QnA Matching for Stack Overflow

CS 6350 Summer 2018: Final Project Report

Siddhant Sahu, Ram Anand Vutukuru
sxs173732, rxv162130

August 1, 2018

1 Abstract

One non-trivial community that was constructed and which is growing rapidly on a simple idea of asking questions pertaining to wide variety of technological questions and seeking relevant answers is Stack Overflow. Stack Overflow has been the host for over 16 million questions by mid 2018. With such a large number of questions, more often than not, there would have been a similar question which would have been asked previously. With this project, we try to address that issue by identifying duplicate questions. We will be presented the data with questions posted using various tags such as python, java, javascript, etc. Our intention is to identify these repetitive questions based on these tags and reduce redundancy using a bag-of-words approach.

2 Related Work

One of the quickest ways to find a solution to a common question is most likely to be found in FAQ (Frequently Answered Questions) section of most organizations. FAQs are as nearly as perfect as it gets to answer open ended questions by multiple users online. The idea is to match an open-ended question to a pre-formatted FAQ. There is been a lot of research being done in this domain and there have been multiple approaches as well. To name a few, in *An Effective Similarity Measurement for FAQ Question Answering System*[5], the author addresses the problems by using a semantic knowledge base in order to improve its ability to match a particular question to its relevant answer. In *Ontology-supported FAQ processing and ranking techniques*[6], speaks about an enhanced ranking technique to show query relevant answers by accumulating FAQ information from heterogeneous sources and saves in on an ontological database. The irrelevant words are trimmed, and a partial or full word match is run in-order to get relevant results during the retrieval of FAQs.

3 Dataset

3.1 Raw Data

We use the StackSample[2] dataset, a dataset curated by Stack Overflow containing 10% of all Stack Overflow questions and answers on programming topics, and PostLinks data, obtained from StackExchange data archive. The raw dataset consists of four files:

1. `Questions.csv`: title, body, score, date, closed date (if applicable) and owner ID for each question
2. `Answers.csv`: body, score, date and owner ID for each answer.
3. `Tags.csv`: tags for each question
4. `Postlinks.xml`: Related posts for each post with link type (`duplicate` or `linked`)

3.2 Processed Data

The raw data has a lot of information that is not directly relevant to our main classifier module. The processed data has the following schema:

Dataset	Field	Type	Description
questions	id	Integer	Question id (primary key)
	answerId	Integer	Unique answer id for a question
	date	Timestamp	Date and time when the question was first asked
	text	String	The raw text of the body of the question
duplicates	id	Integer	Duplication id (primary key)
	answerId	Integer	Unique answer id for the duplicate question
	date	Timestamp	Date and time when the duplicate question was first asked
	text	String	The raw text of the body of the duplicate

3.3 ETL Pipeline

Spark[1] is unable to parse multi-line csv files even after the added support in Spark 2.2. Thus, for our experiments, we created an ETL pipeline as follows to convert the raw data to processed data.

1. Parse csv files using pandas[3] (pandas has excellent support for parsing csv files, including multi-line csv files) and load it into sqlite database. Also parse the xml file in python, create a csv file and load it into the sqlite[4] database (about 4.5 GB).

2. Since the process involves joins and the csv files are too big to fit in memory, we rely on Spark SQL to do the joins and save the final output to parquet files.

4 Methodology

4.1 Pre-processing

Duplicates are those questions that have the same answer as the original question. We consider these questions equivalent, semantically. In the table below is the distribution of answer classes against the number of duplicate questions linked for tag `python`. Almost 90% of the answers are linked to 4 or fewer questions. It is too expensive (computationally) and doesn't yield as much benefit to train the algorithm for all the answer classes. Therefore, we set our threshold to 20 answer classes while training the classifier.

Duplicates count	Number of answer classes
1-2	1653
3-4	198
5-10	121
11-20	34
21+	26

4.2 Feature Extraction

We use only the text column to determine duplicates. Limited on time, we choose only the feature extraction methods provided by Spark ML. The first step is to clean the body of the post – this means removing code blocks, html tags and urls/links. We use a bag-of-words model to determine the vector embedding for the words. We tokenize the text, remove stopwords and compute the tf-idf score for each word and limit the vocabulary to 500 words. We implemented these using Spark's Pipeline.

4.3 Modeling

We formulate the problem as a classification problem – with the task of classifying which answerID is the most likely answer for a given question. In simple terms, the input is a list of question-answer pairs (including duplicates) for a particular *tag*. For a new question, our model returns as output, a list of scores for each answer class in the training set.

We use the *One-vs-Rest* strategy to break down this multi-class classification problem into several binary classification problems. Spark's One-vs-Rest classifier does a great job but doesn't output probabilities of classes. So, we write a custom one-vs-rest classifier with Logistic Regression as the underlying algorithm to tackle this problem.

With popular tags, like `python` or `java`, with a lot of questions, the number of answer classes can be as high as 200 or 250 even after the pre-processing step. This presents an imbalanced classification problem for each underlying classifier. For example, some answer classes will have a few positive examples when compared with the rest of the dataset (negative examples). One solution is to take a random subset of the negative examples so that the overall ratio of positive to negative examples is balanced. However, this approach involves writing a custom function to subsample negative examples and is also troublesome while combining outputs from all classifiers. Thus, we resorted to using the weights approach while training Logistic Regression models. We specified weights for both positive and negative examples in each class that effectively imposes a much larger penalty on misclassifying the positive class. Specifically, if there are p positive examples (usually fewer than the number of negative examples) and the size of the dataset is n, we assign a weight of $\frac{n-p}{n}$ to the positive examples and $\frac{p}{n}$ to the negative examples. If there are k answer classes, we train k binary classifiers for each of the answer class and collect the coefficients of the positive examples across all classifiers to rank them.

4.4 Evaluation

We use stratified random sampling to split our data into training (75%) and test sets (25%). Stratified sampling proportionately allocates 75% of the questions *per answer class* in the training set and the rest in the test set. Spark doesn't have an inbuilt method for stratified sampling so we use a different approach using window and rank function to accomplish this.

Training set = Original questions + 75% of duplicate questions
Test set = Rest 25% of duplicate questions

With our current choice of training strategy, i.e. One-vs-rest, using cross-validation and hyperparameter tuning was out of the question, mainly because the model was too expensive computationally.

We use **average rank** to evaluate our classifier. Average rank is a learning-to-rank evaluation metric that calculates the average rank of the correct answer class in the list of all answer classes. Essentially, the closer this metric is to 1, the better.

The average rank, AR for the evaluation set is given by:

$$AR = \frac{\sum_{\forall i \in N} \text{Rank of correct answer}_i}{N}$$

where N = number of questions in the evaluation set.

5 Experiments

We ran a few experiments – classifiers for the most popular tags on stackoverflow: `python`, `java` and `javascript`. For our experiments, we chose the threshold on the minimum number of duplicate questions per answer class to be 20 (ideally, it should be around 12-13 but that takes too long to finish because of the

dramatic increase in the number of answer classes) and the number of features for tf-idf to be 300.

Tag	Average Rank
java	2.27
javascript	2.87
python	4.4

Setup: One master instance of **m4.large** and 6 task instances of **m4.large** machines.

Time: Average time taken per tag was approximately 30 minutes. We found this prohibitively expensive and didn't pursue the experiment using the entire Stack Overflow data. Recall that this is just 10% of entire Stack Overflow's data and we have already set a high threshold (20) on the minimum number of duplicate questions.

6 Future Work

There is ample scope of improvement of our model. Given more time and computational power, we'd like to explore along the following areas.

- Use word2vec embeddings instead of tf-idf scores.
- Formulate it as a ranking problem, that can be solved using the xgboost library.
- Learn useful phrases (also known as *collocations*) to see how they impact the overall rank.
- Use syntactic features by learning *part-of-speech* tags in text. In addition, use several word level and sentence level features.
- Explore deep learning and ensemble models.

7 Conclusion

In this project, we used very common text-mining techniques and classification algorithms in Spark to produce a good base model for detecting duplicates. Spark provides a very suitable platform to build apps that handle scale effortlessly with increase in data size. Ultimately, how well this algorithm leads a user to a pre-existing question-answer pair that solves their problem is the true measure of how successful the model is.

8 Contribution

- Ram Anand Vutukuru: 80% of his time was spent on ETL tasks, i.e. setting up pipeline to parse raw data, load it to SQLite database and create processed data. 20% on the code to build the classifier pipeline, especially the initial data cleaning phase.
- Siddhant Sahu: Most of his time was spent in building the pipeline, writing the custom one-vs-rest classifier and evaluation metric and running experiments on EMR.

  Both contributed equally in writing the report – initial sections were written by Ram and later sections by Siddhant.

References

[1] Apache Spark `https://spark.apache.org/docs/latest/`

[2] Stacksample dataset `https://www.kaggle.com/stackoverflow/stacksample`

[3] Pandas `https://pandas.pydata.org/`

[4] SQLite `https://www.sqlite.org/index.html`

[5] Juan, Zhong Min. "An effective similarity measurement for faq question answering system." *Electrical and Control Engineering (ICECE), 2010 International Conference* on. IEEE, 2010.

[6] Yang, Sheng-Yuan, Fang-Chen Chuang, and Cheng-Seen Ho. "Ontology-supported FAQ processing and ranking techniques." *Journal of Intelligent Information Systems* 28.3 (2007): 233-251.