CIS 700-003 Final Project
Identifying Semantically Identical Question Pairs From Quora

Dean Fulgoni DFULGONI@SEAS.UPENN.EDU
Matthew Goodman GOODMANM@SEAS.UPENN.EDU

1. Introduction

Our project is in the domain of natural language processing and natural language understanding. The broad problem we address is how to judge the semantic intent of language irrespective of its syntactic structure, specifically on individual sentences. What constitutes the meaning of a sentence is subjective and never fully tangible. Regardless, it is clear that sentences convey intent, such that a human could tell if two sentences are semantically identical, and even the degree to which this is true. How this is done by the mind is difficult to understand in itself, which makes modeling such a process quite challenging.

Observe the following two sentences:

1. Is breast cancer preventable?

2. How do I prevent breast cancer?

Looking at the two questions, we can quickly determine they do not have the same intent. Sentence 1 asks whether breast cancer is preventable. Sentence 2 presupposes that breast cancer is preventable, and asks what measures can be taken to prevent breast cancer. The intent of the questions is clearly different. Natural language processing studies in part the mechanisms by which our brains determine that the two questions are different and to what extent, and how these mechanisms can be reproduced in computers. A nave approach would be to look at the content words (i.e. nouns, verbs, etc.) and see if they match up between the two sentences. Both contain the phrase ”breast cancer” and the prefix ”prevent”. Using this measure would produce the wrong classification because it ignores how syntactic structure shapes semantic meaning. Clearly, the grammatical structure of language shapes the meaning, making the problem of understanding semantic intent nontrivial.

Furthermore, this problem is difficult because of the sloppy nature of written text, especially when found on the Internet. Grammar is often incorrect or nonstandard, and spelling errors can be frequent. While our brains are highly adept at knowing when grammar matters, a computer does not have the same intuition. A great challenge is designing an algorithm that knows when to closely apply the structure of language and when to ignore it.

Problem Statement

For this project, we are competing in the Quora Question Pairs challenge on the Kaggle competition platform. The goal of the challenge is to accurately classify whether a pair of questions ”share the same intent”. Thus, we attempt to design a classifier that perceives identical intent of questions exactly as a human would.

Background

This task has been explored by the engineering team at Quora [1]. They tried three different approaches, with models such as Long Short Term Memory (LSTM) neural networks. Their pipeline involves initial features as embeddings of the sentences in vector spaces, and later representation layers derived from these embeddings. They then looked at metrics such as the distance and angle between these representation vectors. While we found this approach interesting, we wanted to try a different direction: to be creative with features and less focused on model selection.

Intuition

To accurately identify semantically identical pairs, we need a system that grasps exactly what makes two questions have the same intent. Since this is ultimately subjective, we turned to the labeled examples to understand how questions could be similar or different. In many cases, it was obvious: Duplicate questions might differ by only an adjective or article, while non-duplicates could mention wildly different topics and share no common structure. However, many examples seemed deliberately constructed to make simple systems fail. Observe the following examples:

1. Does Fab currently offer new employees stock options or RSUs? Does Uber currently offer new employees stock options or RSUs? [Not Duplicate]

[1] https://data.quora.com/Applying-Deep-Learning-to-Detecting-Duplicate-Questions-on-Quora

2. What is the ideal life after retirement? What's life after retirement? [Not Duplicate]

3. Why my question was marked as needing imrovement? How can I ask a question without getting marked as 'need to improve'? [Duplicate]

The pattern in example (1) is common. A sentence may have nearly identical structure, but a single noun or named entity mismatch can completely flip the label to non duplicate. In (2), we see that even a difference of an adjective can mean non duplicate, yet for most examples an adjective difference will not be consequential. Finally, (3) shows how a fairly ambiguous pair can be labeled as duplicate, and further that grammar and spelling may not be correct, all in the same training instance!

From this investigation, we began to understand what was necessary of a competent system. It should understand identical segments of semantics and syntax as well as possible, because this is the basis for most examples. However, it should have ways to flag particular differences in sentences that are consequential, such as unmatched nouns, adjectives, verbs, and named entities. It only through a combination of these feature ideas that our system can attempt to perform well on this difficult task.

2. Data

The dataset is a large list of actual question pairs from Quora, a community for question answering. It was acquired from Kaggle [2], an online source for machine learning competitions. Quora and Kaggle provided the training and test data in csv files, so the data was already fully cleaned and easily readable into a DataFrame.

The training data consists of 404,290 rows, with six simple fields: the observation id, an id for each individual question, the raw text of each question, and a binary indicator of whether or not the question pair is deemed identical in intent, called *is duplicate*. Of these rows, about 37% were labeled true for *is duplicate*. The questions were usually one sentence and ranged from very brief to long-winded, with an average character length of 59.8. All questions were in English, but were not free from incorrect grammar or spelling errors. They also contain many non-ASCII characters.

Important to note is that the *is duplicate* labels were given manually, according to human experts. Quora acknowledges that their labels may have some errors because of this. Skimming through the question pairs by hand, it is clear that some examples are much more definitively duplicates than others. There are even cases of blatant mislabeling. All of this adds an additional challenge to the learning task, as our system is always given a hard label of zero or one for each question pair, despite differences in ambiguity.

[2]https://www.kaggle.com/c/quora-question-pairs

3. Data Processing

While we didnt have much to do for preprocessing, it was necessary to get sentences in a useful format for deriving features. Common approaches include tokenizing the sentences into individual words and stemming. Our features rely on more than tokenization, as we need more information such as the part of speech of each token, as well as syntactic dependencies between each of the words. Thus, we used Stanford CoreNLP [3], a software package that allows us to run state of the art versions of the necessary algorithms. Using this software, we ran an annotation on each sentence to produce the following outputs: tokens, part of speech parse, basic dependency paths, and recognized named entities.

Speed and Efficiency

Although this processing is necessary for our features, it became clear that the runtime of generating this output could be a limitation. We found that, even for the fastest of Stanfords parsers, a local machine takes at least 0.08 seconds to process a pair of sentences for the dependency parsing computation alone. The training set has over 400,000 examples, which means the output would take about 10 hours to generate. Even worse, the testing set has over 2 million examples, and this would mean our feature generation would take an infeasible amount of time.

To solve this issue, we decided to build a single large file containing all the necessary data for feature generation. The final structured output is a nested map with a key for each question pair. Each pair contains two sentences. Each sentence maps each of its token indices to its word, that words part of speech, and a list of token indices of dependencies pointing from that word index to another. Each of these dependency indices is also mapped to the basic dependency type. Finally, each sentence also has an associated constituency (part of speech) tree, which is hierarchical. This needed to be built by a custom regular expression function that parsed through the original Stanford constituency parse output, which was simply text and not a tree.

Once these files were created, feature generation could now be done each time on the order of minutes instead of hours. However, even generating these files once was challengingly time intensive, as the training and test sets would take approximately 10 and 50 hours respectively. Thus, we

[3]https://stanfordnlp.github.io/CoreNLP/

turned to Amazon Web Services [4] to aid the computation.

Amazon Web Services

We set up an AWS c4.4xlarge compute optimized instance to handle the sentence annotation preprocessing. The major bottleneck in working with AWS was that the instance only had 15 GB of disk space. Before any other work, 8 GB of disk space was already taken for the distribution of Python with necessary libraries, Stanford Core NLP library, and the Stanford Shift-Reduce Parser which provides a 3x speedup over the generic Stanford parser. While the test data set file was only 200 MB, the trees and parse tokens that Stanford NLP generated were took up too much disk space even when compressed with gzip. Thus, we couldn't preprocess the entire test set all at once. The solution was to break the test set into 5 parts, each part approximately the same size as the training set, process them individually on the AWS server, and scp them back to our local development environment. Preprocessing the training set took 10 hours, while preprocessing the test set took 40 hours. We also used the AWS instance to feature generation for the Tree Kernel based features.

4. Features

One primary challenge of this task is feature engineering, so much of our efforts were directed towards this end. We hoped to capture semantic alignment of sentences, syntactic alignment of sentences, differences in unaligned words, part of speech differences, and specific words that might be very important. Individually each of these features has clear weaknesses, but together they cover a wide range of identifying and discriminating semantic intent.

4.1. Semantic Alignment

A first important feature is to quantify how similar two questions are by aligning word pairs across them that have the same intent. This informs us of how and to what extent the two questions are related. To this end, we closely implemented a version of a monolingual textual aligner by Sultan et al. [1]. With this algorithm, the theory behind alignment can be expressed simply: Two words will be aligned if they share both known semantic intent and contextual similarity. Also, no word in either sentence can be aligned with more than one word in the other. While this is an obvious limitation, its simplicity is rewarding.

4.1.1. Word Similarity

In order to be a candidate aligned pair, two words must be deemed similar in meaning. We used three methods of similarity verification, each providing a positive similarity score. First, if there is an exact match of the two words, then a perfect score of 1 is given. If the words are not identical, we refer to two well known lexical resources to attempt to identify pairs that are synonyms or paraphrases. The first is WordNet, a handcrafted lexical database containing known synonym pairs [2]. The second is the Paraphrase Database (PPDB), a scaled version of WordNet that has a wider coverage of synonyms but with less precision [3]. We used a middle size (XL) of the PPDB's lexical paraphrase package. For WordNet, the resource was traversed for all nouns, verbs, and adjectives, and a JSON file mapping each word to all its synonyms was created. A similar approach was used for PPDB. If the word pair exists as synonyms in WordNet, a similarity score of 0.9 is given. If not, but the pair exists in PPDB, a similarity score of 0.7 is given. If none of these methods of similarity match, the pair is given a score of 0.

We also considered going beyond lexical resources by using distributional similarity measures, such as Word2vec [5], which can reveal how often two words appear in the same linguistic context. However, due to the nature of our task this is not particularly useful. We want to very heavily weight differences in semantic meaning between individual words, because those differences can completely change the perceived intent.

4.1.2. Contextual Evidence

The second component of an aligned pair is sharing contextual similarity. This can be achieved in two different ways.

The first is by syntactic alignment, based on basic dependency paths. Following Sultan et al., given two sentences S and T, two words $s \in S$ and $t \in T$ are a candidate for an alignment pair if exists an $r_s \in S$ and $r_t \in T$ such that

1. $sim(s,t) > 0$ and $sim(r_s, r_t) > 0$

2. $dependency(s, r_s) \sim dependency(t, r_t)$

Not only does (s,t) need to be semantically similar, but there needs to be another pair (r_s, r_t) between S and T that are similar, and s is related to r_s in the same way t is related to r_t. In our implementation, the relatedness has to be an identical dependency type, and no equivalence class of dependencies are used for simplicity. Note that these dependency relations can go in either direction for both pairs, allowing for 4 different orientations that can be used as context.

[4]https://aws.amazon.com/

[5]https://en.wikipedia.org/wiki/Word2vec

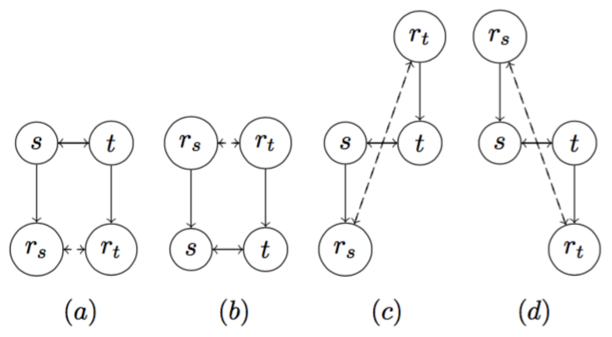

Figure 1. All orientations of parent-child dependencies, from Sultan et al. Double arrows indicate word similarity, where single arrows indicate dependency.

The second way is by the textual neighborhood. We now look at the $(3, 3)$ window of tokens surrounding each word in a word pair. A context is created for each combination of content word pairs (with stopwords now removed) in this window. Stopwords were obtained from the Natural Language Toolkit (NLTK) [6]. This method is used to improve recall on getting sufficient context, as dependency paths are limited by the accuracy of the parser.

4.1.3. Alignment Algorithm

The final alignment algorithm returns the set of index pairs between sentences S and T that have been semantically aligned. It achieves this by calling alignment helper functions that sequentially build up the matched pair set. In our algorithm, we use two of these functions, $cwDepAlign$ and $cwTextAlign$, which make use of the dependency context and textual neighborhood context, respectively. The dependency alignment is performed first, because it has higher precision, as we remember that once a pair (s, t) is matched, neither s nor t can be matched in later functions.

The procedure of both helper functions are outlined as follows. Iterate over all token indices $i \in S$ and $j \in T$. Proceed only if neither index is already aligned and $sim(s_i, t_j) > 0$. Generate the context of (i, j), either from dependencies or textual neighborhood, depending on the function. Sum the word similarity scores for each pair in the context. If this score is positive, we know (i, j) should be aligned, as it meets the relevant criteria. However, we add (i, j) to the aligned set in the descending order of the scores for each (i, j). This way, higher contextual similarity scores will take precedence over previous ones, in the case of competing alignment for either index.

[6]http://www.nltk.org/

4.1.4. Semantic Alignment Features

The alignment algorithm returns all index pairs that have matched between questions S and T, but this information still needs to be encoded into features. Because we are interested in overall semantic similarity, the one feature was percentage semantic similarity. This is the number of aligned tokens in a sentence divided by its total number of tokens. This was done individually for each sentence. Additionally, one more feature was created as the combined percentage semantic similarity across both sentences. These three features give us an accurate estimate of how well the sentences were aligned.

4.2. Part of Speech

As discussed in the introduction, semantic alignment is an important base feature but needs to be complemented by features that emphasize the differences between two sentences. To achieve this, 24 features based on unmatched part of speech words were created. There are 4 for each of the following part of speech tags: noun, adjective, verb, personal pronoun, wh-pronoun, and cardinal number. Of these 4, there are 2 types that are repeated for each sentence. The 2 unique feature types are the following:

1. Number of POS words in that sentence but not in the other.
2. Percentage of POS words in that sentence but not in the other.

The categories were selected based on their importance in classification, which can be observed by manually investigating many question pairs and their labels. For example, some question pairs were identical in every respect except for a mismatched noun, which was the sole reason for the non duplicate label. A decision was made to keep these features separate for the first and second sentence. In some duplicate questions, one sentence may be completely entailed by another and thus not have any mismatched words. We believe giving more features would help in these edge cases where a combined mismatch

We also note that we only consider unmatched words in either sentence from those not matched by semantic alignment. Clearly, if aligned, these words are shared across the sentences and should not be counted by this method, which intends to highlight differences.

4.3. Named Entity Recognition (NER)

In addition to unmatched part of speech world, we separately consider unmatched named entities between the sentences. In many training examples, a non duplicate label was given for the sole reason of a mismatched named entity. Named entity labels are given from the Stanford parser in a similar way to part of speech tags. The 2 derived NER features used were just the number of named entities that appear in each sentence but not in the other. We believe this simple feature can be very effective for improving classification, as it is almost never the case that questions are identical if they do not mention the same named entities. Because some sentences seemed to have many named entities (even above 10), we cutoff this feature to have a maximum value of 3, as that gives enough relevant information.

4.4. Sentence Length

Other simple features added were derived from the lengths of the two sentences. The first is the absolute difference in sentence length according to the number of tokens, and the second is this difference divided by the average length of the sentences. While perhaps not particularly useful alone, these features could be telling in combination with others.

4.5. Tree Kernels

As part of our search for unique features to encode syntactic and semantic similarity, we found that so called Tree Kernels had been shown to be highly effective in augmenting SVMs designed to solve problems in question classification, question answering, Semantic Role Labeling, and named entity recognition. The broad idea behind the theory is that syntactic features must exist to learn semantic structures. To do so, we us design kernels that map from tree structures (e.g. parse, dependency, etc.) to scores that measure the syntactic similarity of the two trees.

4.5.1. Core Theory and Definitions

A tree kernel is a function: $f : (tree_1, tree_2) \rightarrow \mathbf{R}^+$

A tree is defined recursively as a set of nodes, each of which has a value and a set of children, each of which are trees. A node is a ”leaf” if it has no children. A node is ”preterminal” if all of its child nodes are leaves. The ”production” of a node is the subtree containing the node and its children. The production of two nodes are the same if the nodes have the same value and their children have the same value. Because we work are working with ordered trees, the children must have the same values in the same order to be part of the same production.

Key to the effectiveness of tree kernel features in solving problems is the choice of tree structure. Moschitti shows in [4,5] that different tree kernels are effective for different choices of syntax trees in solving different classes of problems. We used Stanford NLP to generate parse trees from the input sentences. A parse tree is an ordered tree that represents the syntax of a sentence according to some context-free grammar. There are two main classes of parse trees: constituency and dependency. A dependency parse trees is the tree structure generated by the dependencies of a sentence. While dependencies can be highly useful in generating features, as described in Section 4.1, Moschitti found that constituency parse trees were most effective in solving classification related problems. A constituency parse tree breaks down the tokens of sentence into their constituent grammars.

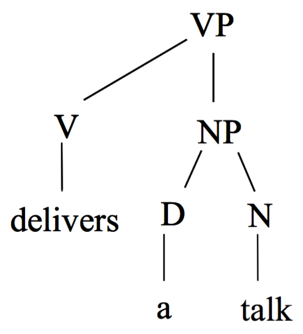

Figure 2. Example of constituency parse tree, from Moschitti ACL Tutorial 2012. Sentence tokens are always leaves and in order.

4.5.2. Collins-Duffy Tree Kernel

For our tree kernel features we only implemented the Collins-Duffy tree kernel [6]. The general idea of the Collins-Duffy Tree Kernel is that we want to compare the number of common subtrees between two input trees. The naive approach in exponential time is to enumerate all possible subtrees of the two input trees and then perform the dot-product. Collins and Duffy propose the following derivation.

Define the tree kernel $K : f(T_1, T_2) \rightarrow \mathbf{R}^+$ where T_1 and T_2 are well-formed trees. The dot-product approach above is written:

$$K(T_1, T_2) = h(T_1) \cdot h(T_2)$$

Where h is a function that produces all of the subtrees of a tree. Again, the dot-product between $h(T_1)$ and $h(T_2)$ will count all of the subtrees T_1 and T_2 have in common. They define the indicator function $I_i(n)$ to be 1 if the subtree i is rooted at node n and 0 otherwise. Therefore, we can rewrite the dot-product above.

$$h(T_1) \cdot h(T_2) = \sum_i h_i(T_1) h_i(T_2)$$

$$= \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \sum_i I_i(n_1) I_i(n_2)$$

$$= \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} C(n_1, n_2)$$

Where $C(n_1, n_2)$ is defined as $\sum_i I_i(n_1) I_i(n_2)$. We note that $C(n_1, n_2)$ can be computed recursively in polynomial time:

$C(n_1, n_2) = 0$ if the productions at n_1 and n_2 are different.

$C(n_1, n_2) = \lambda$ if the productions at n_1 and n_2 are the same and n_1 and n_2 are pre-terminal.

$$C(n_1, n_2) = \lambda \prod_{j=1}^{nc(n_1)} (\sigma + C(ch(n_1, j), ch(n_2, j)))$$

Where $0 < \lambda \leq 1$ is a factor that down weights subtrees exponentially with their size, $nc(n_i)$ returns the number of children of node i, binary variable $\sigma \in 0, 1$ determines whether we use the ST or SST kernel, and $ch(n_i, j)$ selects the jth child of node i. The ST or subtree kernel is where $\sigma = 0$. For the ST kernel, the entire subtree must match to return a non-zero score. The SST or subset tree kernel is where $\sigma = 1$. For the SST kernel, non-zero scores can be achieved even if all of the subtrees of two nodes do not match.

4.5.3. Experimental Use

The distinction between the ST and SST kernels in the previous section is that the SST kernel is more forgiving than the ST kernel. The SST kernel can still return high scores even if the exact sentences are not the same. Likewise, the ST kernel heavily penalizes minor differences between constituency trees. The factor λ can be viewed as a penalty we utilize so that kernel does not over-inflate the importance of matches between larger subtrees. These parameters become less important when we normalize the kernel score using:

$$K_{norm}(T_1, T_2) = \frac{K(T_1, T_2)}{K(T_1, T_1) K(T_2, T_2)}$$

We generate features with the ST and SST tree kernels for each question pair using $\lambda \in \{1, 0.8, 0.5, 0.2, 0.1, 0.05, 0.2\}$ totaling 14 tree kernel features.

4.6. Summary

Overall, our system used 3 derived features from semantic alignment, 24 from part of speech tags, 2 from named entity recognition, 2 from sentence length, and 14 from tree kernels, for a total of 45 features.

5. Models

For this task, Kaggle is evaluating the classification not based on accuracy but log loss. Thus, we need to build a classifier that outputs a probability over the two classes. This renders certain models such as support vector machines unusable, unless complex modifications allow them to output probabilistic classifications. We decided to train and evaluate two well known classifiers. The first is a logistic regression model. This model naturally models probabilities over classes, and with good feature engineering, we expect the problem to be linearly separable. The second is a random forest model. This is an ensemble method that classifies based on a weighted vote of weak learners, so it also naturally has a probability distribution over the classes. Overall, we did not want to explore too many models, or spend that much time worrying about tuning and overfitting. We chose these models because both are reliable and robust, and while they can be tuned and optimized, they are known to work well out of the box.

Tuning

We used scikit-learn to generate the base models with their LogisticRegression() and RandomForestClassifier() functions. For tuning, we used grid search to find optimal hyperparameters for each classifier. For logistic regression, we searched over three regularization (C) values of $[0.001, 1, 10]$, and both penalty schemes of L1 and L2 norms. We found that the default L2 norm penalty and $C = 10$ were the best hyperparameters for our log loss metric. For the random forest, the main parameters to tune are the number of estimators and maximum number of features to use. Training time goes up substantially for a larger number of estimators, and it is known that there is a cutoff point for how much classification improves with more of them. We tried a few values of *n estimators* manually, and found that performance did not improve much beyond a value of 200, and training took a while even at this value. We then did a grid search on values of maximum number of features in $[5, 10, 20]$. We found that the optimal hyperparameters were 200 estimators and 5 as the maximum number of features.

Training

To train and score the models, we used scikit-learn's built in cross validation scoring module. Thus, for both models 3-fold cross validation was used over the entire training set. This was done to output both binary classifications and probabilistic classifications. In both cases, all scores were averaged to give us the final values.

6. Evaluation

The evaluation of our features and models was done in various ways. Although we mainly desired a low error of log loss, we also wanted to see how well our models performed in standard binary classification.

6.1. Feature Analysis

First, we attempted basic feature selection to understand how well our features could separate the data. We used univariate feature selection, a process by which features are selected based on statistical tests for the variance of each feature. For exploration, we used scikit-learn's SelectKBest to see which features remained after various values of k. This function takes a certain kind of statistical test, for which we simply used chi-squared.

When $k = 3$, we saw that the best discriminating features were the number of unmatched nouns in each sentence and the difference in sentence length. When $k = 10$, the set was augmented to include more of the unmatched part of speech features, this time for verbs and even adjectives, as well as unmatched named entities. By $k = 15$, semantic similarity features were included, and by $k = 20$, tree kernel features came in. While this is not necessarily an indication of how critical the features are for performance, it aided our general understanding of their power. We decided to plot a few of the features against the data, to visualize how well they are separating the two labels [7].

In the end, we found that decreasing the number of features used to train the models did not improve the performance of either classifier. This is likely because we only had 45 features, and the models naturally learned which features were most relevant, while the noise generated by the more extraneous features was irrelevant.

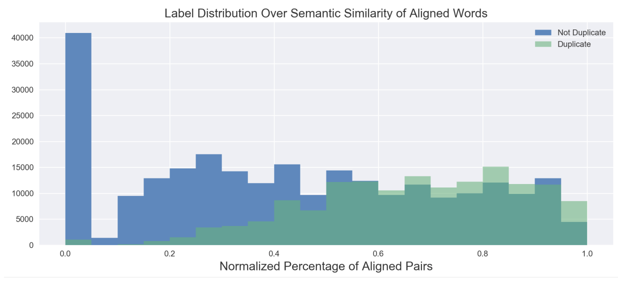

Figure 3. Semantic Similarity Feature. It separates the data reasonably well, notably when the feature value is low.

[7]Code for plotting was modeled on a Kaggle post https://www.kaggle.com/anokas/data-analysis-xgboost-starter-0-35460-lb

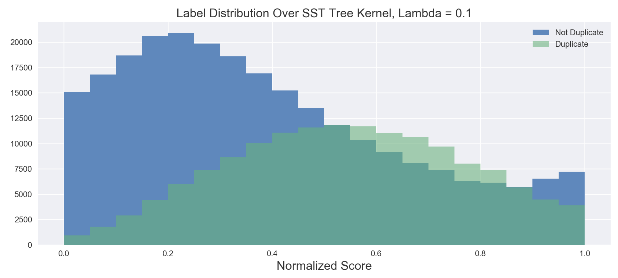

Figure 4. Tree Kernel Feature. Also performs well for low values.

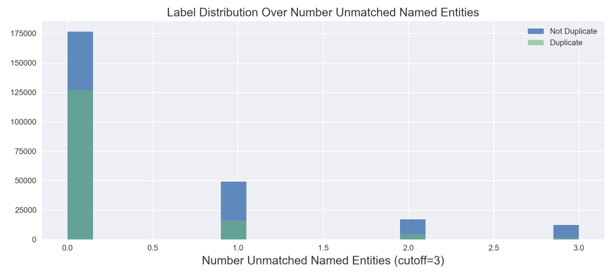

Figure 5. Unmatched Named Entities Feature. We see that higher values indicate a better chance of a non duplicate label.

6.2. Metric Scores

For each classifier, we decided to evaluate their predictions based on the usual metrics of accuracy, precision, recall, and F_1 score, as well as log loss. The following table shows the score breakdowns of each model:

Metric	Logistic Regression	Random Forest
Accuracy	0.712	0.755
Precision	0.620	0.679
Recall	0.569	0.635
F_1 Score	0.594	0.657
Log Loss	0.526	0.489

Table 1. Evaluation metrics for logistic regression and random forest classifiers.

The accuracy for both models is well above chance, but precision and recall are not particularly strong. The Quora team achieved accuracies of about 87%, which is quite high in comparison to our best value of 75.5%. The baseline value for log loss is 0.693 (when all values are predicted at 0.5), so we also have considerably lower than baseline log loss scores, but not dramatically so.

The random forest clearly has a higher overall performance. We believe this could be the case because our feature space was not as linearly separable as we had imagined. It seems that so many pairs share common syntactic structure, such that the patterns that discriminate their pairs are subtle and could be particular combinations of indicators.

6.3. ROC Curve

When tuning the logistic regression classifier, we also found it useful to plot its ROC curve for different regularization values. The ROC curve shows how well the true positive classification rate (or recall) grows as the classifier is allowed to have a higher false positive rate. Its area is a measure of how accurate the predictions are overall.

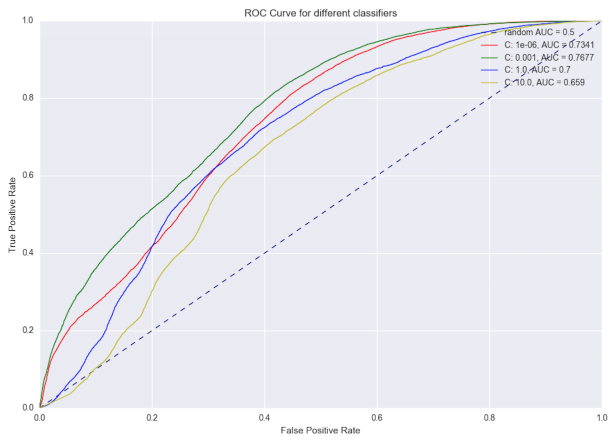

Figure 6. ROC curve for the logistic regression model for various regularization values.

Here we actually see a best accuracy performance from the regularization value $C = 0.001$, and in fact the lowest from our optimally discovered value $C = 10$. This is because we optimized not for accuracy, but log loss. In a way this makes sense, because log loss heavily penalizes overconfident, incorrect predictions. The more pairs these other classifiers would accurately predict would be offset by the extreme loss on fewer examples.

6.4. Performance on Kaggle

After tuning all parameters, we were ready to submit predictions to the Kaggle competition. We made multiple submissions, with different sets of features and trained models. The history of our submissions is outlined in Table 2.

Submission	Model	Features	Score	Rank
1	LR	Tree Kernels	0.47767	1799
2	LR	All	0.46304	1672
3	LR	All Final	0.46247	1651
4	RF	All Final	0.46084	1646

Table 2. Submissions to Kaggle competition and associated information. LR is logistic regression, RF is random forest.

7. Discussion

Features

We were a bit surprised to find that neither the semantic similarity features nor the tree kernel features were the most powerful features overall. It is clear that they discriminate well when the sentences are quite different, but they do not when sentences are similar. This was expected, as there are many question pairs that have very similar semantic and syntactic structure, yet have a non duplicate label due to small differences. However, we did not know just how many examples would fall into this category. Upon further examination, many of Quora's question pairs are deliberately constructed to be challenging in this way.

On the other hand, the features of identifying unmatched words were strong given their simplicity. This is less surprising, as we knew this would be a component of any competent system. As we thought, differences in the number of mentioned named entities is very important, specifically when there are multiple that remain unmatched.

For additional features, we would next look into sentence embeddings and vector space representations. There are so many other combinations of unmatched word features to try, but we have likely reaped much of the power of this approach already.

Optimization

Our modeling tuning optimized for log loss because that is the evaluation metric of the competition. We saw from the ROC curve of our logistic regression model that its binary classification accuracy could have been much higher (up to 76.8%) if it had been optimized for accuracy alone. This explains why the accuracy of our logistic regression classifier seemed low. We also note that the random forest was never tuned to optimize for accuracy, yet it still achieved a performance of 75.5%. We did not attempt this due to long training times for the random forest across different parameters.

There are also approaches to specifically optimize for log loss that we did not implement. Because there is such heavy penalty for wrong predictions, it could be useful to experiment with making our estimates more conservative in general. We did not have time to research heavily into these options, but we found discussions on Kaggle that alluded to just how effective some of these techniques could be for optimizing log loss score.

Submissions

The progression of our submission scores was also unexpected. We began with only a logistic regression model, so our first few submissions solely used that model. The very first submission only used features derived from Tree Kernels, as well as some simple length difference features. We achieved a very high score, given we lacked many of components of our total feature set. Adding in the semantic alignment, part of speech, and NER features improved our score, but not significantly. Finally, we tweaked some features and submitted predictions from both our logistic regression and random forest classifiers. This too resulted in minimal improvement.

There are a few notable points of bewilderment from this progression. The first is that we were surprised to see Tree Kernels perform so well on their own, as they appeared to have some of the lowest individual variance of all features in the set. The second is how little improvement was made after the addition of the other features, given their higher variance. Lastly, we saw a clear improvement in log loss score from the introduction of the random forest model on our cross validation, yet on the test set the improvement was not nearly as large as expected. We hope to continue to explore why these expectations of improvement were not met.

8. Conclusion

Despite lack of expected improvement, we are satisfied with our results. Our classification system is complex, using ambitious features that many teams did not consider. We shared our Tree Kernel features on the Kaggle page, and top teams were interested in them. We were able to generate these computationally intense features for almost 3 million total data points with cloud computing. Two models were tuned and trained on a large dataset, and we evaluated their predictions as both binary values and probabilities.

In the end, we currently rank 1646 on the leaderboard of 2453 teams. Although our competition ranking is not superb, we are happy to have competed in our first Kaggle competition and look forward to continuing to improve our system.

9. Bibliography

[1] Sultan, M. A., Bethard S., and Sumner T. Back to Basics for Monolingual Alignment: Exploiting Word Similarity and Contextual Evidence. Association for Computational Linguistics. 2014.

[2] Miller, George A., Beckwith, R., Felbaum, C., Gross, D., and Miller, K. Introduction to WordNet: An On-line Lexical Database. 1993.

[3] Ganitkevitch, Juri and Van Durme, Benjamin and Callison-Burch, Chris. PPDB: The Paraphares Database. Association for Computational Linguistics. 2013. http://cs.jhu.edu/ ccb/publications/ppdb.pdf

[4] Moschitti, A. (n.d.). Making Tree Kernels practical for Natural Language Learning.” EACL 2006.

[5] Moschitti, A. (n.d.). Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees. ECML 2006.

[6] Collins, M. Duffy, N. (n.d.). Convolution Kernels for Natural Language.