

Semantic Extensions to Syntactic Analysis of Queries

Ben Handy, Rohini Rajaraman

Abstract

We intend to show that leveraging semantic features can improve precision and recall of query results in information retrieval (IR) systems. Nearly all existing IR systems are built around the 'bag-of-words' model. One major shortcoming of this model is that semantic information in documents and queries is completely ignored. It has been shown in a limited setting that some syntactic information can be useful to IR [1]. We intend to show that leveraging semantic features such as synonyms can improve precision and recall of query results, especially when used in conjunction with syntactic relationships. In this paper, we describe the various machine learning algorithms and feature sets that we use to learn a binary relevance classification function. At this stage, we are not attempting to build an actual IR system using our techniques, but we hope to show the benefits of such a system.

Motivation

Traditional IR systems that are based on the 'bag-of-words' representation of documents often return irrelevant documents simply because they contain words that appear in the query. Such systems also ignore relevant documents that may contain information related to the query albeit expressed in words that aren't exact matches but are closely related to words in the query. As an example, consider the query that contains the phrase "reduce cost but increases pollution". This query should return documents that discuss "cutting price although raising contamination" as more relevant than documents that contain "increases cost and reduces pollution". Another example is that if a query contains "increase pollution", then it should avoid results that contain "does not increase pollution".

The Data

We used the NYT, APW, and XIE collections of the 1999 AQUAINT corpus. These are three of the five collections used for the TREC 2003 HARD track. Our query topics and relevance judgments are also taken from the TREC 2003 HARD track. These three collections contain a total of 320,380 news articles amounting to 1305 megabytes of text. The TREC 2003 HARD track contains 50 topics, and we used a subset of the *DESCR* tagged questions as our queries. We chose only questions for which there was a reasonable number of both relevant and irrelevant examples. In our training and testing, we chose to ignore documents on which the judges could not agree, as it is not clear how we would like our system to classify these documents. Since we need the syntactic information in the documents, we have pre-processed the documents with the natural language parser MINIPAR so each document is transformed into a list of relations of the form: word1:word1type:relation:word2type:word2 where relation is one of twenty-nine different relation types and wordtype1 and wordtype2 can take on twenty-two different part-of-speech values. In this initial investigation, we do not attempt to retrieve random documents out of the document collection. Instead, we will examine the same set of documents that have been rated by human judges. In our experiments, we consider both the case where our testing sets consist of documents-query pairs for the same queries as our training set, and the case where we are testing on a different set of queries than we have trained on. While the goal is to train a system to perform well on queries it has never seen, we can still learn much about the shortcomings and potential of our system by testing and training with the same queries.

The Features

To determine whether a candidate document is relevant to a query, we compare the MINIPAR relations in the query to the MINIPAR relations in the document. However, rather than direct relation comparison, we first would like to find other possible relations with similar meaning. For each query relation, we use WordNet to find other words that are related to word1 and word2. We maintain the following different similarity classes for each word in the query: synonym, hypernym, hyponym, holonym, and meronym. Now we can compare each 5-tuple MINIPAR relation in the query with each 5-tuple in the document, and identify both perfect matches and partial matches that may serve as reasonable substitutes.

For our initial investigations, we kept a count of exact matches, off-by-synonym matches, off-by-hypernym matches, etc. For all matches and partial matches, we require that the relation type be the same in the query relation and document relation. We also required that one word in the relation be an exact match, but this yielded extremely sparse vector. In addition, we consider an off-by-synonym match to be a different kind of similarity for different

relation types. This results in similarity vectors for our query-document pair consisting of `num_syntactic_relations` x `num_semantic_relations` entries, each representing the count of matches of a different similarity type. Using the five semantic features from WordNet listed above plus exact matches, and all 29 semantic relations from MINIPAR resulted in vectors of size 1. (203)

After testing this approach with several queries, we realized that our feature vectors were extremely sparse. In order to get more information to base our classification on, we relaxed the constraint that one word in the relation must form an exact match. Now the off-by-hypernym partial match category encompassed relation matches where neither the first words nor the second words formed a weaker match than hypernym. In addition, we added 7 features to the end of the vector, one for each semantic similarity class. These values counted total partial matches for each similarity class, regardless of relation type. While these new features had some dependence on existing features, it was useful to have a few more non-zero features on which the classifier could base its decision.

In a third investigation, we tried using a much smaller number of features that would hopefully capture a similar amount of information. Now we had only one feature for each relation type, and we used this to keep a weighted count of complete and partial matches. We chose to give partial matches $\frac{1}{2}$ the weight that exact matches received. This approach allows us to better learn the appropriate weighting to give to each type of relationship, but we do not learn the perhaps more subtle weighting between the different similarity classes.

We did not have a chance to experiment with other possible feature sets, but there were several that we considered. One subtle variation to our approach would be to add an additional `num_relation_types` number of features to the end of each vector counting the number of occurrences of each relation type in the query. This could help ensure that a new classification is based primarily on training examples for syntactically similar queries. Larger feature vectors could be created by counting different kinds of relation mismatches. However, without some way of grouping the relation mismatches, this would cause the feature vector to become prohibitively large and require an unreasonable amount of training data.

WordNet also has a much richer set of features than the ones we are currently using, and our initial experiments can help us determine which features we should drill down to a finer granularity. For example, we are currently using the recursive search functionality to gather all of the words in a similarity class, but by counting the words at each level differently, we could learn more subtle weightings for each partial match.

It is also possible to take completely different approaches to constructing a query vector. For instance, it would be possible to implement a more elaborate sentence-matching algorithm, based on the Minipar relations in a given sentence, possibly based on graph matching [2]. This technique has been effective in question answering systems, and may be useful in information retrieval as well. As a simpler modification, it may also be useful to generate a set of features that captures the proximity of matches and partial matches in the document. For example, we could reduce our vector to only hits that occur in the “best” few sentences. Alternatively, keep count of how many times multiple matches occur in a consecutive sequence of document relations, such as counting the times there are two matches in four consecutive relations, or three matches in a sentence.

Learning the Relevance Function

We now have a set of document-query similarity vectors, but we do not know how this relates to the actual relevance of the document to the query. It is clear that vectors with a higher sum of values indicates a better match, but we still must learn exactly which weights it is appropriate to give to each match. As we are ignoring documents in which judges disagreed or did not make a relevance judgement, we have binary relevant-irrelevant labels for each query-document pair. The Naive Bayes algorithm and Support Vector Machines are two machine learning algorithms that are effective in binary classification problems that involve large feature spaces such as this.

We implemented Naïve Bayes and experimented with boosting. The motivation to use boosting was to improve the performance of our Naive Bayes classifier on test samples drawn from queries that our system hadn't seen during training. As our test results reveal, the naive bayes learner performed better than the basic IR system on test samples drawn from queries used during training, but performed poorly on queries not seen during training. We expected boosting to help because in many cases the classifier did perform better than a random classifier, in terms of accuracy, thereby leading us to believe that it was a weak classifier, hence amenable to boosting. But, boosting didn't improve the performance of our classifier on the test set. We found that while the performance on the training

set improved significantly, as expected, with the number of weak classifiers used, the performance on the test queries showed little or no improvement up to three weak classifiers and began to decrease thereafter. This could be because in some cases, due to lack of sufficient relevance structure in the test queries, our naive bayes classifier did worse than a random classifier on queries it hadn't seen during training. It could therefore not be considered a 'weak classifier' in the sense of the term used in the context of boosting.

We also experimented with Support Vector Machines (SVMs) to learn a relevance function. We used the libsvm package [5] which allowed rapid prototyping and easy experimentation. Using a simple linear kernel, and some of the optimizations available in the package, we were able to get better results than with our Naïve Bayes attempts. In addition, SVMs have a natural confidence measure, which corresponds to distance from the test point to the separating hyper-plane. We found this confidence measure to be a much more reasonable way to rank results than the probability ratio we contrived for the Naïve Bayes scenario. This confirms the result that computing confidence is more reliable with Support Vector machines than Naïve Bayes [3].

Testing

After training one of our machine learning algorithms, we have a weight vector that can be used to make the relevant-irrelevant decisions with new query-document pairs. To apply our function, we convert the new query-document pairs into features as previously described, and use our hypothesis function with learned weights to determine relevance. One way to compute the error of our relevance function is to divide the number of misclassifications by the size of the test set. However, to get figures that are comparable to other information retrieval systems, we must compute precision and recall. Computing precision and recall requires us to rank the results, rather than just classifying them as relevant. We will use the confidence measures described in the previous section to rank our results for various learning methods.

In the results section, we compare the results of our system to that of a traditional IR implementation. We use Lemur [6] to get our traditional IR results with a normalized TF-IDF scoring scheme. As this system crawls all documents to come up with its results, its task is significantly more difficult. Recall that our system merely ranks the results that human judges have rated consistently. To compute the precision of the Lemur results, we first removed all results that had not been consistently ranked by human judges from the ranked list, and then computed how many of the top returned documents were relevant. Computing recall is significantly more muddled, as we are clearly not working with the whole set of documents that are relevant to the query.

Now that we have a ranked list of results, we can compute the precision p when returning the top 10, 20, or 30 documents for a query. We found that these precision rates were the best way to evaluate our performance against a traditional IR system.

At a high level, there are two ways to divide up the query-document pairs into a training set and a testing set. The first way is to include documents from all queries in both the training set and the test set. This gives our system a huge advantage, because it can leverage similarities that may only apply to a single query, since there were training examples for that query that were used to train the system. It is much more fair to only test using query-document pairs where the query is not a part of any query-document pair that was used in training. This testing provides stronger evidence that we have truly learned a general relevance function. We tested using both of these methodologies, because there is something to learn from both. In addition, we tried varying the number of queries that were used to train the system from as few as one query to as many as 10 queries.

Results

Using large feature vectors, our classifiers performed well on queries seen during training (Table 1). The SVM classifier did a better job than the Naïve Bayes classifier at predicting the relevance of documents for both types of queries –

1. 'Familiar' queries: queries that weren't used to train the classifiers.
2. 'Unseen' queries: queries that were used to train the classifiers.

However, even the Svm classifier's performance wasn't satisfactory on 'unseen' queries. We looked at this as a problem of high variance. Although the test sets contained a different set of documents even for the queries seen during testing, the classifier had already been trained on the structure and semantic content of the query, thereby making these test samples more like training data.

Table 1

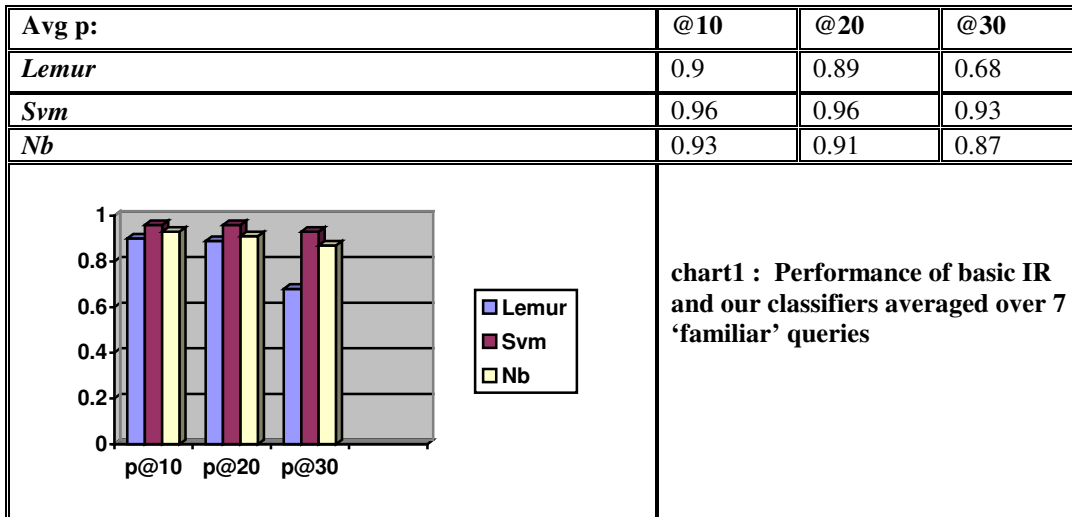
| Query | Classifier | p@10 | p@20 | p@30 |
|-------|--------------|------|------|------|
| 48 | <i>Lemur</i> | 1 | 1 | 0 |
| | <i>Svm</i> | 1 | 1 | 1 |
| | <i>Nb</i> | 1 | 0.97 | 0.98 |
| 146 | <i>Lemur</i> | 0.5 | 0.7 | 0.57 |
| | <i>Svm</i> | 0.75 | 0.81 | 0.74 |
| | <i>Nb</i> | 0.7 | 0.79 | 0.68 |
| 147 | <i>Lemur</i> | 0.8 | 0.85 | 0.7 |
| | <i>Svm</i> | 1 | 0.92 | 0.88 |
| | <i>Nb</i> | 0.85 | 0.81 | 0.73 |
| 223 | <i>Lemur</i> | 1 | 1 | 1 |
| | <i>Svm</i> | 1 | 1 | 1 |
| | <i>Nb</i> | 1 | 0.98 | 0.95 |
| 102 | <i>Lemur</i> | 1 | 1 | 0.93 |
| | <i>Svm</i> | 1 | 0.99 | 0.96 |
| | <i>Nb</i> | 0.98 | 0.99 | 0.94 |
| 234 | <i>Lemur</i> | 1 | 0.75 | 0.67 |
| | <i>Svm</i> | 1 | 1 | 0.94 |
| | <i>Nb</i> | 0.98 | 0.87 | 0.85 |
| 65 | <i>Lemur</i> | 1 | 0.95 | 0.9 |
| | <i>Svm</i> | 1 | 1 | 1 |
| | <i>Nb</i> | 1 | 0.97 | 0.93 |

Table 2

| Query | Classifier | p@10 | p@20 | p@30 |
|-------|--------------------|------|------|------|
| 53 | <i>Lemur</i> | 0.7 | 0.55 | 0.63 |
| | <i>Svm (long)</i> | 0.58 | 0.61 | 0.45 |
| | <i>Svm (short)</i> | 0.73 | 0.69 | 0.6 |
| 69 | <i>Lemur</i> | 0.3 | 0.45 | 0.43 |
| | <i>Svm (long)</i> | 0.2 | 0.21 | 0.33 |
| | <i>Svm (short)</i> | 0.51 | 0.45 | 0.48 |
| 77 | <i>Lemur</i> | 0.9 | 0.95 | 0.9 |
| | <i>Svm (long)</i> | 0.52 | 0.6 | 0.58 |
| | <i>Svm (short)</i> | 0.68 | 0.71 | 0.73 |
| 180 | <i>Lemur</i> | 0.5 | 0.55 | 0.53 |
| | <i>Svm (long)</i> | 0.38 | 0.51 | 0.49 |
| | <i>Svm (short)</i> | 0.51 | 0.63 | 0.68 |
| 196 | <i>Lemur</i> | 0.9 | 0.9 | 0.9 |
| | <i>Svm (long)</i> | 0.73 | 0.64 | 0.67 |
| | <i>Svm (short)</i> | 0.71 | 0.70 | 0.73 |
| 222 | <i>Lemur</i> | 0.6 | 0.65 | 0 |
| | <i>Svm (long)</i> | 0.42 | 0.53 | 0.49 |
| | <i>Svm (short)</i> | 0.61 | 0.59 | 0.54 |
| 237 | <i>Lemur</i> | 0.6 | 0.6 | 0.7 |
| | <i>Svm (long)</i> | 0.54 | 0.47 | 0.41 |
| | <i>Svm (short)</i> | 0.65 | 0.58 | 0.53 |

Table 1: Performance measured by precision in top 10, top 20 and top 30 documents retrieved by the two classifiers on ‘familiar’ queries. Table 2: Performance measured by precision in top 10, top 20 and top 30 documents retrieved by the two classifiers on ‘unseen’ queries

Table 3: Performance of basic IR and our classifiers averaged over 7 ‘familiar’ queries



Reduced Feature Vectors:

We then experimented with a reduced set of features to reduce variance. These new features, while still capturing the syntactic or structural similarity weighted by semantic matches, avoided over-fitting and generalized better to ‘unseen’ queries. We tabulate below, the results obtained by running the Svm classifier on these reduced feature vectors as compared to its performance on the longer feature vectors. The table also gives the performance of the basic IR (Lemur) system on these queries, for comparison.

Table 4: Performance of basic IR and our classifiers averaged over 7 ‘unseen’ queries

| Avg p: | @10 | @20 | @30 |
|------------------------------------|------|------|------|
| <i>Lemur</i> | 0.64 | 0.66 | 0.58 |
| <i>Svm (Long Feature Vectors)</i> | 0.48 | 0.51 | 0.49 |
| <i>Svm (Short Feature Vectors)</i> | 0.63 | 0.62 | 0.61 |

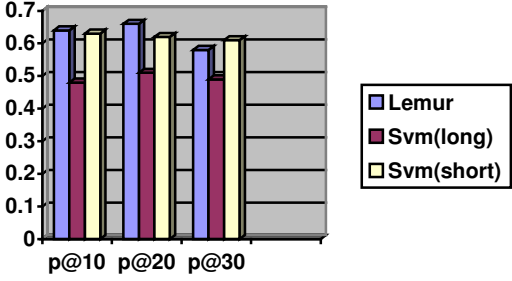


Chart2: Performance of basic IR and the Svm classifier using long and short feature vectors, averaged over 7 ‘unseen’ queries

Conclusions

Our classifier did very well in judging the relevance of new documents to queries it was trained on; queries whose structure and semantic content the classifier was familiar with. It however did not do well on most ‘unseen’ queries. We suggest that the anomalies to the latter case are queries whose structure is similar to one of the queries that the classifier was trained on. This further suggests that *the performance of the IR system can be improved* over a restricted domain of queries by identifying the types of queries in the domain, based on their syntactic structure, and training the classifier with representative queries from each class, so that with a high probability, all ‘unseen’ queries are ‘familiar’ to the classifier.

While we have shown that better result ranking can be done by a syntactically and semantically aware system, this is not the same as doing full information retrieval. It will take significantly more work to design a complete IR system that can leverage some of the techniques we have demonstrated. Several of these techniques will not scale well to a large-scale high performance query engine. Another area for future research is examining what kinds of compromises can allow us to minimize the computational cost while still reaping some of the benefits we have seen.

References

- 1: [Saria, Taylor; 2004; Document Retrieval using Syntactic Analysis of Queries]
- 2: [Haghighi, Ng, Manning; 2005; Robust Textual Inference Via Graph Matching]
- 3: [Delany, Cunningham, Doyle; 2005; Generating Estimates of Classification Confidence for a Case-Based Spam Filter]
- 4: [Kim, Hahn, Zhang; 2000; Text Filtering by Boosting Naive Bayes Classifiers]
- 5: [Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines, 2001. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>]
- 6: [Ogilvie, Callan; 2001; Experiments Using the Lemur Toolkit]