2007-10-07

ADtrees for Sequential Data and N-gram Counting

Robert Van Dam
rvandam00@gmail.com

Dan A. Ventura
ventura@cs.byu.edu

Original Publication Citation
Rob Van Dam and Dan Ventura, "ADtrees for Sequential Data and N-gram Counting", Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, pp. 492-497, 27.

ADtrees for Sequential Data and N-gram Counting

Rob Van Dam and Dan Ventura

***Abstract*— We consider the problem of efficiently storing n-gram counts for large n over very large corpora. In such cases, the efficient storage of sufficient statistics can have a dramatic impact on system performance. One popular model for storing such data derived from tabular data sets with many attributes is the ADtree. Here, we adapt the ADtree to benefit from the sequential structure of corpora-type data. We demonstrate the usefulness of our approach on a portion of the well-known Wall Street Journal corpus from the Penn Treebank and show that our approach is exponentially more efficient than the naïve approach to storing n-grams and is also significantly more efficient than a traditional prefix tree.**

I. Introduction

Hidden Markov Models (HMMs), otherwise known as n-gram models, originated as a tool in natural language processing for problems such as language identification, spelling correction, part of speech tagging [1], [2], [3], [4], [5] and have since been appropriated by researchers in many other fields, particularly information retrieval, bioinformatics, and data compression [6], [7], [8]. N-grams have become so ubiquitous due in great part to their flexibility. Not only can models be built which take into account vastly different amounts of context (the n in n-gram) but the specific tokens which are counted (the "gram" in n-gram) can vary from one problem to the next. Most frequently in the problems noted above an n-gram model is built using either words or characters from source text.

Unfortunately, n-gram models can grow very large, particular for higher order models. Although not all possible sequences of words or characters actually occur (consider the 6-gram "zvyqxx" in English), n-gram models have the potential to grow exponentially in the number of unique tokens being counted. For English character HMMs, this means a model that has an upper bound of 26^n. For even the simplest Japanese word models on the other hand, the model grows as approximately 2000^n. English word-based models tend to be even larger since a typical document can contain several thousand unique words (even after eliminating capitalization differences). The Wall Street Journal section of the Penn Treebank [9] contains slightly more than $40,000$ words while the much larger n-gram dataset released by Google contains more than 13 million unique words [10]. Although increases in computer memory allow ever increasing n-gram models to be constructed, models that increase exponentially with bases in the thousands or millions will always outstrip the pace of development in the computer memory industry. Additionally, several studies on smoothing n-gram models have found that the best smoothing techniques utilize some (often linear) combination of smaller n-grams when a larger n-gram has not been seen or not been seen sufficiently often to provide reliable statistics [2], [3]. This means that any method for storing n-grams must store all shorter sequences and their counts as well.

Robert Van Dam and Dan Ventura are with the Department of Computer Science, Brigham Young University, Provo, UT 84602 (email: rvandam00@gmail.com, ventura@cs.byu.edu)

Here we present a method for applying ADtrees (described below) to the problem of efficiently storing n-gram counts. Although ADtrees were designed to store counts derived from tabular datasets with many attributes, each "gram" in an n-gram can be viewed as a distinct attribute. Of course, if a typical training set for an n-gram model is converted to explicitly store each of these attributes, the dataset would become n times larger. Since this does not scale well for large data sets and large n, it is more reasonable to adapt ADtrees to automatically handle n-gram models so that these models can benefit from the space savings of ADtrees and the ADtrees can benefit from the known sequential Markov assumptions.

II. Previous Work

The most naïve method for storing the counts needed for n-gram models simply enumerates all possible combinations of the unique tokens. This method will match the previous exponential expressions within a constant factor because they store a value for all possible combinations, even those which do not occur in the given dataset. Obviously this is overly inefficient although it can be manipulated in a program using a simple multi-dimensional array thereby simplifying the programming logic. The next most naïve storage method simply eliminates those n-grams which were not found in the dataset. This greatly reduces the amount of storage required but is still approximately exponential (the growth actually follows a logistic curve but the inflection point often occurs at too large an n to be useful).

For character models, a prefix tree, also called a trie, can be used to store n-gram counts (it is designed to store counts only for the largest n needed but could be modified easily to store counts of shorter sequences as well) [3]. Prefix trees explicitly store all observed n-grams and so in the worst case, they grow exponentially with n. Common structures for word models follow the same basic concept of prefix trees and therefore provide no additional space savings.

Although n-gram models are strongly tied to sequential datasets, an efficient data structure has been developed for tabular datasets that could potentially provide significant space savings for storing counts of n-grams. The All-Dimensions tree (ADtree) was originally introduced by Moore and Lee [11] as a generalization of the kd-tree and

1-4244-0991-8/07/$25.00/©2007 IEEE

provides substantially improved space requirements for storing all conjunctive counting queries over symbolic datasets. Further refinements to the structure ([12], [13]) have made ADtrees widely applicable to a large number of problems that require efficient access to counts of combinations of features.

A. Description of the ADtree

ADtrees are similar to kd-trees and decision trees because they recursively subdivide the dataset. However, ADtrees split the tree along every dimension and every value so that the result is a tree which implicitly or explicitly stores a count for every event or combination of feature/value pairs. The full ADtree contains two types of nodes which alternate along every path in the tree. ADnodes store the count of one conjunctive query. The children of ADnodes are known as Vary nodes. Vary nodes do not store counts but instead group ADnodes according to a single attribute. The Vary node child of an ADnode for attribute a_i has one child for each value v_j. These grandchildren ADnodes specialize the grandparent's query Q by storing the counts of $Q \wedge a_i = v_j$.

For example, Fig. 1 shows several levels of part of an example ADtree. The top ADnode stores the count of how many times a particular attribute value occurs; for example the word "rib" occurs 97 times (in conjunction with whatever occurs above it in the tree). The Vary nodes that are its children correspond to attributes for the ADnode; for example these might include such things as "word to my left", "word to my right", "my part-of-speech", etc. The Vary nodes' children are again ADnodes, one for each possible attribute value. So, the Vary node labeled "Left" would have an ADnode child for each word that occurs to the left of the word "rib", each storing the number of times that it occurs just before "rib". For example, there might be a node for "tasty" with a count of 32 because the phrase "tasty rib" occurs 32 times, and so forth.

ADtrees make use of several specific space saving techniques that facilitate their improved space savings while minimizing the time penalties when performing queries to retrieve counts from the tree. First, any node which would contain a count of zero is not included (and so any feature combination whose path does not exist is assumed to occur zero times).

Second, when the count along any path is refined below some predetermined threshold, the tree is no longer subdivided and instead the indices into the dataset corresponding to the remaining query matches are stored as a list. These lists of indices are referred to as *leaf-lists*, and they represent a time/space trade-off — the additional query time to follow the indices in the list and count the data is traded to save maintenance overhead and an often significant number of leaf nodes with small counts. For example, in Fig. 1, if our leaf-list threshold was 30, then the ADnode labeled "saucy" would not have children but would instead contain a list of indices into the original data to all locations that contained the pattern "... saucy rib ..."; the same would be true for the "to" node, with pointers into the data for occurrences of the pattern "... rib to ...". The additional query time incurred to process a leaf-list is rather insignificant — a conservative estimate is on the order of 10ms per query (for a threshold of 100) and 100ms per query (for a threshold of 1000). On the other hand, as we shall demonstrate in Sec. IV, the space savings are nontrivial.

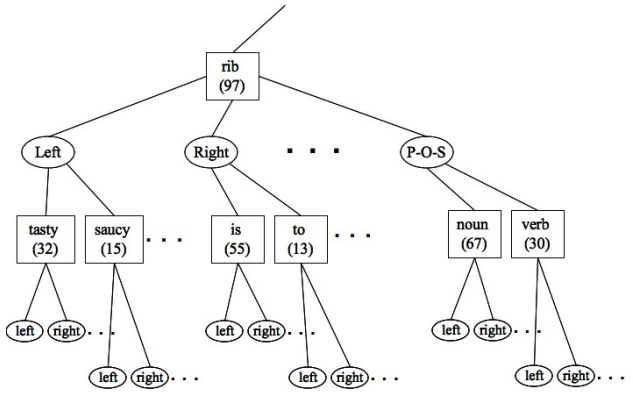

Fig. 1. Several levels of part of an example ADtree. ADnodes store counts of conjunctive queries. Vary nodes represent a particular attribute, grouping ADnodes accordingly. Three measures are taken to save space: ADnodes that correspond to counts of zero are not included in the tree; near the bottom, lists of indices into the original data (leaf-lists) are maintained to avoid creating many leaves with small counts; and for each Vary node, the child corresponding to the most common value (MCV) count is not stored explicitly.

Third, the most common value (MCV) in each dimension is not included because it can be recovered from the difference between the more general count stored higher in the tree and the sum of the other values of the current feature/dimension. Since the ADnode grandchildren of an ADnode for query Q represent all non-zero specializations of Q, the count of Q is equal to the sum of their counts. Therefore, the count of one of these ADnodes can be recovered by subtracting the sum of its siblings from the count of the grandparent using Eq. 1

$$count(Q \wedge a_i = v_k) = count(Q) - \sum_{j \neq k} count(Q \wedge a_i = v_j) \quad (1)$$

If k is chosen such that v_k is the Most Common Value (MCV) or the largest among its siblings, then its removal provides the largest expected space-savings without sacrificing the ability to recover any counts. For a tree built for M binary attributes, the worst case size of the tree is reduced from 3^M to 2^M upon removal of the MCV ADnodes. This technique assumes that the trade-off of increasing average query time is reasonable in comparison to the expected space savings.

A dynamic version of ADtrees demonstrated in [12] follows the same basic structure but constructs the tree in a lazy, "as-needed" fashion. Since it is built lazily, at any given point only a portion of the tree will have been expanded. The dynamic tree also contains some additional support nodes used to temporarily cache information needed for later

expansion. Fully expanded portions of the tree no longer require these extra support nodes.

The experiments reported in Sec. IV use a static version of the tree, but the modifications presented in the next section could just as easily be applied to a dynamic tree (such as in situations involving extremely large datasets where the static tree won't fit in memory). This could be particularly beneficial if the tree could be stored to disk between uses, allowing subsequent uses to skip initialization time (as with a static tree) but without any restrictions on how large of a n can be queried.

III. Applying ADtrees to n-grams

Because ADtrees are fairly general in their applicability, it is not unreasonable to assume that at least some of their space saving techniques can be applied to storing n-gram counts in a more efficient manner. The primary obstacle is determining how to adapt a typical n-gram data set to work with an ADtree. ADtrees expect tabular data with multiple dimensions and values in each dimension while the data sets used for n-gram problems are generally simply text which are treated under the Markov Model as a sequence of grams (be it words or characters). The solution that will be presented here will conceptually approach the sequential data as being potentially M-dimensional where M is the number of words or characters.

Let the M dimensions be named word0, word1, word2, etc (or char0, char1, etc). Then the first row in this new tabular version of the data has each word (character) of the dataset in its respective column. The second row does the same except with all the words (characters) shifted to the left one column. Similarly for the third column and so on. Technically row k will have its last k columns empty but this will not present a problem. Obviously this expansion of the original dataset into tabular form would be extremely inefficient, taking a dataset of M words and creating one of $M(M+1)/2$ words. However, there is no need to actually construct this dataset. Rather it serves as a conceptual motivation for applying the ADtree to the problem of storing n-gram counts.

Assuming the existence of such a dataset, constructing the corresponding ADtree is straightforward. However, the same construction can be accomplished virtually by using the numbered subscripts on the features as a type of relative index into the data. Instead of iterating through the rows in a table, the tree constructor can iterate through words and at each word iterate through its successors to determine the n-gram rooted at that word. In this manner, all prefixes of the n-gram or lower order n-gram sequences can be obtained for free in the process of obtaining the desired n-gram. Additionally, queries into the tree are simplified to be a list of the desired words or characters instead of list of feature value pairs because the pseudo-"features" can be inferred from the order in the list (that is, the first element of the list is word0, the second is word1, etc).

In addition to the benefits derived from the ADtrees, this new n-gram storage method can benefit from the assumptions implicit in the Markov model. An n-gram model is concerned only with consecutive sequences of words (characters) and all other combinations are inconsequential. In terms of the new pseudo-tabular dataset, there is no need to store counts for the query (word5:the, word17:up, word238:useless) since that is not an n-gram. ADtrees however are designed to split along every possible dimension and that includes many paths that are not needed for this application.

Unfortunately, it is not straightforward to immediately remove all these extraneous paths. As was noted, the most common value for each dimension at each level in the tree is removed since that count can be recovered indirectly from other counts stored in the tree. The problem is that counts for MCVs along the paths that are valid n-grams require counts from paths that are not valid n-grams. A concrete example will illustrate this problem. The most common trigram in the Wall Street Journal (WSJ) corpus of the Penn Treebank is the phrase "the company 's" where "'s" is treated as a separate term. Given a standard ADtree built on the WSJ corpus, the word "the" would be the MCV and therefore not stored. The needed count can be recovered from the formula

$$f(\text{the company 's}) = f(\text{* company 's}) - \left(\sum_{x \neq \text{"the"}} f(\text{x company 's}) \right) \quad (2)$$

where "*" represents any word. But now consider the case where "'s" is the MCV given any word followed by the word "company" then Eq. 2 reduces to

$$f(\text{the company 's}) = f(\text{* * 's}) - \left(\sum_{x \neq \text{"company"}} f(\text{* x 's}) \right) - \left(\sum_{x \neq \text{"the"}} f(\text{x company 's}) \right) \quad (3)$$

which is easily handled by the ADtree. However, the first query and all the queries inside the first summation are not strictly n-gram queries. Instead they correspond to queries using the features (word1, word2) and (word2) which do not fit the definition used to construct the pseudo-tabular dataset. Additionally, with the exception of some border cases, these queries are redundant to the bigram and unigram queries by simply renormalizing the subscripts to start at 0.

Perhaps somewhat counter-intuitively, in the case of storing n-gram counts that we consider here, the space reduction technique of implicitly storing MCV counts is actually more detrimental than if the node had been stored explicitly and the assumptions of the Markov model more aggressively exploited. Note in particular the first summation in Eq. 3 sums over all possible values for x with no restraint on the first word of the trigram, thereby permitting most if not all of the $40000+$ words. The result is that the exclusion of one

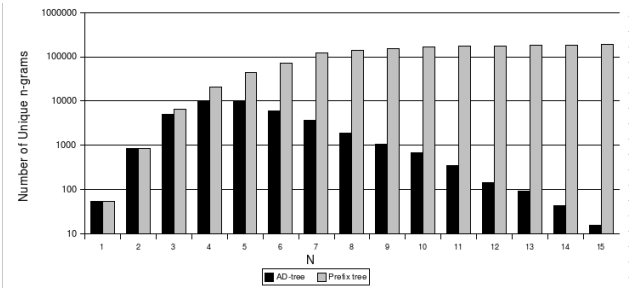

Fig. 2. Distribution of the number of unique n-grams stored for increasing values of n under the character n-gram model for the ADtree and the prefix tree. The ADtree stores fewer long n-grams because they often occur infrequently enough to be handled by leaf-lists.

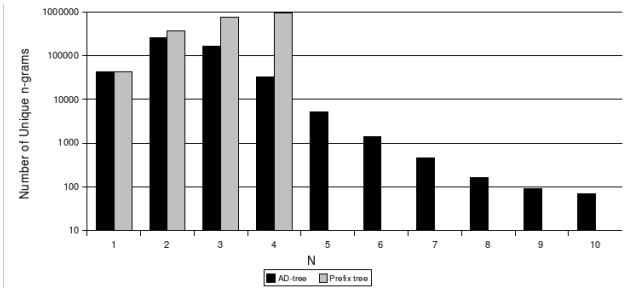

Fig. 4. Distribution of the number of unique n-grams stored for increasing values of n under the word n-gram model for the ADtree with several leaf-list thresholds and for the prefix tree. Note that the prefix tree runs out of memory for $n > 4$. Like the character-based version, the ADtree stores fewer long n-grams because they often occur infrequently enough to be handled by leaf-lists.

node (and its subtree) in order to save space actually creates a demand for an additional $40000+$ nodes and their subtrees. This is not a desirable trade-off and so the experiments with ADtrees will only make use of the other two space saving techniques (not storing zeros and terminating the leaves early as leaf-lists).

IV. Empirical Results

In order to test the space usage of this new modification and application of ADtrees, two different datasets were created from the Wall Street Journal corpus. The first takes a single document (document 03) from the corpus of 24 documents and uses it to build a character model. The second uses the first 22 documents (typically used for training models in natural language problems [1]) in order create a word model. Both documents were preprocessed to eliminate all capitalization. This is not a required preprocessing step, but it helps simplify the more naïve models.

The document used for the character model had 54 unique characters (including space). In order to give a reasonable

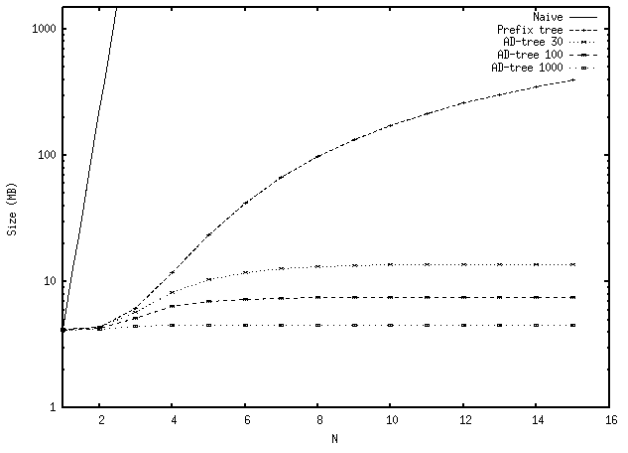

Fig. 3. Memory requirements for increasing values of n under the character n-gram model for the ADtree with several leaf-list thresholds (30, 100 and 1000) and for the prefix tree. For comparison, the naïve baseline is also included. As n increases, the ADtree exhibits better than an order of magnitude improvement over the prefix tree. (Note the logarithmic scale on the y-axis.)

comparison, both a prefix tree and a modified ADtree were created based on the same character n-grams. The prefix tree explicitly stores every n-gram from the dataset. The ADtree is essentially the prefix tree with leaf-lists (and a little extra overhead). Fig. 2 shows the distributions of the number of counts stored for increasing values of n for each data structure. Notice that the ADtree stores fewer long n-grams than does the prefix tree. This is because longer n-grams often occur infrequently enough to be handled by leaf-lists (and thus are never stored as counts in the ADtree). Fig. 3 shows the amount of memory required to store the prefix and ADtrees. In order to demonstrate the significant effect of the leaf-list threshold value, several different ADtree models were built using threshold values of 30, 100 and 1000. Although the naïve structure that includes non-existent n-grams was not actually created it is relatively easy to calculate its approximate memory usage and that is added as a baseline to the figures showing memory usage.

The text used for the word model had $42,136$ unique words (including counting all punctuation as words). The word n-gram distribution and memory usage are shown in Figs. 4 and 5 and are similar to the character model except that the scale is substantially different. To construct the "prefix" tree for the word model, the concept of the prefix was extended to include words instead of simple characters. The only change is the additional memory needed to store the longer strings. The graphs for the prefix tree stop after 4-grams because they were too big to fit into main memory.

As was noted previously, the prefix tree has an exponential upper bound, but the logarithmic scale in the graphs shows that the average case is more likely asymptotic (in contrast to the clearly exponential growth of the naïve line). Both the prefix tree and the ADtree appear to follow a logistic curve that has a finite asymptote. The ADtree simply has a much smaller asymptote than the prefix tree, due mainly to the leaf-lists. With a prefix tree, all non-zero counts are stored explicitly, often resulting in very leafy trees, with each leaf representing a very small count; employing leaf lists in the ADtree allows the tree size to be pruned back in return for a modest increase in average time to query a count. By

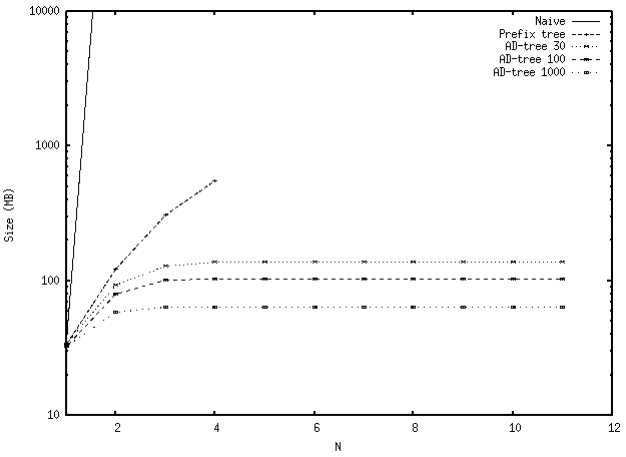

Fig. 5. Memory requirements for increasing values of n under the word n-gram model for the ADtree with several leaf-list thresholds (30, 100 and 1000) and the prefix tree. For comparison, the naïve baseline is also included. As n increases, the ADtree again exhibits better than an order of magnitude improvement over the prefix tree (the prefix tree exceeds main memory for $n > 4$). (Note the logarithmic scale on the y-axis.)

adjusting the threshold used for the leaf list, it is possible to scale the memory asymptote as desired. The direct tradeoff is that a larger threshold results generally in larger lists which must be iterated over in order to get the final requested count. Fortunately, the iteration operation is linear in the size of the list (whereas a query not using a leaf list is linear in n) and so as long as the leaf list size is within 1-2 orders of magnitude of the desired n, the time penalty should not be very noticeable.

For example, with a leaf-list threshold of 30, the character model using the ADtree has an upper limit on memory size of 13.8 MB and the word model using an ADtree has an upper limit of 138.88 MB, both of which are very reasonable given current computer memory limits (Figs. 3 and 5).

V. Discussion

It has been demonstrated that ADtrees can be adapted to the storing of n-gram counts in sequential data such as that in large natural language corpora. This adaptation of the ADtree provides a significant improvement in memory usage that can be utilized in a large number of applications where n-grams have previously been used. However, this improvement does not come without a caveat that could potentially limit its usage in some cases. Specifically, the fact that the leaf-lists store indices into the dataset means that the n-grams can not be separated from the original data (as in the case of the Google n-gram data) and that the cost of accessing the dataset through these indices must be considered. If the dataset is of a size such that it can be contained in memory along with the ADtree, then this is not a problem. However, if the dataset must be accessed from disk or remotely over a network, then the extra time required to access the dataset in these cases may be prohibitive.

An alternative approach to the storage/retrieval of n-gram counts involves trading time for space by storing a specially crafted lookup table from which n-grams can be more easily calculated [14]. This technique requires $O(l \log l)$ time to create the table (this is a character-based model and l is the number of characters in the corpus), about $6l$ bytes to store it, and $O(nl)$ time to calculate the n-gram counts. Reported results include calculating n-grams for $n \leq 255$ on a 4 MB file of Japanese with 4000 unique characters in about one hour, on a 8 MB file in about 2 hours, and on a 59 MB file in about 24 hours. For comparison, our technique handles a 6 MB file of 1 million words (42,000 unique) in a few seconds and requires 138 MB of storage (because the majority of the counts are precomputed and stored explicitly). This comparison should not be taken too literally, but it does demonstrate, again, that our technique performs favorably to the state-of-the-art.

It is interesting to note that the primary difference between the prefix tree and modified ADtree is the leaf-list concept. Although our approach was motivated by the space savings exhibited by ADtrees, the main reason for our improvement over traditional prefix trees is the leaf-list. This suggests that a simple modification could be made to existing implementations of prefix trees so that they make use of leaf-lists and thereby enjoy a similar reduction in memory requirements. Additionally, generalizing the prefix tree to do word-based n-grams shows that prefix trees can be more widely applied than to just character prefixes as they usually are.

As mentioned in Sec. II, some work has been done to improve the memory profile of ADtrees, particularly where the full ADtree may not be able to fit in memory. This includes dynamic (lazy) generation as well as pruning less recently used nodes. Both of these adaptations could be utilized to extend the work presented here to much larger datasets. For instance, if the original Google dataset or a comparably sized one became available, the resulting modified ADtree could easily be too large for main memory at fairly small n given that it has over 13 million unique words — several orders of magnitude larger than the data set considered here.

In addition, the idea of an ADtree for storing n-grams may be applied to fields such as bio-informatics, where gene sequence matching and related problems may benefit from an efficient storage mechanism for sequences (n-grams) and "n-grams" of sequences. Indeed, one can imagine generalizing the approach to other non-Markovian "n-gram" models (i.e. the modified ADtree approach could be used to store counts of more general co-occurance statistics).

References

[1] T. Brants, "Tnt: A statistical part-of-speech tagger," in *Proceedings of the 6th Applied Natural Language Processing Conference*, 2000, pp. 224–231.

[2] B. Carpenter, "Scaling high-order character language models to gigabytes," in *Proceedings of the Association for Computational Linguistics Workshop on Software*, 2005.

[3] S. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Computer Speech and Language*, vol. 13, no. 44, pp. 359–394, 1999.

[4] F. Damerau, *Markov Models and Linguistic Theory; An Experimental Study of a Model for English*. Walter de Gruyter, Inc., 1971.

[5] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.

[6] Z. Gu and D. Berleant, "Hash table sizes for storing n-grams for text processing," Software Research Lab, Department of Electrical and Computer Engineering, Iowa State University, Tech. Rep. Tech Report 10-00a, 2000.
[7] E. Keedwell, *Intelligent Bioinformatics: The Application of Artificial Intelligence Techniques to Bioinformatics Problems*. John Wiley and Sons, 2005.
[8] C. Manning and H. Schultze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
[9] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1994.
[10] L. D. Consortium, " http://www.ldc.upenn.edu/catalog/catalogentry.jsp?catalogid=ldc2006t13," 2006.
[11] A. Moore and M. Lee, "Cached sufficient statistics for efficient machine learning with large datasets," *Journal of Artificial Intelligence Research*, vol. 8, pp. 67–91, 1998.
[12] P. Komarek and A. Moore, "A dynamic adaptation of ad-trees for efficient machine learning on large data sets," in *Proceedings of the International Conference on Machine Learning*, 2000, pp. 495–502.
[13] J. Roure and A. Moore, "Sequential update of adtrees," in *Proceedings of the International Conference on Machine Learning*, 2006, pp. 769–776.
[14] M. Nagao and S. Mori, "A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese," in *Proceedings of the International Conference on Computational Linguistics*, 1994, pp. 611–615.