Is Feature Engineering Futile?

Stanford CS224N Default Project
Track: SQuAD

Jacob A. Smith
Department of Computer Science
Stanford University
jacobas@stanford.edu

Abstract

Question answering is a problem that we have approached from a naive perspective, using word embeddings as the extent of our feature engineering. While this has produced mediocre results, applying additional feature engineering can improve upon this baseline. Some well motivated changes produced improved results, such as an exact match feature, while others actually harmed the performance compared to the baseline, a combination of exact match and a lemma match feature. My findings indicate that enhanced feature engineering is useful for fine-tuning a model's performance on a task, but it will not produce significantly improved results.

1 Key Information to include

- Mentor: Angelica Sun
- External Collaborators: None
- Sharing project: No

2 Introduction

The goal of question answering is to create a model capable of answering a question given context which may or may not contain an answer. This is akin to basic reading comprehension tasks assigned to children in the early stages of their education, and is a useful problem to be able to solve. Given context, it is desirable for a machine to produce answers to questions like 'What was Martin Luther's 95 Theses?' or 'What caused the downfall of the Roman empire?' given that a human has produced answers to such questions. This type of question answering system is challenging because the knowledge contained within human language is not easily interpretable for machines. On the other hand, this type of QA system does not rely on much more intensive knowledge base structures. A well implemented system is easy to update because the only change required is to fix the source of truth. Example datasets like SQuAD which are based off of Wikipedia are great examples because Wikipedia is a datasource that is constantly updated and well maintained.

Current methods for approaching question answering with well defined contexts and questions involve large language models like Bert. The results, as expected for such large models, are very good. However, this project aims to improve upon earlier models for QA, such as the Bidirectional Attention Flow (BiDAF) model. BiDAF achieves decent results on QA tasks using just 300 dimensional GloVe embeddings. Such straightforward input to vector approaches for deep learning networks is very common. The intuition for such simplistic approaches is that the deep learning networks should be able to learn more complex syntactic and semantic representations within the network. In that case, my question is, what if we jump start that process by passing in information that a human answering similar questions might be using to come up with a response. For example, a human might highlight words in the context that appear in the question. Humans might also connect words that share a lemma root, words like run, runs, and running. In addition, certain question words will often ask for different types of answers. The word ‘how’ might indicate that your answer should include some kind of verb, while the words like ‘who’ and ‘where’ might tend to be asking for proper nouns like names and locations. Therefore, a part of speech tag for the context may be useful.

3 Related Work

This approach of augmenting context with useful engineered features is not entirely new. For example, Danqi Chen et. al. saw some improvements from a set of similar features in their paper “Reading Wikipedia to Answer Open-Domain Questions.” [1] Ultimately, the aim is to understand if putting the extra work into creating more high level input features is worth the extra effort. My findings will lead to the answer that it is not necessarily yes or no, but more generally, it might not be worthwhile. Their work illustrated how features could be useful, but I want ultimately they were not the focus of the paper. In this work, my goal was to do a deeper analysis of the impacts of the engineered features.

The model that I am using to extend upon is based off of the work of Minjoon Seo et. al. in their paper “Bidirectional Attention Flow for Machine Comprehension” [2] Their approach to machine comprehension involved modeling the relationships between words in the context and query using attention mechanisms between each word. The model improved upon the state of the art at its time, and provides a useful baseline for performance enhancements in my own model. I extended upon their work by providing the above mentioned feature augmentations with the intention of kickstarting the attention process with certain implicit-made-explicit details of the input.

4 Approach

This section will outline the approaches that I took to implementing my engineered features. The inspiration for my model’s additional features, “Reading Wikipedia to Answer Open-Domain Questions” [1], was not specific about implementation or calculation of the features. I began by implementing the exact match feature. This feature’s purpose is to indicate if a word in the context also appears in the query. I initially calculated this feature at runtime by reading the batch of context words and calculating a binary feature that was one if the word matched any word in the query and it was not the pad word, and was zero otherwise. This approach of calculating at runtime proved to be inefficient, and I moved to expanding the dataset by calculating the exact match feature as the Dataset was loaded. I called this dataset SQuAD with Exact Match. To accommodate this additional feature, I lowered the dimensional output of the embedding layer by 1 and concatenated this feature to the end for each word in the context. I also zero padded the query words since I only wanted this feature calculated for the context.

The next feature that I implemented was an indicator for if the lemma form of each word in the context had a corresponding match to a lemma word in the query. I took my learnings from the last feature and created an augmented dataset which I called SQuAD with Lemma Match. In order to calculate the lemma form of words, I used the NLTK WordNetLemmatizer, which in its simplest form, is like getting the form of each word as you’d find it in the dictionary. I then used a similar matching process to what I used for the exact match feature, except that I had to pass every word into the Lemmatizer in order to compare Lemma forms of the word. As above, I lowered the dimensional output of the embedding layer by 1 to accommodate the single dimension of the lemma match feature.

I then wanted to use both an exact match and a lemma match feature in conjunction. Following the same intuition as above, I lowered the embedding layer output’s dimension by 2 and concatenated these two binary indicator features at the end.

Next I wanted to expand the model with Part of Speech (pos) tagging. I created another dataset which upon loading would take the words indexes and pass them to a pos tagger function. For both the context and query words, the function would convert them from their indexes to actual words. To compute the part of speech, I used the NLTK pos_tag function, which uses the UPenn tagset. In total, including the various symbol tags, there were 45 total possible options. My first approach was with a naive one-hot encoding. Each word would then have a corresponding 45 length vector to indicate its part of speech. I concatenated the word embeddings of dimension 100 with the one-hot encoded part of speech tag, then passed that through a linear layer from dimensionality of 145 to 100. The rest of the model remained unchanged. I made a few practical speedups to this process as well since the calculation of the part of speech tags was a very time consuming process. First, I filtered out the pad tokens before getting the tags, then re-padded them up to the max sequence length of 401 with a zero vector. Secondly, I saved the output so that multiple runs could load the tags instead of recalculating them.

I then iterated upon this approach, by creating an embedding layer for the tags instead of a one-hot representation. The embedding layer took in values up to 45, with 0 reserved for the untagged pad tokens. Each of these values was the same as the one hot index in the last approach. The embedding layer output size was a tunable hyperparameter. I then extended the hidden size by that output size for the remainder for the model.

5 Experiments

5.1 Data

This project makes use of the Official SQuAD 2.0 dataset. The data is split into a train, dev, and test set with each containing 129,941 examples, 6078 examples, and 5915 examples respectively. The SquAD dataset is composed of paragraphs for context and a question. These context and question pairs are the inputs and the goal is to predict the input and output indices of the context where the answer is contained. What is specific to the 2.0 variation of the dataset used for this project is that some questions will be unanswerable with the paragraphs given to them. The model can predict 0 for the start and end index, which corresponds to a "not answerable response". The Paragraphs and questions are filtered to only have a maximum word count of 400, so the inputs for SQuAD 2.0, with the not answerable response, can be up to 401 words long. Not that because the answer is already in the inputs, the goal is for text selection as opposed to text generation. Here is a SQuAD example question, context, and answer:

Question: When B cells and T cells begin to replicate, what do some of their offspring cells become?

Context: When B cells and T cells are activated and begin to replicate, some of their offspring become long-lived memory cells. Throughout the lifetime of an animal, these memory cells remember each specific pathogen encountered and can mount a strong response if the pathogen is detected again. This is "adaptive" because it occurs during the lifetime of an individual as an adaptation to infection with that pathogen and prepares the immune system for future challenges. Immunological memory can be in the form of either passive short-term memory or active long-term memory.

Answer: long-lived memory cells

And here is an example where the question is not answerable:

Question: After 1945, what challenged the British empire?

Context: In World War II, Charles de Gaulle and the Free French used the overseas colonies as bases from which they fought to liberate France. However after 1945 anti-colonial movements began to challenge the Empire. France fought and lost a bitter war in Vietnam in the 1950s. Whereas they won the war in Algeria, the French leader at the time, Charles de Gaulle, decided to grant Algeria independence anyway in 1962. Its settlers and many local supporters relocated to France. Nearly all of France's colonies gained independence by 1960, but France retained great financial and diplomatic influence. It has repeatedly sent troops to assist its former colonies in Africa in suppressing insurrections and coups d'état.

Answer: N/A

5.2 Evaluation method

The primary metric of evaluation is F1, the harmonic mean of precision and recall. Mathematically, it is 2 x prediction x recall / (precision + recall). We also have an Exact Match (EM) score to indicate the percentage of exactly correct answers. Note that this EM score is not the same as the exact match feature defined above.

5.3 Experimental details

The BiDAF model is composed of an Embedding layer, an RNN encoding layer, a BiDAF attention layer, another RNN encoding layer, and finally the BiDAF output layer. Further details about the layers are described in the original paper [2]. We used a hidden size of 100 and a drop probability of 0.2. We used a batch size of 64, an ema decay of 0.999, and a learning rate of 0.5. We trained for 30 epochs. In order to accommodate for the extra length of one along the hidden size dimension added by the exact match feature, I reduced the output size of the embedding layer by one to 99. I also configured the model to take an additional input, the exact match feature, at each forward step. The same is true for the lemma match feature. For the model using exact match and lemma match, I reduced the output size of the embedding layer by two to 98.

The BiDAF model with Part of Speech using the naive one-hot encoding process remained largely the same, with the additional linear layer to go from an input size of 145 to 100. The BiDAF model with Part of Speech using the embedding layer, I ran with two different output sizes. I tried 6 and 10. My intuition for these sizes was that 2^6 could represent all of the possible tags using only binary, and then a size 10 embedding would allow for a more diverse representation of the part of speech, possibly leading to a more useful embedding. For the size 6 embedding the hidden size was 106 and for the size 10 embedding the hidden size was 110.

5.4 Results

The naive BiDAF model that used 300 dimension GloVe embeddings was trained for 30 epochs on the training dataset. The results for the baseline model and the baseline augmented with the engineered features on the dev set are reported in the table below: Based on these performances, I used the Exact

Table 1: Results for baseline and feature augmented models on the dev set

	EM	F1
Baseline	57.64	61.26
Exact Match feature	60.41	63.7
Lemma Match feature	59.44	62.49
Exact Match and Lemma Match feature	58.46	61.84
Part of Speech One Hot	56.34	59.75
Part of Speech Embedding (6)	58.36	61.67
Part of Speech Embedding (10)	56.75	60.22

Match feature to create my submission for the test leaderboard. My results using this model compared to the baseline listed on the leaderboard is in Table 2. These results are quite fascinating, because

Table 2: Results for baseline and feature augmented models on the test set

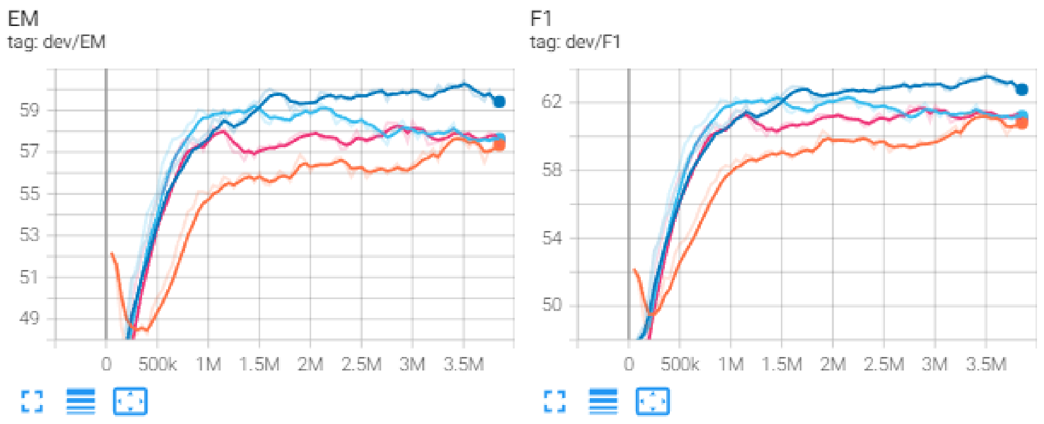
	EM	F1
Baseline	57.524	61.01
Exact Match feature	57.63	60.90

I expected more complicated features to produce more useful representations, such as the part of speech embeddings. However, this was not the case. Even the two highest performing features, the exact match and lemma match, did worse when combined. The addition of these two features did not contribute significantly more than was lost by decreasing the representation of the word embeddings to accommodate them. I am somewhat surprised by the worse performance of the exact match feature on the test set versus the dev set, as it appeared to be a significant improvement over the baseline model. However, it appears that the realized gains on the dev set did not generalize well.

6 Analysis

We have seen that this feature engineering has not provided significant benefits in terms of accuracy compared to the baseline model. However, there are some other benefits that may have a real world value. The most significant of these was the rate at which the exact match, lemma match, and exact/lemma match augmented models learned. Compared to the baseline, which first took a dip before improving in F1 and EM, these augmented models began improving immediately and more rapidly.

Figure 1: Training progress of various augmented models compared to the baseline



For figure 1, the baseline model is in orange, the exact match is dark blue, the lemma match is cyan, and the exact/lemma match is pink. We can see here that these augmented models achieved similar dev set performance compared with the baseline model's best performance in about a third of the time. I believe that the reason these models achieve similar performance so much faster is that the baseline model takes millions of iterations to encode the exact relationships through attention that we augmented the model with. That is why the ultimate performance of all of these models end up in a similar place, they all have the same information, just represented differently. This kickstarting of the attention mechanism has real world value in systems where compute is the limited resource. This principle could apply more generally as well, where we capture some characteristic from the structure of the data and encode it directly as a feature. However, despite improvements in training requirements, the fact that all of the models end up at a similar destination goes to show that in environments where compute is not a limiting resource, the baseline without feature engineering is a sufficient solution.

7 Conclusion

So where does feature engineering fall in the world of deep learning? Ultimately, feature engineering provides a clear tradeoff with a limited scope in value. A useful target for feature engineering in the context of NLP and attention is something that captures a structural regularity in the data that is not already explicit. The value of augmenting your model with such a feature is the potential to reduce the training requirements to achieve otherwise similar performance. However, the beauty of deep learning is that it is capable of learning the well defined patterns in data without needing to be told so, if only eventually. In general, my recommendation would be to limit structural feature engineering and focus on using a state-of-the-art model architecture that frees the model to learn the kinds of patterns that you want it to capture. The additional benefit is that it can learn patterns that you may be unaware of or do not fully understand. However, this feature engineering can be a useful concept to keep in your toolkit for constrained environments.

References

[1] Danqi Chen, Adam Fish, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions. In *arXiv preprint arXiv:1704.00051*, 2017.

[2] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectionalattention flow for machine comprehension. In *arXiv preprint arXiv:1611.01603*, 2016.