Mining Local Correlation Patterns in Sets of Sequences

Antti Ukkonen
Helsinki University of Technology & HIIT
`antti.ukkonen@hiit.fi`

Abstract

Given a set of (possibly infinite) sequences, we consider the problem of detecting events where a subset of the sequences is correlated for a short period. In other words, we want to find cases where a number of the sequences output exactly the same substring at the same time. We call such substrings *local correlation patterns.* In practice we only want to find patterns that are longer than γ and appear in at least σ sequences.

Our main contribution is an algorithm for mining such patterns in an online case, where the sequences are read in parallel one symbol at a time (no random access) and the patterns must be reported as soon as they occur.

We conduct experiments on both artificial and real data. The results show that the proposed algorithm scales well as the number of sequences increases. We also conduct a case study using a public EEG dataset. We show that the local correlation patterns capture essential features that can be used to automatically distinguish subjects diagnosed with a genetic predisposition to alcoholism from a control group.

1 Introduction

Multidimensional time series and streams arise in a number of applications, such as finance (prices of securities at a stock exchange), medicine (multichannel EEG measurements) or telecommunications systems. Mining patterns in such data is a well studied topic, see for example [2, 8, 4, 14, 7].

In this paper we consider a case where the input consists of a set of sequences over some finite alphabet that are each read one symbol at a time. We propose a novel pattern class that represents local correlations among a subset of such sequences. More specifically, given the sequences, we consider the problem of finding subsets of sequences that are correlated for short periods of time by containing the same substring starting at the same position. We call such (subset, substring) pairs *local correlation patterns.*

For example, consider six time series that show the daily stock price of six different companies $C_1, \ldots, C_6$ over a number of days. We can create a modified set of time series where we mark for each day only whether the price of the stock went **u**p, **d**own, or stayed the **s**ame when compared to the previous quote. This gives us six sequences over the alphabet {u, d, s}. Below are the values of these sequences over a period of seven days:

	-6	-5	-4	-3	-2	-1	0
C_1:	u	**u**	**d**	**u**	**s**	s	d
C_2:	d	**u**	**d**	**u**	**s**	u	s
C_3:	s	s	u	d	s	d	s
C_4:	u	u	d	d	u	u	d
C_5:	s	**u**	**d**	**u**	**s**	s	u
C_6:	u	u	d	s	s	d	d

The last column, labeled with a 0, indicates the current day, while the column labeled with a -6 contains values from six days ago. The elements indicated in bold form a local correlation pattern starting at -5 with companies C_1, C_2 and C_4 and the substring ⟨**u**, **d**, **u**, **s**⟩. Another example is the substring ⟨**u**, **u**, **d**⟩ that starts at -6 and concerns companies C_1, C_4 and C_6.

In practice we have to be more specific when defining what counts as a local correlation pattern. Obviously it is possible that we observe temporary correlations in the sequences merely due to random chance, especially if there are many (e.g. hundreds of) of them. The first criteria we use is the *length* of the common substring. The second one is the *support*, i.e., the set of sequences that all contain the substring at the given position. That is, we do not expect to see a large number of sequences behaving in exactly the same way for several time steps simply by coincidence. More precisely: *We say that a string and its support form a local correlation pattern if the string is longer, and the support is larger than specified threshold values.*

The task is to efficiently find all local correlation patterns given that we obtain one symbol of each sequence at a time in an "online" fashion. Note that we could also consider an "offline" version of the problem, where all sequences support random access. However, this variant is not so interesting as it can be solved efficiently by existing algorithms. The main contribution of this paper is an efficient algorithm for the online setting.

Also, the data does not necessarily have to consist of multiple parallel sequences for our approach to be of interest. We can construct an input of the format discussed above from a single (long) string s by letting the suffixes of s be the individual sequences. That is, the suffix of s starting at position i is the ith sequence of the input. With this construction we can use our algorithm to find substrings that occur frequently inside a window of predetermined size in the string s.

In the experiments we give an example where local correlation patterns are used to classify EEG measurements. It turns out that a simple nearest mean classifier using features computed from sets of local correlation patterns can accurately distinguish subjects diagnosed with a genetic predisposition to alcoholism from the control group (see Section 4.3). This is quite interesting as we make no domain specific assumptions about the structure of the streams.

The rest of this paper is structured as follows. We give formal definitions for the problems of finding local correlation patterns in Section 2. The proposed algorithm for mining local correlation patterns is discussed in Section 3. Empirical experiments and their results are described in Section 4. Related work is covered in Section 5, and Section 6 is a short conclusion.

2 Problem definition

Let Σ be an alphabet of size $|\Sigma|$, and let s be a sequence of symbols from Σ. Symbols of Σ are denoted with letters a, b, c, ... Denote by $s(i)$ the i:th symbol of s, and by $|s|$ the length of s. Sequences are indexed starting from 1. Let the pair (i, p) denote a *pattern* where i is a positive integer and p a string over the alphabet Σ. We say the sequence s *supports* the pattern (i, p) if p appears as a substring in s starting at position i. That is, if we have

$$s(i + j - 1) = p(j) \text{ for all } j \in \{1, \ldots, |p|\}.$$

Let $D = \{s_1, \ldots, s_n\}$ be a set of sequences over Σ that are all of the same length. Denote by $\theta(i, p)$ the set of sequences in D that support the pattern (i, p). The pattern (i, p) is a *local correlation pattern* in D given the parameters γ and σ, if and only if $|p| \geq \gamma$ and $|\theta(i, p)| \geq \sigma$.

We first briefly consider the problem of finding all local correlation patterns in a set D that supports random access to the sequences.

Problem LCP-OFFLINE: Given D, γ and σ, find all local correlation patterns in D.

This can be solved using existing string-indexing techniques. We transform the strings in D by replacing the symbol $s(i)$ with $(i, s(i))$, that is, we create an extended alphabet where the positions are encoded in the symbols. Denote the new set of strings by D'. To solve LCP-OFFLINE for D, we simply find all frequent substrings in D' that are at least of length γ. This can be done efficiently by constructing either a suffix tree [13, 10, 12] or a suffix array [6] over D'.

A more interesting variant of the problem concerns an online setting where we can only read one symbol from each sequence at every time step. That is, at time step t, we read the symbol $s(t)$ from each $s \in D$. Moreover, the length of the sequences may be unbounded.

Problem LCP-ONLINE: Given γ, σ, and n sequences that each output one symbol from Σ at each time step, find all local correlation patterns and output them as soon as they appear.

In practice this definition is somewhat inconvenient, because it is possible that a local correlation pattern found at step t is only a prefix of a pattern found at step $t+1$. Consider the stock price example in the introduction. If we have set $\gamma = 3$ and $\sigma = 3$, we would first output the pattern $(2, \langle \text{u d u} \rangle)$ at step 4, and the pattern $(2, \langle \text{u d u s} \rangle)$ at step 5. This behaviour is clearly undesirable.

To overcome this issue we propose to find only the *maximal local correlation patterns*. Let ap and pa denote the sequence p with the symbol a appended to its beginning and end, respectively. The pattern (i, p) is maximal if there is no $a \in \Sigma$, such that $|\theta(i, pa)| \geq \sigma$ or $|\theta(i - 1, ap)| \geq \sigma$. In other words, the pattern (i, p) is maximal if it is not the prefix or suffix of another local correlation pattern.

Problem MAXIMAL-LCP-ONLINE: Given γ, σ, and n sequences that each output one symbol from Σ at each time step, find all maximal local correlation patterns and output them as soon as they appear.

3 An algorithm for MAXIMAL-LCP-ONLINE

In this section we describe an algorithm for the MAXIMAL-LCP-ONLINE problem. First we give an overview of the algorithm, and subsequently add details that address some issues with the general approach.

3.1 A general approach

The algorithm we propose maintains a set of candidate patterns that are prefixes of strings that may later result in a local correlation pattern with respect to some of the input sequences. The candidates must all have a support at least of size σ, but they are in general shorter than γ. Some of the candidates may be longer than γ, because we want to find maximal local correlation patterns. The candidates may thus qualify as local correlation patterns themselves, but before returning them as new patterns, we have to make sure that they can not be extended without reducing the size of their support below σ. It is easy to see that the number of candidates is trivially upper bounded by the number of sequences and the support thershold σ. Given n sequences we may have at most n/σ candidates at any given time.

Consider the following situation at step $t+1$. Suppose that $p = p(1)p(2)\ldots p(k)$ is a candidate. That is, there are at least σ sequences that all behave as specified by p, starting from $t-k+1$ and ending at t. The length of p may or may not exceed γ. At time $t+1$ we must check for all $a \in \Sigma$ what happens with the support of the extended string pa.

Obviously there are two alternatives. If the support of pa remains above σ it becomes a candidate itself and we are done. However, if the support of pa drops below σ we have to form a new candidate, and possibly output p as a maximal local correlation pattern if $|p| \geq \gamma$. To find the new candidate, note that some suffix of pa may be the prefix of some other maximal local correlation pattern. The new candidate is *the longest suffix of pa with a support larger than σ*. We may obtain several candidates based on p depending how pa behaves for different $a \in \Sigma$.

A high-level description of this idea is given in Algorithm 1. At every step $t > 1$ we call LCP0 with the set of candidates $\mathcal{C}_{t-1}$ obtained in the previous step. At step 1 we set $\mathcal{C}_1 = \{a \in \Sigma : |supp(1, a)| \geq \sigma\}$, that is, every symbol of the alphabet with a large enough support forms a candidate by itself in the beginning. The algorithm returns an updated set of candidates $\mathcal{C}_t$ and outputs maximal local correlation patterns if any are found. Note that Algorithm 1 is only meant to illustrate the general approach. It does not specify any details on how to compute the supports and what parts of the sequences to store for processing.

3.2 A detailed algorithm

Now we address some details that are needed to develop an efficient implementation of LCP0. Most importantly, we must define what the algorithm has to keep in memory in order to process the sequences. Of course a trivial implementation of LCP0 could simply store everything it reads and preform the support computations on this stored data.

Algorithm 1 LCP0: A high-level algorithm for solving MAXIMAL-LCP-ONLINE.

```
1:  LCP0($\mathcal{C}_{t-1}, \sigma, \gamma, t$)
2:  $\mathcal{C}_t \leftarrow \emptyset$
3:  for $p \in \mathcal{C}_{t-1}$ do
4:     extensionFound ← false
5:     for $a \in \Sigma$ do
6:        if $|\theta(t - |pa| + 1, pa)| \geq \sigma$ then
7:           $\mathcal{C}_t \leftarrow \mathcal{C}_t \cup \{pa\}$
8:           extensionFound ← true
9:        else
10:          $p' \leftarrow$ longest suffix $\hat{p}$ of $pa$, st. $|\theta(t - |\hat{p}| + 1, \hat{p})| \geq \sigma$
11:          $\mathcal{C}_t \leftarrow \mathcal{C}_t \cup \{p'\}$
12:       end if
13:    end for
14:    if extensionFound = false and $|p| \geq \gamma$ then
15:       output pattern $(t - |p| + 1, p)$
16:    end if
17: end for
18: return $\mathcal{C}_t$
```

3.2.1 Problems with LCP0

The first problem we address is related to representing the candidate set. First we observe that Algorithm 1 can do some unnecessary work on line 11 by adding the same string p' multiple times as a new candidate pattern. To see this, note that some candidates will have the same suffix. This suffix, appended with some $a \in \Sigma$, can be added multiple times to $\mathcal{C}_t$.

For example, let $\Sigma = \{+, -\}$, and consider the four sequences given on the left side of Figure 1. Let $\sigma = 2$. When we are at step 3 the set of candidates from the previous step is $\mathcal{C}_2 = \{+-, --\}$. First LCP0 processes the candidate $+-$. It checks the supports of both $+-+$ and $+--$, and finds both to be of size less than σ. The longest suffixes of $+-+$ and $+--$ with enough support are $-+$ and $--$, respectively. These are both added to $\mathcal{C}_3$. After this the algorithm processes the second candidate in $\mathcal{C}_2$, namely $--$. Again it finds that neither $--+$ nor $---$ have a support large enough, but the suffixes $-+$ and $--$, both of which were already added to $\mathcal{C}_3$ when processing $+-$, are again found to have enough support and are added to $\mathcal{C}_3$ for a second time.

Also, it can happen that redundant candidates are added to $\mathcal{C}_t$. These are strings that are *suffixes of some other candidates* that are added to $\mathcal{C}_t$ as well. Such a candidate is redundant, because they will be added to $\mathcal{C}$ at some later step anyway. Returning to the example, suppose that in Figure 1 the symbol appearing in sequence $s2$ at step 3 is a $-$ instead of a $+$. The set of candidates is still $\mathcal{C}_2 = \{+-, --\}$. This means that when LCP0 extends the 1st candidate with a $-$, it finds that $+--$ has enough support and adds it as a new candidate to $\mathcal{C}_3$. However, when appending a third $-$ to the 2nd candidate, it turns out that the support of $---$ is no longer at least σ. It's suffix $--$ has a support of size 3, and will be added as a new candidate, and we end up with $\mathcal{C}_3 = \{+--, --\}$. At step 4 the candidate $+--$ can not be extended with either $+$ or $-$ without decreasing the support below σ, but it's suffix $--$, appended with a $+$ has a large enough support, and $--+$ will be added to $\mathcal{C}_4$. But this will happen twice, as $--$ is the other candidate in $\mathcal{C}_3$.

3.2.2 The candidate trie

To avoid finding duplicate or redundant candidates, we represent $\mathcal{C}$ with a trie having symbols of Σ appear as labels of its edges. Let $\mathcal{T}_t$ be the trie that corresponds to the set of candidates $\mathcal{C}_t$. We will denote an internal node of the trie by N, and a leaf by L. To each node is associated a symbol $a \in \Sigma$, denoted $\text{sym}(N)$, that is the label of the edge leading to the node. Let $C(N)$ denote the set of child nodes of the node N.

We construct $\mathcal{T}_t$ so that every path starting from its root and ending at a leaf corresponds to *the reversal* of one candidate string in $\mathcal{C}_t$. That is, for every $p \in \mathcal{C}_t$ of length k, we have in $\mathcal{T}_t$ a path starting from the root and ending at a leaf L, so that the edges on the path are labeled with the symbols $p(k), p(k-1), \ldots, p(1)$. Moreover, a path in $\mathcal{T}_t$ starting from the root and ending at an arbitrary node N corresponds to the suffix of a candidate, or the common suffix of a number of candidates.

We also associate to every leaf L of $\mathcal{T}_t$ the set of sequences that support the string defined by the path from the root of $\mathcal{T}_t$ to L. Denote this by $\theta(L)$. For example, if L is the node of $\mathcal{T}_t$ that is reachable from the root by first following the edge labeled with a $-$ and then the edge labeled with a $+$, the set $\theta(L)$ contains the identifiers of all sequences that have the symbol $+$ at position $t-1$ and the symbol $-$ at position t. The trie $\mathcal{T}_2$ that corresponds to $\mathcal{C}_2 = \{+-, --\}$ of the previous example is depicted on the right side of Figure 1.

In practice we also need the supports of the suffixes of each candidate. Of course a suffix of candidate p may be supported by a number of sequences that do not belong to $\theta(p)$. This is illustrated by an example in Figure 2. The trie on the left shows all possible strings of length 2, together with their supports in some imaginary data that is not shown. Since $++$ and $--$ are only supported by one sequence each, we do not consider them frequent with $\sigma = 2$. According to the definition of $\mathcal{T}_t$ given above, the leafs corresponding to $++$ and $--$ are not stored at all. Still it is clear that $+$, which is a suffix of candidate $-+$, is supported by sequences $s1$, $s2$ and $s5$. To represent this in $\mathcal{T}_t$, we include $s1$ to the node that corresponds to $+$, as shown in the final candidate trie on the right in Fig. 2. We call this the *local support* of N, denoted $\theta_l(N)$. Given a trie defined in this way we can read the support of a string represented by the internal node N simply by computing the the union of its local support and the support of its child nodes. That is, we have

$$\theta(N) = \big\{\theta_l(N) \cup \bigcup_{N' \in C(N)} \theta(N')\big\}.$$

Before discussing the algorithm, we make some remarks about the size of $\mathcal{T}_t$. We already stated that the size of $\mathcal{C}_t$ is upper bounded by n/σ. This means that $\mathcal{T}_t$ can have at most n/σ leaf nodes. However, the length of the paths leading to the leaf nodes from the root of $\mathcal{T}_t$ can in theory be unbounded. In practice we can set an upper bound $h_{\max} > \gamma$ on the height of $\mathcal{T}_t$. Thus, $\mathcal{T}_t$ requires $O(h_{\max} n/\sigma) = O(n)$ space. Another simple but important observation is that any sequence can support at most one candidate at a time, and thus appears in at most one support list associated with a leaf node of $\mathcal{T}_t$. If a sequence does not support a candidate, it supports a suffix, and therefore has to appear in a local support

	1	2	3	4
$s1$:	$+$	$-$	$-$	$+$
$s2$:	$+$	$-$	$+$	$-$
$s3$:	$-$	$-$	$+$	$-$
$s4$:	$-$	$-$	$-$	$+$

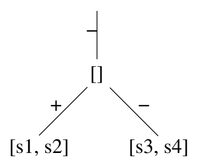

Figure 1: Left: An example data of four sequences ($s1$, $s2$, $s3$ and $s4$) of length 4 over $\Sigma = \{+, -\}$. Right: A trie representation of the candidate set $\mathcal{C}_2 = \{+-, --\}$ of the data on the left.

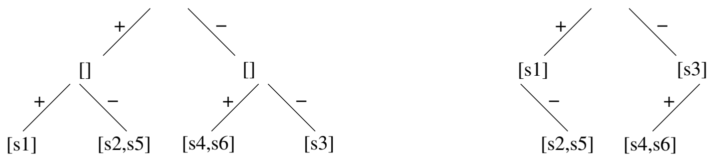

Figure 2: Left: Trie showing four strings together with their supports. Two of these, $-+$ and $+-$, are frequent at $\sigma = 2$ and are hence considered as candidates. Right: The pruned trie showing only the candidates. Additional identifiers of sequences that support candidate suffixes ($+$ and $-$ in this case) are stored in the internal nodes.

list associated with some internal node of $\mathcal{T}_t$. As a consequence the support lists also need $O(n)$ space in total.

3.2.3 Algorithm LCP1

Now, instead of processing the candidates in $\mathcal{C}_{t-1}$ individually, we traverse $\mathcal{T}_{t-1}$ to produce the updated set of candidates $\mathcal{C}_t$, represented by the trie $\mathcal{T}_t$. This algorithm is given in Algorithm 2. In short, LCP1 updates the set of candidate patterns by traversing $\mathcal{T}_{t-1}$ once for each $a \in \Sigma$ starting from its root node. Each of these traversals, implemented by the PROCESS_TRIE function shown in Algorithm 3, corresponds to appending the symbol $a \in \Sigma$ to the end of the candidates. PROCESS_TRIE returns the root node of an updated trie that will be added to $\mathcal{T}_t$. Once all symbols in Σ have been considered, we traverse $\mathcal{T}_{t-1}$ one more time and output the maximal local correlation patterns. These can be found at those leaf nodes of $\mathcal{T}_{t-1}$ that become infrequent for every $a \in \Sigma$. More precisely, every leaf L of $\mathcal{T}_{t-1}$ for which $|\theta(L) \cap S(a,t)| < \sigma$ for all $a \in \Sigma$ represents a maximal local correlation pattern.

The actual work of updating the candidate set is carried out in the PROCESS_TRIE function shown in Algorithm 3. It will return a new trie rooted at the node $\tilde{N}$ that initially has no local support or child nodes. First the algorithm recursievly processes the children of the node N. For each child N_c we obtain the new trie rooted at N'_c (line 4). This will be added as a child of $\tilde{N}$ (line 6) if it has a large enough support. Otherwise we only add it's support to the local support of $\tilde{N}$ (line 8). On line 11 we update the support of the candidate that corresponds to node N. Lines 12–14 are needed to find those leaf nodes of $\mathcal{T}_{t-1}$ that can be returned as local correlation patterns.

Above we argued that the size of a candidate trie $\mathcal{T}$ is of order $O(n)$, where n is the number of sequences. In particular, the number of leafs is upper bounded by n and σ. Thus, the trie can be traversed in time $O(n)$ if we consider $h_{\max}$ a constant. On a first look it would seem that the overall complexity of PROCESS_TRIE is higher than

Algorithm 2

```
1:  LCP1( $\mathcal{T}_{t-1}$ )
2:  $\mathcal{T}_t \leftarrow$ new trie
3:  $N \leftarrow$ root of $\mathcal{T}_{t-1}$
4:  for $a \in \Sigma$ do
5:      Let $S(a,t)$ be the set of sequences with symbol $a$ at position $t$.
6:      $N' \leftarrow$ PROCESS_TRIE( $N$, $S(a,t)$ )
7:      Add $N'$ as a new subtree to root of $\mathcal{T}_t$.
8:  end for
9:  For all leaves $L$ of $\mathcal{T}_{t-1}$ that were not extended by PROCESS_TRIE above, output
    the corresponding local correlation pattern if the path to $L$ is at least of length $\gamma$.
10: return $\mathcal{T}_t$
```

Algorithm 3

```
1:  PROCESS_TRIE( $N$, $S$ )
2:  $\tilde{N} \leftarrow$ new trie node
3:  for $N_c \in C(N)$ do
4:      $N'_c \leftarrow$ PROCESS_TRIE( $N_c$, $S$ )
5:      if $|\theta(N'_c)| \geq \sigma$ then
6:          $C(\tilde{N}) \leftarrow C(\tilde{N}) \cup N'_c$
7:      else
8:          $\theta_l(\tilde{N}) \leftarrow \theta_l(\tilde{N}) \cup \theta(N'_c)$
9:      end if
10: end for
11: $\theta_l(\tilde{N}) \leftarrow \theta_l(\tilde{N}) \cup \{\theta_l(N) \cap S\}$
12: if $N$ is a leaf and $|\theta(\tilde{N})| \geq \sigma$ then
13:     mark $N$ as extended
14: end if
15: return $\tilde{N}$
```

this, since on line 11 we compute the intersection of the local support $\theta_l(N)$ and S. But since each sequence identifier appears only at one node N (most of them appear at the leafs), the total cost of line 11 over all recursive calls of PROCESS_TRIE is $O(n)$. Also, implementing lines 5 and 8 in time $O(1)$ requires some additional bookkeeping that is not shown in the pseudocode of Algorithm 3. Essentially when PROCESS_TRIE returns it must in addition to $\tilde{N}$ also return a list of sequence identifiers that can be found in the subtrie below $\tilde{N}$. These lists must be constructed in such a way, that computing the union on line 8 is simply a matter of concatenation. Here we also use the property that a sequence identifier can appear in the trie only once.

Hence, the overall complexity of LCP1 is $O(|\Sigma|n)$, since we must traverse $\mathcal{T}$ once for each $a \in \Sigma$. Note that simply reading the next symbol from each sequence is an $O(n)$ operation.

		σ				
n	$\|\Sigma\|$	2	4	8	16	32
50	2	13661	22471	27777	30959	31347
	4	14925	19920	26385	26455	26385
	8	13280	18867	19157	19193	19120
	16	9442	10893	10905	10869	10881
	32	5599	5837	5743	5685	5698
200	2	3367	6906	9469	10952	11918
	4	4012	6925	8650	9765	10111
	8	3376	5414	6910	6973	7032
	16	2461	3996	4196	4206	4171
	32	1674	2079	2076	2071	2067
500	2	874	2464	3910	4580	5055
	4	1164	2731	3442	3987	4130
	8	1033	2122	2525	3003	3012
	16	751	1417	1800	1803	1804
	32	569	918	936	933	934
1000	2	377	903	1542	1975	2228
	4	515	1018	1468	1670	1960
	8	476	886	1110	1214	1419
	16	373	554	809	845	849
	32	249	439	469	469	468

		σ				
n	$\|\Sigma\|$	2	4	8	16	32
50	2	35.6	11.6	4.6	1.8	1.5
	4	17.9	5.8	2.6	2.5	2.5
	8	11.7	4.7	4.5	4.5	4.5
	16	10.4	8.2	8.2	8.2	8.2
	32	13.8	13.2	13.3	13.2	13.2
200	2	143.5	47.5	20.0	8.9	3.8
	4	72.1	23.7	10.4	4.0	2.5
	8	48.6	16.9	5.0	4.5	4.5
	16	34.7	9.5	8.5	8.5	8.5
	32	25.3	16.5	16.5	16.5	16.5
500	2	359.7	119.6	51.0	23.6	11.3
	4	180.4	58.9	27.3	10.7	6.3
	8	118.0	39.5	21.2	4.7	4.5
	16	98.0	25.7	8.6	8.5	8.5
	32	62.8	17.3	16.5	16.5	16.5
1000	2	720.4	240.1	102.5	47.8	23.0
	4	360.7	121.9	48.0	26.4	10.5
	8	245.8	71.1	36.3	20.4	4.5
	16	179.8	78.9	14.3	8.5	8.5
	32	155.0	25.5	16.5	16.5	16.5

Table 1: *Left:* Average number of steps processed per second by LCP1 for different combinations of parameter values with random inputs. *Right:* Average size of the candidate trie with different combinations of parameters for randomly generated inputs.

4 Experiments

4.1 Performance of LCP1

In this section we study the behavior of LCP1 with different parameters of the input. We are interested in how the size (number of nodes) of the candidate trie $\mathcal{T}$ behaves and how the number of found patterns varies for different values of σ and γ. The implementation used is written in Java, and can be obtained from the web site of the author[1]. The experiments are run on a 2.2GHz Intel CPU.

4.1.1 Artificial data

Artificial data is generated using a model with n sequences that each output a uniformly at random chosen symbol of Σ independent of each other at every step. We do not plant any patterns into the input, and hence the test indicates only how the algorithm responses to noise. We let $n \in \{50, 200, 500, 1000\}$, $\sigma \in \{2, 4, 8, 16, 32\}$, and $|\Sigma| \in \{2, 4, 8, 16, 32\}$, and run LCP1 for 10000 steps with every combination of n, σ, and $|\Sigma|$. In each case we measure the running time and average size of the candidate trie.

Results for both are shown in Table 1. On the left of Table 1 we show the average number of time steps that LCP1 processes in one second for various parameter combinations. Clearly the algorithm becomes faster when σ is increased, since the size of $\mathcal{T}_i$ dramatically decreases due to the n/σ upper bound on the number of candidates. Another observation

[1] `http://www.cis.hut.fi/aukkonen`

σ	4	6	8	10	12
avg. size of $\mathcal{T}_i$	15.69	9.12	6.39	4.97	4.14

Table 2: Average size of $\mathcal{T}_i$ for different σ in the Dow Jones data ($n = 30$, $|\Sigma| = 2$).

	$\sigma = 4$	$\sigma = 6$	$\sigma = 8$	$\sigma = 10$	$\sigma = 12$
$\gamma = 4$	8510	3712	1638	650	286
$\gamma = 6$	2970	738	201	56	11
$\gamma = 8$	766	119	19	4	0
$\gamma = 10$	159	17	0	0	0
$\gamma = 12$	35	0	0	0	0

Table 3: Number of maximal local correlation patterns found in the Dow Jones data with different combinations of σ and γ.

is that larger alphabets are slower to process despite the fact that the average size of $\mathcal{T}_i$ decreases when $|\Sigma|$ increases. This is also obvious, as the complexity of the algorithm is $O(|\Sigma|n)$, because on ever step we traverse the trie once for every $a \in \Sigma$. Finally we note that these numbers represent idealized conditions, since the input is being generated on the fly, and thus no data was read from any device, which is bound to be the bottleneck in many real applications.

4.1.2 Real data

For this experiment we consider a stock market data[2] that contains the daily opening and closing prices of the Dow Jones 30 index between years 1985 and 2003. The index consists of 30 selected companies. We modify the data so that each day is labeled with a $+$ if the price of the stock went up or with a $-$ if the price went down. We recognize that this approach to discretization is not without problems as the magnitude of the variation is hidden, but for the purposes of demonstration we consider it sufficiently accurate. In real applications one might consider using a more sophisticated approach to discretizing the data.

First we investigate how the size of the candidate trie behaves in time. We run our algorithm on the Dow Jones data set using again different values for σ and γ and record the size of $\mathcal{T}_t$ at every t. The average size of the trie is independent of γ, and is shown in Table 2 for different values of σ. We can also study the number of local correlation patterns found. This is shown for the Dow Jones data in Table 3 for different combinations of σ and γ. Obviously the number of patterns found for small parameter values is orders of magnitude larger than for larger ones. Increasing γ has a stronger effect.

4.2 Using local correlation patterns to compare the sequences

In this section we give an example on how to use local correlation patterns for comparing the sequences in the input. Consider the Dow Jones data used in the previous experiment. For each company X, we can look at the set of patterns that contain the sequence corresponding to X in their support. Denote this set by $P(X)$. Given a set of patterns, we can

[2]`http://lib.stat.cmu.edu/datasets/DJ30-1985-2003.zip` (May 15. 2009)

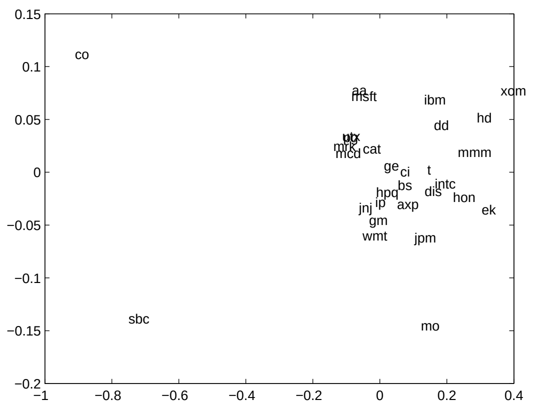

Figure 3: A principal components plot showing ticker symbols of companies in the DJ data. The data is based on the local correlation patterns. We observe that SBC and CO are different from the other companies given the set of local correlation patterns they belong to.

describe it by using a feature vector, where the features are some simple characteristics of the patterns. These could be the average length of patterns, the average number of alternations of the symbols in the patterns, the average size of the support, etc. Given all patterns that are found from the DJ data with $\sigma = 4$ and $\gamma = 4$, we compute the aforementioned features based on $P(X)$ for every company X. We obtain a small data set with one row for each company. Figure 3 shows a scatterplot with the 1st and 2nd principal component of this data on the X and Y-axis, respectively. Clearly companies with the ticker symbols CO and SBC differ somehow from the rest. When looking at the actual values of the features, it turns out that these companies tend to belong to supports of patterns that are longer and contain a larger number of alternations between the symbols.

4.3 Using local correlation patterns for classification

In this section we study how local correlation patterns can be used to classify time series. We use data of EEG measurements made available by Henri Begleiter at the Neurodynamics Laboratory at the State University of New York Health Center at Brooklyn. The data can be downloaded from the UCI KDD repository[3]. Originally the data was used in [15].

The data consists of EEG measurements conducted with a number of subjects performing object recognition tasks. Part of the subjects had been diagnosed with a genetic predisposition to alcoholism (group $\mathcal{A}$), while the remaining ones belong to a control group (group $\bar{\mathcal{A}}$). Our aim in this experiment is to study if the local correlation patterns differ between the two groups, and if the patterns can be used to build a classifier for predicting the condition of unseen subjects. We point out that classification of this particular EEG data has been studied successfully in existing literature (see e.g. [5]), and we do not claim that the approach discussed here is superior to existing techniques. The purpose of this experiment is to study the applicability of local correlation patterns for classification in

[3]`http://kdd.ics.uci.edu/databases/eeg/eeg.html`

group	σ	γ	$E[\phi_n]$	$E[\phi_s]$	$E[\phi_l]$	$E[\phi_a]$
$\mathcal{A}$	6	6	0.20	0.02	-0.15	0.06
$\bar{\mathcal{A}}$	6	6	-0.34	-0.04	0.27	-0.10
$\mathcal{A}$	12	8	-0.17	0.01	-0.15	0.10
$\bar{\mathcal{A}}$	12	8	0.30	-0.01	0.26	-0.18
$\mathcal{A}$	24	6	-0.21	-0.02	-0.14	0.06
$\bar{\mathcal{A}}$	24	6	0.37	0.03	0.24	-0.11

Table 4: Group specific means for different features computed from patterns found in an EEG data set with different values of σ and γ. We observe that the group of alcoholic subjects differs from the control group in every case.

general.

There are 123 subjects in total, each of whom has completed 120 measurements (up to a small number of exceptions). Number of subjects in groups $\mathcal{A}$ and $\bar{\mathcal{A}}$ is 77 and 45, respectively. Each measurement contains 256 samples on 64 channels that represent one second of activity in the brain while the subject was exposed to a visual stimulus. The EEG output consists of floating point values which we discretize to obtain sequences over the alphabet $\{-,+\}$, where a $-$ $(+)$ means that the measurement value decreased (increased) from the previous step.

First we compare the sets of patterns found, and see if there is any difference between the two groups of subjects. To this end we compute the set P_{ij} of local correlation patterns with different values of σ and γ for every measurement j of every subject i. Using the patterns in P_{ij} we compute the feature vector f_{ij}. The features we use are the number of patterns in P_{ij} denoted ϕ_n, average length of the patterns in P_{ij} denoted ϕ_l, average size of the support of the patterns in P_{ij} denoted ϕ_s, and average number of alternations between a $+$ and $-$ in the patterns denoted ϕ_a. All features are normalized to zero mean and unit variance.

We compare the within group means of each feature in Table 4. There are a number of things worth noting. First, the average number of patterns (ϕ_n) is larger in group $\bar{\mathcal{A}}$ with larger values of σ. Second, the average length of a pattern (ϕ_l) is always larger in group $\bar{\mathcal{A}}$, while the average number of alternations between a $+$ and a $-$ (ϕ_a) is always slightly larger in group $\mathcal{A}$. This result indicates that the sets of patterns found in measurements of alcoholic subjects differ from the sets of patterns found in measurements of subjects belonging to the control group.

As a next step we build a simple nearest mean classifier using these features. Let $\mathcal{A}_{\text{train}}$ and $\bar{\mathcal{A}}_{\text{train}}$ be the sets of alcoholic and control subjects used for training. The training data consists of the feature vectors f_{ij} for all j and for all $i \in \{\mathcal{A}_{\text{train}} \cup \bar{\mathcal{A}}_{\text{train}}\}$. First we normalize the features in the training data to zero mean and unit variance. Then we compute the group specific means $E[\mathcal{A}_{\text{train}}]$ and $E[\bar{\mathcal{A}}_{\text{train}}]$. A new subject k is classified by considering the feature vectors f_{kj} separately for each j. The classifier assigns subject k to group $\mathcal{A}$ if the number of vectors f_{kj} that are closer to $E[\mathcal{A}_{\text{train}}]$ is larger than the number of vectors f_{kj} that are closer to $E[\bar{\mathcal{A}}_{\text{train}}]$, and to the class $\bar{\mathcal{A}}$ otherwise.

We test the approach using 1-fold cross-validation, i.e., one subject is classified at a time with a training set that contains all other subjects but not the one we are classifying. This experiment is repeated using patterns computed with the same values of σ and γ used above in Table 4. In every case we get the following confusion matrix:

	$\mathcal{A}$	$\bar{\mathcal{A}}$
$\mathcal{A}$	77	0
$\bar{\mathcal{A}}$	1	44

All alcoholic subjects are classified correctly, and one control subject is classified incorrectly to group $\mathcal{A}$. Moreover, the parameters used when mining the patterns seem to have a negligible effect on the classifier. This result is not surprising considering that in [5] it is reported that alcoholic and control subjects differ significantly in a number of traditional features used in EEG analysis. However, our experiments show that simple and general, *non-application specific features* based on local correlation patterns can also be very effective for classification.

5 Related work

Mining patterns in streams and time series is a well studied topic. However, to the best of our knowledge, the local correlation patterns introduced here have not appeared previously in literature. However, methods for mining several other kinds of patterns from time series and data streams have been proposed.

In [2] an algorithm is presented for finding rules in streams. These are frequently occurring patterns of the form "if A occurs then B occurs within a certain time". A related approach is that of finding "motifs" [8], where one seeks parts of the time series that occur repeatedly. In some applications it is not enough to have a pattern repeat itself, but it must do so in a certain period. Such patterns are mined for example in [4, 14]. Yet another pattern class is defined by so called surprising patterns [7] that are parts of a stream that are unexpected given some previously observed history. Another related line of research is about mining frequent patterns in streams. Examples include frequent itemsets [9, 3], sequential patterns [11], and trees [1].

A common characteristic of these examples is that they are concerned with finding repeating occurrences of the pattern within a window of the stream. In this work we want to find *simultaneous* occurrences of the pattern in multiple parallel streams.

6 Conclusion

We have introduced local correlation patterns as a method for analyzing multidimensional time series. We model such time series as sets of sequences over some finite alphabet. A local correlation pattern is defined as a point t in time together with a string of symbols from the alphabet. The pattern tells that starting from time t a subset of the sequences all simultaneously output the specified string. In practice we want to find patterns where the sequence is at least of length γ and the subset of sequences is of size at least σ. Moreover, we are only interested in maximal local correlation patterns, i.e., patterns where the sequence is not the prefix nor suffix of any other local correlation pattern.

We proposed an algorithm that mines maximal local correlation patterns from a set of sequences in an online setting. The algorithm works by maintaining a trie of candidate patterns that is updated at every step when new symbols are read from the sequences. The complexity of our algorithm is $O(|\Sigma|n)$ for each timestep, where n is the number of sequences and $|\Sigma|$ the size of the alphabet. New patterns are output by the algorithm as soon as they are found.

We conducted experiments that show the algorithm is fast. A simple implementation running on a regular PC can process up to thousands of steps per second. Even for small support thresholds and a large number of sequences (say, $n = 1000$), the algorithm is capable of processing over a hundred steps per second. We also show that the local correlation patterns can be used for classifying EEG time series. In our experiment a simple nearest mean classifier based on the patterns had nearly 100 percent accuracy.

A problem with some practical applications is that we cannot assume the sequences to be perfectly aligned. In such situations we must allow small variations in the starting time of the pattern in a sequence. That is, a sequence s would support the pattern (i, p) if p occurs as a substring in s at the position $j = i \pm \delta$. Another question is how to incorporate wildcards into the pattern string, or how to allow (a small number of) mismatches in the supporting sequences.

References

[1] T. Asai, H. Arimura, K. Abe, S. Kawasoe, and S. Arikawa. Online algorithms for mining semi-structured data stream. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, page 27, 2002.

[2] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 16–22, 1998.

[3] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. *Data Mining: Next Generation Challenges and Future Directions*, chapter Mining Frequent Patterns in Data Streams at Multiple Granularities. MIT Press, 2004.

[4] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 106–115, 1999.

[5] N. Kannathal, U. Acharya, C. Lim, and P. Sadasivan. Characterization of eeg – a comparative study. *Computer Methods and Programs in Biomedicine*, 80(1):17–23, 2005.

[6] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[7] E. Keogh, S. Leonardi, and B. Chiu. Finding surprising patterns in a time series database in linear time and space. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 550–556, 2002.

[8] J. Lin, E. Keogh, S. Lonardi, and P. Patel. Finding motifs in time series. In *Proceedings of the Second Workshop on Temporal Data Mining*, 2002.

[9] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357, 2002.

[10] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 23(2):262–272, 1976.

[11] C. Raïssi, P. Poncelet, and M. Teisseire. Speed: Mining maximal sequential patterns over data streams. In *Proceedings of the 3rd International IEEE Conference on Intelligent Systems*, pages 546–552, 2006.

[12] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[13] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[14] J. Yang, W. Wang, and P. S. Yu. Mining asynchronous periodic patterns in time series data. *IEEE Transactions on Knowledge Engineering*, 15(3):613–628, 2003.

[15] X. L. Zhang, H. Begleiter, B. Porjesz, W. Wang, and A. Litke. Event related potentials during object recognition tasks. *Brain Research Bulletin*, 38(6):531–538, 1995.