Final Default Project

Charles Vidrine
`cvidrine@stanford.edu`

Charlie Furrer
`cfurrer@stanford.edu`

Priya Chatwani
`priyac5@stanford.edu`

Abstract

This paper proposes to approach the problem of context question answering via clever feature engineering. Building off of the baseline model, we incorporated additional training features into the word vectors to enhance our model's answer retrieval. Finally, we used the SQuAD 2.0 dataset to evaluate the results. Our main findings revolve around the use of a TF-IDF score in feature engineering as that was the most novel and time-intensive feature that we implemented.

1 Introduction

This paper looks at the problem of answering factual questions in a context setting using purely feature engineering. This is a topic that many other researchers have tackled. Throughout our project, we looked to *Reading Wikipedia to Answer Open-Domain Questions* (aka Dr.QA) to guide many details of our work. However, while our approach draws heavily from Dr.QA, it is fundamentally different because we are looking at context question answering rather than open-domain question answering. In Dr.QA, researchers Danqi Chen, Adam Fisch, Jason Weston and Antoine Bordes use all of Wikipedia as their knowledge source. In answering each question, their model cleverly retrieves the articles that seem most relevant to the question among 5 million different items. In our dataset, we are already given the paragraph and context for which our question is being drawn. Thus, we are not concerned with document retrieval; rather, we are concerned with training our model to accurately pull from the context that corresponds with the given question. Since we are tackling context question answering, we decided to focus on feature engineering rather than alter the baseline model given to us.

In implementing our feature engineering, we draw heavily upon what was implemented in the document reader portion of Dr.QA. We first concatenate character embeddings with the existing word embeddings. From work done in machine translation, we know that character-level encoders now consistently outperform subword-level encoders on many language pairs; thus, we added a character-level embedding to our model. Next, we computed a TF-IDF score for each word and added that to our embedding as well. While the Term Frequency - Inverse Data Frequency score, or TF-IDF, was the most difficult to compute and index, we saw that Dr.QA was successful in using TF-IDF to retrieve relevant documents and hoped it would similarly help our model accurately retrieve query answers. Finally, we tried to implement exact match as an additional feature, but we were unsuccessful in improving our score.

2 Related Work

In addition to Dr.QA, our work relies heavily on the SQuAD reading comprehension dataset. SQuAD, presented in 2016 by Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang, revolutionized the domain of question-answering by providing 100,000+ questions posed on Wikipedia articles, in addition to the text segments containing the corresponding answers.[2] We used the SQuAD 2.0 dataset to train and evaluate our model.

We also looked to past work in sub-word models to inform our implementation of a character-level embedding. Beginning in 2015 and 2016, researchers became successful in purely character-level neural machine translation models. In *Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models*, Minh-Thang Luong and Christopher D. Manning built a hybrid model that mostly relies on word embeddings but uses character embeddings for rare words.[3] This hybrid model, unlike purely word-level models, never produces out-of-vocabulary words, marking a significant strength in character-level translation. These Word-Character Models prove that relevant information can be encoded in character-level representations. Using this information, we concatenated word embeddings with their corresponding character-embeddings in order to give our model more information with which to train and more accurately perform.

Finally, we relied on past work and research on TF-IDF scores. In the 2003 paper *Using TF-IDF to Determine Word Relevance in Document Queries*, Juan Ramos evaluates the relevance of Term Frequency Inverse Document Frequency (TF-IDF) in query retrieval.[4] Given a corpus of documents, Ramos shows that TF-IDF is "an efficient and simple algorithm" for matching relevant documents to a given query.

3 Approach

For our baseline, we used the Bidirectional Attention Flow (BiDAF) model without a character embedding layer. This was taken directly from the default assignment handout. To improve upon the baseline, we added character embeddings, TF-IDF score to the original word embeddings to produce our paragraph-level embeddings.

3.1 Embedding Layer

The embedding layer takes a vector of word indices $w_1, ..., w_n \in R$, and produces a vector $\tilde{p_i}$ for each token in a paragraph such that:

$char\_embeddings = c_i \in R^{\ e\_char\ *\ max\_len}$ (note that we flattened embeddings)

$tf\_idf\_score \in R$

$exact\_match \in R$

$word\_embeddings = w_i \in R^{w_e}$

$\tilde{p_i} = [char\_embeddings, word\_embeddings, tf\_idf\_score, exact\_match]$

Then, the embedding layer projects each embedding to have dimensionality H, giving us a new vector $h_1, ..., h_n \in R^H$. This hidden vector is then fed to a two-layer highway network that refines the embeddings. A discussion of the features follows below.

3.1.1 Character Embeddings

The character embeddings are created using a Linear Embedding matrix. Since the resultant embeddings have an extra dimension (a vector at the character level rather than word level), we flatten the vector to be of the same dimension as word embeddings.

3.1.2 TF-IDF Scoring

TF-IDF reveals how important a word is to a document within a collection of documents. TF-IDF score is usually used within the context of page ranking for search results. We had the original idea to use a slightly modified version of it as a feature for our embeddings. TF-IDF is mathematically defined as such:

TF = $tf_{d,i} * \frac{1}{\sum_d tf_{d,i}}$

IDF = $log(\frac{N}{df_i})$

TF-IDF = $TF * IDF$

Where $tf_{d,i}$ is the term frequency of term i in document d, and N is the total documents in the corpus, and df_i is the number of documents word i appears in for the whole corpus.

To produce the TF-IDF score for each word, we stored the total number of documents and document occurrences in a dictionary during preprocessing. Then, we used a vector the length of our vocabulary to map word indexes to relevant TF-IDF information. We made use of Pytorch's excellent batch indexing capabilities to retrieve the information at training time. The vectorization of this calculation allowed for much faster training than a typical iterative approach would allow.

3.1.3 Exact matching

This was a binary feature that was 1 if the exact word in the question was in the context, and 0 otherwise. Generally, a word having an exact match is a good clue that it is important to answering the question. We calculate these exact matches in the initialization of the model, so that we don't have to do a scan for each token during the training process.

4 Experiments and Results

4.1 Data

For our dataset, we used the SQuAD 2.0 dataset provided as part of the default project. For evaluation of our model, we relied on the default scoreboard metrics outlined in the default project handout, namely Dev NLL, EM, F1, and AvNA. See the default project handout for a more in-depth analysis of evaluation metrics. In order to maintain continuity and understand the impact that each feature had on the model, we used the default hyperparameters during our training. For each model we trained until near convergence.

4.2 Results

4.2.1 Baseline

The scores we achieved with the baseline model were:

Dev NLL: 03.36
F1: 59.82
EM: 56.28
AvNA: 67.11

4.2.2 Character Embedding

After adding the character embeddings, on the dev set we had a slightly higher Dev NLL loss, but our other three scores showed improvement. On the test set, we received scores of:

F1: 60.9
EM: 56.87

This model took longer to train due to the increased number of parameters to learn and size of the embeddings, but in return we saw significant improvement in our accuracy.

4.2.3 TF-IDF

The addition of the TF-IDF feature led to another improvement for our model. Originally, we stopped training TF-IDF after 30 epochs and tested it on the dev set. We received the following results:

Dev NLL: 3.42
F1: 62.22
EM: 58.80
AvNA: 69.75

However, these dev results were lower than our model that only included the character embeddings, and so we did not evaluate this model on the test set. We concluded that the additional feature required more time to train, and so we allowed it to continue training for 5 more epochs. After the additional training, the dev results were greater than the character embedding dev results. The TF-IDF model with additional training scored the following on the test set:

F1: 61.925
EM: 58.225

Compared to the character embedding model, this model took a similar amount of time per epoch to train, but required more epochs to converge, leading to a longer training time. However, we again were able to see improvement in both the F1 and EM accuracy.

4.2.4 Exact Match

Another feature that we attempted to implement was exact matching. However, we found that the exact match feature actually made performance significantly worse compared to the baseline. On the dev set, exact matching scored:

Dev NLL: 4.92
F1: 37.80
EM: 34.73
AvNA: 61.28

This extremely poor performance on the dev set led to us scrapping the feature instead of submitting it for testing.

4.2.5 Smaller Hidden Size

After implementing our desired features, we proceeded to tune the hidden size of the model to see if we could make improvements. By using a hidden size of 75 instead of 100, we were able to achieve the following scores on the test set:

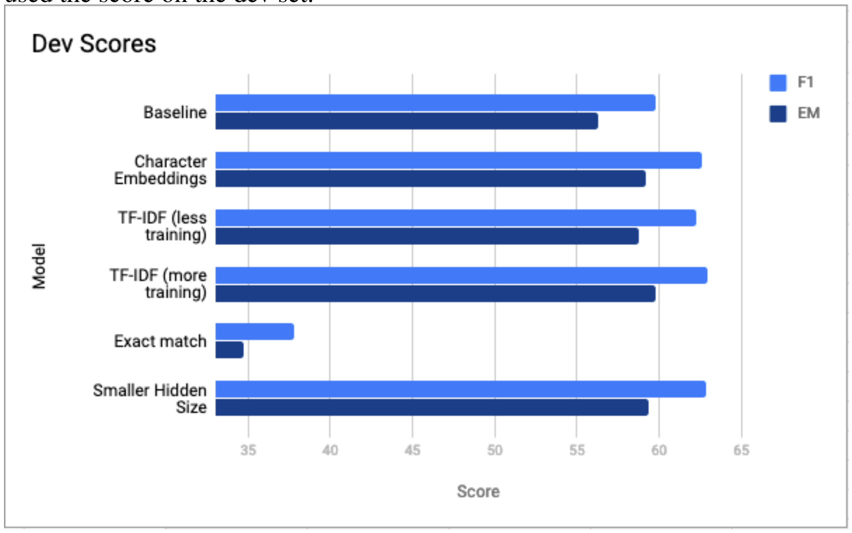
F1: 61.3
EM: 57.8

Even though these scores were slightly worse than our TF-IDF model, they were still an improvement over the model with only character embeddings. We also found that this model was able to train over twice as fast as our other models due to the computational impact of shrinking the hidden size. In the encoder, the hidden size was similar, but in the modeling and attention layers the impact was more profound. Sacrificing a small amount of accuracy led to a significant boost in training speed.

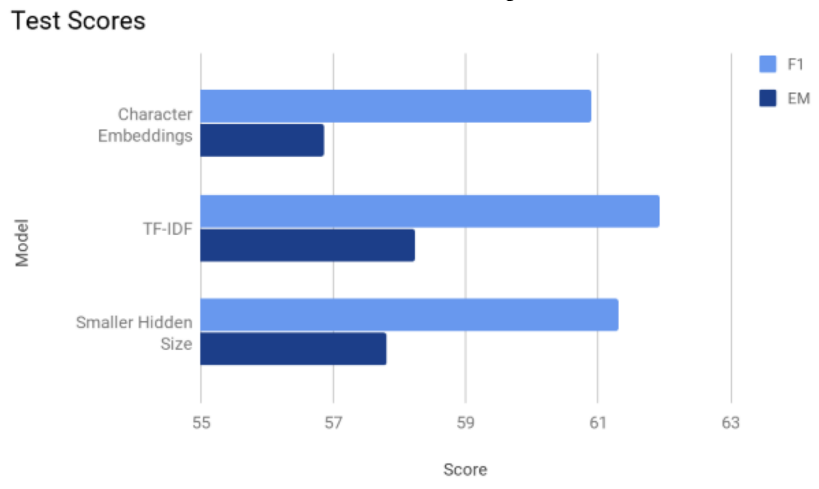
5 Analysis

5.1 Results

We iteratively improved our model by adding features to the baseline. The largest growth was observed by adding character embeddings to the original glove word embeddings. We then saw incremental growth by adding a TF-IDF score to the embedding for each word. Finally, we saw horrible performance by the exact match model. After feature engineering, we tuned the size of the hidden layer to decrease model complexity. To determine if a model was good enough to test, we used the score on the dev set.



The exact match performance was horrible, leading to us excluding that model from our test set analysis. TF-IDF with less training was slightly worse than character embeddings, which led to us training it for longer in the hopes of better results. Once we gained improved results on the dev set with TF-IDF, we chose our top three models to submit to the test leaderboard.



As you can see, our best performing model was the model that included the TF-IDF feature in the embeddings. However, we found that the TF-IDF model with a hidden size of 75 instead of 100 still outperformed the character embedding model and was only slightly worse than the TF-IDF model. This model was able to train much faster, taking 20 hours to reach convergence as opposed to the 40+ that we saw for the other models. Even though it performed slightly worse, this trade-off made it the best model in our opinion.

5.2 Feature Engineering

Due to our focus on feature engineering, we did not get a chance to change the model or tune the hyper parameters as much as we would have liked. In order to accurately measure the effects of each feature, we had to preserve the model across training and testing. Given more time, we would have complemented our feature engineering with the tuning of various hyper parameters so that we could better optimize our results.

6 Conclusion

6.1 Main Findings

Our main findings revolve around the application of TF-IDF to question answering in a context-specific environment, rather than an open-domain setting. Although the implementation of TF-IDF was non-trivial in terms of time, space, and complicated, vectorized indexing, the outcome was satisfactory. Thus, our results show that while TF-IDF scoring is traditionally used in document retrieval, it is also helpful in context-specific query retrieval.

We also drew out significant findings from our model that decreased the hidden size from 100 to 75. With a smaller hidden size, the model trained much faster, almost twice as fast as the previous models, and it still improved upon the baseline. The faster training time stems from the fact that a smaller hidden size results in less computational complexity. Thus, we showed that even in reducing some of the complexity of our model, we were still able to train a model that learned from the features we added.

6.2 Limitations

In following our approach to this project, we wanted to evaluate the effects of different features across an identical environment. For this reason, we did not tune any hyper parameters while simultaneously adding features. At the end of our project, once we had identified our best model with regards to features, we did not have time to tune as many hyper parameters as we wanted to. Each model took between 1.5 and 2 days to train, so we were heavily limited by time.

Time also became a constraint when our TF-IDF model needed more time to train and when our implementation of Exact Match was unsuccessful. We would have liked to work more on Exact Match and let all of our models train for longer.

6.3 Future Work

In a future iteration of this project, we would like to add more features to our word embeddings. Ideally, we would adjust our exact match implementation and incorporate more context-specific features, like in Dr.QA. For example, we could have trained on noun and pronoun tagging, named entity recognition, etc. We also would have liked to add an interaction term to our model. For example, we could have implemented an interaction term that calculates the TF-IDF score times the Exact Match score. This may have helped the model learn that rare words found in both the document and query are more important than common articles of speech found in both the document and query.

Additionally, we would have tuned additional hyper parameters besides just hidden size to optimize our F1 and EM results.

References

[1] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. *Reading Wikipedia to Answer Open-Domain Questions*. CoRR abs/1704.00051, 2017.

[2] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. *Squad: 100, 000+ questions for machine comprehension of text*. EMNLP, 2016.

[3] Luong, M., and Manning, C. D. *Achieving open vocabulary neural machine translation with hybrid word-character models*. CoRR abs/1604.00788, 2016.

[4] Ramos, J. Using TF-IDF to Determine Word Relevance in Document Queries. Rutgers University, 2003.