Classifying Articles as Fake or Real

Language and Statistics
Spring 2007
Eric Davis, Jason Adams, Shay Cohen

1 Project Description

Is it real or fake? That is the question. A discrimination task that may seem trivial to humans can be extremely complicated for a machine. Humans make use of a "makes sense" feature, which relies on world knowledge including Linguistics, to distinguish between real and fake articles. Unfortunately such a feature does not exist for machines yet. As such, to solve a relatively mundane problem for humans, it is necessary to rely on varied and sometimes complicated statistical and linguistic techniques to attempt to allow a machine to make such a distinction. Our goal in this project was to determine what combination of various different statistical and linguistic features would best discover and exploit the deficiencies in a convential trigram model that was used to generate fake articles from a broadcast news vocabulary. We used features that were statistically (backward bigrams for example), syntactically (parsing), and semantically (topic modeling) motivated to try to determine deficiencies in the trigram model. By exploiting these deficiences (namely the inability to track long-distance relations), we hoped to design a program that could discriminate between real Broadcast News articles and articles generated by the trigram model.

The rest of this project paper is organized as follows: §2 describes some preprocessing we did to the data. §3 describes the features that we used to construct a discriminating model. §4 describes the tools we used to actually construct the model. §5 describes the results we had with the best model.

2 Balancing the Training Set

The "holy grail" of most Machine Learning algorithms is that the samples at training and testing time come from the same distribution, usually independently (the i.i.d. assumption). However, in our case, the distributions of number of sentences in each article for the training set and for the development set are very different. In order to accomodate for that difference, we employed a simple algorithm that

BalanceArticles($TrainingSet$, l)

1. $P := TrainingSet$
2. $P^\star := \emptyset$
3. While (true)
   - 3.1. Foreach $Size$ in (1,1,2,3,4,5,7,10,15,20)
     - 3.1.1. $a^\star := \text{argmin}_{a \in P, Length(a) \geq Size, Label(a)=l} \, Length(a)$
     - 3.1.2. If no such $a^\star$ exists, return $P^\star$
     - 3.1.3. $a_0^\star$:= First $Size$ sentences in $a^\star$
     - 3.1.4. $a_1^\star$:= $a^\star$ without the first $Size$ sentences
     - 3.1.3. $P^\star := P^\star \cup \{(a_0^\star, l)\}$
     - 3.1.4. Remove $(a^\star, l)$ from P and add $(a_1^\star, l)$ instead
   - 3.2. End Foreach
4. End While

Figure 1: The algorithm for balancing the training set. The input is the original training set $TrainingSet$ (being pairs of an article and a label) and l, denoting the subset of articles to balance (fake or real). The output is the balanced training set for that subset, $P^\star$. Label(a) is the label of the article a, Length(a) is the number of sentences in an article. The algorithm is run separately for the fake articles and for the real articles.

balances the dataset, so that the distribution of number of sentences in each article will be identical to the one of the development set (and the prospective test set). The algorithm works iteratively, while maintaining a pool of partial articles from the original training set. At each iteration it adds a new article to the new training set from that pool, until there are no more candidate articles to add. The exact algorithm is given in Fig. 2. After running the algorithm, we had 8,897 articles in the training set, with the articles' length distribution in the training set and the development set being identical.

3 Features Extracted from the Training Data

In our experiments, we tried first to extract different features from the data, and check their discriminative power. To do that, we plotted a graph of the density of the features for real and fake articles on the development set, and tried to see if the plots differed significantly. In the following sections we describe the features that we tried.

3.1 Backward Bi-grams

In our experiments, we tried to create a bi-gram model using the additional training data which works from right to left instead of left to right. We thought that such model might give a larger perplexity on the fake articles compared to the real articles, because it would capture different local dependencies from the ones captured by a left to right tri-gram model (namely, the distributions $P(w_{i+1} \mid w_i)$ would be significantly different than the distributions $P(w_i \mid w_{i+1})$.

However, our experiments showed that the perplexity distribution of the articles in the training set for the real articles is almost identical to the one in the development set. We were a little disappointed, but after carefully thinking about it, we realized that such a result is not surprising.

Let $P_A(w_{i+1} \mid w_i)$ be the left to right bi-gram model and $P_B(w_i \mid w_{i+1})$ be the right to left bi-gram model. Let $s = w_0 w_1 w_2 ... w_n$ be a sentence. In that case, the likelihood of the sentence according to model A would be:

$$P_A(s) = \prod_{i=1}^{n} P_A(w_i \mid w_{i-1}) \quad (1)$$

$$= \prod_{i=1}^{n} \frac{P_A(w_i, w_{i-1})}{P_A(w_{i-1})} \quad (2)$$

Similarly, the likelihood of the sentence according to model B would be:

$$P_B(s) = \prod_{i=1}^{n} P_B(w_{i-1} \mid w_i) \quad (3)$$

$$= \prod_{i=1}^{n} \frac{P_B(w_i, w_{i-1})}{P_B(w_i)} \quad (4)$$

This means that the ratio between $P_A(s)$ to $P_B(s)$ would be:

$$\frac{P_A(s)}{P_B(s)} = \prod_{i=1}^{n} \frac{P_A(w_i, w_{i-1}) P_B(w_{i-1}, w_i)}{P_A(w_i) P_B(w_i)} \quad (5)$$

It is sensible to assume that the joint probabilities $P(w_i, w_{i+1})$ would be similar under both models (they are close to MLE estimates which rely on counts). Similarly, it is also sensible to assume that $P(w_i)$ would also be similar for both model A and model B. This means that the ratio in Eq. 5 will be very close to 1, and that checking the perplexity of the right to left bi-gram model is equivalent to checking the perplexity of left to right bi-gram model. However, we already know that the distributions of perplexities given a (left to right) bi-gram for the fake articles and the real article are similar. Sampling from the estimated left to right model gives similar perplexities to the perplexities of real articles.

3.2 Topic Modeling

It is common knowledge that n-gram models do not capture long-range dependencies between words dispersed through the article. In order to exploit this fact, we tried several methods to capture long-range dependencies between words, most of them based on the semantics of words.

3.2.1 LDA Topic Modeling

Using the Topic Modeling Toolkit (Griffiths & Steyvers, 2004) we tried finding hidden topics in the documents and later extracting features using these topics. This toolkit uses LDA to identify such topics in an unsupervised manner.

Specifically, we used the toolkit to identify 100 topics and the 10 most frequently occurring words for each of these topics. Then, we calculated two kind of features (given an article):

- $Topic(t)$ - For each topic t, we calculated the fraction of words in the article that belong to that topic.
- $TopicCover(t)$ - For each topic t, we calculated the fraction of the 10 common words for that topic that are covered by the article (an article "covers" a word if that word appears in the article).

This led to 200 features which we experimented with. Using only these features led to an accuracy of 67% on the development set with *LIBSVM* (see §4).

However, using these features with the stronger features (such as the features composed from a Triggers Network in §3.2.2 and the Self-Triggers feature in §3.2.3) degraded the performance of the stronger features considerably. For that reason, we decided not to use these features.

3.2.2 Triggers Network

We used trigger models to use the occurrence of one word to attempt to predict the occurence of other words. This relies on the observation that for many pairs of words (w_1, w_2) the prior probability of w_1 appearing in a article may be significantly different than the posterior probability of w_1 appearing given that w_2 appeared. For example, seeing the word "STOCKS" may significantly increase the probability of seeing the word "RISE".

The difference between the prior and posterior probabilities can capture long-range dependencies, which is what we planned to exploit. To do that, we define for an article $d = (w_1, w_2, ..., w_n)$ its *trigger network*. A trigger network $G(d, r, s)$ is an undirected weighted graph. The nodes in the graph are the non-function words in d and of words in d, w_i and w_j, with $r \leq |i - j| \leq s$ there is an edge. The trigger network as a whole captures how consistent the selection of words in the article with respect to a topic.

- r - The trigger network connects words that are at least of distance r from one another. The reason for using such lower bound is to avoid modeling the semantic consistency which exists locally with n-gram models; words very close to each other in a sample from n-gram model are likely to be semantically linked.

- s - The trigger network connects words that are at most of distance s from one another. The reason for doing so is mainly compuational: the size of the graph grows linearly with s. If we chose s to be the size of the article, the size of the graph would be quadratic in the number of words, making it hard to perform calculations on.

- The weights - the weights between the words represent how semantically linked two words are. We used pointwise mutual information to denote the strength of the link, based on joint distributions of words on articles calculated from the large news wire corpus.

Once we have a trigger network for a article, we can perform all kind of calculations on it. In our experiments, we first tried to use the average weight of all

edges in the trigger network as a feature in the model (with $r = 5, s = 15$, for computational reasons). This led to an accuracy of 82% on the development set, together with the self-triggers feature (see §3.2.3).

We later postulated that weights coming from small distances between words may need to be treated differently than distances of larger value. In order to capture this difference in our model, we added 11 features to the model, for each distance between 5 to 15. Each such feature was the average weight of all edges of that distance. When doing so, our performance improved to 82.5% on the development set.

3.2.3 Self-Triggers

Another kind of long-range dependency that is not captured by n-gram models is the "burstness" of a word in an article. Namely, a word that appeared once in an article is more likely to appear again in the same article. To capture this kind of dependency in our model, we calculated the ratio between the types and tokens for each article and used that value as a feature.

We noticed that this value changes as the articles become longer. Specifically, the longer the article, the smaller this value is. In order to capture the dependency of that value on the article's length in our model, we did the following: for each length between 1 to 700, we sampled 500 articles from the large corpus, and for each of them we calculated the ratio between types and tokens. Then, we created a new feature for each article d which estimates the following p-value using the samples:

$$p(d) \quad = \quad P(U < u(d) \mid |d|) \tag{6}$$

where U is a random variable denoting the ratio between tokens and types in articles, $u(d)$ is that value for a article d and $|d|$ is the number of words in d.

The smaller $p(d)$ is, the more we expect the article to be a real one, because there were not too many articles in the samples of articles of that length that had less types. Unfortunately, this feature degraded performance. It seems like this feature still does not discriminate between fake and real articles for articles of short length. We therefore decided not to use that feature, and instead we used the original simple feature which calculates the ratio between types and tokens.

3.2.4 Using EM for Topic Modeling

One might assume that there is a hidden variable coming with each article which is its topic. The distribution of words appearing in an article will change according to that topic. Specifically, similarly to with pLSA (Probabilistic Latent Semantic Analysis, (Hofmann, 1999)) we may have a generative model which first generates a topic and then generates a bag of words $d = (w_1, ..., w_n)$, being the article, according to that topic t:

$$P(t, d) = P(t) \prod_{i=1}^{n} P(w_i \mid t) \tag{7}$$

Using the training data, we need to estimate the values of $P(t)$ and $P(w \mid t)$. Since t is a hidden variable we have to resort to techniques such as EM to estimate these parameters. Indeed, we derived the following EM algorithm:

$$P^{(l)}(t) = \frac{1}{n} \sum_{i=1}^{n} P^{(l-1)}(t \mid d_i) \tag{8}$$

$$P^{(l)}(w \mid t) = \frac{\sum_{d_i : w \in d_i} P^{(l-1)}(t \mid d_i)}{\sum_{d_i} |d_i| P^{(l-1)}(t \mid d_i)} \tag{9}$$

where l is the iteration number, d_i is the ith article in the training set and $P^{(l-1)}(t \mid d_i)$ is calculated based on the model parameters at iteration $l - 1$ using the following formula:

$$P(t \mid d) = \frac{P(t) \prod_{i=1}^{n} P(w_i \mid t)}{\sum_t P(t) \prod_{i=1}^{n} P(w_i \mid t)} \tag{10}$$

Once we are equipped with the parameters for the joint model $P(t, d)$ we can perform all kind of calculations to estimate the probability of an article being fake. For example, given an article, we can calculate $P(t \mid d)$ using Eq. 10 for each topic, and see whether this probability distribution is closer to the topics distribution of the fake articles (in which case we would classify it as a fake article) or closer to the topics ditribution of real articles (in which case we would suspect it is a real article). The notion of how close two distributions are can be implemented by calculating KL-divergence between them.

Unfortunately, implementing this EM algorithm requires tedious careful programming which we did not have time for. However, we did feel the idea is worth mentioning, and given enough time, we believe it could give quite good results.

3.3 Linguistically Motivated Features

We learned in class that despite good intentions of modeling language with linguistically motivated features, e.g., decision trees, the end result is still not as good as a "simple" n-gram model. However, this does not imply that linguistically motivated features, such as part of speech cannot aid in building a better overall model. With this motivation in mind, we looked at two main linguistic features (each main feature had multiple sub-features): part-of-speech tagging and parsing.

3.3.1 Part-Of-Speech Tagging

We hoped to find a difference between the real and fake articles in terms of part of speech distribution. To take a closer look, we used MXPOST, a maximum entropy part-of-speech tagger (Ratnaparkhi, 1996). The off-the-shelf tagger was trained on sections 0-18 of the Penn TreeBank.

Our data is fairly similar in nature to the material in the Penn TreeBank in terms of style and content. However, we wanted to normalize the data. To do this, we removed punctuation from the training data and transformed each character to uppercase. We then retrained MXPOST on sections 00-25 of the Penn TreeBank. After retraining the POS tagger, we separated the training data and development data into sentence from real articles and sentences from fake articles. Then we ran MXPOST on the real and fake sentence from the training set and the development set.

There were 33 POS, including function words such as determiners (DT) and cardinal numbers (CD) as well as content words, such as nouns (e.g. NN, NNS) and verbs (e.g. VB, VBZ). First we looked at the ratio of function words to content words per sentence. Our intuition was that perhaps the sentences generated by the tri-gram model would have a higher ratio of function words to content words because the function words occur much more frequently in the data. However, this belief turned out to be incorrect, as the ratio of content words to function words was approximately equal for sentences from both real and fake articles. Next, we looked at the relative frequency of the 33 POS tags regardless of position in the sentence. Again, we did not find anything significant, as the frequencies for each POS were almost identical for fake and real articles. Fig. 3.3.1 describes a graph of five common POS tags and their frequencies for both real and fake. In addition, we thought that there might be a difference between real and fake in terms of the frequency of POS by position in the sentence. We hoped that certain POS such

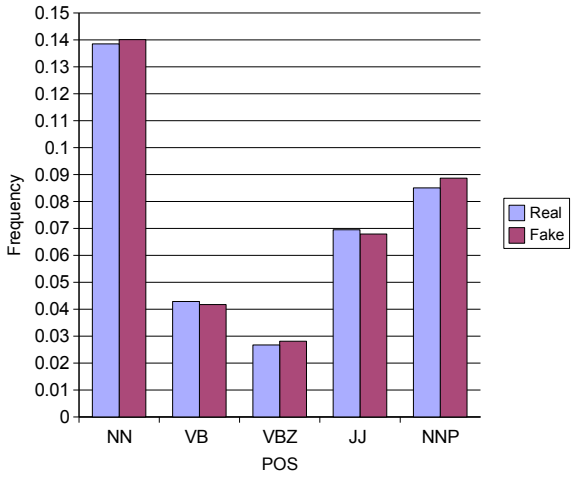

Figure 2: Distribution of salient tags for the real articles and the fake articles

as proper nouns and prepositions would occur much more frequently in certain positions in sentences from real articles versus sentences from fake articles or vice versa. Unfortunately, this did not pan out either. The distributions for some POS (namely prepositions and proper nouns) were not identical, but they only differed by .02 or .03, so we decided that the difference was not statistically significant. Finally, we looked at the ratio of proper nouns to all other words in a sentence. Our hope was that since the trigram model cannot model long distance relations, it would not be able to "remember" an antecedent and would thus use more proper nouns than would be expected in real sentences. Our assumption was not met again, however.

Despite our best efforts, we could not derive any useful features from POS tagging. We believe that this is the case because the POS tagger training relies on local dependencies. The tagger derives the next part of speech from the previous POS or previous 2 parts-of-speech. As such, it did not help us capture any of the long distance dependencies, thereby helping to distinguish fake sentences generated by the distance handicapped trigram model and real sentences without such limitations. Also parts-of-speech depend more on local relations (for the most part, excluding antecedents), so it is not surprising that there was not a huge difference between the distribution of POS tags of sentences from real articles and sentences from fake articles.

3.3.2 Parsing

Despite the shortcomings of the POS tagging approach described above, we were not ready to give up on linguistically motivated features. Our next inspiration came from Charniak (2001) and Ahmed, Wang, and Wu (2006). The Charniak parser can be used as a language model of sorts, and the parser outputs perplexity of sentences based on an immediate-bihead grammatical model and an immediate-trihead model. This parser is especially promising because it improved on a the trigram model base-line by 24 percent and improve on the grammar-based language model by 13 percent.

In order to use the parser as a language model, we retrained it on sections 00-19 of the Penn TreeBank and tuned it on sections 20-25. We then parsed both the training and development sets. For each sentence, the parser returns a parse for that sentence (or fragment) as well as the average log-probability from the grammatical model, the average log-probability from the trigram model, and the average log-probability from a mixture model. Our hope was that the parser would help us capture long distance dependencies and that there would be a significant difference between the real and fake articles in terms of perplexities and average log-probabilities.

First we looked at the average log-probabilities for the trigram model, the grammar model, and the mixture model. We plotted a graph of the above probabilities for real versus fake and determined that the average log-probability of the trigram model showed the most promise (real articles were much more likely in a range between -200 and -150 or so). This was our first parse-based feature. Then, we looked at the maximum log-probabilities of the 3 above-mentioned models of a single sentence within a single article. This gave us 3 more parse-based features: maximum log-probability for the grammar model per article, maximum log-probability for the trigram model per article, and maximum log-probability for the mixture model per article.

Next, we looked at the minimum log-probabilities for the grammar and trigram models per article (we determined that the minimum log-probability for the mixture model was not significant). This gave us two more features. Finally, we looked at the maximum log-probability for the grammar and trigram models per article averaged over all the words of a single sentence (as above the average per word log-probability for the mixture model did not prove to be significant). This gave us two more features, and we now had eight parse-based features. Upon further review, we noticed that the average log-probability of the trigram model per sentence did not help in discriminating real versus fake articles as much as we

had anticipated. As such, we removed this feature. Thus, in the end, we had 7 parse-based features:

- maximum average log-probability for the grammar model per article
- maximum average log-probability for the trigram mode per article
- maximum average log-probability for the mixture model per article
- minimum average log-probability for the grammar model per article
- minimum average log-probability for the trigram model per article
- maximum average log-probability for the grammar model per article per word
- maximum average log-probability for the trigram model per article per word

Fig. 3 shows two graphs for real versus fake articles for minimum average log-probability for the grammar model and maximum average log-probability for the trigram model per article per word.

Thus, it is our belief that parsing the sentences and measuring the average log-probability of the above models helped us capture some of the weaknesses of the trigram model: namely long distance relations. Short sentences (five or six words or less) still pose a problem, but the parser should show less "surprise" in parsing a sentence from a real article versus parsing a sentence from a fake article. We believe that this is clearly reflected in the average log-probabilities of the above three models.

3.3.3 Miscellaneous Linguistic Features

After examining the data, we noted several grammatical errors occurring in the fake articles that we thought we could exploit with some additional features. These features involved pronoun case violations following pronouns, distance between wh-words (such as *who* and *what*), distance between wh-determiners (such as *which* and *that*), and distance between stop words. We used a stop word list of about 300 words consisting mostly of function words and closed-class words. These features were calculated for each sentence in a document, then averaged across all sentences in the document to produce a single, document-level feature for each. It is our hope that these features occur in different distributions in real and fake documents and so will improve the classifier.

(A) (B)

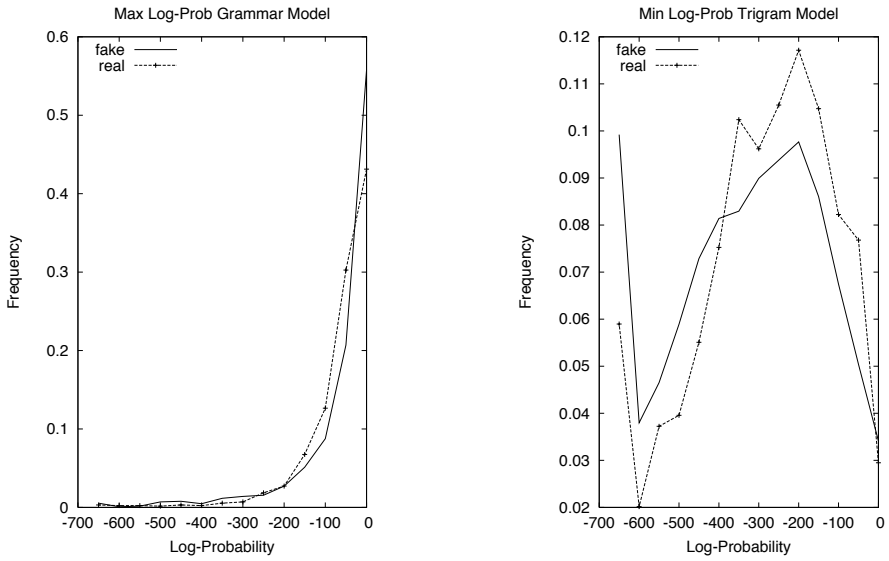

Figure 3: (A) Distribution of minimum average log probability for the grammar model. (B) Distribution of maximum average log probability for the trigram model.

To compute pronoun case violations, we looked for co-occurrences of prepositions and pronouns in each sentence. A pronoun case violation is the occurrence of a pronoun in the nominative case (such as *I*, *we*, *she*, etc.). This is not impossible or else the trigram model would never see it, but it seemed to occur in more impossible settings in fake articles. For each co-occurrence, a count of one was registered. These counts were then normalized by the length of the sentence and again by the length of the document. Similarly, distances for the other features were calculated by finding the average distance from one occurrence to the next in a sentence. In this case, the values were first normalized by the total number of occurrences of the phenomena being measured (be it wh-words, wh-determiners, or stop words). The final values were normalized by the number of sentences in the document.

3.4 Term Frequency - Inverse Document Frequency

In document classification, the co-occurrence of certain word types is a strong indicator for the topic of the article. In information retrieval, one common method of weighting word types that occur in a document is by using a term frequency - inverse document frequency score. The term (word type) frequency for term i in a document is straightforward:

$$tf_i = \frac{count(w_i)}{|W|}. \tag{11}$$

In other words, it is the number of times the term appears in the document divided by the total number of tokens in the document.

The inverse document frequency is a measure that seeks to capture how valuable a term is. The more documents a term appears in, the less information it offers in distinguishing between document topics. The inverse document frequency for all documents d containing term i is calculated as follows:

$$idf_i = log\frac{|D|}{|d|}. \tag{12}$$

That is, the log of the total number of all documents divided by the number of documents containing term i. And finally,

$$tfidf = tf \times idf. \tag{13}$$

Our hypothesis was that real articles would maintain some coherency with document topic while fake documents would not. Unfortunately, using terms as

features weighted by their tf-idf scores only resulted in confusing the classifier. The term features overwhelmed the more discriminative features and resulted in very poor accuracy. We suspect this is due to the fact that while real articles have higher topic cohesion, the fake articles trigger so many different topics that there is no clear topic division. Also, tf-idf is usually used for discriminating between topics, while *real* and *fake* are not topics themselves. That is, there is no topic to distinguish. Therefore, based on both theoretical and empirical grounds, we believe tf-idf is not particularly useful for this task.

4 Classification Tools

For this project, we used off the shelf modelling tools including:

- *Logistic Regression* - Logistic regression model is a specific case of an exponential model (MaxEnt model), in which the predicted outcome is a binary variable (or a variable from a limited domain). We used the *logreg* version 1.1 package[1].

- *SVMLight* - *SVMLight* (Joachims, 1999) is an implementation of Support Vector Machines, which have been successfully applied to many domains, including language. In principle, Support Vector Machines are discriminative linear classifiers (though the space where the linear classification is actually performed does not have to be Euclidian).

- *LIBSVM* - *LIBSVM*[2] (Chang & Lin, 2001) is another implementation of SVMs. We noticed that there is a performance difference between *SVMLight* and *LIBSVM* even when using the same parameters. We are not sure what is the source of the difference, but it gave us a good reason to inspect both packages. We suspect the difference arises in the optimization and termination criteria.

After experimenting with the above three classifiers, we decided to use *LIBSVM* as our final classifier. *LIBSVM* had a performance advantage over both of the other packages, and a more specific advantage over the *SVMLight* package: it outputs probabilities which are in the $[0, 1]$ domain. While automatically producing probabilities was convenient, it was not strictly necessary, since there is

[1] *http://www2.nict.go.jp/x/x161/members/mutiyama/software.html*
[2] *http://www.csie.ntu.edu.tw/ cjlin/libsvm/*

rbf	-	$e^{-\gamma\|u-v\|^2}$
polynomial	-	$(\gamma u'v + c)^d$
sigmoid	-	$tanh(\gamma u'v + c)$

Figure 4: Kernel methods used by LIBSVM and SVMLight.

a standard way to obtain probability estimates from SVM output using the *pool-adjacent-violators* algorithm (Zadrozny & Elkan, 2002). One of the main benefits of using an SVM is the use of a kernel function. The kernel function allows the classifier to transform a non-linearly separable feature space into one that is linearly separable. In § 4.1, we discuss the kernel functions we tried and further issues with using SVMs, such as parameter tuning. In § 4.2 we discuss methods we used to obtain probability estimates from the classifiers.

4.1 Kernels and Parameter Tuning

Support Vector Machines seek to find the maximum margin hyperplane that separates examples in the feature space. Often, feature spaces are convoluted and there is no hyperplane that separates the data correctly. SVMs offer two key features that allow them to cope with these situations. The first feature is the use of slack variables, by which they can essentially *ignore* examples that do not conform to the general trend of other examples. The second feature is the use of a kernel function. The kernel function transforms the feature space into one that is linearly separable. In our experiments we tried all the standard kernels that came with the SVMLight and LIBSVM packages: linear, radial basis function (rbf), polynomial, and sigmoid.

The linear kernel is the identity kernel. It does not transform the space in any way and the SVM finds the hyperplane directly. The functions for the other kernel methods are given in Fig. 4. Each kernel has certain parameters that can be tuned to a certain data set. In some cases this will result in overfitting, but generally when the test data is drawn from a similar distribution as the development data used for tuning, the amount of overfitting is not significant.

4.2 Probability Estimates from SVMs

In the case of LIBSVM, probability estimates are given as part of the output so no further work needed to be done with those results. However, we wanted to evaluate SVMLight as well, which outputs the distance from the test example

to the hyperplane. This is a sort of confidence value for the classification based on the training data. Different approaches have been used in the past, such as applying the logistic function to the output:

$$f(x) = \frac{1}{1 + e^{-ax+b}}. \tag{14}$$

This results in values in the interval $(0, 1)$ with numeric input.

An alternative method for computing probabilities that has been reported to match the actual probability distribution more closely is found using the *pool-adjacent-violators* (PAV) algorithm (Zadrozny & Elkan, 2002). This algorithm performs isotonic regression on the labeled SVM output to produce a monotonically increasing list of probabilities. First, the output of the SVM is sorted in increasing order. The correct labels for each example are matched up to each SVM output value. If this list is increasing, nothing need be done since the SVM correctly classifies everything as negative below a given value and positive above it. The probabilities are 0 and 1. However, it is more likely that there will be a large number of adjacent violators, where the list of labels are not monotonically increasing. Starting from left to right, the PAV algorithm replaces any two such violators with their average. The algorithm then backtracks, finding any such violators from that position that were created after averaging them. If there are any right-to-left violators, all violators in the growing pool are replaced by their average. This continues until there are no more violators or the algorithm has reached the beginning of the list. Execution then resumes from left to right until the end of the list has been reached and there are no more adjacent violators. The resulting list can be used to create bins for estimating the probability of new examples.

5 Results

5.1 Final Model

The final model we used was a mixture of two models for long and short sentences. We chose the LIBSVM package due to its convenient probability estimates and higher classification accuracy on the same parameters. We found that one set of features (Fig. 5) yielded better classification accuracy on shorter articles (less than 150 words), while only the first two of those features yielded better results on longer articles when used by themselves. The shorter article model used the polynomial kernel with degree three, γ of 1.0 and a constant value of three (see

- Uniqueness - type to token ratio (§ 3.2.3)
- Triggers Network (§ 3.2.2)
- Parse Features (§ 3.3.2)
- Miscellaneous Features (§ 3.3.3)

Figure 5: Features used in the final model.

Data Set	Accuracy	Log Perplexity
Dev Set	86.5%	-0.37649
Training Set	85.4%	-0.33663

Figure 6: Final results.

Fig. 4). The longer article model used the linear kernel, which has no additional parameters. It is also possible to tune the cost of positive over negative examples, however we found that it was better to leave the cost at the default value of one.

Our final results are reported in Fig. **??**. We calculated accuracy and the average log posterior of the correct label on the development and training sets. While the results for our development set were slightly higher, we did not feel that overfitting was too great a concern. We do expect there to be some overfitting and we expect our test results to be lower than our development set results. As articles become longer, the classification task becomes easier, as is evident in Fig. 7.

5.2 Balancing the Results

One common thing to do when constructing a model is to take into account any additional constraints about the form of the model, if we know they exist. For example, if we know that the model's parameters come from a certain subset of parameters of the model family, we will constrain the model to be parameterized according to parameters from that subset.

This comes into play in our case when making predictions. When predicting the labels for the development set, we know what the distribution of the *fake/real* labels in the predictions should be , and we can fit the model to adhere to these constraints.

Since in our case, for each article length, half the articles should be fake and

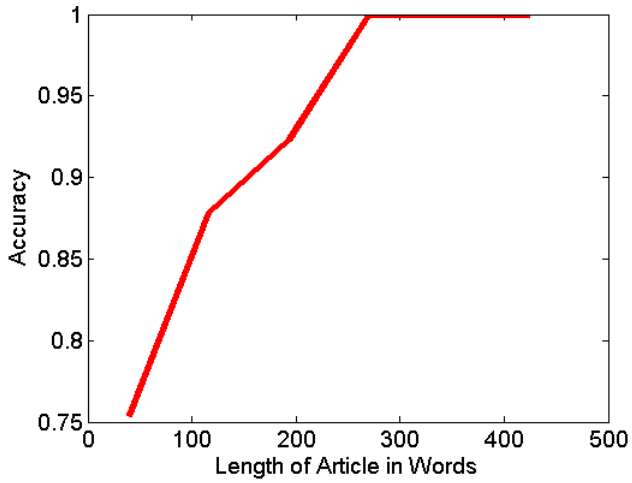

Figure 7: Classification accuracy by article length in words.

half the articles should be real, we tried to balance the predictions according to the following method:

1. Calculate the predictions of the best model constructed on the development data.

2. For each article length l, calculate the median of the probability of being real article in the development set, $p(l)$.

3. For all articles of length l with probability of being a real article higher than $p(l)$ according to the model, attach the label *real*. Attach the label *fake* for the rest.

We hoped that by constraining the output to be from the exact distribution we expected, our performance would improve. Unfortunately, this did not happen, and in fact our performance degraded by about 2%. We are not sure why this happened, but we suspect that it might be the result of the i.i.d. assumption: the probabilities calculated for each article are independent from the rest, and therefore should not be considered together with the rest as we did.

6 Conclusion

This project was challenging. Although the task seems easy at first glance (especially since it is so easy for humans), we found out that it is quite hard for machines, especially for short articles. In fact, the final model made most of its errors on short articles.

The fact this task is actually a challenge for computers should come as no surprise, if one thinks about it more carefully. In general, n-gram models are currently the state-of-the-art models for language modeling. If it were easy to distinguish an n-gram model generated article from a real article, it might be possible to improve the language model using techniques used in the classification.

Yet, we achieved a good accuracy on the development set (in the mid 80s). This indeed brings into consideration the question of improving a language model using the techniques mentioned in this paper, such as topic modeling or syntactic modeling. In fact, it should come as no surprise that there is on-going research about using long-distance related features (such as syntactic or topical features) to improve language models over n-gram models, as was mentioned in class.

7 Comments and Suggestions

So where does this leave us? Not surprisingly, machines do not classify as well as humans... yet. However, with some clever heuristics and features that make use of real-world knowledge and language models, a high classification accuracy is possible. As computer become more and more powerful, more subtle properties of language (that may only be subconscious knowledge for humans) may become apparent, thereby raising the accuracy of the decision task even further.

We thoroughly enjoyed this project. First and least important, reading news articles from around 1996 proved highly entertaining. Next, it is always nice to gain hands-on experience using the statistical techniques that we learned in class. Lecture is one thing, but language modeling, n-grams, and other statistical techniques do not truly come to life, until they are applied to an actual project. Finally, who doesn't love a good challenge? This project proved to be quit difficult, but we gained confidence with each new feature that added a percent or two of discriminatory power.

We have a couple suggestions to improve this project for next time. As with anything that is difficult, more time would prove beneficial. We were hoping to implement EM, but as mentioned above, we just did not have the time to ensure

that it functioned correctly. An extra week would have been more than enough to resolve any time issues. Finally, we understand that consistency is important, but it might be interesting to have articles from multiple sources. For example, we would enjoy articles from periodicals and other "lighter" sources. It would be interesting to see if there would be any differences in discriminating between the various types of articles. Overall, this is a well-designed project, and the competition with other teams does nothing but add to it.

Acknowledgments

The triggers network construction was partially inspired by a previous year project cited below.

The syntactic features (parsing and POS tagging) were also partially inspired by a project from last year as cited below.

References

Ahmed, A., Wang, M., & Wu, Y. (2006). *Detecting fake from real article.* Language and Statistics Course.

Chang, C.-C., & Lin, C.-J. (2001). *LIBSVM: a library for support vector machines.* (Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`)

Charniak, E. (2001). Immediate-head parsing for language models. In *Meeting of the association for computational linguistics* (p. 116-123).

Griffiths, T. L., & Steyvers, M. (2004, April). Finding scientific topics. *Proc Natl Acad Sci U S A*, *101 Suppl 1*, 5228–5235.

Hofmann, T. (1999). Probabilistic latent semantic analysis. In *Proc. of uncertainty in artificial intelligence, uai'99.* Stockholm.

Joachims, T. (1999). Making large-scale support vector machine learning practical. *Advances in Kernel Methods: Support Vector Learning*, 169–184.

Ratnaparkhi, A. (1996). A maximum entropy model for part-of-speech tagging. In E. Brill & K. Church (Eds.), *Proceedings of the conference on empirical methods in natural language processing* (pp. 133–142). Somerset, New Jersey: Association for Computational Linguistics.

Zadrozny, B., & Elkan, C. (2002). Transforming classifier scores into accurate multiclass probability estimates. In *Kdd '02: Proceedings of the eighth acm*

sigkdd international conference on knowledge discovery and data mining (pp. 694–699). New York, NY, USA: ACM Press.