# An Efficient Algorithm for Computing Entropic Measures of Feature Subsets

Frédéric Pennerath[1,2](✉)

[1] Université de Lorraine, CentraleSupélec, CNRS, LORIA, 57000 Metz, France
frederic.pennerath@centralesupelec.fr
[2] Université Paris Saclay, CentraleSupélec, CNRS, LORIA, 57000 Metz, France

**Abstract.** Entropic measures such as conditional entropy or mutual information have been used numerous times in pattern mining, for instance to characterize valuable itemsets or approximate functional dependencies. Strangely enough the fundamental problem of designing efficient algorithms to compute entropy of subsets of features (or mutual information of feature subsets relatively to some target feature) has received little attention compared to the analog problem of computing frequency of itemsets. The present article proposes to fill this gap: it introduces a fast and scalable method that computes entropy and mutual information for a large number of feature subsets by adopting the divide and conquer strategy used by *FP-growth* – one of the most efficient frequent itemset mining algorithm. In order to illustrate its practical interest, the algorithm is then used to solve the recently introduced problem of mining *reliable approximate functional dependencies*. It finally provides empirical evidences that in the context of non-redundant pattern extraction, the proposed method outperforms existing algorithms for both speed and scalability. Code related to this chapter is available at: https://github.com/P-Fred/HFP-Growth.

**Keywords:** Pattern mining · Entropic measures
Algorithm efficiency · Approximate functional dependency
Pattern redundancy

## 1 Introduction

Entropic measures such as conditional entropy or mutual information have been used numerous times in pattern mining, for instance to characterize valuable itemsets [5–7] or approximate functional dependencies [3,8,9]. In such setting, one considers datasets where data are described by nominal features, i.e. features with a finite number of possible values. These data are interpreted as IID samples of some distribution for which the set $\mathcal{F}$ of features are seen as categorical random variables. For every considered subset $\mathcal{X} \subseteq \mathcal{F}$ of features, the entropy $\mathrm{H}(\mathcal{X})$ can be approximated by an empirical estimation $\hat{\mathrm{H}}(\mathcal{X}) = -\sum_t \sigma_{\mathcal{D}}(t) \log_2(\sigma_{\mathcal{D}}(t))$ where frequencies $\sigma_{\mathcal{D}}(t)$ are computed for all value combinations $t$ (latter called tuples)

M. Berlingerio et al. (Eds.): ECML PKDD 2018, LNAI 11052, pp. 483–499, 2019.
https://doi.org/10.1007/978-3-030-10928-8_29

of $\mathcal{X}$ observed in the dataset. A similar expression allows to empirically estimate mutual information $\mathrm{I}(\mathcal{X};\mathcal{Y}) = H(\mathcal{X}) + H(\mathcal{Y}) - H(X \cup \mathcal{Y})$ between $\mathcal{X}$ and a target feature subset $\mathcal{Y}$, usually restricted to a single target feature.

Entropy $\mathrm{H}(\mathcal{X})$ measures the amount of uncertainty when guessing samples of $\mathcal{X}$, or equivalently, the quantity of information conveyed by it. Similarly mutual information $\mathrm{I}(\mathcal{X};\mathcal{Y})$ is the amount of information shared between feature subsets $\mathcal{X}$ and $\mathcal{Y}$. Both quantities have interesting properties. In particular $\mathcal{X} \mapsto \mathrm{H}(\mathcal{X})$ and $\mathcal{X} \mapsto \mathrm{I}(\mathcal{X};\mathcal{Y})$ are non negative monotonic functions in lattice $(2^{\mathcal{F}}, \subseteq)$ of feature subsets. This property builds up a formal analogy with the anti-monotonic property of itemset frequency so that some frequent itemset mining techniques such as the levelwise search used by Apriori [1] have been transposed for the computation of entropic measures (see for instance [8]). Despite this analogy the problem of designing fast and scalable algorithms to compute entropy of feature subsets has received little attention compared to the problem of computing frequency of itemsets, for which many algorithms have been proposed [2]. The present article addresses this problem as it introduces a new algorithm to compute entropy and mutual information for a large number of feature subsets, adopting the same divide and conquer strategy used by *FP-growth* [4] – one of the most efficient frequent itemset mining algorithm [2].

In order to illustrate its practical interest, the algorithm is then used to solve specifically the recently introduced problem of mining *reliable approximate functional dependencies* [9]. Given a target feature $Y$, the problem consists in finding the top-k feature sets $\mathcal{X}_{1 \leq i \leq k}$ which have the $k$ highest *reliable fractions of information* relatively to $Y$. This score denoted $\hat{F}_0(\mathcal{X};Y)$ is a robust estimation of the normalized mutual information between $\mathcal{X}$ and $Y$ that is unbiased and equal to 0 in case of independence between $\mathcal{X}$ and $Y$. This prevents from misinterpreting strong observed dependencies between $\mathcal{X}$ and $Y$ that are not statistically representative because they are based on a too small number of data. In the same article, an algorithm is proposed to mine exactly or approximatively the top-k reliable approximate functional dependencies (RAFD) using a parallelized beam search strategy coupled to a branch and bound pruning optimization.

While authors of [9] focus on small values for $k$ (mainly $k = 1$), we are interested by much larger values, typically $k = 10^4$. This interest seems counterintuitive as the only presumable effect of increasing $k$ is to produce more uninteresting patterns with lower scores. In reality, top-k patterns provide highly redundant pieces of information as similar patterns are likely to have similar scores. Increasing $k$ provides a substantial list of top-$k$ patterns from which can be extracted a reduced set of still highly scored but non redundant patterns called *Locally Optimal Patterns* (LOP) [10,11]: Given a pattern scoring function and given a neighbourhood function that maps every pattern to a set of neighbouring patterns, a pattern $P$ is *locally optimal* if its score is maximal within $P$'s neighbourhood. The neighbourhood generally used is a $\delta$-metric neighbourhood: two sets of features $\mathcal{X}_1$ and $\mathcal{X}_2$ are neighbours if their distance $d(\mathcal{X}, \mathcal{X}')$ defined as the cardinalily $|\mathcal{X} \Delta \mathcal{X}'|$ of their symmetric difference is not greater than $\delta$. Once top-k patterns have been mined, LOPs can easily be extracted by checking for every top-k pattern if some better ranked pattern is one of its neighbour (however this naive algorithm with a complexity in $\Theta(k^2)$ only works for relatively small values of $k$. For more elaborate algorithm, see [10]).

For sake of illustration, the amount of redundancy of a top-k pattern $\mathcal{X}$ can be assessed by the minimal distance $\delta_{min}(\mathcal{X})$ between $\mathcal{X}$ and any better scored pattern: the higher the distance, the more original the pattern. The first columns of Table 1 provide the histogram of $\delta_{min}$ of top-2 to top-10000 patterns, computed on some datasets used in the evaluation section. For almost every dataset, between 97 % to 100 % of top-k patterns differ only with one single feature from a better scored pattern. On the other side, the last four columns of Table 1 provide the rank distribution of LOPs (for $\delta = 1$) in the sorted list of top-k patterns.

**Table 1.** Histograms of $\delta_{min}$ and LOPs' rank among the top-10000 patterns.

| Dataset | Distance $\delta_{min}$ | | | | | Rank of LOP | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | >1 | 1–10 | 11–100 | 101–1000 | 1001–10000 |
| german | 9968 | 29 | 2 | | 32 | 4 | 8 | 13 | 7 |
| lymphography | 9964 | 33 | 1 | 1 | 36 | 4 | 5 | 15 | 12 |
| vehicle | 9943 | 54 | 1 | 1 | 57 | 6 | 20 | 17 | 14 |
| sonar | 9753 | 233 | 5 | 7 | 247 | 7 | 27 | 77 | 136 |
| penbased | 9719 | 279 | 0 | 1 | 281 | 10 | 76 | 154 | 41 |
| segment | 9961 | 38 | 0 | 0 | 39 | 5 | 5 | 22 | 7 |
| specftheart | 9902 | 95 | 0 | 1 | 98 | 3 | 8 | 36 | 51 |
| twonorm | 5157 | 4840 | 2 | 0 | 4843 | 10 | 90 | 900 | 3843 |
| wdbc | 9865 | 130 | 2 | 1 | 135 | 9 | 19 | 41 | 66 |

One notices that a significant part of LOPs have large ranks. It is thus essential to be able to mine top-k patterns with large values for $k$. Main contributions of this paper are:

1. An algorithm to compute entropy and mutual information of feature subsets, resulting from a non straightforward adaptation of the frequent itemset mining algorithm *FP-growth* [4].
2. An adaptation of the previous algorithm to address the problem of discovering the top-$k$ *Reliable Approximate Functional Dependencies* [9]. The algorithm mines large numbers of patterns with low memory footprint so that it becomes possible to extract "hard-to-reach" locally optimal patterns.
3. Empirical evidences that for both problems, proposed algorithms outperform existing methods for both speed and scalability.

The rest of the paper is structured as follows: Sect. 2 considers the general problem of fast and scalable computation of entropic measures on sets of features and introduces the algorithm *HFP-growth*. Section 3 introduces algorithm *IFP-growth* to compute mutual information relatively to some target feature and applies it to the discovery of Reliable Approximate Functional Dependencies. Section 4 presents comparative tests performed to evaluate speed and scalability of HFP-growth and IFP-growth, before Sect. 5 concludes.

## 2 An Algorithm to Compute Entropy of Feature Subsets

### 2.1 Definitions and Problem Statement

Let's first define properly the required notions of entropy and data partitions. Given a dataset $\mathcal{D}$ of $n$ data described by a set $\mathcal{F}$ of nominal features, data are interpreted as IID samples of some distribution where every feature $X \in \mathcal{F}$ is seen as a categorical random variable defined over some domain denoted $\mathcal{D}_X$. Given a subset $\mathcal{X} = \{X_1, \ldots, X_k\}$ of features listed in some arbitrary order, its *joint distribution* $P_\mathcal{X}$ is defined over the cartesian product $T(\mathcal{X}) = \mathcal{D}_{X_1} \times \cdots \times \mathcal{D}_{X_k}$ containing all $k$-tuples $(x_1, \ldots, x_k)$ for all $x_1 \in \mathcal{D}_{X_1}, \ldots, x_k \in \mathcal{D}_{X_k}$. Assuming the undefined form $0 \times \log_2(0)$ is equal to zero, the *entropy* $\mathrm{H}(\mathcal{X})$ of this joint distribution is defined as:

$$\mathrm{H}(\mathcal{X}) \overset{\text{def}}{=} \mathbb{E}\left(\log_2\left(\frac{1}{P_\mathcal{X}}\right)\right) = -\sum_{t \in T(\mathcal{X})} P_\mathcal{X}(t) \log_2(P_\mathcal{X}(t)) \tag{1}$$

The *empirical entropy* $\hat{\mathrm{H}}(\mathcal{X})$ estimates this entropy from the available samples, replacing probability $P_\mathcal{X}(t)$ of tuple $t$ with its *relative frequency* $\sigma_\mathcal{D}(t)$ in $\mathcal{D}$. Entropy is a monotonic function: given two feature subsets $\mathcal{X}_1$ and $\mathcal{X}_2$, $\mathcal{X}_1 \subseteq \mathcal{X}_2$ implies $\hat{\mathrm{H}}(\mathcal{X}_1) \leq \hat{\mathrm{H}}(\mathcal{X}_2)$. The entropy is minimal and equal to zero for the empty set; it is maximal for the whole set $\mathcal{F}$ of features. In order to formalize the problem of computing efficiently the entropy of a large number of feature subsets, one considers the analog problem of computing frequency of frequent patterns. To this end, one defines the *relative entropy* $\hat{h}(\mathcal{X})$ as the ratio of $\hat{\mathrm{H}}(\mathcal{X})$ over the maximal possible entropy $\hat{\mathrm{H}}(\mathcal{F})$ of all features so that its values are always between 0 and 1. One then says a subset $\mathcal{X}$ of features is *definite* relatively to some threshold $h_{max} \in [0, 1]$ if $\hat{h}(\mathcal{X}) \leq h_{max}$ (Definite subsets of binary features are also called low-entropy sets in [6]). The considered problem is then the following:

*Problem 1.* Given a dataset $\mathcal{D}$ of nominal features and a threshold $h_{max} \in [0, 1]$, the *problem of mining definite feature subsets* consists in computing the empirical entropy of every definite subset of features relatively to $h_{max}$ and $\mathcal{D}$.

While this problem can naively be solved by implementing an APriori like algorithm [1] based on formula 1, this method is highly unefficient, not only because the APriori approach is not the best strategy but also because expression 1 requires for every feature subset $\mathcal{X}$ to compute frequencies of all possible tuples whose number increases exponentially with the size of $\mathcal{X}$. In order to provide a more efficient algorithm, empirical entropy should be defined as a function of data partitions. A data partition is any partition of the dataset $\mathcal{D}$. Any subset $\mathcal{X}$ of features can be mapped to a data partition. For this purpose, let's say two data $d_1$ and $d_2$ are *equivalent* relatively to $\mathcal{X}$ if for every feature $X$ of $\mathcal{X}$, their respective values $X(d_1)$ and $X(d_2)$ are equal. The set of equivalence classes defines the said *data partition* denoted $\mathcal{P}(\mathcal{X})$ of $\mathcal{X}$. The empirical entropy $\hat{\mathrm{H}}(\mathcal{X})$ can thus be rewritten as the entropy $\mathrm{H}(\mathcal{P}(\mathcal{X}))$ of its data partition defined as:

$$\mathrm{H}(\mathcal{P}(\mathcal{X})) \overset{\text{def}}{=} \sum_{P \in \mathcal{P}(\mathcal{X})} \frac{|P|}{|\mathcal{D}|} \log_2\left(\frac{|\mathcal{D}|}{|P|}\right) = \log_2(n) - \frac{\sum_{P\in\mathcal{P}(\mathcal{X})} |P| \log_2(|P|)}{n} \quad (2)$$

The set of data partitions builds a lattice with intersection and union operators that necessarily induces an ordering relation called refinement relation: a partition $\mathcal{P}_1$ is a *refinement of* (or is *included in*) a partition $\mathcal{P}_2$ if every part of $\mathcal{P}_1$ is included in any part of $\mathcal{P}_2$. The intersection $\mathcal{P}_1 \cap \mathcal{P}_2$ is the most general refinement of $\mathcal{P}_1$ and $\mathcal{P}_2$:

$$\mathcal{P}_1 \cap \mathcal{P}_2 = \{\, P_1 \cap P_2 \,/\, P_1 \in \mathcal{P}_1,\ P_2 \in \mathcal{P}_2,\ P_1 \cap P_2 \neq \emptyset\} \quad (3)$$

It is easy to prove by double inclusion that the data partition of a set $\mathcal{X}$ of features is the intersection of data partitions of all features of $\mathcal{X}$:

$$\mathcal{P}(\mathcal{X}) = \bigcap_{X\in\mathcal{X}} \mathcal{P}(\{X\}) \quad (4)$$

This latter property is essential to design an efficient mining algorithm. Indeed let's assume there exists some encoding of data partitions that enables an efficient procedure to compute the intersection of two data partitions and its entropy. Under this hypothesis, it gets possible to enumerate efficiently in a depth first search manner the definite feature subsets, by intersecting the data partition $\mathcal{P}(\mathcal{X})$ of the current pattern $\mathcal{X}$ with the data partition $\mathcal{P}(\{Y\})$ of the next feature $Y \notin \mathcal{X}$ to add and then by pruning the current branch as soon as the entropy of resulting partition $\mathcal{P}(\mathcal{X} \cup \{Y\})$ is larger than $H_{limit} = h_{max} \times \hat{\mathrm{H}}(\mathcal{F})$. The next section explains the algorithm in details.

### 2.2 Algorithm HFP-growth

The presented algorithm is called *HFP-growth* since it adopts FP-growth's data structure called *FP-tree* along with its divide and conquer strategy [4]. The symbol of entropy "H" emphasizes the fact HFP-growth computes entropy of definite features sets instead of frequency of frequent itemsets. The next paragraphs develop the three main components of HFP-growth: first, the *HFP-tree* data structure that is an adaptation of FP-trees, then the algorithmic primitives to process the HFP-tree which are the most different part compared to FP-growth, finally the global algorithm with its divide and conquer approach.

**Table 2.** Dataset

| $F_1$ | $F_2$ | $F_3$ |
|---|---|---|
| a | c | f |
| a | c | g |
| a | d | f |
| a | e | f |
| b | c | f |
| b | c | g |
| b | d | f |
| b | e | f |

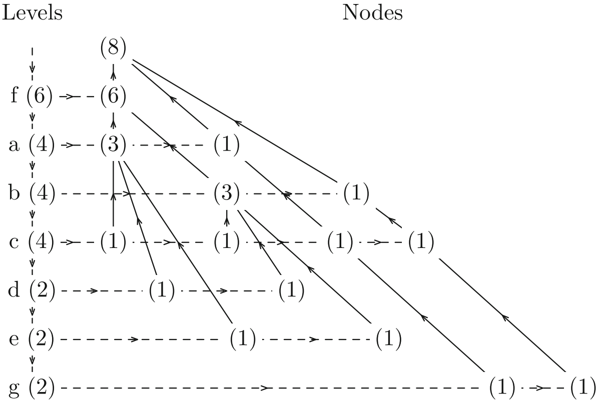


**Fig. 1.** Equivalent FP-tree

**HFP-Tree.** FP-growth is a frequent itemset mining algorithm that stores the whole dataset in memory thanks to a compact data structure called *FP-tree*. Many variants of FP-tree exist. Only the simplest most essential form is presented here: An FP-tree is mainly a lexicographic tree, also called trie, that encodes a dataset viewed as a collection of itemsets. In order for the trie to have the smallest memory footprint and thus the smallest number of nodes, the unfrequent items are removed and the remaining frequent items are sorted in decreasing order of frequency. The FP-tree represented on Fig. 1 is built from dataset shown on Table 2. An FP-tree also provides for every item $i$ a single linked list enumerating nodes representing item $i$ in the lexicographic tree, traversing the trie from left to right. These lists are called *levels* hereafter. Levels are represented with dashed lines on Fig. 1. A node $n$ represents the itemset containing items of levels intersecting the branch from the root node up to $n$. For instance rightmost node of $g$'s level represents itemset $bcg$. Every node essentially stores pointers to its parent node and to its right sibling in its level, along with a counter set to the number of data containing node's itemset.

HFP-tree is an FP-tree with some differences as shown on Fig. 2:

- An HFP-tree has additional components called *groups* of levels. Every group corresponds to some feature $X$. $X$'s group is the entry point for levels, one for each possible value of $X$. Levels attached to $X$'s group thus represent the parts of $\mathcal{P}(\{X\})$.
- A group also stores entropy $\hat{H}(\{X\})$ of $X$ computed at startup when reading the dataset. Features whose entropy $\hat{H}(\{X\})$ is greater than $H_{limit}$ are not considered as they cannot be part of a definite feature set. This allows to sort groups in increasing order of entropy so that nodes representing the most definite features are close to the root node whereas leaf nodes at the bottom of the tree represent the most fluctuating features: this trick reduces the number of nodes, while remaining compatible with the intersection procedure

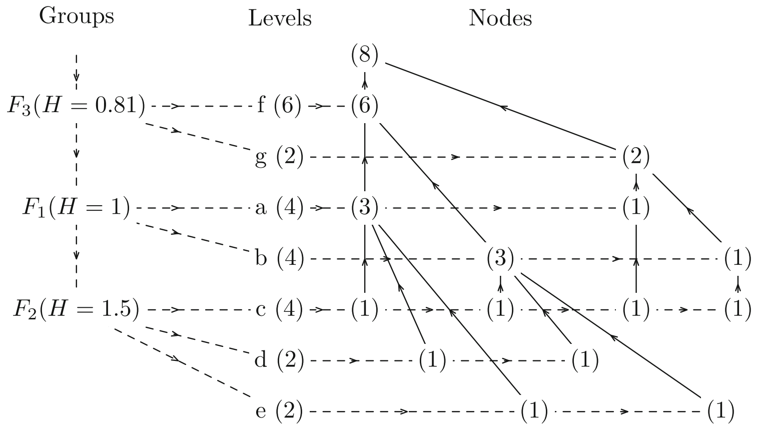


**Fig. 2.** An HFP-tree

explained later. In particular it does not interlace levels of different groups as standard FP-tree would do (like $f$ and $g$ levels of group $F_3$ on Fig. 1).

- A node stores in addition to fields already mentioned for FP-tree, two additional node pointers called *master* and *heir* that are essential for processing the tree as explained in the next subsection.
- Another difference compared to FP-growth is that only one HFP-tree is built to represent the input dataset. This characteristic is very convenient as HFP-growth does not dynamically allocate memory but at startup. If HFP-growth succeeds in building its tree (and in practice it always does on current datasets less than few gigabytes), it will eventually complete without running out of memory after potentially several hours of processing. In comparison FP-growth clones many FP-trees during its run. However this advantage has to be mitigated as some later FP-growth implementations have managed to avoid FP-tree cloning.

**Processing HFP-Tree.** The interest of HFP-tree is to enable a fast computation of data partitions when feature subsets are enumerated in a depth search order consistent with the way features are indexed. To explain why, let's describe in a first stage how a data partition $\mathcal{P}\left(\mathcal{X}\right)$ is encoded in the HFP-tree for some given subset $\mathcal{X}$ of features without explaining how this encoding can be built. In the followings, index $i$ of feature $X_i$ refers to the feature of the $i^{\text{th}}$ group: feature $X_1$ matches the first group, i.e the closest of the tree root, feature $X_2$ matches the $2^{\text{nd}}$ closest group and so on. Let's assume $\mathcal{X} = \{X_{i_1}, \ldots, X_{i_k}\}$ with $i_1 < \cdots < i_k$. Individual nodes in $X_{i_k}$'s levels represent parts of $\mathcal{P}\left(\cup_{i=1}^{i_k}\{X_i\}\right) = \cap_{i=1}^{i_k}\mathcal{P}\left(\{X_i\}\right)$. Since $\mathcal{P}\left(\cup_{i=1}^{i_k}\{X_i\}\right)$ is a refinement of $\mathcal{P}\left(\mathcal{X}\right)$, for any given part $P$ of $\mathcal{P}\left(\mathcal{X}\right)$, there are several nodes of $X_{i_k}$'s group that are part of $P$, or put another way, data members of $P$ are covered by different nodes of $X_{i_k}$'s group. These nodes representing $P$ might spread among several levels of the ${i_k}^{\text{th}}$ group but they can be identified as those sharing a same reference to a representative node called *master node*. This master node can be any node of any previous or current group (i.e whose index is not greater than $i_k$). The only requirement is that two nodes belong to the same part if and only if their master nodes are the same. The use of master nodes to implicitely represent parts of partitions avoids dynamic memory allocation of objects to explicitely represent these parts. This implementation technique substantially improves speed. At startup, the only available partition encoding is represented by the root node, which is a special node as it is the only one not to be a member of any level of any group. The root node is its own master node. It represents the zero entropy partition $\mathcal{P}(\emptyset) = \{\mathcal{D}\}$ of the empty set of features, made of a single part containing the whole dataset.

Now the remaining issue is to define the recursive generation of partition encodings: given a current pattern $\mathcal{X}$ whose partition is encoded on nodes of the ${i_k}^{\text{th}}$ group, how can be generated partition encoding for any child pattern $\mathcal{X} \cup \{X_{i_{k+1}}\}$ of $\mathcal{X}$? Put another way, let's consider some new feature $X_j$ with $j > i_k$ and let's assume nodes of the previous $(j-1)^{\text{th}}$ group encode partition of $\mathcal{X}$, then the $j^{\text{th}}$ group must generate two different exploration branches:

- Either one adds feature $X_j$ to current pattern $\mathcal{X}$. In this case, one has to encode on nodes of the $j^{\text{th}}$ group, partition $\mathcal{P}(\mathcal{X} \cup \{X_j\}) = \mathcal{P}(\mathcal{X}) \cap \mathcal{P}(\{X_j\})$ using (1) the available encoding of partition $\mathcal{P}(\mathcal{X})$ by nodes of the $(j-1)^{\text{th}}$ group and (2) the partition $\mathcal{P}(\{X_j\})$ whose parts are levels of the $j^{\text{th}}$ group. This is called the `intersect` operation applied to the $j^{\text{th}}$ group.
- Or feature $X_j$ is not added to $\mathcal{X}$. In this case, one simply has to forward the available encoding of $\mathcal{P}(\mathcal{X})$ from nodes of the $(j-1)^{\text{th}}$ group to nodes of the current $j^{\text{th}}$ group. This is called the `skip` operation applied to the $j^{\text{th}}$ group.

The `skip` operation can easily be parallelized as it simply consists for every node $n$ of every level of the $j^{\text{th}}$ group to declare its master node to be the master node of its parent node. The `intersect` operation is more subtle as every part of the $j - 1^{\text{th}}$ group might be intersected by different levels of the $j^{\text{th}}$ group. For every level $L$ of the $j^{\text{th}}$ group, one has to gather nodes of $L$ whose parents are member of the same part of $\mathcal{P}(\mathcal{X})$, say otherwise, whose parents have the same master node. These subsets define new parts of $\mathcal{P}(\mathcal{X} \cup \{X_j\})$ as illustrated on Fig. 3.

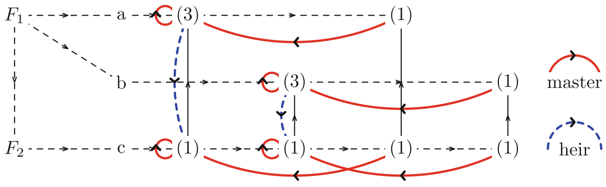


**Fig. 3.** Running `intersect` on $c$'s level after skipping $F_3$ and intersecting $F_1$

More precisely, every time the parent of a node $n$ of $L$ has a master $m$ not processed yet, a new part $P$ of $\mathcal{P}\left(\mathcal{X} \cup \{X_j\}\right)$ is discovered. $n$ is then designated as the master of $P$ and its reference is saved in master $m$ as its *heir* node. When other nodes are found such that the master $m$ of their parents is the same as the one for $n$, they receive the heir of $m$ as their master. After processing nodes of a level, all discovered parts are complete so that the sum appearing in expression 2 can be updated by their cardinalities. After processing all levels of the group $j$, its nodes completely encode partition $\mathcal{P}\left(\mathcal{X} \cup \{X_j\}\right)$ and since the sum of expression 2 is completed, `intersect` can return entropy $\hat{\mathrm{H}}(\mathcal{X} \cup \{X_j\})$.

**Algorithm.** Once the HFP-tree has been built, HFP-growth uses a divide and conquer strategy based on the two previous operations `skip` and `intersect`. This strategy is similar with the one of FP-growth but while `skip` and `intersect` require a top down recursion (i.e. from the top of the tree to the bottom), FP-growth's algorithm requires to process levels in a bottom up order, starting from the deepest levels in the FP-tree. Let $m$ be the number of groups in the HFP-tree, i.e. the number of features of $\mathcal{F}$ after removing feature $X$ with entropy less than $H_{limit}$. The algorithm is based on a recursive `mine` function as shown by pseudocode 1. For every feature index $i$ from 1 to $m-1$, it recursively calls `mine` on the next $i+1^{\text{th}}$ feature twice: first after applying the procedure `intersect` on the $i^{\text{th}}$ group in the case the returned entropy is not greater than $H_{limit}$, second after appling `skip` to the $i^{\text{th}}$ group systematically.

```
Inputs : A dataset $\mathcal{D}$ and a threshold $h_{max} \in [0,1]$
Output: List of definite feature sets $\mathcal{X}$ with their entropy $\hat{\mathrm{H}}(\mathcal{X})$

HFP-tree $\mathcal{T}$, $H_{limit}$ ← build-HFP-tree($\mathcal{D}, h_{max}$) ;
output($\emptyset, 0$) ;
mine($\emptyset, 1$)

function mine($\mathcal{X}, j$) is
    if $j \leq$ number $m$ of groups of $\mathcal{T}$ then
        $H$ ← intersect($\mathcal{T}, j$) ;
        if $H \leq H_{limit}$ then
            output($\mathcal{X} \cup \{X_j\}, H$) ;
            mine($\mathcal{X} \cup \{X_j\}, j+1$)
        end
        skip($\mathcal{T}, j$) ;
        mine($\mathcal{X}, j+1$)
    end
end
```

**Algorithm 1.** The HFP-growth algorithm

# 3 An Algorithm to Compute Mutual Information of Feature Subsets

In this section one shows how HFP-growth can be adapted to compute efficiently mutual information of feature subsets with some target feature, along with an unbiased variant of it introduced in [9]. Because the main subject of this paper is the efficient computation of entropic measures, one could limit the study to mutual information since computing the unbiased variant does not fundamentally change the algorithm. However this provides at the same time a sounder statistical problem to solve and an existing algorithm to compare with. The resulting algorithm is called *IFP-growth*, "I" standing for information.

## 3.1 Problem Statement

Mutual information $\mathrm{I}(\mathcal{X};\mathcal{Y}) = H(\mathcal{X}) + H(\mathcal{Y}) - H(X \cup \mathcal{Y})$ estimates the amount of information shared by two feature subsets $\mathcal{X}$ and $\mathcal{Y}$. It is equal to zero in case $\mathcal{X}$ and $\mathcal{Y}$ are independent. Mutual information is particularly interesting in supervised problems where $\mathcal{Y}$ is restricted to a single target feature $Y$. In such problems, one searches to predict $Y$ from highly dependent feature subsets $\mathcal{X}$, i.e. with a large mutual information $\mathrm{I}(\mathcal{X};\{Y\})$. Mutual information can be estimated empirically from data partitions according to:

$$\hat{\mathrm{I}}(\mathcal{X};\mathcal{Y}) = \hat{\mathrm{H}}(\mathcal{P}(\mathcal{Y})) + \hat{\mathrm{H}}(\mathcal{P}(\mathcal{X})) - \hat{\mathrm{H}}(\mathcal{P}(\mathcal{X}) \cap \mathcal{P}(\mathcal{Y})) \qquad (5)$$

Because $\mathrm{H}(\mathcal{Y})$ is an upper bound for $\mathrm{I}(\mathcal{X};\mathcal{Y})$, one often uses a normalized mutual information $F(\mathcal{X};\mathcal{Y}) = \frac{\mathrm{I}(\mathcal{X};\mathcal{Y})}{\mathrm{H}(\mathcal{Y})}$ within the range $[0,1]$ called *fraction of information ratio* in [3,9]. As stated in the introduction, mutual information $\mathcal{X} \mapsto \hat{\mathrm{I}}(\mathcal{X},\mathcal{Y})$ is a non decreasing function of $\mathcal{X}$. Obviously the more features in $\mathcal{X}$, the more predictable $\mathcal{Y}$ from $\mathcal{X}$. However one should not forget that a dataset $\mathcal{D}$ is a limited sampling of the real joint distribution of $\mathcal{X}$ and $\mathcal{Y}$. For large feature sets $\mathcal{X}$, data partition of $\mathcal{P}(\mathcal{X} \cup \mathcal{Y})$ is the intersection $\cap_{X \in \mathcal{X} \cup \mathcal{Y}} \mathcal{P}(\{X\})$ of many data partitions. Therefore parts of $\mathcal{P}(\mathcal{X} \cup \mathcal{Y})$ get statistically very small when the size of $\mathcal{X}$ increases. Within these small parts, strong but spurious dependencies appear between $\mathcal{X}$ and $\mathcal{Y}$ even when $\mathcal{X}$ and $\mathcal{Y}$ are drawn from independent distributions. New entropic measures have since been proposed in [9,12–14]. These measures are similar in spirit with mutual information but robust to the previous "just by chance" artefact. One of these mesures considered in [9] is the *reliable fraction of information* $\hat{F}_0(\mathcal{X};\mathcal{Y})$ that is unbiased, i.e whose expected value is 0 when $\mathcal{X}$ and $\mathcal{Y}$ are independent. More precisely $\hat{F}_0(\mathcal{X};\mathcal{Y}) = \hat{F}(\mathcal{X};\mathcal{Y}) - \frac{\hat{m}_0(\mathcal{X},\mathcal{Y})}{\hat{\mathrm{H}}(\mathcal{Y})}$ where $\hat{m}_0(\mathcal{X},\mathcal{Y})$ is the expected value of $\hat{\mathrm{I}}(\mathcal{X};\mathcal{Y})$ under hypothesis of independence between $\mathcal{X}$ and $\mathcal{Y}$. This bias is computed using a permutation model defined over contingency tables of $\mathcal{X}$ and $\mathcal{Y}$ respectively. Since the elements in these tables are nothing else than the cardinalities of parts in $\mathcal{P}(\mathcal{X})$ and $\mathcal{P}(\mathcal{Y})$, $\hat{m}_0(\mathcal{X},\mathcal{Y})$ can be rewritten as $\hat{m}_0(\mathcal{P}(\mathcal{X}), \mathcal{P}(\mathcal{Y}))$, i.e as a function depending only on $\mathcal{P}(\mathcal{X})$ and $\mathcal{P}(\mathcal{Y})$. The exact expression of $\hat{m}_0(\mathcal{P}(\mathcal{X}), \mathcal{P}(\mathcal{Y}))$ is given by a sum of expected values for hypergeometric distributions. The detailed equation and derivation details are provided in [9] with reference to [12,14]. The expression of $\hat{F}_0(\mathcal{X};\mathcal{Y})$ to compute is finally:

$$\hat{F}_0(\mathcal{X};\mathcal{Y}) = 1 + \frac{\hat{\mathrm{H}}(\mathcal{P}(\mathcal{X})) - \hat{\mathrm{H}}(\mathcal{P}(\mathcal{X}) \cap \mathcal{P}(\mathcal{Y})) - \hat{m}_0(\mathcal{P}(\mathcal{X}), \mathcal{P}(\mathcal{Y}))}{\hat{\mathrm{H}}(\mathcal{P}(\mathcal{Y}))} \quad (6)$$

It is worth noting $\mathcal{X} \mapsto \hat{F}_0(\mathcal{X};\mathcal{Y})$ is not a monotonic function as mutual information is. It has the expected nice property of penalizing with low scores not only short non informative sets of features $\mathcal{X}$ but also long informative but not statistically representative patterns. Finding the top-$k$ feature sets $\mathcal{X}$ with highest score $\hat{F}_0(\mathcal{X};\{Y\})$ relatively to a target feature $Y$ is thus a sound and non trivial optimization problem addressed in [9]: the resulting associations $\mathcal{X} \to Y$ are called the top-$k$ *reliable approximate functional dependencies* (RAFD). A mining algorithm is also proposed in [9] whose implementation is called `dora`. This algorithm finds these dependencies using a beam search strategy coupled to a branch and bound pruning: it backtracks the current branch as soon as the upper bound $1 - \hat{m}_0(\mathcal{X};\{Y\})/\hat{\mathrm{H}}(\{Y\})$ of scores accessible from the current pattern $\mathcal{X}$ is not greater than score $\hat{F}_0(\mathcal{X}_k;\{Y\})$ of the worst top-$k$ pattern $\mathcal{X}_k$ found so far. In order to process difficult datasets, the algorithm can also solve a relaxed version of this problem: it consists in replacing the previous pruning condition by predicate $\alpha \times (1 - \hat{m}_0(\mathcal{X};\{Y\})/\hat{\mathrm{H}}(\{Y\})) \leq \hat{F}_0(\mathcal{X}_k;\{Y\})$ for some parameter $\alpha \in ]0,1]$, a value $\alpha = 1$ corresponding to the exact resolution. This approximation amounts to find $k$ feature subsets $(\hat{\mathcal{X}}_i)_{1 \leq i \leq k}$ so that the lowest score $\hat{F}_0(\hat{\mathcal{X}}_k;\{Y\})$ of these patterns is not lower than $\alpha \times \hat{F}_0(\mathcal{X}_k;\{Y\})$ where $\mathcal{X}_k$ is the pattern with the lowest score among the real top-$k$ patterns $(\mathcal{X}_i)_{1 \leq i \leq k}$. In summary, mining *reliable approximate functional dependencies* is defined by a dataset $\mathcal{D}$, a target feature $Y$, a number $k$ and an $\alpha$ coefficient. A new algorithm to address this problem is proposed in the next section.

### 3.2 Algorithm IFP-Growth

Two preliminary remarks can be done about Eq. 6: first $\mathcal{P}(\mathcal{Y})$ and a fortiori $\hat{\mathrm{H}}(\mathcal{P}(\mathcal{Y}))$ are constants independent of $\mathcal{X}$ so that they can be computed once forever at startup. Second $\hat{F}_0(\mathcal{X};\mathcal{Y})$ could be computed directly with two parallel HFP-trees: one whose groups encode $\mathcal{P}(\{X\})$ for every feature $X \in \mathcal{F} \setminus \{Y\}$, the second whose groups encode $\mathcal{P}(\{X\} \cap \mathcal{P}(\{Y\}))$. However this approach is memory-costly while it is possible to get a solution with a unique HFP-tree. The resulting algorithm *IFP-growth* is summarized by pseudocode 2.

A first trick of `IFP-growth` is to put the group of target feature $Y$ at the bottom of HFP-tree, independently of its entropy. This last group is processed differently than the others. Another trick is to switch the order `HFP-growth`

```
Inputs : A dataset $\mathcal{D}$, target feature $Y$, number $k$, coef. $\alpha \in [0,1]$
Output: List of top-$k$ feature sets $(\mathcal{X}_i)_{1\leq i\leq k}$ with their score $\hat{F}_0(\mathcal{X}_i)$
$F_0^{kth} \leftarrow 0$ ; $Q \leftarrow$ build-min-priority-queue() ;
HFP-tree $\mathcal{T}$, $H_Y \leftarrow$ build-IFP-tree($\mathcal{D}, Y$) ;
mine($\emptyset, 1$) ; Output content of $Q$
function mine $(\mathcal{X}, j)$ is
    if $j <$ number $m$ of groups of $\mathcal{T}$ then
        skip($\mathcal{T}, j$) ; mine($\mathcal{X}, j+1$) ;
        $H_X \leftarrow$ intersect($\mathcal{T}, j$)  $m_0 \leftarrow$ compute-bias($\mathcal{T}, j$) ;
        if $1 - m_0/H_Y > F_0^{kth}/\alpha$ then
            mine($\mathcal{X} \cup \{X_j\}, j+1$)
        end
    else
        $H_{XY} \leftarrow$ intersect($\mathcal{T}, j$) ;
        $F_0 \leftarrow 1 + (H_X - H_{XY} - m_0)/H_Y$ ;
        $Q$.insert(($\mathcal{X}, F_0$)) ;
        if $Q$.size $> k$ then
            $Q$.pop-min() ; $F_0^{kth} \leftarrow$ min($Q$).$F_0$
        end
    end
end
```

**Algorithm 2.** The IFP-growth algorithm

calls intersect and skip: IFP-growth first develops the skip 's branch before intersect 's one. When intersect is applied to $i_k$th group in order to generate encoding of $\mathcal{P}(\{X_{i_1}, \ldots, X_{i_k}\})$, combination of these two changes allow:

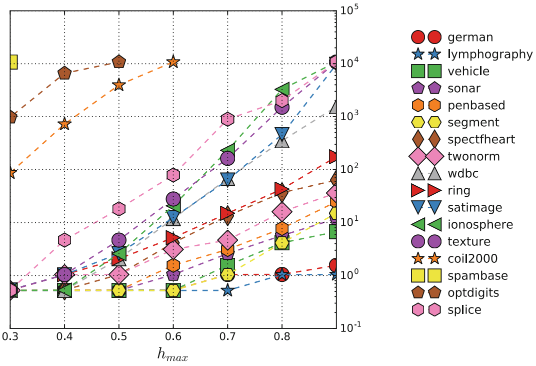
- First to save in global variables the entropy $\hat{\mathrm{H}}(\mathcal{X})$ (that intersect just returned) and the bias $\hat{m}_0(\mathcal{X}; \{Y\})$ that can be computed from constant $\mathcal{P}(\mathcal{Y})$ and from $\mathcal{P}(\mathcal{X})$ (whose encoding has also been computed by intersect).
- Then to call recursively the skip operation on the successive groups up to reaching the last group of $Y$ without modifying values of the two global variables. Calling intersect then returns $\hat{\mathrm{H}}(\mathcal{X} \cup \{Y\})$. At this points all terms of Eq. 6 are available to compute $\hat{F}_0(\mathcal{X}_k; \{Y\})$ and see if this score is sufficient for it to be inserted in the priority queue $Q$ storing the top-$k$ patterns.

As shown by pseudocode 2, the branch and bound pruning strategy with coefficient $\alpha$ can seamlessly be integrated in IFP-growth. However IFP-growth pruning is assumed to be less efficent than dora's one since dora uses a beam search strategy converging quickly to good top-k pattern candidates while HFP-tree dictates more rigidly the order in which subsets must be enumerated.

## 4 Empirical Evaluation

For sake of comparison, datasets considered in [9] are reused (see the KEEL data repository at http://www.keel.es). However in order to limit their number, only the most challenging datasets given on Table 3 are considered, defined as those whose processing by either `IFP-growth` or `dora` requires more than 10 s for $k = 1$. In order to be fair with `dora` whose beam search strategy is compatible with an intensive use of multithreading, tests are run on an Intel Xeon Silver 4414 biprocessor with a total of 20 hyper-threaded cores. The memory footprint of every running algorithm is monitored (internal Java Virtual Machine's heap size for `dora` and processus heap size for other algorithms) and limited to 45 GB. Source codes of HFP-growth, IFP-growth and HApriori can be downloaded from https://github.com/P-Fred/HFP-Growth whereas `dora` is available from http://eda.mmci.uni-saarland.de/prj/dora.

**Evaluation of HFP-Growth.** Since HFP-growth cannot be straightforwardly compared with an existing algorithm, one studies to what extent the well known performance gap between FP-growth and the baseline algorithm APriori [2] is reproduced between HFP-growth and the APriori counterpart, specially implemented for this purpose. This latter algorithm, hereafter called `H-APriori`, uses the same levelwise pruning method as APriori to remove candidate subsets having at least one predecessor that is not definite. It then computes in one pass over the dataset the entropies of all remaining candidates using formula 1. HFP-growth's processing times (resp. memory footprints) as functions of $h_{max}$ are given on Fig. 4 (resp. Fig. 5). The time (resp. memory) gain factor defined as the ratio of H-APriori's processing time (resp. memory footprint) over HFP-growth's one is provided on Fig. 6 (resp. Fig. 7).



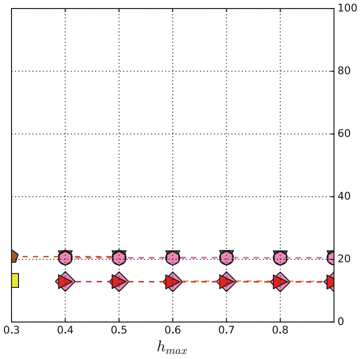


**Fig. 4.** HFP-growth's processing times (in seconds)

**Fig. 5.** HFP-growth's memory footprints (in megabytes)

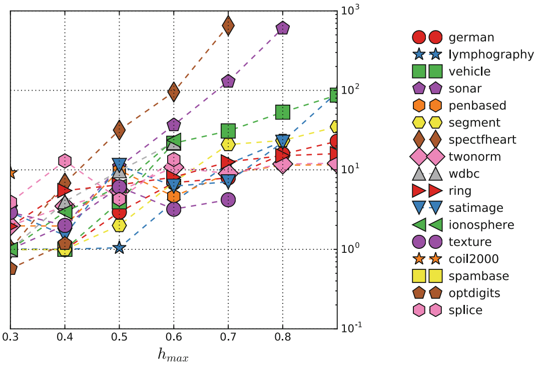


**Fig. 6.** Time gain factors

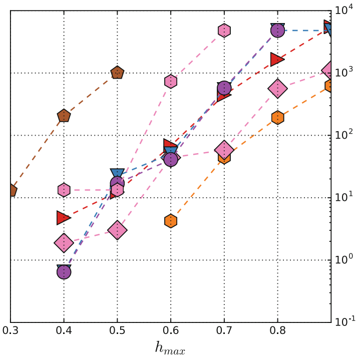


**Fig. 7.** Memory gain factors

HFP-growth appears 1 to 3 orders of magnitude faster than H-APriori, with a speedup factor increasing with threshold $h_{max}$. Memory gain is even bigger: HFP-growth has a low constant memory footprint of few megabytes (values on Fig. 5 appear to be rounded by the OS's heap allocation mechanism) whereas H-APriori shortly requires many gigabytes of memory to store candidate patterns. While HFP-growth can run as long as necessary, H-APriori often runs out of memory before completing.

**Evaluation of IFP-Growth.** In order to limit the number of tests to compare IFP-growth and `dora`, one only considers the exact problem of discovering RAFD, i.e for $\alpha = 1$. However similar conclusions can be drawn for values of $\alpha$ less than one. Table 3 summarizes processing times and memory footprints of both algorithms for the selected datasets and for increasing values of $k$ from 1 to $10^6$. A memory footprint of ">45" means the algorithm prematurely stopped as it ran out of memory. A processing time of ">3 h" means the algorithm was interrupted after a time limit of 3 h.

Results are similar with the ones obtained for HFP-growth: For all datasets and all values of $k$, IFP-growth appears always faster than `dora`. The gap between processing time tends to increase with $k$. While `dora` cannot complete 15 over the 16 datasets for $k = 10^6$ as it requires more than 45 GB, IFP-growth never runs out of memory as it requires at least $10^3$ times less memory than `dora`, with a memory footprint always much less than 1 GB, even for large values of $k$.

**Table 3.** Comparison of processing times and memory footprints of IFP-growth (`IFPG`) and `dora` for various datasets and values for $k$. Datasets are sorted from the easiest to the most difficult (according to `IFPG`'s processing time for $k = 10^6$). Tuples under dataset names provide the main dataset characteristics: for a tuple $(n, m, c)$, $n$ is the number of data, $m$ is the number of features and $c$ is the number of classes for the target feature.

| Dataset | Algo. | Processing time (s.) | | | | Memory footprint (GB) | | | |
|---|---|---|---|---|---|---|---|---|---|
| $(n, m, c)$ | | $k = 1$ | $k = 10^2$ | $k = 10^4$ | $k = 10^6$ | $k = 1$ | $k = 10^2$ | $k = 10^4$ | $k = 10^6$ |
| german | `IFPG` | 21 | 22 | 23 | 22 | 0.005 | 0.006 | 0.007 | 0.010 |
| (1000,20,2) | `dora` | 110 | 110 | 130 | 150 | 14 | 15 | 16 | 18 |
| lymphography | `IFPG` | 46 | 50 | 73 | 110 | 0.005 | 0.005 | 0.010 | 0.036 |
| (148,18,4) | `dora` | 69 | 80 | 120 | 410 | 14 | 16 | 17 | 27 |
| vehicle | `IFPG` | 130 | 140 | 160 | 160 | 0.006 | 0.007 | 0.008 | 0.36 |
| (846,18,4) | `dora` | 820 | 840 | 880 | ??? | 26 | 26 | 26 | $> 45$ |
| sonar | `IFPG` | 220 | 260 | 320 | 470 | 0.005 | 0.005 | 0.007 | 0.12 |
| (208,60,2) | `dora` | 1300 | 1600 | 2000 | ??? | 38 | 40 | 42 | $> 45$ |
| penbased | `IFPG` | 30 | 38 | 225 | 500 | 0.023 | 0.023 | 0.024 | 0.030 |
| (10992,16,10) | `dora` | 160 | 260 | 5000 | ??? | 15 | 16 | 43 | $> 45$ |
| segment | `IFPG` | 12 | 20 | 110 | 600 | 0.008 | 0.009 | 0.015 | 0.044 |
| (2310,19,7) | `dora` | 43 | 84 | 390 | ??? | 15 | 15 | 22 | $> 45$ |
| spectfheart | `IFPG` | 1800 | 2000 | 2200 | 2300 | 0.005 | 0.006 | 0.007 | 0.13 |
| (267,44,2) | `dora` | ??? | ??? | ??? | ??? | $> 45$ | $> 45$ | $> 45$ | $> 45$ |
| twonorm | `IFPG` | 160 | 170 | 370 | 4600 | 0.019 | 0.019 | 0.020 | 0.10 |
| (7400,21,3) | `dora` | 2800 | 3600 | ??? | ??? | 43 | 43 | $> 45$ | $> 45$ |
| wdbc | `IFPG` | 120 | 300 | 710 | 6330 | 0.007 | 0.007 | 0.008 | 0.12 |
| (569,30,2) | `dora` | 510 | 770 | 1800 | ??? | 18 | 21 | 32 | $> 45$ |
| ring | `IFPG` | 970 | 1100 | 1900 | 6700 | 0.019 | 0.019 | 0.020 | 0.140 |
| (7400,20,2) | `dora` | $> 3h.$ | ??? | ??? | ??? | ??? | $> 45$ | $> 45$ | $> 45$ |
| satimage | `IFPG` | 780 | 1300 | 3700 | $> 3h.$ | 0.026 | 0.026 | 0.028 | ??? |
| (6435,36,7) | `dora` | 4500 | $> 3h.$ | ??? | ??? | 31 | ??? | $> 45$ | $> 45$ |
| ionosphere | `IFPG` | 640 | 2300 | $> 3h.$ | $> 3h.$ | 0.010 | 0.010 | ??? | ??? |
| (351,33,2) | `dora` | 1900 | $> 3h.$ | ??? | ??? | 30 | ??? | $> 45$ | $> 45$ |
| texture | `IFPG` | 3000 | 4500 | $> 3h.$ | $> 3h.$ | 0.025 | 0.025 | ??? | ??? |
| (5500,40,11) | `dora` | $> 3h.$ | $> 3h.$ | ??? | ??? | ??? | ??? | $> 45$ | $> 45$ |
| coil2000 | `IFPG` | $> 3h.$ | $> 3h.$ | $> 3h.$ | $> 3h.$ | ??? | ??? | ??? | ??? |
| (9822,85,2) | `dora` | $> 3h.$ | $> 3h.$ | $> 3h.$ | ??? | ??? | ??? | ??? | $> 45$ |
| spambase | `IFPG` | $> 3h.$ | $> 3h.$ | $> 3h.$ | $> 3h.$ | ??? | ??? | ??? | ??? |
| (4597,57,2) | `dora` | $> 3h.$ | $> 3h.$ | $> 3h.$ | ??? | ??? | ??? | ??? | $> 45$ |
| optdigits | `IFPG` | $> 3h.$ | $> 3h.$ | $> 3h.$ | $> 3h.$ | ??? | ??? | ??? | ??? |
| (5620,64,10) | `dora` | ??? | ??? | ??? | ??? | $> 45$ | $> 45$ | $> 45$ | $> 45$ |
| splice | `IFPG` | $> 3h.$ | $> 3h.$ | $> 3h.$ | $> 3h.$ | ??? | ??? | ??? | ??? |
| (3190,60,3) | `dora` | ??? | ??? | ??? | ??? | $> 45$ | $> 45$ | $> 45$ | $> 45$ |

## 5 Conclusion

In this paper, an efficient algorithm computes entropy of definite feature subsets likewise efficient algorithms exist to compute frequency of frequent itemsets. This algorithm proves to be much faster and scalable than its counterpart based on a levelwise approach. This algorithm is extended to compute mutual information and reliable fraction of information relatively to some target feature. Again, when applied to the discovery of the top-k reliable approximate functional dependencies, this algorithm shows an important gain of time and space efficiency compared to the existing algorithm. While these algorithms have been instanciated on two specific problems, they should be considered as generic algorithmic building blocks enabling to solve various data mining problems relying on entropic measures such as entropy or mutual information. These algorithms can also be easily adapted for computing any other non entropic measures whose expression depends mainly on data partitions. This is the case of the first scores proposed in the context of approximate functional dependencies. The level of performance of these algorithms also enable a more systematic search of sets of non redundant informative subsets of features. Another promising application is the extraction of Bayesian networks from data as these models can be seen as solutions of entropy-based optimization problems.

**Ackowledgements.** This work has been partially supported by the European Interreg Grande Région project “GRONE”.

## References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of 20th International Conference on Very Large Data Bases, pp. 487–499. Morgan Kaufmann, Santiago de Chile (1994)
2. Bayardo, R., Goethals, B., Zaki, M.J. (eds.) Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations. Brighton (2004). http://fimi.ua.ac.be/
3. Dalkilic, M.M., Roberston, E.L.: Information dependencies. In: Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 245–253. ACM, Dallas (2000)
4. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min. Knowl. Discov. **8**(1), 53–87 (2004)
5. Heikinheimo, H., Hinkkanen, E., Mannila, H., Mielikinen, T., Seppnen, J.K.: Finding low-entropy sets and trees from binary data. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p. 350. ACM, San Jose (2007)
6. Heikinheimo, H., Vreeken, J., Siebes, A., Mannila, H.: Low-entropy set selection. In: Proceedings of the SIAM International Conference on Data Mining, pp. 569–580. SIAM, Sparks (2009)
7. Knobbe, A.J., Ho, E.K.Y.: Maximally informative k-itemsets and their efficient discovery. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 237–244. ACM, Philadelphia (2006)

8. Mampaey, M.: Mining non-redundant information-theoretic dependencies between itemsets. In: Bach Pedersen, T., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2010. LNCS, vol. 6263, pp. 130–141. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15105-7_11
9. Mandros, P., Boley, M., Vreeken, J.: Discovering reliable approximate functional dependencies. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 355–363. ACM, Halifax (2017)
10. Pennerath, F.: Fast extraction of locally optimal patterns based on consistent pattern function variations. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010. LNCS (LNAI), vol. 6323, pp. 34–49. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15939-8_3
11. Pennerath, F., Napoli, A.: The model of most informative patterns and its application to knowledge extraction from graph databases. In: Buntine, W., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. LNCS (LNAI), vol. 5782, pp. 205–220. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04174-7_14
12. Romano, S., Bailey, J., Nguyen, V., Verspoor, K.: Standardized mutual information for clustering comparisons: one step further in adjustment for chance. In: Proceedings of the 31st International Conference on Machine Learning, vol. 32, pp. 1143–1151. PMLR, Bejing (2014)
13. Romano, S., Vinh, N.X., Bailey, J., Verspoor, K.: A framework to adjust dependency measure estimates for chance. In: Proceedings of the 2016 SIAM International Conference on Data Mining, pp. 423–431. SIAM, Miami (2016)
14. Vinh, N.X., Epps, J., Bailey, J.: Information theoretic measures for clusterings comparison: variants, properties, normalization and correction for chance. J. Mach. Learn. Res. **11**, 2837–2854 (2010)