Stack Overflow Question Title Quality

Tyler Chi

December 2, 2022

Abstract

Stack Overflow is a very popular platform for people to post technical questions and receive answers. Answers can then be up or down-voted, based on how helpful the users find the answer to be. A common workflow that happens among tech users is: running into an error, browsing Stack Overflow to see if others have hit a similar issue, identifying a post with a similar problem, and then browsing through the top rated answers to the question. In this process, it becomes clear that there are many questions on Stack Overflow which have received no answers, or answers with very few up-votes. The goal of my models in this project was to predict if a Stack Overflow question would receive an answer with at least one up vote, given the question's title.

Introduction

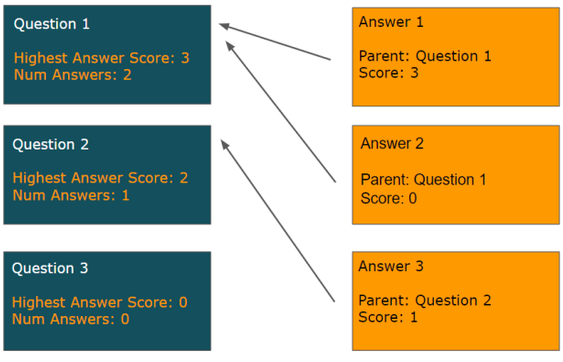
My goal for this project was to build a model that is able to identify questions that will perform well on Stack Overflow, as evidenced by the number of answers it receives, and the quality of the answers (judged based on the rating of these answers). I believe that such a model would be very useful to developers, to understand how to phrase questions that are more likely to get good answers on Stack Overflow. It will be interesting to see what type of patterns such a model would pick up – perhaps phrasing a question in a certain way is better. Or perhaps there are just certain topics that typically get high quality answers – regardless of how well the question is phrased.

Data Sourcing

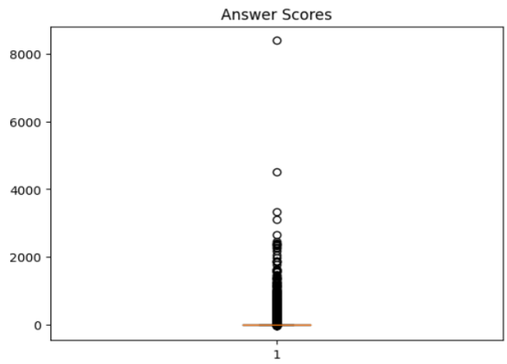
Given the huge amount of data on Stack Overflow, I wanted to limit the amount of data that I would train my models on. I found a dataset on Kaggle that had been filtered to only include Python questions from 2008-2016, as well as the associated answers[1] The timeframe of these questions was somewhat irrelevant, but the fact that they are all Python related would allow the model to focus on just Python questions, which might have different patterns from other questions.

Data Processing and Feature Engineering

By itself, the raw data was not ready for modeling. The data consisted of a questions table, and an answers table, with an id linking back to the parent. For each question, I queried the answers table for the children answers. Once retrieving the set of related answers, I calculated two numbers: the number of answers associated with the question, as well as the highest scored answer among these answers (see Figure 1). For questions that received no answers at all, I defaulted the highest score answer to 0, instead of leaving it as null.



[1] https://www.kaggle.com/datasets/stackoverflow/pythonquestions?select=Questions.csv



count	987122.000000
mean	3.028437
std	21.263246
min	-38.000000
25%	0.000000
50%	1.000000
75%	3.000000
max	8384.000000

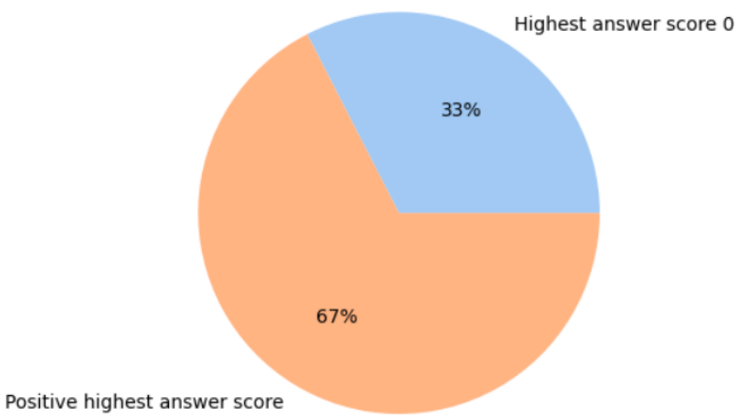
Figure 1: Answer score distribution.

Exploratory Data Analysis

After collecting and processing the data, the next step in this project was to explore the data, to choose a specific metric for a question's success. An initial look at the distribution of answer scores is quite surprising. Most answers have scores between 0 and 3, but there are quite long tails, specifically towards the higher numbers. One approach could have been to take the logarithm of all score values, to reduce the gap between low and high values. However, I thought that this would make the model results a little harder to interpret. I concluded that the best metric to use for success would be **whether or not a question received a score with a positive answer**.

Target Metric Analysis

After determining the **has positive answer** field for all of the questions, I took a look at the binary split:



Luckily, the split is not too heavily skewed towards either side, with 33% of the questions not having a positive answer, and 67% of the questions having a positive answer. That being said, in order to prevent the models from just over-predicting the positive class, one of two options will be used: **undersampling** the majority (positive) class, over sampling the minority (negative) class, or applying **class weights** to the training process. Given my limited compute resources, I did not want to oversample the minority class, as that would greatly increase training time. I also did not want to undersample the majority class, for fear of losing training data. Therefore, I decided to apply **class weights** to the training process. During the training process, this should affect the loss function by punishing the model more for false positive predictions, thereby weighting the negative classes more than the positive classes.

Modeling Overview

Over the course of this project, I tried a myriad of different modeling approaches. Some of these approaches were order agnostic, in that they looked at the words in the text, without taking the phrasing into account. These tended to not perform quite as well as the BERT based models, which took into account phrasing. This does make sense, as it is unlikely that there are some keywords that would make a question more successful – it is more likely how the question is phrased to be understandable, such that other users would understand the question and post quality answers.

Natural Language Models

All of the natural language models I made were either BERT or RoBERTA based. The usage of the pooled output from the BERT model is well documented, and is used often in sentiment analysis and text classification. The sequence output is also used in conjunction with convolutional layers, in order to pick up on phrases that might help with classification.

Model	Description	AUC (Test Set)	Precision	F1-Score
Bert + CNN	CNN layer on top of BERT sequence output, with class weights	0.64369	0.67157	0.80351
BERT + SVM	Using BERT pooled token and Support Vector Machine	0.5572883	0.70797	0.61391
BERT (Pooled Output)	BERT model passing pooled output into neural network, using class weights to prevent over-predicting positive	0.61305	0.72520	0.69613
RoBERTa Model (pooled output)	RoBERTa model passing pooled output into neural network, using class weights to prevent over-predicting positive.	0.62777	0.76105	0.61086

My baseline model for the natural language models was a neural network using the pooled output of BERT, which had an AUC of roughly 0.61. Of the other natural language models I experimented with, the BERT + CNN model as well as the RoBERTA model outperformed this baseline model.
Oh the other hand, the performance of the BERT + SVM model was quite poor, after adjusting the parameters a lot. Its AUC score was only 0.55. Perhaps further experimentation would have led to better results

Other studies have suggested that RoBERTa outperforms BERT for text classification [2]. I also tested using the pre-trained RoBERTa model instead of the BERT model, and found minor improvement (additional 0.01 AUC value).

From these 4 models, it appears that using the convolutional layers on top of BERT's sequence output yields the best results. This indicates to me that small individual phrasings within the title matter more than the overall representation of the entire title.

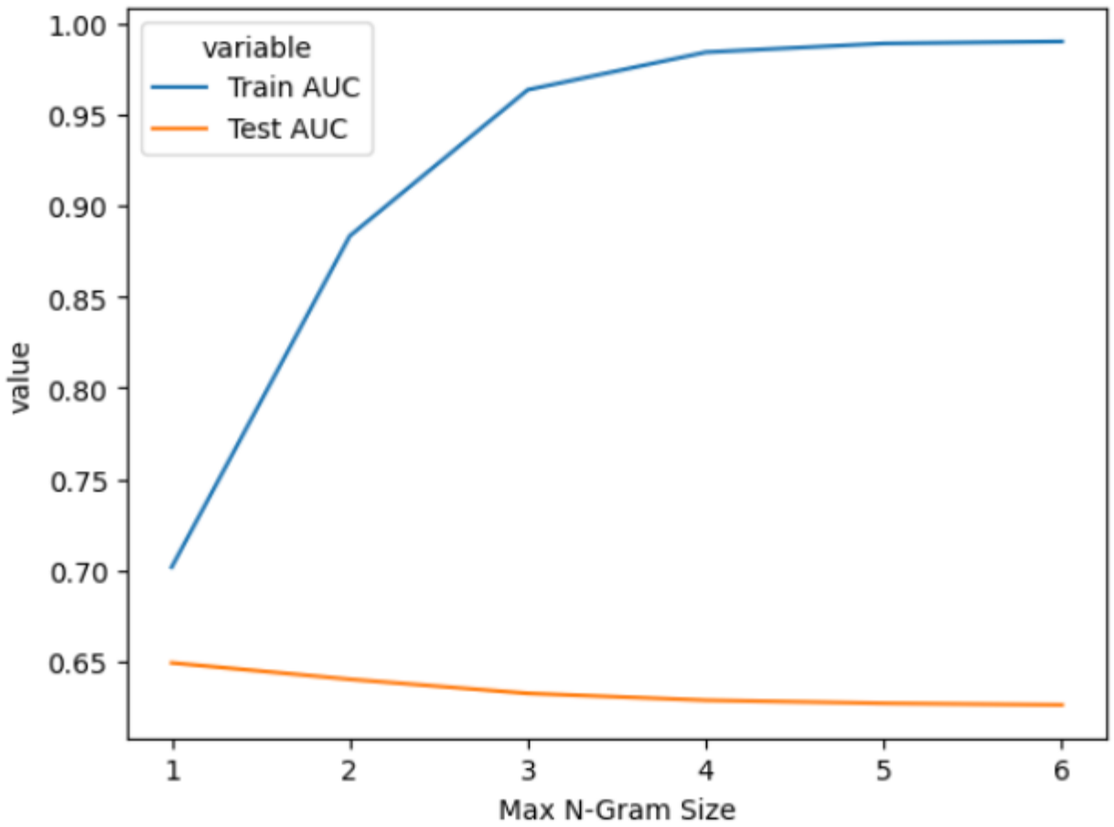
Bag-Of-Words Approaches

In the past, it was not uncommon for classification techniques to focus on the occurrence of words, rather than the order and meaning of the words. For this project, I tried three different models which were agnostic of the order that the words appeared in the question titles:

Model	Description	AUC (Test Set)	Precision	F1-Score
Naive Bayes	Multinomial Naive Bayes model, unigrams only	0.64938	0.72054	0.76378
TF-IDF + NN	TF-IDF outputs, fed into a neural network	0.50154	0.71083	0.01643
TF-IDF + SVM	TF-IDF outputs, fed into a neural network with SVM (hinge)	0.52384	0.68191	0.79853

Of these, it seems like the Naive Bayes model approach worked the best. In a similar project conducted by researchers in India, researchers also tried using a **Naive Bayes** model, in order to do sentiment analysis on Stack Overflow titles, to predict how well they would perform on the platform [3]. Purely using Naive Bayes, the researchers were able to achieve an accuracy of 72%, and suggested that using bi-grams would further improve the accuracy. In my own experimentation though, this was not the case. Adding n-grams to the default uni-grams seemed to progressively worsen the performance of the Naive Bayes model. In my own experiments, that was not the case though:

[2] https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9617586
[3] https://www.iosrjournals.org/iosr-jce/papers/Vol17-issue4/Version-1/C017411728.pdf



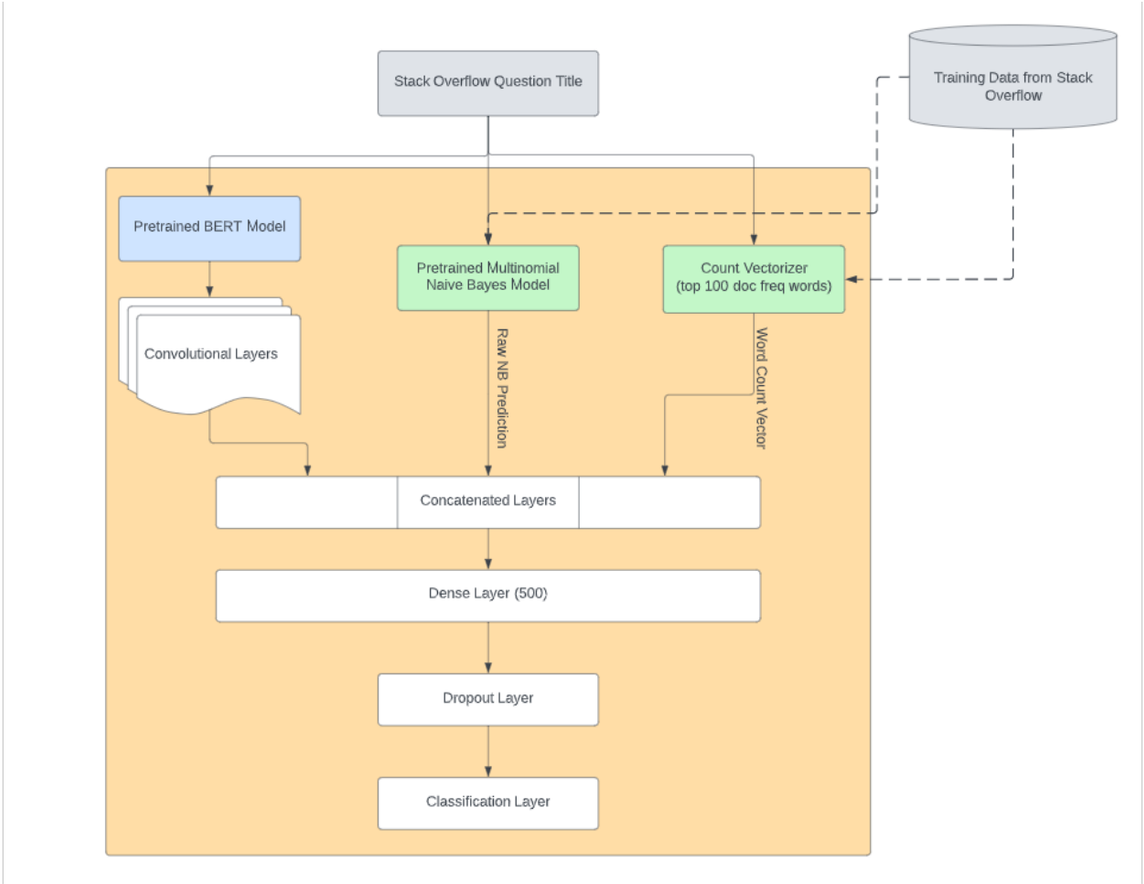
I found that the optimal test AUC came from having a max n-gram size of 1, basically just having unigrams. Increasing the max n-gram size seemed to slightly decrease the Test AUC, but dramatically increase the train AUC. This is likely because the count vectorizer was picking up on very specific patterns in the training data, that was only applicable to a handful of samples, and did not generalize well.

To get an idea of what words performed well in the eyes of the Naive Bayes model, I looked at the vocabulary of the count vectorizer, and ran each word individually through the model, to see the prediction it got. Among the top 10 words, the following stood out: pythonic, comprehension, question, tuples, regular, expressions, expression. I'd imagine that titles like this ask a very specific question, commonly about regex. Among the bottom 10 words, the following stood out: raspberry, pyinstaller, pi, jupyter, android, device, 500, odoo.

Combining the Approaches

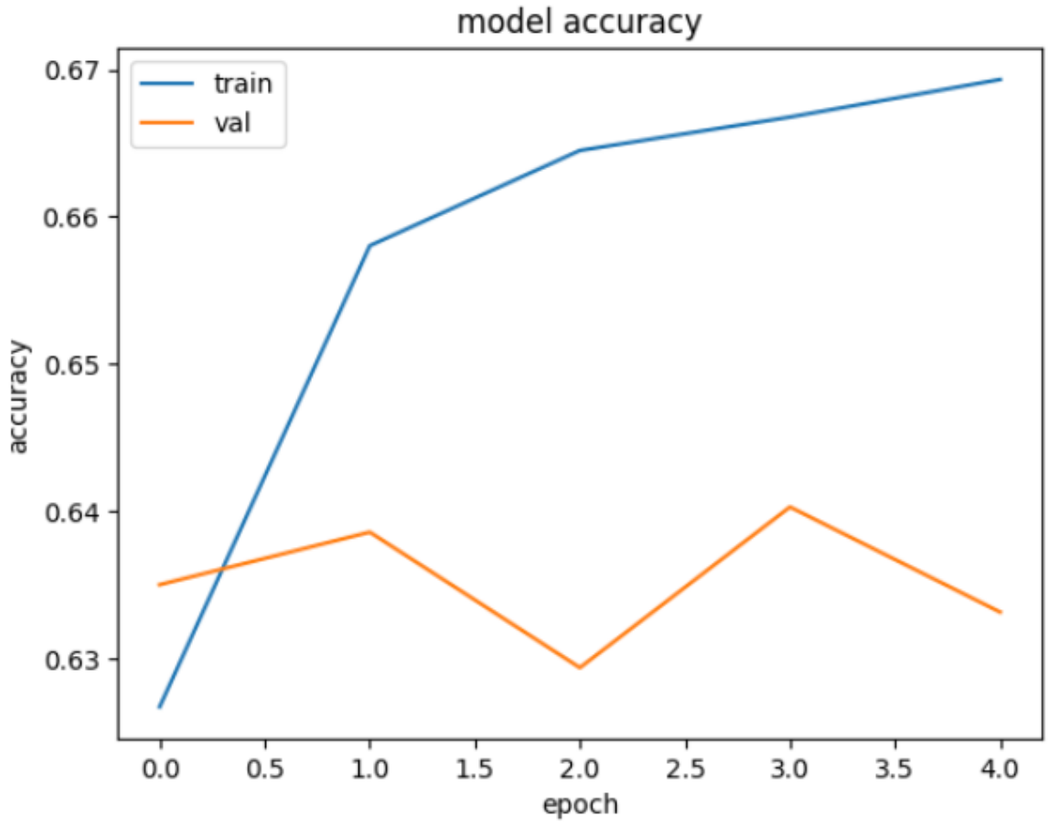
Given that neither the language based models nor the Bag-Of-Words models performed particularly well by themselves, I was curious to see how they would perform in conjunction with each other. I chose the best natural language model (BERT+CNN) and the best bag of words model (Naive Bayes) in order to combine into a single model. My hope was that the resulting predictions would be better than the models by themselves.

I did this by first pre-training the Naive Bayes model strictly on only the training data, in order to avoid data leakage. These predictions were then added as a feature into the combined models. Furthermore, given that the Naive Bayes model performed similarly well to the language based models, I believed that the presence of certain words were pretty useful features for models to train on. Therefore, I also included a word count vector as part of the input to the final model. Originally, I wanted to just use the words that were most polarizing in the Naive Bayes model (had the highest or lowest probabilities). However, I found that this was unfeasible, because of their low document frequency. So instead, I went with roughly the top 100 words ranked by document frequency, so that the word count vector would not be too sparse.



Best Model Analysis

This resulting model had an AUC of 0.66, which is marginally better than the Naive Bayes and the CNN model by themselves, both of which got 0.64. This implies that the presence of some words provided the model predictive power in classifying the question. The specificity of the resulting model was 0.54, meaning that it catches slightly over half ”bad” question titles.



Conclusion

One bottle neck for me during this project was my limited compute resources. Because of this, for all of my models, I froze the BERT model's layers, in order to reduce the number of trainable parameters. I believed that this was fine when comparing models (in that freezing these layers would result in the same dip of AUC for each model). If I had more resources, I would like to see the performance increase of these models if I allow the BERT layers to be updated during model training.

Overall, I was hoping for a slightly stronger AUC (perhaps upwards of 0.8). That being said, it is clear that the performance of a Stack Overflow question is influenced much more than just by its title. My findings in this project indicated that certain phrasings in conjunction with certain topics allowed the model to weakly distinguish well vs poorly performing Stack Overflow questions.

Further interesting studies would be trying to predict if an answer will receive upvotes, based on the content of the question. This would be useful for people posting questions, and they receive answers, but don't know which ones are high quality.

Related Links

Code Repository:
https://github.com/Tyler-Chi/w266_finalproject
Best Model Code:
https://github.com/Tyler-Chi/w266_finalproject/blob/main/Bert_CNN_CV_NB_Long.ipynb