution via Monte Carlo Tree Search*[1] and *DR.FILL: Crosswords and an Implemented Solver for Singly Weighted CSPs*[5] were both significantly larger than the one used in this project. This is contrasted by the dataset used by the solver described in *A Probabilistic Approach to Solving Crossword Puzzles*[2] which is much smaller, only drawing on a dataset consisting of 5142 unique crossword puzzles. (Littman et al, 2002)[2]

3

Data Collection & the Corpus

3.1 Utilities

There are a few comprehensive and publicly accessible data-sets available which compile crossword clues along with their corresponding answers from a wide array of crossword puzzle publishers.

Representations of entire puzzles on the other hand, that is clues/answers in addition to a map of the puzzle grid, are generally a bit more closely guarded by crossword publishers, and as such, large scale data-sets of full crossword puzzles are not nearly so readily accessible as clue/answer data. There are however accessible tools that can be used to put together such a data-set. One such tool is Xword-dl[7], which is a Python script created by Parker Higgins that can be run from a command line and allows users to either download a puzzle using a direct link to the publisher's site hosting the puzzle file, or specify arguments like the puzzle's publisher and date in order to grab and download

that particular puzzle from the publisher's servers. Xword-dl was utilized to collect all of the puzzle data that is used in this project.

The crossword puzzle industry standard for storing digitized crossword puzzles used by most publishers is a propriety file format called PUZ[4]. There is no publicly available official documentation for this file format, however crossword enthusiasts have been able to more or less entirely reverse engineer the format in such a way that files with the ".puz" filename extension can be easily converted into more readable and thereby usable formats. For this project, PuzPy[8], which is a program written by Alex Dejarnatt that parses ".puz" files, was used to parse the files ".puz" files downloaded from puzzle publishers so that the puzzle data contained within those files could be read and then written into either csv files or some other form of text files, loaded into objects and Pickled for later use, or simply put to use at parse time, either wrapped in some data structure or in their raw post-parsing form.

3.2 Corpus

By combining these tools, it is a very straightforward process to write a script that runs through a range of dates, downloads a puzzle from each date, parses it, saves it in some way, and then continues on to the next date. As of writing this I've used this method to download and convert to csv files a total of 18,437 puzzles from the following sources: The LA Times, Newsday, The New Yorker, The NY Times Mini, and The NY Times. This set comprises all of the puzzles put out by those outlets from the earliest available on their site up until 12/1/2023 (except the puzzles from dates that generated errors and could not be found, these dates are also saved to a csv file). The table below shows each particular source along with the number of puzzles that were downloaded for use over the course of this project:

Outlet	Num. Puzzles
LA Times	822
Newsday	2370
New Yorker	906
New York Times Mini	3390
New York Times	10949

Table 3.2.1: Puzzle Publisher and Number of Puzzles Downloaded

The *Outlet* column of the table lists the crossword puzzle publisher, while the *Num. Puzzles* column shows how many puzzles from each respective publisher are included in the dataset used for this project. Later, in Sections 4.3.3 and 4.3.5, we will describe the manner in which this dataset is pre-processed and eventually subdivided for use in this project's experiments.

4

Methods

4.1 Design Notions & Preliminary Discussion

In order to solve a crossword puzzle, a computer model needs be able to at once derive some meaning from clues in order to narrow down the list of prospective answers it must consider, while also making sure to take the restrictions imposed by the puzzle's grid into account. The goal of this project is to train a model that can take in a current puzzle state that may be almost solved, totally unsolved, or anywhere in between. The model should then examine all of the puzzles' unanswered clues in its current state, answer the one that it is most confident about, and then repeat the process by feeding the next puzzle state (i.e., the previous state with the addition of the most recent guess) back through the model.

There is a wide array of different sorts of NLP approaches that one could conceivably take in trying to put together a system that is able to iteratively work towards the desired solution state of any particular crossword puzzle, with that of course being the state in

which all clues are answered correctly. Just as there is no clear singular optimal way for even the most qualified and experienced human solver to approach solving a CWP, there is not a single obvious best manner in which one ought to go about representing a puzzle digitally such that the puzzle may be solved by a model employing any given combination of NLP techniques. Indeed, it may well be the case that there are infinite ways of breaking down or re-framing the problem of solving crossword puzzles either in terms of smaller NLP problems or by focusing solution efforts on one particular aspect of the puzzle. As such, as a programmer it is necessary to make certain decisions in designing a CWP solving model that focus and give shape to the solver's exploration on the areas of the massive solution space which are most likely to advance the model along a path whose end is the goal state. These decisions inform each step that the final model would take, as well as how exactly the model will orient itself towards however it represents its goal, and then take steps toward that goal, in addition to dictating the model's performance more broadly.

As far as this project is concerned, crossword puzzles as a whole are understood to be a kind of constraint satisfaction problem, where puzzle constraints are represented as constraints placed on our representation of puzzle state. This representation will be described in more detail but for now it is important to note that a puzzle state in this context is essentially a representation of the puzzle itself, as well as the collection of all of the fills that have been played on that particular puzzle. Each unique puzzle-board comes along with its own equally unique and corresponding set of constraints which are placed upon the problem of generating fills for the current clue of the given puzzle that do not produce conflicts, and both fit the space available on the puzzle-board, in addition to falling in line with the semantic information indicated by each clue. A puzzle is not solved until all of its constraints are met. As such, a model for solving crossword puzzles must have some way of representing the constraints imposed by each individual puzzle.

Moreover, this information must be available and quickly accessible by the solver system at both training and testing times when processing each individual puzzle.

Beyond simple representation of the constraints which functionally constitute a CWP, a solver system needs to be able to employ some kind of method that allows it to make progress in the direction of its goal state. As a consequence of the nature of crossword puzzles and their objective which is, of course, the correct answering of all clues, the only form that this progress can take is that of novel (with respect to the current puzzle state) answer fills. To put it more concisely, the only way to solve a CWP is by associating the correct fills with each clue on the puzzle-board. This point dictates the form a that CWP solving system must take. The approach we explore is a system that must generate possible fills, evaluate those fills in terms of the degree of their impact on the current puzzle state as it relates to the goal state, select from among possible fills that fill which maximizes some metric of proximity between the current puzzle state and the goal puzzle state, and finally repeat that process until each clue is answered and each of the puzzles' constraints are met. Evaluation could conceivably take place at the level of the state, in whatever form it is represented by the model, or at the level of the fill. Since the difference between states over solver iterations will be expressed in terms of completed fills added to the puzzle state at hand, it may make more sense both in terms of computational complexity and memory efficiency to generate some metric to gauge a given fill candidate's viability using as little information as possible about the puzzle's state as possible. For instance, in generating a fill for some clue under puzzle's across section, the system could evaluate potential fills using minimal information by only passing on references to the down clues whose answer spaces intersect along the puzzle board, as well as the target lengths of the intersecting clues.

The trade-off inherent in implementing this sort of scheme for handling fill generation and simplified state evaluation is that information will certainly be lost which will inevitably eventually be necessary in order to generate fills that are even remotely related

to the target fill for certain kinds of clues. For instance, any clue which makes reference to another clue on the puzzle-board whose answer spot does not physically overlap with the answer spot corresponding to our initial clue would no doubt be misinterpreted by this sort of system, since the model would not have access to the clue which is being referred to due to their lack of physical proximity and intersection on the puzzle-board. It is certainly possible to provide for these kinds of clues within a puzzle solving model by building in the functionality that would be necessary to recognize these special kinds of clues, and redirect the solver model's approach in some way as a function of the solver's understanding of the various interrelated clues as well as their respective meanings in relation to one another. (Ginsberg, 2014)[5] This could be accomplished by creating some kind of helper function for the solver model which looks through a given test clue and parses it to determine whether it refers to any clues outside of its immediate vicinity on the puzzle board. A similar problem arises in the case of novel test clues which rely on an overall theme which has either been embedded in the meaning of the test clue's puzzle at large or to some particular set of that puzzle's clues, as this kind of system has no way of representing thematic information or any kind of information beyond the scope of the intersecting clues.

Having these considerations in mind, Python made sense as the language to use in the development of a model for solving CWPs. There are many massively useful, well documented, and widely used libraries out there for Python which provide the language with a lot of the functionality that's necessary in this kind of project. As mentioned in Section 3.1, this project uses the library PuzPy for parsing the crossword puzzle files from the form in which they are made available into something more useful in terms of this project, the details of which are explained in more precisely in Section 4.2. Additionally, this project uses the Python libraries numPy, NLTK, sklearn, and PyTorch (among others). The majority of the libraries used in this project, and all of the ones listed above are

popular and well supported choices for the sorts of NLP and machine learning problems that arise over the course of this project.

4.2 Data Structures

4.2.1 Data Structures: Overview

The collection of crossword puzzle data must be represented and accessed by the model. Various factors played into this. These factors include memory use, ease of access, particularly at the solver model's runtime, and, of course, time considerations, both in terms of the timeline for the project as a whole in addition to the time considerations relating to training and testing the different applications that are examined over the course of this project. Each of these factors played some part in the decisions that influenced the shape this project took, although some considerations certainly shifted in the degree to which they influenced this project as the project progressed. Time considerations for example, while always being kept in mind, naturally became more pressing as the project progressed. Similarly, memory considerations were not of much concern during this project's early stages, when experimentation was most important, it became important to have some code that is able to run on a personal computer in a reasonable amount of time.

The most pressing of these considerations, and the one which had the most significant impact on the design of the primary data structure utilized by this project, is the need for accessibility. The information comprising a complete representation of a crossword puzzle is needed for each puzzle across our entire corpus. Our solver model and any kind of computer model designed to solve CWP needs to be able to refer to this information at test time, but also at training time. This is especially true in the context of the specific machine learning techniques whose performance in the context of a CWP problem this project explores, which are discussed at greater length in Section 4.3 with their performance being examined in detail in Chapters 5, 6, and 7. Once a '.puz' file in our

dataset is parsed, the data returned by the PuzPy parser associated with each puzzle in our corpus needs to be organized in some way that allows for us to refer to it during training and testing in a way that's fast enough that it does not impede our ability to train and test in reasonable time.

With these factors in mind, Python dictionaries emerged as a reasonable solution. Python dictionaries allow us to associate keys with their corresponding values by way of a hash function, which has $O(1)$ search time and is less computationally taxing at runtime, whether testing or training our solver, than $O(n)$ lookup in either a Python list or a NumPy array would be. The layout of the information that comprises any given CWP in our dataset is such that said data is able to be organized and wrapped in multi-level dictionaries. In this way, the manner in which the individual components of puzzles within our data are naturally organized gives rise to a certain kind of structure. This structure needs to be represented adequately in some way by the dictionaries that our solver will be referring to at runtime. Conveniently, Python dictionaries lend themselves to this sort of application, in that they are well suited for this kind of implementation of a multi-tiered look-up system. Since Python dictionaries can accept other dictionaries as the values to which the dictionaries' various keys are mapped, it makes sense to imagine a system in which dictionaries are configured in a way that allows for the information that a CWP solver requires regarding any generic puzzle that the solver might encounter at runtime to be accessed by said solver. For the purposes of this project, as is explained at greater length in Section 4.2.2, it made the most sense to convert the puzzle data contained within individual ".puz" files directly from the form in which it is returned by the PuzPy parser into dictionaries and lists that are more relevant and useful in the context of what this project is trying to accomplish than the data that's returned directly from the parser. This regrouped puzzle data is then passed to a constructor in creating an object of a data structure that was created for this project, called PuzRep. PuzRep's features and design as well as insight into the choices which shaped them are the subject of Sections

4.2.3 through 4.2.5. For now, it is worth mentioning that the dictionaries underlying the PuzRep contain all of the clues and answers of a given puzzle, separated into across and down sections, as well as a cell number denoting each of the clues' lengths and starting position with respect to the puzzle's grid. More information on the grid, its layout, and design can be found in Section 4.2.4.

4.2.2 Data Structures: PuzPy & Data Filtering

As mentioned briefly in Sections 3.1 and 4.2.1, this project uses the open-source PuzPy library for the task of making sense of the otherwise illegible '.puz' files. This file type was created by commercial crossword puzzle distributors to be intentionally somewhat uninterpretable without the aid of specialized software, but has since been reverse engineered by CWP enthusiasts. (Myer et al)[4] PuzPy allows us to pass it ".puz" files in their raw, unprocessed form, and then returns a parsed puzzle object that has certain attributes. By referencing these attributes, we are able to extract the information outlining a particular CWP from its corresponding ".puz" file.

With the parsed puzzle data returned by PuzPy, we are free to organize the dictionary representation so long as the representation chosen is able to encapsulate all of the information we will require in generating puzzle fills for the in the process of solving a puzzle, or in testing our solutions. For this project I decided that it would make the most sense to write a helper function which generates a dictionary representation of a given puzzles clues and answers, as well as information regarding the lengths of particular clues' corresponding answers, their indexing with respect to the representation of the puzzle grid, clue numbering, as well as the status of a particular clue as belonging to either the 'across' or 'down' categories of clues. This helper method is inventively called *get_puz_info*, and while its precise functionality will be explored more in Sections 4.2.3 and 4.2.4, it is mainly important to note now that this function creates the layered dictionaries described above, as well as a visual representation of each puzzle's corresponding grid stored in a two di-

mensional Python list which contains a uni-code character based representation of both the puzzle layout itself, as well as its numbering scheme.

4.2.3 Data Structures: PuzRep

As previously stated, in order to have the puzzle information in this project's corpus organized in a more accessible and usable way, the information returned by *get_puz_info* is immediately returned to an object instantiation, which creates an instance of a PuzRep object. PuzRep allows for various useful operations to be performed on a somewhat simplified and lightweight representation of a CWP. These operations are useful not only in the context of visualizing a puzzle, and especially visualizing a whole puzzle as it is being solved, but are also absolutely necessary in the process of solving a CWP itself. By creating a class of objects to represent puzzles in a way that is easier to work with, we are given much more freedom in the ways we can manipulate and interact with the puzzles themselves than might be had otherwise.

4.2.4 Data Structures: The Grid, Numbering, & Constraints

One such operation involving the representation of puzzles, which the PuzRep class simplifies drastically both in terms of computational efficiency upon assesses but also in terms of general conceptual clarity, is the ability of PuzRep to maintain and manipulate an easily malleable representation of the physical grid on which the particular puzzle in question would be played. The PuzPy library returns a puzzle object as the result of each '.puz' file parse which includes a minimal representation of the puzzle grid using '-' characters to represent open spots on the grid and '.' characters to represent the puzzle grid spots which are blocked off. By modifying this representation in a few simple yet impactful ways, we are able to arrive at one that is marginally only more visually appealing, yet significantly improves interpretability when compared with the representation that PuzPy returns.

Using the numbering dictionary which PuzPy provides as an attribute of the puz object corresponding to each puzzle parsed by the library, it is possible to find the cells on the grid representation of the puzzle returned by PuzPy on which any particular clue begins as well as the number corresponding with the clue in question. Using that information, I put together a representation of the puzzle grid associated with each individual puzzle in this project's corpus that consists of a list of strings that is twice the size of the puzzle grid itself. The discrepancy in size between the PuzPy representation and my own PuzRep one stems from the fact that the PuzRep representation includes an additional line for each line in the puzzle, on which the puzzle numbering is displayed. Due to the fact that puzzle clue numbering in this project's corpus extend well into the triple digits, while falling well short of the quadruple digits, it was also necessary to insert three spaces between '-' symbols in the strings representing the puzzle grid. These spaces allow for the puzzle's clue numbering to be displayed in a readable manner, such that the numbering appears on the line in our list of strings just above the the spaced out grid, in line with its appropriate starting cell.

4.2.5 Data Structures: Fills

Another class of puzzle operations that are critical to the design of a CWP solver, and also representation of and interaction with CWPs more broadly, are fills. Having a reasonable way of representing fills as they are being generated by the solver is critical to developing any sort of solver for CWPs, since a CWP solution is necessarily some special predetermined collection of individual fills. It would also be reasonable to assume that a CWP solving model might need to be able to perform some sort of backtracking in the process of generating the collection of fills that will comprise a puzzle solution. It therefore stands to reason that the PuzRep puzzle grid representation ought to have some sort of 'erase' function that allows a solver to remove a particular fill from its grid representation. It is precisely this functionality that is provided by PuzRep's *fill_word*

and *clear_word* methods. As described in Section 4.2.4, the grid representation employed by this project stores the grid associated with each puzzle as a list of strings, where each alternating string is either a numbering line, or a line representing the puzzle's actual grid. Each of these two methods rely on the function of the PuzRep object called *fill_cell*. This function simply takes in any Unicode character and inserts said character into the string representation of the puzzle's grid at the appropriate index with respect to the string. Both *fill_word* and *clear_word* function by making repeated calls to *fill_cell*, the only distinction being that *fill_word* also accepts a word as input (which must match the clues required length in order to work correctly) and calls *fill_cell* on each cell in the clue's answer spot using each individual character in the word that is to be filled onto the puzzle grid as the input characters for the *fill_cell* function. Meanwhile, *clear_word* performs an almost identical task, only instead of breaking a work into characters and inserting those characters into the grid, *clear_word* simply inserts '-' characters across or down a words entire length, thereby removing all characters of the word in question from our PuzRep object's grid representation. Both *fill_word* and *clear_word* also take in a string parameter which indicates whether the clue is to be filled or cleared in the down or across directions. Additionally, the PuzRep object class allows us to print the current state of the puzzle, return the current state as a string, or reveal all of the solutions to a given puzzle by filling in all of its correct answers using the fill function described above as well as the true answers stored in the dictionaries that are outlined in Section 4.2.1. The full code for the implementation of the data structures designed as part of this project for use in the development of CWP solvers can be found in Appendix A.

4.3 Clue-Fill Mapping

4.3.1 Clue-Fill Mapping: Overview

The first problem that presents itself in the course of designing a solver for crosswords, and the problem that is central to solving any kind of traditional CWP is the problem of creating some kind of mapping between any arbitrary clue that the solver encounters, and either one predicted fill or some subset of the possible fill space. Human crossword solvers must perform a version of this mapping too. A consequence of the fact that the solution to any given CWP is unavoidably and by definition going to be a collection of fills is of course that any CWP solver must be able to in some way arrive at some fills which would comprise such a collection.

It is also the case that the best kind of mapping that the crossword solving model could apply toward restricting the set of all possible fills to some subset of relevant possible fills given the all of the puzzles constraints, would naturally be a mapping that is able to somehow take all of those constraints into account and generate a set of predicted fills. These puzzle constraints include of course target length, but also intersecting answers, among others described previously. If a CWP solver can narrow the field of possible fills in a way that reflects accurately the puzzle constraints, the solver should be able to make progress towards the set of puzzle fills it is looking for. The rest of Chapter 4 will explore the background behind, and this project's implementation of, two approaches towards creating the kind of mapping described above that differ greatly from one another in just about every way imaginable.

4.3.2 Clue-Fill Mapping: KNN - Background

The K-Nearest Neighbors (KNN) algorithm is a non-parametric supervised learning method that is used in different kinds of applications, generally for either classification and regression. (Jurafsky, Martin, 2024)[3] Since this project's corpus consists of many

puzzles, which each in turn consist of two sets of clues and answers (that is, the 'across' and 'down' sets) it seems to make sense to view the association between clues and answers as a kind of classification problem in which the set of possible classes for a given input is the set of possible fills. The KNN algorithm in the context of classification essentially functions exactly as its name would indicate. The KNN algorithm's training phase is not really much of a training phase at all, since all that the algorithm typically does in the way of training is store the input samples (in this case: some representation of our training set of clues that is explored in Section 4.3.3) alongside their corresponding known true answers. At test time, the algorithm simply uses some user determined distance metric to find some particular number of examples from training that are closest to the test input that the algorithm is being given. The number of points from the training set considered is denoted K as in K-Nearest. Once the algorithm has arrived at the K-nearest examples, it calculates the majority class label (fill) associated with each of the nearest train points. The KNN algorithm then finally returns the label that was held by the majority of its K-nearest Neighbors.

The KNN algorithm made some sense as a first choice in generating a baseline in the performance of a mapping between clues and fills that a CWP solver might make use of for a couple of main reasons. The first of these is that KNN is conceptually pretty simple, which is nice because that means its performance might be easier to make sense of in the context of CWPs. KNN does not require much prepossessing in order to arrive at reasonable classifications, and also the algorithm does not rely on anything aside from the training data in drawing its conclusions. It is important to note however, that the KNN algorithm can not on its own calculate distance between strings of text, and so words used by the algorithm must first be converted into a vector of numbers for which a distance measure exists.

4.3.3 Clue-Fill Mapping: KNN - Implementation

The first order of business in applying KNN to the problem of generating a clue-fill mapping as it has been outlined in the previous sections is to assemble our corpus by extracting all of the clue-answer data that has already been wrapped in PuzRep objects and storing that data in two Python lists of the same length, so that it can be easily referred back to later. This is done by simply iterating over all of the relevant values in the dictionaries described in 4.2, that is to say, all of the clues and also all of their answers for every PuzRep object in the corpus.

At this stage, the clues-answer data needs to be prepossessed to some extent, and so this project uses the *nltk.tokenize.RegexpTokenizer* from the NLTK library to tokenize the sentence(s) within each string representation of a CWP clue, thereby converting the strings of sentence representation of each clues into one that consists of a list of individual word strings, removing all punctuation, and converting each word into lowercase. Next, I removed all clue-answer pairs from the dataset in which the clues contain only symbols, and were thus left completely empty as a consequence of the previous step. This is done by simply removing any empty lists from the dataset alongside their answers. Then, I created dictionaries that associate each word in the clue and answer vocabularies separately with their individual frequency counts across the whole corpus. With these steps completed, I then began the process of converting each clue in the corpus into a distinct clue vector.

There were many different approaches that could have been taken at this point in the project. There are plenty of effective techniques for generating vector representations of individual words, each with their own sets of benefits and drawbacks. The method that seemed the most interesting in the context of this project and CWPs as a whole was using Word2Vec to create word embeddings that are meaningful in the sense that they capture some of a word's meaning, and are especially capable in representing the meaning of words relative to their particular context in the dataset on which the embeddings were trained. (Mikolov et al, 2013)[6]

For this project the Gensim implementation of Word2Vec is utilized in generating embeddings for the purposes just described. The Gensim Python library makes training embeddings on a custom dataset incredibly straightfoward, simply requiring that one pass Word2Vec a list of tokenized strings from which it will generate word embeddings. Since the clue-fill mapping for the CWP solver needs to be able to represent the meaning of the words within the clues in our dataset in the context of the clues in which those words appear, but also in a larger context that extends beyond the training set. As such, it seemed reasonable to train the Word2Vec word embeddings used in this project on a dataset that combines the all of the clues within this project's corpus with some dataset containing a lot of common words in their usual contexts. The *Brown* corpus, as provided by NLTK seemed like a fine choice. In order to be able to gauge whether or not the dimensionality of the word embeddings used in this project has any impact on the KNN clue-fill mappings' performance, embeddings were trained at three different sizes. The sizes of embeddings trained for use in this project being 50, 100, and 300 dimensional, while attempts to train 500 dimensional embeddings resulted in memory errors.

Once trained and saved, these new embeddings were used to convert each of the clues throughout the entire dataset from lists of token strings, into singular clue vectors of consistent sizes. This is achieved by converting each token string in the original lists into a NumPy vector using the Word2Vec word embeddings, and then just averaging them out by summing all vectors in the list (i.e., embeddings corresponding to words in a given clue) and dividing them by the list's length (the number of words in the clue).

With list representations of the clues in the dataset now replaced by the clue vector representation, the set of all of clue vectors along with their corresponding classifier labels (fills) was split into train, test, and validation sets. An 80-20 split was first performed between the training and testing sets. Then a subsequent 80-20 split was performed on the train set in order to create a validation set.

With the data arranged properly, all that was left was to pass the new training set over to the scikit-learn KNN classifer module in order to to create a model, and then evaluate that model's performance over the test set. In the hopes of understanding whether or not the number of neighbors considered by the KNN classifer has any significant impact on the model's performance, the testing phase was repeated not only on each of the sizes of word embeddings that were trained, but also with both the values 3 and 5 being taken as the model's K parameter. Testing for each of these variations on the KNN clue-fill mapping consisted of simply running the KNN classifer on the test set, recording its classifications in a list, and then using scikit-learn's metrics module to get a sense for the mapping's performance across the test set. Additionally, one specialized metric was examined for use in the context of CWPs, which is termed 'length accuracy' and measures the ratio of cases in which the classifier predicted a fill that was the correct length given the constraints of the puzzle. That is, the length of the predicted fill matches the length of the actual answer. The results of these tests can be found in Chapter 5, and the code for the full implementation of the KNN classifier can be found in Appendix B.

4.3.4 Clue-Fill Mapping: RNN - Background

Another means of generating a mapping between clues and fills for CWPs is by applying a neural network to the problem. Neural Networks (NNs) as a whole are good at learning decision boundaries that are not linearly separable, the classic example of this being the XOR problem, which Neural Networks have no problem learning. (Jurafsky, Martin, 2024)[3] This property of NNs, as well as their wide-ranging success in a variety of other applications made them an attractive option in the development of a clue-fill mapping for the solving of CWPs. Recurrent Neural Networks (or RNNs) are generally considered to be an especially effective variety of NN in problems that involve sequential data as a result of the bidirectional manner in which RNNs propagate information through their layers. Because the clues in this project's corpus can be understood as being sequences of

the words which are contained in those clues, the RNN model seems to lend itself nicely to the problem of mapping clues to fills.

4.3.5 Clue-Fill Mapping: RNN - Implementation

This project uses the PyTorch library in order to create an RNN for the purpose of mapping between CWP clues and possible fill candidates. PyTorch significantly streamlines the process of creating and training NNs in Python, and this project uses PyTorch to that end in investing

It was necessary to convert the clues which are part of this project's corpus from their original string form into a numerical form that can be put to use by the RNN. This project uses four Python dictionaries to arrive at this sort of representation. The first two of these are the the same clue and answer vocabulary-term frequency dictionaries used for the KNN clue-fill mapping and discussed in Section 4.3.3. The second pair of dictionaries used rely on the the first pair, and are constructed by simply iterating over all of the keys in each of the first pair (i.e., each unique term in the clue and answer sets, separately) and assigning a unique integer to which each term term corresponds. Then, I iterated over the entire corpus, and used these dictionaries to convert each list of strings across this project's whole corpus in to lists of integers where each unique string in the original lists have unique integers that they are associated with.

Next, the integer version of our clue corpus was padded using zeros such that each each new clue vector of integer was the same length the longest clue in the dataset. This maximum clue length happened to be 43. The numeric list representation of this project's corpus was then split in the same manner described in Section 4.3.3, using the same value for the *random_state* parameter so that the splits are consistent across the models for clue-fill mapping that are examined in this project. Once split, these lists of integers representing the clue-answer dataset were converted into NumPy vectors, and then PyTorch tensors using *torch.from_numpy()* so that they can be used by the RNN

clue-fill mapping model. Once converted into PyTorch tensors, the data is wrapped in PyTorch TensorDataset objects, and then passed on to DataLoader for ease of access during the RNNs training and testing.

With the training data now in a form that can be directly handed over to a PyTorch RNN, the next important matter to be discussed is the RNN's overall design. This project utilizes a pretty straightforward RNN architecture with the hopes of investigating how a more or less standard RNN performs in the uniquely specialized context of CWPs and the problem of arriving at a useful clue-fill mapping. As such, the network features a single hidden layer that is initialized randomly using *torch.rand()*. In the network's foward function, the random initial hidden state is then passed to a two layer RNN created using the *torch.nn.RNN()* function. The outputs of this RNN layer are then passed to a fully connected output layer and then returned. The network's hidden layers are 100 dimensional. The network takes 43 dimensional input, and returns a vector the length of the clue vocabulary dictionary representing a probability distibution, which is then maximized at test time using *torch.max()* in order to find the fill deemed most likely by the network from within the space of possible fills defined by our training set.

The RNN described above is trained using the Adam optimizer and Cross-Entropy Loss. At test time, the RNN model is evaluated using the same scikit-learn metrics, as described in Section 4.3.3. These being f1-score, precision, accuracy, 'length accuracy', and recall. The results of these tests are shown in Chapter 5, and the full code for this RNN implementation can be found in Appendix C.

5

Results

5.1 KNN: 50

Number of Neighbors Examined: 3

Accuracy	Precision	Recall	F1	Length Accuracy
0.277	0.211	0.091	0.076	0.444

Table 5.1.1: KNN classifer scores over the test set using clue vectors of size 50 and where K = 3.

Number of Neighbors Examined: 5

Accuracy	Precision	Recall	F1	Length Accuracy
0.255	0.214	0.079	0.066	0.429

Table 5.1.2: KNN classifer scores over the test set using clue vectors of size 50 and where K = 5.

5.2 KNN: 100

Number of Neighbors Examined: 3

Accuracy	Precision	Recall	F1	Length Accuracy
0.277	0.212	0.091	0.076	0.444

Table 5.2.1: KNN classifer scores over the test set using clue vectors of size 100 and where K = 3.

Number of Neighbors Examined: 5

Accuracy	Precision	Recall	F1	Length Accuracy
0.253	0.215	0.078	0.066	0.428

Table 5.2.2: KNN classifer scores over the test set using clue vectors of size 100 and where K = 5.

5.3 KNN: 300

Number of Neighbors Examined: 3

Accuracy	Precision	Recall	F1	Length Accuracy
0.277	0.212	0.091	0.076	0.444

Table 5.3.1: KNN classifer scores over the test set using clue vectors of size 300 and where K = 3.

Number of Neighbors Examined: 5

Accuracy	Precision	Recall	F1	Length Accuracy
0.253	0.215	0.077	0.066	0.428

Table 5.3.2: KNN classifer scores over the test set using clue vectors of size 300 and where K = 5.

5.4 RNN

Untrained (random):

Accuracy	Precision	Recall	F1	Length Accuracy
$7.769e^{-6}$	$1.141e^{-6}$	$2.062e^{-5}$	$2.279e^{-8}$	0.053

Table 5.4.1: RNN implementation's performance in classification before any training is done on the model.

After two epochs of training:

Accuracy	Precision	Recall	F1	Length Accuracy
0.001	$4.960e^{-7}$	$2.656e^{-5}$	$9.228e^{-7}$	0.245

Table 5.4.2: RNN implementation's performance in classification once two training epochs (i.e., full passes over the training set) are complete.

6

Analysis

6.1 KNN

The KNN implementation of a clue-fill mapping was modestly effective at predicting fills that matched length of the target answer across the testing segment of this project's dataset, scoring a length accuracy of 0.444 in all versions of the classifier tested that used a K value of 3 (i.e., examined considered the class labels of the three nearest neighbors to the test point from training). This score indicates that the KNN classifer was able to predict fills that matched the length of the target answer associated with any individual test clue 44% of the time, after generating a prediction for each clue in the test set and comparing that predictions length with the length of the true answer for the clue at hand. Similarly, the KNN classifier scored a length accuracy of either 0.429 (for size 50 clue embeddings) or 0.428 (for sizes 100 and 300) when the K value used was 5, which means that the classifier predicted possible fills that would fit in the CWP in terms of the length constraint approximately 43% of the time. The consistency in these scores across the three

different sizes of clue embeddings used by the classifier seems to show that the dimension of the embedding vectors generated using Word2Vec has little impact on the classifiers ability to arrive at fills that are the desired length for the puzzle in question. The slight decline (approximately 1%) in performance as far as length accuracy is concerned between the classifiers that considered the 3 nearest neighbors, and those that considered the 5 nearest neighbors for each test point shows that the classifier performs marginally worse in the context of this problem when it takes more neighbors from training into account. The discrepancy between these results seems to point toward the fact that the classifier may be just barely more likely to have its predictions influenced by a points from the training set that are irrelevant in the context of the test clues that are being used to generate the predictions.

The KNN classifer performs significantly worse across the board in terms of its accuracy score than it did with regard to length accuracy. In this context, the accuracy score is a measure of the rate at which the different variations of the KNN classifier applied to generating a clue-fill mapping were able to predict a fill that matched the true answer exactly, across the whole test set. The classifiers that used a K value of 3 all scored 0.277 in accuracy regardless of the sizes of the clue embeddings used by the classifiers. This means that for 50, 100, and 300 dimensional clue embeddings the KNN classifiers that considered 3 nearest neighbors were able to correctly predict an answer based on a given clue as well as the labels of the three nearest points (clue vectors) in the training set. When the K value passed to the classifiers was 5, and so the 5 nearest neighbors were considered, the KNN classifiers scored either 0.255 (for size 50 clue vectors) in accuracy, or 0.253 (for both sizes 100 and 300). The difference between these two scores is slight, and these scores start to seem to point towards a pattern in the performance of these classifiers, which is that the size of the clue vectors used by the classifiers does not seem to matter generally in the classifiers ability to arrive at correct fills.

Indeed, in examining all of the different kinds of scores produced by the different KNN classifier implementations that are tested in this project, located in Chapter 5, a clear pattern is consistently demonstrated. To put it simply, this pattern is characterized by the fact that there is very little variation between any of the different scores produced by the KNN classifers that are examined in this project on the basis of the dimension of the clue embedding relied upon by the classifier. Classifiers using 50, 100, and 300 dimensional clue vectors produced almost identical scores in all of accuracy, precision, f1 score, and length accuracy across the test set. This is a somewhat unexpected result since larger word embeddings are capable of storing more information, and representing context in a more detailed manner, it seems like the classifiers that are able to draw on larger clue vectors should perform noticeably better. Nonetheless, the pattern seems to support one of two conclusions that could be drawn about the word vectors as well as this projects dataset that might explain the unexpected result. The first of these is that perhaps all of the clue vectors derived from the puzzles contained in this project's corpus are able to fully represent the meaning of a particular clue within the space of 50 dimensions. This would mean that there is in a sense no need to create clue embeddings that are any larger than 50 dimensions, since in this case the size 50 Word2Vec clue embeddings trained for and used by this project would be more than adequately sized. Alternatively, this pattern among the results produced by the different classifier implementations could be a consequence of the fact that there is simply very little difference between the clue representations produced by the three sizes of clue vectors that are examined in this project, even though their may be some noticeable difference in performance if larger or smaller clue vector sizes were also taken into consideration. It does not appear as though a definite conclusion can be made between these two potential explanations, as both could seemingly feasibly account for the results observed throughout this project's experiments.

6.2 RNN

The RNN based attempt at generating a clue-fill mapping for use as part of a CWP solving computer model was on the whole much less effective than the mapping which relied on KNN, whose performance was analyzed in Section 6.1. Across all of the metrics recorded, the RNN based mapping implementation scored much worse than all of the implementations based on KNN. This is true both in the case of the untrained RNN, which should simply approximate a random selection from among the answers in the training set, as well as in the case of the RNN implementation which was trained for two complete epochs before being tested. This outcome indicates that the RNN as it is implemented by this project is unable to learn the sort of clue-fill mapping that this project sought to find.

This result could conceivably be the product of many different factors, and so it is hard to say with absolute certainty why exactly the network appears to be unable to learn and which factors exactly are to blame. With that being said, one possible reason the RNN may have been unable to learn some association between the clues in the training set and the answers to which those clues correspond could be that the RNN simply did not have enough time to train. Due to the time limitations of this project as well as the size of the training set, the RNN implementation was only allowed to complete two full passes over the training data before being tested, and so it is possible that if it was given more runs through the training data, the model's performance would have eventually started to improve. This explanation seems unlikely, however since in unrecorded training phases that were lost due to a program crash at test time which extended up to 10 epochs, training loss never seemed to consistently trend downward, and would instead slowly decrease for some time before suddenly spiking back up to around where it began. With the 0.00001 learning rate value used in the RNN clue-fill mapping experiments whose results are shown in Section 5.4, training loss values ranged from around 13 on the high

end, and 7 on the low end. In the experiments that were lost due to a crash during test time in which the RNN was allowed to train for 10 epochs, training loss seemed to range from around 5 to 13. In neither case however, did the training loss continually trend downward. It is also worth noting that these training loss values would be considered very high, which indicates that the model is struggling to learn the problem.

That being said, despite the consistently high training loss just described, the RNN mapping did seem to improve to a certain extent after completing two full epochs of training. This improvement can be observed in the noticeably increased scores both in terms of accuracy, and length accuracy produced by the trained RNN over the test set. This result may suggest that if allowed more training epochs the model could eventually arrive at a clue-fill mapping that yields accuracy more comparable with that produced by the KNN based clue-fill maps.

7

Conclusion

The KNN classifier based clue-fill mappings that this project implements are clearly more successful overall than the RNN based ones. This is apparent from the fact that each of the KNN based mappings seemed to perform better than the two RNN based ones that this project explores. However, neither approach is without its drawbacks. For instance, the RNN takes much longer to train than KNN, but KNN takes much longer at test time.

Both kinds of classifiers could more than likely be applied to the problem of generating a clue-fill mapping in more effective ways by way of further honing the implementations of each classifier in the context of the clue-fill mapping aspect of the crossword solving problem. In terms of the KNN classifier, this could potentially be accomplished by finding a more useful kind of clue embedding that is able to better represent the clues used to train the classifier. This type of embedding seems like it could lead to improved results for the classifier at testing. Whether finding this kind of embedding means abandoning Word2Vec as method of generating clue embeddings, or simply making better use of

Word2Vec in the creation of vector representations of the clues in this projects dataset, stronger clue embeddings have to potential to improve the classifiers performance overall.

As far as the RNN classifier for clue-fill mapping is concerned, perhaps the model as currently constructed could eventually learn a more effective mapping if given more time to train. A more effective RNN variant could likely be developed for this application given RNNs success in some somewhat conceptually similar applications like sentiment analysis.

These kinds of clue-fill mapping algorithms could be used in future work involving computer models for solving CWPs. The data structures presented in this project could also be useful as a somewhat lightweight and user-friendly representation of CWPs that can be generated from ".puz" files in further work surrounding crossword solvers.

Appendix A

Code for Data Structures

```
import numpy as np
import pandas
import pandas as pd
import puz
import codecs
import os

class PuzRep:  # Class of objects representing puzzles from dataset.
    def __init__(self, puzl_grid_rep, puzl_dict, puzl_dims):
        self.p_gr = puzl_grid_rep  # Lists representing puzzle grid.
        self.p_dc = puzl_dict  # Dictionaries assiciated with puzzle.
        self.dims = puzl_dims  # Size of puzzle. All of these must be
   given to the object in the form they are returned
                               # by get_puz_info function.

    def to_str(self) -> str:  # Returns all grid info as a string.
```

```
16          s = 'Grid: '
17          for row in self.p_gr:
18              s += ('\n' + row)
19          return s
20
21      def dump_grid_rep(self):  # Returns list rep of puzzle grid.
22          return self.p_gr
23
24      def print_puz(self, *args):  # Print all info for current state of
    the puzzle if *args is empty.
25          if len(args) == 0:       # Can pass 'dict' or 'grid' to specify
    what's returned.
26              print('Grid: ')
27              for row in self.p_gr:
28                  print(row)
29              print(f'Dictionary: \n{self.p_dc}')
30          elif args[0] == 'grid' and len(args) == 1:
31              print('Grid: ')
32              for row in self.p_gr:
33                  print(row)
34          elif args[0] == 'dict' and len(args) == 1:
35              print(f'Dictionary: \n{self.p_dc}')
36          elif len(args) > 1:
37              print('Grid: ')
38              for row in self.p_gr:
39                  print(row)
40              print(f'Dictionary: \n{self.p_dc}')
41
42      def fill_cell(self, character, cell):  # Method for filling a
    specified cell on the puzzle grid with some char.
43          grid = self.p_gr
44          target_col = (cell % self.dims[0]) * 4  # ie. cell % puzzle
    width
```

```
45          target_row = ((cell // self.dims[0]) * 2) + 1  # floor division
46          grid[target_row] = grid[target_row][:target_col] + character +
    grid[target_row][target_col + 1:]
47          self.p_gr = grid  # Store updated grid as attribute of the
    PuzRep object.
48
49      def clear_cell(self, cell):  # Clear specified cell. (aka fill it
    with '-')
50          self.fill_cell('-', cell)
51
52      def fill_word(self, mode_str, num, word):  # Write a given word onto
     puzzle grid based on its clue number.
53          if mode_str == 'across':  # Mode_str is either 'across' or 'down
    '.
54              start_cell = self.p_dc['a'][num]['cell']
55              word_length = self.p_dc['a'][num]['len']
56              position = start_cell
57              tracker = 0
58              while (position + 1) <= (start_cell + word_length):
59                  self.fill_cell(word[tracker], position)
60                  tracker += 1
61                  position += 1
62          elif mode_str == 'down':
63              start_cell = self.p_dc['d'][num]['cell']
64              word_length = self.p_dc['d'][num]['len']
65              position = start_cell
66              tracker = 0
67              while position < start_cell + (word_length * self.dims[0]):
68                  self.fill_cell(word[tracker], position)
69                  tracker += 1
70                  position += self.dims[0]
71          else:
```

```
72              print('error bad mode string given: please provide fill mode
    \'across\'/\'down\' ')
73
74      def clear_word(self, mode_str, num):  # Fill a word with '-' symbols
    to the correct length.
75          if mode_str == 'across':
76              word_length = self.p_dc['a'][num]['len']
77              in_word = '-' * word_length
78              self.fill_word(mode_str, num, in_word)
79          elif mode_str == 'down':
80              word_length = self.p_dc['d'][num]['len']
81              in_word = '-' * word_length
82              self.fill_word(mode_str, num, in_word)
83
84      def reveal_solutions(self):  # Fill in all true clue solutions for a
    given puz. Use with print to see solved puzzle.
85          for number in self.p_dc['a']:
86              self.fill_word('across', number, self.p_dc['a-answers'][
   number].lower())
87          for number in self.p_dc['d']:
88              self.fill_word('down', number, self.p_dc['d-answers'][number
   ].lower())
89
90
91 ##HELPER FUNCTIONS FOR PUZREP
92 def str_insert_n(str_in, n, index):  # Helper that inserts n spaces at
   specified index in given string.
93      str_in = str_in[:index] + (' ' * n) + str_in[index:]
94      return str_in
95
96
97 def cut_end_spaces(str_in):  # Simple function to remove spaces from the
    end of a given string.
```

```
98      b = True
99      while b:
100         try:
101             if str_in[len(str_in) - 1] == ' ':
102                 str_in = str_in[:-1]
103             else:
104                 b = False
105         except:
106             b = False
107     return str_in
108 
109 
110 # Main helper function that retrieves puzzle information from a PuzPy
        puzzle object and converts it to a form which can
111 # be passed on to a PuzRep Object
112 def get_puz_info(puz_in):  # Returns x, y, z (grid, dict, dims) for
        given puzpy object.
113     clue_cells = {}  # Dict. used to keep track of the starting cells
        associated with each clue.
114     numbering = puz_in.clue_numbering()
115     a = numbering.across
116     d = numbering.down
117     inner_dict_a = {}  # Initialize empty dictionaries to store puzzle
        info.
118     inner_dict_d = {}  # Separated by across and down.
119     inner_a_ans = {}
120     inner_d_ans = {}
121     outer_d = {}
122 
123     for item in a:  # Iterates over every clue in a given puzzle, saving
        all info to dicts.
124         clue_cells[item['cell']] = item['num']  # For aligning unicode
        numbering.
```

```
125         inner_dict_a[item['num']] = item
126         inner_a_ans[item['num']] = ''.join(puz_in.solution[item['cell']
    + i] for i in range(item['len']))
127     for item in d:
128         clue_cells[item['cell']] = item['num']
129         inner_dict_d[item['num']] = item
130         inner_d_ans[item['num']] = ''.join(puz_in.solution[item['cell']
    + i * numbering.width] for i in
131                                            range(item['len']))
132 
133     outer_d['a'] = inner_dict_a
134     outer_d['a-answers'] = inner_a_ans
135     outer_d['d'] = inner_dict_d
136     outer_d['d-answers'] = inner_d_ans
137     out_list = []
138     height = puz_in.height
139     width = puz_in.width
140     for row in range(height):  # Print grid.
141         cell = row * width
142         line_string = ''
143         g_st = '   '.join(puz_in.fill[cell:cell + width])  # Space out
    the grid rep to make room for numbering.
144         cell_list = []
145         for spot in range(width):
146             cell += spot
147             if cell in clue_cells.keys():
148                 if 0 < clue_cells[cell] < 10:
149                     line_string += ((get_unicode(clue_cells[cell])) + '
      ')
150                 elif 9 < clue_cells[cell] < 100:
151                     line_string += ((get_unicode(clue_cells[cell])) + '
     ')
152                 elif 99 < clue_cells[cell] < 300:
```

```
153                     line_string += ((get_unicode(clue_cells[cell])) + '
    ')
154                 else:
155                     print('error puz too big')
156                 cell_list.append(cell % width)
157             cell = row * width
158         line_string = cut_end_spaces(line_string)
159         for dex, thing in enumerate(cell_list):
160             if dex == 0 and thing != 0:  # If first num in c_l not 0,
    prepend 4 spaces for each cell of difference.
161                 line_string = str_insert_n(line_string, 4*thing, 0)
162             elif dex == 0:
163                 pass  # Otherwise if dex = 0, leave first num in place
164             else:
165                 if dex < len(cell_list) - 1:  # If not at end of
    cell_list.
166                     delta = thing - cell_list[dex - 1]
167                     if delta > 1:
168                         in_point = ((cell_list[dex - 1]) * 4) + 4
169                         line_string = str_insert_n(line_string, (delta -
     1) * 4, in_point)
170                 else:
171                     delta = cell_list[dex] - cell_list[dex - 1]
172                     if delta > 1:
173                         in_point2 = ((cell_list[dex - 1]) * 4) + 4
174                         line_string = str_insert_n(line_string, (delta -
    1) * 4, in_point2)
175         out_list.append(line_string)
176         out_list.append(g_st)
177     dims = (width, height)
178     return out_list, outer_d, dims
179
180
```

```
181 def get_unicode(num):  # Helper function that converts a specified
    number into unicode superscript for the printing of
182     if num < 0:         # puzzles to terminal.
183         print("error: negative digits should not be superscripted")
184         return str(0)
185     elif num == 0:
186         return codecs.decode(r'\u2070'.format(num), 'unicode_escape')
187     elif num == 1:
188         return codecs.decode(r'\u00b{0}'.format(num+8), 'unicode_escape
    ')
189     elif num == 2 or num == 3:
190         return codecs.decode(r'\u00b{0}'.format(num), 'unicode_escape')
191     elif 3 < num < 10:
192         return codecs.decode(r'\u207{0}'.format(num), 'unicode_escape')
193     elif 9 < num < 200:
194         st = ''
195         for ch in str(num):
196             st += (get_unicode(int(ch)))
197         return st
198     else:
199         print("error: outside range for superscript")
200         return str(0)
```

Appendix B

Code for KNN

```
import csv
import numpy as np
import pandas
import pandas as pd
import puz
import codecs
import os
from sklearn.model_selection import train_test_split
from nltk.tokenize import RegexpTokenizer
from scipy import stats
from gensim.models import Word2Vec
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, f1_score, precision_score,
    recall_score
from gensim.models import Word2Vec
from nltk.corpus import brown
```

```
16
17 ext = '.puz'
18 all_data = []
19 all_file_names = []
20 mini_data = []
21 nyt_data = []
22 mini_file_names = []
23 nyt_file_names = []
24 print('Scanning directories...')  # Read in and parse full set of
   puzzles from wherever they are stored.
25 dir_name = 'C:/Users/finnf/Documents/sproj_4/nyt_minis_8
   -21-2013--12-1-2023' #NY Times Minis
26 for files in os.scandir(dir_name):
27     if files.path.endswith(ext):
28         xyz = get_puz_info(puz.read(files))  # Parse with PuzPY and then
    run get_puz_info.
29         all_data.append(PuzRep(xyz[0], xyz[1], xyz[2]))  # Hand info
   over to PuzRep objects.
30         all_file_names.append(files)  # Then save those PuzRep objects
   for later use.
31 dir_name = 'C:/Users/finnf/Documents/sproj_4/nyt-xword_11
   -21-1993--12-1-2023' #NY Times
32 for files in os.scandir(dir_name):
33     if files.path.endswith(ext):
34         xyz = get_puz_info(puz.read(files))
35         all_data.append(PuzRep(xyz[0], xyz[1], xyz[2]))
36         all_file_names.append(files)
37 dir_name = 'C:/Users/finnf/Documents/sproj_4/nyer_4-30-2018--12-1-2023'
   #New Yorker
38 for files in os.scandir(dir_name):
39     if files.path.endswith(ext):
40         xyz = get_puz_info(puz.read(files))
41         all_data.append(PuzRep(xyz[0], xyz[1], xyz[2]))
```

```
42          all_file_names.append(files)
43 dir_name = 'C:/Users/finnf/Documents/sproj_4/lat_9-1-2021--12-1-2023' #
       LA Times
44 for files in os.scandir(dir_name):
45      if files.path.endswith(ext):
46          xyz = get_puz_info(puz.read(files))
47          all_data.append(PuzRep(xyz[0], xyz[1], xyz[2]))
48          all_file_names.append(files)
49 dir_name = 'C:/Users/finnf/Documents/sproj_4/nd_6-4-2017--12-1-2023' #
       Newsday
50 for files in os.scandir(dir_name):
51      if files.path.endswith(ext):
52          xyz = get_puz_info(puz.read(files))
53          all_data.append(PuzRep(xyz[0], xyz[1], xyz[2]))
54          all_file_names.append(files)
55
56
57 print('Building X,Y...')  # i.e., Take the puzzles now stored in PuzRep
       objects and extract their
58 x, y = [], []               # clues (x) / answers (y).
59 x_vocab, y_vocab = {}, {}  # Key: unique token (word, space delimited)
       in x / y. -> val: token freq
60 init_count = 0
61 for puzzle in all_data:
62      for item in puzzle.p_dc['a'].values():  # Load clues and answers to
       x, y for all puzzles in all_data.
63          tokenizer = RegexpTokenizer(r'\w+')
64          x_in = tokenizer.tokenize(item['clue'].lower())
65          if len(x_in) == 0:
66              init_count += 1
67          x.append(x_in)  # All lowercase, tokenized.
68          for word in x_in:
69              if word not in x_vocab.keys():
```

```
                x_vocab[word] = 1
            else:
                x_vocab[word] += 1
        y_in = puzzle.p_dc['a-answers'][item['num']].lower()
        y.append(y_in)
        if y_in not in y_vocab.keys():
            y_vocab[y_in] = 1
        else:
            y_vocab[y_in] += 1
    for item in puzzle.p_dc['d'].values():  # Repeat for down clues and
    their answers.
        tokenizer = RegexpTokenizer(r'\w+')
        x_in = tokenizer.tokenize(item['clue'].lower())
        if len(x_in) == 0:
            init_count += 1
        x.append(x_in)  # All lowercase, tokenized.
        for word in x_in:
            if word not in x_vocab.keys():
                x_vocab[word] = 1
            else:
                x_vocab[word] += 1
        y_in = puzzle.p_dc['d-answers'][item['num']].lower()
        y.append(y_in)
        if y_in not in y_vocab.keys():
            y_vocab[y_in] = 1
        else:
            y_vocab[y_in] += 1

# Train Clue Embeddings, size 50.
if __name__ == '__main__':
    train_set = brown.sents()
    ls = []
   # [token.lower() for token in train_set]
```

```
    for item in train_set:
        sentence = [w.lower() for w in item]
        ls.append(sentence)
    for item in x:
        ls.append(item)
    for item in y:
        yls = []
        yls.append(item)
        ls.append(yls)

print('Starting Vectorization...')
x_vec_freq, y_vec_freq = {}, {}  # Used for comparison and debugging classifiers.
embed_model = Word2Vec.load('newVecsXandY_50.model')  # Load embeddings
dimension = 50
xv, yv = [], []  # Lists of np vectors representing clues, answers respectively.

word_count = 0
err_count = 0
error_indexes = {}
cluvec_2_clue = {}
for i, clue in enumerate(x):
    clue_vec = np.zeros(dimension, dtype=np.float32)
    for word in clue:
        clue_vec = np.add(clue_vec, embed_model.wv[word])
        word_count += 0
    divisor = len(clue)
    if divisor == 0:
        error_indexes[i] = i
        err_count += 1
    else:
        clue_vec = np.divide(clue_vec, divisor)
```

```
133         cluvec_2_clue[tuple(clue_vec)] = clue
134         xv.append(clue_vec)
135
136 index_2_answer = {}
137 answer_2_index = {}
138
139 for j, ans in enumerate(y):
140     if j not in error_indexes:
141         if ans not in index_2_answer.values():
142             y_in = j
143             yv.append(y_in)
144             index_2_answer[j] = ans
145             answer_2_index[ans] = j
146         else:
147             y_in = answer_2_index[ans]
148             yv.append(y_in)
149
150 print("Vectorization done.")
151 print('Doing data splits.')
152 x_train, x_test, y_train, y_test = train_test_split(xv, yv, test_size
    =0.2)  # Do data splits train / test here.
153 x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
    test_size=0.2)  # and then train / val here.
154 print("Doing df conversions.")
155 x_train_df = pd.DataFrame(x_train)  # First row is first clue in train
    etc (vector rep of 1st clue)
156 y_train_df = pd.DataFrame(y_train) #Same as ^^ but answer instead of
    clue.
157 neigh = KNeighborsClassifier(n_neighbors=3)
158 neigh.fit(x_train_df, y_train_df)
159 prediction = neigh.predict([x_test[0]])
160 y_pred_set = []
161
```

```
print('Predicting...')
for k, clue in enumerate(x_test):
    pred = neigh.predict([clue])
    if k % 1000 == 0:
        print(k, pred[0])
    y_pred_set.append(pred[0])
print('Built prediction set.')
print('Accuracy score: ', accuracy_score(y_test, y_pred_set))
c = 0
c2 = 0
c3 = 0
for l, item in enumerate(y_test):
    if item == y_pred_set[l]:
        c += 1
    if index_2_answer[item] == index_2_answer[y_pred_set[l]]:
        c2 += 1
    if len(index_2_answer[item]) == len(index_2_answer[y_pred_set[l]]):
        c3 += 1

acc3 = c3 / len(y_test)  # Calculate 'length accuracy'.
print('Length Accuracy: ', acc3)  # Print metrics.
print('F1: ', f1_score(y_test, y_pred_set, average="macro",
    zero_division=np.nan))
print('Precision: ', precision_score(y_test, y_pred_set, average="macro
    ", zero_division=np.nan))
print('Recall: ', recall_score(y_test, y_pred_set, average="macro",
    zero_division=np.nan))
```

Appendix C

Code for RNN

```
import numpy as np
import pandas as pd
import puz
import codecs
import os
from sklearn.model_selection import train_test_split
from nltk.tokenize import RegexpTokenizer
from sklearn.metrics import accuracy_score, f1_score, precision_score,
    recall_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

print('BUILDING X,Y...')
x, y = [], []  # x contains clues, y contains answers.
```

```
x_vocab_freq, y_vocab_freq = {}, {}  # Key: unique token (word, space delimited) in x / y. -> val: token freq

# Build x, y, and vocabs.
init_count = 0
for puzzle in all_data:  # Get clues/ans's with target lengths.
    for item in puzzle.p_dc['a'].values():  # Load clues and answers to x, y for all puzzles in all_data.
        tokenizer = RegexpTokenizer(r'\w+')
        x_in = []
        x_in.extend(tokenizer.tokenize(item['clue'].lower()))
        if len(x_in) == 1:
            init_count += 1
        x.append(x_in) # All lowercase, tokenized.
        raw_x = x_in
        for word in raw_x:
            if word not in x_vocab_freq.keys():
                x_vocab_freq[word] = 1
            else:
                x_vocab_freq[word] += 1
        y_in = puzzle.p_dc['a-answers'][item['num']].lower()
        y.append(y_in)
        if y_in not in y_vocab_freq.keys():
            y_vocab_freq[y_in] = 1
        else:
            y_vocab_freq[y_in] += 1
    for item in puzzle.p_dc['d'].values():  # Load clues and answers to x, y for all puzzles in all_data>
        tokenizer = RegexpTokenizer(r'\w+')
        x_in = []
        x_in.extend(tokenizer.tokenize(item['clue'].lower()))
        if len(x_in) == 1:
            init_count += 1
```

```
        x.append(x_in)  # All lowercase, tokenized.
        raw_x = x_in
        for word in raw_x:
            if word not in x_vocab_freq.keys():
                x_vocab_freq[word] = 1
            else:
                x_vocab_freq[word] += 1
        y_in = puzzle.p_dc['d-answers'][item['num']].lower()
        y.append(y_in)
        if y_in not in y_vocab_freq.keys():
            y_vocab_freq[y_in] = 1
        else:
            y_vocab_freq[y_in] += 1

# Create dictionaries for simple counting implementation of integer
    encoding.
x_to_int, y_to_int = {}, {}
c = 2
for token in x_vocab_freq.keys():
    if c == 2:
        x_to_int[token] = 2
    elif token not in x_to_int.keys():
        x_to_int[token] = c
    c += 1
c2 = 1
for token in y_vocab_freq.keys():
    if c2 == 1:
        y_to_int[token] = 1
    elif token not in y_to_int.keys():
        y_to_int[token] = c2
    c2 += 1
int_to_y = {v: k for k, v in y_to_int.items()}  # Reverse the answer to
    integer dictionary for later use.
```

```
77 max_len = 0
78 max_target_len = 0
79 x_as_ints, y_as_ints = [], []
80 x_lens = []
81 for item in x:
82     x_int_line = []
83     if len(item) > max_len:
84         max_len = len(item)
85     for tok in item:
86         x_int_line.append(x_to_int[tok])
87     x_as_ints.append(x_int_line)
88 
89 for item in y:
90     y_as_ints.append(np.array([y_to_int[item]]))
91 
92 if 1 in x_to_int.values():
93     print(True)
94 else:
95     print(False)
96 
97 padded_x = []
98 for item in x_as_ints:  # Pad clues so that each clue tensor will be of
       the same length.
99     row_out = item
100    difference = 0
101    if len(item) < max_len:
102        difference = max_len - len(item)
103    while difference > 0:
104        row_out.append(0)
105        difference -= 1
106    padded_x.append(row_out)
107 
```

```
np_p_x = np.array(padded_x)  # Convert padded integer clues and integer answers to numpy arrays.
np_y_int = np.array(y_as_ints)
seq_length = 43
# Do splits on np array version of data.
x_train, x_test, y_train, y_test = train_test_split(np_p_x, np_y_int, test_size=0.2, random_state=42)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)


class NewRNN(nn.Module):  # Network for clue-fill mapping.
    def __init__(self, input_size, hid_size, num_lay, num_class):
        super(NewRNN, self).__init__()
        self.hid_size = hid_size
        self.num_lay = num_lay
        self.rnn = nn.RNN(input_size, hid_size, num_lay, batch_first=True)
        self.fc = nn.Linear(hid_size, num_class)

    def forward(self, x_input):
        h0 = torch.rand(x_input.size(0)*self.num_lay, self.hid_size)
        out, h = self.rnn(x_input, h0)
        out = self.fc(out)
        return out


device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
feature_size = 43
n_layer = 2
h_size = 100
batch_size = 1
learn_rate = 0.00001
```

```
137 n_class = len(y_vocab_freq)
138 model = NewRNN(feature_size, h_size, n_layer, n_class)  # Creates the
    model.
139 criterion = nn.CrossEntropyLoss()
140 optimizer = optim.Adam(model.parameters(), lr=learn_rate)
141
142 # Converts to datasets and dataloaders.
143
144 train_data = TensorDataset(torch.from_numpy(x_train), torch.from_numpy(
    y_train))
145 valid_data = TensorDataset(torch.from_numpy(x_val), torch.from_numpy(
    y_val))
146 test_data = TensorDataset(torch.from_numpy(x_test), torch.from_numpy(
    y_test))
147 train_loader = DataLoader(train_data, shuffle=False, batch_size=
    batch_size)
148 valid_loader = DataLoader(valid_data, shuffle=False, batch_size=
    batch_size)
149 test_loader = DataLoader(test_data, shuffle=False, batch_size=batch_size
    )
150
151 print('Done converting to loaders.')
152 print('Training RNN...')
153 # RNN training loop.
154 n_total_steps = len(train_loader)
155 num_epochs = 1
156 for epoch in range(num_epochs):
157     for i, (clues, labels) in enumerate(train_loader):  # Iterate over
    each clue label pair in the data set.
158         clues = clues.to(torch.float32)
159         labels = labels.item()
160         labels = torch.LongTensor([labels])
161         outputs = model(clues)
```

```
162         loss = criterion(outputs, labels)  # Defines loss function.
163         optimizer.zero_grad()
164         loss.backward()
165         optimizer.step()
166 
167         if (i + 1) % 100 == 0:  # Print train loss.
168             print(f'Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1}/{
    n_total_steps}], Loss: {loss.item():.4f}')
169 
170 # Testing the model.
171 y_pred_set = []
172 with torch.no_grad():
173     n_correct = 0
174     n_samples = 0
175     for i, (clues, labels) in enumerate(test_loader):  # For each item
    in test set, predict label using model and then
176         clues = torch.tensor(clues, dtype=torch.float32)  # save the
    result to a list for calculating scores.
177         labels = labels.to(device)
178         outputs = model(clues)
179         _, predicted = torch.max(outputs.data, 1)
180         y_pred_set.append(predicted.item())
181         n_samples += labels.size(0)
182         if predicted.item() == labels.item():
183             n_correct += 1
184         if i % 10000 == 0:
185             print('n_corr: ', n_correct)
186             print('i: ', i)
187     acc = 100.0 * n_correct / n_samples
188 
189 # Print scores.
190 print('f1: ', f1_score(y_test, y_pred_set, average="macro",
    zero_division=0))
```

```
191 print('precision: ', precision_score(y_test, y_pred_set, average="macro
    ", zero_division=0))
192 print('recall: ', recall_score(y_test, y_pred_set, average="macro",
    zero_division=0))
193 print('accuracy score: ', accuracy_score(y_test, y_pred_set))
194
195 c = 0
196 c2 = 0
197 c3 = 0
198 for l, item in enumerate(y_test):  # Retest for length accuracy and
    verification.
199     if item == y_pred_set[l]:
200         c += 1
201     if len(int_to_y[item[0]]) == len(int_to_y[y_pred_set[l]]):  #
    Calculate 'length accuracy' over the tes set.
202         c2 += 1
203 acc2 = c2 / len(y_test)
204 print('acc2: ', acc2) # Print LA.
```

Appendix D

Photo sources

Sources:

1) Link to Figure 1.0.1 host.

2) Link to Figure 1.0.2 host.

Bibliography

[1] Lihan Chen et al., *Crossword Puzzle Resolution via Monte Carlo Tree Search* (2022).

[2] Michael L. Littman et al., *A probabilistic approach to solving crossword puzzles*, Artificial Intelligence **134** (2002), 23-55.

[3] Daniel Jurafsky and James H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, Stanford University (2024).

[4] Josh Myer and Evan Martin and Chris Casinghino and Michael Greenberg, *Puz - FileFormat.wiki*, FileFormat.wiki, Google Code.

[5] Matthew L. Ginsberg, *Dr.Fill: Crosswords and an Implemented Solver for Singly Weighted CSPs*, CoRR (2014).

[6] Tomas Mikolov and Kai Chen and Greg Corrado and Jeffrey Dean, *Efficient Estimation of Word Representations in Vector Space* (2013).

[7] Parker Higgins, *xword-dl*, GitHub (2022).

[8] Alex Dejarnatt, *puzpy*, GitHub (2010).