A Scalable Constant-Memory Sampling Algorithm for Pattern Discovery in Large Databases*

Tobias Scheffer[1] and Stefan Wrobel[2,3]

[1] University of Magdeburg, FIN/IWS, P.O. Box 4120, 39016 Magdeburg, Germany
scheffer@iws.cs.uni-magdeburg.de
[2] FhG AiS, Schloß Birlinghoven 53754 Sankt Augustin, Germany
[3] University of Bonn, Informatik III, Römerstr. 164, 53117 Bonn, Germany
wrobel@ais.fhg.de

Abstract. Many data mining tasks can be seen as an instance of the problem of finding the most interesting (according to some utility function) patterns in a large database. In recent years, significant progress has been achieved in scaling algorithms for this task to very large databases through the use of sequential sampling techniques. However, except for sampling-based greedy algorithms which cannot give absolute quality guarantees, the scalability of existing approaches to this problem is only with respect to the *data*, not with respect to the *size of the pattern space*: it is universally assumed that the entire hypothesis space fits in main memory. In this paper, we describe how this class of algorithms can be extended to hypothesis spaces that do not fit in memory while maintaining the algorithms' precise $\varepsilon - \delta$ quality guarantees. We present a constant memory algorithm for this task and prove that it possesses the required properties. In an empirical comparison, we compare variable memory and constant memory sampling.

1 Introduction

In many machine learning settings, an agent has to find a hypothesis which maximizes a given utility criterion. This criterion can be as simple as classification accuracy, or it can be a combination of generality and accuracy of, for instance, an association rule. The utility of a hypothesis can only be estimated based on data; it cannot be determined exactly (this would generally require processing very large, or even infinite amounts of data). Algorithms can still give stochastic guarantees on the optimality of the returned hypotheses, but guarantees that hold for *all possible* problems usually requires impractically large samples.

Past work on algorithms with stochastic guarantees has pursued two approaches — either processing a fixed amount of data and making the *guarantee* dependent on the observed empirical utility values (*e.g.,* [4, 11]), or demanding a

* Proceedings of the European Conference on Principles and Practice of Knowledge Discovery and Data Mining. 2002.

certain fixed quality and making the number of *examples* dependent on the observed utility values [17, 12, 5, 3] (this is often referred to as *sequential sampling*). The GSS algorithm [14, 15] generalizes other sequential sampling algorithms by working for arbitrary utility functions that are to be maximized, as long as it is possible to estimate the utility with bounded error.

"General purpose" sampling algorithms like GSS suffer from the necessity of representing the hypothesis space explicitly in main memory. Clearly, this is only feasible for the smallest hypothesis spaces. In this paper, we present a sampling algorithm that has a constant memory usage, independently of the size of the hypothesis space that is to be searched.

The paper is organized as follows. In Section 2, we discuss related research. We define the problem setting in Section 3. For the reader's convenience, we briefly recall the GSS sampling algorithm in Section 4 before we present our constant-memory algorithm in Section 5 and discuss the algorithm's properties. We discuss experimental results on a purchase transactions database in Section 6; Section 7 concludes.

2 Prior Work

While many practical learning algorithms heuristically try to limit the risk of returning a sub-optimal hypothesis, it is clearly desirable to arrive at learning algorithms that can give precise guarantees about the quality of their solutions. If the learning algorithm is not allowed to look at any data before specifying the guarantee or fixing the required sample size ("data-independent"), we arrive at impractically large bounds as they arise, for instance, when applying PAC learning (*e.g.,* [6]) in a data-independent way. Researchers have therefore turned to algorithms that are allowed to look at (parts of) the data first.

We can then ask two questions. Knowing that our sample will be of size m, we can ask about the quality guarantee that results. On the other hand, knowing that we would like a particular quality guarantee, we can ask how large a sample we need to draw to ensure that guarantee. The former question has been addressed for predictive learning in work on self-bounding learning algorithms [4] and shell decomposition bounds [7, 11].

For our purposes here, the latter question is more interesting. We assume that samples can be requested incrementally from an oracle ("incremental learning"). We can then dynamically adjust the required sample size based on the characteristics of the data that have already been seen; this idea has originally been referred to as sequential analysis [2, 17]. Note that even when a (very large) database is given, it is useful to assume that examples are drawn incrementally from this database, potentially allowing termination before processing the entire database (referred to as *sampling* in KDD; [16]).

For predictive learning, the idea of sequential analysis has been developed into the Hoeffding race algorithm [12]. It processes examples incrementally, updates the utility values simultaneously, and outputs (or discards) hypotheses as soon as it becomes very likely that some hypothesis is near-optimal (or very poor, respectively). The incremental greedy learning algorithm PALO [5] has been reported to require many times fewer examples than the worst-case bounds suggest.

In a KDD context, similar improvements have been achieved with the sequential algorithm of [3]. The GSS algorithm [14] sequentially samples large databases and maximizes arbitrary utility functions. For the special case of decision trees, the algorithm of [8] samples a database and finds a hypothesis that is very similar to the hypothesis that C4.5 would have found after looking at all available data.

3 Problem Setting

In many cases, it is more natural for a user to ask for the n best solutions instead of the single best or all hypotheses above a threshold. For instance, a user might find a small number of the most interesting patterns in a database, as is the case for association rule [1] or subgroup discovery [10, 18].

We thus arrive at the following problem statement and quality guarantee.

Definition 1. (Approximate n-best hypotheses problem) *Let D be a distribution on instances, H a set of hypotheses, $f : H \rightarrow I\!R^{\geq 0}$ a function that assigns a utility value to each hypothesis and n a number of desired solutions. Then let δ, $0 < \delta \leq 1$, be a user-specified confidence, and $\varepsilon \in I\!R^{+}$ a user-specified maximal error. The* approximate n-best hypotheses problem *is to find a set $G \subseteq H$ of size n such that*

> *with confidence $1-\delta$, there is no $h' \in H$: $h' \notin G$ and $f(h', D) > f_{min} + \varepsilon$, where $f_{min} := min_{h \in G} f(h, D)$.*

Previous sampling algorithms assume that all hypotheses can be represented explicitly in main memory along with the statistics of each hypothesis (*e.g.,* [12, 3, 14, 15]). Clearly, this is only possible for very small hypothesis spaces. In this paper, we only assume that there exists a generator function that enumerates all hypotheses in the hypothesis space. However, only a constant number of hypotheses and their statistics can be kept in main memory. Such generator functions exist for all practically relevant hypothesis spaces (it is easy to come up with an algorithm that generates all decision trees, or all association rules).

Most previous work has focused on the particular class of *instance-averaging* utility functions where the utility of a hypothesis h is the average of utilities defined locally for each instance. While prediction error clearly is an instance-averaging utility function, popular utility functions for other learning or discovery tasks often combine the generality of hypotheses with distributional properties in a way that cannot be expressed as average over the data records [10].

A popular example of such a discovery task is *subgroup discovery* [10]. Subgroups characterize subsets of database records within which the average value of the target attributes differs from the global average value, without actually conjecturing a value of that attribute. For instance, a subgroup might characterize a population which is particularly likely (or unlikely) to buy a certain product. The generality of a subgroup is the fraction of all database records that belong to that subgroup. The term *statistical unusualness* refers to the difference between the default probability p_0 (the target attribute taking value one in the whole database) and the probability p of a target value of one within the subgroup. Usually, subgroups are desired to be both general (large g) and statistically unusual (large $|p - p_0|$). There are many possible utility functions [10] for subgroup discovery, none of which can be expressed as the average (over all instances) of an instance utility function.

Like [14], in order to avoid unduly restricting our algorithm, we will not make syntactic assumptions about f. In particular, we will not assume that f is based on averages of instance properties. Instead, we only assume that it is possible to determine a two-sided *confidence interval* f that bounds the possible difference between true utility and estimated utility (on a sample) with a certain confidence. Finding such confidence intervals is straightforward for classification accuracy, and is also possible for all but one of the popular utility functions from association rule and subgroup discovery [14].

Definition 2 (Utility confidence interval). *Let f be a utility function, let $h \in H$ be a hypotheses. Let $f(h)$ denote the true utility of h on the instance distribution D, $\hat{f}(h, Q_m)$ its estimated quality computed based on a sample Q_m of size m, drawn* iid *from the distribution D. Then $E : \mathbb{N} \times \mathbb{R} \to \mathbb{R}$ is a* utility confidence bound *for f iff for any δ, $0 < \delta < 1$,*

$$Pr_{Q_m}[|\hat{f}(h, Q_m) - f(h)| \leq E(m, \delta)] \geq 1 - \delta \tag{1}$$

We sometimes write the confidence interval for a specific hypothesis h as $E_h(m, \delta)$. Thus, we allow the confidence interval to depend on characteristics of h, such as the variance of one or more random variables that the utility of h depends on.

4 Sequential Sampling

In this section, we summarize the generalized sequential sampling algorithm of [14] for the reader's convenience.

The algorithm (Table 1), combines sequential sampling with the popular "loop reversal" technique found in many KDD algorithms. In step 3b, we collect data incrementally and apply these to all remaining hypotheses simultaneously (step 3c). This strategy allows the algorithm to be easily implemented on top of database systems (assuming they are capable of drawing samples), *and* enables us to terminate earlier.

After the statistics of each remaining hypothesis have been updated, the algorithm checks all remaining hypotheses and (step 3(e)i) outputs those where it can be sufficiently certain that the number of better hypotheses is no larger than the number of hypotheses still to be found (so they can all become solutions), or (Step 3(e)ii) discards those hypotheses where it can be sufficiently certain that the number of better other hypotheses is at least the number of hypotheses still

Table 1. Generic sequential sampling algorithm for the n-best hypotheses problem

---

Algorithm GSS. Input: n (number of desired hypotheses), H (hypothesis space), ε and δ (approximation and confidence parameters). **Output:** n approximately best hypotheses (with confidence $1-\delta$).

1. **Let** $n_1 = n$ (the number of hypotheses that we still need to find) and **Let** $H_1 = H$ (the set of hypotheses that have, so far, neither been discarded nor accepted). **Let** $Q_0 = \emptyset$ (no sample drawn yet). **Let** $i = 1$ (loop counter).
2. **Let** M be the smallest number such that $E(M, \frac{\delta}{2|H|}) \leq \frac{\varepsilon}{2}$.
3. **Repeat until** $n_i = 0$ **Or** $|H_{i+1}| = n_i$ **Or** $E(i, \frac{\delta}{2|H_i|}) \leq \frac{\varepsilon}{2}$
   (a) **Let** $H_{i+1} = H_i$.
   (b) Query a random item of the database q_i.
   (c) Update the empirical utility $\hat{f}$ of the hypotheses in the cache; update the sample size m_i
   (d) **Let** H_i^* be the n_i hypotheses from H_i which maximize the empirical utility $\hat{f}$.
   (e) **For** $h \in H_i$ **While** $n_i > 0$ **And** $|H_i| > n_i$
      i. **If** $h \in H_i^*$ (h appears good) **And** $\hat{f}(h, Q_i) \geq E_h(i, \frac{\delta}{2M|H_i|}) + \max_{h_k \in H_i \setminus H_i^*} \left\{ \hat{f}(h_k, Q_i) + E_{h_k}(i, \frac{\delta}{2M|H_i|}) \right\} - \varepsilon$ **Then Output** hypothesis h and then **Delete** h from H_{i+1} and **Let** $n_{i+1} = n_i - 1$. **Let** H_i^* be the new set of empirically best hypotheses.
      ii. **Else If** $\hat{f}(h, Q_i) \leq \min_{h_k \in H_i^*} \left\{ \hat{f}(h_k, Q_i) - E_{h_k}(i, \frac{\delta}{2M|H_i|}) \right\} - E_h(i, \frac{\delta}{2M|H_i|})$ (h appears poor) **Then Delete** h from H_{i+1}. **Let** H_i^* be the new set of empirically best hypotheses.
   (f) **Increment** i.
4. **Output** the n_i hypotheses from H_i which have the highest empirical utility.

---

to be found (so it can be sure the current hypothesis does not need to be in the solutions). When the algorithm has gathered enough information to distinguish the good hypotheses that remain to be found from the bad ones with sufficient probability, it exits in step 3. Indeed it can be shown that this strategy leads to a total error probability less than δ as required [14].

In order to implement the algorithm for a given interestingness function a confidence bound $E(m, \delta)$ is required that satisfies Equation 1 for that specific f. In Table 2 we present a list of confidence intervals. We ask the reader to refer to [15] for a detailed treatment. All confidence intervals are based on normal approximation rather than the loose Chernoff or Hoeffding bound. z refers to the inverse normal distribution.

The simplest form of a utility function is the average, over all example queries, of some instance utility function $f_{inst}(h, q_i)$. The utility is then defined as $f(h) = \int f_{inst}(h, q_i) D(q_i) dq_i$ (the average over the instance distribution) and the estimated utility is $\hat{f}(h, Q_m) = \frac{1}{m} \sum_{i=1}^{m} f_{inst}(h, q_i)$ (average over the example queries). An easy example of an instance-averaging utility is the classification accuracy.

Table 2. Utility functions and the corresponding utility confidence bounds

$f(h)$	$E(m,\delta)$
instance-averaging	$E(m,\delta) = -\frac{z_{1-\frac{\delta}{2}}\Lambda}{2\sqrt{m}}$; $\quad E_h(m,\delta) = -z_{1-\frac{\delta}{2}} s_h$
$g(p-p_0)$ $g\lvert p-p_0\rvert$ $g\frac{1}{c}\sum_{i=1}^{c}\lvert p_i - p_{0_i}\rvert$	$E(m,\delta) = \frac{z_{1-\frac{\delta}{4}}}{\sqrt{m}} + \frac{(z_{1-\frac{\delta}{4}})^2}{4m}$ $E_h(m,\delta) = z_{1-\frac{\delta}{4}}(s_g + s_p + z_{1-\frac{\delta}{4}} s_g s_p)$
$g^2(p-p_0)$ $g^2\lvert p-p_0\rvert$ $g^2\frac{1}{c}\sum_{i=1}^{c}\lvert p_i - p_{0_i}\rvert$ $\sqrt{g}(p-p_0)$ $\sqrt{g}\lvert p-p_0\rvert$ $\sqrt{g}\frac{1}{c}\sum_{i=1}^{c}\lvert p_i - p_{0_i}\rvert$	$E(m,\delta) = \frac{3}{2\sqrt{m}} z_{1-\frac{\delta}{2}} + \frac{m+\sqrt{m}}{4m\sqrt{m}}(z_{1-\frac{\delta}{2}})^2 + \frac{1}{8m\sqrt{m}}(z_{1-\frac{\delta}{2}})^3$ $E_h(m,\delta) = (2s_g + s_p) z_{1-\frac{\delta}{2}} + (s_g^2 + 2 s_g s_p)(z_{1-\frac{\delta}{2}})^2 + s_p s_g^2 (z_{1-\frac{\delta}{2}})^3$ $E(m,\delta) = \sqrt{\frac{z_{1-\frac{\delta}{4}}}{2\sqrt{m}}} + \frac{z_{1-\frac{\delta}{4}}}{2\sqrt{m}} + \sqrt{\frac{z_{1-\frac{\delta}{4}}}{2\sqrt{m}}} \frac{z_{1-\frac{\delta}{4}}}{2\sqrt{m}}$ $E_h(m,\delta) = \sqrt{s_g z_{1-\frac{\delta}{4}}} + s_p z_{1-\frac{\delta}{4}} + \sqrt{s_g z_{1-\frac{\delta}{4}}} s_p z_{1-\frac{\delta}{4}}$

In many KDD tasks, utility functions are common that weight the generality g of a subgroup and the deviation of the probability of a certain feature p from the default probability p_0 equally [13]. Hence, these functions multiply generality and distributional unusualness of subgroups. Another class of utility functions is derived from the binomial test heuristic [9].

5 Constant Memory Sampling

Algorithm GSS as described in the preceding section has empirically been shown to improve efficiency significantly, sometimes up to several orders of magnitude [14]. However, this works only as long as the hypothesis space H can be kept in main memory in its entirety. In this section, we will therefore now develop a constant-memory algorithm which removes this restriction, i.e., processes arbitrarily large hypothesis spaces in a fixed buffer size.

To this end, assume that we can allocate a constant amount of random-access memory large enough to hold b hypotheses along with their associated statistics. The idea of the algorithm is to iteratively load as many hypotheses into our buffer as will fit, and "compress" them into a set of n solution candidates using the generic sequential sampling algorithm GSS of [14]. We store these potential solutions (let us call these first-level candidates $C^{(1)}$) in our buffer, and iterate until we have processed all of H (the ideal case), or until $C^{(1)}$ fills so much of the buffer that less than n spaces are left for new hypotheses.

To gain new space, we now compress $C^{(1)}$ in turn into a set of n candidates using the GSS algorithm, adding these to the candidate set $C^{(2)}$ at the next higher level. $C^{(2)}$ of course is also stored in the buffer. Note that we may not always gain space when compressing at level d ($d = 1, \ldots$), since the buffer may have been exhausted before $C^{(d)}$ has acquired more than n hypotheses. Thus, we repeat the compression upwards until we finally have gained space for at least one more new hypotheses. We then load as many new hypotheses as will fit, and continue the process.

When H is finally exhausted, there will be sets of candidates $C^{(d)}$ at different levels d, so we need to do one final iterated round of compressions until we are finally left with n hypotheses at the topmost level which we can return.

Table 3. Algorithm LCM-GSS

Algorithm Layered Constant-Memory Sequential Sampling. Input: n (number of desired hypotheses), ε and δ (approximation and confidence parameters), $b > n$ size of hypothesis buffer. **Output:** n approximately best hypotheses (with confidence $1 - \delta$).

1. **Let** d_{max} the smallest number such that $cap(d_{max}, b, n) \geq |H|$
2. **Let** $C^{(d)} := \emptyset$ for all $d \in \{1, \ldots, d_{max}\}$
3. **Let** $FreeMemory := b$
4. **While** $H \neq \emptyset$
   (a) **While** $FreeMemory \geq 0$
      i. **Let** $B := GEN(FreeMemory, H)$.
      ii. **Let** $C^{(1)} := C^{(1)} \cup GSS(n, \frac{\varepsilon}{d_{max}}, \delta(B), B)$.
      iii. **Let** $FreeMemory := FreeMemory - min(|B|, n)$
   (b) **Let** $d := 1$
   (c) **While** $FreeMemory = 0$
      i. **If** $C^{(d)} \neq \emptyset$ **Then**
         A. **Let** $C^{(d+1)} := C^{(d+1)} \cup GSS(n, \frac{\varepsilon}{d_{max}}, \delta(C^{(d)}), C^{(d)})$.
         B. **Let** $C^{(d)} := \emptyset$
         C. **Let** $FreeMemory := FreeMemory + |C^{(d)}| - n$
      ii. **Let** $d := d + 1$
5. **Let** $d := 1$
6. **While** $\exists d' > d : C^{(d')} \neq \emptyset$
   (a) **If** $C^{(d)} \neq \emptyset$ **Then**
      i. **Let** $C^{(d+1)} := C^{(d+1)} \cup GSS(n, \frac{\varepsilon}{d_{max}}, \delta(C^{(d)}), C^{(d)})$.
   (b) **Let** $d := d + 1$
7. **Return** $GSS(n, \varepsilon_{i+1}, \delta_{i+1}, C^{(d)})$.

The algorithm is given in detail in Table 3. In writing up our algorithm, we assume that we are given a generator GEN which can provide us with a requested number of previously unseen hypotheses from H. Such a generator can easily be defined for most hypothesis spaces using a refinement operator and a proper indexing scheme.

As the main subroutine of LCM-GSS, we use the GSS algorithm described in the preceding section. Since we use this algorithm on selected subsets of H, we must make the available hypothesis space an explicit parameter as described in Table 1. Note also that the GSS algorithm, when used on the upper levels of our algorithm, can keep the test statistics of each hypotheses acquired on lower levels, thus further reducing the need for additional samples.

In step 1, we determine the needed number of levels of compression based on the following lemma.

Lemma 1. *When using d_{max} levels, buffer size $b > n$, solution size n, algorithm LCM-GSS will process*

$$min(|H|, cap(d_{max}, b, n))$$

hypotheses, where

$$cap(d, b, n) := (b - n \cdot \lfloor(\frac{b}{n})\rfloor) + \Sigma_{i=1}^{\lfloor(\frac{b}{n})\rfloor} cap(d-1, b, n)$$

and $cap(1, b, n) := b$

Proof. (Sketch) Let us first consider the simple case of an empty buffer of size b. If we want to use only a single layer of compression, all we can do is fill the buffer and compress using GSS, so we can handle $cap(1, b, n) := b$ hypotheses. When we allow a second level of compression, in the first iteration of step 4(a)i, we can load and compress b hypotheses. In the next iteration, we need to store the n candidates from the previous iteration, so we can load only $b - n$ new hypotheses. Since at each iteration, n additional candidates need to be stored, we can repeat at most $\lfloor(\frac{b}{n})\rfloor$ times. We will then have filled $n \cdot \lfloor(\frac{b}{n})\rfloor$ buffer elements. Since the remainder is smaller than n, we can simply fill the buffer with $b-n \cdot \lfloor(\frac{b}{n})\rfloor$ additional elements, and then compress the entire buffer into a final solution (Step 7). Thus, in total, using two levels of compression the number of hyposhtese we can handle is given in Equation 3

$$cap(2, b, n) := (b - n \cdot \lfloor(\frac{b}{n}))\rfloor + \Sigma_{i=1}^{\lfloor(\frac{b}{n})\rfloor} b - (i-1)n \tag{2}$$

$$= (b - n \cdot \lfloor(\frac{b}{n})\rfloor) + \Sigma_{i=1}^{\lfloor(\frac{b}{n})\rfloor} cap(1, n, b - (i-1)n) \tag{3}$$

A similar argument can be applied when d levels are being used. We first can run $d - 1$ levels starting with an empty buffer which is finally reduced to n hypotheses, so we can then run another $d - 1$ levels, but only in a buffer of size $b - n$, etc. Again we can repeat this at most $\lfloor(\frac{b}{n})\rfloor$ times, and can then fill the remaining buffer space with less than n additional hypotheses. In general, the recursion given in the lemma results, where of course the total number of hypotheses processed will not be larger than $|H|$ due to step 4. $\diamond$

The following corollary justifies the restriction of the algorithm to buffer sizes that are larger than the number of desired solutions and shows that when this restriction is met, the algorithm is guaranteed to handle arbitrarily large hypothesis spaces.

Corollary 1. *As long as $b > n$, algorithm LCM-GSS can process arbitrarily large hypothesis spaces.*

Proof. Consider choosing $b := n + 1$. We then have $\lfloor(\frac{b}{n})\rfloor = 1$, and thus

$$cap(1, b, n) = n + 1 \tag{4}$$

$$cap(2, b, n) = (b - n \cdot \lfloor(\frac{b}{n})\rfloor) + \Sigma_{i=1}^{\lfloor(\frac{b}{n})\rfloor} cap(1, b, n) \tag{5}$$

$$= (n + 1 - n) + n + 1 = n + 2 \tag{6}$$

and so on. $\diamond$

Perhaps it is instructive to illustrate how the algorithm will operate with $b = n+1$. When first exiting the while loop (4a), $C^{(1)}$ will contain $n+2$ elements, which will be reduced to n elements of $C^{(2)}$ by the while loop (4c). The next iteration of the while loop (4a) will simply add one hypothesis to $C^{(1)}$, which will finally be compressed into n hypotheses in $C^{(3)}$ by the while loop (4c). Thus, when using d levels in this way, we can process $n + d$ hypotheses.

Lemma 2. *Algorithm LCM-GSS never stores more than b hypotheses, and is thus a constant memory algorithm.*

We are now ready to state the central theorem that shows that our algorithm indeed delivers the desired results with the required confidence.

Theorem 1. *When using buffer size b, solution size n, as long as $b > n$, algorithm LCM-GSS will output a group G of exactly n hypotheses (assuming that $|H| > n$) such that, with confidence $1 - \delta$, no other hypothesis in H has a utility which is more than ε higher than the utility of any hypothesis that has been returned:*

$$Pr[\exists h \in H \setminus G : f(h) > f_{min} + \varepsilon] \leq \delta \tag{7}$$

where $f_{min} = \min_{h' \in G}\{f(h')\}$; assuming that $|H| \geq n$.

Proof. We only sketch the proof here. Clearly, each individual compression using GSS is guaranteed to meet the guarantees based on the parameters given to GSS. When several layers are combined, unfortunately the maximal errors made in each compression layer sum up. Since we are using d_{max} layers, the total error is bounded by

$$\Sigma_{i=1}^{d_{max}} \frac{\varepsilon}{d_{max}} = d_{max} \cdot \frac{\varepsilon}{d_{max}} = \varepsilon$$

For confidence δ, we have to choose $\delta(S)$ for a hypothesis set S properly when running the algorithm. Now note that each hypothesis in H gets processed only once at the lowest level (when creating $C^{(1)}$). The capacity of this lowest level is $cap(d_{max}, b, n)$. At the next level up, the winners of the first level get processed again, up to the top-most level, so the total number of hypotheses processed (many of them several times) is

$$M := \Sigma_{i=0}^{d_{max}-1} cap(d_{max} - i, b, n)$$

Thus, the union of all hypothesis sets ever compressed in the algorithm has at most this size M. Therefore, if we allocate $\delta(S) := \delta \cdot \frac{|S|}{M}$ we know that the sum of all the individual $\delta(S)$ will not exceed δ as required. $\diamond$

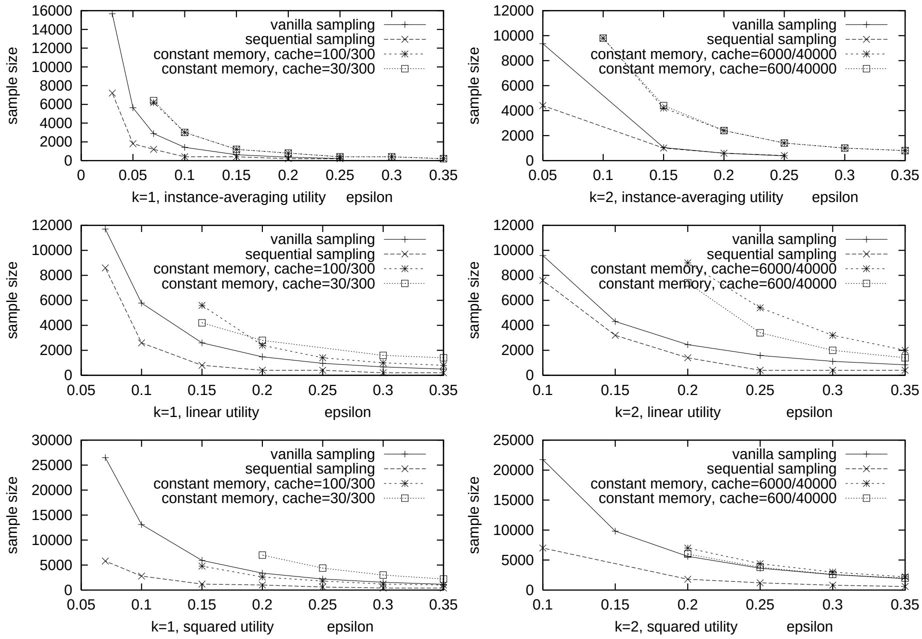

Fig. 1. Sample sizes for the juice purchases database.

6 Experiments

In our experiments, we want to study the order of magnitude of examples which are required by our algorithm for realistic tasks. Furthermore, we want to measure how many additional examples the constant memory sampling algorithm needs, compared to an algorithm with unbounded memory usage.

As baseline, we use a "vanilla" sampling algorithm that determines its sample bound without reference to the data. The vanilla algorithm determines the smallest number M that suffices to bound the utility of each hypothesis in the space with an error of up to $\epsilon/2$. Like our sequential algorithm, the vanilla sampling algorithm uses normal approximations instead of the loose Chernoff bounds.

We implemented a simple subgroup discovery algorithm. Hypotheses consist of conjunctions of up to k attribute value tests, continuous attributes are discretized in advance. The vanilla sampling algorithm determines a sample size M like our algorithm does in step 2, but using the full available error probability δ rather than only $\frac{\delta}{2}$. Hence, the non-sequential algorithm has a lower worst-case sample size than the sequential one but never exits or returns any hypothesis before that worst-case sample bound has been reached. Sequential and vanilla sampling algorithm use the same normal approximation and come with identical guarantees on the quality of the returned solution.

We used a database of 14,000 fruit juice purchase transactions. Each transaction is described by 29 attributes which specify properties of the purchased juice as well as customer attributes. The task is to identify groups of customers that differ from the overall average with respect to their preference for cans, recyclable bottles, or non-recyclable bottles. We studied hypothesis spaces of size 288 ($k = 1$, hypotheses test one attribute) and 37,717 ($k = 2$, conjunctions of two tests). We used the LCM-GSS algorithm with a cache equals to the hypothesis space, and with two decreasingly smaller cache sizes.

Since δ has only a minor (logarithmic) influence on the resulting sample size, all results presented in Figure 1 were obtained with $\delta = 0.1$. We varied the utility function; the target attribute has three possible values, so we used the utility functions $f_0 = \frac{1}{3}\sum_{i=1}^{3}|p_i - p_{0_i}|$, $f_1 = g\frac{1}{3}\sum_{i=1}^{3}|p_i - p_{0_i}|$, and $f_2 = g^2\frac{1}{3}\sum_{i=1}^{3}|p_i - p_{0_i}|$.

Figure 1 shows the sample size of the vanilla algorithm as well as the sample size required before the sequential algorithm returns the last (tenth) hypothesis and terminates. Figure 1 also shows the sample size required by LCM-GSS with two different cache sizes. In every single experiment, the sequential algorithm terminated earlier than the vanilla sampling algorithm; as ε becomes small, the relative benefit of sequential sampling can reach orders of magnitude.

When the cache is smaller than the hypothesis space, then the constant memory property has to be paid for by a larger sample size. The error constant ϵ is split up into the number of levels of the decision process. Based on the algorithm, our expectation was that LCM-GSS with two layers and error constant ϵ needs roughly as many examples as GSS with error constant $\frac{\epsilon}{2}$ when the cache is smaller than the hypothesis space but as least as large as the desired number of solutions times the number of caches needed. Our experiments confirm that, as a rule of thumb, LCM-GSS with error constant ϵ and GSS with error constant $\frac{\epsilon}{2}$ need similarly many examples.

7 Discussion

Sequential analysis is a very promising approach to reducing the sample size required to guarantee a high quality of the returned hypotheses. Sample sizes in the order of what the Chernoff and Hoeffding bounds suggest are only required when all hypotheses exhibit identical empirical utility values (in this case, identifying which one is really best is difficult). In all other cases, the single best, or the n best hypotheses can be identified much earlier.

The main contribution of this paper is a generalization of sequential analysis to arbitrarily large hypothesis spaces which we achieve by providing a fixed-memory sampling algorithm. We have to pay for the fixed-memory property by taking slightly larger sample sizes into account.

In machine learning, the small amount of available data is often the limiting factor. By contrast, in KDD the databases are typically so large that, when a machine learning algorithm is applied, computation time becomes critical.

Sampling algorithms like the GSS algorithm enable mining very (arbitrarily) large databases; the limiting factor is the main memory since all hypotheses and their statistics have to be represented explicitly. The LCM-GSS algorithm overcomes this limitation of sequential sampling and thereby enables mining very large databases with large hypothesis spaces. When we decrease the acceptable error threshold ϵ, then the computation time required to process the necessary sample size becomes the limiting factor again.

Acknowledgement

The research reported here was partially supported by Grant "Information Fusion / Active Learning" of the German Research Council (DFG), and was partially carried out when Stefan Wrobel was at the University of Magdeburg.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, 1996.
2. H. Dodge and H. Romig. A method of sampling inspection. *The Bell System Technical Journal*, 8:613–631, 1929.
3. C. Domingo, R. Gavelda, and O. Watanabe. Adaptive sampling methods for scaling up knowledge discovery algorithms. Technical Report TR-C131, Dept. de LSI, Politecnica de Catalunya, 1999.
4. Y. Freund. Self-bounding learning algorithms. In *Proceedings of the International Workshop on Computational Learning Theory (COLT-98)*, 1998.
5. Russell Greiner. PALO: A probabilistic hill-climbing algorithm. *Artificial Intelligence*, 83(1–2), July 1996.
6. D. Haussler. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation*, 100(1):78–150, 1992.
7. D. Haussler, M. Kearns, S. Seung, and N. Tishby. Rigorous learning curve bounds from statistical mechanics. *Machine Learning*, 25, 1996.
8. G. Hulten and P. Domingos. Mining high-speed data streams. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2000.
9. W. Klösgen. Problems in knowledge discovery in databases and their treatment in the statistics interpreter explora. *Journal of Intelligent Systems*, 7:649–673, 1992.
10. W. Klösgen. Explora: A multipattern and multistrategy discovery assistant. In *Advances in Knowledge Discovery and Data Mining*, pages 249–271. AAAI, 1996.
11. J. Langford and D. McAllester. Computable shell decomposition bounds. In *Proceedings of the International Conference on Computational Learning Theory*, 2000.
12. O. Maron and A. Moore. Hoeffding races: Accelerating model selection search for classification and function approximating. In *Advances in Neural Information Processing Systems*, pages 59–66, 1994.
13. G. Piatetski-Shapiro. Discovery, analysis, and presentation of strong rules. In *Knowledge Discovery in Databases*, pages 229–248, 1991.
14. T. Scheffer and S. Wrobel. Incremental maximization of non-instance-averaging utility functions with applications to knowledge discovery problems. In *Proceedings of the International Conference on Machine Learning*, 2001.

15. T. Scheffer and S. Wrobel. Finding the most interesting patterns in a database quickly by using sequential sampling. *Journal of Machine Learning Research*, In Print.
16. H. Toivonen. Sampling large databases for association rules. In *Proc. VLDB Conference*, 1996.
17. A. Wald. *Sequential Analysis.* Wiley, 1947.
18. Stefan Wrobel. An algorithm for multi-relational discovery of subgroups. In *Proc. First European Symposion on Principles of Data Mining and Knowledge Discovery (PKDD-97)*, pages 78–87, Berlin, 1997.