An Accelerated Chow and Liu Algorithm: Fitting Tree Distributions to High-Dimensional Sparse Data

Marina Meilă
Center for Biological and Computational Learning
Massachusetts Institute of Technology
Cambridge, MA 02142, USA
mmp@ai.mit.edu

Abstract

Chow and Liu introduced an algorithm for fitting a multivariate distribution with a tree (i.e. a density model that assumes that there are only pairwise dependencies between variables) and that the graph of these dependencies is a spanning tree. The original algorithm is quadratic in the dimension of the domain, and linear in the number of data points that define the target distribution P. This paper shows that for sparse, discrete data, fitting a tree distribution can be done in time and memory that is typically subquadratic in the number of variables and the size of the data set. The new algorithm, called the **acCL** algorithm, achieves speed ups of several orders of magnitude in the experiments.

1 Introduction

[Chow and Liu, 1968] introduced an algorithm for fitting a multivariate distribution with a tree, i. e. a density model that assumes that there are only pairwise dependencies between variables and that the graph of these dependencies is a spanning tree. Recently, the interest in this algorithm, hitherto called the CL algorithm, has been revived by the introduction of several probabilistic models that extend the spanning tree model. The mixture of trees of [Meilă and Jordan, 1998] is a mixture model in which each component is a spanning tree with a possibly different topology over the set of variables. The TANB (Tree Augmented Naive Bayes) model of [Friedman et al., 1997] is a mixture of trees designed for classification.

Trees enjoy many properties that make them attractive as modeling tools: they are intuitively appealing, have low complexity and yet a flexible topology, have efficient and simple algorithms for sampling and inference. Moreover, trees are one of the few classes of graphical models for which structure search is tractable [Heckerman et al., 1995]. The CL algorithm finds the structure and parameters of the tree T that best fits a given distribution P as measured by the Kullback-Leibler (KL) divergence

$$KL(P\,||\,T) \;=\; \sum_x P(x)\log\frac{P(x)}{T(x)} \qquad (1)$$

The algorithm uses the mutual information I_{uv} under P between each pair of variables u, v in the domain. When P is an empirical distribution obtained from and i.i.d. set of data, the computation of all the mutual information values takes time quadratic in the number of variables n and linear in the size of the dataset N. This time is acceptable for many problems but becomes prohibitive when the dimension n of the domain becomes large.

For example, in information retrieval, the data points are documents and the variables are words from a vocabulary. A document is represented as a binary vector with a component equal to 1 for each word v that is present in the document and a 0 component for each of the non-appearing words. The number N of documents in the data set is of the order of $10^3 - 10^4$ and so is the vocabulary size. This means that fitting a tree to the data requires $n^2N \sim 10^9 - 10^{12}$ counting operations. But this domain has a particularity: each document contains only a relatively small number of words (of the order of 10^2) and therefore most of the components of its binary vector are 0. This property is called data *sparsity*.

The present paper shows how to take advantage of sparsity to improve on the time and memory requirements of the CL algorithm in order to allow the use of tree density models for higher dimensional domains. This improvement can be carried over to learning mixtures of trees or TANB models. One idea is to compare mutual informations between pairs of variables **without actually computing them** when this is possible. A second idea is to exploit the sparsity of the data to speed up the computation of the sufficient statistics. Combining the two will result in an algorithm that typically is jointly subquadratic in N and n w.r.t. both running time and memory. This algorithm, that we call **accelerated CL** algorithm (**acCL**) will be presented and analyzed in the rest of this paper.

2 Fitting a tree to a distribution

We start by introducing the tree distribution and the notation that will be used throughout the paper. For more detail the reader is recommended to consult [Chow and Liu, 1968, Meilă and Jordan, 1998, Meilă-Predoviciu, 1999]. Let V denote the set of variables of interest, $|V| = n$. Let v be a variable in V, r_v its number of values (it is assumed that all the variables are discrete), x_v a particular value of v and x an n-dimensional vector representing an assignment to all the variables in V.

Tree distributions. We use trees as graphical representations for families of probability distributions over V that satisfy a common set of independence relationships encoded in the tree topology[1]. V are the tree's vertices; E denotes its edge set. Edges show direct dependencies, or, more precisely, the absence of an edge between two variables signifies that they are independent, given all the other variables in V. Now we define a probability distribution T that is *conformal* with a tree. Let us denote by T_{uv} and T_v the marginals of T:

$$\begin{aligned} T_{uv}(x_u, x_v) &= \sum_{x_{V-\{u,v\}}} T(x_u, x_v, x_{V-\{u,v\}}) \\ T_v(x_v) &= \sum_{x_{V-\{v\}}} T(x_v, x_{V-\{v\}}). \end{aligned}$$

Let $\deg v$ be the *degree* of vertex v, e.g. the number of edges incident to $v \in V$. The distribution T is conformal with the tree (V, E) if it can be factored as:

$$T(x) = \frac{\prod_{(u,v)\in E} T_{uv}(x_u, x_v)}{\prod_{v\in V} T_v(x_v)^{\deg v - 1}} \qquad (2)$$

[1] We shall call a graph that has no cycles a *tree*. Note that in the graph theory literature, our definition corresponds to a *forest*.

The distribution itself will be called a tree when no confusion is possible.

The CL algorithm Let us now turn to learning trees from data. The observed data are denoted by $\mathcal{D} = \{x^1, x^2, \ldots, x^N\}$, where each x^i represents a vector of observations for all the variables in V. Learning a tree in the Maximum Likelihood framework means finding T^{opt} that satisfies

$$T^{opt} = \operatorname*{argmax}_T \sum_{i=1}^{N} \log T(x^i) \qquad (3)$$

This problem is a special case of the more general task of fitting a tree to a distribution P by minimizing $KL(P\,||\,T)$. The fact that the tree model is factorable allows for an efficient and elegant algorithm for solving the latter problem, owed to Chow and Liu [Chow and Liu, 1968]. The algorithm is briefly described here.

The algorithm is based on the fact that all the information about a distribution P that is necessary to fit a tree is contained in its *pairwise marginals* $P_{uv}(x_u, x_v) = \sum_{x_{V-\{u,v\}}} P(x_u, x_v, x_{V-\{u,v\}})$. If the distribution is an empirical distribution defined by a data set $\mathcal{D}$, computing these marginals from data takes $\mathcal{O}(n^2 N)$ operations.

Further, to find the tree structure, given by its set of edges E, one has to compute the mutual information between each pair of variables in V under the target distribution P

$$I_{uv} = \sum_{x_u x_v} P_{uv}(x_u, x_v) \log \frac{P_{uv}(x_u, x_v)}{P_u(x_u)P_v(x_v)}, \quad u, v \in V, u \neq v. \qquad (4)$$

Since V has n variables, there are $n(n-1)/2$ mutual informations to be computed. After that, the optimal tree structure E is found by a *Maximum Weight Spanning Tree* (MWST) algorithm using I_{uv} as the weight for edge $(u, v), \forall u, v \in V$. The MWST can be found in several ways [Cormen et al., 1990, Tarjan, 1983, Fredman and Tarjan, 1987]. Here we use the Kruskal algorithm [Cormen et al., 1990], which is essentially a greedy algorithm. The candidate edges are sorted in decreasing order of their weights (i.e. mutual informations). Then, starting with an empty E, the algorithm examines one edge at a time (in the order resulting from the sort operation), checks if it forms a cycle with the edges already in E and, if not, adds it to E. The algorithm ends when $n-1$ edges have been added to E; in this process $n_K \geq n-1$ edges are examined.

The time required by the Kruskal algorithm is $\mathcal{O}(n^2 \log n)$ corresponding to the time necessary to sort the $n(n-1)/2$ mutual informations. The Kruskal algorithm is not the fastest MWST algorithm, but it has the advantage to be compatible with the accelerating method that is proposed here. All the other known MWST algorithms have a running time at least proportional to the number of candidate edges. In this case, they will be at least quadratic in n, having only a $\log n$ advantage over the Kruskal algorithm.

Once the tree is found, its marginals T_{uv}, $(u,v) \in E$ are exactly equal to the corresponding marginals P_{uv} of the target distribution P.

$$T_{uv}^{opt} \;\equiv\; P_{uv} \;\text{ for } (u,v) \in E \tag{5}$$

In conclusion, the running time of the CL algorithm is dominated by the computation of the pairwise marginals and is

$$\mathcal{O}(n^2(N + \log n)). \tag{6}$$

3 Assumptions

Here are presented the assumptions underlying the accelerated CL algorithm. Of the four assumptions, only the sparsity assumption is essential; we shall see later on that the first two can be relaxed.

Binary variables. All variables in V take values in the set $\{0,1\}$. When a variables takes value 1 we say that it is "on", otherwise we say it is "off". Without loss of generality we can further assume that a variable is off more times than it is on in the given dataset.

The target distribution P is derived from a set of observations of size N. Let us denote by N_v the number of times variable v is on in the dataset and by N_{uv} the number of times variables u and v are simultaneously on. We call each of the latter events a *cooccurrence* of u and v. The marginal P_{uv} of u and v is given by

$$\begin{aligned} N.P_{uv}(1,1) &= N_{uv} && (7)\\ N.P_{uv}(1,0) &= N_u - N_{uv} && (8)\\ N.P_{uv}(0,1) &= N_v - N_{uv} && (9)\\ N.P_{uv}(0,0) &= N - N_v - N_u + N_{uv} && (10) \end{aligned}$$

All the information about P that is necessary for fitting the tree is summarized in the counts N, N_v and N_{uv}, $u,v = 1,\ldots,n$ that are assumed to be non-negative. From now on we will consider P to be represented by these counts. In the rest of this paper, for simplicity, the counts will be considered to be integers. But the reader should note that this condition can be dropped if necessary (as, for example, if the data points are "weighted" by real numbers).

Sparse data. Let us denote by $0 \leq |x| \leq n$ the number of variables that are on in observation x. Further, define s, the *sparsity* of the data by

$$s = \max_{i=1,N} |x^i| \tag{11}$$

If, for example, the data are documents and the variables represent words from a vocabulary, then s represents the maximum number of distinct words in a document. The time and memory requirements of the accelerated CL algorithm that we are going to introduce depend on the sparsity s. The lower the sparsity, the more efficient the algorithm. From now on, s will be assumed to be a constant and $s << n$, N.

Data/dimension ratio bounded The ratio of the number of data points N vs. the dimension of the domain n is bounded. $\frac{N}{n} \leq R$. This is a technical assumption that is plausible for large n and N.

4 The accelerated CL algorithm

First idea: Comparing mutual informations between binary non-cooccurring variables Let us focus on the pairs u,v that do not cooccur, i.e. for which $N_{uv} = 0$. For such a pair, the mutual information is

$$\begin{aligned} I_{uv} &= \qquad (12)\\ &= \frac{1}{N}\{-(N-N_u)\log(N-N_u) - (N-N_v)\log(N-N_v)\\ &\quad + (N-N_u-N_v)\log(N-N_u-N_v) + N\log N\} \end{aligned}$$

Knowing that N is fixed for the dataset, it follows that I_{uv} in the 0 cooccurrence case is function of 2 variables: N_v and N_u. Let us fix u (and consequently N_u) and analyze the variation of the mutual information w.r.t. N_v:

$$\frac{\partial I_{uv}}{\partial N_v} \;=\; \log\frac{N-N_v}{N-N_u-N_v} \;>\; 0 \tag{13}$$

It follows that if $N_{uv} = 0$ then the mutual information I_{uv} is monotonically increasing with N_v for a fixed u. Put in other terms, for a given variable u and any two variables v, v' for which $N_{uv} = N_{uv'} = 0$

$$N_v > N_{v'} \;\text{ implies that }\; I_{uv} > I_{uv'}.$$

This observation allows us to partially sort the mutual informations I_{uv} for non-cooccurring pairs u,v,

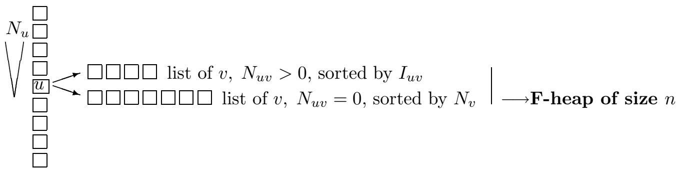

Figure 1: The data structure that supplies the next candidate edge. Vertically to the left are the variables, sorted by decreasing N_u. For a given u, there are two lists: $C(u)$, sorted by decreasing I_{uv} and (the virtual list) $V_0(u)$ sorted by decreasing N_v. The maximum of the two first elements of these lists is is inserted into an F-heap.The overall maximum of I_{uv} can then be extracted as the maximum of the F-heap.

without computing them. First, we have to sort all the variables by their number of occurrences N_v. We store the result in a list L. This gives an ordering "$\succ$" for all the variables in V.

$$v \succ u \;\Leftrightarrow\; N_v > N_u \tag{14}$$

For each u, we define the list of variable following u and not cooccurring with it:

$$V_0(u) \;=\; \{v \in V,\; v \prec u,\; N_{uv} = 0\} \tag{15}$$

This list is sorted by decreasing N_v and therefore, implicitly, by decreasing I_{uv}. Since the data are sparse, most pairs of variables do not cooccur. Therefore, by creating the lists $V_0(u)$, a large number of the mutual informations are partially sorted. Before showing how to use this construction, let us examine an efficient way of computing the N_{uv} counts when the data are sparse.

Second idea: computing cooccurrences in a list representation Let $\mathcal{D} = \{x^1, \ldots x^N\}$ be a set of observations over n binary variables. If $s << n$ it is efficient to represent each observation in $\mathcal{D}$ as a list of the variables that are on in the respective observation. Thus, data point $x^i, i = 1, \ldots N$ will be represented by the list $xlist^i = \text{list}\{v \in V | x^i_v = 1\}$. The space required by the lists is no more than sN and much smaller than the space required by the binary vector representation of the same data (i.e. nN).

Now, note that the total number N_C of cooccurrences in the dataset $\mathcal{D}$ is

$$N_C \;=\; \sum_{v \succ u} N_{uv} \;\le\; \frac{1}{2}s^2 N. \tag{16}$$

Indeed, each data point x contains at most s variables that are on, therefore contributing at most $s(s-1)/2$ cooccurrences to the sum in (16).

For the variables that cooccur with u, a set of *cooccurrence lists* $C(u)$ is created. Each $C(u)$ contains records $(v, I_{uv}),\; v \prec u, N_{uv} > 0$ and is sorted by decreasing I_{uv}. To represent the lists $V_0(u)$ (that contain many more elements) we use their "complements" $\overline{V_0}(u) = \{v \in V,\; v \prec u,\; N_{uv} > 0\}$. $\overline{V_0}(u)$ contains the same variables as $C(u)$ but no mutual informations, and is sorted by decreasing N_v. Accessing any $V_0(u)$ sequentially (in decreasing order of the [not computed] I_{uv}) is done by traversing the list L starting at the successor of u, skipping the variables contained in $\overline{V}_0(u)$.

It can be shown [Meilă-Predoviciu, 1999] that computing all the cooccurrence counts and constructing the lists $C(u), \overline{V}_0(u),\; u \in V$ takes an amount of time proportional to the number of cooccurrences N_C up to a logarithmic factor, namely

$$\mathcal{O}(s^2 N \log(s^2 N/n)). \tag{17}$$

Compare this value with $\mathcal{O}(n^2 N)$ which is the time to compute P_{uv} in the CL algorithm to see that (if the logarithmic factor is ignored) the present method replaces the dimension of the domain n by s the number of variables observed to be on in the expression of the running time. The memory requirements for the lists are also at most proportional to N_C, hence $\mathcal{O}(s^2 N)$.

Putting it all together: the acCL algorithm and its data structures. So far, we have efficient methods for computing the cooccurrences and for partially sorting the mutual informations. What we aim for is to create a mechanism that will output the edges (uv) in the decreasing order of their mutual information. If this is achieved, then the Kruskal algorithm can be used to construct the optimal tree.

We shall set up this mechanism in the form of a *Fibonacci heap* (F-heap) [Fredman and Tarjan, 1987] called *vheap* that contains an element for each $u \in V$, represented by the edge with the highest mutual information among the edges $(uv),\; v \prec u$. The maximum over this set will obviously be the maximum over the mutual informations of all the possible edges not yet eliminated. The record in *vheap* is of the form (c, u, v, I_{uv}), with $v \prec u$ and I_{uv} being the key used for sorting. Once the maximum is extracted, the used edge has to be replaced by the next largest (in terms of I_{uv}) edge in u's lists.

To perform this latter task it easy now: for each

u we have the lists $C(u)$ and $V_0(u)$. All we have to do is to compute I_{uv_0} with v_0 being the head of $V_0(u)$. By comparing I_{uv_0} with the mutual information of the head of $C(u)$ we find $\max_{v \prec u} I_{uv}$. This value, together with its corresponding u, v, N_{uv} forms the record that is inserted in the F-heap. Every time an edge (u, v, N_{uv}, I_{uv}) is inserted in *vheap*, v is deleted from $C(u)$ or respectively $V_0(u)$. The data structures involved in this process are schematically shown in figure 1 (b). Now the Kruskal algorithm can be used to construct the desired spanning tree.

Algorithm **acCL**

Input variable set V of size n
dataset $\mathcal{D} = \{xlist^i,\ i = 1, \ldots N\}$
1. compute N_v for $v \in V$;create L, list of variables in V sorted by decreasing N_v
2. compute cooccurrences N_{uv}; create lists $C(u)$, $\overline{V}_0(u),\ u \in V$
3. create *vheap*
for $u \in vlist$
$v = \underset{\text{head}C_u, \text{head}V_0(u)}{\text{argmax}} I_{uv}$
insert (c, u, v, I_{uv}) in *vheap*
4. E = **Kruskal**(*vheap*); store $c = N_{uv}$ for the edges (uv) added to E
5. for $(uv) \in E$
compute the probability table T_{uv} using N_u, N_v, N_{uv} and N.

Output T

Running time The algorithm takes: $\mathcal{O}(sn)$ for computing $N_v, v \in V$, $\mathcal{O}(n \log n)$ for sorting the variables, $\mathcal{O}(s^2 N \log(s^2 N/n))$ for step 2, $\mathcal{O}(n)$ for step 3[2], $\mathcal{O}(n_K \log n + N_C)$ for the Kruskal algorithm (n_K is the number of edges examined by the algorithm, $\log n$ is the time for an extraction from *vheap*, N_C is an upper bound for the number of elements in the lists $\overline{V}_0$ whose elements need to be skipped occasionally when we extract variables from the virtual lists V_0), $\mathcal{O}(n)$ for creating the the $n-1$ probability tables in step 5. Summing these terms we obtain the upper bound for the running time of the acCL algorithm as

$$\mathcal{O}(n \log n + sn + s^2 N \log \frac{s^2 N}{n} + n_K \log n). \quad (18)$$

If we ignore the logarithmic factor this simplifies to

$$\tilde{\mathcal{O}}(sn + s^2 N + n_K). \quad (19)$$

[2]Plus the time to skip some elements of $\overline{V}_0(u)$ if necessary. This time is accounted for by the bound for step 4.

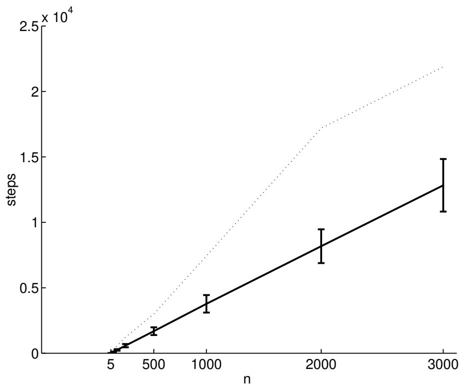

Figure 2: The mean (full line), standard deviation and maximum (dotted line) of Kruskal algorithm steps n_K over 1000 runs plotted against $n \log n$. n ranges from 5 to 3000. The edge weights were sampled from a uniform distribution.

For constant or bounded s, the above bound is a polynomial of degree 1 in the three variables n, N and n_K. But we know that n_K the total number of edges inspected by Kruskal's algorithm has the range $n - 1 \leq n_K \leq \frac{n(n-1)}{2}$. Hence, in the worst case the above algorithm is quadratic in n. However, there are reasons to believe that in practice the dependence of n_K on n is subquadratic. Random graph theory implies that if the distribution of the weight values is the same for all edges, then Kruskal's algorithm should take a number of steps proportional to $n \log n$ [West, 1996]. This result is sustained by experiments we have conducted: we ran Kruskal algorithm on sets of random weights over domains of dimension up to $n = 3000$. For each n, 1000 runs were performed. Figure 2 shows the average and maximum n_K plotted versus $n \log n$. The curves display a close to linear dependence.

Memory requirements Besides the memory for data and results, we need $\mathcal{O}(N_C)$ to store the cooccurrences and the lists $C(u), \overline{V}_0(u)$ and $\mathcal{O}(n)$ for L, *vheap* and the Kruskal algorithm. Hence, the additional space used by the acCL algorithm is $\mathcal{O}(s^2 N + n)$.

5 Generalization to discrete variables of arbitrary arity

This section will show how the Chow and Liu algorithm can be accelerated in discrete domains where the variables can take more than two values. We shall assume that a variable $v \in V$ takes r_v values, $2 \leq r_v \leq r_{MAX}$. The algorithm that we are going to develop is a simple extension of the acCL algorithm. Therefore, we shall present only the ideas that lead to extension and the modifications to the acCL algorithm that they imply.

First we redefine the data sparsity: in the forthcoming we shall assume that for each variable exists a special value that appears with higher frequency than all the other values. This value will be denoted by 0, without loss of generality. For example, in a medical domain, the value 0 for a variable would represent the "normal" value, whereas the abnormal values of each variable would be designated by non-zero values. An occurrence for variable v will be the event $v \neq 0$ and a cooccurrence of u and v means that u and v are both non-zero in the same data point. We define $|x|$ as the number of non-zero values in observation x. The sparsity s will be the maximum of $|x|$ over the data set, as before.

It is anticipated that the high frequency of the 0 values will help accelerate the tree learning algorithm. As before, we shall represent only the occurrences explicitly, creating thereby a compact and efficient data structure. Moreover, we shall demonstrate that presorting mutual informations for non-cooccurring variables in this case can be done in the same way as for binary variables.

Computing cooccurrences Following the previously introduced idea of not representing 0 values explicitly, each data point x will be replaced by the list *xlist* of the variables that occur in it: $xlist = \text{list}\{(v, x_v), v \in V, x_v \neq 0\}$.

A cooccurrence will be represented by the quadruple (u, x_u, v, x_v), $x_u, x_v \neq 0$. Counting and storing cooccurrences can be done in the same time as before and with a proportionally larger amount on memory, required by the additional need to store the (non-zero) variable values.

Instead of one cooccurrence count N_{uv} we shall now have a two-way contingency table N_{uv}^{ij}. Each N_{uv}^{ij} represents the number of data points where $u = i, v = j$, $i, j \neq 0$. This contingency table, together with the marginal counts N_v^j (defined as the number of data points where $v = j$, $j \neq 0$) and with N completely determine the joint distribution of u and v (and consequently the mutual information I_{uv}). Constructing the cooccurrence contingency tables multiplies the storage requirements of this step of the algorithm by $\mathcal{O}(r_{MAX}^2)$ but does not change the running time significantly.

Presorting mutual informations Our goal is to presort the mutual informations I_{uv} for all $v \prec u$ that do not cooccur with u. We show that this can be done exactly as before. The derivations below will be clearer if they are made in terms of probabilities; therefore, we use the notations:

$$P_v(i) \;=\; \frac{N_v^i}{N}, \quad i \neq 0; \quad P_{v0} \;\equiv\; P_v(0) \;=\; 1 - \sum_{i \neq 0} P_v(i) \tag{20}$$

The above quantities represent the (empirical) probabilities of v taking value $i \neq 0$ and 0 respectively. Entropies will be denoted by H.

A "chain rule" expression for the entropy of a discrete variable. The entropy H_v of any multi-valued discrete variable v can be decomposed in the following way:

$$\begin{aligned}
H_v \;=& \\
=&\;\; -P_{v0} \log P_{v0} - \sum_{i \neq 0} P_v(i) \log P_v(i) \\
=&\;\; \underbrace{-P_{v0} \log P_{v0} - (1 - P_{v0}) \log(1 - P_{v0})}_{H_{v0}} - (1 - P_{v0}) \\
=&\;\; \underbrace{\sum_{i \neq 0} \frac{P_v(i)}{(1 - P_{v0})} \log \frac{P_v(i)}{(1 - P_{v0})}}_{-H_{\overline{v}}} \\
=&\;\; H_{v0} + (1 - P_{v0}) H_{\overline{v}}
\end{aligned} \tag{21}$$

This decomposition represents a sampling model where first we choose whether v will be zero or not, and then, if the outcome is "non-zero" we choose one of the remaining values by sampling from the distribution $P_{v|v \neq 0}(i) = \frac{P_v(i)}{1 - P_{v0}}$. H_{v0} is the uncertainty associated with the first choice, whereas $H_{\overline{v}} \equiv H_{v|v \neq 0}$ is the entropy of the outcome of the second one. The advantage of this decomposition for our purpose is that it separates the 0 outcome from the others and "encapsulates" the uncertainty of the latters in the number $H_{\overline{v}}$.

The mutual information of two non-cooccurring variables is $I_{uv} = H_u - H_{u|v}$. The second term, the conditional entropy of u given v is

$$H_{u|v} \;=\; P_{v0} H_{u|v=0} + \sum_{j \neq 0} P_v(j) \underbrace{H_{u|v=j}}_{0} \tag{22}$$

The last term in the above formula is 0 because, for any non-zero value of v, the condition $N_{uv} = 0$ implies that u has to be 0. Let us now develop $H_{u|v=0}$ using the decomposition in equation (21).

$$H_{u|v=0} \;=\; H_{u0|v=0} + (1 - P_{u=0|v=0}) H_{u|u \neq 0, v=0} \tag{23}$$

Because u and v are never non-zero in the same time, all non-zero values of u are paired with zero values of v. Consequently $Pr[u = i|u \neq 0, v = 0] = Pr[u = i|u \neq 0]$ and $H_{u|u\neq 0,v=0} = H_{\overline{u}}$. The term $H_{u=0|v=0}$ is the entropy of a binary variable whose probability is $Pr[u = 0|v = 0]$. This probability equals

$$Pr[u = 0|v = 0] \;=\; 1 - \frac{1 - P_{u0}}{1 - P_{v0}} \tag{24}$$

Note that in order to obtain a non-negative probability in the above equation one needs $1 - P_{u0} \;\leq\; P_{v0}$, a condition that is always satisfied if u and v do not cooccur. Replacing the previous three equations in the formula of the mutual information, we get

$$\begin{aligned} I_{uv} &= P_{u0}\log P_{u0} - P_{v0}\log P_{v0} \qquad (25) \\ &= +(P_{u0} + P_{v0} - 1)\log(P_{u0} + P_{v0} - 1) \end{aligned}$$

This expression, remarkably, depends only on P_{u0} and P_{v0}. Taking its partial derivative with respect to P_{v0} yields

$$\frac{\partial I_{uv}}{\partial P_{v0}} \;=\; \log\frac{P_{v0} + P_{u0} - 1}{P_{v0}} \;<\; 0 \tag{26}$$

a value that is always negative, independently of P_{v0}. This shows the mutual information increases monotonically with the "occurrence frequency" of v given by $1 - P_{v0}$. Note also that the above expression for the derivative is a rewrite of the result obtained for binary variables in (13).

We have shown that the acCL algorithm can be extended to variables taking more than two values by making only one (minor) modification: the replacement of the scalar counts N_v and N_{uv} by the vectors N_v^j, $j \neq 0$ and, respectively, the contingency tables N_{uv}^{ij}, $i, j \neq 0$.

6 Experiments

The following experiments compare the speed of the acCL algorithm w.r.t the traditional Chow and Liu method under controlled conditions on artificial data.

The binary domain dimension n varies from 50 to 1000. Each data point has a fixed number s of variables being on. The sparsity s takes the values 5, 10, 15 and 100. The small values were chosen to gauge the advantage of the accelerated algorithm under extremely favorable conditions. The larger value will help us see how the performance degrades under more realistic circumstances. The data were generated from an artificially constructed non-uniform, non-factored distribution. For each pair n, s a set of 10,000 points was created.

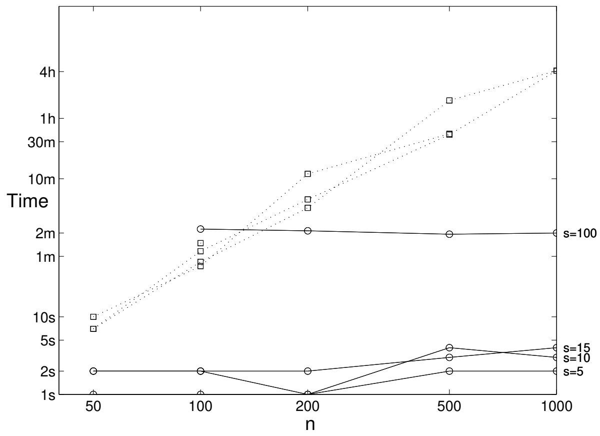

Figure 3: Real running time for the accelerated (full line) and traditional (dotted line) CL algorithms versus number of vertices n for different values of the sparsity s.

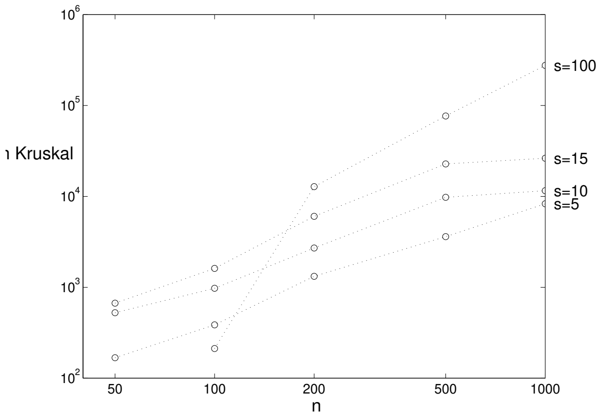

Figure 4: Number of steps of the Kruskal algorithm n_K versus domain size n measured for the acCL algorithm for different values of s.

Each data set was used by both CL and acCL to fit one tree distribution and the running times were recorded and plotted in figure 3. The improvements over the traditional version for sparse data are spectacular: learning a tree over 1000 variables from 10,000 data points takes 4 hours by the traditional algorithm and only 4 seconds by the accelerated version when the data are sparse ($s = 15$). For $s = 100$ the acCL algorithm takes 2 minutes to complete, improving on the traditional algorithm by a factor of "only" 120.

Note also that the running time of the accelerated algorithm seems to be almost independent of the dimension of the domain. On the other side, the number of steps n_K (figure 4) grows with n. This implies that the bulk of the computation lies with the steps preceding the Kruskal algorithm proper. Namely, that it is in computing cooccurrences and organizing the data that most of the time is spent. Figure 3 also confirms that the running time of the traditional CL algorithm grows quadratically with n and is independent of s.

7 Concluding remarks

This paper has presented a way of taking advantage of sparsity in the data to accelerate the tree learning algorithm. The method achieves its performance by exploiting characteristics of the data (sparsity) and of the problem (the weights represent mutual informations) that are external to the Maximum Weight Spanning Tree algorithm proper. The algorithm obtained is worst case $\tilde{\mathcal{O}}(sn + s^2N + n^2)$ but typically $\tilde{\mathcal{O}}(sn + s^2N)$ a significant asymptotic improvement over the $\mathcal{O}(n^2N)$ of the traditional algorithm. Throughout the paper we have assumed that s is constant and small. If s is large, then the acCL algorithm degrades to a traditional CL algorithm. If s varies, then the formulas provided will allow one to establish the corresponding growth rates.

The algorithm extends to non-integer counts, hence being applicable to the much richer class of mixtures of trees as well. It has been shown empirically that a very significant part of the algorithms' running time is spent in computing cooccurrences. This prompts future work on learning statistical models over large domains to focus on the efficient computation and usage of the relevant sufficient statistics. Work in this directions includes the structural EM algorithm [Friedman, 1998] and its variant [Friedman and Getoor, 1999] as well as the A-D trees of [Moore and Lee, 1998]. The latter are closely related to our representation of the pairwise marginals P_{uv} by counts. In fact, our representation can be viewed as a "reduced" A-D tree that stores only pairwise statistics. Consequently, when an A-D tree representation is already computed, it can be exploited in steps 1 and 2 of the acCL algorithm. But constructing the whole A-D tree for the data set is considerably more demanding computationally than computing the cooccurrence counts alone. The works of [de Marcken, 1998, Steeg, 1997] on discovering dependencies in subquadratic time propose heuristic methods aimed at modeling very high dimensional domains that are not sparse.

Acknowledgements

I am indebted to Paul Viola for suggesting this topic and to David Karger for many stimulating discussions along the way.

References

[Chow and Liu, 1968] Chow, C. K. and Liu, C. N. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, IT-14(3):462–467.

[Cormen et al., 1990] Cormen, T. H., Leiserson, C. E., and Rivest, R. R. (1990). *Introduction to Algorithms*. MIT Press.

[de Marcken, 1998] de Marcken, C. G. (1998). Dependency detection in subquadratic time.

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3):596–615.

[Friedman, 1998] Friedman, N. (1998). The Bayesian structural EM algorithm. In *Proceedings of the 14th Conference on Uncertainty in AI*, pages 129–138. Morgan Kaufmann.

[Friedman et al., 1997] Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, 29:131–163.

[Friedman and Getoor, 1999] Friedman, N. and Getoor, L. (1999). Efficient learning using constrained sufficient statistics. In *Proceedings of the 7th International Workshop on Artificial Intelligence and Statistics AISTATS-99*.

[Heckerman et al., 1995] Heckerman, D., Geiger, D., and Chickering, D. M. (1995). Learning Bayesian networks: the combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243.

[Meilă and Jordan, 1998] Meilă, M. and Jordan, M. I. (1998). Estimating dependency structure as a hidden variable. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Neural Information Processing Systems*, number 10, pages 584–590. MIT Press.

[Meilă-Predoviciu, 1999] Meilă-Predoviciu, M. (1999). *Learning with Mixtures of Trees*. PhD thesis, Massachusetts Institute of Technology.

[Moore and Lee, 1998] Moore, A. W. and Lee, M. S. (1998). Cached sufficient statistics for efficient machine learning with large datasets. *Journal for Artificial Intelligence Research*, 8:67–91.

[Steeg, 1997] Steeg, E. W. (1997). *Automated motif discovery in protein structure prediction*. PhD thesis, University of Toronto.

[Tarjan, 1983] Tarjan, R. E. (1983). *Data structures and network algorithms*. Society for Industrial and Applied Mathematics.

[West, 1996] West, D. B. (1996). *Introduction to Graph Theory*. Prentice Hall.