STREAMKRIMP: Detecting Change in Data Streams

Matthijs van Leeuwen and Arno Siebes

Department of Computer Science
Universiteit Utrecht
{mleeuwen,arno}@cs.uu.nl

Abstract. Data streams are ubiquitous. Examples range from sensor networks to financial transactions and website logs. In fact, even market basket data can be seen as a stream of sales. Detecting changes in the distribution a stream is sampled from is one of the most challenging problems in stream mining, as only limited storage can be used. In this paper we analyse this problem for streams of transaction data from an MDL perspective. Based on this analysis we introduce the STREAMKRIMP algorithm, which uses the KRIMP algorithm to characterise probability distributions with code tables. With these code tables, STREAMKRIMP partitions the stream into a sequence of substreams. Each switch of code table indicates a change in the underlying distribution. Experiments on both real and artificial streams show that STREAMKRIMP detects the changes while using only a very limited amount of data storage.

1 Introduction

Data streams are rapidly becoming a dominant data type or data source. Common examples of streams are sensor data, financial transactions, network traffic and website logs. Actually, it would also be appropriate to regard supermarket basket data as a stream, as this is often a – seemingly never-ending – flow of transactions.

Detecting change in streams has traditionally attracted a lot of attention [1,7,10,13], both because it has many possible applications and because it is a hard problem. In the financial world, for example, quick reactions to changes in the market are paramount. Supermarkets and on-line stores have to respond quickly to changing interests of their customers. As a final example, web hosts have to respond to changes in the way users use their websites.

The unbounded growth of a stream causes the biggest challenges in stream mining: after all, only limited storage and computational capacity is available. To address this, many existing algorithms use a sliding window [1,7,10]. The problem with this approach is that often a fixed window size has to be set in advance, which strongly influences the results. Some algorithms avoid this by using an adaptive window size [16]. Many current methods focus on single-dimensional item streams or multi-dimensional real-valued streams [1,2,10,11,13].

In this paper, we address the problem of detecting change in what we call *data streams*, that is, streams of transactions. A change in such a stream of transactions is a change in the distribution the transactions are sampled from. So, given a data stream,

W. Daelemans et al. (Eds.): ECML PKDD 2008, Part I, LNAI 5211, pp. 672–687, 2008.

we would like to identify, on-line, a series of consecutive substreams that have different sampling distributions.

Our approach to the problem is based on the MDL principle. We define pattern based models, called code tables, that compress the data stream. These code tables characterise the sampling distributions and allow the detection of shifts between such distributions. If we would have unbounded data storage, the MDL-optimal partitioning of the data stream would be that one that minimises the total compressed length. However, unbounded storage is clearly not realistic and we will have to resort to a solution that is at best locally optimal.

With bounded storage, the best approach is to first identify distribution P_1 at the beginning of the stream and then look for a shift to distribution P_2. When P_2 has been identified, we look for the next shift, etc. We give MDL-based solutions for both of these sub-problems.

We turn this MDL-based analysis of the problem into algorithms using our earlier Krimp algorithm [12]. It induces code tables that characterise data distributions in detail [14,15]. Here, we use the code tables given by this algorithm as foundation for the change detection algorithm StreamKrimp, which characterises streams on-the-fly.

We empirically test StreamKrimp on a variety of data streams. Results on two types of artificial streams show that changes in distribution are detected at the right moment. Furthermore, the experiment on a real stream shows that large data streams pose no problem and changes are accurately spotted while noise is neglected.

2 The Problem Assuming Unbounded Storage

2.1 Preliminaries

We assume that our data consists of a *stream* of *transactions* over a fixed set of *items* $\mathcal{I}$. That is, each transaction is simply a subset of $\mathcal{I}$ and a stream S is an unbounded ordered sequence of transactions, i.e., $S = s_1 s_2 s_3 \cdots$ in which $s_i \subseteq \mathcal{I}$. The individual transactions in a stream are identified by an integer; without loss of generality we assume that this index is consecutive.

A *finite* stream T is a *substream* of S if T consists of consecutive elements of S. In particular $S(i, j)$ is the substream that starts at the i-th element and stops at the j-th.

2.2 The Problem in Distribution Terms

We assume that S consists of a, possibly infinite, set of consecutive non-overlapping subsequences $S = S_1 S_2 S_3 \cdots$ such that

- S_i is drawn i.i.d. from distribution P_i on $\mathcal{P}(\mathcal{I})$.
- $\forall i \in \mathbb{N} : P_i \neq P_{i+1}$

So, informally, the problem is:

Given such a sequence S, identify the subsequences S_i.

The, somewhat loosely formulated, assumption underlying this problem is that the S_i are big enough to identify. If the source of the stream would change the sample distribution at every transaction it emits, identification would be impossible.

2.3 From Distributions to MDL

The two main ingredients of the problem are, of course:

1. How do we identify the distributions P_i?
2. How do we recognise the shift from P_i to P_{i+1}?

If both these problems are solved, the identification of the S_i is trivial.

If the P_i would belong to some well-known family of distributions, the first problem would be solvable by parameter estimation. Unfortunately, this is not a reasonable assumption.

Rather than trying to estimate the underlying distributions directly, we resort, again, to the Minimum Description Length principle (MDL) [9]. The MDL principle can be paraphrased as: *Induction by Compression*. Slightly more formal, this can be described as follows.

Given a set of models $\mathcal{H}$, the best model $H \in \mathcal{H}$ is the one that minimises

$$L(H) + L(D|H)$$

in which

- $L(H)$ is the length, in bits, of the description of H, and
- $L(D|H)$ is the length, in bits, of the description of the data when encoded with H.

In our earlier research on MDL for item set data we have shown that MDL captures the underlying distribution very well indeed [12,14]. In this paper, we employ MDL both to identify the P_i and to identify the shifts from P_i to P_{i+1}.

Streams of transactions are subtly different from transaction databases. The most important difference is that streams are unbounded. This means, e.g., that some care has to be taken to define the support of an item set in a stream.

2.4 Item Sets in Streams

An item set I is, as usual, a set of items. That is, $I \subseteq \mathcal{I}$. An item set I *occurs* in a transaction s_i in stream S, iff $I \subseteq s_i$. While streams may be infinite, at any point in time we will only have seen a finite substream. In other words, we only have to consider the support of item sets on finite streams. The support of an item set I on a finite stream S is defined as usual: the number of transactions in S in which I occurs.

2.5 Coding Finite Data Streams

As in our previous work, we use code tables to compress data streams. Such a code table is defined as follows.

Definition 1. *Let $\mathcal{I}$ be a set of items and $\mathcal{C}$ a set of code words. A code table CT for $\mathcal{I}$ and $\mathcal{C}$ is a two column table such that:*

1. *The first column contains item sets over $\mathcal{I}$, this column contains at least all singleton item sets and is ordered descending on item set 1) length and 2) support.*
2. *The second column contains elements from $\mathcal{C}$, such that each element of $\mathcal{C}$ occurs at most once.*

An item set $I \in \mathcal{P}(\mathcal{I})$ occurs in CT, denoted by $I \in CT$, iff I occurs in the first column of CT, similarly for a code $C \in \mathcal{C}$. For $I \in CT$, $code_{CT}(I)$ denotes its code, i.e., the corresponding element in the second column.

To encode a finite data stream *S* over $\mathcal{I}$ with code table *CT*, we use the Cover algorithm from [12] given in Algorithm 1. Its parameters are a code table *CT* and a transaction *s*, the result is a set of elements of *CT* that cover *s*. Cover is a well-defined function on any code table and any transaction *s*, since *CT* contains at least the singletons.

Algorithm 1. Cover

```
Cover($CT$, $s$)
    $T$ := first element $c \in CT$ for which $c \subseteq s$
    if $s \setminus T = \emptyset$
        then $Res$ := $\{T\}$
        else $Res$:= $\{T\} \cup$ Cover($CT$, $s \setminus T$)
    return $Res$
```

To encode finite stream *S*, we simply replace each transaction $s \in S$ by the codes of the item sets in its cover. Note, to ensure that we can decode an encoded stream uniquely, we assume that $\mathcal{C}$ is a *prefix code*.

Since MDL is concerned with the best compression, the codes in *CT* should be chosen such that the most often used code has the shortest length. That is, we should use an optimal prefix code, i.e., the Shannon code. To define this for our code tables, we need to know how often a code is used. We define the *frequency* of an item set *I* in *CT* as the number of transactions in *S* in which *I* occurs in its cover. Normalised, this frequency represents the probability that that code is used in the encoding of an arbitrary $s \in S$:

$$P(I|S) = \frac{freq_S(I)}{\sum_{J \in \mathcal{I}} freq_S(J)}$$

The optimal code length is then $-\log$ of this probability and the coding table is optimal if all its codes have their optimal length. That is, a code is optimal for *S* iff

$$|code_{CT}(I)| = -\log(P(I|S))$$

CT is code-optimal for *S* if all its codes $C \in CT$ are optimal for *S*. From now on, we assume that code tables are code-optimal, unless we state differently.

For any finite data stream *S* and any (code-optimal) code table *CT*, we can now compute *L*(*S* | *CT*). The encoded size of a transaction *s*, denoted $l_{CT}(s)$, is simply the sum of the sizes of the codes of the item sets in its cover:

$$l_{CT}(s) = \sum_{I \in \text{Cover}(CT,s)} |code_{CT}(I)|$$

The size of a data stream *S*, denoted $L_{CT}(S)$, is simply the sum of the sizes of its transactions:

$$L_{CT}(S) \quad = \quad \sum_{s \in S} l_{CT}(s)$$

The remaining problem is, what is the size of a code table? For the second column this is clear as we know the size of each of the codes, but what about the first column? For this, we use the simplest code table, i.e, the code table that contains only the singleton elements. This code table, with optimal code lengths for a finite data stream *S*, is called the standard code table for *S*, denoted by *ST*. With this choice, the size of *CT*, denoted by $L_S(CT)$, is given by:

$$L_S(CT) = \sum_{I \in CT} |code_{ST}(I)| + |code_{CT}(I)|$$

With these results, we know the total size of our encoded data stream. It is simply the sum of the size of the encoded data stream plus the size of the code table. That is, we have the following theorem.

Theorem 1. *Let S be a finite data stream over $\mathcal{I}$ and let CT be a code table that is code-optimal for S. The total size of the encoded data stream, denoted by $L(CT, S)$, is given by:*

$$L(CT, S) = L_{CT}(S) + L_S(CT)$$

Clearly, two different code tables will yield a different encoded size, an *optimal* code table is one that minimises the total size.

Definition 2. *Let S be a finite data stream over $\mathcal{I}$ and let $\mathcal{CT}$ be the set of code tables that are code-optimal for S. $CT^{opt} \in \mathcal{CT}$ is called* optimal *if*

$$CT^{opt} = \underset{CT' \in \mathcal{CT}}{\operatorname{argmin}} \, L(CT', S)$$

The total size of the stream S encoded with an optimal code table CT^{opt} is called its optimal size *and is denoted by $\mathcal{L}(S)$:*

$$\mathcal{L}(S) = L(CT^{opt}, S)$$

2.6 The Problem in MDL Terms

Now that we know how to code finite data streams, we can formulise our problem in MDL terminology:

> *Let S be a finite data stream,* partition *S into consecutive substreams $S_1, \ldots, S_k$, such that*
>
> $$\sum_{i=1}^{k} \mathcal{L}(S_i) \text{ is minimised}$$

3 The Problem Assuming Bounded Storage

3.1 The Problem of Streams

Let *S*, *T* and *U* be finite data streams, such that *U* is the concatenation of *S* and *T*. There is no guarantee that the optimal partition of *U* coincides with the optimal partition of *S* on the *S*-part of *U*. This observation points out two disadvantages of the problem as stated above.

1. It assumes knowledge of the complete stream; this is a flagrant contradiction to the main idea of data streams: they are too big to store.
2. It disregards the dynamic nature of data streams. Changes in the underlying distribution can only be detected after the whole stream has been observed. Clearly, such a posteriori results are not that useful.

In other words, we will have to settle for a partitioning that is at best *locally optimal*.

3.2 Too Large to Store: One Distribution

If the stream *S* is sampled i.i.d. from one distribution only, the estimates of $P(I \mid S(1,n))$ get closer and closer to their true value. That is, we have the following lemma.

Lemma 3. *Let data stream S be drawn i.i.d from distribution Q on* $\mathcal{P}(\mathcal{I})$*, then*

$$\forall I \in \mathcal{I} : \lim_{n \to \infty} P(I|S(1,n)) = Q(I)$$

This well-known statistical result has an interesting result for code tables: code tables converge! To make this more precise, denote by CT_n an optimal code table on S_n. Moreover, let $CT(S(1,j))$ be a shorthand for $L_{CT}(S(1,j))$.

Theorem 2. *Let data stream S be drawn i.i.d from distribution Q on* $\mathcal{P}(\mathcal{I})$*, then*

$$\forall k \in \mathbb{N} : \lim_{n \to \infty} |CT_n(S(1,n)) - CT_{n+k}(S(1,n))| = 0$$

Proof. Let *FCT* be a code table in which only the left-hand column is specified. Lemma 3 implies that

$$\forall I \in \mathcal{I}\ \forall k \in \mathbb{N} : \lim_{n \to \infty} |P(I|S(1,n)) - P(I|S(1,n+k))| = 0$$

In other words, the optimal codes we assign to the item sets in *FCT* become the same in the limit. But this implies that an optimal code table on $S(1, n + k)$ is, in the limit, also an optimal code table on $S(1, n)$. □

That is, if our stream comes from one distribution only, we do not need to store the complete stream to induce the optimal code table. A large enough sample suffices. Denote by $CT^{app}(S)$ the optimal code table induced from a large enough "head" of the stream, i.e., after convergence has set in. This head of the stream is denoted by $H(S)$.

Note that, Theorem 2 also suggests a way to check that the sample is large enough. If for some reasonable sized *k*,

$$|L(S(1,n), CT_n) - L(S(1,n), CT_{n+k})|$$

gets small, we may conclude convergence. Small is, of course, a relative notion: if $L(S(1, n), CT_n)$ is millions of bits, a difference of a few thousand bits can already be considered as small. Hence, it is better to look at a weighted version; which is our improvement rate, defined as follows.

Definition 3. *With the notations from above, the* Improvement Rate *IR is given by:*

$$\frac{|L(S(1,n),CT_n) - L(S(1,n),CT_{n+k})|}{L(S(1,n),CT_n)}$$

When *IR* becomes small in an absolute sense, we may conclude convergence. We return to this observation later.

3.3 Too Large to Store: Detecting Change

So, for a data stream that comes from one distribution, the problem is easy. The optimal code table can be computed from a large enough head of the stream. After this code table has been computed, no further data storage is necessary anymore. The problem is, however, that after a while the distribution changes. How can we detect that?

Let the, finite, stream $S = S_1S_2$ such that S_i is sampled from distribution P_i. Moreover, let CT_i denote the optimal code table on S_i. To detect the change in distribution, we need that:

$$L(S_1, CT_1) + L(S_2, CT_2) < L(S, CT)$$

This equation translates to:

$$L(S_1, CT_1^{app}) + L(S_2, CT_2^{app}) < L(S_1, CT_1^{app}) + L(S_2|CT_1^{app})$$

Note that $L(S, CT)$ translates to the sum of the two heads encoded with CT_1^{app} because CT_1^{app} has converged. That is, if there is *no* change in the underlying distribution, CT_1^{app} is still the correct code table. The second summand has the bar |, since we count $L(CT_1^{app})$ only once.

Because S may be too big to store, we store $H(S)$. To weigh both code tables equally, we approximate the inequality as follows in the definition of a *split*.

Definition 4. *Let the, finite, stream $S = S_1S_2$ such that S_i is sampled from distribution P_i. Moreover, let CT_i^{app} denote the approximated optimal code table for S_i. The pair (S_1, S_2) is called a* split *of S if:*

$$L(H(S_2), CT_2^{app}) < L(H(S_2), CT_1^{app})$$

A split is called minimal *if there is no other split (T_1, T_2) of S such that T_1 is a substream of S_1.*

Note that this definition implies that we do *not* have to store $H(S_1)$ to detect a change in the underlying definition. CT_1^{app} provides sufficient information.

3.4 The Problem for Data Streams with Bounded Storage

We can now formalise our problem for data streams with bounded storage.

> *Let S be a data stream,* partition *S into consecutive substreams $S_1, \ldots, S_k, \ldots$, such that*
>
> $$\forall S_i : (S_i, S_{i+1}) \text{ is the minimal split of } S_i S_{i+1}$$

4 The Algorithm

4.1 Krimp Preliminaries

In [12] we proposed a heuristic algorithm – later called Krimp – to approximate the optimal code table from a database. For this, it needs a database and a set of candidate item sets. As candidates, all or closed frequent item sets up to a given *minsup* are used. The candidate set is ordered descending on support, item set length and lexicographically. The algorithm starts with the standard code table *ST*. The code table is ordered descending on length and support. One by one, each pattern in the candidate set is added to the code table to see if it helps to improve database compression. If it does, it is kept in the code table, otherwise it is removed. After this decision, the next candidate is tested. Pruning is applied in all experiments reported in this paper, meaning that each time an item set is kept in the code table, all other elements are tested to see whether they still contribute to compression. Elements that don't are permanently removed. See [12] for further details.

Furthermore, in [15] we introduced a method that can be used to generate databases from a Krimp code table. All statistics showed that the generated databases are very similar to the original databases from which the code tables were induced. We will use this method to generate synthetic streams in the experiment section.

4.3 Finding the Right Code Table on a Stream

We can now translate the formal scheme presented in Subsection 3.2 to a practical implementation: assume that the stream *S* is sampled i.i.d. from one distribution only and find CT^{app} using Krimp.

The general idea of the algorithm presented in Algorithm 2 is simple: run Krimp on the growing head of a stream *S* until the resulting code tables converge. As we know that individual transactions don't make a large difference, we work with blocks of *blockSize* transactions. Start with one block and obtain a code table. For each block added to the head, a new code table is induced and the Improvement Rate is computed. Whenever the IR drops below *maxIR*, the code table is good enough and returned.

The other parameters are an *offset* that makes it possible to start anywhere within the stream and the *minsup* used for mining Krimp candidates.

ALGORITHM 2. FINDCODETABLEONSTREAM

```
FindCodeTableOnStream(S, offset, blockSize, minsup, maxIR)
    numTransactions = blockSize
    CT = Krimp(S(offset, offset+numTransactions), minsup)
    ir = Infinite
    while ir > maxIR
        numTransactions += blockSize
        newCT = Krimp(S(offset, offset+numTransactions), minsup)
        ir = ComputeIR(CT, newCT)
        CT = newCT
    return CT
```

Moreover, a Laplace correction is applied to each code table returned by KRIMP; this to ensure that each code table can encode each possible transaction.

4.4 Detecting Change in a Stream

Given a code table induced on the head of a data stream, we would now like to detect change in the sampling distribution of the rest of the stream. More formally, we would like to detect the minimal split given CT_1^{app}.

The minimal split can be found by inducing code tables on consecutive heads of the stream until a split is encountered. We would rather avoid building a code table for each and every consecutive head, but luckily we can speed things up in two different ways. First of all, change does not come in a single transaction, so again we iterate over blocks instead. Secondly, we can skip each block that obviously belongs to CT_1^{app}.

For this second optimisation, we apply a statistical test that tests whether the encoded size of the current block deviates from the expected size. If it does not, discard it and skip to the next block. Before discarding the head of a converged code table, this data is used to randomly sample encoded block sizes from. Both the lower and upper *leaveOut* percent samples are removed. If the encoded size of a new block falls within the range of the remaining samples, the block is considered to belong to the distribution of CT_1^{app} and skipped.

For each block that is not skipped, we have to test whether it marks a split or not. For this, we have to induce a code table CT_2^{app}. To be able to reject a code table that is only just better than the previous one, we introduce the Code Table Difference:

Definition 5. *Given a code table* CT_1^{app} *and a code table* CT_2^{app} *induced on H(S_2), the* Code Table Difference *CTD is given by:*

$$\frac{L(H(S_2), CT_1^{app}) - L(H(S_2), CT_2^{app})}{L(H(S_2), CT_2^{app})}$$

Normalised the same way as the Improvement Rate, the CTD tells us how many percent CT_2^{app} compresses the new head better than CT_1^{app}. We can now define a minimum CTD in the overall algorithm, which is presented next.

Algorithm 3: StreamKrimp

```
1   StreamKrimp(S, minsup, blockSize, maxIR, leaveOut, minCTD)
2       $i = 1$
3       $CT_i$ = FindCodeTableOnStream(S, 0, blockSize, minsup, maxIR)
4       pos = $CT_i$.endPos
5       while pos < sizeof(S)
6           pos = SkipBlocks(S, $CT_i$, pos, blockSize, leaveOut)
7           candCT = FindCodeTableOnStream(S, pos, blockSize, minsup, maxIR)
8           if ComputeCTD(S, $CT_i$, candCT) >= minCTD
9               i++
10              $CT_i$ = candCT
11              pos = candCT.endPos
12          else
13              pos += blockSize
14      return CT
```

4.5 StreamKrimp

Putting together the algorithms of the previous subsections, we are able to partition a stream into consecutive substreams with minimal splits. The complete algorithm is shown in Algorithm 3.

It starts with finding the code table on the head of the stream (line 3) and then iterates over the rest of the stream. Each iteration starts with skipping as many blocks as possible (6). When a block cannot be skipped straightaway, it is used as starting position for a new candidate code table (7). The Code Table Difference of this candidate to the current code table is computed (8) and the code table is either accepted (9-11) or rejected (13). When finished, the complete set of code tables is returned (14). Naturally, these could be inspected and used while the algorithm is still running as well.

4.6 How Large is Zero?

Or: How should we set our parameters? We will here motivate the default values we suggest for the algorithm, which we will use throughout the rest of the paper.

minsup – Lower *minsup* levels result in more candidate patterns and therefore better compression and better quality code tables. Sensitivity of the change detection scheme is influenced through this parameter: lower values result in a higher sensitivity. To avoid overfitting on very small data segments, we use a *minsup* of 20 in the experiments presented in this paper.

blockSize – The resolution at which StreamKrimp works should always be high enough. This will be the case if we choose the size of a block such that, on average, every possible item occurs once in every block. Therefore, we choose *blockSize* to be equal to $|\mathcal{I}|$.

leaveOut – Set to 0.01: both the lower 1% and the upper 1% of the randomly sampled blocksizes are discarded by SkipBlocks.

maxIR – Set to 0.02: if a new code table compresses less than 2% better than its predecessor, we decide it has converged.

minCTD – Set to 0.10: a new code table is accepted only if it compresses at least 10% better than the previous code table.

The choices for *maxIR* and *minCTD* may seem arbitrary, but this is not the case. They are actually comparable to the dissimilarity values we reported before [15]. Dissimilarity values between random samples from a single dataset range from 0.045 to 0.177 on UCI datasets (also reported on in the next section). Therefore, 0.02 and 0.10 are very conservative and may be considered zero for all practical purposes: with these thresholds, code tables converge and significant changes are detected.

5 Experiments

5.1 Artificial Streams – UCI Datasets

The first series of experiments is done on a selection of the largest datasets from the well-known UCI repository [6], as shown in Table 1. These datasets are transaction databases and not streams, but they have the advantage that each of them consists of multiple known classes. This allows for easy validation of the identified splits.

To transform a UCI dataset into a stream, each dataset is split on class label and the class labels are removed from all transactions. This results in a transaction database per class. The transactions within each database are ordered (randomly) to form a stream. The resulting streams are concatenated into one single stream (in random order). Because of this randomisation, each dataset is turned into a stream 10 times.

The main results are summarised in Table 2. The '#CTs' column tells us how many code tables have been identified for each of the datasets. If we compare these numbers to the actual number of classes in Table 1, we see that STREAMKRIMP finds the right number of distributions in the stream. Only for Chess, the algorithm doesn't find enough splits, but this is not surprising as there are quite many classes and some of them are rather small. Analysing the splits reveals that indeed the larger classes are identified and only the smallest ones go undetected.

The next column, 'Blocks per CT', tells us that approximately 4 to 6 blocks are enough for code table construction on these datasets. For some datasets, such as Adult, Chess and Nursery, quite some code tables are rejected, as is shown under '#CTs rejected'. However, also quite some blocks are skipped by SkipBlocks. These values vary quite a bit for the different datasets, telling us that the conservative statistical skip test seems to work better for one dataset than for another.

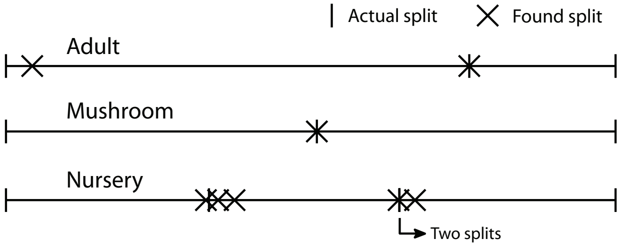

Fig. 1. Actual and found splits for three of the datasets

The last columns show baseline and obtained purity values. Purity is the size of the majority class relative to the entire segment, averaged over all identified segments. Baseline purity is the size of the largest class. Although the transactions of the classes are not interleaved and the task is therefore easier than clustering, the attained purity values are very high. This indicates that the algorithm correctly identifies the boundaries of the classes in the streams. This is supported by Figure 1, which depicts actual and found splits for three datasets. No expected splits are missed by StreamKrimp and the shift moments are accurately detected. (For each dataset, a single run with average performance is shown.)

Table 1. Properties of 7 UCI datasets: number of rows, classes and items

Dataset	**#rows**	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{I}\rvert$
Adult	48842	2	97
Chess (kr-k)	28056	18	58
Led7	3200	10	24
LetRecog	20000	26	102
Mushroom	8124	2	119
Nursery	12960	5	32
PenDigits	10992	10	86

5.2 Artificial Streams – Synthetic

The experiments in the previous subsection show that the proposed algorithm accurately detects changes in a stream. The objective of the following experiments is simple: which elementary distribution changes are detected?

We manually create simple code tables and use the Krimp data generator [15] to generate a series of synthetic datasets. Each generated stream consists of two parts: 5000 rows generated with one code table, followed by 5000 rows generated by a variation on this code table. In these experiments, $|\mathcal{I}|$ is 10 and each item is in the code table with a count of 1 (i.e., all individual items are generated with equal probability). The databases have 5 two-valued attributes (resulting in a total of 10 possible items). So, each transaction consists of 5 items.

Because the number of different items is very small (only 10) we manually set the *blockSize* for these experiments to 200. This way, we ensure that Krimp gets enough data and candidate patterns to learn the structure that is in the data.

Table 2. Results for 7 UCI datasets. For each dataset, the following is listed: the number of code tables found, the average number of blocks used for construction per CT, the number of code tables rejected, the number of blocks skipped and finally base and obtained purity. Averages over 10 randomisations (class order, transaction order within classes).

Dataset	**#CTs**	**Blocks per CT**	**#CTs rejected**	**Blocks skipped**	**Purity**	
					Baseline	*Actual*
Adult	3.4	4.7	118	367	76.1%	99.7%
Chess (kr-k)	13	4.2	165	264	17.8%	80.1%
Led7	12	3.9	3.5	82	10.9%	95.2%
LetRecog	27	5.9	0.2	32	4.1%	80.1%
Mushroom*	2.7	6.2	6.2	44	51.7%	96.5%
Nursery	6.2	5.7	140	228	33.3%	98.5%
PenDigits	15	6.0	2.3	34	10.4%	87.2%

* Closed frequent item sets used as candidates instead of all frequent item sets.

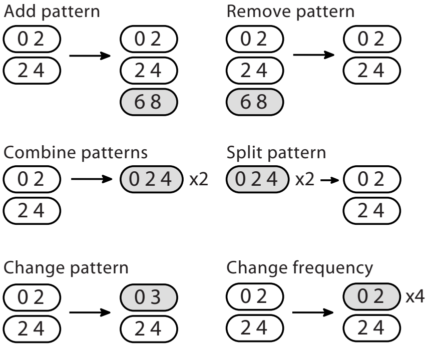

Fig. 2. Changes inflicted in the code tables used for stream generation

The basic distribution changes made halfway in the synthetic datasets (through changing the code tables) are depicted in Figure 2. Each rounded box represents an item set, with the individual items given as numbers. All counts are set to 5, except for those item sets where a multiplier is shown (x4 means 5x4=20). As data generation is a stochastic process, 10 streams were generated for each change and the main results shown in Table 3 are averaged over these. The last column indicates the number of times the algorithm found the optimal solution; two code tables and 100% purity (i.e., a split after 5000 rows). From the results it is clear that StreamKrimp is very capable at detecting 4 out of 6 of the tested types of change. Only detection of the subtle addition of a single (small) pattern and changing the frequency of an existing pattern turns out to be difficult occasionally. In these cases, change is often detected but this takes some blocks, resulting in lower purity values.

5.3 A Real Stream – Accidents Dataset

A more realistic dataset is taken from [8]. It contains data obtained from the National Institute of Statistics (NIS) for the region of Flanders (Belgium) for the period 1991-2000. More specifically, the data are obtained from the Belgian 'Analysis Form for Traffic Accidents' that should be filled out by a police officer for each traffic accident that occurs with injured or deadly wounded casualties on a public road in Belgium. In total, 340,184 traffic accident records are included in the data set.

No timestamps are available, but accidents are ordered on time and it is an interesting question whether structural changes can be detected. With over 340,000 transactions over a large number of items ($|\mathcal{I}| = 468$), running any regular pattern mining algorithm on the entire dataset is a challenge. Therefore, it is a perfect target for finding 'good enough' code tables and detecting change. As Krimp candidates we use closed frequent item sets with minimum support 500 and we rounded the block size to 500.

Table 3. Results for 6 synthetic streams. For each dataset, the following is listed: the number of code tables found, the number of code tables rejected, obtained purity and the number of optimal solutions found. Averages over 10 generated streams (except for the last column).

Change	#CTs	#CTs rejected	Purity %	Optimal (out of 10)
Add pattern	1	1.8	79.8	0
Remove pattern	1.3	3.8	97.0	6
Combine patterns	1.3	2.8	98.6	6
Split pattern	1.1	2.6	98.8	8
Change pattern	1.1	1.8	98.6	7
Change frequency	1.9	15.6	75.2	1

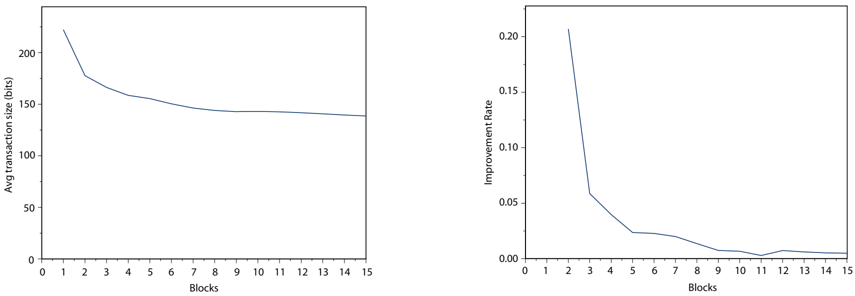

Fig. 3. Average encoded length per transaction (left) and improvement rates (right) for the code tables built on 15 consecutive blocks from the Accidents dataset

An important question we have not yet addressed with the (much smaller) artificial streams is how well the FindCodeTableOnStream algorithm approximates the best possible code table on a stream. To assess this, the average encoded size per transaction is plotted for a series of code tables in Figure 3 on the left. On the right, Figure 3 shows the computed Improvement Rates for the same set of code tables. Each code table is built on a head of *x* blocks, where *x* is the number of blocks indicated on the x-axis. Average encoded size is computed on all transactions the code table is induced from. The graphs clearly show that the most gain in compression is obtained in the first few blocks. After that, the average size per transaction decreases only slowly and this is also reflected in the Improvement Rate. With *maxIR* set to 0.02, StreamKrimp would pick the code table built on 8 blocks, which seems a good choice: after that, improvement is marginal.

Running StreamKrimp on the entire dataset resulted in only 14 code tables that characterise the entire stream of over 340,000 transactions. 140 blocks were skipped, 429 code tables were built but rejected. On average, 7.43 blocks of data were required for a code table to converge and the average Code Table Difference of accepted code tables was 0.14. This means that each consecutive distribution differs about 14% from its predecessor in terms of compression!

To further illustrate the differences between the identified substreams, Figure 4 shows compressed block sizes over the entire dataset for three consecutive code tables. Substreams clearly consist of blocks that are equally well compressed. The split the end of

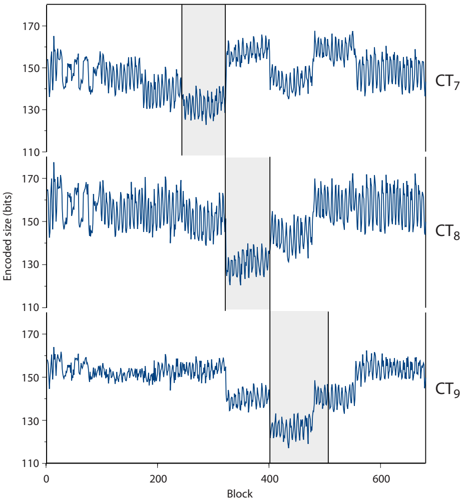

Fig. 4. Encoded length per block for three consecutive substreams on Accidents. The blocks belonging to each of the code tables are indicated with the grey blocks.

the last substream shown seems to be a tad late, the rest is spot on. In other words, the change detection is both quick and accurate, also on large datasets.

6 Related Work

Stream mining has attracted a lot of attention lately, which is nicely illustrated by the recent book by Aggarwal et al. [3]. Here, we focus on change detection.

In many cases, streams are considered to be sequences of single (real-valued) numbers or items. Kifer et al. [10] use two sliding windows to obtain two samples of which it is statistically determined whether they belong to different distributions. Papadimitriou et al. [13] use autoregressive modelling and wavelets, Muthukrishnan et al. [11] avoid a fixed window size by introducing a method based on sequential hypothesis testing.

A second class of stream mining algorithms considers multi-dimensional, real-valued streams. Aggarwal et al. [1] visualise evolving streams using velocity density estimation. The visualisation is inherently 2-dimensional and it is not possible to accurately estimate densities with increasing dimensionality. In [2], Aggarwal et al. use a polynomial regression technique to compute statistically expected values.

Dasu et al. [7] take an information-theoretic approach by using the Kullback-Leibler distance to measure the difference between distributions. They experiment on multi-dimensional real-valued data, but claim the method can also be applied to categorical data. However, a fixed window size strongly influences the changes that can be detected and the method seems better suited for relatively few dimensions (<10).

Widmer and Kubat [16] use an adaptive window size to do online learning in domains with concept drift. Predictive accuracy is used to detect drift and adjust the window size heuristically. This does require (known) binary class labels though.

The final class of algorithms considers streams of categorical transactions, as we do in this paper. Chen et al. [5] propose a method to visualise changes in the clustering structure of such streams. A disadvantage is that snapshots of these visualisations have to be manually analysed. Recently, Calders et al. [4] proposed an alternative 'minimum support' measure for patterns in streams called max-frequency. This measure uses flexible windows to maintain the max-frequency on patterns in the stream.

7 Discussion

The results on both the artificial and realistic streams show that StreamKrimp is very capable at detecting changes in large data streams. No actual splits are missed and the results on the synthetic streams show that even small modifications in the distribution can be detected.

The algorithm satisfies the general requirements for stream mining, as only very limited data storage is required and online mining is possible. Also, the resulting code tables are much smaller than the data itself and can therefore be stored for a much longer time. This means that it is possible to store a full characterisation of the entire stream.

In many stream mining algorithms, a window size has to be defined. This window size determines what changes can and can not be found; nothing outside the window is seen. Contrary, the block size of our algorithm is only the resolution which determines how quickly a distribution is detected and characterised.

8 Conclusions

We introduce StreamKrimp, an algorithm that detects changes in the sampling distribution of a stream of transactions. Based on an analysis from MDL perspective, it partitions a stream into a sequence of substreams. For each substream, it uses Krimp to characterise its distribution with a code table and each subsequent code table indicates a change in the underlying distribution. Only a very limited amount of data storage is required and StreamKrimp facilitates online mining of streams.

The results of experiments on both artificial and realistic streams show that StreamKrimp detects the changes that make a difference, no relevant changes are missed and noise is neglected. Finally, large streams with many attributes pose no problems.

References

1. Aggarwal, C.C.: A framework for diagnosing changes in evolving data streams. In: Proceedings of ACM SIGMOD 2003 (2003)
2. Aggarwal, C.C.: On Abnormality Detection in Spuriously Populated Data Streams. In: Proceedings of SIAM Conference on Data Mining 2005 (2005)
3. Aggarwal, C.C. (ed.): Data Streams: Models and Algorithms. Springer, Heidelberg (2007)
4. Calders, T., Dexters, N., Goethals, G.: Mining Frequent Itemsets in a Stream. In: Proceedings of IEEE ICDM 2007 (2007)
5. Chen, K., Liu, L.: Detecting the Change of Clustering Structure in Categorical Data Streams. In: Proceedings of SIAM Conference on Data Mining 2006 (2006)
6. Coenen, F. The LUCS-KDD Discretised/normalised ARM and CARM Data Library (2003), `http://www.csc.liv.ac.uk/~frans/KDD/Software/`
7. Dasu, T., Krishnan, S., Venkatasubramanian, S., Yi, K.: An Information-Theoretic Approach to Detecting Changes in Multi-Dimensional Data Streams. In: Proceedings of Interface 2006 (2006)
8. Geurts, K., Wets, G., Brijs, T., Vanhoof, K.: Profiling of high-frequency accident locations using association rules. In Transportation research record 1840 (2003)
9. Grünwald, P.D.: Minimum description length tutorial. In: Grünwald, P.D., Myung, I.J., Pitt, M.A. (eds.) Advances in Minimum Description Length. MIT Press, Cambridge (2005)
10. Kifer, D., Ben-David, S., Gehrke, J.: Detecting Change in Data Streams. In: Proceedings of VLDB 2004 (2004)
11. Muthukrishnan, S., van den Berg, E., Wu, Y.: Sequential Change Detection on Data Streams. In: Proceedings of the ICDM Workshops 2007 (2007)
12. Siebes, A., Vreeken, J., Van Leeuwen, M.: Item Sets That Compress. In: Proc. of the ACM SIAM Conference on Data Mining, pp. 393–404 (2006)
13. Papadimitriou, S., Brockwell, A., Faloutsos, C.: Adaptive, unsupervised stream mining. The VLDB Journal 13(3), 222–239 (2004)
14. Vreeken, J., Van Leeuwen, M., Siebes, A.: Characterising the Difference. In: Proceedings of ACM SIGKDD 2007 (2007)
15. Vreeken, J., Van Leeuwen, M., Siebes, A.: Preserving Privacy through Generation. In: Proceedings of IEEE ICDM 2007 (2007)
16. Widmer, G., Kubat, M.: Learning in the Presence of Concept Drift and Hidden Contexts. Machine Learning 23, 69–101 (1996)