VOUW: Geometric Pattern Mining using the MDL Principle

Micky Faas and Matthijs van Leeuwen

Leiden Institute for Advances Computer Science

Abstract

- MDL is a principle that utilizes compression as a means of describing data
- There exist many MDL-based algorithms for a variety of problems
- Few (no?) solutions exist to mine grid-like data based on MDL
- VOUW is a novel approach to discover patterns and structural relations in 2D, discrete datasets
- We propose both a theoretical framework as well as a complete, optimized implementation

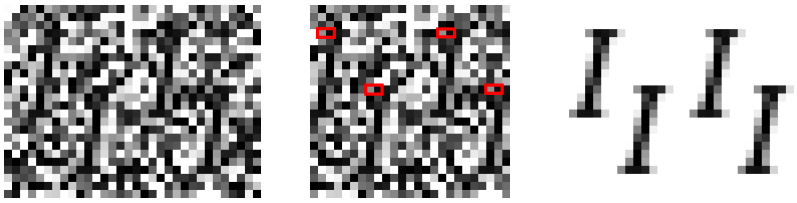
1 Introduction

Frequent pattern mining is the subfield of data mining that aims to find and extract recurring substructures as a means of data analytics. These patterns can be any kind of datastructure, for example item sets (transactions), graphs (networks) and sequences (in time). So far, little research has been done in applying pattern mining to raster-based, tabular data or matrices. Patterns in these types of data can be more complex as elements cannot only coincide, but their geometric relation is also important. As the first contribution of this paper, we introduce exactly this problem, that we will call **geometric pattern mining** and formally introduce notation and theoretical constructions for this problem. Potential applications include analysis of (satellite) imagery, texture recognition and clustering of matrices.

Geometric Pattern Mining Geometric pattern mining is the problem of finding recurring local structure or **patterns** in matrices. It is different from graph mining, as a matrix is more rigid and each element has a fixed degree of connectedness/adjacency. It is also unrelated to linear algebra, other then using the term 'matrix' and a comparable style of notation. Furthermore in this context, matrix elements can only be discrete, rows and columns have a fixed ordering and the semantics of a value is position-independent. Although the concept applies to any number of dimensions, we will limit the scope to two dimensional data from here on.

The problem of geometric pattern mining can roughly be divided into three classes. The first class consists of three subclasses: finding identical patterns in

(1a) an otherwise empty (sparse) matrix, (1b) differently distributed noise and (1c) similarly distributed noise. The second class contains the same subclasses but adds that the patterns can also be overlaid with noise and are therefore not identical. The third class is also a continuation of the first class and requires that the patterns are identical after some optional transformation (such as mirror, inverse, rotate, etc.). These classes also represent an increasing difficulty level and serves as a rough benchmark for the performance of an algorithm.



(a) 32×24 noise 'matrix' (b) Pair $(146, 11)$ (c) Pattern 'I' occurs four times.

Fig. 1: Brief example of geometric pattern mining

Let us demonstrate the problem of geometric pattern mining by giving a brief example. Figure 6a shows a grayscale image that we assume is random 'white noise'. For convenience, we will interpret the image as a 32×24 matrix with values on the interval $[0; 255]$. If we were to look at all horizontal pairs of elements, we would find that the pair $(146, 11)$ is, among others, statistically more prevalent than the initial assumption of random would suggest. Figure 6b highlights the locations where these pairs occur: we have just discovered the first repeating structure in our dataset! If we would continue to try all combinations of elements that 'stand out' from the background noise, Figure 6d shows that we will eventually find that the matrix contains four copies of the letter 'I' set in 16 point Garamond Italic.

The 35 elements that make up a single 'I' in the example, are said to form a *pattern*. We can use this pattern to describe the matrix in an other way than '768 unrelated values'. For example, we could describe it as 628 unrelated values plus pattern 'I' at locations $(5, 4), (11, 11), (20, 3), (25, 10)$, separating the structure from the accidental (noise) data. Since this requires less storage space than before, we have also compressed the original data. See how at the same time we have learned something about the data: we did not know about the 'hidden I's before.

Datamining by Compression In recent years, a class of algorithms utilizing the *Minimum Description Length (MDL) principle* [5, 2] have become more and more common in the field of explanatory data analysis. Examples of such approaches include Krimp [7] or more recently Classy [4]. The MDL principle was first described by Rissanen in 1987 [5] as a practical implementation of Kolmogorov Complexity [3]. Central to MDL is the notion that 'learning' can be

thought of as 'finding regularity' and that regularity itself is a property of data that is exploited by *compressing* said data. Therefore by compressing a dataset, we actually learn its structure — how regular it is, where this regularity occurs, what it looks like — at the same time.

The problem that MDL solves first and foremost, is that of *model selection*: given a multitude of explanations (models), select the one that fits the data best. In addition to this, MDL has also been demonstrated to be very effective in materialization of a specific model given the data. In this case, the model is predetermined and we want to find the parameters to fit the data. A similar problem class is solved in pattern mining: here the 'parameters' are the discrete building blocks that make up patterns in the data. We will specifically look at a variant of MDL called two-part MDL. As the second contribution of this paper, we present a geometric pattern mining algorithm that (1) is precise, (2) requires no parameters and (3) is tolerant to noise in the data, based on two-part MDL.

2 Related Work

Krimp [7] is probably one of the first explanatory data mining approaches using MDL and also one of the sources of inspiration for this paper. Since then, many papers on this topic have been published, such as Slim **??** (frequent item set mining, like Krimp), Classy **??** (mining classifiers), *TODO: list more*

The work by Campana et al. [1] also uses matrix-like input data (textures) and develops a similarity measure based on MDL. Their method, however, cannot be used for *explanatory* data analysis as they use a generic image compression algorithm that is essentially a black box.

This is still far from complete

3 Theoretical Framework

We define spatial pattern mining on bounded, discrete and two-dimensional geometric data. We represent this data as an $M \times N$ matrix A whose rows and columns are finite and in a fixed ordering (i.e. reordering rows and columns semantically alters the matrix). Elements $a_{i,j}$, where row i is on $[0; N)$ and column j is on $[0; M)$, holds that $a_{i,j} \in S$, the finite set of symbols occurring in A.

$$A = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & 1 \\ \cdot & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & 1 \\ 1 & \cdot & 1 & \cdot & \cdot \\ \cdot & 1 & \cdot & 1 & \cdot \end{bmatrix}, \; I = \begin{bmatrix} X & \cdot & \cdot & \cdot & Y \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ X & \cdot & \cdot & X & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ X & \cdot & X & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}, \; H = \{X = \begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}, Y = \begin{bmatrix} 1 \end{bmatrix}\}$$

Fig. 2: Example decomposition of A into instantiation $\bar{H}$ and patterns X, Y

MDL tells us to search for the shortest (optimal) description of A. This description then tells us everything there is to know about A in the most succinct way possible. The principle behind MDL is that we can approximate an optimal description through the compression of the original data. This optimal description, let us call it A', is only optimal if we can unambiguously reconstruct A from it and nothing more — the compression is both minimal and 'lossless'. Intuitively compression means that we would like to have a way to just define A using as few building blocks as possible. We illustrate this using the example in Figure 2. Given the matrix A we try to decompose it in recurring substructures which we will call **patterns**, denoted X and Y in the example. At any given time, the set of patterns which we call the **model** of a A, is denoted H_A. Given only these patterns, A cannot be reconstructed because we need a mapping from the model H_A back to A. This mapping represents in what MDL calls the **matrix A given the model H_A**. In statistics this is often the structural versus the accidental information, but in this context we think of it merely as 'instructions' of how to reconstruct A. Let us call the set of all instructions required to rebuild A from H_A the **instantiation** of H_A. It is denoted by I_A in the example. Notice how this instantiation essentially tells us where in A each pattern from H_A was originally located. The result is a notation that allows us to express matrix A as if decomposed into sets of local and global spatial information.

Now that we have this top-down concept of the decomposition of A, we will continue to describe its constituents in more detail.

3.1 Patterns and Instances

We can define a pattern as a submatrix X of the original matrix A. This submatrix is not necessarily complete (elements may be $\cdot$, the empty element), which gives us the ability to precisely cut-out any irregular-shaped part of A. We additionally require the elements of X to be adjacent (horizontal, vertical or diagonal) to at least one non-empty element. While this limits the amount of possible patterns somewhat, it will later on also reduce the computational effort dramatically. We will now define a pattern to be the smallest submatrix to completely contain all elements of X.

From this definition we see that the dimensions $M_X \times N_X$ give essentially the smallest rectangle around X (the *bounding box)*. As a more useful measure we therefore also define the cardinality $|X|$ of X as the number of non-empty elements. We call a pattern X with $|X| = 1$ a **singleton pattern**, i.e. a pattern containing exactly one element of A.

One element in each pattern is given the special function of **pivot**: $pivot(X)$ of pattern X is the first non-empty element of X in the first non-empty column of the first (non-empty) row of X. A pivot can be thought of as a fixed point in a pattern X which we can use to position its elements in relation to A. The precise translation we apply to a particular pattern we call an **offset**. An offset is a tuple $q = (i, j)$ that is on the same domain as an index in A. We realize this translation by placing all elements of X on an empty $M \times X$ size matrix in

such way that the pivot element is at (i, j). We formalize this concept with the **instantiation operator** $\otimes$:

▷ *In the context of an $M \times N$ matrix A, we define the* ***instance*** *$X \otimes q$ as the incomplete $M \times N$ matrix containing all elements of X such that* $\text{pivot}(X)$ *is at index (i, j) and the distances between all original elements are preserved. The resulting matrix contains no additional non-empty elements.*

Obviously this does not yield a valid result for an arbitrary offset (i, j). We want to limit ourselves to the space of pattern instances that are actually valid in relation to matrix A. Therefore two simple constraints are needed: (1) an instance must be **well-defined**: *placing* $\text{pivot}(X)$ *is at index (i, j) results in a $M \times N$ matrix that contains all elements of X, and (2) elements of instances cannot overlap, meaning each element of A should be described at most once.* This allows for a description that is both unambiguous and minimal.

▷ *Two pattern instances $X \otimes q$ and $Y \otimes r$, with $q \neq r$ are* ***non-overlapping*** *if $|(X \otimes q) + (Y \otimes r)| = |X| + |Y|$.*

From here on we will use the same letter in lower case to denote an arbitrary instance of a pattern, e.g. $x = X \otimes q$ when the exact value of q is unimportant.

Given a set of patterns H over A, we would like to have a set of 'instructions' of where instances of each pattern should be positioned in order to obtain A. When looking at the example in Figure 2, this mapping has the shape of an $M \times N$ matrix.

▷ *Given the set of patterns H, the* ***instantiation (matrix)*** *I is an incomplete $M \times N$ matrix with the set of possible elements being H, i.e. $I_{i,j} \in H$ for all (i, j). For all non-empty elements $I_{i,j}$ it holds that $I_{i,j} \otimes (i, j)$ is an instance of $I_{i,j}$ in A that is not non-overlapping with any other instance $I_{i',j'} \otimes (i', j')$.*

The above definition creates the interesting proposition that the offset to each instance is unique. Given that a pattern's pivot is placed exactly at one offset and that instances must be non-overlapping, makes this indeed believable. Later when we inductively define the set I of all instantiation matrices, this will be shown to be true more formally.

3.2 The Problem and its Solution Space

Constructing Patterns Patterns can be joined to create increasingly large patterns and, as we will later see, this concept forms the basis of the search algorithm. However, two patterns could be joined in any different number of ways. We can define the exact way two patterns should be joined by enumerating the distance of their respective pivots. We use pattern instances as an intermediate step.

Recall that instances are simply patterns projected on an $M \times N$ matrix, containing the same elements as the patterns at their original distances. This makes $\otimes$ trivially reversible by removing all completely empty rows and columns:

▷ *Let $X \otimes q$ be an instance of X, then by definition we say that $\oslash(X \otimes q) = X$.*

We can now define a sum on two instances to get a new instance that combines both operands. Even though the result is not a pattern, we can always obtain it by using $\oslash$. The example in Figure 3 illustrates this process. We start by instantiating X and Y with offsets $(1,0)$ and $(1,1)$ respectively. This yields the non-overlapping x and y, which we simply add up to obtain z. The new pattern contained in z can be easily identified by removing the top empty row. We formally describe this mechanism in Theorem 1. As we will see later, this will become the fundamental operation of our algorithm.

$$X = \begin{bmatrix} 1 & \cdot \\ \cdot & 1 \end{bmatrix}, Y = \begin{bmatrix} 1 \end{bmatrix},\ x = X \otimes (1,0) = \begin{bmatrix} \cdot & \cdot \\ 1 & \cdot \\ \cdot & 1 \end{bmatrix},\ y = Y \otimes (1,1) = \begin{bmatrix} \cdot & \cdot \\ \cdot & 1 \\ \cdot & \cdot \end{bmatrix},$$

$$x + y = \begin{bmatrix} \cdot & \cdot \\ 1 & 1 \\ \cdot & 1 \end{bmatrix},\ Z = \oslash(x+y) = \begin{bmatrix} 1 & 1 \\ \cdot & 1 \end{bmatrix}$$

Fig. 3: Example of Theorem 1. The 2×3 input matrix is not shown.

Theorem 1. *Given two non-overlapping instances $x = X \otimes q$ and $y = Y \otimes r$, the sum of the matrices $x + y$ is another instance. We observe that pattern $Z = \oslash(x + y)$ such that $x + y = Z \otimes q$.*

Notice that this sum has the limitation that two instances can only be summed if they do not overlap. While this is a serious limitation, we will show in the next subsection that it is not of any practical relevance.

The Sets $\mathcal{H}_A$ and $\mathcal{I}_A$ We define the **model class** $\mathcal{H}$, the set of all possible models for all possible inputs. Without any prior knowledge, this is the search space of our algorithm. We will first look at a more bounded subset of $\mathcal{H}$, namely the set $\mathcal{H}_A$ of all possible models for A, and its counterpart, the set $\mathcal{I}_A$ of all possible instantiations to these models. We will also take H_A^0 to be the model with only singleton patterns (patterns of length 1). As singletons are just individual elements of A, we can simply say that $H_A^0 = S$. The instantiation matrix corresponding to H_A^0 is denoted I_A^0. Given that each element of this matrix must correspond to exactly one element of A in H_A^0, we see that each $I_{i,j} = a_{i,j}$ and so I_A^0 is equal to A.

Using H_A^0 and I_A^0 as base cases we can now inductively define the set $\mathcal{I}_A$ of all instantiations of all models over A:
Base case: $I_A^0 \in \mathcal{I}_A$
By induction: If I is in $\mathcal{I}_A$ then take any pair $I_{i,j}, I_{k,l} \in I$ such that $(i,j) \leq (k,l)$ in lexicographical order. Then the set I' is also in $\mathcal{I}_A$, providing I' equals I except

$$I'_{i,j} := \oslash\big(I_{i,j} \otimes (i,j) + I_{k,l} \otimes (k,l)\big)$$
$$I'_{k,l} := \cdot$$

In this definition of I_A^0 we inductively replace two instances with their sum. Indeed we can add any two instances together in any order and eventually this results in just one big instance that is equal to A. The elegance in this is that, by this inductive definition, the instances never overlap and thus their sum is always a valid instance on A. Note that when we take two elements $I_{i,j}, I_{k,l} \in I$ we force $(i,j) \leq (k,l)$ to be in lexicographical order, not only to eliminate different routes to the same instance matrix, but also such that the pivot of the new pattern coincides with $I_{i,j}$. We can then leave $I_{k,l}$ empty.

The construction of instantiation matrices also implicitly defines the corresponding models. While this may seem odd — defining models for instantiations instead of the other way around — note that there is no unambiguous way to find an instantiation matrix for a given model. Instead we find the following trivial definition by applying the inductive construction rule above:

$$\mathcal{H}_A = \big\{\{\oslash(I) \mid I \in I\} \;\big|\; I \in \mathcal{I}_A\big\}.$$

So for any instantiation $I \in \mathcal{I}_A$ there is a corresponding set in $\mathcal{H}_A$ of all patterns that occur in I. This results in an interesting symbiosis between model and instantiation: increasing the complexity of one decreases that of the other. When plotting this construction a tightly connected lattice appears such as that of Figure 4.

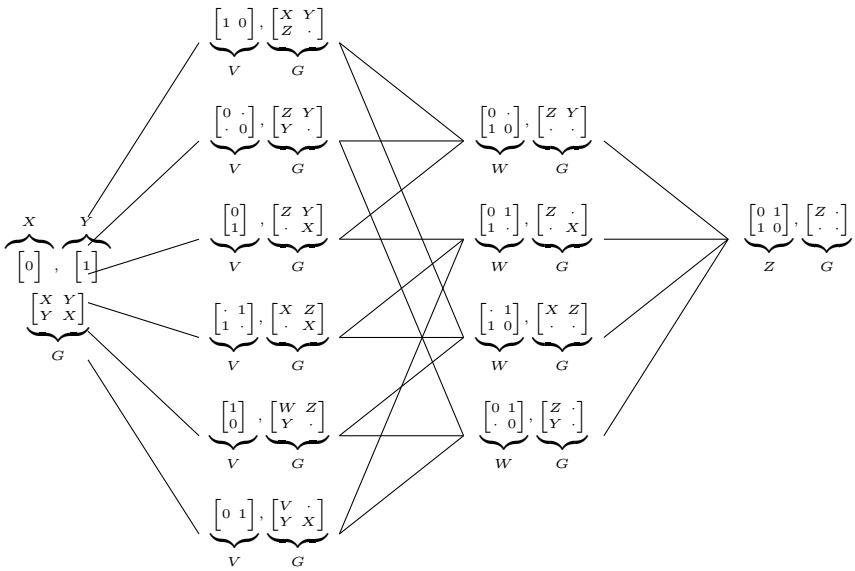


Fig. 4: The model space lattice for a 2×2 Boolean matrix. Here X and Y are the initial singleton patterns and G is the instance matrix. V and W are intermediate patterns and Z is the final, completely over fit, pattern.

3.3 Encoding Models and Instances

From all parametrized models in $\mathcal{H}_A$ we want to select (approximate) the model that describes A best. We use two-part MDL to quantify how well a given model and instantiation matrix fit A. Two-part MDL tells us to minimize the sum of $L_1(H_A) + L_2(A|H_A)$, two functions that give the length of the model and the length of 'the data given the model', respectively. In this context, the model is the set of patterns H_A and the data given the model is the accidental information needed to reconstruct the data from H_A — which in this case is the instantiation matrix I_A.

In order to give said length functions, we need to decide on a way to encode H and I first. This encoding is of great influence on the length functions and therefore on the ability to accurately quantify the fit of a given H and I to A. The decision for this encoding is obviously prone to bias, which is one of the few disadvantages of two-part MDL. Fortunately, practice shows that good results can be had once certain conditions are met: (1) all data is encoded (lossless) and (2) the encoding is as concise as possible (nothing but the data is encoded). Based on these conditions we give length functions for pattern sets and instance matrices, but we do not actually need to encode them.

The fictional encoder sequentially sends each symbol in the datastream (either pattern set or instance matrix) to the 'decoder' using a code word. Information theory tells us that we optimal length of a code word is given by $-\log(p)$, where p is the exact probability that the code word occurs in the output. We therefore need not compute the actual code words, just their probabilities. For this to work, both the encoder and hypothetical decoder must know either the exact probability distribution or agree upon an approximation beforehand. Such approximation is often called a **prior** and is used to fix 'prior knowledge' that does not have to be encoded explicitly.

To encode the instance matrix we will use the **prequential plug-in code** [**?**]. The prequential plug-in code is defined for sequences of one item at a time and updates the probability of each item as it is encoded, such that the probability need not be known in advance. It has the favorable property of being asymptotically equal to the optimal code for large sequences. Say we want to encode all elements $I_i \in I$, we define:

$$P_{plugin}(y_i = I_i \mid y^{i-1}) = \frac{|\{y \in y^{i-1} \mid y = I_i\}| + \epsilon}{\sum_{X \in H} |\{y \in y^{i-1} \mid y = X\}| + \epsilon}$$

Here y_i is the i-th element to be encoded and y^{i-1} is the sequence of elements encoded so far. We initialize the base case (no element has been sent yet) with a pseudocount ϵ, which gives $P_{plugin}(y_1 = I \mid y^0) = \frac{\epsilon}{\epsilon|H|}$. We pick $\epsilon = 0.5$ as it is used generally with good results.

Let us adapt this principle to the problem of encoding patterns. The first step here is to determine the probability that each unique element (instance of a pattern) in I occurs.

▷ *Given a set of instances I, we define* $\text{usage}(X) = |\{I_i \in I \mid I_i = X\}|$.

From this definition we see that the **usage** of a pattern is a sum of how often it occurs as an instance. We can use this function to simplify things a little by realizing that we actually know the precise number of instances per pattern on the side of the decoder, but not as the decoder. This information can be used to slightly rephrase Definition 3.3 to be able to encode items in arbitrary order. This produces the length function of the instance matrix I as follows:

$$\begin{aligned} L_{pp}(I \mid P_{plugin}) &= \sum_{i=1}^{|I|} -\log \frac{|\{y \in y^{i-1} \mid y = I_i\}| + \epsilon}{\sum_{X \in H} |\{y \in y^{i-1} \mid y = X\}| + \epsilon} \\ &= -\log \frac{\prod^{X_i \in H} \prod_{j=0}^{\text{usage}(X_i)} j + \epsilon}{\prod_{j=0}^{|I|-1} j + \epsilon|H|} \\ &= -\sum_{X_i \in h}^{|H|} \left[\log \frac{\Gamma(\text{usage}(X_i) + \epsilon)}{\Gamma(\epsilon)}\right] + \log \frac{\Gamma(|I| + \epsilon|H|)}{\Gamma(\epsilon|H|)} \end{aligned}$$

The length function for incomplete matrices. To losslessly encode A' we have to encode both H and I individually. Recall that both instances and patterns are both matrices. It is therefore tempting to utilize the same length function for both. Empirical evidence has shown that this is not a good idea though and the main reason for this is the fact that prequential plug-in code behaves different for small sequences (patterns) than it does for large sequences (the instance matrix). Furthermore, we do not consider certain values such as the size of the instance matrix because it is constant, however, the size of each individual pattern is not. We therefore have to construct a different length function for each type of matrix. These are listed in Table 1.

Table 1: Length computation for the different classes of matrices. The total length is the sum of the listed terms.

	Matrix	Bounds	# Elements	Positions	Symbols
$L_p(X)$	Pattern	$\log(MN)$	$L_{\mathbb{N}}(\binom{M_X N_X}{\lvert X\rvert})$		$\log(\lvert S\rvert)$
$L_1(H)$	Model	*N/A*	$L_N(\lvert H\rvert)$	*N/A*	$L_p(X \in H)$
$L_2(I)$	Inst. mat.	*constant*	$\log(MN)$	*implicit*	$L_{pp}(I)$

Each length function has four (optional) terms. First we encode the total size of the matrix. Since we assume MN to be known/constant, we can use this constant to define the distribution $\log(MN)$. This is simply an uniform distribution that encodes an arbitrary index of A with equal lengths for each index. Next we encode the number of elements that are non-empty. Notice how for patterns, this value is encoded together with the third term, namely the positions of the non-empty elements. Because we have encoded $M_X N_X$ in the

first term, we may now use it as a constant. We use it in the binominal function to enumerate the number of ways we can place the non-empty elements ($|X|$) onto a grid of $M_X N_X$. This gives us both *how many* non-empties there are as well as *where* they are. Finally the fourth term is the length of the actual symbols that encode the elements of matrix. In case we encode single elements of A, we simply assume that each unique value in A has an equal possibility of occurring. For the instance matrix, which encodes symbols to patterns, the prequential code is used as demonstrated before. Notice that L_N is the universal prior for integers[**?**] that can be used to encode integers on an arbitrary range.

4 A Search Algorithm

Pattern mining problems often yield vast search spaces and geometric pattern mining is no exception. Since we are already looking for an approximate result (by definition of two-part MDL) it makes sense to use a greedy strategy, a heuristic widely used in many MDL-based approaches [7, 6, 4]. Another decision we make a priori is that we only find patterns that are *contiguous*: containing only elements that are adjacent to at least one other elements in the same pattern. This decision results in a strong focus on local structure while also dramatically reducing the search space.

Given these heuristics, an inductive algorithm is devised that is unsurprisingly similar to the lattice shown in Figure 4: we start with a completely underfit model (the left of the lattice), where there is one instance for each matrix element. On each iteration we want to combine two patterns, resulting in one or more pairs of instances to be merged (one step right in the lattice). We pick the pair of patterns that improve the compression ratio the most and we repeats these steps until no improvements to the compression can be made.

4.1 Finding candidates

The first step of the algorithm is to find the 'best' *candidate* pair of patterns to merge. Candidates are denoted as a tuple (X, Y, δ), where X and Y are patterns and δ the relative offset between them. All possibilities form a vector space that grows easily too large to completely enumerate. Fortunately, we need only pairs of patterns and offsets that actually occur in the instance matrix. This means at each step we can directly enumerate all candidates from the instance matrix and never even look at the original data (remember that we start with an instance matrix that contains exactly the original data).

Still it is not completely trivial to extract candidates from the instance matrix, as one candidate occurs multiple times in 'mirrored' configurations, such as (X, Y, δ) and $(Y, X, -\delta)$, which are equivalent but can still be found separately. Furthermore, we must know which instances can considered as candidates. Due to the restriction of only finding contiguous patterns, many potential candidates cannot be considered by the simple fact that their elements are not adjacent. Lastly there is the problem is self-overlap, which only happens when both patterns

in the tuple are equal, i.e. (X, X, δ). In this case, too many or too few copies may be counted. Think of this by imagining a straight line of five instances of X. There are four unique pairs of two X's, but only two can be merged at the same time, in three different ways.

For each instance $\bar{X}$ we define its *periphery*. The periphery is a set of instances that are positioned in such a way that their union with $\bar{X}$ would give a contiguous pattern. We furthermore split this set into the *anterior*- $\mathrm{ANT}(X)$ and *posterior* $\mathrm{POST}(X)$ peripheries. These contain instances that come before and after $\bar{X}$ in lexicographical order, respectively. This enables us to scan the instance matrix once, in lexicographical order. For each instance $\bar{X}$, we only consider the instances $\mathrm{POST}(\bar{X})$ as candidates. This immediately eliminates possible (mirrored) duplicates.

When considering candidates of the form (X, X, δ), we also compute an *overlap coefficient*. This coefficient c is given by the equation $c = (2w+1)\delta_i + \delta_j + w$, where w represents the width of the bounding box around pattern X. This equation essentially transforms δ into a one-dimensional coordinate space of all possible ways that X could be arranged *after* and *adjacent* to itself. For each instance $\bar{X}_1$ a vector of bits $V(c)$ is used to remember if we have already encountered a candidate (X, X_1, δ) with coefficient c, such that we do not count a candidate (X_1, X, δ) with an equal coefficient. This eliminates the problem of incorrect counting due to self-overlap.

4.2 Gain computation

After the candidate search we have a set of candidates C and their respective usages in the current instance matrix. The next step is to select the candidate that gives the best *gain*: the improvement in compression of the data. For each candidate $c = (X, Y, \delta)$ the gain $\Delta L(A', c)$ is comprised of two parts: (1) the negative gain of adding the union pattern Z to the model H, resulting in H' and (2) the gain of replacing all instances $\bar{X}, \bar{Y}$ with relative offset δ by Z in $\bar{H}$, resulting in $\bar{H}'$. We use the length functions for the model and instance matrix L_1, L_2 to derive an equation for gain:

$$\begin{aligned} \Delta L(c) &= \Big(L_1(H') + L_2(\bar{H}')\Big) - \Big(L_1(H) + L_2(\bar{H})\Big) \\ &= L_0(|H|) - L_0(|H|+1) - L_p(Z) + \Big(L_2(\bar{H}') - L_2(\bar{H})\Big) \end{aligned} \tag{1}$$

As we can see, the terms with L_1 are simplified to $-L_p(Z)$ and the model's length because L_1 is simply a summation of individual pattern lengths. The equation of L_2 requires the recomputation of the entire instance matrix' length, which is expensive considering we need to perform it for *every candidate*, *every iteration*. However, we can rework the function L_{pp} in Equation (**??**) by observing that we can isolate the logarithms and generalize them into:

$$\log_G(a, b) = \log \frac{\Gamma(a + b\epsilon)}{\Gamma(b\epsilon)} = \log \Gamma(a + b\epsilon) - \log \Gamma(b\epsilon) \tag{2}$$

Algorithm 1 Candidate search

Input: $\bar{H}$
Output: C, $usage()$

for all $\bar{X} \in \bar{H}$ **do**
  for all $\bar{Y} \in \text{POST}(\bar{X})$ **do**
    $X \leftarrow \ominus(\bar{x}_i),\ Y \leftarrow \ominus(\bar{y})$
    $\delta \leftarrow index(\bar{y}) - index(\bar{y})$
    if $\bar{X} = \bar{Y}$ **then**
      if $V(c) = 1$ **then continue**
      $V(c) \leftarrow 1$
    end if
    $C \cup (X, Y, \delta)$
    increment $usage(X, Y, \delta)$ by 1
  end for
end for

Algorithm 2 Baseline VOUW

Input: A
Output: A'

repeat
  $C \leftarrow$ find candidates
  $C_{best} = (X, Y, \delta) \in C : \forall_{c \in C}\ \texttt{gain}(c) \leq \texttt{gain}(C_{best})$ ▷ Select best candidate
  $\Delta L_{best} = \texttt{gain}(C_{best})$
  if $\Delta L_{best} > 0$ **then**
    $Z \leftarrow \ominus(X \oplus (0,0) + (Y \oplus \delta))$
    $h \leftarrow h \cup \{Z\}$ ▷ Add the union pattern to the model
    for all $\bar{x}_i \in \bar{H} \mid \ominus(\bar{x}_i) = X_{c_{best}}$ **do**
      for all $\bar{y} \in \texttt{adjacent}(\bar{x}_i) \mid \bar{y} = Y_{c_{best}}$ **do**
        $\bar{h} := Z$
        $\bar{y} := \cdot$
      end for
    end for
  end if
until $\Delta L < 0$

Which can be used to rework the second part of Equation (1) in such way that the gain equation can be computed in constant time complexity.

$$\begin{aligned} L_2(\bar{H}') - L_2(\bar{H}) = &\log_G(U(X), 1) + \log_G(U(Y), 1) \\ &- \log_G(U(X) - U(Z), 1) - \log_G(U(Y) - U(Z), 1) \\ &- \log_G(U(Z), 1) + \log_G(|\bar{H}|, |H|) - \log_G(|\bar{H}'|, |H'|) \end{aligned} \tag{3}$$

Notice that in some cases the usages of X and Y equal that of Z, which means additional gain is created by removing X and Y from the model.

4.3 Mining patterns

Finding candidates and computing gain for these candidates, is the first part of the algorithm. In the second part we select the candidate (X, Y, δ) with the best gain and merge X and Y to form Z, as explained in Section 3.2. Instances $\bar{X}$ and $\bar{Y}$ with offset δ must now be replaced by instances of Z. First the instance matrix $\bar{H}$ is linearly traversed to find all occurrences of $\bar{X}$ and $\bar{Y}$ with offset δ. Recall that we constructed candidate (X, Y, δ) by looking in the posterior periphery of all $\bar{X}$ to find Y and δ, which means that Y always comes after X in lexicographical

order. The pivot of a pattern is the first element in lexicographical order, therefore $\text{pivot}(Z) = \text{pivot}(X)$. This means that we can replace all matching $\bar{X}$ with an instance of Z and all matching $\bar{Y}$ with $\cdot$.

This concludes the baseline version of the algorithm, which is listed in Algorithm 2.

4.4 Local search

Given that some pattern X is found in a given matrix, the baseline algorithm could have arrived to X in different ways. Exploring these combinatorics can tell us how efficiently the algorithm arrives at X. By definition we know that the fundamental operation is to combine exactly two patterns into a new pattern on each step. Given this, the number of steps in which X can be constructed lies between $\log_2 |X|$ and $|X| - 1$.

To improve efficiency on large patterns without sacrificing the objectivity of the original heuristics, the algorithm is augmented with an additional local search. We call this local search *flood fill* because it is similar to the image algorithm with the same name. It is a result of the observation that the algorithm generates a large pattern X by adding small elements to a incrementally growing pattern, resulting in a behaviour that approaches $|X| - 1$ steps. Instead of taking all $|X| - 1$ steps to arrive at X, we can try to predict which elements will be added to X and merge them directly. Given that we selected candidate (X, Y, δ) and merged X and Y into Z, now for all m resulting instances $z_i \in z_0, \ldots, z_{m-1}$ we try to find W such that:

$$\exists w \in \text{POST}(z_i) \iff \tag{4}$$

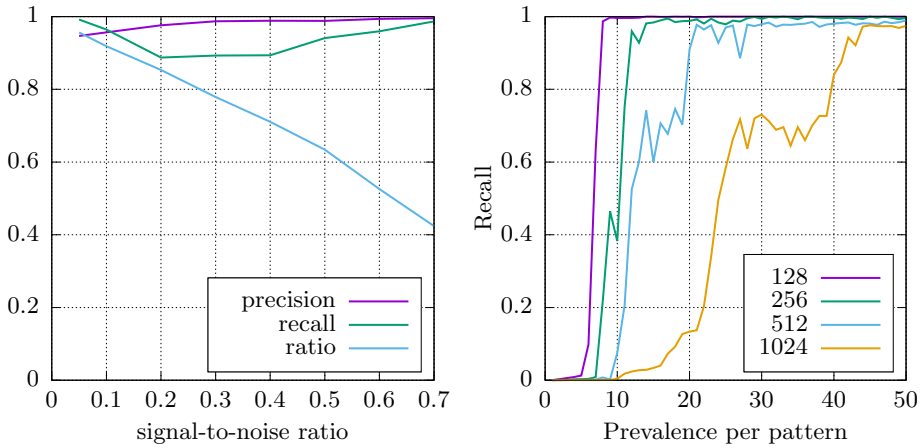
5 Experiments and Results

To asses the practical performance of the VOUW algorithm, we will primarily use the synthetic dataset generator RIL that was developed specifically for this purpose. RIL utilizes random walks to populate a matrix with patterns of a given size and prevalence, up to a specified density. We fill the remainder of the matrix with uniform noise which allows us to think of the density of patterns as the *signal-to-noise ratio* (SNR). The objective of the resulting experiment is that we try to find all of the signal (the patterns) and none of the noise.

5.1 Metrics for evaluation

Completely random data (noise) cannot be compressed (citation needed). Therefore if any compression is achieved, structure must be present in the original data. The SNR tells us how much noise is present in the data and thus conveniently gives us an upper bound of how much compression could be achieved. We use the ground truth SNR versus the resulting compression ratio as a benchmark to tell us how close we are in finding all the structure in the ground truth.

Because the compression ratio alone does not tell us the quality of the results, we also compare the ground truth matrix with the compressed result. In order to do this, we use the notion that elements that have been encoded with singleton patterns, could evidently not be compressed. These elements must therefore be noise. We reconstruct the original matrix from the compressed result, while we omit any singleton patterns. This essentially gives us a matrix of 'positives' (signal) and 'negatives' (noise). By comparing each element with the corresponding element in the ground truth matrix, the 'true positives' can be calculated. This subsequently gives us traditional figures for *precision* and *recall*.



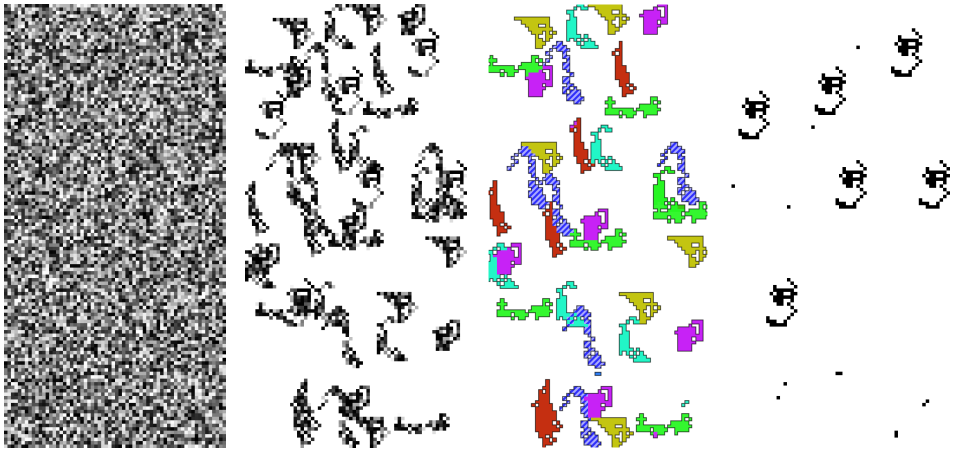
(a) The influence of the signal-to-noise ratio in the ground truth on the algorithm's performance

(b) Prevalence versus quality of the result (recall)

6 Conclusion

References

1. Bilson JL Campana and Eamonn J Keogh. A compression-based distance measure for texture. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 3(6):381–398, 2010.
2. Peter D Grünwald and Abhijit Grunwald. *The minimum description length principle*. MIT press, 2007.
3. Ming Li, Paul Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*, volume 3. Springer, 2008.
4. Hugo M Proença and Matthijs van Leeuwen. Interpretable multiclass classification by mdl-based rule lists. *arXiv preprint arXiv:1905.00328*, 2019.



(a) Generated matrix (b) Ground truth patterns (c) Found patterns (d) Difference

Fig. 6: Example of how synthetic input is generated and evaluated.

5. Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.
6. Koen Smets and Jilles Vreeken. Slim: Directly mining descriptive patterns. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 236–247. SIAM, 2012.
7. Jilles Vreeken, Matthijs Van Leeuwen, and Arno Siebes. Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214, 2011.