CMPS 140 Final Report: Avrae Smart Search

Andrew Zhu

angzhu@ucsc.edu

Abstract. The goal of this project was to develop a fast, accurate, and memory-efficient search algorithm to map user queries to results. This was accomplished using a variety of neural network-based models, and the final resulting algorithm is about 50% faster and 15% more accurate than the industry standard.

Keywords: search · neural network

1 Introduction

This project was based around training a neural network to use as an accurate, fast, and memory efficient search algorithm to best map user queries to results. Because of my passion for D&D, and the fact that I could collect data on searches easily, the network was trained in the domain of Dungeons & Dragons spells.

1.1 Goal and Impacts

While the original goal of the project was to develop the algorithm for D&D spells only, it could easily be extendable to many other search use cases.

1.2 Related Work

The industry standard search algorithm I wanted to improve was the Levenshtein Distance algorithm, which chooses a result based on the amount of characters that differ between a query and all possible results.

2 Data

The final network was trained based on data collected over about 4 months, with over 860,000 pairs of queries and results. This data consists of about 19,000 unique spell queries, and 501 categories of spells. This data was preprocessed to create three different tokenizations of each query, account for vague searches (i.e. queries that could match multiple different results), and create an L1-normalized one-hot encoded output vector.

The amount of queries per result is not equal for each result, however, after preprocessing, all duplicate queries were consolidated into one training point. If one query mapped to more than one result, it was represented by having multiple activations in the L1-normalized output vector (e.g. $fireball \rightarrow$ [1.0 0.0 ... 0.0], but $fire \rightarrow$ [0.7 0.1 ... 0.2]).

A release of the unprocessed search data, as well as some statistics about the data, can be found at https://github.com/mommothazaz123/datasets/tree/master/avrae.

3 Methodology

Five models were used over the course of this project: two baseline, and three neural network-based models.

3.1 Baselines

The two baseline models used to evaluate the neural nets were a simple partial match algorithm, and the industry standard Levenshtein Distance algorithm.

The partial match algorithm ($baseline1$) simply iterates over all possible results, and checks if a given query is a substring of each result. For each result, if the query is a match, its confidence is the length of the query divided by the length of the result name, or 0 otherwise. It was chosen as a baseline algorithm because it was fast and easily/commonly implemented, but not very accurate.

The Levenshtein Distance algorithm ($baseline2$) iterates over all possible results, and computes a distance between the query and each result based on how many character edits it would take to transform the result into the query. For each result, it generates a confidence value based on this edit distance. It was chosen as a baseline algorithm because it was fairly accurate and commonly used, but slow.

3.2 Neural Network Models

Many neural network-based models were used over the course of the project, but three in particular stand out. All networks were trained using the same training data, and evaluated on the same test set.

1. $magic1\_dense$: This model was the first neural network-based model used in this project. It consisted of three fully-connected layers: the input layer, a 16D vector using magic string encoding (see 3.3), a 128D hidden layer using ReLU activation, and a 501D one-hot encoded output vector using softmax activation.
2. $magic2\_conv\_smaller$: This model improved upon the initial fully-connected network by optimizing the magic string used to encode inputs to better account for fat-finger typos, and adding a 1D convolution layer to extract patterns from the text in a similar manner to images. It consists of 3 layers, and is laid out in this order:
   (a) 16D Magic string encoded input layer
   (b) Convolution layer with 25 filters, step size 1, and mask size 2 (ReLU)
   (c) Average pooling layer
   (d) 501D One-hot encoded output vector (softmax)
3. $embedding\_conv\_maxpool$: This model further improved upon the previous convolutional model by replacing the input layer with an embedding layer (see 3.4), allowing the network to learn how to best represent characters on its own, rather than using magic string encoding. It is laid out in this order:
   (a) 16D Embedding layer
   (b) Convolution layer with 25 filters, step size 1, and mask size 3 (ReLU)
   (c) MaxPool layer
   (d) 501D One-hot output vector (softmax)

3.3 Magic String Encoding

The first method of encoding user inputs for training in the neural networks was called Magic String Encoding. To encode a string in this manner, I examined the first 16 characters of a query, and encoded a 16D vector using this code:

```
INPUT_LENGTH = 16
MAGIC_1 = "abcdefghijklmnopqrstuvwxyz '"
MAGIC_2 = "qwertyuiopasdfghjkl'zxcvbnm "

def encode(query):
    filtered = query.lower()
    filtered = ''.join(c for c in filtered if c in MAGIC_1)
    cleaned = filtered[:INPUT_LENGTH].strip()
    num_chars = len(MAGIC_1)
    tokenized = [0.] * INPUT_LENGTH
    for i, char in enumerate(cleaned):
        tokenized[i] = (MAGIC_1.index(char) + 1) / num_chars
    return tokenized
```

Replacing `MAGIC_1` with `MAGIC_2` to encode using a different magic string. Using `MAGIC_2` to encode reduced the impact fat-finger typos made on the network's prediction, as letters closer together on the keyboard had a small difference.

3.4 Embedding

The second method of encoding user inputs used in the final algorithm was embedding, where the network learns how best to represent each character. This technique is commonly used in bag-of-words models to represent words, but I found that it was effective to represent characters as well. In the final model, I found that using a 16D vector to represent each character greatly increased accuracy in the final evaluation.

3.5 Misc. Discussion

1. Why a 16D input?
   I chose to only look at the first 16 characters of a query based on a few reasons:
   (a) The length of the average user query is 10.26 characters.
   (b) Users tend to be lazy, and most information can be gleaned from the first 16 characters of any query.
   (c) Each spell has at least one unique 16-character substring.
   (d) It's a nice round number.
2. Why use a convolutional structure?
   I chose to use a convolutional structure to extract features from text in more or less the same manner as a convolutional structure extracts features from images. This could help in recognizing common substrings, like "fire," "wall," or "bolt."

3. Why have an L1 normalized output?
   In the training data, the output vector is L1 normalized because of vague queries: queries that had multiple possible outputs. In this case, it's desired for the neural network to have a lower confidence on multiple possible results, allowing the user to pick from a list, rather than randomly guessing one of the possibilities.
4. How is the network used?
   When a user inputs a query, the activation at a given index of the network's output vector represents its confidence that the user is searching for the result at that index. Usually, the user is presented with the network's top 5 guesses, and selects one; this data is then stored for further training.
5. How fast can this network be trained?
   On my laptop (a 2015 Macbook Pro), it takes about 5 minutes to train the network on a preprocessed training set of about 19,000 data points. The preprocessing itself takes about 2 minutes.

4 Evaluation/Results

In this project, I used two different methods of evaluation: a more concrete method, where all different models were tested on the same set of 17,000 queries, and the accuracy and speed of the model was recorded, as well as a more abstract evaluation where end users were asked if they found a notable improvement when using a neural network as opposed to a baseline model.

4.1 User Evaluation

Due to time limitations, I was only able to run a user evaluation on the $magic1\_dense$ model. When comparing $magic1\_dense$ to $baseline1$, 61.1% of users said that there was a notable improvement in the accuracy of the search results, 33.3% said that there was no notable difference, and 5.6% said that there was a notable decrease in the accuracy.

4.2 Data-Based Evaluation

To run this data-based evaluation, I chose a set of 16,927 unique queries and results from the dataset, and evaluated the different models based on both the accuracy of the model, and the time it took for the model to make predictions for the dataset. In this evaluation, I consider accuracy to be the number of queries where a model correctly predicted the corresponding result as its top choice; however, results are included for when a model predicted the correct result in 2nd to 3rd place, and 4th to 10th place.

Baseline Models

Model	Top 1	Top 3	Top 10	Accuracy	Time (sec)
$baseline1$	3,326	643	323	19.6%	5.05
$baseline2$	10,788	2,391	1,150	63.7%	73.93

As shown here, the two baseline models leave something to want; the first (naive partial match) is fast but inaccurate, while the second (Levenshtein Distance) is accurate but slow. Additionally, both these algorithms run in linear (for $baseline1$) or polynomial (for $baseline2$) time.

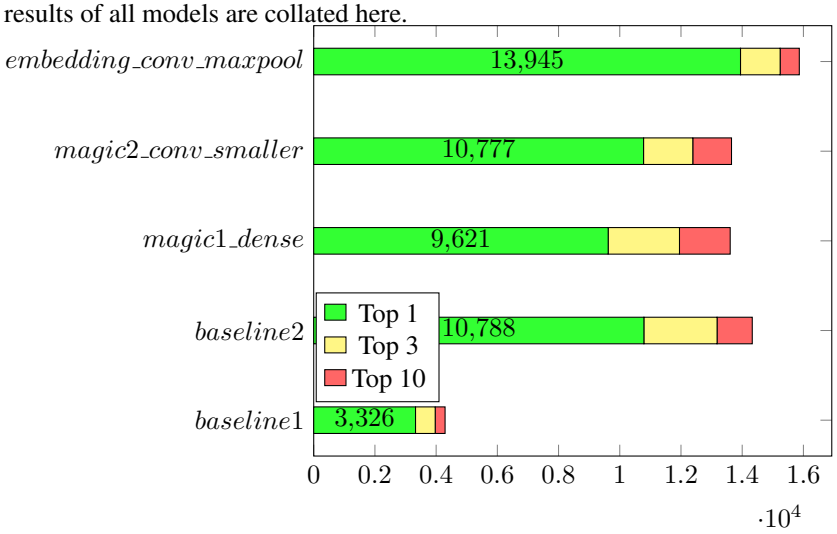
Network-Based Models

Model	Top 1	Top 3	Top 10	Accuracy	Time (sec)
$magic1\_dense$	9,621	2,330	1,657	56.8%	17.03
$magic2\_conv\_smaller$	10,777	1,609	1,265	63.6%	38.46
$embedding\_conv\_maxpool$	13,945	1,295	626	82.4%	39.90

Compared to the two baseline models, the first two smart models seem to sit in a happy medium, with comparable accuracy to $baseline2$ while running significantly faster. When embedding is introduced in $embedding\_conv\_maxpool$, however, the network-based model becomes significantly more accurate than $baseline2$, while still running significantly faster. Overall, $embedding\_conv\_maxpool$ is 18.7% more accurate than $baseline2$, and runs 46% faster. Additionally, all smart models run in constant time, making them highly scalable.

4.3 Results

The results of all models are collated here.



5 Conclusion

It is possible to create a fast and accurate model using character embedding that can be trained quickly on various datasets. This algorithm is robust against fat-finger typos, off-by-one typos, and common substitutions (given previous training data), and runs in linear time. Overall, on a set of about 17,000 evaluation queries in a codomain of 501 spells, it is 18.7% more accurate and 46% faster than the naive industry standard.

6 Code

Neural Net implementation, data preprocessors:
https://github.com/mommothazaz123/avrae-search-nn
Data:
https://github.com/mommothazaz123/datasets
Production implementation:
https://github.com/avrae/avrae/blob/master/cogs5e/funcs/lookup_ml.py