REASONING-BASED ADAPTIVE LANGUAGE PARSING

Jonathan A. Zdziarski

ABSTRACT

Modern day language classification employs conceptual machine learning, which relies heavily on the quality of data that can be extracted from the input text. Data is typically extracted using a parser with a static set of parsing rules. This presents a problem for learning machines needing to parse different languages, many with different structural rules. Hand-written rules can also provide less-than-optimal parsing even for languages they were designed for, as fractions of data can sometimes be more useful than complete recognized words. This paper outlines a reasoning-based approach capable of being added to an existing statistical language classifier to intelligently identify the best characters or pattern to use as text separators, and therefore the best overall technique for parsing a corpus of text. This technique applies to the building and further revising of a tokenizer's parsing technique using learned information about the effectiveness of each potential text delimiter, pattern, or expression. Moreover, this technique allows for constant adaptation to new classes of text and/or character sets to continually improve upon its own effectiveness. The benefit of this application is an approach to intelligent parsing without any prior knowledge of the input language(s). As this paper will show, using an adaptive, reasoning-based parser can provide equal or greater efficiency than a static parser, especially when dealing with foreign languages and character sets. At the lowest level of explanation, the described approach causes a language parser to constantly reprogram itself to break up data that it does not find useful until it finds a way to parse that data so that it is considered more useful than the rest.

Categories and Subject Descriptors

I.5.1 [**Pattern Recognition**]: Implementation – s*tatistical, structural.*

I.7.m [**Document Text and Processing]**: Miscellaneous

General Terms

Algorithms

Keywords

parsing, adaptive, token, delimiter, classification, text, filtering

1.INTRODUCTION

Reasoning machines bear the task of improving their responses regularly to provide better results. Because the quality of the input data is so closely related to the quality of the classification, classifiers should also be able to detect, or try to detect, their level of efficiency at extracting quality data from the input sample. Poor data can lead a classifier to cause errors, simply by "looking at the wrong things" – the garbage in, garbage out principle[2]. Manual tokenization is like manual filtering. A manual tokenizer can only be improved reactively, providing improved results only after experiencing an error[5][6]. Statistical language classifiers typically employ a static set of rules to parse messages presumed to be in a supported target language. Such classifiers use a language parser to generate "tokens", or small pieces of representative data from the input text, which are then used for analysis. These are typically words, short phrases, patterns, or expressions. Data is often generated using a static set of word separators or expressions suitable for the language being parsed. The problem with this approach has historically been that what may appear to represent sensible parsing rules for a given language may not necessarily provide the optimum data for a statistical classifier. What's more, language classification is a large-scale problem, and different languages require different parsing rules. Some languages do not provide whitespace characters, making it very difficult to parse a text without prior knowledge of the language it is composed in.

To adapt a parser to identify the optimal way to read a given input text, a method is used to reason the best parsing technique for the given input. The philosophic rationale behind this is to break down *syntactic synonymy* (structural signs which convey similar meaning[8]) within a document, only the synonymy here is based on relevance more so than meaning; that is, to identify the most unique structures and parse the message according to these rules rather than basic rules of language. This is accomplished by calculating a "usefulness" vector for every potential parsing technique. In our examples, these techniques will be choices of token separator and/or expression, however they need not be limited to simply delimiters, as we'll explain in sections two and five.

The value calculated for each parsing technique predicts the likelihood that the character will be found within data of little interest to the classifier. The characters with the highest likelihood of being found in uninteresting data are then used as parsing separators (word delimiters) and the remaining characters (those likely to be found in interesting data) are treated as constituent characters.

The only requirement to implementing this technique is that the language classifier must have a mechanism to

determine whether data is of high interest or low interest – a common function of any classifier. Most language classifiers do this by means of assigning a probability to a token in the range of 0.0 to 1.0, where tokens having probabilities closer to 0.5 for all classes are considered uninteresting, and tokens with probabilities closer to the edges are considered of high interest. Alternative mechanisms may be used as well, so long as the filter is capable of placing a given token within such a dichotomy. The mechanics of choosing the thresholds for this are left up to the implementer.

2.ESTABLISHING DELIMITER SETS

The concept of a token delimiter is common among most language classifiers. Delimiters are used to determine how to parse a sample text into smaller components – words, phrases, and the like. Delimiters are sometimes elusive, especially when parsing languages without whitespace, such as Asian languages. These languages traditionally have required the identification of words for parsing[10]. The goal of establishing a delimiter set (or other adaptive parsing technique) for such languages, here, is to isolate individual words or other interesting structures without knowledge of a specific language's lexicon.

The parser (also known as the tokenizer) is by far the most heuristic, hard-coded component of most classifiers, and arguably the central plane of error[4][3]. Data that is considered "uninteresting" is largely useless to a classifier, and therefore it is advantageous to have less "uninteresting" data (typically data with a probability closest to neutral), and more "interesting" data[3]. Uninteresting data is considered such because it appears in a similar proportion across the different classes being categorized (for example, spam and non-spam). By altering the method in which data is parsed, data that would otherwise be considered uninteresting might, instead, be parsed in a different fashion that is more native to one particular class of data.

The decision for which delimiters to use for a given classification process is reached by building a hypothesis space containing all possible delimiters for consideration and then calculating the probability that each delimiter will be present in uninteresting data. Depending on whether ASCII characters, wide characters, or other such devices are used to fill the hypothesis space (such as patterns or expressions) - is relevant only to the extent of granularity and resources the implementer wants to account for. Every potential token separator (hypothesis) retains a memory of the number of times it has appeared in a "high interest" data point (or token) and a "low interest" data point. A global statistic is also maintained, keeping track of the total number of "high interest" and "low interest" tokens analyzed in the filter's history. This memory can and should be unique to different realms of data, such as header analysis vs. body analysis, and within the same scope as the tokens themselves (e.g. per-user, global database, etc.). For data sets spanning many different languages and/or character sets, a separate tokenizer memory for each language may help improve and maintain performance. As classification progresses, the tokenizer incrementally enhances its knowledge of the target language's construction[7], from a machine's point of view.

For each potential token separator (delimiter), the Bayesian Chain Rule is used to calculate the probability that the delimiter will appear in a low-interest token:

$$\text{probability(delimiter)} = \frac{\text{LowCounter} / \text{TotalLow}}{(\text{LowCounter} / \text{TotalLow}) + (\text{HighCounter} / \text{TotalHigh})}$$

Where LowCounter and HighCounter are the number of occurrences that the given character has appeared in each class of token, and TotalLow and TotalHigh represent the total number of such tokens that have been analyzed. This is the same formula cited in [6] to calculate the disposition of a token.

Using this formula, we could calculate, for example, the likelihood that every ASCII character from 1 – 255 would appear in an uninteresting token. We then take the *n* tokens with the highest probability and use those as token delimiters. After each message has been processed, the LowCounter, HighCounter, TotalLow, and TotalHigh counters are updated in the tokenizer's memory. For example, if the message processed saw that the letter A was present in 10 tokens that were interesting and 5 tokens that were uninteresting, HighCounter is incremented by 10 and LowCounter is incremented by 5 within the tokenizer's memory.

2.1.Bootstrapping

When there is not enough data to establish a full delimiter set, two options can be employed. The first option is to simply use a default delimiter set, such as whitespace, until a reasonable number of adaptive delimiters can be established. This could prove problematic, however, for languages not employing whitespace. The second, and more statistically sound technique, is to use a limited delimiter set. In this scenario, characters that do not have enough data gathered for them are assigned a default probability of 0.5, which will ensure that there are always token separators in the pipeline. As the tokenizer begins to make initial observations about the sample text, more useful delimiters will quickly resolve to a more useful probability. This approach is referred to in this paper as "pure" analysis, because the delimiter set is entirely derived from means of learning. For the tests conducted here, we determine a token to have sufficient data when

LowCounter and HighCounter are greater than zero, and LowCounter + HighCounter > 10.

2.2.Alternative Delimiter Hypothesis Spaces

This paper presumes to use the standard ASCII character set to derive its delimiter set from. While many languages employ multi-byte characters, using an ASCII set remains to work quite well as we'll illustrate in this paper. This could be due to its machine-native range, in that is covers all possible values for a single byte, or possibly that because the text is being machine-processed, sub-character data shows to be sufficiently interesting.

Other techniques for building a hypothesis space for delimiters may include using the wide character data type (wchar) or possibly even multiple byte combinations for delimiters. Another technique may involve maintaining a hypothesis space with duplicates of each delimiter and an identifier determining whether to split before, after, or on the delimiter. Additional delimiter techniques will be explored in section 5.

3.ANALYSIS

The overall process can be summarized in this fashion:

```
load memory
for each character
  calculate probability(delimiter)
  if probability(delimiter) >= 0.5
    add to sort
end
for top windowSize in sort
  [sort by probability descending]:
  add to token separators list
end
```

Fig. 3.0.1 Analysis Process

The windowSize vector represents the sort window size, and ultimately the maximum number of character delimiters to be used. The tests outlined in this paper specify the different window sizes tested with W*n* denoted in the test.

3.1.Tokenizer Example

This adaptive approach allows the machine to choose the best text delimiters it believes will parse the sample to provide the highest level of confidence in its decision. The following example is the final delimiter set chosen in one configured run of the SpamAssassin corpus (described in section four).

Header Delimiters:

```
378z049Y;mF:w"O@!^N\%$(>
```

Body Delimiters:

```
T?N,I?OS.pEmroaicthldesn
```

Many characters we would expect to see are present in one or both sets of delimiters. For example, the "at" sign (@) allows headers to be parsed for address and domain information, and the colon (:) is automatically selected as a general separator in headers. The exclamation point (!) and dollar sign ($) were considered largely uninteresting in the message headers, but were used as constituent characters in the message body. This is probably because most uses of these characters are found in the message body, where they were considered constituent characters. In the message body, amidst some basic punctuation, the classifier found many letters commonly not included in a tokenizer set. As we'll see, some of these letters were so common in uninteresting words that it made more sense to the machine to treat them as a token separator, which provided much better data to work with. Finally, some of the question marks in the above example were actually non-printable characters that were believed to make good token separators.

In the data shown below, we see many familiar text fragments that, when tokenized in this fashion, provided a very useful data point for the classifier. The *s* and *i* values represent the number of times the token has appeared in spam and non-spam, respectively. Plus signs (+) represent the joining of two adjacent tokens together (e.g. biGrams), which is a default feature of the classifier used.

```
[0.990000] ,+click (8s, 0i)
```

Here, we see that the word 'click' alone was not as interesting as when preceded by a comma (,), which would have normally been used as a token separator.

```
[0.940828] igh (105s, 2i)
```

This word fraction probably came from words such as 'high' as in 'high interest mortgage'. It is likely that the letter H was considered a token separator due to finding the word in different cases, and/or using symbols to try and obfuscate the word itself (such as |-|igh). It may have also been part of a larger word that, when broken up, was found as a pattern across many spam.

```
[0.990000] $888 (15s, 0i)
```

That the dollar sign can make a good constituent character in many messages, where it is very useful in identifying dollar amounts used in spam.

```
[0.990000] ional_Inc.+Now (6s, 0i)
```

This phrase fragment would normally have been broken up into several smaller tokens, using underscore and period as token separators. Instead, the classifier determined that these made better constituent characters to give us a very specific identifier for one particular series of obfuscated spam. Interestingly, this pattern could cover many different company names: National Inc, International Inc, Promotional Inc, and others.

During this series, the classifier likely used the letter 't' as a token separator.

```
[0.990000] s0r+C|ubs (12s, 0i)
```

Here, we see the number zero and the pipe (|) character used to obfuscate a series of messages. The classifier determined that, when considered constituent characters, helped identify obfuscations.

```
[0.990000]
      !+ESC(B (50s, 0i)
      ESC$B(-ESC(B (19s, 0i)
      ESC$B!!!!!!!!!!ESC(B (29s, 0i)
```

This message was in a Japanese character set. Most humans would have no idea what they were looking at, but the classifier – without understanding Japanese, found these pattern of characters to be particularly native to spam.

4.SUPPORTING DATA

To measure this technique's effectiveness, tests were run to compare the overall accuracy of a statistical classifier with various parser configurations, including the approach outlined here. These tests were performed using the DSPAM classifier with three different configurations:

1. Whitespace Only Tokenizer: A tokenizer with parsing rules to only break words with whitespace (spaces, tabs, and new lines).
2. Static Default Set: The default tokenizer with parsing rules using a static set of hard-coded word delimiters.
3. Adaptive Tokenizer: A tokenizer using the techniques outlined in this paper to intelligently choose the best delimiters.

The raw values provided represent the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

The tests were conducted by presenting each message for classification using an even distribution for the ratio of spam and non-spam in each corpus. If the resulting classification was incorrect, the message would be processed for training. Regardles of the classification, the filter would observe and evaluate the occurrence of each potential delimiter and add to its memory.

4.1 Comparison of SpamAssassin Public Corpus

The SpamAssassin Public Corpus is a widely accepted public email corpus designed for training and testing spam filters. It consists of over 6,000 email messages - both spam and non-spam. Using whitespace and static tokenizers as a baseline, the adaptive tokenization method increased classification accuracy by as much as 90%. Four separate adaptive tests were run – two using whitespace (spaces, tabs, and new lines) as default separators, and two "pure" configurations that did not make any presuppositions about whitespace. The pure configuration allowed the parser to identify whitespace on its own, which it did quickly. The window size was increased by four in the pure tests to account for the whitespace characters omitted. All adaptive sets outperformed the baseline in terms of efficiency and overall score, however as with such a dramatic increase in efficiency, this resulted in some additional false positives. With such a small measure of false positives to begin with, the significance of this is questionable and should be investigated further.

	TP	TN	FP	FN
Whitespace	1640	4143	7	256
Static Defaults	1654	4145	5	242
Adaptive,W24	1760	4138	12	136
Adaptive,W32	1723	4140	10	173
Pure,W28	1756	4137	13	140
Pure,W36	1666	4142	8	230

Fig. 4.1.1 SA Corpus Raw Test Results

	Precision	Recall	FScore
Whitespace	0.9957	0.8649	0.925
Static Defaults	0.9969	0.8723	0.930
Adaptive,W24	0.9932	0.9282	0.959
Adaptive,W32	0.9942	0.9087	0.949
Pure,W28	0.9926	0.9261	0.958
Pure,W36	0.9952	0.8786	0.933

Fig. 4.1.2 SA Corpus Test Metrics

4.1.1 Counter-Example of the SpamAssassin Corpus

To qualify the increase in overall classification accuracy, the SpamAssassin Corpus tested above was re-run with the formula reversed – that is, the characters found to have the highest probability of being found in *useful* data, instead of uninteresting data, was used instead. As could be expected, this greatly impaired accuracy as shown in Fig. 4.3 below. In this example, the Adaptive,W24 approach was used with an identical configuration as before, with the calculation reversed.

With the drop in efficiency and score, the rate of false positives naturally also dropped, however the change is relatively insignificant compared to the overall results.

	TP	TN	FP	FN
Counter-Example	1578	4146	4	318

Fig. 4.1.3 SA Corpus Counter-Example Raw Test Data

	Precision	Recall	FScore
Counter-Example	0.9974	0.8322	0.9073

Fig. 4.1.4 SA Corpus Counter-Example Metrics

4.2 Comparison of ISP Asian Corpus

The Asian corpus used consists of nearly 35,000 email messages, with a 1:4 mix of legitimate mail vs. spam. An unnamed Chinese Internet Service Provider assembled the corpus primarily for academic purposes such as this. Only the message bodies were used in parsing the Asian corpus, to measure the classifier's efficiency at classifying the Chinese language. The DSPAM classifier itself does not support multibyte characters, and so both Chinese characters and the adaptive parsing characters were processed as single-byte patterns. The total number of messages analyzed was somewhat inconsistent, because each different tokenizer was unable to parse some of the messages from the corpus at all. So while these statistics show a significant improvement in filtering accuracy, the adaptive tokenizer was also able to process more messages that would have otherwise generated an error in the classifier. Overall, the adaptive technique increased classification accuracy significantly.

This test also factored in a comparison to the Kakasi language-processing filter, which was designed to convert between certain Asian character sets to provide a set of ASCII-readable tokens delimited by white space. It's important to note that the Kakasi filter was originally designed for Japanese, and not Chinese, however it did reasonably well in spite of this and scored better than standard heuristic tokenizers. Unfortunately, a Japanese corpus was unavailable for testing.

	TP	TN	FP	FN
Whitespace	21085	9034	24	3699
Static Defaults	22787	8782	66	2109
Kakasi*	18745	8904	108	1438
Adaptive,W24	23952	9224	40	1104
Adaptive,W32	24068	4222	42	988

Fig. 4.2.1 Asian Corpus Raw Test Results

	Precision	Recall	FScore
Whitespace	0.9988	0.8507	0.918
Static Defaults	0.9971	0.9152	0.954
Kakasi*	0.9942	0.9287	0.960
Adaptive,W24	0.9983	0.9559	0.976
Adaptive,W32	0.9982	0.9596	0.978

Fig. 4.2.2 Asian Corpus Test Metrics

* Kakasi was not intended for Chinese text

4.3 Comparison of TREC Corpus

The TREC corpus is a large corpus of email constructed for the NIST Text Retrieval Conference's spam track. The corpus consists of more than 35,000 spam and non-spam email in more than 100 different character encodings. Unlike all other tests of adaptive tokenization, this time a reasoning-based approach yielded mixed results, providing little advantage, but did manage to keep up with the other tokenizers. The TREC corpus as a whole contains an overabundance of mixed mail from a plethora of different languages, encodings, and of varying quality. This proves useful as a "worst case scenario" for filter tests.

	TP	TN	FP	FN
Whitespace	24426	12878	32	486
Static Defaults	24510	12885	25	402
Adaptive,W24	24474	12879	26	438
Adaptive,W32	24455	12878	27	457

Fig. 4.3.1 TREC Corpus Raw Test Results

	Precision	Recall	FScore
Whitespace	0.9986	0.9804	0.989
Static Defaults	0.9989	0.9838	0.991
Adaptive,W24	0.9989	0.9824	0.990
Adaptive,W32	0.9988	0.9816	0.990

Fig. 4.3.2 TREC Corpus Test Metrics

5.FUTURE WORK

Other areas to research include extending the simple ASCII delimiter set to include bigram and trigram delimiters and incorporate delimiter placement (before or after the delimiter). This would allow the parser to intelligently identify inflectional endings in many languages and perform statistical word stemming, if it believed such parsing would improve overall output. The Greek and Chinese language are an excellent example of this, as the prefixes, cases, and inflectional endings applied to words can cause them to change dramatically[10]. Allowing the parser to determine whether the prefix/suffix better serves as a separator or a constituent component would allow the parser to root certain words it found to be more interesting in such a context, and apply the prefixes or suffixes in more general words that took on a more specialized disposition with the component's presence.

6.CONCLUSION

By adaptively reconfiguring the parser, a lexical machine can quickly learn and begin to parse many different forms of data on its own, without prior knowledge of the language by the filter author. This makes a reasoning-based approach to language parsing a very powerful way to generate data, and further optimize existing data. This technique's unsupervised form of training allows for quick and accurate parsing without additional work by the end-user, or previous knowledge of the language being parsed.

Further improvements might be made by using a separate tokenizer memory for each known character set introduced. Adjusting thresholds for determining usefulness of data can also play a role in the overall benefits of this approach.

While further tuning may be necessary for highly diverse corpora, this approach in general provides a

significant increase in overall classification accuracy for most texts. This technique can be applied to many facets of language classification including character-set identification, pattern recognition, document fingerprinting, and fuzzy data mining operations.

6.REFERENCES

[1] Zdziarski J, Ending Spam, Bayesian Content Filtering and the Art of Statistical Language Classification. "*Adaptive Tokenization"*, No Starch Press, ISBN 1593270526

[2] Zdziarski, J. Detecting Contextual Anomalies in Lexical Reasoning Machines, January 2006, Preceedings of the MIT Spam Conference 2005

[3] Graham, P. *So Far, So Good*, August 2003, http://www.paulgraham.com/sofar.html

[4] Graham, P. *Better Bayesian Filtering*, January 2003, http://www.paulgraham.com/better.html

[5] Yerazunis, W. *The Spam Filtering Accuracy Plateau and How to Get Past It*, Preceedings of the MIT Spam Conference 2003

[6] Graham, P. *A Plan for Spam*, August 2002, http://www.paulgraham.com/spam.html

[7] Roark B., Johnson M., *Efficient probabilistic top-down and left-corner parsing*, Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics, p.421-428, June 20-26, 1999, College Park, Maryland

[8] Kozerenko E., *Semantic Approach to Language Structures,* Institute for Informatics Problems of the Russian Academy of Sciences, Moscow, Russia

[9] Collins, M. *Three generative, lexicalised models for statistical parsing*. ACL 35/EACL 8, pp. 16-23, 1997.

[10] Wu D., Fung P. *Improving Chinese Tokenization With Linguistic Filters on Statistical Lexical Acquisition*, University of Science and Technology, Clear Water Bay, Hong Kong