Noname manuscript No.
(will be inserted by the editor)

Algorithmic cache of sorted tables for feature selection

Speeding up methods based on consistency and information theory measures

Antonio Arauzo-Azofra · Alfonso Jiménez-Vílchez · José Molina-Baena · María Luque-Rodriguez

Received: date / Accepted: date

Abstract Feature selection is a mechanism used in Machine Learning to reduce the complexity and improve the speed of the learning process by using a subset of features from the data set. There are several measures which are used to assign a score to a subset of features and, therefore, are able to compare them and decide which one is the best. The bottle neck of consistence measures is having the information of the different examples available to check their class by groups. To handle it, this paper proposes the concept of an algorithmic cache, which stores sorted tables to speed up the access to example information. The work carries out an empirical study using 34 real-world data sets and four representative search strategies combined with different table caching strategies and three sorting methods. The experiments calculate four different consistency and one information measures, showing that the proposed sorted tables cache reduces computation time and it is competitive with hash table structures.

Keywords Feature selection · Attribute selection · Consistency measures · Information theory · Data Reduction · Algorithmic cache

1 Introduction

Feature selection (FS) is a common dimensionality reduction technique used in Machine Learning. Its objective is to choose a subset of the available features

This research is partially supported by projects: TIN2013-47210-P of the Ministerio de Economía y Competitividad (Spain), P12-TIC-2958 and TIC1582 of the Consejeria de Economia, Innovacion, Ciencia y Empleo from Junta de Andalucia (Spain).

A. Arauzo-Azofra · A. Jiménez-Vílchez · J. Molina-Baena · M. Luque-Rodriguez
School of Engineering Sciences
Universidad de Cordoba (Spain)
Fax: +34-957-218550
E-mail: mluque (at) uco.es

which, when used, allows the learning process to obtain better or acceptable results. Using less features speeds up the learning process and increases its comprehensibility.

Having n as the number of features in a data set, there are 2^n possible subsets (including the full set and the empty set). Given that large amount of possibilities, one way of addressing FS is by scoring features individually and selecting them based on these scores (Arauzo-Azofra et al, 2011). This is the fastest approach but it is worse at considering multi-feature dependencies. Advanced techniques combine this approach with clustering (Song et al, 2013) to perform a more effective feature selection.

Other approaches analyze all features at once. Some examples are spectral feature selection (Zhao and Liu, 2007) and graph-based feature selection (Chen et al, 2011). These approaches are specially useful in feature selection when many dimensions are considered, like in information retrieval (Bharti and Singh, 2015), ranking problems (Geng et al, 2007) and clustering problems (Gui et al, 2017). Furthermore, sparse feature selection (Liu and Zhang, 2016) is specially suited for very high dimensional data.

In any case, addressing FS as a search problem also allows to consider dependencies of several features, since feature sets are evaluated together as a whole. This approach is adequate for multi-class classification problems and is considered as the state of the art, for example, in time series prediction (Koprinska et al, 2015). Methods based on this approach use some strategies to search among these sets and some measures to compare them and choose the best one, as shown in Fig. 1. In this modularized approach, the measure is computed many times, so its efficiency is very important.

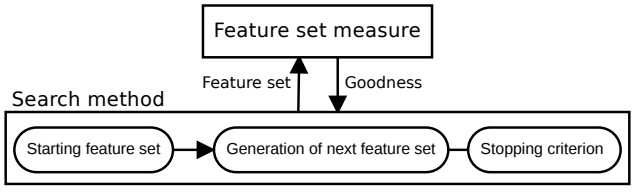


Fig. 1 Feature selection approached with a search process

Many measures have been proposed to evaluate feature sets. Considering the supervised learning problem of classification, Dash and Liu (2003) identified the following types:

- **Accuracy**: the accuracy of a learning algorithm using the selected feature set against test data.
- **Dependence**: quantifies the ability to predict the class with some correlation measures.
- **Distance**: the difference between the class conditional probabilities when using the selected features.
- **Information**: information gained according to Shannon's Information Theory.

– **Consistency**: proximity to the nonexistence of examples which share the same feature values but belong to different classes.

Each type of measure has its advantages and disadvantages. For example, using the accuracy of the same learning algorithm that will later be used in the learning process as a measure is known as the wrapper approach (Kohavi and John, 1997). It is considered to be the most exact. However, it could be slow or even non-applicable if the learning algorithm cannot handle a large number of features. On the other hand, consistency measures are currently used (Onan, 2015) because they are fast and can provide very good results.

There are several consistency measures but, in all of them, its calculation process is similar. Examples (data set instances) with identical values for the selected features form groups and the class distribution in each group is analyzed somehow. This process can calculate information measures as well. To make this process faster, the examples can be stored in hash tables (Arauzo-Azofra et al, 2008).

Recently, Shin and Miyazaki (2016) proposed CWC, a highly efficient algorithm for FS. To calculate the binary consistency measure for CWC, they compared it by using hash tables with sorted tables as data structures. Their paper clearly showed that sorted tables are faster when used inside the CWC algorithm. This efficient performance is partly due to CWC removing features in a predictable order. This allows the sorted table to be reused or resorted just by one feature, heavily reducing computing time. However, it is not known how these data structures compare in practice while calculating consistency in a more general search context.

Motivated by the idea of reusing sorted tables with the goal of improving performance, not only for CWC, but for any method that repeatedly evaluates consistency or information theory measures as well, e.g. (Zheng and Wang, 2018), in this paper we aim to:

– Compare hash tables and sorted tables with other search approaches to FS.
– Generalize the reuse of sorted tables, saving them in an algorithmic cache system, to improve the performance of FS methods that use different search strategies.
– Evaluate how this improvement in performance depends on the number of features, the consistency or information measure used and the search algorithm applied.
– Study several strategies to choose the order in which features will be used to sort the table to improve its re-usability.

Besides feature selection, feature set evaluation is also used in another contexts like discretization, or instance selection (García et al, 2016).

The rest of the paper is organized as follows. Section 2 explains the measures, how they can be calculated and how they can be optimized. Section 3 describes the cache system proposal and its variations. Sections 4 and 5 describe the empirical study and its results, respectively. Finally, the conclusions are summarized in section 6.

2 Calculation of feature set evaluation measures

This section, first, introduces the measures, and then, explains how to implement fast algorithms to calculate them. Table 1 introduces the notation and symbols used along the paper.

Table 1 Notation

Symbol	Definition
$F = \{f_1, f_2, ..., f_n\}$	The set of all features considered in the data set.
$S \subseteq F$	A set of selected features.
$\mathcal{P}(F)$	Power-set, set of all possible subsets of F.
$n = \|F\|$	The number of features available.
$s = \|S\|$	The number of selected features.
f_j	The finite set of values the j^{th} feature may take.
$E = \{e_1, e_2, ..., e_m\}$	The set of all examples (data instances) in the data set.
$m = \|E\|$	The number of examples in the data set.
$e_i = \langle v_{i1}, v_{i2}, ..., v_{in}\rangle, v_{ij} \in f_j$	An example, the feature values of an observed object.
Class or C	The set of possible class values,
$v_{i,(n+1)}$	The class assigned to example i.
$T = \langle e_1, e_2, ..., e_m\rangle$	A table, a sequence of all examples in the data set.
$\Pi_S(E)$	The relational projection of examples in T onto S features (remove columns of the other features)
$O = \langle f_{\sigma(1)}, f_{\sigma(2)}, ..., f_{\sigma(s)}\rangle, s \leq n$	Order index, a sequence of features.
$\sigma(j)$	A permutation or a partial permutation that maps the position j in an ordered sequence to the index of an element (feature).
$T_O = \langle e_{\pi(1)}, e_{\pi(2)}, ..., e_{\pi(m)}\rangle$	An ordered table of all examples in the data set, sorted considering the values of the features in O.
$\pi(i)$	A permutation that maps the position i in an ordered sequence to the index of an element (example).
$\langle ...\rangle^\frown\langle ...\rangle$	Concatenation of two sequences.
Variable'	Modified state of the Variable after step.
$tail\langle ...\rangle$	Sequence without the first element.

2.1 Consistency measures

According to Cambridge English Dictionary, consistency is "the quality of always behaving or performing in a similar way". In feature selection literature, consistency refers to the absence of examples (data set instances) that share the same selected feature values but belong to different classes. These examples are likely to confuse classifiers so, the lower the presence of this kind of examples, the better.

In this way, consistency can be used to evaluate and compare feature sets. Looking at equation (1), if the projection operation from relational algebra

applied to a data set (E) with just some selected features (S) is more consistent than the projection with another set of features(R), the first set is considered a better feature set.

$$S \text{ is better than } R \leftrightarrow \text{Consistency}\big(\Pi_S(E)\big) > \text{Consistency}\big(\Pi_R(E)\big) \quad (1)$$

As an illustration, Figure 2a defines a dataset of ten examples, which belong either to class A or B, characterized by features f_1, f_2 and f_3. If only f_1 and f_3 are selected it remains consistent (Figure 2b). However, if only f_2 and f_3 are selected, it becomes inconsistent (Figure 2c). All these measures only handle discrete values; if continuous values are present, they need to be discretized first. By applying discretization, these measures could also be used for feature selection in approximation or regression but they are defined for classification problems.

Using this idea, several measures estimate the distance that separates a data set from the state of being fully consistent. An intuitive description of some of them follows, for a formal definition and a discussion about their properties see (Shin et al, 2011).

The consistency measures calculated in this paper are:

– **Binary consistency measure (BIN)**: First used in FOCUS (Almuallim and Dietterich, 1991), as a stop criteria (called "Sufficiency test"), this is the simplest consistency measure.

$$\text{BIN} = \begin{cases} 0 & \text{if data set contains any inconsistency} \\ 1 & \text{if data set is fully consistent} \end{cases} \quad (2)$$

– **Rough Sets measure (RSM)**: Proposed by Pawlak (1982), it counts how many examples are in the positive region (examples with the same values and assigned to the same class), disregarding the rest of examples as inconsistent. This value can be regarded as an estimate to the probability of an example belonging to a consistent group.

$$\text{RSM} = \frac{\text{number of fully consistent examples}}{\text{number of examples}} \quad (3)$$

– **Inconsistent Examples measure (IE)**: Proposed by Dash and Liu (2003). In addition to all the RSM consistent examples, IE considers consistent examples those that belong to the majority class, in each group of examples with the same values for the selected features. The rest are considered inconsistent.

$$\text{IE} = \frac{\text{number of inconsistent examples}}{\text{number of examples}} \quad (4)$$

– **Inconsistent Example Pairs measure (IEP)**: Used as an heuristic to guide the search in a greedy algorithm presented with FOCUS-2 (Almuallim and Dietterich, 1994). Defining a conflict as a pair of examples with

the same values but different class, IEP counts how many of such pairs can be formed in the data set.

$$\text{IEP} = \frac{\text{number of pairs of inconsistent examples}}{\text{number of pairs of examples}} \tag{5}$$

IE and IEP are measuring inconsistency. As it is measured in range $[0, 1]$, being 0 the absence of inconsistency, it can be converted to consistency with:

$$\text{Consistency} = 1 - \text{Inconsistency} \tag{6}$$

2.2 Information measures

There are several evaluation measures for feature sets based on the Information Theory of Shannon (Cover and Thomas, 1991). For this reason, they are usually identified in its own category (Dash and Liu, 2003; Molina et al, 2002). Nevertheless, they can also be considered as consistency measures according to the definition and properties identified by Shin et al (2011). Thanks to these properties, all these measures can benefit from a similar implementation with a first step analyzing the set of examples. In this way, they can benefit from the algorithmic cache proposed in this paper. The cache is tested with the following representative of these measures:

- **Information measure (INF)**: Widely used in FS (Qian and Shu, 2015), it is defined as the difference between the entropy of the class and the entropy of the class conditioned to know the values of the selected features.

$$I(C, S) = H(C) - H(C|S) \tag{7}$$

2.3 Data structures for fast consistency evaluation

In order to measure consistency, it is necessary to check which class each example belongs to. This information can be stored in two data structures:

- **Hash table**. This kind of structure, also known as dictionary, is constructed by going through the list of examples and, for each example, storing the class it belongs to in a list using their feature values as a key for quick grouping of identical examples (considering the selected features). As an illustration, Figures 2b and 2c are hash tables created from the set of examples of table from Figure 2a, selecting features f_1 and f_3 in the first case and f_2 and f_3 in the second.
- **Sorted table**. This structure is a regular table with one row for each example, which has as many columns as features plus an extra column for the class value. It is sorted by the feature values so that elements with the same values are contiguous. Figures 2d and 2e are the sorted tables created from the set of examples of Figure 2a, selected and ordered only by features f_1 and f_3 in the first case and f_2 and f_3 in the second.

f_1	f_2	f_3	Class
1	1	1	A
3	2	2	A
3	3	2	A
3	2	2	A
2	1	1	A
1	2	2	B
2	3	3	B
2	2	2	B
3	3	3	B
1	1	2	B

(a) Example of elements table

f_1	f_3	Classes
1	1	[A]
3	2	[A,A,A]
2	1	[A]
1	2	[B,B]
2	3	[B]
2	2	[B]
3	3	[B]

(b) Consistent hash table

f_2	f_3	Classes
1	1	[A, A]
2	**2**	**[A,A,B,B]**
3	2	[A]
3	3	[B,B]
1	2	[B]

(c) Inconsistent hash table

f_1	f_3	Class
1	1	A
1	2	B
1	2	B
2	1	A
2	2	B
2	3	B
3	2	A
3	2	A
3	2	A
3	3	B

(d) Consistent sorted table

f_2	f_3	Class
1	1	A
1	1	A
1	2	B
2	**2**	**A**
2	**2**	**A**
2	**2**	**B**
2	**2**	**B**
3	2	A
3	3	B
3	3	B

(e) Inconsistent sorted table

Fig. 2 Example with data structures for fast consistency evaluation

By using hash tables, the lack of consistency is detected when, for a given key, the stored list contains different classes (see in Figure 2c the inconsistency marked in bold) whereas, by using sorted tables, the lack of consistency is detected when, going through the table rows, two examples have the same values in their feature columns but different values in their class (see Figure 2e, sorted by f_2 and f_3, with the inconsistency marked in bold).

In both cases, having the data structure built, the cost of calculating any of the identified measures is just the cost of going through the data structure, whose size is the number of examples by the number of selected features. Therefore, the efficiency of calculating the measures is $\mathcal{O}(s\,m)$, which seems hard to reduce if all examples are being considered, plus the efficiency of building the data structure. Then, having the data structure available is the bottle neck and here is where the efforts should be made.

- Building the hash table for measure evaluation has a worst case efficiency of $\mathcal{O}(s\,m^2)$ and a best case efficiency of $\mathcal{O}(s\,m)$. The expected efficiency when the hash table has enough space available is near the best case.
- Sorting the table for measure evaluation using Tim sort Auger et al (2015) has a worst case efficiency of $\mathcal{O}(s\,m\log m)$ and a best case efficiency of $\mathcal{O}(s\,m)$. The sort algorithm chosen has an expected efficiency near the best case when the tables are partially ordered.

This article focus on avoiding the cost of re-sorting for the sort-based algorithms or, at least, reduce its cost by reusing previously sorted tables as much as possible. If new features are added to (or removed from) the set of selected features, the hash table needs to be completely rebuilt. However, when using sorted tables, if the selected features are part of the indexes by which the table is already sorted, there is a chance that we can skip or reduce the example sorting phase by reusing the table.

Our cache proposals increase the probabilities of finding a table which can be reused without resorting it or, at least, provide a partially reusable table in which a few resorting operations must be performed. Nevertheless, adding the required table storing, copying and selection operations to the process may suppose an extra cost which, according to our experiments, is worth it.

Our sorting strategies follow two different philosophies:

- To keep the table fully sorted by performing as less sorting operations as possible
- To reduce the amount of sorting operations by sorting the table only by the needed features.

3 Algorithmic cache of sorted tables

In the computing context, a cache is a hardware or software component that stores some data so that they can be served faster in future requests. For example, a CPU cache memory keeps some parts of the main memory that are frequently accessed and a web browser cache keeps web pages to avoid having to download them again. Cache systems are used between different system components (CPU $\leftrightarrow$ memory, web browser $\leftrightarrow$ Internet web servers). The cache memory size is usually very limited compared with the large space being cached (main memory, Internet). For this reason, any cache system faces uncertainty about which data to store and which data to drop. To alleviate this, the cache systems try to make the best predictions to improve its hit ratio. On the other hand, there are cache-aware algorithms designed to take profit of a known underlying cache (Kowarschik and Weiß, 2003) and cache-oblivious algorithms designed to improve performance over any cache system (Frigo et al, 2012).

Cache systems are also used to reduce computing effort by avoiding the repetition of some tasks. For example, there are several plug-ins for well known web server frameworks that keep the generated web pages to avoid recreating them from the data base. A similar idea is used in evolutionary algorithms that keep the evaluation result of an individual to avoid reevaluating it, as it is done, for example, in DEAP (Fortin et al, 2012). In these cases, the space being cached is not a physical memory but a virtual space of solutions that can even be larger and more costly to access.

Within an algorithm, if you save some limited data from a large space of computation result data, which may be used again but there is uncertainty about its reuse, this can be modeled as a cache system. If this is studied inside

the logic structure of an algorithm, we think it can be called an *algorithmic cache*.

In FS, modularization (see Fig. 1) allows specialization by focusing on some parts of the algorithms. Besides, it allows creating many different algorithms by means of the combination of different search methods and measures (Arauzo-Azofra et al, 2008). However, a module may have to repeat some computations or recreate data structures. This happens with the feature set evaluation module. Still, with search algorithms where no feature set is evaluated twice, some data structures can be reused when evaluating other feature sets.

Instead of analyzing the whole set of examples and sorting them by the selected features on each iteration, CWC (Shin and Miyazaki, 2016) stored the sorted table used in the previous iteration so that it could be reused, if possible. Inspired by this idea, we propose to define an algorithmic cache keeping several sorted tables, used for consistency evaluation of feature sets. The evaluation module can be combined with several search strategies. This introduces uncertainty about which feature sets will be evaluated in the future. Besides, the space is limited, as saving all the orders of the examples needed for the possible 2^n feature sets is not feasible.

Next, the algorithmic cache proposed is structured in two areas: table caching strategy (how many and what tables are stored) and sorting method (by which features the table is sorted).

3.1 Table caching strategies

The cache system is better described with an object-oriented design, as it will be an object that stores previous knowledge in instance variables and has a method to compute the consistency measure. Figure 3 shows the class diagram of the cache system.

Two specializations are considered: keeping a number of tables given by a parameter (k) and adapting to algorithms that follow an order based on the levels of the search space (sets with the same number of features).

Algorithm 1 describes the general process of the system. First, it selects the best cached table. Then, it completes the sort process as needed and updates the cache. Finally, it evaluates the measure using the sorted table.

Algorithm 1 evaluate(S)

$\langle O_{\text{best}}, \pi_{\text{best}} \rangle \leftarrow$ selectCachedTable(S) ▷ $\mathcal{O}(1)$
if suitability(O, S) $< |S|$ **then** ▷ If the table needs some sorting
  $\langle O_{\text{new}}, \pi_{\text{new}} \rangle \leftarrow$ sortMethod ($\langle O_{\text{best}}, \pi_{\text{best}} \rangle, S, E$) ▷ $\mathcal{O}(s\,m \log m)$
  updateSortedTables($\langle O_{\text{new}}, \pi_{\text{new}} \rangle$) ▷ $\mathcal{O}(1)$
for e_i in $T_{O_{\text{new}}}$ **do** ▷ Calculate one of the measures $\mathcal{O}(s\,m)$
  — Compute measure —
return — computed measure —

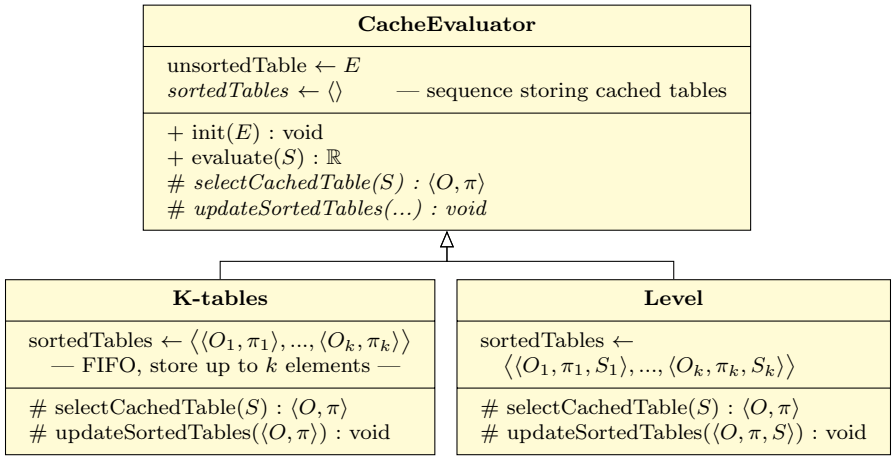


Fig. 3 UML2 Class diagram of the cache evaluators

3.1.1 k-tables

This cache strategy stores up to k tables using the First-Input-First-Output (FIFO) approach. Algorithm 2 describes how the most suitable cached table is selected. It uses the functions defined in equations (8) and (9).

$$\begin{aligned} \text{suitability} &: \mathcal{P}(O) \times \mathcal{P}(F) \to \mathbb{Z} \\ \text{suitability}(O, S) &= \max_{1 \leqslant i \leqslant j} \text{score}(i, O, S) \end{aligned} \tag{8}$$

$$\begin{aligned} \text{score} &: \mathbb{Z} \times \mathcal{P}(O) \times \mathcal{P}(F) \to \mathbb{Z} \\ \text{score}(i, O, S) &= \begin{cases} i & \text{if} \quad \forall_{j \leqslant i} O_j \in S \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{9}$$

Algorithm 2 selectCachedTable(S) ▷ K-tables object

```
bestSuitability ← -1
for ⟨O, π⟩ in sortedTables do                ▷ Choose the best table to reuse
    suitGrade ← suitability(O, S)
    if suitGrade > bestSuitability then
        O_best ← 0
        π_best ← π
        bestSuitability ← suitGrade
return ⟨O_best, π_best⟩
```

Algorithm 3 describes how the cache system is updated. As k increases, the probability of finding a suitable, reusable table increases but, logically, memory usage grows as well. Moreover, extra computation time is needed to choose the most suitable one among the cached tables.

Algorithm 3 updateSortedTables($\langle O, \pi \rangle$) ▷ K-tables object

```
if |sortedTables| ⩽ k then
    sortedTables' ← tail sortedTables
sortedTables" ← sortedTables' ⌢⟨⟨O, π⟩⟩
```

3.1.2 Level

Sequential algorithms such as SFS or SBS (see Section 4.4) evaluate many feature sets generated by adding a feature to (or removing a feature from) the current set of selected features. Consequently, several candidate sets with the same cardinality are evaluated before confirming which feature to add (or to remove).

This strategy consists in storing only the sorted table of the current set of selected features (parent set) and the sorted tables of the candidate sets that are being evaluated (child sets). The child sets are those whose selected features are of the same size, with one feature more or one feature less than the parent. In this way, when a feature is definitely added or removed from the set of selected features, the table used in the evaluation of such set is already stored in memory. Algorithm 4 describes how the best table is selected in this specialization.

Algorithm 4 selectCachedTable(S) ▷ Level object

```
⟨O_parent, π_parent, S_parent⟩ ← sortedTables[0]   ▷ The first cached table is the parent
if abs(|S_parent| − |S|) > 1 then                   ▷ If the algorithm has jumped to the next level
    bestSuitability ← -1
    for ⟨O, π, S_child⟩ in sortedTables[1:] do      ▷ Search for the new parent table
        suitGrade ← suitability(S_child, S)
        if suitGrade > bestSuitability then
            O_parent ← O
            π_parent ← π
            S_parent ← S_child
            bestSuitability ← suitGrade
    sortedTables' ← ⟨⟨O_parent, π_parent, S_parent⟩⟩   ▷ Restart cache
return ⟨O_parent, π_parent⟩
```

Algorithm 5 describes how the cache system is updated in the typical case (storing new table). Note that the cache gets reset when algorithm 4 detects a change in level.

Algorithm 5 updateSortedTables($\langle O, \pi, S \rangle$) ▷ Level object

```
sortedTables' ← sortedTables ⌢⟨⟨O, π, S⟩⟩
```

3.2 Sorting methods

The order index sequence O denotes the order in which features are considered when sorting the examples in the table. In this way, considering that it exists a total order relation among the example values (numerical, alphabetical or any other), the order relation between two elements of the table is defined by equations (10-12):

$$e_i < e_j \leftrightarrow \exists f_{\sigma(k)} \in O : v_{i,\sigma(k)} < v_{j\sigma(k)} \wedge \forall m < k, v_{i,\sigma(m)} = v_{j,\sigma(m)} \tag{10}$$

$$e_i = e_j \leftrightarrow \forall f_{\sigma(k)} \in O, v_{i,\sigma(k)} = v_{j\sigma(k)} \tag{11}$$

$$e_i > e_j \leftrightarrow e_i \not\leq e_j \tag{12}$$

If the number of elements in O is the total number of features, we can stand that the table is fully ordered. This may be a desirable property, as the chances that the table can be reused increase due to the fact that the table is already ordered by all the features (selected or not).

The sorting algorithm will arrange examples in the sorted table (T_O) according to the order relation defined above (this is $\forall i, j \in \{1..m\}, i < j \leftrightarrow e_{\pi(i)} < e_{\pi(j)}$). The algorithm used is TimSort (Auger et al, 2015), implemented in Python standard library. It is an adequate algorithm because it is stable (it allows sorting by one feature while keeping the table ordered by some others and it is more efficient in partially ordered sequences, as it is the case with previously ordered tables by other features.

With this algorithm, a sort operation denoted by sort(f_i), sorting a table ordered by $O = \langle f_\sigma(1), ... \rangle$ considering now f_i, will keep the table ordered by the features behind it i.e. T'_O will have examples ordered by $O' = \langle f_i, f_\sigma(1), ... \rangle$.

For convenience, in the following examples the features will be denoted by letters, using the mapping of (13). Table 2 is provided to illustrate these examples.

$$\begin{aligned} \text{Rename} &: F \rightarrow \text{LowerCaseLetters} \\ \text{Rename}(f_i) &= i^{\text{th}} \text{ lower case letter in alphabetical order} \end{aligned} \tag{13}$$

The ordered table (T_O) can be used to evaluate the set of selected features (S) if all the selected features are contained in the index of order of the table and there is no an unselected feature before any selected feature, i.e. if equation (14) is true. Otherwise, it will be necessary to sort it again, to make the initial order positions correspond to the selected features.

$$\forall f_{\sigma(j)} \in S, \left(f_{\sigma(j)} \in O \wedge \nexists f_{\sigma(k)} \in O \setminus S : k < j \right) \tag{14}$$

In this way, if a feature f_j is removed from the index of order, the table will not be ordered by the features behind it, i.e. if $f_j = b$ and $O = \langle a, b, c \rangle$ then $O' = \langle a \rangle$

Table 2 Example of sorted table T_O

	a	b	c	d	e	f	g	h	Class
$e_{\pi(1)}$	9	1	1	7	1	6	2	1	A
$e_{\pi(2)}$	2	1	2	4	3	5	0	4	B
$e_{\pi(3)}$	5	4	2	7	2	2	1	2	A
$e_{\pi(4)}$	6	5	3	2	4	2	2	5	B
$e_{\pi(5)}$	2	2	4	7	5	1	3	2	B
$e_{\pi(6)}$	8	7	4	7	3	1	2	9	B
$e_{\pi(7)}$	9	6	5	1	2	3	1	4	A
$e_{\pi(8)}$	9	3	5	1	5	4	1	2	A
$e_{\pi(9)}$	2	8	5	1	7	4	5	2	A
$e_{\pi(10)}$	1	8	6	0	4	7	3	2	B
$e_{\pi(11)}$	6	7	7	2	7	8	2	1	A
$e_{\pi(12)}$	4	4	9	6	4	0	0	5	A

$F = \{a, b, c, d, e, f, g, h\}$ $\quad O = \langle c, d, f, h, g, a, e, b\rangle$

3.2.1 The Lazy Sort Method

The aim of this proposal is to reorder the table only by the selected features, reusing the current order when possible. The order of the rest of features is undetermined.

As an example, the sort operations needed to be able to use the sorted table represented by Table 2 for $S = \{c, a, b\}$ using the lazy sort method would be:

$$\begin{aligned} \langle \mathbf{c}, d, f, h, g, \mathbf{a}, e, \mathbf{b}\rangle &\xrightarrow[\text{sort}(a)]{} \langle \mathbf{a}, \mathbf{c}, d, f, h, g\rangle \\ \langle \mathbf{a}, \mathbf{c}, d, f, h, g\rangle &\xrightarrow[\text{sort}(b)]{} \langle \mathbf{b}, \mathbf{a}, \mathbf{c}, d, f, h, g\rangle \end{aligned} \tag{15}$$

3.2.2 Basic Full Sort Method

This is a simple proposal: when the table cannot be reused, first, sort the table by the unselected features and then the selected features so that it remains fully ordered.

3.2.3 Smart Full Sort Method

The smart full sort method tries to reuse the current order to perform as few sort operations as possible while keeping the table fully ordered. If $f_{\sigma(j)}$ is the first selected feature which is preceded by an unselected feature in the order index (16):

$$f_{\sigma(j)} \in O, f_{\sigma(j)} \in S, f_{\sigma(j-1)} \notin S \wedge \nexists f_{\sigma(i)} \in O : \big(i < j \wedge f_{\sigma(i)} \in S \wedge f_{\sigma(i-1)} \notin S\big) \tag{16}$$

We can find the following situations:

– If there is no selected feature preceded by an unselected feature ($\nexists f_{\sigma(j)}$) and the table needs to be resorted, it means that the table is partially

ordered and it has not been sorted by one or more of the selected features. The smart method will sort it by the selected features which are not present in O.

For example, the sort operations needed to be able to use the sorted table whose order is represented by $O = \langle e, b, a, d, f, h, g \rangle$ for $S = \{c, e, b\}$ using the smart sort method would be:

$$\langle \mathbf{e}, \mathbf{b}, a, d, h, g, f \rangle \xrightarrow[\text{sort}(c)]{} \langle \mathbf{c}, \mathbf{e}, \mathbf{b}, a, d, h, g, f \rangle \tag{17}$$

– If $f_{\sigma(j)}$ exists and the following features are all selected features, that is, $\forall\, f_{\sigma(k)} \in \langle f_{\sigma(j)}, ..., f_{\sigma(n)} \rangle, f_{\sigma(k)} \in S$, the smart full sort method will sort the table by the features in the interval $\langle f_{\sigma(j)}, ..., f_{\sigma(n)} \rangle$.
For instance, the sort operations needed to be able to use the sorted table represented by Table 2 for $S = \{c, e, b\}$ using the smart sort method would be:

$$\begin{aligned} \langle \mathbf{c}, d, f, h, g, a, \mathbf{e}, \mathbf{b} \rangle &\xrightarrow[\text{sort}(e)]{} \langle \mathbf{e}, \mathbf{c}, d, f, h, g, a \rangle \\ \langle \mathbf{e}, \mathbf{c}, d, f, h, g, a \rangle &\xrightarrow[\text{sort}(b)]{} \langle \mathbf{b}, \mathbf{e}, \mathbf{c}, d, f, h, g, a \rangle \end{aligned} \tag{18}$$

– If $f_{\sigma(j)}$ exists and there is at least one unselected feature among the following selected features, that is, $\exists\, f_{\sigma(k)} \in \langle f_{\sigma(j)}, ..., f_{\sigma(n)} \rangle, f_{\sigma(k)} \notin S$, the procedure will depend on whether the first index, $f_{\sigma(0)}$, is a selected feature:
  – If $f_{\sigma(0)} \notin S$ smart sort method will sort the table by the unselected features in the interval $\langle f_{\sigma(j)}, ..., f_{\sigma(n)} \rangle$ and then by the selected features in the same interval. This way the order $\langle f_{\sigma(0)}, f_{\sigma(j-1)} \rangle$ is reused.
  As an example, the sort operations needed to be able to use the sorted table represented by Table 2 for $S = \{g, e, b\}$ using the smart sort method would be:

$$\begin{aligned} \langle c, d, f, h, \mathbf{g}, a, \mathbf{e}, \mathbf{b} \rangle &\xrightarrow[\text{sort}(a)]{} \langle a, c, d, f, h, \mathbf{g} \rangle \\ \langle a, c, d, f, h, \mathbf{g} \rangle &\xrightarrow[\text{sort}(g)]{} \langle \mathbf{g}, a, c, d, f, h \rangle \\ \langle \mathbf{g}, a, c, d, f, h \rangle &\xrightarrow[\text{sort}(e)]{} \langle \mathbf{e}, \mathbf{g}, a, c, d, f, h \rangle \\ \langle \mathbf{e}, \mathbf{g}, a, c, d, f, h \rangle &\xrightarrow[\text{sort}(b)]{} \langle \mathbf{b}, \mathbf{e}, \mathbf{g}, a, c, d, f, h \rangle \end{aligned} \tag{19}$$

  – If $f_\sigma(0) \in S$, we will need to sort the entire table the same way the basic full sort method does.

4 Empirical study

The main goal of this empirical study is to confirm if the proposed cache system improves the performance of consistency measures computation, in the context of a feature selection search. In addition, we want to know which of the strategies applied in the cache system perform better. The options are summarized in the following schemata.

Cache strategies
- **1-table**: caches the last used table.
- ***k*-tables**: caches the last k used tables.
- **level**: caches all tables with the same number of selected features.

Sort methods
- **Basic**: sorts by all features to keep the table fully ordered.
- **Smart**: performs the least sort operations to keep the table fully ordered.
- **Lazy**: performs the least sort operations to keep the table ordered by the selected features.

In order to formally answer those questions, the following hypotheses are tested:

- **H1**. *Hash tables perform faster than sorted tables without cache.* Our intuition from previous experience is that FS algorithms based on hash tables calculate the measures faster. We aim at testing whether this is generally true or it depends on the size of the feature set.
- **H2**. When using sorted tables, *using cache increases performance.* As the sorted table can sometimes be reused when using cache, we expect this method to perform faster.
- **H3**. *Having more cached tables increases performance.* We expect that the benefits of increasing table reuse chances overcome the cost of using multiple cache tables, up to a certain point. As we deal with more tables, there are more table copy operations and the number of tables to look up for the most suitable one to reuse increases and so does processing time.
- **H4**. *Level cache strategy is the best cache strategy for sequential search algorithms.* As it is specifically designed for this kind of algorithms, we expect its performance to be better than the other generic cache strategies.
- **H5**. *Some sort methods perform better than the others.* We compare the different sorting options available when using sorted tables with cache. We expect to improve performance of the cache system.
- **H6**. *The best sorted table cache implementation performs better than hash tables.* Sorted tables have already been proved faster in CWC algorithm (Shin and Miyazaki, 2016), which uses a list of features sequentially. We want to know if this can be generalized to other FS methods with different search strategies.

4.1 Experimental design

Two types of experiments have been designed. The first type of experiment tests the measures independently. For a given dataset, n feature sets of each size are generated and the time taken by the measure is recorded. The empty set and the set with all features are discarded as non-representative.

The second type of experiment tests the measures inside the search process. We have performed a set of feature selection experiments combining the sort and cache strategies proposed with different data sets, search algorithms and consistency measures. A full factorial experimental design has been applied. The 15 implementations for each of the 5 consistency measures, created by combining the five cache strategies (including using no cache, 10-tables and 100-tables) with the three sort methods, are applied with the 4 search strategies identified below over the 34 data sets described below. This leads to 10200 feature selection scenarios. In order to obtain more reliable results, a 10-fold cross validation has been applied in every scenario run. The times considered are the averages of the ten runs. Only the time taken for feature selection is recorded, discarding previous table loading and splitting times. The table with all data from these experiments is published in an OSF project: `https://osf.io/zkhnr/`.

4.2 Computing environment

The code was written in Python using Orange Data Mining Toolbox (Demšar et al, 2013). This constitutes a stable and extensively used platform, assuring that the code and data structures used are well proven and optimized. To obtain the most accurate results, depending on the algorithm and not on implementation inefficiencies, our code has been passed through Pylint code analyzer and several revisions with Python profilers (Lanaro, 2013): *cProfile* and *line_profiler* (Kern, 2016). The source code can be provided upon request.

The hardware platform has been a cluster consisting of 8 identical Intel Xeon E5420 (2.50GHz, 12 MB Cache) nodes, with 8 GB RAM each. Each execution thread has had its own execution core and no other processes have been run in those machines during the execution phase in order to reduce the influence of external factors and keep the execution conditions between threads as similar as possible .

4.3 Data sets

In this experimentation, we have used diverse data sets —with different numbers of classes, examples and features; and different type of features. They are taken from these public repositories: UCI (Newman and Merz, 1998), ESL (Hastie et al, 2001) and Org (Demšar et al, 2013). The sample is composed by

Table 3 Data sets used

Dataset	Examples	Features	Type of features	Classes
Adult	32561	14	Mixed	2
Anneal	898	38	Mixed	5
Audiology	226	69	Discrete	24
Balance-Scale	625	4	Discrete	3
Breast-cancer	286	9	Mixed	2
Bupa	345	6	Continuous	2
Car	1728	6	Discrete	4
Credit	690	15	Mixed	2
Echocardiogram	131	10	Mixed	2
Horse-colic	368	26	Mixed	2
House-votes84	435	16	Discrete	2
Ionosphere	351	32	Continuous	2
Iris	150	4	Continuous	3
Labor-neg	57	16	Mixed	2
Led24-10000	10000	24	Discrete	10
Lung-Cancer	32	56	Discrete	3
Lymphography	148	18	Discrete	4
Mushrooms	8416	22	Discrete	2
Parity3+3	500	12	Discrete	2
Pima	768	8	Continuous	2
Post-operative	90	8	Mixed	3
Primary-tumor	339	17	Discrete	21
Promoters	106	57	Discrete	2
Saheart	462	9	Mixed	2
Shuttle-landing-control	253	6	Discrete	2
Splice	3190	60	Discrete	3
Tic-tac-toe	958	9	Discrete	2
Vehicle	846	18	Continuous	4
Vowel	90	10	Continuous	11
Wdbc	569	20	Continuous	2
Wine	178	13	Continuous	3
Yeast	1484	8	Continuous	10
Yeast-class-RPR	186	79	Continuous	3
Zoo	101	16	Discrete	7

Table 4 Data set used in the high-dimensionality experiment

Dataset	Examples	Features	Type of features	Classes
CNAE-9	1080	857	Discrete	9

34 regular data sets and a high-dimensional data set, used to study the influence of a large number of features. Their characteristics are listed in Tables 3 and 4:

- **Dataset**: Data set name.
- **Examples**: Number of examples (tuples) in the data set.
- **Features**: Number of features.
- **Type of features**: they can be: all discrete, all continuous or mixed, if both types of features appear.
- **Classes**: Number of classes.

Consistency measures can only handle discrete features. Although discretization should be fitted to each data set and learning algorithm, as a compromise to obtain something which is common for all our experiments, continuous features have been discretized using six equal width intervals, as this method has worked well in previous experiments with these data sets (Arauzo-Azofra et al, 2011).

4.4 Feature selection search algorithm

We have chosen four different well-known search strategies to study the influence of our proposal in their performance. These are common strategies used inside many other recent feature subset selection methods, so we think they are appropriate to cover diversity and generalize the results. For each, we have chosen a representative algorithm:

- **Incremental**: the set of selected features starts from the empty set and then it expands by including features. The algorithm chosen is Sequential Forward Selection (SFS), introduced by Whitney (1971).
- **Decremental**: inverse to the incremental one. It starts from the full set of features and then reduces it by removing features. The algorithm chosen is Sequential Backward Selection (SBS) (Marill and Green, 1963).
- **Probabilistic**: it generates random feature sets and keeps the one with the best score. The algorithm chosen is Las Vegas Filter (LVF) (Atallah and Fox, 1998).
- **Meta-heuristic**: uses an heuristic to obtain another heuristic which optimizes the search process. The algorithm chosen is Simulated Annealing (SA), proposed by Kirkpatrick et al (1983).

These search algorithms are combined with the consistency measures described in Section 2 to perform FS. The cache implementations of these measures may have different performance depending on the different search algorithms.

5 Experimental results

This section shows the most relevant experimental results organized by the hypotheses established in section 4. For each one, the conclusions reached are explained and justified. At the end, the relative performance of all approaches is compared and the influence of a large number of features is studied.

H1 *Hash tables perform faster than the Sorted tables without cache*

The first experiment assesses the implementation of the measures independently of the search in which it may be integrated. No cache effect is con-

sidered yet. Two implementations of measures are compared, one using hash tables and the other using sorted tables. As the tables will not be reused, the sorted tables implementation uses the lazy sort method (as there is no future benefit in sorting by the other features). We have run both implementations of all measures over a sample of feature sets of all sizes.

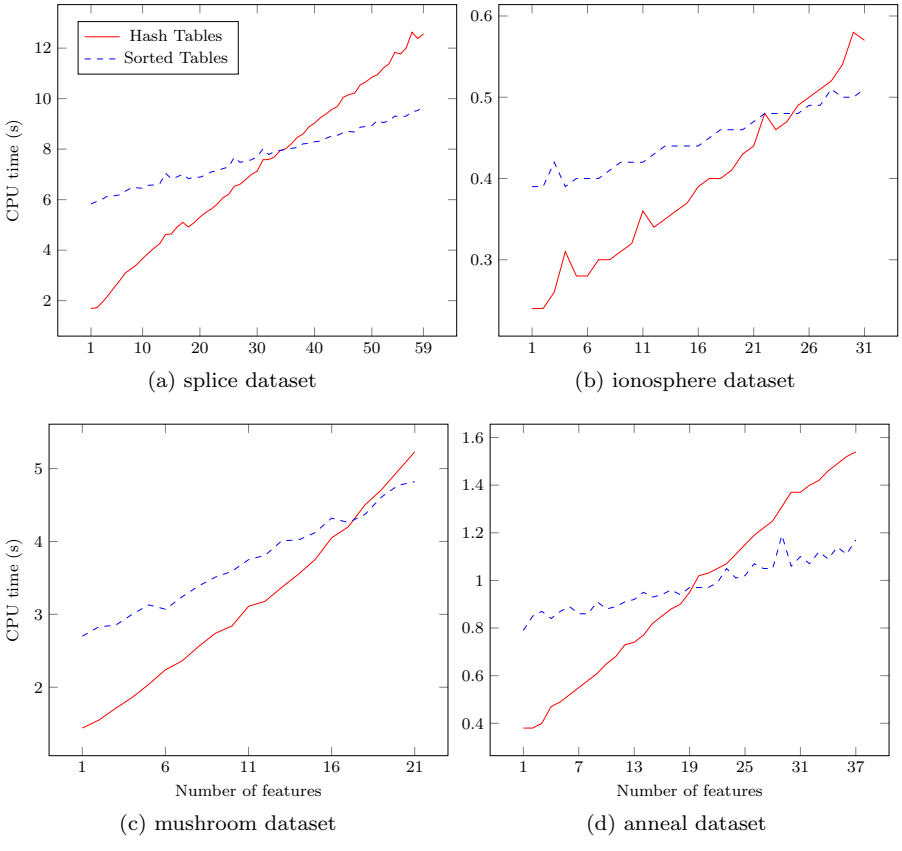


Fig. 4 Average CPU time taken to calculate IE measure over feature sets of every size for each data set

Fig. 4 shows the CPU time taken for every possible size of feature set by both implementations of the inconsistent examples (IE) measure, over four of the largest data sets from Table 3. Hash tables version is faster for small feature sets while sorted tables version outruns hash tables in large feature sets. All the other measures have similar results, except for the binary consistency measure.

The case for the binary consistency measure is shown in figure 5. Although the tendency is similar, the sorted tables version is not able to outrun hash tables or, in the case of anneal data set, it needs a higher number of features.

The reason behind this is that, as the hash table is populated using the selected features, it can be stopped when two examples with different classes are found (an inconsistency is detected and the binary consistency value becomes 0). It does not need to scan the whole set of examples. However, this is not possible when sorting, as the features are taken one by one and the rest of the features may have a different value for the conflicting examples. The other data sets exhibit similar results.

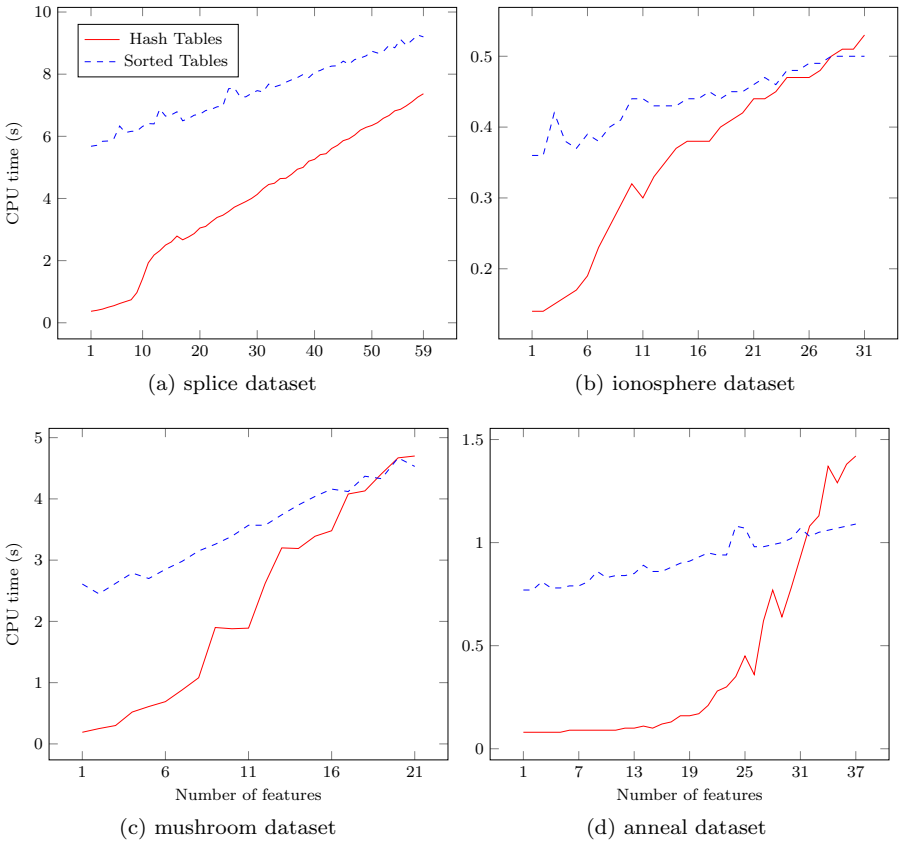


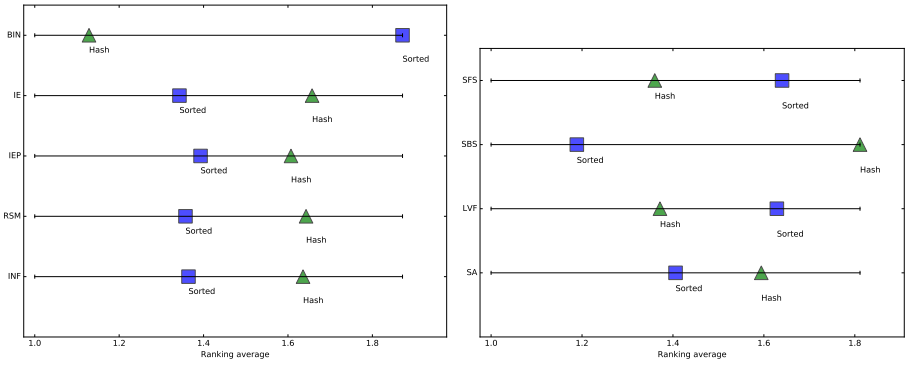
Fig. 5 Average CPU time taken to calculate Binary measure over feature sets of every size

As there are situations in which sorted tables perform better, it is interesting to test them inside the search process of FS algorithms. These searches which evaluate feature sets of whichever sizes are useful in a FS scenario using real-world data sets. Fig. 6 shows a comparison of both implementations for all the measures and with all the searches considered.

The CPU time taken by the algorithms is strongly related to the data set, so the obtained results are not commensurable among data sets. For this

reason, on each data set, the methods applied are ranked from the best to worst performance. As a proper aggregate value that allows comparing algorithms, all the ranking figures (6-12) show the average ranking position for each factor value among all data sets.

In order to discard the influence of external factors and reinforce our conclusions, statistical tests are applied following the methodology recommended by Demsar (2006). In this way, Wilcoxon test is applied when comparing two valued factors. Associated with each figure which illustrates the ranking results, table 5 indicates the Wilcoxon test p-value of every case.



(a) Grouped by consistency measure (b) Grouped by search algorithm

Fig. 6 Average ranking of implementations using hash tables and sorted tables without cache

When grouping by consistency measure (Fig. 6a), for the binary consistency, the hash implementation has been faster in most cases (in around 90% of the cases, as it can be deduced by the ranking values). The average includes all data sets combined with all search algorithms. This difference is significant (see Table 5), so the hypothesis is confirmed in the case of binary consistency. However, for the rest of measures, sorted tables without cache tend to be faster (in around 60% of the cases) but the differences are not significant according to Wilcoxon test.

When grouping by the search algorithm used (Fig. 6b), there are more significant differences. While there is no significant difference for Simulated Annealing (SA), the hash implementation is faster for SFS and LVF and the sorted tables implementation is faster for SBS. The reason for this difference seems to be that SBS works with larger feature set where, as we have just seen in figure 4, sorted tables implementation is faster.

At some situations, the use of sorted tables is already faster. This increases the motivation for the introduction of the cache system, because even a minimal improvement will improve the best performing implementation.

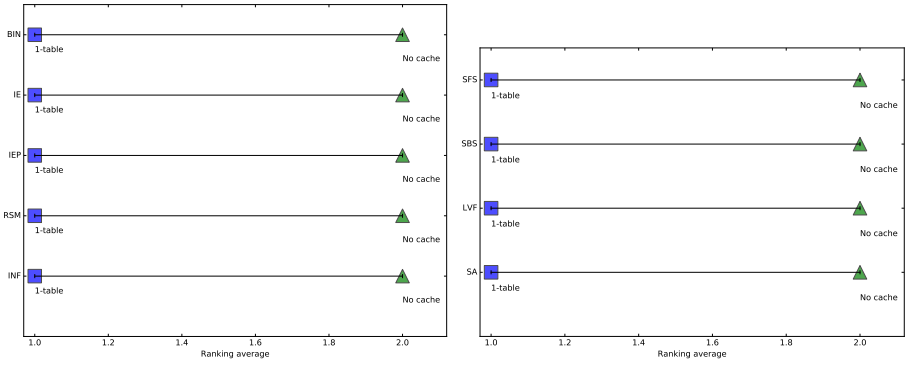
Table 5
Wilcoxon test p-values for the results illustrated in each figure

Search	6b	7b
SFS	1.3e-09	1.8e-30
SBS	3.9e-11	1.8e-30
LVF	4.2e-08	1.8e-30
SA	0.15	1.8e-30

Measure	6a	7a
BIN	1.3e-16	1.0e-24
IE	0.10	1.0e-24
IEP	0.68	1.0e-24
RSM	0.13	1.0e-24
INF	0.11	1.0e-24

H2 *Using cache increases performance*

For every combination of search and measures, on each data set, caching just one table has lead to faster FS. Ranking and significance reach their maximal values, as shown in Fig. 7 and Table 5, so this hypothesis is clearly confirmed.



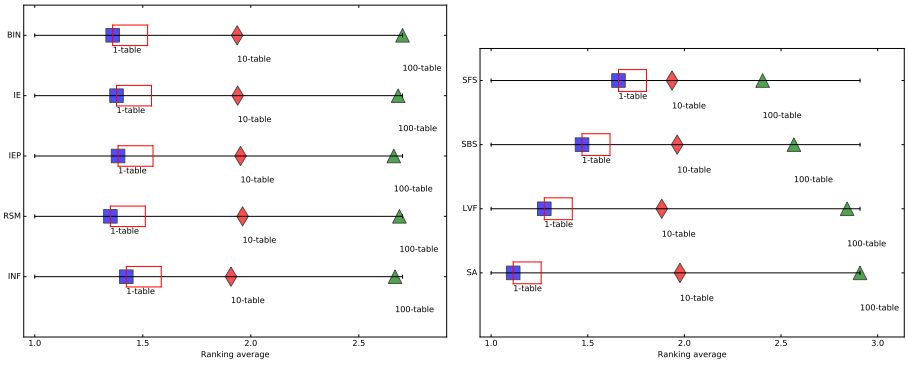
(a) Grouped by consistency measure (b) Grouped by search algorithm

Fig. 7 Average ranking of implementations not using cache and using one cache table when using sorted tables

H3 *Having more cached tables increases performance*

Fig. 8 shows the comparison of three cache sizes (1, 10 and 100). When comparing multiple elements, we use the statistical test by Iman and Davenport, which is stronger than the original Friedman χ^2 test (Demsar, 2006). In all multiple ranking figures (figures 8-12), the null hypothesis (all methods behave similarly) has been rejected with p-value < 0.01.

Once the null hypothesis is rejected, we focus on finding which ones behave better by using Nemenyi post-hoc test (non-parametric Tukey). The critical distance will be calculated according to the equation (20), where k is the number of factor values and n the number of paired samples. Graphically, a rectangle is used to represent the critical distance in these figures. For the sake



(a) Grouped by consistency measure (b) Grouped by search algorithm

Fig. 8 Average ranking of cache strategies

of simplicity, just the rectangle from the first method is represented but all the differences between any pair of methods whose value is bigger than the critical distance are significant.

$$DC = q_{\alpha,\infty,k} \cdot \sqrt{\frac{k(k+1)}{12n}} \tag{20}$$

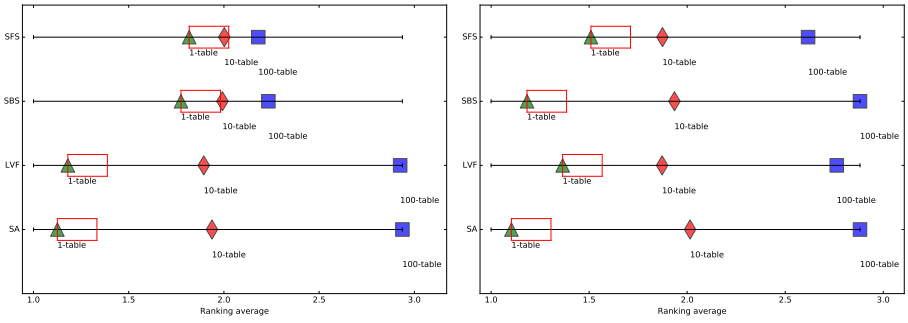
When grouped by consistency measure (Fig. 8a), the pattern is similar among measures so the result does not seem to depend on the measure used. We observe that, the larger the cache size, the worst its performance is. We have also tested with cache sizes of 3 and 5 tables but no conclusions can be reached with statistical significance. It seems to depend heavily on the data set. Nevertheless, using just one table is the best performer in most of those cases.

When grouped by the search algorithm used (Fig. 8b), results are similar among search methods as well but it can be noted that the differences are clearer on random path methods than on sequential search methods. In order to investigate this result in depth, we divided the data sets into two groups according to the number of features, using the median of the number of features as threshold. We can observe the results in Figures 9a and 9b.

It can be concluded that in most cases, the cost of using multiple cache tables outweighs the benefits of increasing the chances of table reuse. So, H3 is rejected.

H4 *Level cache strategy is the best cache strategy for sequential algorithms*

In figure 10, the combinations of the cache strategies and the sequential algorithms (SFS and SBS) are illustrated separated by the sort method because there is a dependence on it. Level is the best cache option for both SFS and SBS, except for the lazy sort, in which SBS in combination with level cache performs poorly.



(a) Data sets with 15 or less features (b) Data sets with more than 15 features

Fig. 9 Average ranking of cache strategies grouped by search algorithm

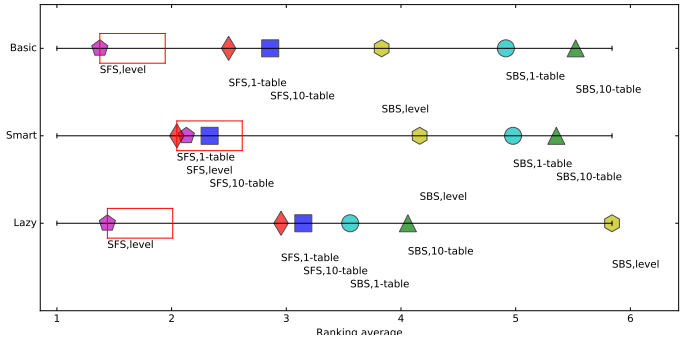


Fig. 10 Average ranking of combinations of search algorithms and cache strategies grouped by sorting methods

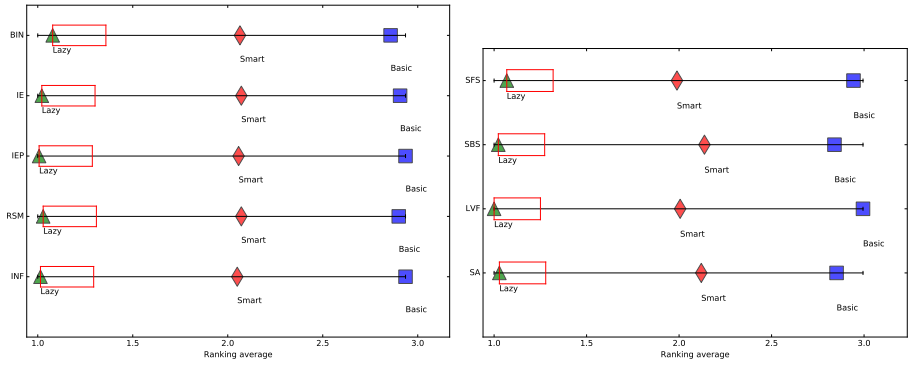
Level cache strategy uses the table from the current set of selected features (also known as the parent table) to evaluate the candidate sets of measures. For ascending algorithms such as SFS, this has shown to be a clear advantage (it only needs to sort the table by the new feature).

On the other hand, for descending algorithms such as SBS, the number of required sort operations is the number of selected features before the removed one in the feature order. This seems to work well with basic and smart sort methods but, as the lazy sort does not keep the tables sorted by many features, the effort of keeping the tables that the level strategy saves is not compensated because of the low reuse.

H5 *Some sort method performs better than the others*

The results are similar when grouped by consistency measure (Fig. 11a) and search algorithm (Fig. 11b). As expected, both the lazy and smart sort methods perform better than the basic sort method. The lazy method has been the fastest in most experiments. This lets us deduce that it is better to save time

by not sorting the table by unselected features than to increase re-usability by keeping it fully ordered.



(a) Grouped by consistency measure (b) Grouped by search algorithm

Fig. 11 Average ranking of sorting methods

H6 *The best sorted tables cache implementation performs better than the hash tables*

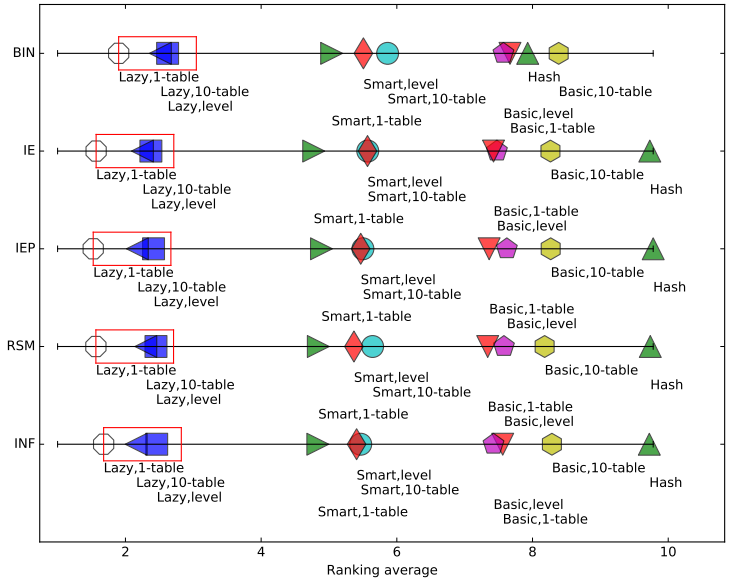
The results of all implementations are shown grouped by consistency measure (Fig. 12a) and search algorithm (Fig. 12b).

Sorted tables using one cached table and the lazy sort method seem to be the best implementation, except for SFS in which the lazy sort with level cache is more effective. Nevertheless, hash tables implementation has been the slowest in most runs, except for the binary measure, where hash outruns sorted tables using basic order and cache of 10 tables. Therefore the hypothesis is confirmed.

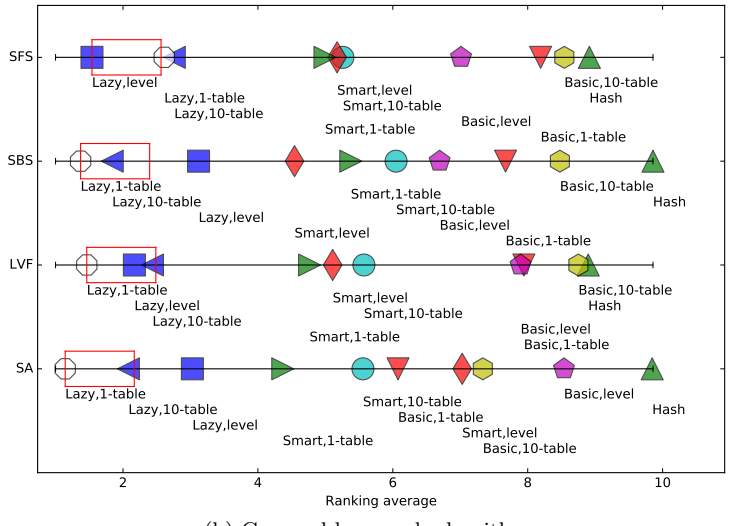
5.1 Relative performance of cache implementations

Using a ranking has given statistical soundness but the magnitude of the differences has been laid aside. Here, the time taken by the original hash implementation is set as the reference time for each run. The time taken by the other implementations is calculated relative to it and then, averaged among all runs. This relative performance of the implementations is shown in Figures 13-16, one figure for each search method. The results shown belong to the IE measure implementations. Nevertheless, the figures for IEP, Inf or RSM are quite similar.

These figures show that the use of any of the cache implementations reduces more than 50% the time needed for FS. Moreover, the use of the best cache options for each search escalates the reduction into between 84% and 89%.



(a) Grouped by consistency measure



(b) Grouped by search algorithm

Fig. 12 Comparison of hash and all combinations of sorting methods and cache strategies

In sequential forward search (SFS, figure 13), the best implementation reduces the time to 89% by using the level strategy and the lazy sort method. In this case, the level strategy reduces the time relative to the non-level based strategies to 30%.

In sequential backward search (SBS, figure 14), the same reduction (89%) is achieved by keeping 10 cached tables with the lazy sort method. In this case,

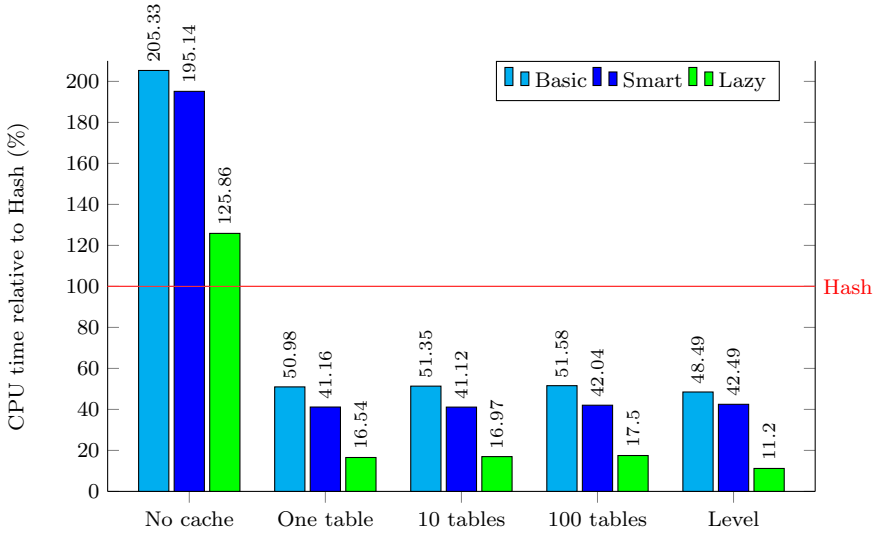


Fig. 13 SFS with IE measure feature selection times, relative to the time using hash

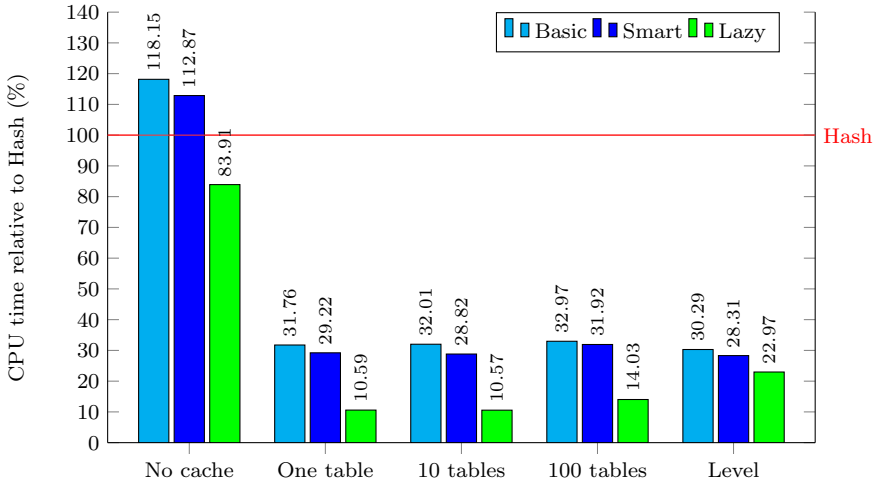


Fig. 14 SBS with IE measure feature selection times, relative to the time using hash

keeping more tables worsened the results, even doubling the time when using the level strategy.

In simulated annealing (SA, figure 16) and Las Vegas filter (LVF, figure 15) the best implementation keeps just one table with the lazy sort. However, while the level strategy slows the times in SA, in LVF the level strategy achieves results similar to the best ones.

For the binary consistency measure, the cache strategies and sorting methods which produce the best results are the same. However, the hash algorithm

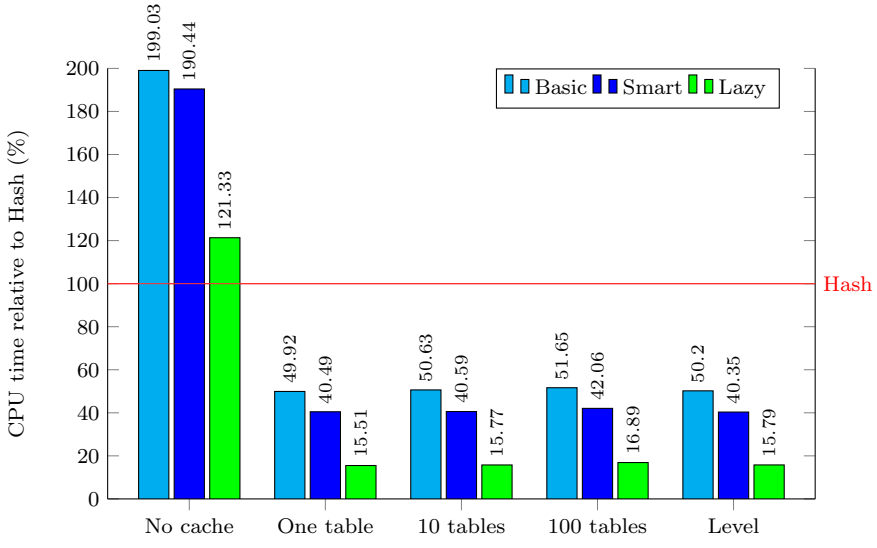


Fig. 15 LVF with IE measure feature selection times, relative to the time using hash

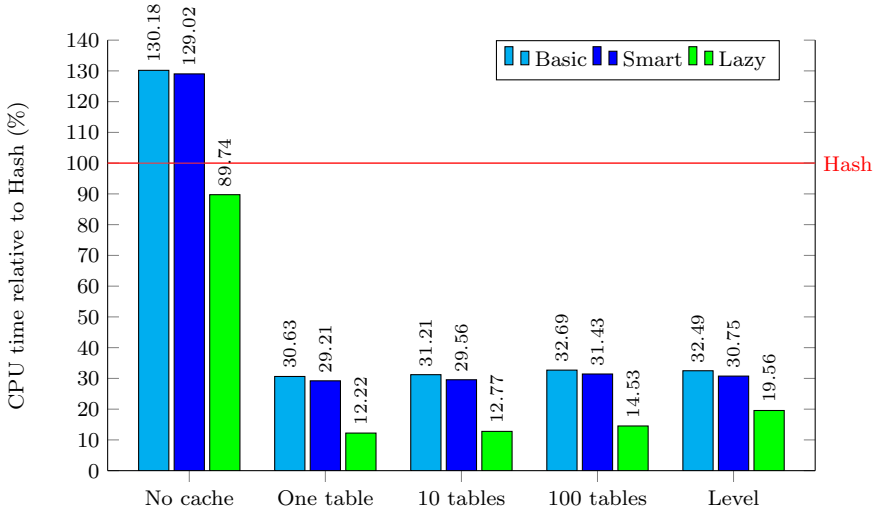


Fig. 16 SA with IE measure feature selection times, relative to the time using hash

performs better than all cache strategies using smart sort or basic sort in SFS, LVF and SA. Nevertheless, the reduction achieved with the best cache in binary consistency consists on 41%, 21% and 68% respectively (SFS, figure 17, SA and LVF perform similarly). In sequential backward search (SBS, figure 18), all cache strategies achieve some reduction, being 86% the maximum achieved.

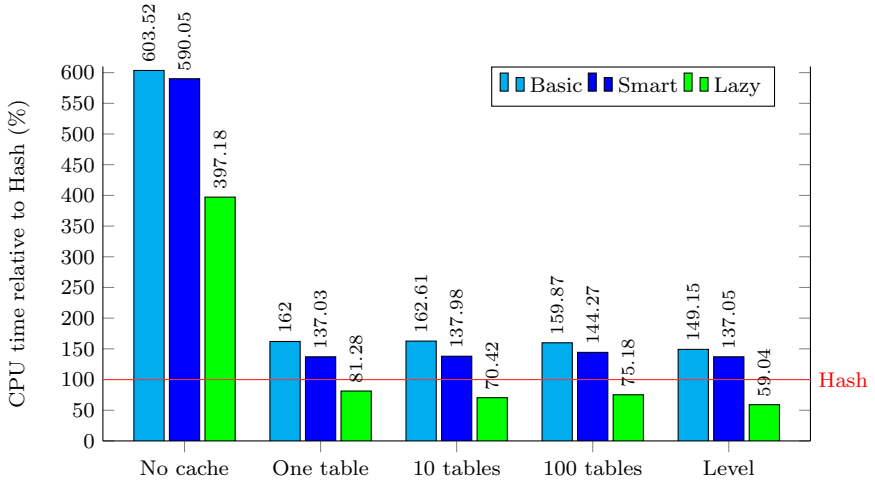


Fig. 17 SFS with BIN measure feature selection times, relative to the time using hash

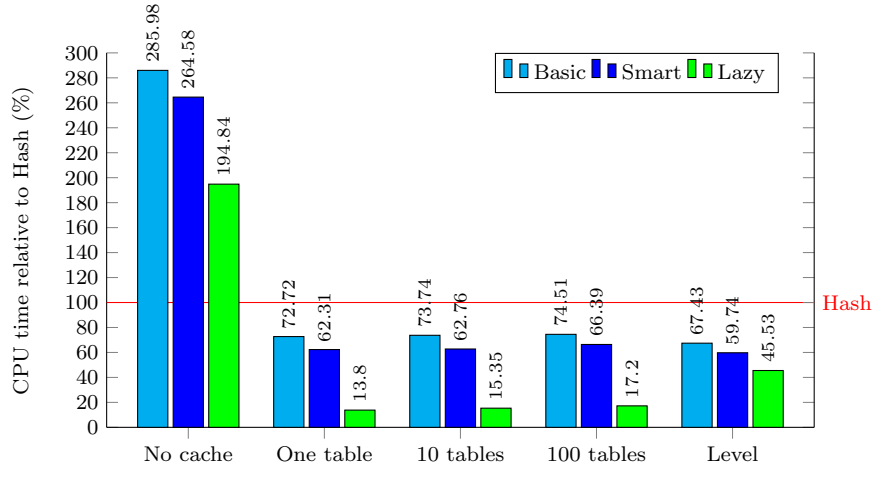


Fig. 18 SBS with BIN measure feature selection times, relative to the time using hash

5.2 Scaling the cache system to high dimensional datasets

In order to study the influence of a larger number of features, two additional experiments have been carried out comparing Hash tables and two variants of sorted tables: lazy sort without cache (No cache) and the best performing cache, lazy sort with one cache table (1-Lazy), over the CNAE-9 dataset, which has a higher number of features than the datasets we have used before.

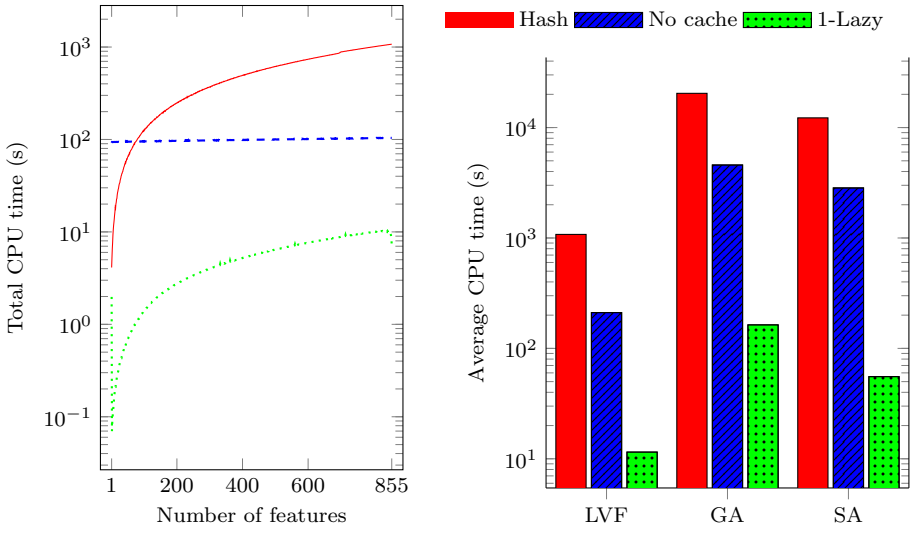
The first experiment consisted in the evaluation of 50 sets of random features of every possible size using IE measure (figure 19a).

Similarly to previous experiments, *Hash* initially performs better than *No cache* but, as the number of features increase, so does its CPU time whereas *No cache* barely changes its performance. In *1-Lazy* graph, we can see an initial peak which corresponds to the generation of the first sorted table. As the first table gets re-used, its CPU time gets reduced as well and it keeps growing along with the number of features, but within small values.

The second experiment consisted in performing executions of the full feature selection process using three different search algorithms (LVF, GA and SA) with the IE measure, and analyzing the time dedicated to feature selection (figure 19b).

We can observe that *Hash* performs poorly when compared to *No cache* and *1-Lazy*. Using the profiling tools, we have learned that, on large feature sets, *Hash* spends most of the time building the hash table, as it has to calculate the hash for each element, whereas the creation of the sorted table in *No cache* or *1-Lazy* was remarkably faster.

Finally, we can conclude that enabling cache strongly speeds up the feature selection process in high dimensional data sets, even more than in data sets with less features. In this case, the computation time gets reduced one order of magnitude. This opens the research question of whether the presented cache system can be extended for big data sets. Given present results, it seems a promising idea.



(a) Computation of IE measure for 50 random feature sets of every size

(b) Feature selection process repeated for 10 cross-validation folds

Fig. 19 Experiments over CNAE dataset, with 856 features

6 Conclusion

This paper has proposed a novel algorithmic cache storing sorted tables inside FS algorithms. It enables faster computation of widely used feature set measures. An empirical study has been conducted to clarify when sorted tables based on implementations are faster than hash tables for computing consistency measures. From this study, we can conclude that:

- Results vary depending on: the number of features of the data set, the search algorithm or the consistency measure considered but, when compared to hash tables, the use of the proposed sorted tables cache results in better performance in most cases.
- As the number of tables stored increases, the benefits of the higher probability of finding a suitable table to reuse get overcome by the increment of computational load, so it is recommended to keep the number of cached tables low. On the other hand, if used with sequential search algorithms, the probability of reusing tables is higher and the level cache method keeping up to 100 tables is a better choice.
- As to sorting choice methods, the lazy sort method is, by far, the best performer. This means that it is better to save time by not sorting the table by the non-selected features than to spend time sorting it completely to increase its chances of being reused.
- The sorted table implementation, even without cache, runs faster than hash tables when the percentage of selected features is high, but it varies for each data set. It would be interesting to determine the influence of factors like the number of features, complexity of attributes or number of examples in order to estimate a threshold value to choose the implementation to run.

For future work, we think that the algorithmic cache idea can be applied to several other Machine Learning related algorithms. The algorithmic cache can also be improved by borrowing many ideas from hardware cache systems. Besides, we feel interest in studying how this proposal can be scaled to distributed big data sets.

References

Almuallim H, Dietterich TG (1991) Learning with many irrelevant features. In: In Proceedings of the Ninth National Conference on Artificial Intelligence, AAAI Press, pp 547–552

Almuallim H, Dietterich TG (1994) Learning boolean concepts in the presence of many irrelevant features. Artificial Intelligence 69(1-2):279–305, URL citeseer.nj.nec.com/almuallim94learning.html

Arauzo-Azofra A, Benítez JM, Castro JL (2008) Consistency measures for feature selection. Journal of Intelligent Information Systems 30(3):273–292, DOI 10.1007/s10844-007-0037-0, URL http://dx.doi.org/10.1007/s10844-007-0037-0

Arauzo-Azofra A, Aznarte JL, Benítez JM (2011) Empirical study of feature selection methods based on individual feature evaluation for classification problems. Expert Systems with Applications 38(7):8170 – 8177, DOI http://dx.doi.org/10.1016/j.eswa.2010.12.160, URL http://www.sciencedirect.com/science/article/pii/S095741741001523X

Atallah MJ, Fox S (eds) (1998) Algorithms and Theory of Computation Handbook, 1st edn. CRC Press, Inc., Boca Raton, FL, USA

Auger N, Nicaud C, Pivoteau C (2015) Merge Strategies: from Merge Sort to TimSort, URL https://hal-upec-upem.archives-ouvertes.fr/hal-01212839, working paper or preprint

Bharti KK, Singh PK (2015) Hybrid dimension reduction by integrating feature selection with feature extraction method for text clustering. Expert Systems with Applications 42(6):3105–3114

Chen X, Fang T, Huo H, Li D (2011) Graph-based feature selection for object-oriented classification in vhr airborne imagery. IEEE Transactions on Geoscience and Remote Sensing 49(1):353–365

Cover TM, Thomas JA (1991) Elements of Information Theory. Wiley-Interscience, New York, NY, USA

Dash M, Liu H (2003) Consistency-based search in feature selection. Artif Intell 151(1-2):155–176, DOI 10.1016/S0004-3702(03)00079-1, URL http://dx.doi.org/10.1016/S0004-3702(03)00079-1

Demsar J (2006) Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research 7:1–30

Demšar J, Curk T, Erjavec A, Črt Gorup, Hočevar T, Milutinovič M, Možina M, Polajnar M, Toplak M, Starič A, Štajdohar M, Umek L, Žagar L, Žbontar J, Žitnik M, Zupan B (2013) Orange: Data mining toolbox in python. Journal of Machine Learning Research 14:2349–2353, URL http://jmlr.org/papers/v14/demsar13a.html

Fortin FA, De Rainville FM, Gardner MA, Parizeau M, Gagné C (2012) DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research 13:2171–2175

Frigo M, Leiserson CE, Prokop H, Ramachandran S (2012) Cache-oblivious algorithms. ACM Trans Algorithms 8(1):4:1–4:22, DOI 10.1145/2071379.2071383, URL http://doi.acm.org/10.1145/2071379.2071383

García S, Luengo J, Herrera F (2016) Data preprocessing in data mining. Springer

Geng X, Liu TY, Qin T, Li H (2007) Feature selection for ranking. In: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, pp 407–414

Gui J, Sun Z, Ji S, Tao D, Tan T (2017) Feature selection based on structured sparsity: A comprehensive study. IEEE transactions on neural networks and learning systems 28(7):1490–1507

Hastie T, Tibshirani R, Friedman J (2001) The Elements of Statistical Learning. Springer Series in Statistics, Springer New York Inc., New York, NY, USA

Kern R (2016) rkern/line_profiler. URL https://github.com/rkern/line_profiler

Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. SCIENCE 220(4598):671–680

Kohavi R, John GH (1997) Wrappers for feature subset selection. Artificial Intelligence 97(1-2):273–324

Koprinska I, Rana M, Agelidis VG (2015) Correlation and instance based feature selection for electricity load forecasting. Knowledge-Based Systems 82:29–40

Kowarschik M, Weiß C (2003) An overview of cache optimization techniques and cache-aware numerical algorithms. Algorithms for Memory Hierarchies pp 213–232

Lanaro G (2013) Python High Performance Programming. Packt Publishing

Liu M, Zhang D (2016) Pairwise constraint-guided sparse learning for feature selection. IEEE transactions on cybernetics 46(1):298–310

Marill T, Green D (1963) On the effectiveness of receptors in recognition systems. Information Theory, IEEE Transactions on 9(1):11–17

Molina L, Belanche L, Nebot A (2002) Feature selection algorithms: a survey and experimental evaluation. In: Data Mining, 2002. ICDM 2002. Proceedings. 2002 IEEE International Conference on, pp 306–313, DOI 10.1109/ICDM.2002.1183917

Newman CBD, Merz C (1998) UCI repository of machine learning databases. URL http://www.ics.uci.edu/$\sim$mlearn/MLRepository.html

Onan A (2015) A fuzzy-rough nearest neighbor classifier combined with consistency-based subset evaluation and instance selection for automated diagnosis of breast cancer. Expert Systems with Applications 42(20):6844–6852

Pawlak Z (1982) Rough sets. International Journal of Computer and Information Science 11:341–356

Qian W, Shu W (2015) Mutual information criterion for feature selection from incomplete data. Neurocomputing 168:210–220, DOI 10.1016/j.neucom.2015.05.105, URL http://dx.doi.org/10.1016/j.neucom.2015.05.105

Shin K, Miyazaki S (2016) A fast and accurate feature selection algorithm based on binary consistency measure. Computational Intelligence 32(4):646–667, DOI 10.1111/coin.12072, URL http://dx.doi.org/10.1111/coin.12072

Shin K, Fernandes D, Miyazaki S (2011) Consistency measures for feature selection: A formal definition, relative sensitivity comparison, and a fast algorithm. In: Walsh T (ed) IJCAI, IJCAI/AAAI, pp 1491–1497, URL http://dblp.uni-trier.de/db/conf/ijcai/ijcai2011.html

Song Q, Ni J, Wang G (2013) A fast clustering-based feature subset selection algorithm for high-dimensional data. IEEE transactions on knowledge and data engineering 25(1):1–14

Whitney AW (1971) A direct method of nonparametric measurement selection. IEEE Trans Comput 20(9):1100–1103, DOI 10.1109/T-C.1971.223410, URL http://dx.doi.org/10.1109/T-C.1971.223410

Zhao Z, Liu H (2007) Spectral feature selection for supervised and unsupervised learning. In: Proceedings of the 24th international conference on Machine learning, ACM, pp 1151–1157

Zheng K, Wang X (2018) Feature selection method with joint maximal information entropy between features and class. Pattern Recognition 77:20 – 29, DOI https://doi.org/10.1016/j.patcog.2017.12.008, URL http://www.sciencedirect.com/science/article/pii/S0031320317304946