Acronym Disambiguation

Benjamin D. Turtel
Courant Institute, New York University
Advisor: Dennis Shasha

Abstract

Acronym disambiguation is the process of determining the correct expansion of an acronym in a given context. We describe a novel approach for expanding acronyms, by identifying acronym / expansion pairs in a large training corpus of text from Wikipedia and using these as a training dataset to expand acronyms based on word frequencies. On instances in which the correct acronym expansion has at least one instance in our training set (therefore making correct expansion possible), and in which the correct expansion is not the only expansion of an acronym seen in our training set (therefore making the expansion decision a non-trivial decision), we achieve an average accuracy of 88.6%. On a second set of experiments using user-submitted documents, we achieve an average accuracy of 81%.

Introduction

Within documents of various kinds, acronyms are often used to shorten complicated or oft-repeated terms that possess such established shortened forms. Usually these acronyms will be conveniently defined at the point of first usage, but sometimes a document will omit the definition entirely, perhaps assuming the reader's familiarity with the acronym. In cases where a reader does not have such prior knowledge, we posit that a program to scan a document for unknown acronyms and predict the correct expansion of these acronyms would be useful. We endeavored to create such a program using machine learning techniques.

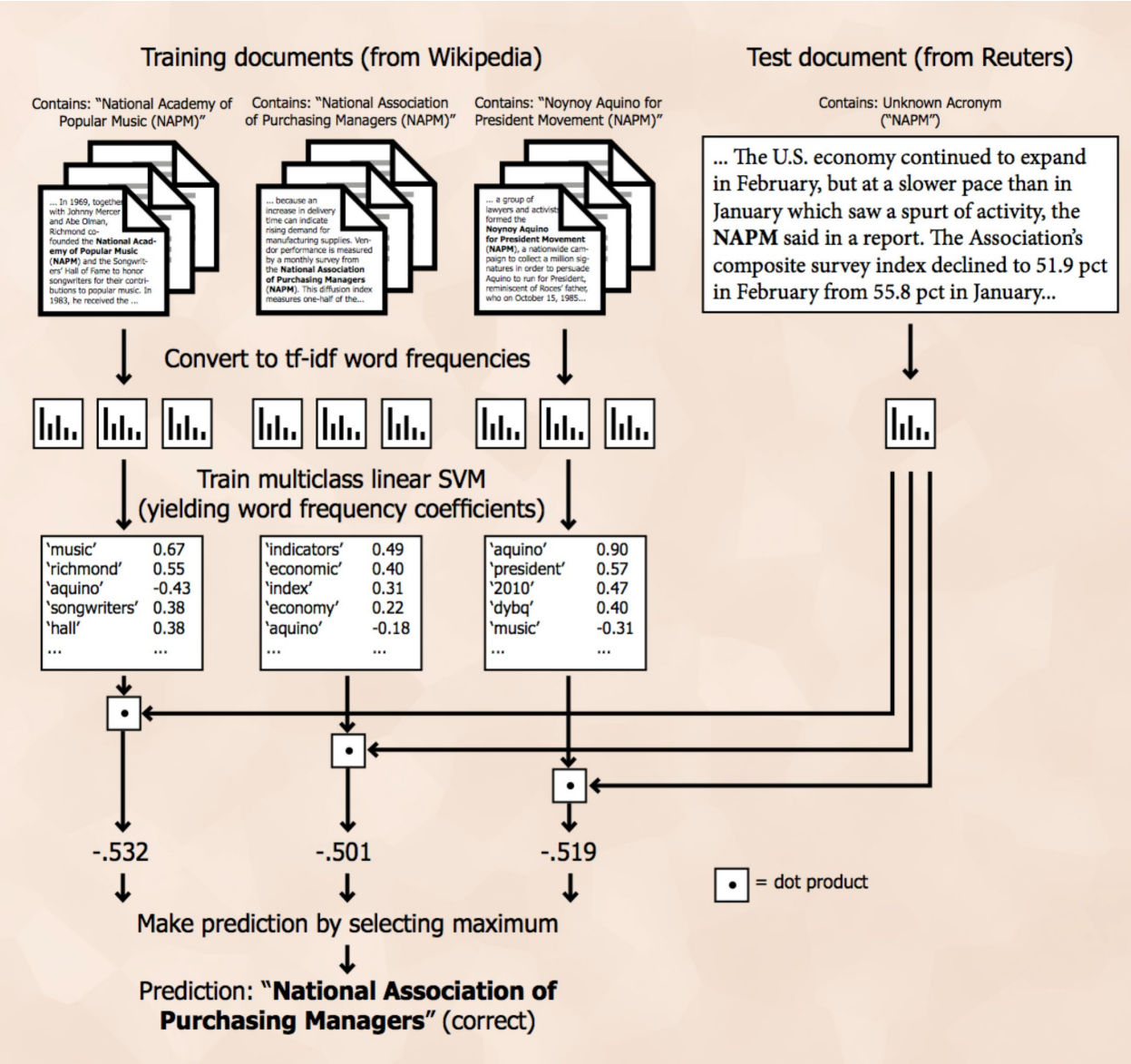
coref. resolu^n?

From the point of view of a user, our system accepts a document containing an acronym with an unknown expanded form, and returns its best guess of the true expanded form by analyzing the words within the document. For example, a user may supply the system with a news article containing, in part, the following sentence: "... The U.S. Economy continued to expand in February but at a slower pace than in January, which saw a spurt of activity, the NAPM said in a report. ...". It will then return its prediction of the true expanded form of "NAPM"; in this case it selects the correct one, "National Association of Purchasing Managers".

weighting words based on distance from acronym.

Given a document containing an unknown acronym, our system makes its prediction in two general phases. First, it creates a training data set of previously scraped expansions and the documents they were discovered in, corresponding to the relevant acronym. In the example given, these expansions are "National Academy of Popular Music", "National Association of Photographic Manufacturers", "National Association of Presort Mailers", "National Association of Purchasing Managers", and "Noynoy Aquino for President Movement".

The documents in the training set, along with the user's document, are then transformed into vectors of relative word frequencies. The transformed vectors become inputs into a classifier, where the various potential expansions of the acronym form the classes. The classifier, a support vector machine, is trained on the transformed training set. Finally, the user's transformed document is used as input to the classifier; the expansion corresponding to the output class is our final prediction. An example of this process is depicted in the diagram below.



We have achieved satisfactory results with this method. In particular, our system obtained 88.6% accuracy on a test set of Reuters news articles containing acronyms. This score excludes cases in which the correct expansion was never encountered during training, and would therefore be impossible for our classifier to predict, and also excludes cases in which the correct expansion is the only expansion encountered during training, and would therefore be impossible for our classifier to predict incorrectly. The overall accuracy, including no-decision

cases, is 75.5%. The gap between these scores suggests that additional data sources would continue to improve the accuracy of this algorithm. These alternate scores are included for reference below. The remainder of this report will primarily detail our learning method, explain why we chose it, and provide more detailed analysis and results regarding its performance.

Problem Definition and Algorithm

Task

Our task is to identify and correctly expand acronyms in text documents. The input to the program is a set of documents containing any number of unknown acronyms. The system will then predict the expanded forms of these acronyms by analyzing the surrounding texts, and output these predictions.

Algorithm

This section will describe our system's behavior at test time, after preprocessing has been completed and the program therefore has as input a document containing an unknown acronym and the relevant Wikipedia articles containing the unknown acronym across all its possible expanded forms. (An input document can have multiple distinct unknown acronyms, but our system considers each individually so it is sufficient to describe the behavior in the case of one.) Because our training set has a maximum of only about 350 documents for a given unexpanded acronym (and frequently this number is in the tens to single digits), we do all training and classification "on-demand" at test time, so this section actually describes, from start to end, what happens at this point.

As an explanatory example, we will examine the case of an input document containing "NAPM", as described previously, but with more detail regarding learning and classification. The following table shows the number of training documents available for each possible expansion of "NAPM":

'noynoy aquino for president movement'	3 articles
'national academy of popular music'	3 articles
'national association of purchasing managers'	1 article
'national association of photographic manufacturers'	1 article
'national association of presort mailers'	1 article

Clearly, our system must cope with cases where there is only a small amount of training data available.

A feature vector is first computed for each document using 'term frequency–inverse document frequency' (td-idf) weighting on contained words. td-idf weighting ensures that words which are frequently used across all documents to a similar degree are not as highly weighted as those which appear frequently in only a subset of considered documents. The formula for calculating tf-idf is as follows:

tf(word, document) = number of appearances of 'word' in 'document' / number of terms in 'document'

idf(word, corpus) = log(number of documents in 'corpus' / number of documents in 'corpus' which contain 'word')

tf-idf(word, document, corpus) = tf(word, document) × idf(word, corpus)

words → base words?
↓
word2vec?

In our case, the corpus is the set of training documents corresponding to a given unexpanded acronym. Our implementation also limits the total vocabulary over the entire corpus to the 10,000 terms with the highest term frequency (excluding common English "stop words" such as "the", "between", "another", etc.). These feature vectors indicate the frequency of each of these 10,000 terms for a given document, so location k in the feature vector of each document corresponds to the same term. The feature vectors, together with the labels given by the expanded forms of the acronyms corresponding to each feature vector, are then used to train a linear support vector machine.

Our SVM utilizes a one-vs-all multiclass classification strategy, whereby a set of coefficients is learned for each possible class (which is a single acronym expansion in our case. Each coefficient set is used during classification to decide whether the test example is likely part of the corresponding class or not (and therefore must belong to some other class). A bias/intercept term is also computed.

Formally, this support vector machine optimizes the following function for each possible class:

$$\begin{aligned} &\underset{w,b,\zeta}{\text{minimize}}\ \tfrac{1}{2}||w||^2 + C\sum_{i=1}^{n}\zeta_i^f \\ &\text{subject to}\ \ y_i(wx_i + b) \geq 1 - \zeta_i \\ &\zeta_i \geq 0, i = 1, ..., n \\ &(C > 0, f = 1, 2) \end{aligned} \qquad (1)$$

In these functions, w represents the weight vector, xi represents an individual training sample vector, and yi is the binary "truth" value of the training sample's class (-1 or 1). zeta represents the "slack" variable, which is 0 if the sample falls within the correct class, with a wide enough margin, but larger than 0 if the sample is classified incorrectly, or ~~too close to~~ the linear separator. The hyperparameter C balances two penalties in the equation: (i) the penalty for incorrectly classifying samples and (ii) the overall size of the w vector. A high C value will weigh correctly classifying all samples over maximizing margin, and a low C value will maximize

margin and be more tolerant of training errors. Since each individual coefficient corresponds to the scaling factor assigned to an individual feature (one of the 10,000 terms) during classification, and since each feature is simply a word frequency (after being transformed by td-idf), we can go back and examine these coefficients to determine which words have the largest impact on the classification decision for each possible expansion. The weight coefficients with the largest absolute values are shown for the possible expansions in the "NAPM" example:

tree? will give feature importance as well.

Expansion	**Associated Features / Coefficients**
national academy of popular music	'music' 0.67 'richmond' 0.55 'aquino' -0.43 'songwriters' 0.38 'hall' 0.38
national association of photographic manufacturers	'imaging' 0.51 'photographic' 0.34 'pima', 0.34 'association' 0.30 'manufacturers' 0.28
national association of presort mailers	'mailers' 0.55 'firmsexternal' 0.27 'httppresortmailerorg' 0.27 'mailing' 0.27 'presort' 0.27
national association of purchasing managers	'indicators' 0.49 'economic' 0.40 'index' 0.31 'economy' 0.22 'aquino' -0.18
noynoy aquino for president movement	'aquino' 0.90 'president' 0.57 '2010' 0.47 'dybq' 0.40 'music' -0.31

For example, for the first expansion, "national academy of popular music", the most highly weighted feature is "music". This means that a document containing the acronym "NAPM" and containing a higher-than-average frequency of the word "music" is more likely to be assigned to this class. The term "aquino", on the other hand, will make a document much less likely to be assigned to the "national academy of popular music" class.

This outcome in fact makes sense when we look closely: the National Academy of Popular Music was co-founded by Howie *Richmond* and administers the *Songwriter's Hall* of Fame; the National Association of Photographic Manufacturers later changed its name to *Photographic* and *Imaging Manufacturers Association*, or *PIMA*; Benigno Aquino ran for *president* of the

Philippines in *2010* and an associate of his campaign is rumored to be seeking to manage *DYBQ*, a radio station in that country. (‘firmsexternal’ and 'httppresortmailerorg' are url/text fragments inadvertently included within the Wikipedia article.)

The input test document is also converted to a tf-idf frequency vector. For each possible class, we compute the dot product of the class’s learned coefficient vector and the test document frequency vector (and add the class’s bias term). This results in a set of scalar values, one for each possible class.

In the case of the “NAPM” example, the set of scalars is as follows:

'noynoy aquino for president movement'	-0.51938689
'national academy of popular music'	-0.53241556
'national association of purchasing managers'	-0.50180741
'national association of photographic manufacturers'	-0.82459512
'national association of presort mailers'	-0.83109989

The scalar with the maximum value is chosen, and its corresponding class is returned as the overall prediction. In this case, -0.50180741 is the largest value, and 'national association of purchasing managers' is predicted. This is in fact the correct answer, as this was indeed the organization to which the test document referred.

We utilized both the tf-idf vectorizer and the linear SVM implementation from the Python scikit-learn library in our efforts.

Can it identify AMA?

Experiments

Data

To build our system, the entirety of English Wikipedia in textual form was first obtained. All articles were then scanned for acronym/definition pairs using a series of regular expressions which matched various forms of such pairs. For example, “NBA (National Basketball Association)”, “National Basketball Association (NBA)”, and “National Basketball Association, or NBA” would all be matched, among other forms. The scanner also ensured that the acronym’s letters were all contained, in order, within the expanded form.

Each of these documents (along with the expanded form of each acronym, for easy access) were then placed in a database, indexed by the shortened form of the acronym. In this way our system is able to efficiently pull up all the articles which contain a shortened form of an acronym (including documents for all of its expanded definitions) at test time. Additionally, as mentioned previously, we used these documents to generate word2vec word embeddings.

The following graph shows the distribution of the number of distinct definitions per acronym within the Wikipedia data. For example, NAPM has 5 possible definitions as outlined above and would contribute to the x=5 column. Clearly most encountered acronyms have a relatively small number of associated definitions, but there are some that have over a hundred.

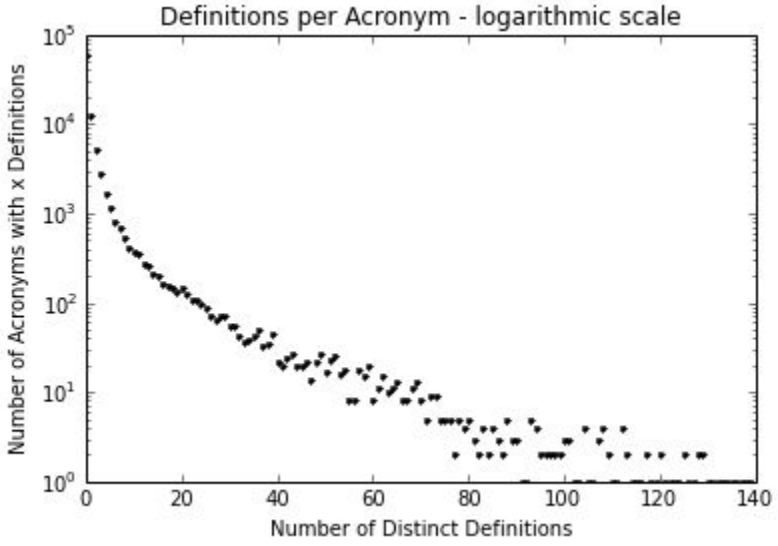


Figure 1

The next graph shows the distribution of the number of training instances seen per distinct definition. For example, 'national academy of popular music (NAPM)' (or a similar arrangement of text) appears in 3 Wikipedia documents, so it would contribute to the x=3 column. Unfortunately, this graph reveals that many acronym definitions within the training data appear in only a few documents; in fact, the largest proportion of them occur in only one article.

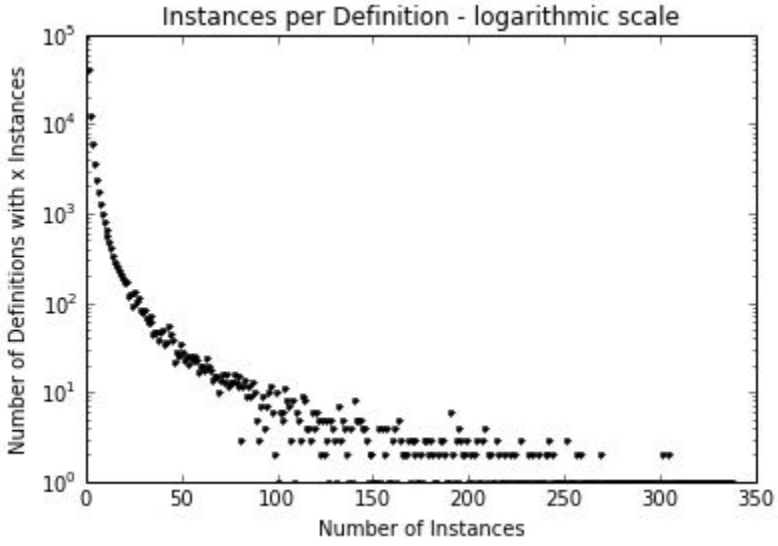


Figure 2

However, this situation regarding lack of training data isn't as dire as these graphs might suggest, since we actually test and intend our system to be proficient on news articles which tend to contain more commonly used acronyms.

Specifically, for traning and testing, we used a freely available set of 19,065 Reuters news articles. Each acronym and adjacent expansion pair together with the text in the surrounding article is considered to be a discrete example. Within each test example, the acronym expansion is the “true” label and the surrounding text (with the acronym expansion erased) is used to create the feature vector.

The following two graphs are the same as the last two, except that we restrict them to acronyms actually encountered in the Reuters news dataset.

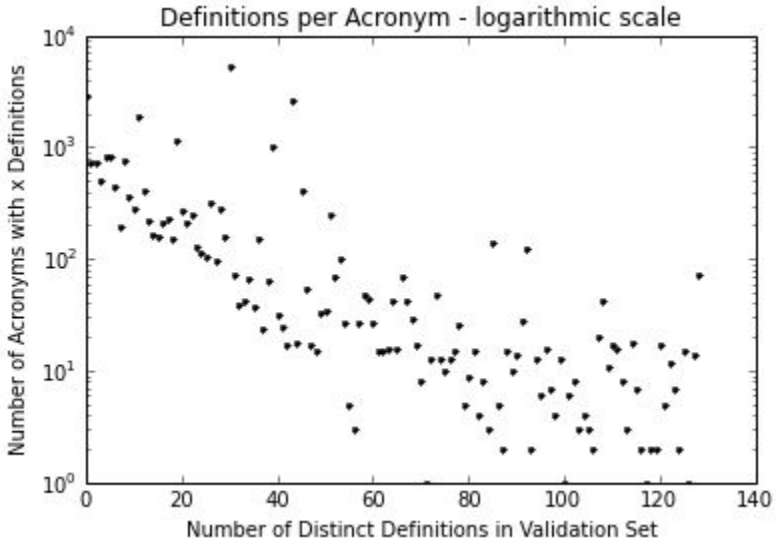


Figure 3

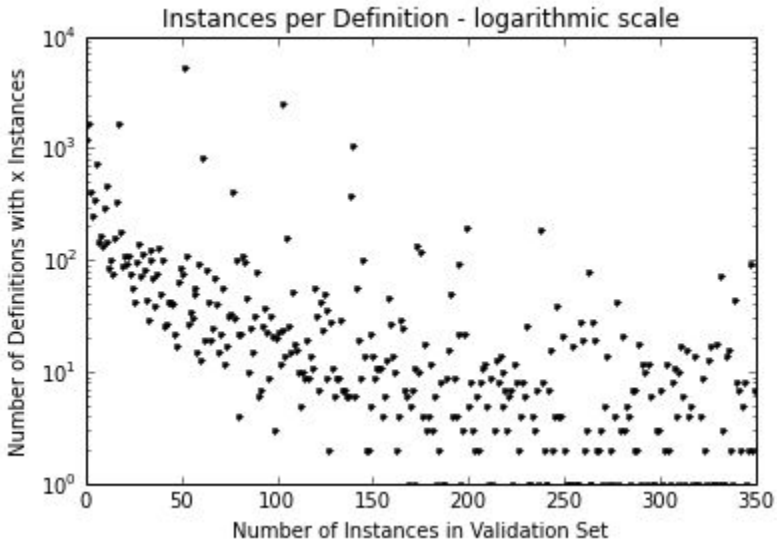


Figure 4

There are still some acronyms encountered in the Reuters dataset for which we have not seen much training data, but this is less common than in the entire Wikipedia dataset.

Methodology

We tested whether the above algorithm would identify the correct expansion of acronyms given the surrounding text. In particular, for cases in which our system has seen the correct expansion in training, we compared the algorithm against the naïve baseline of 1.0/(number of seen distinct expansions).

To facilitate testing, the Reuters news articles were split into training and test sets of 9,705 and 9,360 articles, respectively. The training set contained 825 ~~discrete~~ examples and the test set contained 744 discrete examples.

We used the training set to explore various classification strategies and set parameters. Specifically, we examined various SVM kernel functions, one-vs-one and one-vs-rest strategies (resulting in *n* or *n**(*n*-1)/2 binary classifiers, respectively, where n is the number of possible classes), penalty parameter *C*, and loss function (hinge vs squared hinge).

We also tested on the alternate feature vector constructed from word embeddings. This led to our setting important parameters such as window size and vector size. The window size is the size of the window of words included to the left and to the right of the acronym being considered in a given text document. The vector size is the dimension of the vectors used to represent each word in the word embeddings.

Implementation Details

Data: Training data, the full text from Wikipedia, can be downloaded as compressed files using the Wikipedia data dumps. Other sources offer full text file downloads. Testing and traning data, the well-known Reuters news dataset, can be downloaded from many sources.

Scraping Acronym Pairs: Iterate over each Wikipedia article, filtering out articles with little or no content (such as disambiguation or file pages). Search each article using regular expressions for acronyms, and then for acronym / definition pairs based on the found acronyms. For example, if “NLP” is found, then create regular expression patterns such as “Natural Language Processing (NLP)” and “NLP (Natural Language Processing)”. These regular expression patterns were developed through a combination of empirical observation and testing by collecting samples and finding the regular expressions that covered the most samples while also returning the least mistakes. Each time a pair is found, store the definition and article text (removing punctuation, newline chars, etc) in a dictionary-like structure using the acronym as the key (we used python’s shelf structure, but a database would work for this as well). Normalize the definitions by lowercasing all words, replacing dashes with spaces, and grouping

extremely similar definitions together (we do this using the same_exp function, shown in the appendix). Sample code for scraping acronym pairs can be found in the appendix.

Testing Model: Create a tf-idf vectorizer by randomly sampling Wikipedia articles. For each document (from Reuters news dump) to be tested, create a list of acronyms found in the document. For each unique acronym found, check to see if the acronym is defined in the article. If it is, remove that definition from the article, and save the definition as your target value. Train a multiclass linear SVM using all definition / vectorized Wikipedia article combinations stored in your training data dictionary under the key of the relevant acronym. Vectorize the test Reuters article, and predict a definition using the linear SVM. Sample code for running the test can be found in the appendix.

Results

First we report the various results on the training data which were used to tune our algorithm. We start with the error rate for various SVM kernel functions and classification strategies.

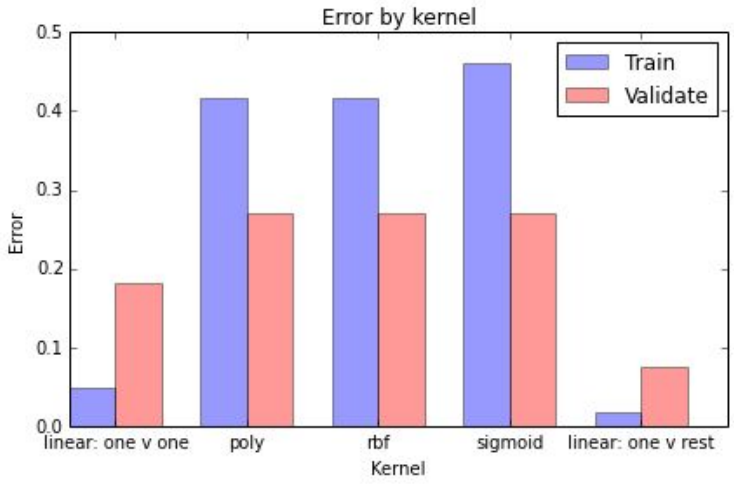


Figure 5

Using a simple linear kernel function gave us good results, and specifically employing a one-vs-rest classification strategy performed the best on our training set.

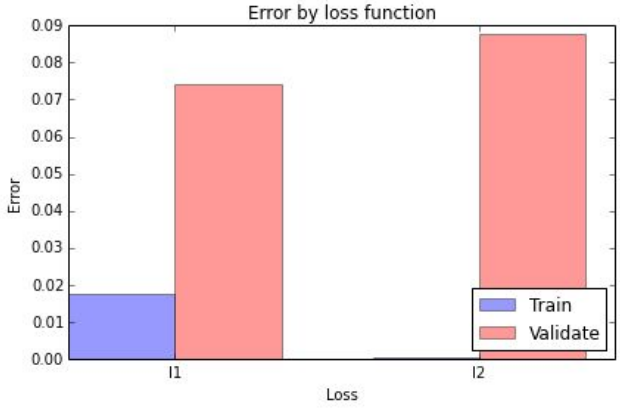


Figure 6

Hinge Loss (L1) outperformed Squared Hinge Loss (L2) during training despite having larger training error. This indicates that L2 loss overfits compared to L1 loss. To maximize our classifier's ability to generalize to unseen data, we thus chose to use L1 loss going forward.

We generated three graphs when testing on the C parameter. The first shows our accuracy when including all training acronyms (acronym instances in the training set used for tuning the algorithm, and held separate from the testing set)

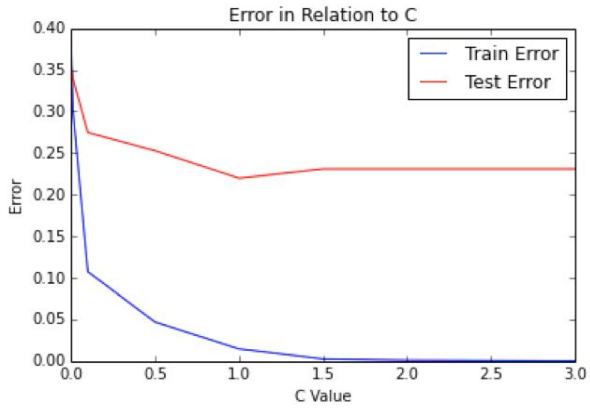


Figure 7(a)

The second graph shows accuracy when acronym expansions we could not have predicted (due to their not appearing in any training documents) are excluded.

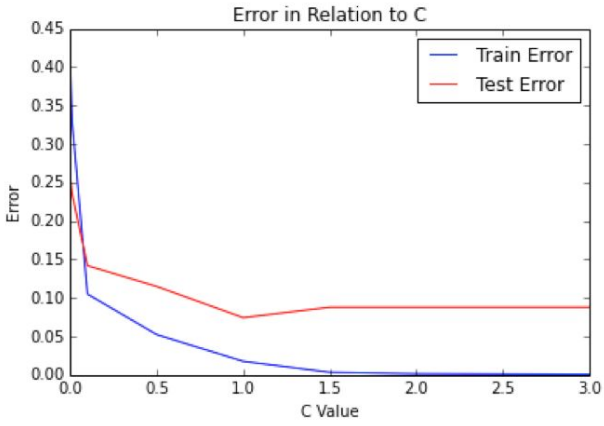


Figure 7(b)

The third graph shows accuracy when (as before) expansions we could not have predicted are excluded, and also cases in which we are guaranteed to get the correct answer (due to having seen only one correct expansion during training) are excluded.

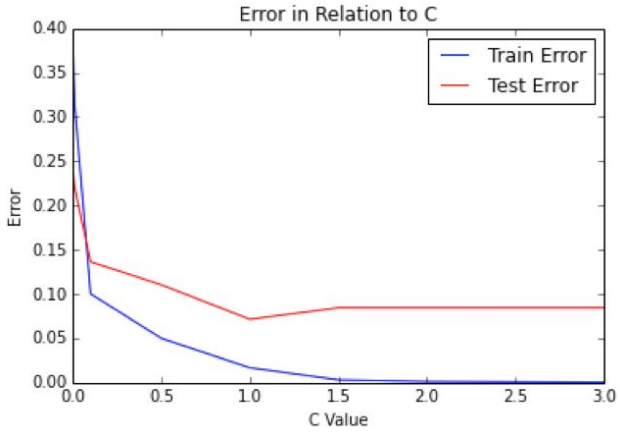


Figure 7(c)

The ideal C value is not affected by simply throwing out results when we are guaranteed to get them right or wrong, but showing these three graphs separately gives a sense of the extent to which factors beyond our machine learning algorithm could affect our system's overall success in making correct predictions.

The results show that the ideal C parameter is close to 1.0, so this value was chosen when scoring the test set.

We considered the idea of varying the C parameter across test examples rather than using a single value for all of them. However, we deemed performing cross validation to determine a unique C value for each test example at test time to be too computationally intensive. We also considered the idea of varying C based on some heuristic involving the number of training examples seen per acronym to be somewhat intractable since these training examples are often distributed among the various distinct expansions in very different ways (sometimes there is an even distribution, sometimes most examples correspond to a single expansion, etc.).

However, we did perform some initial testing to see if the ideal C parameter would vary based on the number of distinct possible acronym expansions. We created graphs of accuracy rates against the number of acronym expansions; for each graph a different C value is used. In general, if a change in C value were to disproportionately impact one part of the graph, that would suggest that varying C based on the number of acronym expansions could be useful.

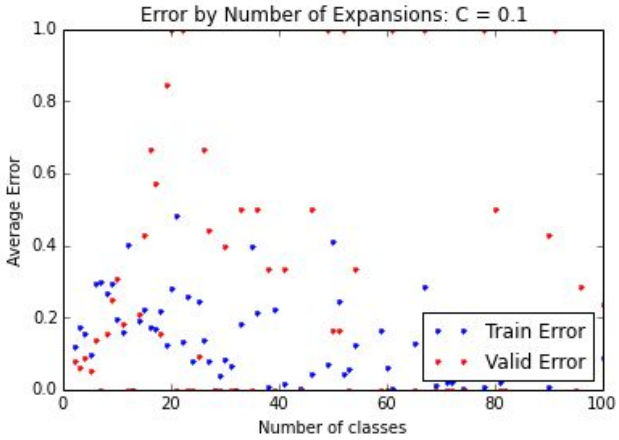


Figure 8 (a)

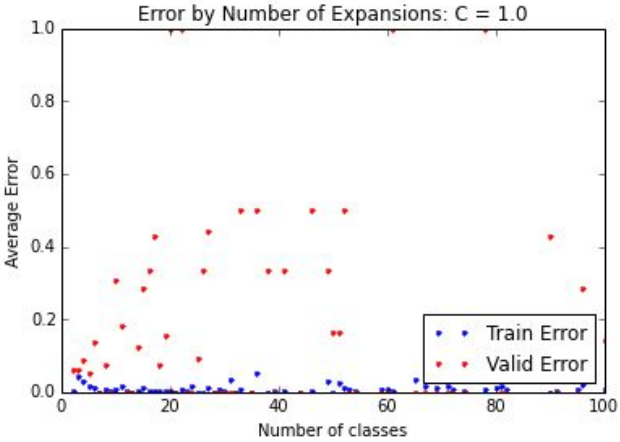


Figure 8(b)

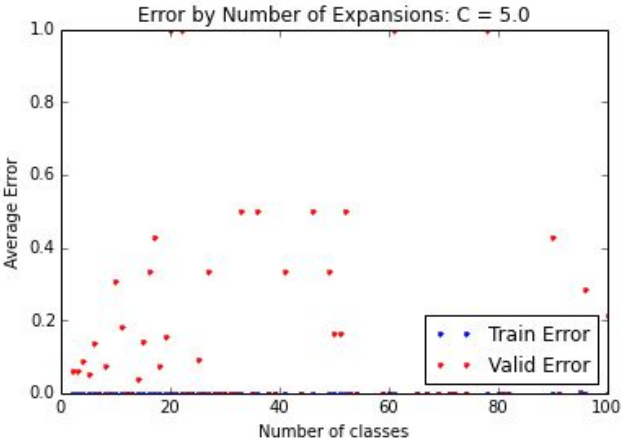


Figure 8(c)

Our next step would have been to devise a heuristic to vary the C parameter at test time as a function of the number of expansions, and to run against the traning and test sets while incorporating this heuristic. However, the generated graphs do not seem to give clear evidence

that varying C in this way would be useful, at least in terms of the Reuters news articles that we test against, so we stopped pursuing this approach.

This last set of graphs gives insight into our performance as a function of the number of possible expansions. Specifically, it is clear that we do indeed do better when there are fewer expansions to pick from. However, we usually do better than 50% even when the number of possibilities reaches from the tens to one hundred.

Significance testing was accomplished using another subset of the Reuters article dataset, with one article for each of 368 unique acronyms. It's important to note that this test set differs significantly from the previous test set, as each acronym was limited to one instance. The algorithm described above was run against a control method of selecting the top definition for each acronym on allacronyms.com, regardless of context, which is typically the most common expansion. On this test set, the algorithm scored with a mean accuracy of 83%, while the control method scored with a mean accuracy of 41%, giving us an observed difference between the two means of 42%. The shuffle test was run first 10,000 times, and then 1,000,000 times, without a single random shuffling obtaining a difference as large as 42%. This suggests that our p value is $p < .000001$.

Alternate Method: Word Embedding Models using Word2Vec

One problem we faced in our acronym expansion task is that many acronym definitions have very few training samples. Preprocessing, and stemming in particular, allow us to identify similar forms of the same word, but don't give us any intuition as for similar words. This seems to be an inherent limitation of the tf-idf vector model. We tried to tackle this problem using ~~deep~~ word embeddings, using the word2vec package.

Our word2vec embeddings were trained on the same Wikipedia data that we used for extracting acronym definitions. The input is a list of documents, each tokenized into a sequence of words. Essentially, word2vec uses this to create word "embeddings" where two words that are often used in a similar context will have similar locations in vector space, so that the distance between the two vectors is smaller than the distance between two unrelated words. Generating vector representations for individual words can provide useful context regarding how words relate to each other. For example, although the words "school" and "university" are close in meaning, they are obviously spelled nothing alike and tf-idf approaches would simply consider them to be two entirely different. However, more recent machine learning approaches such as the vector word representation method developed by Tomas Mikolov et al. can be used to generate vectors for words, such that words close in meaning tend to have vector representations that are also close together (in a mathematical sense).

Mikolov's technique involves training a multilayer neural network using a corpus of text documents as inputs. Each word encountered during training (together with a "window" of words

to its right and left) is used as a discrete training example. The network's inputs are sent to a projection layer, a hidden layer, and finally the output layer. The objective is to minimize the following:

$$\sum_{t=0}^{T} \sum_{-c+t \leq w \leq c+t, w \neq t} \log \frac{e^{v_c \cdot v_w}}{\sum_{i=0}^{W} e^{v_i \cdot v_w}} \quad (2)$$

In this formulation we are training on words w0, w1, … wT. c is the window size and W is the number of words in the vocabulary. vx is the word embedding vector for word x. The assumption as stated in the literature is that minimizing this objective will tend to produce embedding vectors which are nearby when their associated words frequently appear together in the corpus. The provided word2vec tool from Google permits a useful starting point in investigating this model.

Using word vectors, we attempted a similar classification methodology as we did with tf-idf vectors. For each training sample, we collect a window of words to the right and left or each instance of the acronym. We combine the word vectors for each word in these windows, and use the resulting vector as our training input. The same process is used to create the input feature vector in our test set. (The idea of summing adjacent word vectors is the recommended methodology for creating "phrase vectors" according to the word2vec instructional website.)

We generated multiple sets of word embeddings by varying the parameters for vector length and the context window size. In addition, we used a downloaded word embedding trained on 1 billion words from the Google News dataset, created by Mikilov's team at Google, for comparison.

For any acronym found in an input document, we create a combined vector using word embeddings from the words within a specific text window near that acronym. For example, we'll examine the case of an input document containing the acronym "NLP", which begins as follows:

"Automated, deep NLP technology may hold a solution for more efficiently processing text information and enabling understanding connections in text that might not be readily apparent to humans. DARPA created the Deep Exploration and Filtering of Text (DEFT) program to harness the power of NLP. Sophisticated artificial intelligence of this nature has the potential to enable defense analysts to efficiently investigate orders of magnitude more documents, which would enable discovery of implicitly expressed, actionable information within those documents."

In this case, the following words were selected as "window words" -- words nearby the acronyms and which have been previously seen in training:

DEFT, Filtering, Sophisticated, Text, artificial, efficiently, for, harness, has, hold, intelligence, may, more, nature, potential, power, processing, program, solution, technology, the, this

For each of these words, we look up the previously calculated word embedding vector and then combine them together by one of three methods: adding, averaging, or taking the maximum at each column. Adding is said to provide an approximation for the representation of phrases or short sentences, but we also wanted to see whether other methods would be better suited to our task.

We provide results for selected window and vector sizes on our Wikipedia training data (approximately 400,000 articles), and for the pretrained Google News dictionary (100 billion words, window size 5, vector size 300), and for each of the three describe ways of combining the word embeddings when training and testing with the SVM. Our traning set included 9,705 Reuters news articles which contained 825 discrete examples of acronyms with adjacent expansions. (The expansions are erased during testing but used later to compute accuracy.)

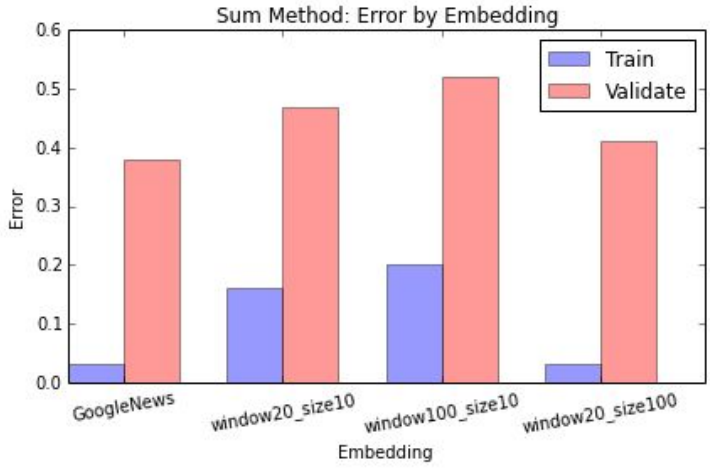


Figure 9(a)

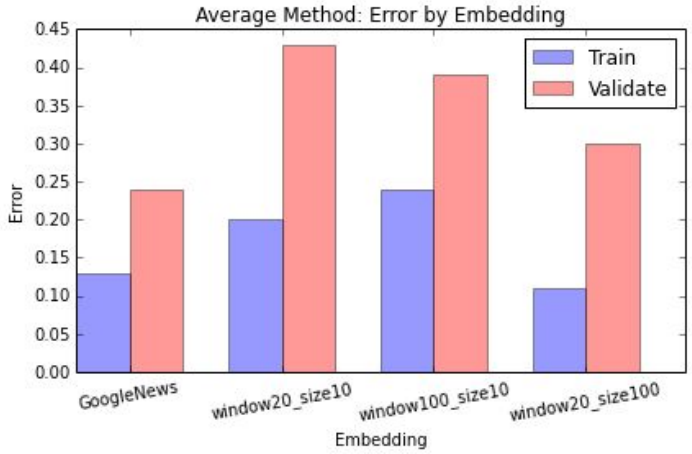


Figure 9(b)

overfit?
paragraph 2 vec?
vary similarity measures

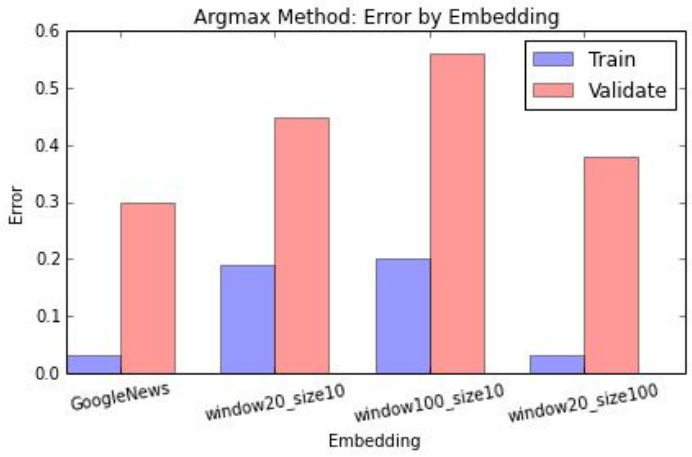


Figure 9(c)

The graphs show that Google's pretrained dictionary performed best, but it also shows that we approached their level of accuracy simply by expanding our window and vector sizes on our own Wikipedia training data. Given more training data (such as all of English Wikipedia rather than only articles containing acronyms) and perhaps even larger window and vector sizes (and the computation time this would require) we think we could come arbitrarily close or even surpass the provided pretrained dictionary. Another area to explore would be topic modeling, because that would offer some generalization among words, but still provide document-level vectors, instead of word-level embeddings]

We also found that using averaging when combining the embedding vectors provided the best accuracy. Although addition is the recommended method for approximating phrase or short sentences representation vectors, perhaps the fact that we are combining the vectors from so many words means averaging leads to a more plausible final form.

After opting to use the Google News pretrained dictionary with averaging for combining the vector representations, we ran our system on an unseen test set of 9,360 Reuters articles containing 744 discrete examples of acronyms with adjacent expansions. We also relaxed the algorithm for determining the acronyms for which we combine surrounding word vector representations. Previously we used a subset of a given acronym's unexpanded forms within an article as anchor points for finding and combining word vector representations; for the final test we use all of them (at greater computational cost).

Our final accuracy score on this test set was 84.8%. This definitely gives strong evidence that word vector representations generated using a deep learning architecture are useful in real applications. This score certainly beats the baseline of 1/(number of possible acronym expansions), which is at the highest 50% in the case where we always choose from two possible expansions (but which in practice would be considerably lower since we usually choose from among more than two expansions). However, we did not achieve the same level of accuracy with this method as we did with the standard tf-idf method (88.6%).

This effort demonstrates that a relatively large number of word vector representations can be combined (ideally by averaging) to provide a useful representation of an entire sentence or even paragraph of text. We also found the method of attaching an SVM classifier to the end of a pretrained network (in our case, converted to a lookup dictionary, but providing the same outputs as if the network were used in real time) could work.

We are excited to see further results of research in this field, as we feel that word vector representations as generated by deep architectures hold promise for extracting useful knowledge from large datasets, even in an unsupervised setting.

Discussion

Our results supported our hypothesis. We perform 42% better than the baseline with a p-value < .000001, and we believe that our approach could be good starting point for building a useful acronym disambiguation program for general use.

Specifically, our tests have shown that a linear SVM training using L1 loss on tf-idf word frequency vectors performs well for this task. These results are supported by previous research. (i) Linear support vector machines are often used in the related problem of word sense disambiguation.. (ii) L1 loss typically outperforms L2 loss in cases where there are many more features than examples (as in our project).

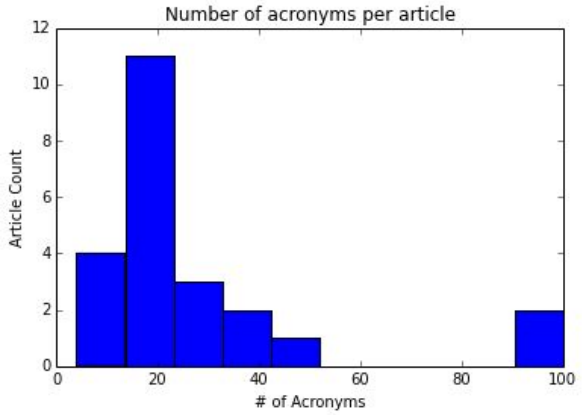
verify

It is worth noting that our results are actually understated, at least slightly. When examining the results case-by-case we found some instances where our system actually chose what would be considered the correct expansion, but which was written in a slightly different way and was therefore counted as incorrect. For example, our accuracy computation would not consider “Industrial Equity Ltd” to be the same as “Industrial Equity Limited”, although it references the same entity. We took some steps to account for this effect, such as comparing only on the first four letters of each word, but some “correct” cases were still reported as incorrect. (Relaxing this equality comparison too much runs the risk of counting some incorrect results as correct.)

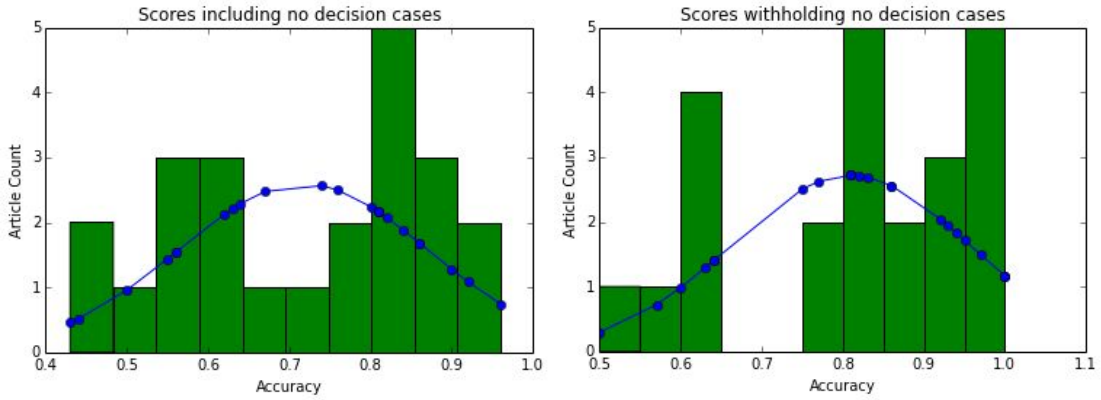
User Testing and Feedback

As a second experiment, using the SVM with L1 loss with tf-idf word frequency vectors, a web application was made to collect user feedback. Over a period of 2 weeks 14 users submitted feedback on a total of 23 documents, mostly academic papers over a wide variety of subjects. This provided a total of 622 acronym instances, with user feedback on both algorithm accuracy and application function.

The median number of unique acronyms (where singular and plural acronyms would be considered different) per document was 17, but this showed lots of variation, with some documents getting up to 100 (the maximum allowed in our system). The overall average was 27 acronyms per document, with a standard deviation of 24.5. The distribution of acronyms per document is shown below.



Scores also showed lots of variation. First, we will go over scores including no-decision cases (cases where the algorithm has not seen the acronym before), and then we will do the same for scores without. The median accuracy including no decision cases was 76%, and the average was 72%, with a standard deviation of 15%. The median accuracy without no-decision cases was 82%, and the average was 81%, with a standard deviation of 15%. These can be seen in the histograms below.

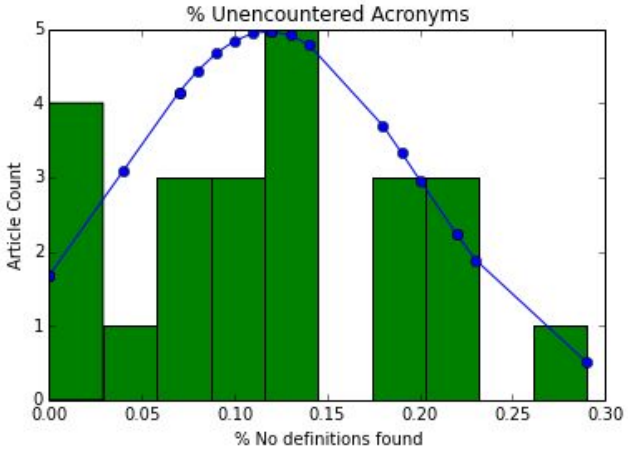


It's interesting to note that the document score distribution does not appear to be normal. Since the acronym expansions are selected based on the tf-idf vector of the entire document, one possible explanation is that the vector is either close to a number of training samples the algorithm has encountered before (i.e. there are Wikipedia pages similar to the document's topic), in which case the algorithm is likely to do well on most acronyms in the document, or the topic is "new territory", and the algorithm is likely to fail on most acronyms in the document.

The total score, across all 622 acronym instances in aggregate, was 74% with no-decision and 86% without. This is significantly higher than the score when averaging the scores per document. This suggests that the algorithm improves when there are more acronyms in the document, which makes intuitive sense, as this is probably a decent proxy for the size of the document overall. A larger document has more context, which may possibly make it easier for the algorithm to accurately decide upon the correct expansion.

It's also interesting to consider the difference in scores between user-submitted documents and the Reuters test set. The Reuters test set average was much higher - 88.6% accuracy with no-decision compared to 81% on user-submitted documents. This also makes somewhat intuitive sense, as the acronyms found in news articles are likely to be much more common than those found in research papers.

We also investigate how often the algorithm encountered an acronym it had not encountered before. On average, the algorithm would not recognize 12% of the acronyms in a given document, with a standard deviation of 8%. Its important to note that this value only demonstrates how often it is that the algorithm had never encountered the acronym before, for *any* context or definition, but it's probably much more common that the algorithm encounters an acronym it has seen before, but has possibly not seen the acronym in the same context, with the same correct expansion. This suggests that the algorithm could improve substantially by being trained on a larger dataset, and perhaps from a wider variety of sources (as opposed to just Wikipedia). The percentages of acronyms encountered with no definitions found per article are shown in the histogram below.



Finally, some of the user feedback was more general in nature. There were instances where the algorithm incorrectly extracted a capitalized word or last name as an acronym. The algorithm skips acronyms shorter than 3 or longer than 8 letters, as these are often not actually acronyms. Roman numerals often trick the algorithm and result in nonsensical expansions. Overall, however, user feedback was positive, and many users noted that the algorithms failed only on very topic specific acronyms.

Discussion

Although our accuracy is somewhat lower on user-submitted documents than on news articles, we still achieve an 81% accuracy on excluding no decision cases. This demonstrates that our

algorithm adapts relatively well to new domains, including obscure and jargon-heavy research material. In addition, the high rate of unrecognized acronyms suggests that the algorithm could improve substantially with additional sources of training data.

Conclusion

The results we obtained (88.6% accuracy on the Reuters test set when “no-decision” cases are thrown out) show that a linear SVM operating on tf-idf word frequency vectors is a useful technique for disambiguating acronyms. Feedback from user testing was generally positive, and we believe this shows the utility of an acronym expanding tool. We believe this shows the utility of our tool.

As for future work, we believe there is promise in utilizing word embeddings for improved generalization, especially in cases where we don’t have much training data for a given set of acronym expansions. Likely, a more sophisticated utilization of them, rather than simply summing/averaging surrounding embeddings and using the result as a feature vector (as we attempted), would give improved results.

Acknowledgements

Thanks to Daniel Rotar for help in developing this paper, experiments, and methodology. Thanks to Morgante Pell for his initial implementation. The machine learning methodology and experiments were advised in part by Professor David Sontag.

Bibliography

Introduction. (n.d.). word2vec - Tool for computing continuous distributed representations of words. Retrieved from https://code.google.com/p/word2vec/.

Sontag, David. Support vector machines (SVMs) [PowerPoint slides]. (n.d.). Retrieved from http://cs.nyu.edu/~dsontag/courses/ml14/slides/lecture2.pdf.

User guide. (n.d.). scikit-learn 0.14 documentation. Retrieved from http://scikit-learn.org/stable/user_guide.html.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representation in vector space. ICLR Workshop, 2013.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality.

Appendix (sample code for test time procedure)

The following pieces of code are provided as examples, from some of our own code, to make replicating the above easier. This code is not fully comprehensive, nor intended to be used as-is, it will need to be adjusted based on file / data types and adjusted to run the different tests involved.

```
import os
import numpy as np
import re
import time
from sklearn.feature_extraction.text import TfidfVectorizer, ENGLISH_STOP_WORDS
from sklearn.metrics.pairwise import pairwise_distances
from sklearn.svm import LinearSVC, SVC
from nltk import distance
from nltk import word_tokenize, regexp_tokenize, clean_html
from nltk.stem import WordNetLemmatizer
import shelve
import glob
import pandas as pd
import string



# Sample code for building necessary functions


def get_acronyms(text): # Find Acronyms in text
   all_english_words = set(word.strip().lower() for word in open(data_path+"wordsEn.txt"))
   english_words = ENGLISH_STOP_WORDS
   pattern = r'\b[A-Z]{3,8}s{0,1}\b'   # Limit length 8
   found_acronyms = re.findall(pattern, text)
   found_acronyms = [acronym for acronym in found_acronyms if acronym.lower() not in english_words]
   found_acronyms = [acronym for acronym in found_acronyms if len(acronym)<4 or acronym.lower() not
in all_english_words]
   return set(found_acronyms)


def definition_patterns(acronym):   # Create definition regex patterns from acronym
   def_pattern1,def_pattern2 = r'',r''
   between_chars1 = r'\w{3,}[-\s](?:\w{2,5}[-\s]){0,1}'
   between_chars2 = r'\w+[-\s]{0,1}(?:\w{2,5}[-\s]{0,1}){0,1}'
   for i,c in enumerate(acronym):
       c = "["+c+c.lower()+"]"
       if i==0:
           def_pattern1 += r'\b'+c+between_chars1
           def_pattern2 += r'\b'+c+between_chars2
       elif i<len(acronym)-1:
           def_pattern1 += c+between_chars1   # acronym letter, chars, periods, space
           def_pattern2 += c+between_chars2
       else:
           def_pattern1 += c+r'\w+\b'
           def_pattern2 += c+r'\w+\b'
   acronym = r''+acronym+r'\b'
   patterns=[]
   for def_pattern in [def_pattern1, def_pattern2]:
```

```
       patterns=patterns+[def_pattern+r'(?=\sor\s{0,1}(?:the\s){0,1}(?:a\s){0,1}'+acronym+r')',
                          def_pattern+r'(?=["(\s,]{2,}(?:or\s){0,1}(?:the\s){0,1}["]{0,1}'+acronym+r
')',
                          r'(?<='+acronym+r'\s\W)'+def_pattern]
   patterns = [re.compile(pattern) for pattern in patterns]
   return patterns


def text_expand(acronym, text, patterns, cut_text=False):   # Search original text for acronyms
    for pattern in patterns:
        pattern_result = re.findall(pattern, text)
        if pattern_result:
            if cut_text:
                text = re.sub(pattern,' ', text)
            return pattern_result[0], text
    return None, text


def same_exp(true_exp, pred_exp):      # Determine if two acronym expansions are the same
    true_exp = true_exp.strip().lower().replace('-',' ')
    pred_exp=' '.join([w[:4] for w in pred_exp.split()])
    true_exp=' '.join([w[:4] for w in true_exp.split()])
    if pred_exp == true_exp:
        return True
    return False



# Sample code for creating the vectorizer

vect_articles = []
while len(vect_articles)<10000:
    k=random.choice(acronymdb.values())[0][1]
    vect_articles.append(articledb[k])

vectorizer = TfidfVectorizer(max_df=1.0, max_features=10000, stop_words='english', use_idf=True,
binary=False, decode_error='ignore')
vectorizer.fit(vect_articles)


# Sample code for scraping acronym / definition pairs

count=0
for zipname in glob.iglob(wiki_path+'*[0-9].txt.gz'):
    if zipname in done_files:
        print zipname
        continue
    count +=1
    if count > max_files:
        break
    print "\nFile number", count, "of", total_files
    scraped_arts, scraped_data = [], []
    f = gzip.open(zipname, 'r')
    articles = f.read().replace('\n', '')
    # All preprocessing here depends on form of downloaded data
    articles = [art.split("]]") for art in articles.strip().split("[[")]   # Subject, article pairs
    if len(articles[0])<2:
        articles.pop(0)
    articles = [[art[0], ' '.join(art[1:])] for art in articles]
    for title, article in articles[1:]:
        if title[:9]=='Wikipedia' or title[-16:]=='(disambiguation)' or title[:5]=='File:' or
title[-4:]=='.css' or title[:8]=='Category' or title[:5].lower()=='media' or
title[:5].lower()=='image' or title[:6].lower()==':image':

            continue
```

```
        print title
        articleid = base64.b64encode('https://en.wikipedia.org/wiki/'+title.replace(' ', '_'))
        acronyms = get_acronyms(article)
        def_count = 0
        for acronym in acronyms:
            patterns = definition_patterns(acronym)
            definition = text_expand(acronym, article, patterns)
            if definition:
                def_count +=1
                scraped_data.append([acronym, definition, articleid, title])
        if def_count > 0:
            scraped_arts.append([articleid, article, zipname])
    for line in scraped_arts:
        line[1] = unicode(line[1], errors ='ignore')
        line[1] = line[1][:30000].encode('ascii', 'ignore')
        line[1] = line[1].translate(table)
        art_writer.writerow(line)
    for line in scraped_data:
        sd_writer.writerow(line)
defs_file.close()
arts_file.close()
```

```
# Sample code for testing tf-idf model


def test_model(text, C=1., loss='l1', kernel='linear'):
    acronyms = list(get_acronyms(text))
    if not acronyms:
        return None
    correct = total = 0
    train_score = []
    for acronym in np.unique(acronyms):
        if acronym not in acronymdb:
            continue
        true_exp, text = text_expand(acronym, text, definition_patterns(acronym), cut_text = True)
        if not true_exp:
            continue
        data = acronymdb[acronym]
        Y = data[:,0]
        if True not in [same_exp(true_exp, y) for y in np.unique(Y)]:
            continue
        num_classes = int(data[-1][-1])+1
        if len(np.unique(Y))<=1:
            continue    # To see only the ones with multi class predictions
#            pred_exp = Y[0]   # Otherwise, predict the only option
#            train_score.append(1.0)   # Training score is obviously 100% in this trivial case
        else:
            total += 1
            X = [articledb[aID] for aID in data[:,1]]
            X = vectorizer.transform(X)
            clf = LinearSVC(C=C,loss=loss)
            clf.fit(X,Y)
            train_score.append(clf.score(X,Y))
            s=vectorizer.transform([text.translate(string.maketrans("",""), string.punctuation)])
            pred_exp = clf.predict(s)[0]
        if same_exp(true_exp, pred_exp):
            correct += 1
            print acronym,':', pred_exp
        else:
```

```
            print acronym+':', pred_exp,';\t', true_exp
    if total>0:
        test_score = float(correct)/total
        return np.average(train_score), test_score, total
    return None



t00=time.time()
article_count=0
train_error, test_error = [],[]
train_scores, test_scores = [],[]

train_scores = []
test_scores = []
for fname in glob.glob(folder_path+'/*.txt'):# test docs
    print '\n',fname
      file_text = open(fname).read()
    file_text = [clean_html(text) for text in file_text.split('<BODY>')]
    for text in file_text:
        result = test_model(text)
        if result:
            train_score, test_score, subcount = result
            train_scores.append(train_score)
            test_scores.append(test_score)
            article_count+=subcount


print "Training Average:", np.average(train_scores)
train_error.append(1.-np.average(train_scores))
print "Testing Average:", np.average(test_scores)
test_error.append(1.-np.average(test_scores))
print "Acronyms Tested:", article_count
print "Time:", time.time() - t00







# Sample code for testing Word Embedding Model

def test_model(text, C):    # Only count multiclass, w/ correct def possible
   acronyms = list(get_acronyms(text))
   if not acronyms:
       return None
   correct = total = 0
   train_score = []
   for acronym in np.unique(acronyms):
       if acronym not in acronymdb:
           continue
       true_exp, text = text_expand(acronym, text, definition_patterns(acronym), cut_text = True)
       if not true_exp:
           continue
       data = acronymdb[acronym]
       Y = data[:,0]
       if True not in [same_exp(true_exp, y) for y in np.unique(Y)]:
           continue    # Don't test acronyms not encountered in training
       if len(np.unique(Y))<=1:
           continue    # To see only acronyms with multi class predictions
       else:
           total += 1
```

```
            X, Y = [], []
            for (d, aid, title, ac) in data:
                train_text = articledb[aid].translate(string.maketrans("",""),
string.punctuation).split()
                vect = [np.zeros(vec_size)]
                for i,wi in enumerate(train_text):
                    if wi == acronym:
                        for w in train_text[i-(window_size/2):i]+train_text[i+1:i+(window_size/2)]:
                            if w in embedding:
                                vect.append(embedding[w])
                if vect:
#                     X.append(np.sum(vect,0))         # Sum method
                    X.append(np.average(vect,0))       # Average method
#                     vect=np.array(vect)
#                     X.append([vect[np.argmax(np.abs(vect[:,i])),i] for i in range(vec_size)]) #
Argmax method
                    Y.append(d)
            clf = LinearSVC(C=C,loss='l1')
            clf.fit(X,Y)
            train_score.append(clf.score(X,Y))
            s = [np.zeros(vec_size)]
#             stext = text.split()
            stext = text.translate(string.maketrans("",""), string.punctuation).split()
            for i,wi in enumerate(stext):
                if wi == acronym:
                    for w in stext[i-(window_size/2):i]+stext[i+1:i+(window_size/2)]:
                        if w in embedding:
                            s.append(embedding[w])
            if len(s)>1:
#                 s=np.sum(s,0)
                s=np.average(s,0)
#                 s=np.array(s)
#                 s=[s[np.argmax(np.abs(s[:,i])),i] for i in range(vec_size)]
                pred_exp = clf.predict(s)[0]
            elif len(s)==1:
                pred_exp = clf.predict(s)[0]
            else:
                continue
        if same_exp(true_exp, pred_exp):
            correct += 1
            print acronym,':', pred_exp
        else:
            print acronym+':', pred_exp,';\t', true_exp
    if total>0:
        test_score = float(correct)/total
        return np.average(train_score), test_score
    return None
```