ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume 22, Number 1, 2019, 85–99

Efficiency Optimizations for Koller and Sahami's Feature Selection Algorithm

Camil Băncioiu[1], Maria Vințan[1], and Lucian Vințan[1]

[1]Faculty of Engineering, *Lucian Blaga* University, Sibiu, Romania
Email: camil.bancioiu@ulbsibiu.ro, maria.vintan@ulbsibiu.ro, lucian.vintan@ulbsibiu.ro

Abstract. This article describes and evaluates four optimizations for Koller and Sahami's Feature Selection algorithm, significantly reducing the time it requires to complete. The optimizations exploit the Information Theory concepts used by the algorithm, its inherent data parallelism and the fact that much of the calculations it performs are redundant. Each proposed optimization was carefully evaluated, showing significant efficiency gains. In particular, a decomposition of conditional mutual information is shown to reduce the time required to calculate its primary heuristic and can be potentially applied to other algorithms which calculate conditional mutual information.

Key-words: Feature Selection, Koller and Sahami's Algorithm, Markov Blankets, Conditional Mutual Information, Data Parallelism, Caching Techniques, Computation Reuse.

1. Introduction

Feature Selection algorithms are often applied in Machine Learning problems to reduce the dimensionality of datasets, which often describe a sample using thousands of attributes (features). The goal is to reduce the complexity of the trained models, to increase their accuracy and also to reduce their consumption of computing resources. A Feature Selection algorithm will attempt to identify and remove irrelevant features, while ensuring that a relevant and informative subset of features remains in the dataset.

Koller and Sahami's algorithm (KS) is such a Feature Selection algorithm, categorized as an information-theoretic feature filter. It has two phases: the first phase calculates a heuristic for pairs of features; the second phase iteratively removes features from the dataset using an elimination criterion based on Markov blankets, a concept from Information Theory. To our knowledge, the KS algorithm is the first Feature Selection algorithm to employ this concept. In their article, Koller and Sahami provide a useful theoretical framework for selecting the optimal subset of

features using Markov blankets. The algorithm itself, though, only approximates Markov blankets using a heuristic [1]. In other words, the KS algorithm does not guarantee finding the true Markov blanket of a variable. Some algorithms that have followed in the footsteps of KS, briefly described in Section 2., have addressed this important shortcoming, thus rendering the KS algorithm obsolete. Still, the core characteristics of the KS algorithm persist in these subsequent algorithms, albeit in different forms:

- the measurement of correlation between features and testing for conditional independence using conditional mutual information;
- the ranking of features among themselves using conditional mutual information;
- the management of subsets of candidate features and testing whether they form a Markov blanket for a given feature.

Due to these shared characteristics, the algorithms that have superseded KS could also benefit from the optimizations proposed in this article.

An original implementation of the KS algorithm was developed and used by us in [2]. During the development of this implementation, optimization opportunities were discovered, which gain efficiency primarily by avoiding redundant calculations and by exploiting the data parallelism inherent to the algorithm. These optimizations are the focus of this article. They are described in detail in Section 5., along with experimental evaluations which illustrate their efficiency gains. In short, the developed optimizations are the following:

1. Gamma Decomposition, an optimization which uses fundamental concepts from Information Theory to rewrite the algorithm's γ heuristic in order to improve the efficiency with which it is calculated. No other similar optimization method was found in literature for the KS algorithm. Also, the theory employed by this optimization is not bound to the KS algorithm and could potentially be used by any other algorithm that needs efficient computation of conditional mutual information. This original optimization is covered by Section 5.1..

2. Removed Features Database, a persistent database which stores the features removed by every iteration of the KS algorithm, every time the algorithm is run. Although it only accelerates the algorithm by skipping iterations if they were previously executed in an identical run, this database greatly simplifies experiments, performs automatic comparisons between different runs and adds robustness to the algorithm. It is described in Section 5.2..

3. In-iteration Parallelism, an optimization which exploits the data parallelism inherent to the KS algorithm. It improves the efficiency of the algorithm by calculating approximate Markov blankets for multiple features in parallel. No implementation of the KS algorithm has been found in literature to exploit its intrinsic data parallelism. This optimization is described in Section 5.3..

4. Iteration Cache, a caching scheme which stores the approximate Markov blankets calculated in an iteration and makes them accessible in the next one. Such a caching mechanism was proposed by Koller and Sahami themselves, but they did not implement and evaluate it [1]. This optimization has been implemented in [2] and is evaluated in the experiment presented here, as described in Section 5.4..

According to our experimental evaluations described in Section 4., a complete run of the unoptimized KS algorithm required, for both phases of the algorithm, an average total time of $22 : 00 : 41$ (22 hours). With the most efficient optimizations enabled, the required time for both phases of a run was reduced to an average of $00 : 57 : 36$ (57 minutes), only 4.36% of the duration of the unoptimized algorithm.

2. Related work

When it was published in 1996, the KS algorithm effectively established a new class of Feature Selection algorithms, which employ Markov blankets. Since then, a number of subsequent algorithms have been published [3] to address the shortcomings of KS, especially the inability to guarantee finding the true Markov blanket of a variable. The first algorithm to be proven to correctly identify Markov blankets was GS [4] in 1999, although it was not efficient and cannot scale to large applications [3]. IAMB [5] was published in 2003 and addressed the efficiency issues of GS. While IAMB is proven correct and performs better than GS, it is data inefficient, requiring many samples to conclude a test of conditional independence [3] [6]. The algorithms MMMB [7] and HITON [8], also published in 2003, attempted to address the data inefficiency of IAMB by reducing the required number of tests of conditional independence, exploiting the underlying local structure of the Bayesian network from which the data is assumed to have been sampled. MMMB was especially influential, as it introduced a two-step methodology in inducing the Markov blanket of a target variable [3] [7]. This methodology is borrowed by the HITON algorithm as well. Unfortunately, while MMMB and HITON did gain efficiency, they were proven incorrect by Pena et al. [6] in 2006, who then proposed their own algorithm, called PCMB, with a proof of correctness. PCMB inherits much of the characteristics of MMMB, including the two-step methodology for Markov blanket induction, as well as the use of the statistical G-test for conditional independence [6]. In 2008, Fu and Desmarais published an algorithm called IPC-MB [9], which is inspired by PCMB and employs an incremental approach of performing conditional tests of independence, avoiding many unneeded tests. To our knowledge, IPC-MB remains the state-of-the-art Feature Selection algorithm employing the concept of Markov blankets at the time of writing this article. As stated in Section 1., these algorithms still present fundamental characteristics inherited from the initial KS algorithm, such as the use of conditional mutual information as a correlation measure, the ranking of features based on this measure and the management of subsets of features as candidates for a Markov blanket. While the KS algorithm has been obsoleted by the algorithms mentioned above, these shared characteristics and the simplicity of KS make it a good candidate for studying and analysis.

A number of papers were found in literature where the respective authors implemented the KS algorithm themselves. In all cases, though, there were little or no details given about the implementation.

The only paper mentioning any optimization for the KS algorithm at all is [1], the original paper by Koller and Sahami where the KS algorithm is proposed. The authors summarily present an idea for a caching mechanism to avoid unneeded recomputation of approximate Markov blankets, but do not elaborate it. They also state that they have not implemented any such caching scheme in their validation experiment. Their idea inspired the development of the Iteration Cache, described in Section 5.4..

Lee and Lee [10] compare the KS algorithm to another Feature Selection algorithm that they proposed but provide no implementation details, nor mention any optimization of the KS

algorithm. Moreover, they do not mention what values of the K parameter were used when running the KS algorithm.

Tsamardinos et al. [11] present a comparison of four Feature Selection algorithms, including the KS algorithm. The other three algorithms in this comparison also employ the concept of Markov blankets. No details are given about the implementation of the KS algorithm, nor about optimizations added to it.

A real-world application of the KS algorithm is described by Pasero et al. [12] in 2008, who present a study of the feasibility of applying Machine Learning to forecast air pollution in the city of Turin, Italy. The authors implement the KS algorithm and use it to select features such as air pollutant concentrations and wind speed in order to predict the concentration of air pollutants after a few days. The authors provide no implementation details and do not mention any optimization for the implemented algorithm.

The KS algorithm has also been used in the field of bioinformatics. Saeys [13] effectively applies the KS algorithm on genetic datasets and compares it with other Feature Selection algorithms (one being SVM), based on the classification accuracy of the selected features. According to the results, the KS algorithm outperformed the other algorithms in some, but not all experiments. The author explicitly states that the KS algorithm was constrained to a single value of its parameter K, set to 1, due to the high computational cost of greater values [13], which is unfortunate. The optimizations proposed in Section 5. would have reduced this cost dramatically and would have enabled a more thorough comparison. The implementation of the KS algorithm used in [13] could not be found, although it appears to have been accessible in the past.

The KS algorithm is also implemented in the *Causal Explorer*, a MATLAB library of algorithms for causal discovery and Feature Selection [14]. The authors made the library available for download on the Internet, although it can only be retrieved in compiled form, without access to the source code. The paper [14] is an overview of the library and the algorithm implementations it contains, but provides no details of how the KS algorithm is implemented. The authors do, however, mention that "a very fast implementation of expected cross entropy is used in the algorithm implementation". It is assumed that "cross entropy" refers to the Kullback-Leibler divergence (relative entropy). It is unclear if they attempted to optimize the calculation of the algorithm heuristics, which indeed use the Kullback-Leibler divergence, or they simply use a fast method to compute the generic Kullback-Leibler divergence. The fact that the authors of Causal Explorer mention a fast implementation suggests that they attempted to improve the efficiency of the KS algorithm, or at least were concerned by it.

This experiment uses the same experimental dataset used by the experiment presented in [2], where the KS algorithm was compared with Information Gain Thresholding in the quality of selected features, evaluated by training Naïve Bayes classifiers.

3. Koller and Sahami's algorithm

3.1. Overview

Koller and Sahami's Feature Selection algorithm is an information-theoretic feature filter. It is designed around the search for the approximate Markov blanket of a certain feature F_i. The approximate Markov blanket is in fact a subset of other features, chosen such that they will render the feature F_i as irrelevant as possible. The process of choosing features for the Markov blanket of a feature F_i is guided by the algorithm's γ heuristic. A second heuristic, named δ, is used by

the algorithm in each iteration to decide which feature is the most irrelevant given its approximate Markov blanket. In each iteration, the feature which scores lowest in the δ heuristic (thus is the most irrelevant feature) is removed.

The KS algorithm has two separate phases:

- Phase 1: Gamma Calculation, in which the values of the γ heuristic are calculated for the features and objective values of the dataset. This phase is only run once per dataset. The calculated γ values are stored and used by all future runs of the KS algorithm without recalculation.
- Phase 2: Iterative Feature Removal, in which the algorithm removes the feature with the lowest δ value in each iteration. Calculating δ for a feature involves composing approximate Markov blankets using the γ values, which were calculated for the dataset in Phase 1.

The parameters of the KS algorithm are the following:

- Q, the number of features to remain in the dataset at the end of the algorithm, used as the stopping condition. Because the algorithm removes one feature per iteration, the number of executed iterations is $n - Q$, where n is the total number of features in the dataset.
- K, the cardinality of the approximate Markov Blankets to assemble for each feature. This parameter controls the number of features to be chosen as the approximate Markov Blanket of a certain feature.

3.2. Formal definition

Let F be the complete set of features in the dataset. Let n be the number of features in F. Let C be the objective variable, also provided by the dataset. The algorithm will select the features from F that are most relevant to C. Let G be a subset of F, representing the "current feature subset" in each iteration. Initially, $G = F$, but a feature is removed from G in each iteration. After all iterations have completed, G will hold the result of the algorithm, i.e. the features selected to be most relevant to C.

Q and K are the algorithm parameters:

- Q is the number of features to select from F. After the last iteration, $card(G) = Q$.
- K is the cardinality of the approximate Markov Blankets to assemble for each feature.

Let F_i and F_j be individual features ($F_i, F_j \in F$). The γ heuristic is defined as:

$$\gamma_{ij} = \gamma(C, F_i, F_j) = \boldsymbol{D_{KL}}(P(C \mid F_i, F_j) \parallel P(C \mid F_j)) \tag{1}$$

In equation (1), the notation D_{KL} represents the Kullback-Leibler divergence, a measure of how different two probability distributions are. It is defined as follows:

$$\boldsymbol{D_{KL}}(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)} \tag{2}$$

$$= \sum_x p(x) \log p(x) - \sum_x p(x) \log q(x) \tag{3}$$

The Kullback-Leibler divergence is interpreted as the amount of information lost, on average, when assuming that q is the distribution of a random variable, when in fact p is the true distribution. Assuming the base of the logarithm used in the definition to be 2, the Kullback-Leibler divergence is measured in bits. Note that it is not a distance, due to its asymmetry in p and q and because it does not satisfy the triangle inequality. Defined this way, the value of γ_{ij} expresses the amount of information brought by F_i with respect to C, when F_j is given.

Using the γ heuristic, the approximate Markov blanket of a feature F_i within an iteration is calculated by the following function:

$$\begin{aligned} aMB\,(C, K, F_i, G) = \{F_j \mid\ & F_j \in G \text{ and } i \neq j \text{ and} \\ & card(\{F_t \mid F_t \in G \text{ and } t \neq i \text{ and } t \neq j \text{ and} \\ & \quad \gamma(C, F_i, F_t) < \gamma(C, F_i, F_j) \\ & \}) < K \\ \} & \end{aligned} \tag{4}$$

The condition in Equation (4) states that all features F_j that will compose the approximate Markov blanket of the feature F_i must have less than K features F_t with a lower γ score than F_j in the current iteration.

Note that $\gamma(C, F_i, F_t) < \gamma(C, F_i, F_j)$ means that F_t is better than F_j from the point of view of the KS algorithm. Thus the result of $aMB(C, K, F_i, G)$ is the best subset of K features F_j from G, when sorted by the γ heuristic. Also note that $aMB(C, K, F_i, G) \subset G \subset F$.

Using the definition of aMB, the δ heuristic can now be defined. Let $MB_i = aMB(C, K, F_i, G)$, $MB_i \subset G$, which is the approximate Markov blanket of the feature F_i in the current iteration. The value of the δ heuristic for the feature F_i is:

$$\delta(C, F_i, MB_i) = \boldsymbol{D_{KL}}(P(C \mid F_i, MB_i) \parallel P(C \mid MB_i)) \tag{5}$$

The KS algorithm revolves around the removal of the features with the lowest value for the δ heuristic. After defining δ, the KS algorithm can be defined as the function:

$$\begin{aligned} & KS(C, F, Q, K) = G_{card(F)-Q} \\ & \quad \text{where} \\ & \quad\quad G_0 = F \\ & \quad\quad G_r = G_{r-1} - \{F_i \mid \operatorname*{argmin}_{F_i}(\ \delta(C, F_i, MB_i)\ )\} \end{aligned} \tag{6}$$

4. Experiment design and implementation

The optimizations were experimentally evaluated one at a time. The KS algorithm was run before and after enabling each proposed optimization and the duration of each run was recorded. Each such evaluation consisted of applying the KS algorithm on a dataset of text documents, from which the algorithm must select the words it deems most relevant to a given class of documents. Thus, from the point of view of the KS algorithm, the set of words represented the set of features F and the class of documents represented the objective variable C.

The Removed Features Database was treated as an exception. Instead of being evaluated by comparing the efficiency of the KS algorithm before and after enabling it, it was permanently enabled and its efficiency gains were not evaluated, due to its nature (it simply skips iterations). Its significant contribution to a simpler design of experiment is discussed instead in Section 5.2., as well as the robustness it grants to the algorithm.

The documents in the dataset and their assignments to classes were extracted from the Reuters Corpus Volume 1 [15]. The selected subset contains $n = 628$ documents with a vocabulary of $m = 2645$ words, represented as a binary document-term matrix X, where each row corresponds to a document, each column corresponds to a word, and the cells of the matrix contain the value 1 if the document corresponding to the row contains the word corresponding to the column, or 0 otherwise. Each document may be independently assigned to one or more of the $c = 7$ classes in the dataset, or to none at all. These document-to-class assignments have been represented as a binary matrix Y, where each row corresponds to a document, each column corresponds to a class and the cells of the matrix contain the value 1 if the document corresponding to the row is assigned to the class corresponding to the column, or 0 otherwise.

Because the chosen experimental dataset contains 7 document classes, the algorithm was run 7 times without any optimization enabled and 7 times whenever an optimization was enabled, once for each document class as the objective variable C.

The quality of the features selected by the algorithm is not considered in the evaluation, neither for the unoptimized run, nor when enabling optimizations. An evaluation of the quality of the features selected by the KS algorithm was presented by us in [2], where the accuracy of Naïve Bayes classifiers is measured on the same dataset, after performing Feature Selection with the KS algorithm. Instead, the experiment presented in this article only focuses on evaluating the efficiency gains of the optimizations added to the KS algorithm. Still, an automatic comparison is performed in every run to verify whether enabling an optimization affects the features that the algorithm selects. For Phase 2, Iterative Feature Removal, all optimized runs produced the same result as the unoptimized run (same selected features). For Phase 1, though, this automatic comparison highlighted some minor differences, as explained in Section 5.1..

For all runs of Phase 2 of the KS algorithm, the Q parameter has been fixed to 1. When $Q = 1$, the algorithm will remove all but one of the features, thus executing the highest number of meaningful iterations (running with $Q = 0$ provides no extra meaningful information). The K algorithm parameter has also been fixed for all runs. The value $K = 5$ was chosen, because our previous experiment presented in [2] showed that $K = 5$ leads to the highest classifier accuracy on this dataset, after evaluating the algorithm for $K \in \{1, 2, 3...20\}$.

The experimental framework, the KS algorithm and its optimizations were all implemented in Python 3. This implementation has been published online [16] under the GPL 3 license. It uses Scikit-learn [17], a rich framework for Machine Learning for Python 3 freely available under the BSD licence.

The experiment was run on a personal computer with a 64-bit Intel i7 (8-core) processor with a clock frequency of 1.73 GHz and 8 GB of DRAM running GNU/Linux (Fedora 26). In order to measure iteration and algorithm durations as accurately as possible, the operating system (OS) was brought to a controlled state by shutting down the desktop environment, networking services and other non-essential system services for the entire duration of the experiment. This is especially important for the In-iteration Parallelism optimization, where the OS has to map the sub-processes spawned by the optimized KS algorithm onto processor cores. This controlled state of the OS also ensured that the unoptimized and optimized runs of the algorithm are exe-

cuted in the same conditions.

5. Optimizations

5.1. Gamma Decomposition

Phase 1 of the algorithm consists of calculating the γ heuristic for all pairs of features (F_i, F_j). Although it is much simpler than Phase 2 of the algorithm, it is still computationally expensive, since it has to compute $n(n-1)$ values of γ, where n is the number of features in the dataset. To improve the efficiency of Phase 1, the Gamma Decomposition (GD) optimization rewrites the γ heuristic as a sum of reusable terms and caches them. To our knowledge, no implementation of the KS algorithm in literature has attempted to decompose γ and to optimize its calculation.

Koller and Sahami define the γ heuristic as:

$$\gamma(C, F_i, F_j) = \boldsymbol{D_{KL}}(P(C \mid F_i, F_j) \parallel P(C \mid F_j)) \tag{7}$$

As can be seen in this definition, γ represents the Kullback-Leibler divergence between two conditional probability distributions:

1. the probability distribution of variable C given both features F_i and F_j
2. the probability distribution of variable C given only feature F_j

The authors of [18] have proven that γ can in fact be rewritten as conditional mutual information. Mutual information, written as $I(A;B)$, is an information-theoretic measure expressing the amount of information about variable B that is contained in variable A. Mutual information is defined as:

$$\begin{aligned} I(A;B) &= H(A) - H(A \mid B) \\ &= H(B) - H(B \mid A) \end{aligned} \tag{8}$$

Here, the notations $H(A)$ and $H(A \mid B)$ represent the Shannon information entropy of variable A and the conditional Shannon entropy of variable A given B, respectively. They are defined as:

$$H(A) = -\sum_{a \in A} p(a) \log p(a) \tag{9}$$

$$H(A \mid B) = -\sum_{a \in A} \sum_{b \in B} p(a,b) \log p(a \mid b) \tag{10}$$

The information entropy of variable A is a measure of the information contained by its probability distribution. The conditional entropy of A given B is interpreted as the amount of information remaining in A after knowing B. Note that the entropy of a variable depends exclusively on its probability distribution, and not on the actual values it takes.

The mutual information of two variables also has a conditional form:

$$I(A; B \mid C) = H(A \mid C) - H(A \mid B, C) \tag{11}$$

The γ heuristic, rewritten as conditional mutual information [18] has the following form:

$$\gamma(Y, A, B) = I(Y; A \mid B) \tag{12}$$

But this identification can be taken one step further: conditional mutual information can be decomposed in a sum of entropies. The following equations illustrate this decomposition, using the notation $Y = C$, $A = F_i$ and $B = F_j$.

$$\begin{aligned}
&\gamma(Y, A, B) && (13)\\
&\quad = I(Y; A \mid B) && (14)\\
&\quad = I(Y; A, B) - I(Y; B) && (15)\\
&\quad = H(Y) - H(Y \mid A, B) - H(Y) + H(Y \mid B) && (16)\\
&\quad = -H(Y, A, B) + H(A, B) + H(Y, B) - H(B) && (17)\\
&\quad = \boldsymbol{H(A,B) + H(Y,B) - H(B) - H(Y,A,B)} && (18)
\end{aligned}$$

Decomposing γ into the sum of entropies shown in Equation (18) is a necessary step for optimizing its calculation. Thus, after this decomposition, three optimization opportunities are revealed:

1. The terms $H(A, B)$ and $-H(Y, A, B)$ are symmetric in A and B, thus they can be shared between the computation of both $\gamma(Y, A, B)$ and $\gamma(Y, B, A)$.

2. The terms $H(A, B)$ and $-H(B)$ are independent of Y and thus can be shared among the calculations for different Y variables. This is especially valuable in case the algorithm must run for more than one objective variable, yet for the same set of features.

3. No terms of the decomposition contain conditional probability distributions. The implementation is thus much simpler and cleaner than that of the original definition of γ, which was a Kullback-Leibler divergence of two conditional probability distributions.

Table 1 shows the durations of the Gamma Calculation phase when enabling the Gamma Decomposition optimization, compared to the unoptimized run.

The results show that decomposing the γ heuristic reduced the time needed by Phase 1 by more than half. This is a significant reduction on its own, but becomes even more important in combination with Phase 2 optimizations. Specifically, the Iteration Cache optimization reduces the time required by Phase 2 so drastically, that Phase 1 becomes the longer phase of the two. As a consequence, a faster Phase 1 becomes even more important, hence the significant value of the Gamma Decomposition optimization.

It can also be seen that the calculations with GD for document class 0 are slightly longer compared to calculations for the rest of the document classes (12% longer than the average duration for the other classes). This happens because the terms $H(A, B)$ and $-H(B)$ are calculated for the first document class, then cached for all the other classes.

Although calculating γ values with the Gamma Decomposition optimization is mathematically equivalent to the unoptimized version, in practice the results did contain minor differences when enabling GD. These differences are caused by the use of 64-bit floating-point numbers in the implementation. The unoptimized implementation of γ consisted of the Kullback-Leibler divergence of two conditional probability distributions, calculated as a sum of 8 terms of the

Table 1. Duration of Phase 1 for each document class, unoptimized versus optimized with Gamma Decomposition (GD).

Document class	**Duration of γ calculation unoptimized** (HH:MM:SS)	**Duration of γ calculation with GD** (HH:MM:SS)	**Duration of γ calculation with GD** (% of unopt. run)
0	01:56:29	00:56:18	48.33%
1	01:56:42	00:50:06	42.93%
2	01:56:57	00:49:58	42.72%
3	01:56:26	00:49:58	42.91%
4	01:56:59	00:50:10	42.88%
5	01:56:43	00:50:10	42.98%
6	01:56:58	00:50:17	42.99%
Total	13:37:14	05:56:57	43.68%

form $p(Y = y, A = a, B = b) \log \frac{p(Y=y|A=a,B=b)}{p(Y=y|B=b)}$. In contrast, the optimized implementation separately calculates the terms of the decomposition as entropies and then adds them together. Due to the differences in operations on floating-point numbers between the two versions, the floating-point representation errors add up differently.

These differences, although very small (about a millionth of a percent), can influence the features selected by the KS algorithm because they affect the sorting of very small γ_{ij} values and, as a consequence, the order in which features are removed from the dataset in Phase 2 of the algorithm.

It is important to note, though, that the affected γ_{ij} values are already very small, thus only the removal order of weak features is affected. Since these weak features are removed in the early iterations of the algorithm, the effect of these small differences in γ_{ij} is contained. The later iterations, where the algorithm removes better features to keep only the best, were not affected and behaved as the unoptimized version.

5.2. Removed Features Database

The Removed Features Database (RFDB) is a persistent database which stores, as individual entries, each feature removed by the KS algorithm from a given dataset, along with the order in which they are removed; therefore it optimizes Phase 2 of the algorithm. Because the algorithm is deterministic, it will always remove the same features in the same order for the same dataset and algorithm parameters. Thus, the algorithm can skip iterations for which the result is already found in the RFDB from a previous identical run. It is important to note that the entries of the RFDB can only be used for the dataset they were created with, therefore each dataset will require its own RFDB to be instantiated and stored separately. Along with the feature removed by an iteration, the RFDB also stores the duration of the iteration that removed the feature. This information is useful in analyzing the efficiency of the algorithm and was used throughout this experiment.

Each entry in the RFDB of a dataset is uniquely identified by the tuple $(ObjectiveVariable, IterationNumber, K)$. This tuple is named the "iteration key", because it unambiguously identifies the iteration that removed the feature across any algorithm run on the current dataset.

If, at the beginning of an iteration, an entry is found in the database for the current iteration key, it means that the result of the current iteration has already been calculated in a previous, identical run. The feature to be removed is retrieved directly from the RFDB and the iteration concluded. Alternatively, the algorithm can be configured to retrieve the feature to be removed from the RFDB, allow the iteration to compute it anyway and then verify if the results match.

Even if it cannot accelerate an algorithm run unless it contains entries created by an identical previous run, the RFDB represents a significant optimization, greatly simplifying the design of experiments which use the KS algorithm. This simplification results from the ability to store metadata within its entries (such as iteration duration), the ability to replay entire runs of the algorithm with ease and the ability to compare two algorithm runs with different optimizations enabled or disabled. This last ability of the RFDB was used in the current experiment to determine whether the evaluated optimizations affect the resulting feature subset selected by the algorithm. The RFDB was also central in the design of the experiment presented in [2].

The fact that the iterations for which the result is found in the RFDB are skipped also means that the algorithm can now continue a previously interrupted run, skipping iterations for which database entries are found. As a consequence, the algorithm is more robust when the RFDB is enabled.

5.3. In-iteration Parallelism

The KS algorithm offers an opportunity to exploit its intrinsic data parallelism. In each iteration, it loops over all features remaining in the subset G and it calls the same two functions for each one: (1) assemble an approximate Markov blanket for the feature and (2) calculate its δ heuristic. Evaluating these two functions for a feature is independent of evaluating them for any other feature in the current iteration. Because of this independence, the SPMD (single program, multiple data) model can be applied. Since it attempts to accelerate the iterations of the KS algorithm, In-iteration Parallelism is an optimization for Phase 2.

To take advantage of this data parallelism, the optimization spawns multiple sub-processes $P_1 ... P_p$ of the main process of the algorithm P_0 and delegates to them the calculations of the approximate Markov blanket and δ heuristic for each individual feature. Such a sub-process receives, as arguments, the feature F_i, the class variable C, the subset of features G remaining in the current iteration, along with the algorithm parameter K. The sub-process then returns a tuple containing the feature F_i itself, the computed approximate Markov blanket and the computed δ value.

$$
\begin{aligned}
subprocess(C, K, F_i, G) = \big(& \\
& F_i, \\
& MB_i = aMB(C, K, F_i, G), \\
& \delta_i = \delta(C, F_i, MB_i) \\
\big) &
\end{aligned}
\tag{19}
$$

The host machine on which the experiment was run has an 8-core processor. For this reason, when running the KS algorithm with the In-iteration Parallelism optimization enabled, it was configured to spawn 7 sub-processes. This brings the total of processes to 8, matching the number of processor cores of the machine. The task of mapping each process to each individual

processor core was left to the operating system during run-time. Attempts to facilitate this mapping consisted of freeing up system resources and to stopping non-essential software services, as already described in Section 4.. Even if these attempts were insufficient, the unoptimized and optimized runs of the algorithm were executed in conditions as similar as possible; therefore the comparison between them is meaningful.

The In-iteration Parallelism optimization was implemented using the class `multiprocessing.Pool`, available in the standard Python 3 library.

Table 2 shows the effect of configuring In-iteration Parallelism to use 7 processes, compared to the unoptimized run.

Table 2. Duration of Phase 2 for each document class, unoptimized versus optimized with In-iteration Parallelism.

Document class	**Duration of unoptimized run** (HH:MM:SS)	**Duration of optimized run** (HH:MM:SS)	**Duration of optimized run** (% of unopt. run)
0	19:57:07	10:31:16	52.73%
1	20:07:36	10:25:53	51.82%
2	20:01:07	10:27:46	52.26%
3	20:14:21	10:23:08	51.31%
4	20:07:22	10:26:31	51.89%
5	19:58:09	10:30:00	52.58%
6	20:01:48	10:29:48	52.40%
Total	140:27:30	73:14:22	52.14%

When the In-iteration Parallelism optimization was enabled and configured to spawn 7 subprocesses, the algorithm required, on average, only 52.14% of the duration required by the unoptimized algorithm. This empirically demonstrates that the KS algorithm indeed presents inherent data parallelism which can be successfully exploited to accelerate the algorithm's run.

5.4. Iteration Cache

In [1], Koller and Sahami mentioned the possibility of increasing the efficiency of the algorithm by storing the approximate Markov blankets of features across iterations, recalculating only those Markov blankets which contained the feature that was removed in the immediately previous iteration. Although they proposed this optimization, Koller and Sahami did not implement it in their algorithm validation experiments [1]. No other mention of this optimization was found in literature either. This optimization has been implemented and used by us in [2] under the name of Iteration Cache (IC) and its efficiency gain is evaluated here. This optimization targets Phase 2 of the KS algorithm.

The Iteration Cache is a caching mechanism which stores the approximate Markov blankets MB_i of every feature F_i for future iterations and keeps track of which feature belongs to which blankets in the cache. Thus, when a feature F_t is removed at the end of an iteration, the IC invalidates only the cached blankets that contained the feature F_t, while the unaffected blankets persist into the next iteration. The IC also stores the calculated $\delta(C, F_i, MB_i)$ of each F_i, avoiding its expensive computation.

The IC exploits the fact that a removed feature is unlikely to be part of many approximate Markov blankets. This leads to significant reuse of approximate Markov blankets across iterations, since very few of them need to be recalculated anew.

Table 3 shows the effect of enabling the Iteration Cache, compared to the unoptimized run.

Table 3. Duration of Phase 2 for each document class, unoptimized and optimized with Iteration Cache.

Document class	**Duration of unoptimized run** (HH:MM:SS)	**Duration of optimized run** (HH:MM:SS)	**Duration of optimized run** (% of unopt. run)
0	19:57:07	00:07:37	0.63%
1	20:07:36	00:05:10	0.42%
2	20:01:07	00:07:07	0.59%
3	20:14:21	00:05:00	0.41%
4	20:07:22	00:06:52	0.56%
5	19:58:09	00:06:52	0.57%
6	20:01:48	00:07:34	0.62%
Total	140:27:30	00:46:12	0.55%

Table 4 shows the average hit rates per iteration of the Iteration Cache per document class. A hit was counted when the approximate Markov blanket of a feature was found in the cache and did not require recalculation in the current iteration. A miss was counted whenever the approximate Markov blanket of a feature had to be calculated, as it was not found in the Iteration Cache.

Table 4. Average hit rates per iteration.

Document class	**Avgerage hit rate**
0	0.994
1	0.996
2	0.994
3	0.996
4	0.994
5	0.994
6	0.994
All	0.994

The Iteration Cache dramatically increases the efficiency of the KS algorithm. The running time of the algorithm with enabled Iteration Cache was, on average, 0.55% of the duration of the unoptimized run, which is remarkable. The efficiency gains of the IC stems from its high average hit rate, 0.994, which means that only very few approximate Markov blankets need to be recalculated after the first iteration, while most of the blankets, on average 99.4%, are found in the cache. This means that the Iteration Cache helps the algorithm to quickly eliminate the noise from the dataset. It will not be as effective, however, if the weakest features in the dataset are still strongly correlated with many other features (as determined by the γ heuristic). This would

cause many misses in the cache. Still, due to the sheer inefficiency of the default behavior of the KS algorithm, which is to recalculate the approximate Markov blankets of every remaining feature in every iteration, the Iteration Cache should provide efficiency gains regardless of the dataset.

6. Conclusions and further work

The four optimizations described in this article greatly increase the efficiency of Koller and Sahami's algorithm and also provide robustness in case of interruptions. They reduce the duration of both of its phases by avoiding redundant calculations, by exploiting the data parallelism inherent in the algorithm and by applying fundamental concepts from Information Theory to simplify its γ heuristic. The results of the presented experiment show that, with Gamma Decomposition and the Iteration Cache enabled, the algorithm requires only 4.36% of the duration of the unoptimized algorithm, on average.

The Iteration Cache optimization was shown to be the most effective Phase 2 optimization, reducing its duration to an average of 0.55% of the unoptimized duration, a remarkable gain in efficiency. This optimization has been originally proposed by Koller and Sahami themselves, although they did not implement it in their experiment.

The Gamma Decomposition optimization, although implemented here in the context of Phase 1 of the KS algorithm, is in fact a general optimization to compute conditional mutual information for large numbers of features. It could potentially be implemented in other algorithms which employ this information-theoretic measure.

An adaptation of Gamma Decomposition for a state-of-the-art Feature Selection algorithm with Markov blankets is currently being developed by us. We are also investigating the possibility of adapting the Iteration Cache to the needs of the state-of-the-art feature selection algorithms.

The Gamma Decomposition optimization also presents inherent data parallelism which could be exploited with further development, making it even more efficient. The Iteration Cache could also benefit from parallelization, although to a lesser extent: it can be combined with the Inter-iteration Parallelism optimization to simultaneously calculate multiple blankets that have been invalidated in the cache, reducing the penalty of a miss.

References

[1] D. Koller and M. Sahami, "Toward optimal feature selection," in *In 13th International Conference on Machine Learning*, 1995, pp. 284–292.

[2] C. Bancioiu and L. Vintan, "A comparison between two feature selection algorithms," in *Proceedings of ICSTCC 2017*, 2017, pp. 242–247.

[3] S. Fu and M. C. Desmarais, "Markov blanket based feature selection: a review of past decade," in *Proceedings of the world congress on engineering*, vol. 1. Newswood Ltd, 2010, pp. 321–328.

[4] D. Margaritis and S. Thrun, "Bayesian network induction via local neighborhoods," in *Advances in neural information processing systems*, 2000, pp. 505–511.

[5] I. Tsamardinos, C. Aliferis, A. Statnikov, and E. Statnikov, "Algorithms for large scale markov blanket discovery," in *In The 16th International FLAIRS Conference, St.* AAAI Press, 2003, pp. 376–380.

[6] J. M. Pena, R. Nilsson, J. Björkegren, and J. Tegnér, "Towards scalable and data efficient learning of markov boundaries," *International Journal of Approximate Reasoning*, vol. 45, no. 2, pp. 211–232, 2007.

[7] I. Tsamardinos, C. F. Aliferis, and A. Statnikov, "Time and sample efficient discovery of markov blankets and direct causal relations," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 673–678.

[8] C. F. Aliferis, I. Tsamardinos, and A. Statnikov, "Hiton: a novel markov blanket algorithm for optimal variable selection," in *AMIA Annual Symposium Proceedings*, vol. 2003. American Medical Informatics Association, 2003, p. 21.

[9] S. Fu and M. C. Desmarais, "Fast markov blanket discovery algorithm via local learning within single pass," in *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer, 2008, pp. 96–107.

[10] C. Lee and G. G. Lee, "Information gain and divergence-based feature selection for machine learning-based text categorization," *Information Processing & Management*, vol. 42, no. 1, pp. 155–165, Jan. 2006.

[11] I. Tsamardinos, C. Aliferis, A. Statnikov, and E. Statnikov, "Algorithms for large scale markov blanket discovery," in *In The 16th International FLAIRS Conference, St*. AAAI Press, 2003, pp. 376–380.

[12] E. Pasero, A. Montuori, W. Moniaci, and G. Raimondo, "An application of data mining to pm10 level medium-term prediction," Ph.D. dissertation, International Environmental Modelling and Software Society, 2008.

[13] Y. Saeys, "Feature selection for classification of nucleic acid sequences," Ph.D. dissertation, Ghent University, 2004.

[14] E. Statnikov, I. Tsamardinos, L. E. Brown, and C. F. Aliferis, "Causal explorer: A matlab library of algorithms for causal discovery and variable selection for classification," 2009.

[15] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *Journal of machine learning research*, vol. 5, pp. 361–397, 2004. [Online]. Available: http://www.jmlr.org/papers/v5/lewis04a.html

[16] "MBFF, an experimental framework for studying Markov Blanket Feature Filters," https://github.com/camilbancioiu/MBFF, 2017.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[18] G. Brown, A. Pocock, M.-J. Zhao, and M. Luján, "Conditional likelihood maximisation: a unifying framework for information theoretic feature selection," *Journal of machine learning research*, vol. 13, no. Jan, pp. 27–66, 2012.