TR-IT-0228

The ATR Parser of English Text

Stephen Eubank

July 1, 1997

Abstract

This paper gives a brief description of how to use the ATR parser, what sort of performance to expect, and an outline of how it operates. It includes a detailed description of the algorithm currently used for pruning the decision tree models.

ATR Interpreting Telecommunications Research Laboratories

Contents

1 Using the ATR Parser and Tagger

The best place to find out how to use the ATR Parser and Tagger is on a set of Web pages I have begun to write. These can be reached via my ATR home page at: www.itl.atr.co.jp/ eubank. See especially the link to the "quick start guide" and information on the meanings and use of configuration parameters.

1.1 Executables and Configuration Parameters

Configuration parameters are currently passed through UNIX environment variables. The two which must be set (ModelPath and GrammarPath) tell the software where to find (machine-dependent) models and the grammar itself. Models and executables are maintained for Sun and DEC Alpha workstations. Pathnames mentioned below assume you are using an Alpha. In general, users should be sure to get the most up-to-date models available, but for experimental purposes in the near future, use "setenv ModelPath /home/as18/eubank/travel/model" and "setenv GrammarPath /home/as18/eubank/travel/grammar"). You may also need to add a path to your "LD_LIBRARY_PATH" environment variable on a DEC Alpha.

Both the tagger and parser (eubank/bin/alpha/TaggerTool and eubank/bin/alpha/ParserTool) read English ASCII text from standard input and write analyses to standard output. They also produce voluminous diagnostic information on standard error.

The tools assume the input consists of a single sentence (utterance) per line. The text will be tokenized by a rudimentary, rule-based tokenizer. There is a description of the rules it follows available from the Web page. Output is in the form of an N best list (N controlled by configuration parameter) with the likelihood of each analysis. Because of the way the models are constructed and used, the likelihoods are not normalized over all possible analyses.

1.2 Resource requirements

The executable itself takes 2 - 4 Megabytes of disk space. The models, grammar, and feature extraction module for both Sun and Alpha take an additional 30 Megabytes. Unfortunately, there are some hard-coded absolute path-names in the compiled version of the grammar, so the grammar and models should not be moved around. Parsing on an ALpha will require anywhere from 100 - 400 Megabytes of RAM, depending on sentence length and the settings of search parameters. Initialization requires from one to several minutes. After initialization it should be possible to tag at a rate of roughly 10 words per second. Parsing time is much more variable, though the results presented below required roughly 30 seconds to 4 minutes per parsed sentence.

2 Performance

Table 1 summarizes the performance of the models mentioned above as a function of sentence length. In general, the performance of the parser decreases for longer sentences because of two effects:

- We score performance on a per-sentence basis rather than a per-word basis, but the number of model evaluations required to produce a parse is roughly quadratic in the number of words.

- The number of parses which are legal according to the grammar increases exponentially with the number of words in a sentence, so finding the best predicted parse becomes exponentially harder.

The table shows *cumulative* performance as a function of sentence length. This reflects the relative frequency of various sentences in our test database, and thus provides a properly weighted estimate of the chance of successfully parsing a sentence picked at random from the test set. For each set of test conditions, the results for both the single sentence predicted to be the best parse and for any of the top ten parses are shown.

The measure of correctness is a very strong one. Each sentence in the test set has been assigned a parse by a human treebanker. There may be other correct parses, but they are not noted in the test set. For a parse to be judged correct, we require an exact match of both the parse tree structure and the labels at each node of the tree with the parse found in the treebank.

The fifth and sixth columns in the table indicates the performance of models trained solely on ATR travel data (a different set from the test set, of course). These models were given correctly tagged text and asked to predict the parse. This does not reflect performance on untagged text, but it allows us to distinguish the performance of models in the parser from those in the tagger.

The results in the third and fourth columns are also based on correctly tagged text, but used models trained on general English text instead of

Table 1: Performance of various models on an ATR travel data test set.

		from correct tags				from predicted tags travel model			
		travel models		general English		template match		exact match	
len	#sents	top 1	top 10	top1	top 10	top 1	top 10	top 1	top 10
1	117	98.3	99.1	98.3	98.3	84.6	85.5	84.6	84.6
2	167	97.0	99.4	94.6	98.8	85.3	94.5	79.1	87.7
3	114	86.8	97.4	70.2	85.1	71.9	82.5	60.5	72.8
4	139	86.3	95.7	67.9	74.5	72.6	80.0	58.5	69.6
5	149	87.9	92.6	61.7	71.8	68.9	75.0	52.0	61.5
6	109	90.8	93.6	66.7	75.0	63.0	64.8	38.0	49.1
7	119	75.6	84.0	60.5	72.3	55.6	62.4	30.8	42.7
8	113	72.6	80.5	47.8	59.3	53.3	56.1	24.3	37.4
9	100	72.0	83.0	47.0	63.0	53.6	60.8	25.8	41.2
10	98	65.3	79.6	45.3	54.7	47.9	51.1	18.1	31.9
11	72	58.3	73.6			32.8	40.3	11.9	19.4
12	67	50.7	64.2			37.1	43.5	19.4	25.8
13	73	48.6	65.3			15.2	19.7	4.5	9.1
14	52	44.2	59.6			19.6	25.5	5.9	7.8
15	48	33.3	58.3			20.9	34.9	7.0	11.6
16	43	37.2	72.1			24.4	34.1	4.9	4.9
17	31	27.6	41.4						
18	29	24.1	37.9						
19	26	23.1	38.5						
20	21	35.0	60.0						
21	20	11.8	11.8						
22	16	31.2	43.8						
23	10	20.0	50.0						
24	7	14.3	28.6						
25	8	12.5	25.0						
26	6	0.0	0.0						
27	12	20	30						
28	7	28.6	28.6						
29	4	0.0	0.0						
30	2	0.0	0.0						

ATR travel data. I expected these models to perform well, because they were trained on more than twice as much data as the ATR travel models. Indeed, on a general English test set, they do perform comparably to the upper curve. The degradation in performance on ATR travel data indicates how task-dependent the parser is. It also emphasizes the need for adaptive models.

The remaining columns in the table give the performance of models trained solely on ATR travel data and tested on raw (i.e. untagged) text. The seventh and eighth columns give results for parsing structure only, ignoring the predicted tags. It is possible to find the correct parse tree topology and label each node correctly even if the tags are not correct, since information in the parse tree labels is primarily syntactic. Thus, if the predicted tags have the correct syntax, even if the semantic portion is incorrect, the parse tree itself may be correct. Finally, the last two columns show the performance of the models predicting from raw text, where a parse is considered correct only if:

- every tag matches the tags in the treebank exactly
- the parse tree topology exactly matches the treebank
- every node of the parse tree receives the same label it did in the treebank.

It is not yet clear exactly how performance of the tagger and parser affect applications which may rely on them, or whether certain types of errors are more serious than others.

3 What the parser does

Raw text (one sentence per line) is tokenized and fed to the parser. Using the ATR English grammar, the parser builds a "chart", which lists all legal parses in a compact form. The parser uses the chart as a guide for creating "parse states", which are incomplete parse trees. Because the parser works from the bottom up, left to right, the intermediate parse states are not necessarily subtrees, but more like collections of subtrees, together with a cursor indicating which subtree's root is currently under consideration.

Parse states are produced by a taking a series of "parse steps" from an initial parse state consisting of the words in a sentence. A parse state's derivation is unique, so its probability can be written down simply as the product of each parse step required in the derivation.

The parser is responsible for providing parse states to a feature extraction module. This module defines methods for navigating around a parse state and asking for information about nodes in the parse state's subtrees. For example, a question might ask about the "pos" feature value of the most important word (head word) in the second constituent to the left of the current position. Answers to all questions are transformed into bitstrings based on mappings defined in ASCII files. These mappings are built either by machine or human clustering.

Parse state dependent bit strings produced by the feature extraction module are passed to the models for scoring. A model estimates the probability of each possible step out of the parse state.

Probabilities are fed into the search, which decides which steps to consider and tells the parser which new parse states to construct.

One surprising development has been that a depth first, or "greedy", search is close to optimal. Apparently, this is because, unlike in the case of speech recognition, the entire context (for example, the entire sentence) is available even when the first decision is made. Thus there is no need to develop a hypothesis about the correct parse which must be changed as more information or context becomoes available.

Of course, it is still possible that the greedy parse is not the best, so I have implemented the following search procedure:

- Find a greedy parse
- Use the greedy parse to seed a stack based decoder - that is, insert all the incomplete parse states from the greedy parse into the appropriate stacks.
- Use the probability of the greedy parse as an absolute threshold for the stack decoder.
- Let the stack decoder suggest some alternative states. (Alternatives can arise from any of the stacks.)
- Find the greedy parse for each alternative and repeat the stack decoding until all parses within the decoder's thresholds have been found.

4 Appendix: Pruning Algorithm for Classification Trees

The problem of determining where to prune is recast as choosing a "significance level" for the proposed refinement. The threshold chosen will, in general, depend on three things:

- the number of input features
- the number of training data points in the partition,
- the sample conditional distribution of classes for those points.

Because the last two of these may vary widely from one partition to another in the same classifier, the criterion must be applied locally in the classifier.

Wray Buntine has done similar work on decision trees before but I do not yet have a citation for his papers. Also see J. Mingers, Machine Learning 4 (1989) p 227 - 243, and references therein.

Some people advocate including a random input to detect overtraining. Others apply a global threshold to the cost reduction.[1] The present approach differs from both these because it frames the question as a statistical test against a null hypothesis with an exact null distribution. That is, it does not rely on a single realization of a random variable to estimate the null distribution; nor does it use an arbitrary global threshold.

4.1 Smoothing

An alternative to pruning decision trees was proposed by ?? In this approach, the estimated distribution for a partition is *smoothed* by forming a linear combination with the smoothed distribution of its parent partition. Ideally, the amount of smoothing is controlled by a separate parameter for each partition. However, this would mean using the cross-validation set to estimate a large number of parameters - roughly the same number as were estimated using the entire training set. Hence the partitions are bucketed, usually according to number of data points, with the result that the same smoothing parameter is used for several different partitions. Optimal values for the parameters are determined using an E-M algorithm on cross-validation data.

[1]Friedman, et al. argue that this approach does not, in fact, improve CART trees.

4.2 Hypothesis Testing

The alternative advocated here is to prune the tree from the bottom up by testing at each node whether any partition is significantly different from the parent. The test becomes a standard test for significance once one chooses a discriminating statistic for measuring differences between two distributions. There are several obvious possibilities, including:

- chi-squared on the sample conditional distributions;
- classification error;
- the impurity function used to split the nodes - entropy, Gini, etc.

At each of the tree's leaf nodes, the algorithm compares the value of the statistic against the distribution of values under the null hypothesis. Any node whose children are all deemed insignificant becomes a leaf node. The process continues up the tree until all leaf nodes have been considered.

The null distribution, and thus the threshold value of the statistic for a given confidence level, will differ from node to node depending on the number of training events which reach that node, the number of input features, and the sample distribution of training event classes. In this sense, the test for significance is "local" in the tree. In particular, it is not reasonable to specify a globally-applicable threshold value for reduction in classification error or impurity.

The experiments reported here use classification error as the discriminating statistic because it is feasible to build the exact null distribution for this statistic quickly. Ideally one could choose among the various statistics on the basis of their power and size in the particular problem of interest. One of the interesting features of the results reported here is that there is no indication that the statistic need be matched to the splitting criterion. For example, even if entropy reduction is used to select a split, classification error is useful for pruning the tree.

4.3 The Null Hypothesis

Once the statistic for measuring the difference between distributions has been chosen, there is only one decision to make about the null hypothesis: what constitutes a "random" partition?

We choose a random partition to be one which assigns each data point to a set independently with uniform probability over sets. In other words, the null hypothesis is that the input features are independent, identically uniformly distributed, and independent of the true class. Of course, the input features may have dramatically different properties, including statistical dependence among features and non-uniform distributions over classes. One would expect a better-specified null hypothesis to improve the pruning.[2] Nevertheless, even this simple null hypothesis will suffice here.

4.4 The Null Distribution

We require the distribution of misclassification rates given by random assignments of data to partitions. This seems a daunting task, given the astronomical number of ways to partition even a small data set. Indeed it is impossible to calculate the classification error separately for each partition. However, by organizing the enumeration of possible partitions as in the example below, the exact distribution can be determined quickly and efficiently. The trick is to count how many ways each possible misclassification rate can be obtained. An additional benefit of this organization is that the counting is required only for the tail of the distribution – those classification error rates less than or equal to the observed rate at any node. Because the total number of ways to partition the data is easily computed, the counts can be normalized to probabilities.

4.4.1 Assumptions: Binary Refinement, Binary Features, Uniform Null

The following case is sufficiently simple to allow a detailed description of the enumeration process:

- discrete-valued input features
- binary refinements

With no loss of generality for the case of discrete-valued inputs, I further assume the input features are binary valued. I also assume the refinement consists of splitting the original set into two subsets, as is the case for a binary decision tree. The predicted class for each element in a partition is the class occurring most commonly in that partition.[3]

[2]It would not be difficult to incorporate bias in partition assignment into the null hypothesis. However, as is often the case, it is much harder to account for dependence among the input features.

[3]In the case of ties, either class can be assigned.

4.4.2 Notation

Let N be the total number of events to be partitioned into two sets A and B. Q is number of events correctly classified by the partition. Of these, q_A are in set A and q_B in set B. The number of partitions which classify exactly Q events correctly is denoted $\#Q$. A partition assigns each of the n_i events of every class i to either set A or set B. In all, $n_{i,A}$ events of class i are assigned to set A and $n_{i,B} = n_i - n_{i,A}$ to set B. Obviously, $q_A = n_{i,A}$ and $q_B = n_{j,B}$ for at least one i and at least one j. It may be the case that $i = j$.

4.4.3 Organize by Misclassification Rate

As discussed above, the first step in evaluating the null distribution is to group together all partitions which yield the same overall number Q of correct classifications. Q must satisfy

$$n_1 \leq Q \leq n_1 + n_2. \tag{1}$$

This constraint is similar in spirit to the observation that the entropy of the partition cannot be larger than the entropy of the original set. The original set, by definition, correctly classifies n_1 events. The partition must divide all n_1 events between the two subsets A and B. Either events of class 1 are the majority in subset A, in which case exactly $n_{1,A}$ are correctly classified, or there is some other class c with more elements in the subset, in which case $n_{c,A} > n_{1,A}$ events are correctly classified. Of course, the same argument applies to subset B, so

$$Q = q_A + q_B \geq n_{1,A} + n_{1,B} = n_1. \tag{2}$$

The maximum Q results from correctly classifying the two largest subsets.

4.4.4 Total number of partitions

There are 2^N possible ways to assign events to two subsets, but since interchanging the identity of A and B makes no difference to the partition, we must divide by two. Furthermore, the partition which sends all events to one subset is not allowed. Thus the total number of ways to partition the events is $2^{N-1} - 1$.

4.4.5 Counting Ways to Classify n_1 Events Correctly

A particularly easy case to enumerate is the worst case, when $Q = n_1$. We must enumerate all possible ways of assigning the events of class 1 to the two subsets, ensuring that they constitute a plurality in each. Consider a partition which assigns $n_{1,A}$ events of class 1 to subset A. There are $\begin{pmatrix} Q \\ n_{1,A} \end{pmatrix}$ ways to arrange this part of the partition. For each of these ways, how many ways are there to assign the $n_2 \leq n_1$ events of class 2, ensuring that no more than $n_{1,A}$ end up in A and $n_{1,B}$ in B? We see that $n_{2,A}$ must satisfy:[4]

$$n_{2,A} \leq n_{1,A} \text{and } n_{2,B} = n_2 - n_{2,A} \leq n_{1,B} \tag{3}$$

$$\Longleftrightarrow n_{1,A} \geq n_{2,A} \geq n_2 - n_{1,B}. \tag{4}$$

Given $n_{2,A}$, there are $\begin{pmatrix} n_2 \\ n_{2,A} \end{pmatrix}$ distinct ways to choose exactly which $n_2 A$ events are placed in set A.[5] This counting must be repeated for each of the remaining $k-2$ classes. That is, for each different value of $n_{1,A}$, there are exactly

$$\prod_{m=2}^{k} \sum_{n_{m,A}=n_m-n_{1,B}}^{n_{1,A}} \begin{pmatrix} n_m \\ n_{m,A} \end{pmatrix} \tag{5}$$

ways of assigning events from the remaining classes to the two sets. Finally, we must allow $n_{1,A}$ to take on all possible values between 1 and $n_1 - 1$. Because of the symmetry between the sets A and B, it is more efficient to allow $n_{1,A}$ to take on only half of its allowed values, throwing in a correction factor of 1/2 for the case in which n_1 is even and $n_{1,A} = n_1/2 = n_{1,B}$. All together, then, we obtain

$$\#Q = \sum_{n_{1,A}=\lceil n_1/2 \rceil}^{Q-1} \begin{pmatrix} Q \\ n_{1,A} \end{pmatrix} 2^{-\delta(n_{1,A},n_{1,B})} \prod_{m=2}^{k} \sum_{n_{mA}=n_m-n_{1,A}}^{p-n_{1,A}} \begin{pmatrix} n_m \\ q \end{pmatrix} \tag{6}$$

Summarizing, there are three factors in this expression:

- a factor enumerating the ways the correct class can be assigned to set A.

[4] Allowing equality in this summation, eliminates the need to consider the possibility that classes other than 1 account for the correctly classified events.

[5] Note that the sets A and B are distinct by virtue of the number of elements of class 1 they contain.

- a factor enumerating the ways of assigning all the remaining events.
- a factor correcting for overcounting.

In the more general case below, each of these factors reappears, together with another factor enumerating the ways the correct class can be assigned to set B.

4.4.6 Counting Ways to Classify Q Events Correctly

Remember that the classes are listed in descending order of their population among the sample events. Define $\alpha(m)$ to be the last class whose population is greater than or equal to m:

$$\alpha(m) \equiv max\{i|n_i > m\} \tag{7}$$

Now some number $q_A < Q$ of the correctly classified events arise from assigning events from the class i to the subset A. That is, for some i, $q_A = n_{iA}$. Without loss of generality, assume $q_A \geq \lceil p/2 \rceil$, as above. There are

$$\sum_{i=1}^{\alpha(q_A)} \binom{n_i}{n_{iA}} \tag{8}$$

ways to do this. For each of these ways, the remaining correctly classified events will appear in subset B and must come either from some different class n_j or (if $Q = n_i$) from the same class, a partition which has already been counted. There are:

$$\sum_{j=1\, j\neq i}^{j=\alpha(q_B)} \binom{n_j}{q_B} \tag{9}$$

ways to accomplish this.

4.4.7 Counting Ways to Apportion Remaining Events

So far, we have taken care of the Q correct events. All the remaining events must now be distributed in all possible ways between the two subsets, subject to the constraint that no more than q_A events from any one class go to subset A, and no more than q_B go to subset B. [6]

All the events in the particular subsets i and j in the summations above are already accounted for. For each of the remaining subsets, we must count

[6]We are guaranteed that this is possible because we know $Q \geq n_1 \geq n_i$ for all i.

all possible ways of distributing events subject to the constraints. For class m, this is:

$$\sum_{n_{m,A}=n_m-q_B}^{q_A} \begin{pmatrix} n_m \\ n_{m,A} \end{pmatrix} \tag{10}$$

A few distributions of events have appeared in other factors. If class $n_{m,A} = q_A$, we will count the contribution again when i takes on the value m. Similarly for $n_{m,B}$ and class j. To account for this, multiply each term by 0 when the constraints above apply. The following expression:

$$\{1 - \delta(n_{m,A}, q_A)\Theta(i,m)\} \tag{11}$$
$$\{1 - \delta(n_{m,B}, q_B)\Theta(j,m)\}, \tag{12}$$

where

$$\Theta(a,b) \equiv 1 \; if \;\; a > b, \;\; 0 \; else, \tag{13}$$

will do the trick.

4.4.8 Probability Distribution for Misclassification Rates

Putting all this together yields the probability distribution for misclassification rates under the null. There is one additional combinatorial factor that must be accounted for:

- If $Q = n_i$, there is a symmetry between subsets A and B which results in overcounting by a factor of 2.

Overall then, the probability that Q events will be correctly classified by a random partition is:

$$P_0(Q) = \quad (2^{n-1} - 1)^{-1} \textstyle\sum_{q_A=\lceil Q/2 \rceil}^{Q-1} \sum_{i=1}^{\alpha(q_A)} 2^{-\delta(Q,n_i)} \begin{pmatrix} n_i \\ q_A \end{pmatrix} \tag{14}$$

$$\textstyle\sum_{j=1,j\neq i}^{j=\alpha(q_B)} \begin{pmatrix} n_j \\ q_B \end{pmatrix} \prod_{m=1,m\neq i,j}^{k} \sum_{n_{1,A}=q_A}^{n_m-q_B} \begin{pmatrix} n_m \\ n_{m,A} \end{pmatrix} \tag{15}$$

$$\{1 - \delta(n_{m,A}, q_A)\Theta(i,m)\}\{1 - \delta(n_{m,B}, q_B)\Theta(j,m)\}, \tag{16}$$

Note that in some cases it is possible to collapse the final summation. For example, if $n_{m,A}$ takes on all possible values $1 \ldots n_m$, it collapses to 2^{n_m}.

4.5 Experiments

4.5.1 Description of the Problem

In the course of building a syntactic and semantic tagger for English text, we create several large decision trees: one, the "pos" model, is used to classify a word according to its part-of-speech; the remaining 15 are used to assign a semantic class to the word, given the predicted part-of-speech classification. There are 33 possible part-of-speech classes. The number of semantic classes varies depending on the part of speech between 2 and 1000. The input vector consists of features of the word and its context judged by a grammarian likely *a prior* to be useful in classifying the word. The specific features vary somewhat from model to model, but in all cases consist of roughly 20,000 binary input features. We use roughly a million events for training the pos model. The other models are trained on subsets containing anywhere from tens of events to tens of thousands of events.

We report here both the classification performance of the individual trees and the overall performance of the (hierarchical) combination of trees. Strictly speaking, the decision trees are used as *class probability estimators* rather than simple classifiers, and it is this use that is reflected in the overall performance results, but not in the results for the individual trees.

4.5.2 Description of the Experiment

We varied several parameters controlling the growing and pruning process:

- Splitting criterion: Gini or entropy
- EM smoothing, CART-style pruning, or significance pruning.
- Splitting events into (a single pair of) training and cross-validation (which is required for CART-style pruning and EM smoothing) or not.

As measures of performance, we cite both cross-entropy and classification error rates on a single out-of-sample set.

The data was divided into an in-sample (90%) and out-of-sample (10%) set. The in-sample set was further split into training and cross-validation sets, where needed. Only one division was used – multiple cross-validation runs were not performed.

4.5.3 Results

In the following,

"orig"	→	model trained on "train" data set, no F/B smoothing or pruning
"big"	→	model trained on "train" + "smooth" data set, no F/B smoothing or pruning
"FB"	→	forward/backward smoothing (using "smooth" set)
"CART"	→	cost-complexity pruning (using "smooth" set)
"prune"	→	significance pruning

Table 2: Tree size - total # nodes in all trees (rounded to nearest 100)

criterion	orig	CART	prune	big	big prune
entropy:	39700	4800	9400	46200	10800
mixed:	40000	6400	10800	46500	12300
Gini:	42250	6300	10600	49100	11700

Table 3: Number of events in train and test sets

	train	big	test
pos	298481	355683	17224
n	90667	107222	5654
v	43595	52100	2202
punct	34672	41025	2155
p	31653	38135	1755
determin	35382	42432	1706
j	19438	23497	1250
numeric	10955	12570	758
ccxx	9002	10744	584
r	10214	12440	563
csxx	5382	6554	243
le	197	233	17

Table 4: Performance breakdown by model - misclassification rate in percent
Models ordered by number of events in test set
A. Entropy criterion

	orig	CART	prune	big	big, prune
pos	8.5	7.9	8.0	8.0	7.7
n	62.0	59.2	59.8	61.3	58.9
v	52.2	52.5	52.2	52.0	51.2
punct	4.8	7.2	4.8	4.9	4.1
p	1.3	1.5	1.3	1.1	1.1
determin	0.2	0.2	0.2	0.3	0.2
j	60.8	57.5	55.0	57.8	53.1
numeric	21.4	24.0	21.0	20.3	19.9
ccxx	0.3	0.5	0.3	0.3	0.3
r	32.7	29.7	28.6	29.5	28.8
csxx	3.7	4.5	3.7	3.7	3.7
le	11.8	11.8	11.8	11.8	11.8

B. Mixed Criterion

model	orig	CART	big	big, pruned
pos	8.4	7.8	7.9	7.6
n	60.2	58.0	59.0	57.6
v	52.2	51.5	50.9	50.4
punct	4.7	7.3	5.1	4.1
p	1.3	1.5	1.1	1.1
determin	0.2	0.4	0.2	0.2
j	58.6	55.3	54.8	54.0
numeric	19.0	23.4	20.3	20.1
ccxx	0.3	0.5	0.3	0.3
r	31.1	28.2	30.6	28.2
csxx	4.1	4.5	3.3	2.5
le	17.6	17.6	17.6	17.6

C. Gini criterion

model	orig	CART	prune	big	big, pruned
pos	8.1	7.5	7.6	7.8	7.5
n	54.8	53.4	53.4	54.0	51.9
v	50.3	47.9	48.3	49.2	47.8
punct	6.0	5.8	5.6	5.4	5.0
p	1.2	1.1	1.2	1.2	1.3
determin	0.3	0.3	0.2	0.3	0.3
j	55.5	50.2	52.0	54.2	51.6
numeric	17.5	26.4	18.5	17.8	18.1
ccxx	1.0	1.4	1.0	0.9	0.9
r	32.0	30.2	30.0	30.6	28.4
csxx	5.3	4.5	4.5	4.5	4.1
le	17.6	17.6	17.6	17.6	17.6