FLIGHT a light weight implementation of Dr. Fill

Seamus Gould
Vassar College / 124 Raymond Avenue
sgould@vassar.edu

Rowan Roshong
Vassar College / 124 Raymond Avenue
rroshong@vassar.edu

Abstract

We describe an approach to solve popular crossword puzzles loosely based on Matt Ginsberg's Dr. Fill. We utilize a corpus of six million crossword words and clues across dozens of publishers to construct a FastText model. We build a crossword solver that leverages this model to get candidate clues, apply regex queries, and either optimize using alpha-beta pruning or greedy algorithms. Like our predecessors, we treat this problem as a constraint satisfaction problem where we attempt to maximize a fill given the other words, and the candidate probabilities. We evaluate our solver on 92 puzzles, measuring accuracy and time taken. Overall, FLIGHT performed well, with the pruning method outperforming greedy selection. We believe that this implementation, though not as precise as Dr. Fill, it is competitive because it solves crosswords in significantly less time.

Introduction and Related Work

We hope to show an approach to the task of computer crossword solving that is both competitive and efficient in terms of computational power and memory. This paper would not be the first to accomplish such a feat. This task has been attempted numerous times. One of the early attempts is from PROVERBS, which was developed by Littman, Keim, and Shazeer in 2002 [Shazeer et al., 2000]. The authors develop a list of heuristics by looking at word matches, sentence similarity, and information retrieval systems. It was able to reach an accuracy of 98.1% for the characters in a time under fifteen minutes. The algorithms found in this paper appear in many other approaches. For example, this paper categorizes this task as a constraint satisfaction problem such that given a list of clues, PROVERBS selects the clues that maximize the probability of the board being a correct one. It uses a combination of thirty different modules, each used to tackle certain aspects of clues, like missing words and clues of writers.

Another implementation comes from WebCrow, which was made by Marco Ernandes, Giovanni Angelini, and Marco Gori [Ernandes et al., 2005]. This was a novel implementation that used the web to create a candidate of words that would be used to fill in the puzzle. Like PROVERBS, WebCrow uses a set of modules to make inference, but it introduces a web based inference. After getting a set of candidate clues, it fills the crossword in using a depth-first approach, which the authors found to be better suited for a constraint satisfaction problem. It performs fairly well by achieving competitive scores of around ninety percent given a fifteen minute time limit on Italian crosswords.

Lastly, the most recent and best performing algorithm is Dr. Fill, which won the 2021 ACPT tournament. Matt Ginsberg writes of Dr. Fill in his paper "Dr.Fill: Crosswords and an Implemented Solver for Singly Weighted CSPs." Though the paper that he released is from 2011, and the version of Dr. Fill had improved since then, Matt Ginsberg lays out the framework for the method of getting candidate words and writing them into the puzzle [Ginsberg, 2011]. He creates a series of heuristics, like similar to previous iterations of the problem, and then fills in the words by representing the problem as a soft constraint satisfaction problem. After reaching a high position for each tournament, year after year, Ginsberg had a breakthrough when he teamed up with the University of Berkeley to help him generate a set of candidate clues. After this, Dr. Fill won its first tournament. At the time that his paper was written, Dr. Fill had already ranked high enough to be in the top fifty in the world. Though Dr. Fill was able to run on a laptop, to be competitive with human solvers, it required immense memory and computational power. Though Dr. Fill eventually would go on to win the crossword championship, it needed two gpus and a lot of memory [Roeder, 2021]. [1].

[1]More details of this can be found here

In each of the previous tasks, almost all generated a list of candidate clues and then filled the puzzle. From our literature search, FLIGHT is the first to implement extreme text classification to generate a candidate list from clues, so our implementation is novel. We chose this implementation for a couple of reasons.

- The first reason is that extreme text classification algorithms shine when it comes to making fast inference. Given that we have a dataset that contains hundreds of thousands of classes (312,000) and millions of features, most algorithms can be eliminated from our selection because they take too long for inference or they require an excessive amount of memory. Our goal is to produce an algorithm that is both competitive and efficient in both computation and memory. Because of this, algorithms like support vector machines, naive bayes, and transformers would not be ideal because they take up time and memory [Kharbanda et al., 2021].
- Another reason that we decided on an extreme text classification was because we assumed that the number of out of vocabulary words would be minimal. Though it would be difficult to categorize every word in the English dictionary, the set of words that are in crosswords is minimal compared to the entire set of words in the English dictionary.
- By using a classification algorithm, we can assign probability distributions to each word, which would be helpful for filling in the board for later.

Though this task is fairly recent, there are already several algorithms that can be used for this task [Bhatia et al., 2016]. One of the best models for this task that is both light and fast is Slice+FastText. In all of the benchmarks, it consistently achieves fast training times and low memory usage. FastText was introduced by Facebook in 2016 to make a scalable model that was on par with the deep learning classifiers that was faster in terms of inference and training [Joulin et al., 2016][2]. For this case, a hierarchical softmax function reduces the training time for the model. This practice is recommended when classifying a large number of classes because the tine complexity of training the data goes from $O(kh)$ to $O(h \log_2 k)$ such that k is equivalent to the number of classes and h is the dimension of the dimension of the text representation.

Facebook produced FastText in 2016 as a model that could efficiently efficiently classify text achieving state of the art results in minimal time. In the paper introducing FastText, the authors highlight that as of the time, it was able to classify text with similar accuracies as the other state of the art models, but FastText was able to achieve this in minimal time. According to the authors of the paper, “We can train FastText on more than one billion words in less than ten minutes using a standard multicore CPU, and classify half a million sentences among 312K classes in less than a minute.”[Joulin et al., 2016].

FastText uses a linear classifier to categorize sentences much like a support vector machine. One of the improvements offered by this model is a hierarchical softmax loss function, which uses a Huffman Algorithm to speed up training.

Methods

Data

The training data for our model comes from a dataset containing dozens of crossword publishers and six million words and clues [3]. After shuffling the data, we split it into a train, test, and validation set with ratios of 99, .5, and .5 respectively. The intuition behind this is that the test and validation sets are consistent given that the corpus is massive. Each dataset had a size of 6396234, 32304, and 32304 respectively.

The crosswords that we ran our tests on came from the xl-word library. This dataset worked for us because they contained thousands of puzzles with the puzzle, clues, and solutions. This allowed us to test on many puzzles at a time and auto evaluate our solver’s performance.

Model

The model is based on the FastText repository[4]. We train the model to optimize for precision with the training data and the validation data using a hierarchical softmax loss function. FastText supports auto-tuning for us, so we did not manually tune the parameters. In addition, as one of the parameters,

[2]According to the original paper, it can be trained on more than a billion words in less than ten minutes using a standard CPU into hundreds of thousands of classes.

[3]The data

[4]The official repository for FastText

we give it a path to GloVe containing 300 dimensions. The only specification that we wanted to include was using hierarchical softmax to compute the loss, which made training faster.

After training our model and evaluating it, we find that it performs very well compared to open domain question and answering algorithms. T5, BART and Retrieval-Augmented Generation models perform well as question and answering datasets, but they do not perform compared to classification models. For example, the Retrieval-Augmented Generation model correctly identifies the correct word given a clue 24% of the time, but this pales in comparison to the 55% that was achieved by using FastText. See Table 1 for a summary of our results compared to other work.

Model	Top-1
T5-base	8.4
BART-large	13.8
RAG wiki	24.2
RAG dict	24.0
FastText	55.1

Table 1: Comparison of Models

The data found in Table 1 is collected from this benchmark for crossword solving using open domain question answering [submission, 2021].

Solving

We decided to solve our puzzle using two main techniques: A greedy approach and an alpha-beta pruning approach. For the greedy approach, these were the steps.

1. We feed all of the clues to our model and collect the most confident answers in an ordered list of tuples containing (index, direction, clue, probability).
2. Add the most confident answers to our board in order of probability, popping them off one by one.
3. In the event of a letter conflict, find the next best answer that fits the partial word Regex query. If there is no word that matches the Regex query, leave blank and move on.
4. Continue until the board cannot fill in any more squares or the list of tuples is empty.

For the alpha-beta pruning approach, these were the steps:

1. Create a list of the most confident clues and answers. We denote this confidence by the probability of the most likely word.
2. Solve a word and then add that word to the board based on the regex matching. If there are multiple possibilities, then copy the board for each word, and add the word to each of them.
3. Save each pair for the board and the probability of that word to a list, multiplying the probabilities.
4. As the puzzle gets filled in, the probabilities will be updated, and after either the board is completed or the list of words is null, then the highest probability board is returned.

In the alpha-beta pruning method, we apply an alpha value, $0 < \alpha \leq 1$, which controls our pruning rate. If α is closer to 1, there will be less pruning and FLIGHT will act in a more greedy manner, accepting fewer possible candidate words. If α is closer to 0, than FLIGHT will apply many more candidate words and will take longer to run. We hypothesize that lower α values will lead to higher accuracy but slower runtimes.

Experimental Setup

Our experiment was built to answer these questions:

1. Does the switch to alpha-beta pruning from greedy selection yield a higher average accuracy?
2. Does the switch to alpha-beta pruning from greedy selection yield a higher number of 100% accuracy solutions?
3. Does the switch to alpha-beta pruning from greedy selection yield faster or slower solve times?
4. How does the change in α value affect time and accuracy?
5. How does our best implementation compare to Dr. Fill.

From our dataset of crossword puzzles, we compiled 92 puzzles for solving. We then wrote a script to solve all of these puzzles with varying

α values while recording time, accuracy, α, and puzzle-solved.

On each of the 92 puzzles, we would solve with a starting α value starting at $\alpha = 1.0$ and then drop the α value by .05 and solve again, recording the data. We repeat this process until we hit an iteration that is requiring more than 15 seconds to input any one new word onto the board. Once this happens, the amount of time it takes to solve each puzzle grows exponentially, and due to time constraints, we decided that we must move to the next puzzle. We repeat the same process for the next puzzle.

After our experiment finishes, we split the data into two sets: Greedy and alpha-beta Pruned. We created the Greedy set by compiling all solutions solved with $\alpha = 1.0$, and we created the alpha-beta Pruned set by compiling all solutions in which $\alpha \neq 1.0$. We can split the data this way because if $\alpha = 1.0$, then the branching rate is equal to zero, meaning that it will always select the first possible candidate word.

Evaluation

Approach	**Time (s)**	**Accuracy**
Greedy Selection	0.52	0.8683
Alpha-Beta Pruning	9.8345	0.8951

Table 2: Average Time and Accuracy by Implementation)

After running our experiment, we evaluated our experimental results on three metrics:

1. Percentage of puzzles that each implementation solved perfectly (meaning every square was correctly filled in) (Table 2).

2. Average Accuracy for each implementation (Table 3).

3. Average time taken for each implementation (Table 3).

These metrics supplied us with enough information to answer the questions described in the Experimental Setup section.

Results and Discussion

Greedy Selection Results

Our greedy selection approach worked as hypothesized. For this implementation to perfectly solve a puzzle, our model must guess correctly for every clue on the first attempt. We knew this would be difficult for our model because it only had an accuracy of 55.10% (Table 1). The greedy selection approach perfectly solved $4/92$ puzzles with an average accuracy of 86.83% and took an average time of 0.52 seconds to complete. This made greedy selection our fastest approach (Table 2).

Alpha-Beta Pruning Results

The alpha-beta pruning approach yielded increased accuracies over greedy-selection as hypothesized. The alpha-beta pruning approach was designed to increase FLIGHT's accuracy through branching, and it worked as seen in Table 2. This implementation perfectly solved $19/92$ puzzles with an average accuracy of 89.51% and average time of 9.8345 seconds (Table 3).

Discussion

We ran several tests adjusting the α values to determine the degree that alpha-beta pruning improved the performance. For ninety two puzzles in our dataset, we compiled the results of the accuracies and the times of the experiments. We find, unsurprisingly, that the lower the α value, the more time was taken to solve the puzzle. This is not surprising because decreasing the α value increases the rate of branching, thus forcing FLIGHT into more computational load.

We also found that a high α value correlated with a decreased probability of a correct fill. This is to be expected. For example, Figure 1 shows a correlation between higher accuracy and lower α. After generating statistics on the time and accuracies, we wanted to find whether or not the greedy approach would suffice. After graphing our results, we identified that the accuracy was not necessarily increasing after every decrease in α. We decided to run a mixed effect regression, to determine the correlation of the terms. We decided on this type of regression because we understood that our data was not independent, i.e. some of our files would yield the similar accuracies and times.

After fitting a mixed effect model, we find that the coefficient between accuracy and α is negative, and in addition, we find that the significance value is nearly zero, thus making it significant.

The results we achieved were very encouraging and followed our hypotheses. We compiled our average time taken to solve and average accuracy for our two implementations which can be seen in

Approach	**Number of Perfect Solutions (of 92)**	**Percentage**
Greedy Selection	4	4.35%
Alpha-Beta Pruning	19	20.65%

Table 3: Number of Perfect Solutions by Implementation

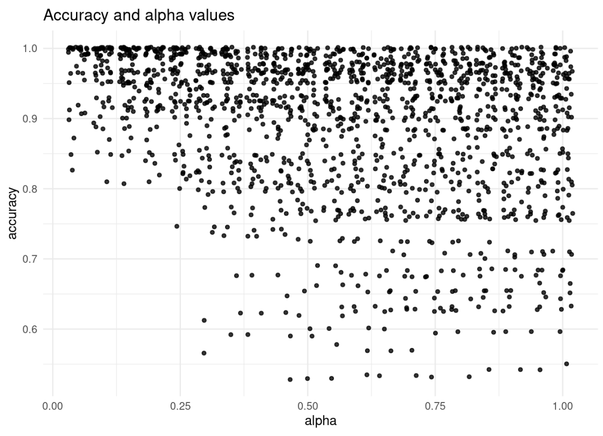

Figure 1: Plot of Accuracy against α

Table 2. We also compiled the number of times each implementation solved a puzzle with 100% accuracy which can be seen in Table 3.

Switching from greedy selection to alpha-beta pruning yielded a 2.68% increase in average accuracy and a 9.3145 second ($1,791.25\%$) increase in average time. The switch also yielded more than four times as many 100% accurate solutions.

Beating Dr. Fill

One of the goals of this project was for FLIGHT to beat Dr. Fill, and in some ways we accomplished this task. When Dr. Fill competed in the 2021 ACPT, it only had two incorrect letters out of seven puzzles. This roughly translates to $2/1,351$ squares being incorrectly filled. Our average accuracy of 89.51% shows that we we're pretty far off; however, FLIGHT is faster. Dr. Fill takes on average about one minute to complete a puzzle, and our most accurate implementation took an average of 9.8345 seconds.

Although we were able to improve the time complexity of this problem significantly, FLIGHT was unable to achieve nearly as high solution accuracy as Dr. Fill. We don't believe this to be to the discredit of FLIGHT, however, because we were able to achieve impressive results nonetheless.

Limitations and Future Work

There were a few limitations with our model. We found that even though out of vocabulary words are rare, they still exist, which makes the model susceptible to error. One of the advantages that Dr. Fill had was that it used dense passage retrieval, so it could make inference on words that were not yet used in crosswords.

We also though of stacking and ensembling models. Given the time constraints, we were not able to accomplish this, but ideally, having a model that uses several algorithms, not just FastText models, could lead to better results. For example, we could have used a character level RNN when the number of dots becomes minimal.

We considered looking for an ideal α value by using stochastic gradient descent, but we found that this was not a good implementation because the crosswords were not consistent. From what we modeled, the best value we found was around a tenth.

References

K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma. The extreme classification repository: Multi-label datasets and code, 2016. URL http://manikvarma.org/downloads/XC/XMLRepository.html.

Marco Ernandes, Giovanni Angelini, and Marco Gori. Webcrow: A web-based system for crossword solving. In *AAAI*, 2005.

Matthew L. Ginsberg. Dr.fill: Crosswords and an implemented solver for singly weighted csps. *ArXiv*, abs/1401.4597, 2011.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

Siddhant Kharbanda, Atmadeep Banerjee, Akash Palrecha, and Rohit Babbar. Embedding convolutions for short text extreme classification with millions of labels, 09 2021.

Oliver Roeder. An a.i. finally won an elite crossword tournament. *SLATE*, 04 2021.

Noam Shazeer, Michael Littman, and Greg Keim. Noam m. shazeer, michael l. littman, greg a. keim. 04 2000.

Anonymous ACL submission. Get your model puzzled: Introducing crossword-solving as a new nlp benchmark. *ArXiv*, 2021.