Machine Learning Engineer Nanodegree

Capstone Project

Andres Mechali
June 8th, 2017

I. Definition

Project Overview

This project is taken from the Kaggle platform, and is currently available under the name Quora Question Pairs in this link: https://www.kaggle.com/c/quora-question-pairs. Quora is a platform where anyone can make a question, and other users give answers. According to Quora, they receive over 100 million visits every month, so it's very common that the question someone asks has already been answered by others. In order to make it easier for someone to find an answer, they use a Random Forest model to identify duplicate questions.

This project is about finding a model that can determine whether a pair of questions has the same meaning or not. The input data is a data set of about 400,000 pair of questions with a human provided label stating if they have the same meaning. As for testing, Kaggle provides a set of unlabeled pairs, which are then compared to their own human labeled results.

Problem Statement

The problem consists on analyzing different pairs of questions made by Quora users, and determining if they are basically asking the same thing. In this way, when a user asks something that has already been asked, he or she can be referred to a prior question which already has answers.

For solving this problem, I will start analyzing the basic aspects from the training set, and obtaining a benchmark predictor. Then, I will preprocess the data and create new features which may be helpful. After this, I will test some models and determine which is the best for using in this case. I will tune the parameters using GridSearchCV, and apply the chosen model to the testing set. Finally, I will upload the results to Kaggle and compare the log-loss obtained with the one from the benchmark. This log-loss should be smaller, which means that the model is better than a naive predictor, and gives some extra information about the data.

Metrics

The metrics that I will use for measuring the performance of the models will be the *log-loss*. This is a very commonly used metrics in models where the output is the probability of a binary outcome, which

is the case of this project. This metrics penalizes wrong classifications depending on how confident the model was about them. Specifically, a prediction with a greater level of confidence is more penalized when wrong.

Another reason for choosing *log-loss* as a metrics is because it is the one used in the Kaggle platform to evaluate the predictions. So, by using this while training, I can compare the results with other Kaggle users.

The formula for this metric is the following:

$$LogLoss = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot log_e(\hat{y}_i) + (1 - y_i) \cdot log_e(1 - \hat{y}_i)]$$

II. Analysis

Data Exploration

In the training data set we can find the following information:

- qid1: Unique ID of question 1
- qid2: Unique ID of question 2
- question1: Content of question 1
- question2: Content of question 2
- is_duplicate: A label stating if both questions are the same (1) or not (0)

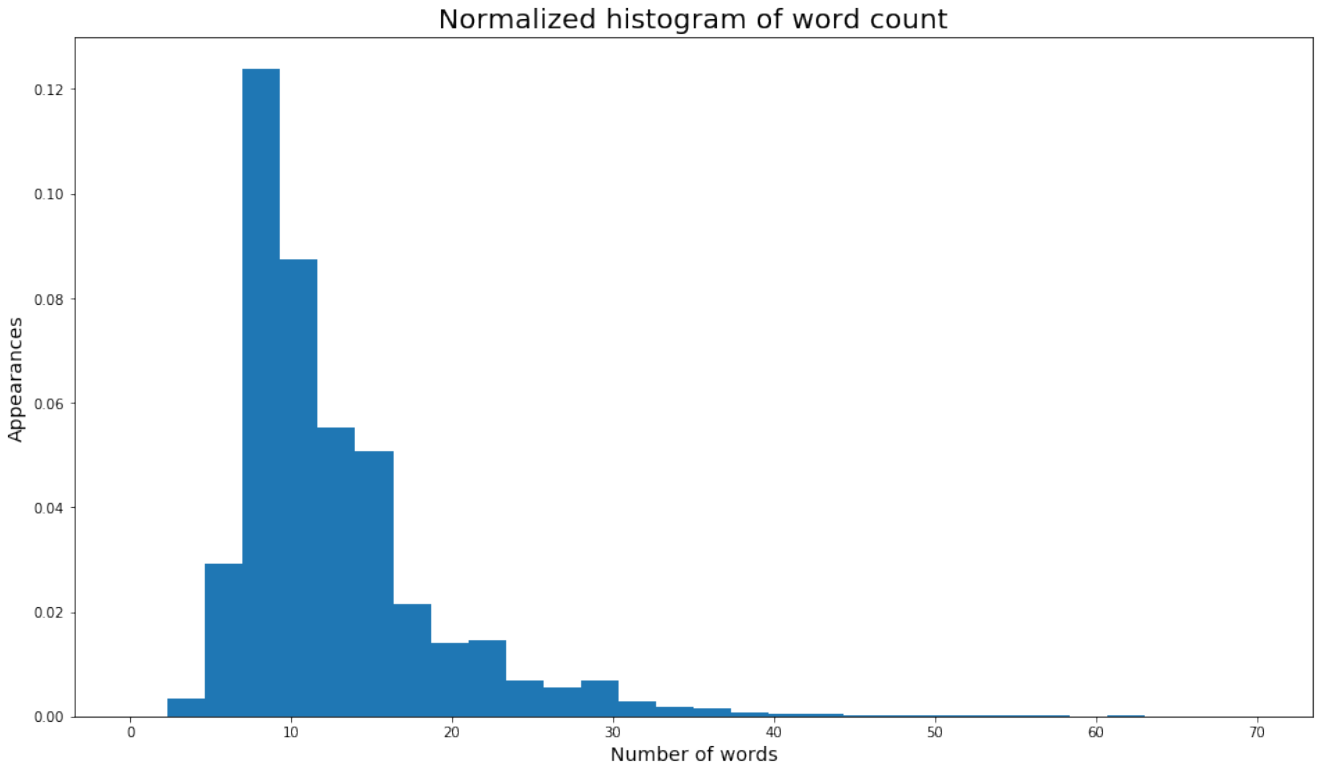
We can also find the following information about this data set:

- Number of rows: 404,290
- Duplicate pair of questions: 149,263
- Not duplicate pair of questions: 255027
- Unique questions: 537933

Exploratory Visualization

Now I will do a brief analysis of the questions in the training set. I will use the Natural Language Tool Kit (nltk) for this, and I will display an histogram to see the frequency of the lengths of the questions.

As we will see here, most of the questions are composed from about 8 to 12 words. There are some of them which can have even more than 50 words. I will not exclude these outliers, because when comparing lengths I will use the absolute value, and not a normalized one. Hence, outliers should not affect my outcome.



Algorithms and Techniques

For this project I will evaluate 3 algorithms: AdaBoost, Random Forest and XGBoost. The reason for this is that they are all ensemble methods, which work well for problems where the features are weak learners. In this problem all the features will be so, because there doesn't seem to be one that is highly correlated with the meaning of the question. I will now provide a brief explanation of each of these models.

AdaBoost

This method uses a *base algorithm* (by default, Decision Trees) and applies it repeatedly, each time improving it. In order to do this, on every iteration it increases the weight assigned to each of the data points were it previously made a wrong classification. After *n* times, the final model is made from the weighted sum of all the learners. These weights are updated based on the error rate of the classifier, which is the total number of misclassifications divided by the training set size. The weight for a given classifier grows exponentially as the error approaches 0. If a classifier has a 50% accuracy it is the same as random, so the weight assigned is 0.

Random Forest

Here, what is called a *bagging* approach is used. To build a decision tree, the algorithm starts with a random subset of the training samples. The final prediction will be an average of individual estimators.

The idea is that each of these estimators provides a low bias, which is desired, but high variance. Averaging them keeps the bias low while also lowering the variance.

XGBoost

This algorithm, which name stands for *Extreme Gradient Boosting*, is a model that takes *weak* predictive models and ensembles them to create a stronger one. Any model that works better than random can be used. This is exactly the same as in AdaBoost. The difference is the way in which each algorithm updates the weight of each predictor. In XGBoost, this is done by taking the gradient of the loss function. In this way, the algorithm updates the weights taking a step in the direction that minimizes the loss function.

I will try the 3 models and determine which has the minimum *log-loss*. After that, I will tune this model to find the best possible parameters for this problem.

Benchmark: Naive predictor

As a Benchmark for testing the effectiveness of this model, I will use a predictor that evaluates if two questions are equal just based on the probability of this happening. I will assign this same probability to every pair of questions. I will upload this result to the Kaggle platform an evaluate the log-loss obtained. This will be my benchmark to see if I can obtain a better model for this.

The log-loss obtained was **0.55411**. This will be used as a benchmark, and any model with a lower value will be considered better.

III. Methodology

Data Preprocessing

I will now pre process the raw data, for which I will need a series of steps stated below:

- **Removing stop words**: The first step will be to remove all words that are considered not to give valuable information about the content of the question. Examples of this will be you, have, the, to, etc., and punctuation.
- **Applying stemming**: In order to consider the same word car and cars, I will apply a stemming algorithm that keeps the root of each word.
- **Tokenizing**: I will split questions into list of words, so that I can loop and compare them easily.
- **Tf-Idf**: With this I will consider the importance of a word related to it's frequency, instead of considering them equal. The fact that 2 question share a word that appears often will be less important than if they share a rare word.
- **DiffLib**: This is a Python library that will help me finding differences between 2 strings. Specifically, I will be using the SequenceMatcher method.

By doing these steps, I will get a new set of features, which will be composed of:

- **shared_weights**: a measure of how many words both questions share, weighted by the tf-idf value of each word.
- **shared_count_scaled**: it will represent the normalized amount of words that they have in common.
- **len_dif**: the difference between the lengths of the questions.
- **z_match_ratio**: the value obtained from the SequenceMatcher method.

I will also check if these features seem relevant to classify between equal and unequal questions. For this, I will use histograms.

Implementation

After applying all the steps stated above, I got a new data set with four features. The following chart shows the first rows of this data set.

	shared_weights	shared_count_scaled	len_dif	z_match_ratio
0	0.950408	1.0	0.2	0.926829
1	0.652899	0.8	1.0	0.661871
2	0.517698	0.6	0.2	0.439394
3	0.000000	0.0	1.0	0.086957
4	0.237645	0.4	1.0	0.365217

Having this new data set, I used the train_test_split method from sklearn to split it between a training and a testing set, consisting on 80% and 20%, respectively.

Now I compared the 3 models chosen before, to see if any of them got a significantly smaller *log loss* than the rest. The results were the following:

- AdaBoost: 0.68
- Random Forest: 0.64
- XGBoost: 0.52

Refinement

As the differences were not too high, I tried some tuning on the parameters for each model, using GridSearchCV. The new results obtained were:

- AdaBoost: 0.54

- Random Forest: 0.50
- XGBoost: 0.49

I chose XGBoost, not only because it got the best log loss, but also because it is the fastest to train.

Before tuning even more the algorithm, I made an adjustment on the testing set. Based on this article from Kaggle and it's comments, I know that the distribution of positive and negative cases in the test data is not the same as the one in the training data: https://www.kaggle.com/davidthaler/quora-question-pairs/howmany-1-s-are-in-the-public-lb/run/1013730. In the training data we have around 37% of positive cases, while the test data has around 16.5%. So for having a better score from Kaggle I need to re balance the data. With this new data, I tried some extra tuning, and got a log loss of 0.35. The parameters used in this final model were the following:

- objective = 'binary:logistic'
- eval_metric = 'logloss'
- n_estimators = 500
- max_depth = 4
- reg_alpha = 0.5
- learning_rate = 0.3
- gamma = 0
- reg_lambda = 1
- min_child_weight = 1

IV. Results

Model Evaluation and Validation

Having already decided the model to use, I applied the same transformations to the provided testing data set, in order to get the new features and be able to make predictions. This provided a different input to test the model, besides the one obtained from the *train_test_split* method. This new testing set doesn't have the labels in order to test it by my own, but it is possible to do it on the *Kaggle* platform. This is good, because this data is not used during the training set, so it provides a very robust indicator.

I think that my model could serve as a first approach for analyzing a problem like this one, but could not be used as a general setting, because it is not as accurate as it should be. In order to be useful for a company to make predictions, *log loss* should be much lower, so that results can be trusted more.

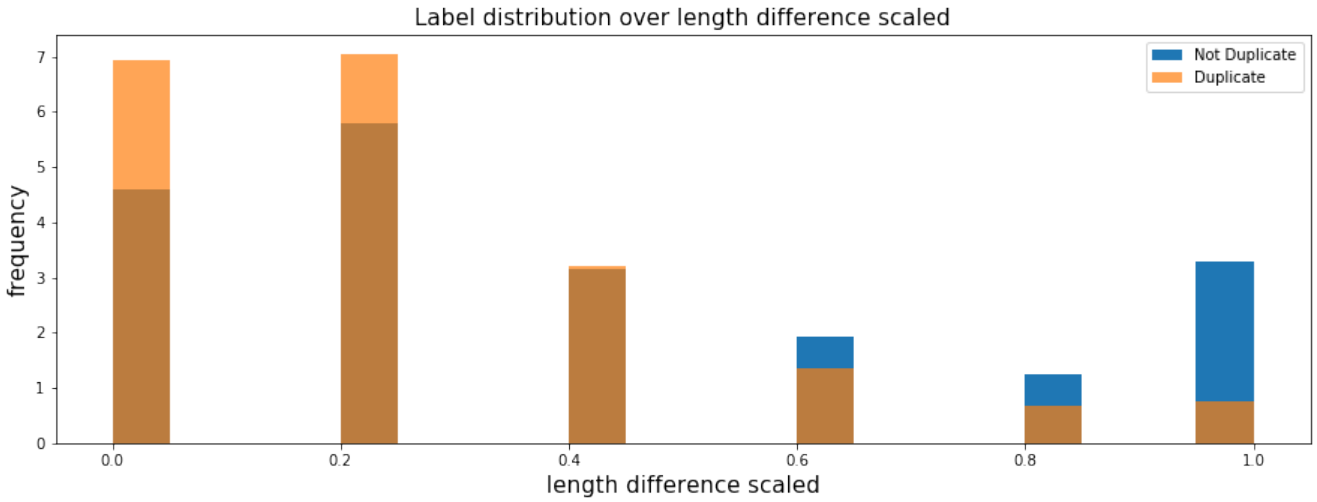
Justification

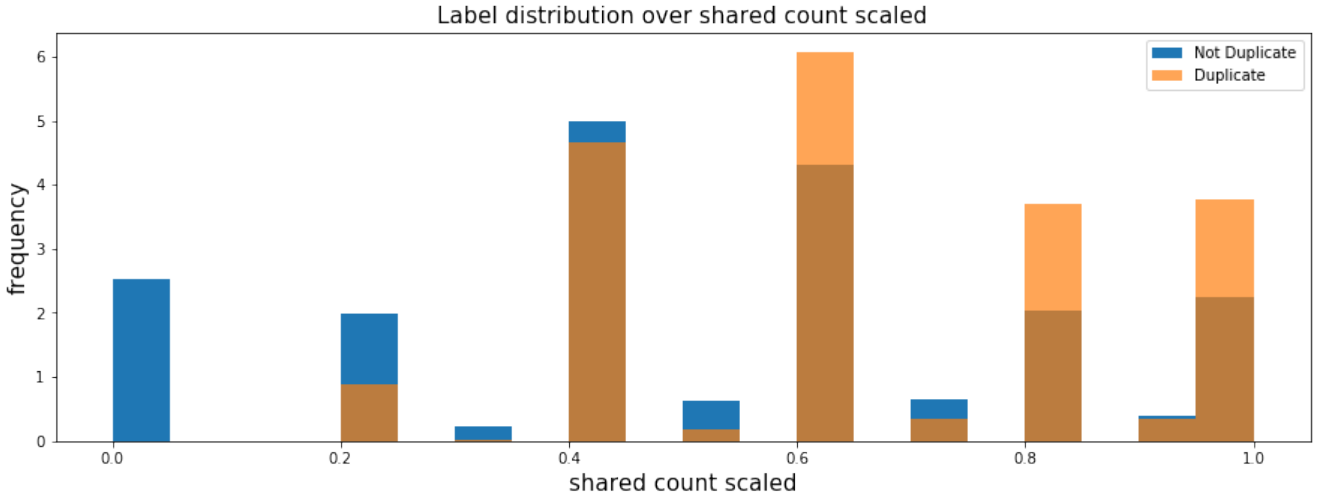
The final score obtained in the *Kaggle* platform is **0.39205**, which is an improvement over the benchmark of 0.55411. This means that when tested with a totally different input, the results are very similar from the ones obtained before. Still, it is not low enough to be considered an optimal solution to the problem. This value should be improved in order to make significantly good recommendations.

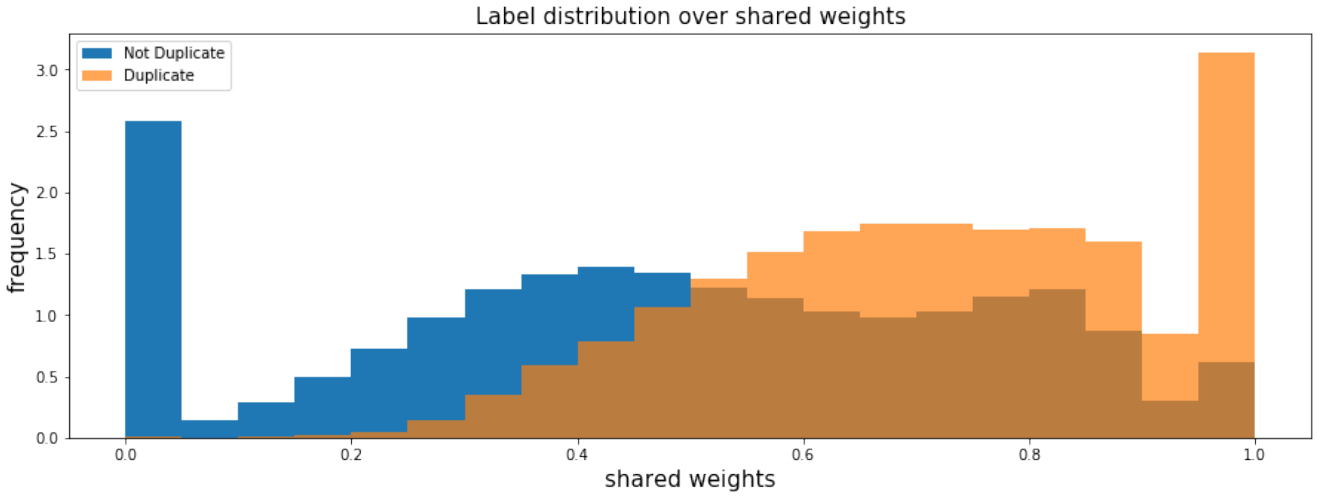
V. Conclusion

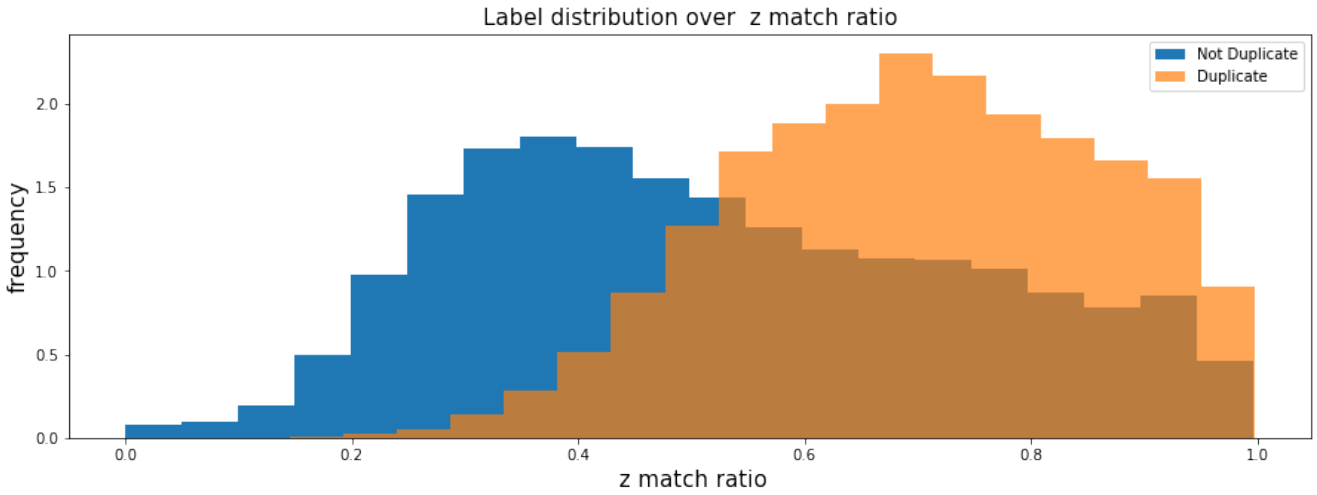
Free-Form Visualization

In this section, I will show the distribution of each of the four new features, for both labels. We will see graphically that all of them behave different for duplicate questions.

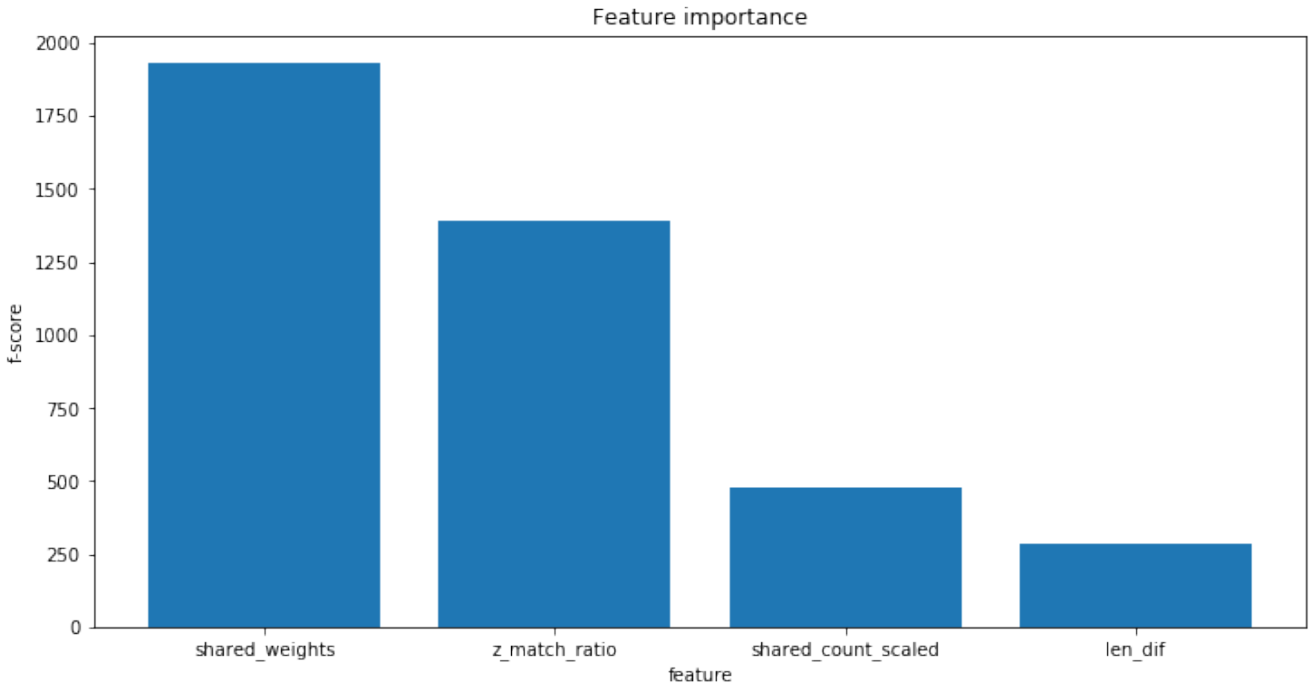








In the following chart, I will show the f-score for each feature. This gives a measure of the relative importance that they have.



We can see that *shared_weights* and *z_match_ratio* are the most representative, and *len_dif* and *shared_count_scaled* are not so important.

Reflection

For this project, I started analyzing the basic aspects of the data set, and getting a naive predictor to use as a benchmark and see if my model would represent an improvement. After this, I pre processed the training data, and tested that each new feature showed graphically a different distribution between duplicate and non duplicate pair of questions.

Having obtained this, I compared the three models which I believed could be useful for this problem. Given that performances were similar when tested with default parameters, I used GridSearchCV for tuning each of them, and comparing the new results. I chose XGBoost for it's good performance, and it's fast training.

I finally pre processed the testing data, used my chosen model for prediction, and tested the performance within the Kaggle framework. I got a better result than the benchmark.

I found this problem to be very interesting, given it's high complexity. Natural language processing is involved, which is a very hard topic. The fact that there are a lot of different ways to ask the same question, some of them very different between each other, makes this problem a very hard one for obtaining a good result. Also, it's not easy to use neural networks in these kind of problems, because each word could be seen as a feature and there are thousands of them.

Improvement

I think there are different ways to improved this algorithm. For example, every word should be contrasted to it's synonyms, in order to group together words that are now consider different but refer to the same. Another improvement could be giving words that start with capital letters a greater weight for TF-IDF analysis.

The problem could also be treated from a different approach, which I would have used if I knew how. As seen in this paper (https://arxiv.org/abs/1408.5882), Convolutional Neural Networks can be used for sentence classification. I think that this approach can yield much better results, given the fact that deep learning catches much better aspects where humans are good at, like it does with image recognition.