Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

featsel: A Framework for Benchmarking of Feature Selection Algorithms and Cost Functions

Marcelo S. Reis
Butantan Institute

Gustavo Estrela
Butantan Institute
University of São Paulo

Carlos E. Ferreira
University of São Paulo

Junior Barrera
University of São Paulo

Abstract

In this paper, we describe **featsel**, a framework for benchmarking of feature selection algorithms and cost functions. This framework allows the user to deal with the search space as a Boolean lattice $(\mathcal{P}(S), \subseteq)$ and is coded in C++ for computational efficiency purposes. Moreover, **featsel** includes Perl scripts to add new algorithms and/or cost functions, as well as to generate random instances, to plot graphs and to organize results into HTML tables. We also provide two examples of how to use the framework to yield HTML tables and graphs with the obtained results, one using synthetic NP-hard instances and the other one employing a data set from the UCI Machine Learning Repository.

Keywords: feature selection, benchmarking, Boolean lattice, combinatorial optimization.

1. Introduction

In the context of Machine Learning and Statistics, feature selection is a procedure to select, from a set S of features, a subset $X \subseteq S$ such that X contains the most relevant features for a given classifier design process. If the relevance of X can be measured through a cost function $c : \mathcal{P}(S) \to \mathbb{R}^+$, then feature selection is reduced to a combinatorial optimization problem called *feature selection problem*, in which the objective is to minimize $c(X)$. It is a well-known fact that the feature selection problem is NP-hard (Guyon and Elisseeff 2003); however, in situations where a cost function c measures an estimation error and is computed with a fixed number of samples, a property of c arises due the "curse of dimensionality": adding new features to the considered subset X decreases the estimation error $c(X)$, until the point that the limitation of samples increases $c(X)$, resulting in a chain of subsets whose graph describes a U-shaped curve. That observation is taken into account in a special case of the feature selection problem: given an instance $\langle S, c\rangle$, if for every $X \subseteq Y \subseteq Z \subseteq S$ it holds that

$c(Y) \leq max\{c(X), c(Z)\}$, then $\langle S, c \rangle$ is also an instance of the *U-curve problem*. Although the U-curve problem is also NP-hard (Reis 2012), many feature selection algorithms rely on strategies that model the search space as an instance of this problem: among them there are suboptimal algorithms (i.e., algorithms that do not guarantee finding a global minimum) like the sequential forward search (SFS) (Whitney 1971), and also optimal ones, such as the U-curve branch-and-bound (UBB) (Ris, Barrera, and Martins-Jr. 2010; Reis 2012). However, in practical feature selection instances, subset chains do not have perfect U-shaped curves, since the former often present oscillations. Although depending the level of oscillations only an exhaustive search (ES) could guarantee a global minimum, there are some suboptimal algorithms, such as the sequential floating forward search (SFFS) (Pudil, Novovicová, and Kittler 1994), that were designed to circumvent that issue.

Nonetheless, different choices for the cost function c impact on the performance of a feature selection algorithm: for example, in instances whose chains have in their subset costs a decreasing tendency toward a global minimum (a condition that can be approximated by the Hamming distance cost function (Reis 2012)), then SFS would be the a good choice for feature selection. However, if the instances have their global minima distributed throughout the search space and their chains describe a perfect U-shaped curve (which can be replicated by a polynomial reduction from the subset sum problem to the U-curve problem (Reis 2012)), then the UBB algorithm would be suitable for an optimal search. Moreover, if these instances have oscillations in their chains (which is observed in practice when the mean conditional entropy is used to estimate morphological operators (Martins-Jr, Cesar-Jr, and Barrera 2006)), then the SFFS algorithm could be employed for a suboptimal search.

Once suitable choices of both the cost function and the algorithm have relevant impact on the performance of the feature selection procedure, new approaches for both are continually proposed by the academic community. However, generally these new algorithms and/or cost functions are introduced with their own implementations, and very often using different programming languages and/or data structures, which makes difficult experimental benchmarking of them, especially in situations where constant and/or polynomial factors associated to the implementation are not overwhelmed by the asymptotic complexity of both algorithm and cost function. Therefore, there is a need for a standardized environment to test different algorithms and cost functions, especially to compare new proposed solutions against well-established ones, the so-called "gold standards".

In this paper, we introduce **featsel**, an open-source framework for benchmarking of feature selection algorithms and cost functions. The core of this framework was coded in C++ and allows the user to deal with the search space as a Boolean lattice $(\mathcal{P}(S), \subseteq)$, which is very helpful since a subset X of the set of features S can be efficiently described as a characteristic vector of X, hence Boolean operations can be applied on it. Moreover, **featsel** includes auxiliary Perl scripts to minimize the efforts in adding new algorithms and/or cost functions, generating random instances, plotting graphs and organizing the benchmarking results into hypertext markup language (HTML) tables. The framework is under GNU GPLv3 license and is available for download at `github.com/msreis/featsel`. The remainder of this paper is organized as follows: in Section 2, we present a high-level description of the framework components and main features. In Section 3, we explain step-by-step how to perform the most important processes during the setting up and execution of two benchmarking experiments, one with synthetic data and the other with a data set from a Machine Learning repository. In Section 4, we show some examples, to this end using algorithms and cost functions that are already available with the initial release of the framework. Finally, in Section 5, we make some conclusion remarks about this work and point out ideas for future improvements.

2. Description of the framework

The core of **featsel** was designed through class-based, object-oriented modeling. The chosen object-oriented programming language to code the framework was C++ (Stroustrup 1995): since benchmarking is our main objective, from the computational efficiency point of view C++ has a better payoff in comparison with interpreted languages such as R (R Core Team 2017), which is relevant to reduce constant factors that could impact the comparison among algorithms and cost functions.

The main object interactions in the framework are as follows: Features are elements of the system. The aggregation of several elements yields a set, while each set can be associated to a subset of it. Subsets can be aggregated into collections of subsets, and also are associated to cost functions. Solvers are composed of collections of subsets (e.g., to store lists of visited subsets) and of a cost function to compute a subset cost. Both cost function and solver are abstract classes, which means that they serve as a basis for concrete implementations of cost functions and algorithms, respectively.

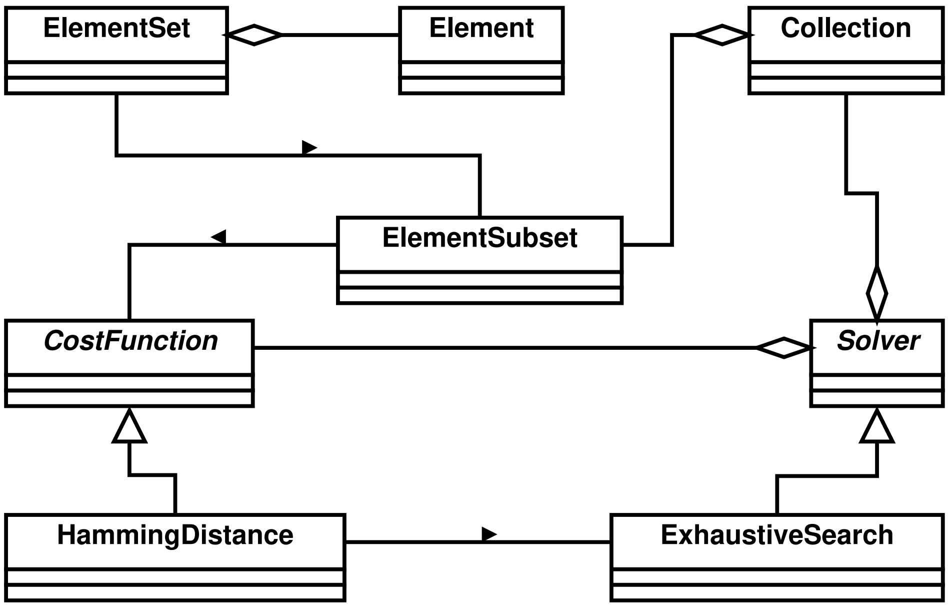


Figure 1: Class diagram in unified modeling language (UML) of **featsel**, in which it is exemplarily showed the derivation of a cost function concrete class *HammingDistance* and of an algorithm concrete class *ExhaustiveSearch*. An object of *ExhaustiveSearch* can be instantiated using an object of *HammingDistance* as the cost function.

In figure 1, we summarize these interactions into a class diagram, which contains six main classes; the most relevant properties of these classes are discussed in the following.

Element. This class represents a feature, and has attributes and methods to store and retrieve its relevant properties: for example, if a feature consists in a pixel of a window during a morphological operator estimation using k samples, then an object of this class stores an array of k integers, each one corresponding to the observed value for that pixel in each of those samples.

ElementSet. An aggregation of elements that compose the complete set S considered during the feature selection procedure. This class has methods to load element data of a given set S from either `dat` (flat file) or `xml` (extended markup language) instance files; to this end, it relies on auxiliary parser classes *DatParserDriver* and *XmlParserDriver*, respectively.

ElementSubset. A subset X of a complete set S is represented as an instance of this class, which is used to explicitly represent its corresponding node in the search space and also to compute the cost of X. The subset representation is accomplished through the usage of the characteristic vector χ^X of X, which is defined as:

$$\chi_s^X = \begin{cases} 1, & \text{if } s \in X; \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

for every element $s \in S$. Once this representation of the subset is binary, Boolean operations can be performed on χ^X through methods of this class that implement union, intersection and complement operations.

Collection. This class implements a collection of objects of the class *ElementSubset*. The main objective of this class is to provide a way to store subsets of the search space $(\mathcal{P}(S), \subseteq)$ that were already visited, and also to register the subsets of minimum cost found along a search procedure. To this end, this class has methods to add, remove and search subsets stored in a given collection.

CostFunction. This is an abstract class, that is, a class that serves as a "scaffolding" for the implementation of a cost function within the framework. To this end, the cost function concrete class inherits from *CostFunction* a virtual constructor and also a virtual method to compute the cost of a given subset; then the user must program the internal logic of her/his own cost function. For instance, for the *HammingDistance* class depicted in Figure 1, one can compute and return the Hamming distance between a subset X received as argument of the *cost* method against a constant subset Y received as argument by the constructor of that class.

Solver. An abstract class used to assist the implementation of an algorithm within the framework. An algorithm concrete class inherits a virtual method called *get_minima_list*, which launches a search whose exact procedure must be programmed by the user. The object representing the cost function to be used during the search must be passed to the algorithm as an argument of a method called *set_parameters*, which in turn must be called before *get_minima_list*. For example, according to the diagram depicted in Figure 1, an instantiated object of *ExhaustiveSearch* can be initialized with an object of *HammingDistance* and then compute the cost function for all $X \subseteq S$; in this case, the element of minimum cost will be the subset $Y \subseteq S$ that was used to instantiate the cost function object.

3. Usage of the framework

We describe now how to set up the **featsel** framework for execution of the main program, and also how to properly add and remove cost functions and algorithms, thus making it ready to carry out a benchmarking experiment.

3.1. Setting up

After either download this framework or clone its repository, open a terminal and type within the directory where it was stored:

```
> make
```

This command will compile **featsel** main file with the standard algorithms and cost functions that come with the framework. To check out the correctness of the framework classes, the user can perform unit tests on them. To this end, type the commands:

```
> make test
> bin/featselTest
```

After these two commands, **featsel** should successfully execute 67 unit tests and display their results into the standard output.

3.2. Running the main program

In order to run the main program, the most important arguments are the following:

- `-f` Together with a string, it defines the instance file path (relative to the framework main directory) and name (file type is either `dat` or `xml`). Example: `-f input/Test_1_2.xml`.
- `-c` This argument is accompanied by a code corresponding to a cost function. The available codes in the initial release are `explicit` (*Explicit*), `hamming_distance` (*HammingDistance*), `mce` (*MeanConditionalEntropy*), and `subset_sum` (*SubSetSum*).
- `-a` It is accompained by an algorithm code. The available codes in the initial release are `es` (*ExhaustiveSearch*), `sffs` (*SFFS*), `sfs` (*SFS*), and `ubb` (*UcurveBranchandBound*).
- `-m` Number of minima (i.e., number of subsets with the lowest cost among all subsets explored during a given search) that should be listed into the standard output.
- `-h` Display a help screen, containing explanations for the arguments listed above and also for additional ones.

Here follows a simple example of the main program usage: consider an instance $S = \{a, b, c\}$, which induces a search space with 8 subsets; assume that the cost function c of this instance is explicitly declared (i.e., c is an instance of the *Explicit* cost function) and whose path and file are `input/explicit/` and `Test_03_B.xml`, respectively. If the user wants to run the SFS algorithm and store the best 3 subsets found during the search, then he/she can type:

```
> bin/featsel -f input/explicit/Test_03_B.xml -c explicit -a sfs -m 3
```

The output of this call should be like this one:

```
== List of best subsets found ==
 X : c(X)
 <101> : 1
 <001> : 3
 <000> : 5

Number of visited subsets: 7
Required time to compute the visited subsets: 468 microseconds
(average 66 microseconds per node)

Elapsed time of execution of the algorithm (in microseconds): 549

== End of processing ==
```

In the output above, the first column is a subset X of the set S, represented by its characteristic vector χ^X, while the second column is the cost of X, that is, the value $c(X)$; in this example, $\chi^{\{a,c\}} = 101$ was the best subset found, since $c(\{a, c\}) = 1$ was the lowest value stored by the program. The output also delivers the number of visited subsets, which corresponds to the number of times c was computed, as well as the elapsed total time of the algorithm execution and also the fraction of that total time that was spent to compute all calls of c.

3.3. Adding and removing algorithms and cost functions

To include new algorithms and cost functions into **featsel**, there are four auxiliary Perl scripts (Wall, Christiansen, and Orwant 2000). To add a new algorithm, type:

```
> perl bin/add_new_algorithm.pl code AlgorithmName
```

where `code` is a code for the algorithm, which will be passed as argument to the main program, and *AlgorithmName* is the name of the algorithm concrete class that will be derived from the abstract *Solver* class. If the user successfully includes a new algorithm, say Foo, then the script will update **featsel** Makefile, main and test files, and the following files will be included into the framework directory tree:

```
 |---\src\
 |    |---\algorithms\
 |        |---\Ping.cpp
 |        |---\Ping.h
 |
 |---\test\
 |    |---\algorithms\
 |        |---\PingTest.cpp
 |        |---\PingTest.h
```

To remove an algorithm, the user just need to type:

```
> perl bin/remove_algorithm.pl code AlgorithmName
```

To add a new cost function, the syntax is:

```
> perl bin/add_new_cost_function.pl code CostFunctionName file_type
```

where `code` is a code for the cost function, which will be passed as argument to the main program, *CostFunctionName* is the name of the cost function concrete class that will be derived from the abstract *CostFunction* class, and `file_type` may be either `dat` or `xml`. If the user successfully includes a new cost function, say Bar, then the script will update **featsel** Makefile, main and test files, and the following files will be included into the framework directory tree:

```
|---\lib\
|   |---\Bar.m
|
|---\src\
|   |---\functions\
|       |---\Bar.cpp
|       |---\Bar.h
|
|---\test\
|   |---\functions\
|       |---\BarTest.cpp
|       |---\BarTest.h
```

Once a new cost function is added, then it can be used straightforwardly in the benchmarking program if the user put manually them into the `/input/tmp` directory (the files must be of the chosen file type). However, to allow the benchmarking auxiliary script to generate random instances of the newly included cost function, he/she must edit a subroutine to this end within the file `/lib/Bar.m`; the subroutine itself will be named as "*random_code_instance*", where *code* is the code of the cost function Bar. Finally, to remove a cost function, the user must type:

```
> perl bin/remove_cost_function.pl code CostFunctionName
```

There is no need to specify the file type in a removal of cost function.

4. Examples of application

In this section, we show two exemplary benchmarking experiments with **featsel**: The first one used synthetic instances derived from a polynomial reduction of a NP-hard problem, while the second one relied on a data set from the UCI Machine Learning Repository (Lichman 2013). These two experiments were carried out in a computer with four Intel® i5-6300U cores at 2.40 GHz each, 8 GB RAM and Ubuntu 16.04 LTS operating system.

The benchmarking experiments were assisted by an auxiliary Perl script that wraps the framework main program, filter its output and organizes the final results into tables and graphs; the syntax this script will be presented in the following.

4.1. Running the benchmarking auxiliary script

In order to run the auxiliary Perl script, the syntax is:

```
> perl bin/run_benchmarking.pl  OUTPUT_FILE_PREFIX  INSTANCE_MODE \
  COST_FUNCTION_CODE  k  n   SEARCH_MODE  [max_number_of_calls]
```

The arguments of the command above are:

`OUTPUT_FILE_PREFIX`: Prefix for the names of the output files, consisting in a HTML table and also in Scalable Vector Graphics (svg) graphs (one graph for each algorithm).

`COST_FUNCTION_CODE`: Code of the employed cost function; in the initial release of this framework, it should be one of the following: `explicit`, `hamming_distance`, `mce`, or `subset_sum`.

`INSTANCE_MODE`: Must be 0 or 1; if the value is 0, then for each size of instance in $[1, \texttt{n}]$, it creates `k` random instances of the chosen cost function. Otherwise, it reads input files that were created previously; it assumes that the instance size ranges from 1 to `n`, and also that there are `k` files per instance size.

`k`: Number of instances per instance size.

`n`: Size of the largest instance.

`SEARCH_MODE`: Must be 0 or 1; if the value is 0, then it performs a complete search, which may be optimal or not depending on the algorithm. Otherwise, it runs a search constrained by `max_number_of_calls` calls of the cost function.

In the following exemplary experiments, we show two different usages of this script.

4.2. Subset sum instances

Let t be an integer, S be a set of integers, and $\langle S, c\rangle$ be an instance of the U-curve problem such that the cost function c is defined as:

$$c(X) = |t - \sum_{x \in X} x|, \tag{2}$$

for all $X \subseteq S$. To find a subset X of minimum cost using Equation 2 is a NP-hard problem (Reis 2012), which makes this cost function interesting to test the performance of feature selection algorithms. Hence, we coded this cost function as a **featsel** cost function concrete class (class *SubsetSum*), whose code is `subset_sum` and file type is `xml`. Now we will use it to carry out a benchmarking experiment among four algorithms that were also implemented in the framework: ES, SFFS, SFS and UBB. In this assay, we will generate random instances with different sizes for S; for each instance size $i := |S|$, $1 \leq i \leq 20$, we will execute the following steps:

1. generate ten instances of size i;
2. for each of these ten instances, run each algorithm, recording the required computational time (total time and c computation time) and number of times c is computed;

3. average the results for each attribute and for each algorithm.

To launch the aforementioned benchmarking experiment, the user just need to call the benchmarking auxiliary script with the following arguments:

```
perl bin/run_benchmarking.pl FirstExample 0 subset_sum 10 20 0
```

In the command above, `FirstExample` is the prefix of the files that summarize the results, the first `0` means that random instances will be generated on-the-fly by the script, `subset_sum` is the cost function code, `10` is the number of instances per instance size, `20` means that the instance size range is $[1, 20]$, and the second `0` means that each algorithm will run until it reaches its default stop criterion.

After execution, it was created a table within of a HTML file called `FirstExample_table.html` (Table 1). From the computational point of view, SFFS and SFS were faster than ES and UBB, which was expected since the former and the latter are suboptimal and optimal algorithms, respectively. However, from the semantic point of view, ES and UBB always found a global minimum, while the performance of SFFS and SFS went increasingly worse as a function of instance size, which was also expected. Moreover, UBB always found a global minimum computing fewer times the cost function and also requiring less computational time than ES. These properties can be better visualized through four graphs automatically created by the benchmarking program, depicting the performance of each algorithm as a function of the instance size (Figures 2a–2d).

4.3. Zoo data set

In this example, we will use the Zoo Data Set, which was obtained from the UCI Machine Learning Repository; this data set contains 15 Boolean features, seven classes and 101 samples. For this assay, we will employ as the cost function the mean conditional entropy, which is defined as follows: let X be a subset of a feature set S, $\mathbf{X}$ be a random variable in $\mathcal{P}(X)$, and $P(Y = y|\mathbf{X} = \mathbf{x})$ be the conditional probability of $Y = y$ given $\mathbf{X} = \mathbf{x}$. The conditional entropy of a random variable Y given $\mathbf{X} = \mathbf{x}$ is:

$$H(Y|\mathbf{X} = \mathbf{x}) = -\sum_{y \in Y} P(Y = y|\mathbf{X} = \mathbf{x}) \log P(Y = y|\mathbf{X} = \mathbf{x}). \tag{3}$$

Departing from Equation 3, we can estimate the mean conditional entropy of Y given $\mathbf{X}$; in this estimation, we will penalize pairs of values $\langle y, \mathbf{x} \rangle$ that have a unique observation, since these rarely observed pairs may be underrepresented: they are considered as having a uniform distribution, thus leading to the highest entropy for these cases. Therefore, the estimation of mean conditional entropy with penalization is:

$$\hat{E}[H(Y|\mathbf{X})] = \frac{N}{t} + \sum_{\mathbf{x} \in \mathbf{X} : \hat{P}(\mathbf{x}) > \frac{1}{t}} \hat{H}(Y|\mathbf{X} = \mathbf{x}) \hat{P}(\mathbf{X} = \mathbf{x}), \tag{4}$$

where N is the number of values of $\mathbf{X}$ with a single occurrence in the samples and t is the total number of samples. We implemented Equation 4 as a cost function concrete class in **featsel** (class *MeanConditionalEntropy*), whose code is `mce` and file type is `dat`. To carry out this benchmarking experiment, we discarded one sample and split the remaining 100 samples into two groups of 50 samples each; for each group, we created 15 instances (`dat` files), the first with one feature, the second with two features, and so forth. The resulting 30 `dat` files were stored into `input/tmp` and the following command was executed:

```
perl bin/run_benchmarking.pl SecondExample 1 mce 2 15 0
```

In the command above, `SecondExample` is the prefix of the files that summarize the results, `1` means that instances will be read from the `input/tmp` directory, `mce` is the cost function code, `2` is the number of instances per instance size, `15` means that the instance size range is $[1, 15]$, and `0` means that each algorithm will run until it reaches its default stop criterion.

After execution, it was created a table in a HTML file called `SecondExample_table.html` (Table 2). As expected, SFFS and SFS were faster than ES and UBB. However, although ES and UBB always found a global minimum, SFFS had a better performance in comparison with the first experiment, losing a global minimum only once for instances of size 9. Moreover, UBB always found a global minimum computing fewer times the cost function and also requiring less computational time than ES, with greater advantage in comparison with the first experiment. These properties also can be analyzed through four graphs generated automatically by the benchmarking program (Figures 3a–3d).

5. Conclusion

In this paper, we introduced **featsel**, an open-source, C++ coded framework to assist the benchmarking of algorithms and cost functions that are used to solve the feature selection problem. **featsel** includes auxiliary Perl scripts to assist the adding of new algorithms and/or cost functions, generating random instances, plotting graphs and organizing the benchmarking results into HTML tables. Additionally, we provided two examples of benchmarking experiments with **featsel**, one using synthetic instances and a second one using a data set from the UCI Machine Learning Repository.

Currently, we are working on the addition of new algorithms and cost functions, as well as in making available new types of graphs. Other ongoing improvements on **featsel** include the usage of reduced ordered binary decision diagrams to represent the search space (Bryant 1986), as well as the inclusion into the framework of methods for implementation of parallelized algorithms, which will be achieved using the **OpenMP** library (Dagum and Menon 1998). Finally, we plan to make the auxiliary scripts also available in R programming language.

Acknowledgements

This work was supported by scholarship #142039/2007-1 and grant #305705/2014-8 from CNPq, and also by grants #2013/03447-6, #2013/07467-1 and #2014/23564-0, São Paulo Research Foundation (FAPESP).

References

Bryant RE (1986). “Graph-based Algorithms for Boolean Function Manipulation.” *IEEE Transactions on Computers*, **100**(8), 677–691.

Dagum L, Menon R (1998). "**OpenMP**: An Industry Standard API for Shared-Memory Programming." *Computational Science & Engineering, IEEE*, **5**(1), 46–55.

Guyon I, Elisseeff A (2003). "An Introduction to Variable and Feature Selection." *Journal of Machine Learning Research*, **3**(Mar), 1157–1182.

Lichman M (2013). "UCI Machine Learning Repository." URL http://archive.ics.uci.edu/ml.

Martins-Jr D, Cesar-Jr R, Barrera J (2006). "W-operator Window Design by Minimization of Mean Conditional Entropy." *Pattern Analysis and Applications*, **9**(2), 139–153.

Pudil P, Novovicová J, Kittler J (1994). "Floating Search Methods in Feature Selection." *Pattern Recognition Letters*, **15**(11), 1119–1125.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org.

Reis M (2012). *Minimization of Decomposable in U-shaped Curves Functions Defined on Poset Chains – Algorithms and Applications*. Ph.D. thesis, Institute of Mathematics and Statistics, University of São Paulo, Brazil. (in Portuguese).

Ris M, Barrera J, Martins-Jr D (2010). "U-curve: A Branch-and-bound Optimization Algorithm for U-shaped Cost Functions on Boolean Lattices Applied to the Feature Selection Problem." *Pattern Recognition*, **43**(3), 557–568.

Stroustrup B (1995). *The C++ Programming Language*. Pearson Education India.

Wall L, Christiansen T, Orwant J (2000). *Programming Perl*. O'Reilly Media.

Whitney A (1971). "A Direct Method of Nonparametric Measurement Selection." *IEEE Transactions on Computers*, **20**(9), 1100–1103.

Affiliation:

Marcelo da Silva Reis
Center of Toxins, Immune-response and Cell Signaling (CeTICS)
Laboratório Especial de Ciclo Celular (LECC)
Instituto Butantan
Avenida Vital Brasil, 1500, ZIP 05503-900, São Paulo, Brazil
E-mail: marcelo.reis@butantan.gov.br
Telephone/Fax: +55/11/2627-9731

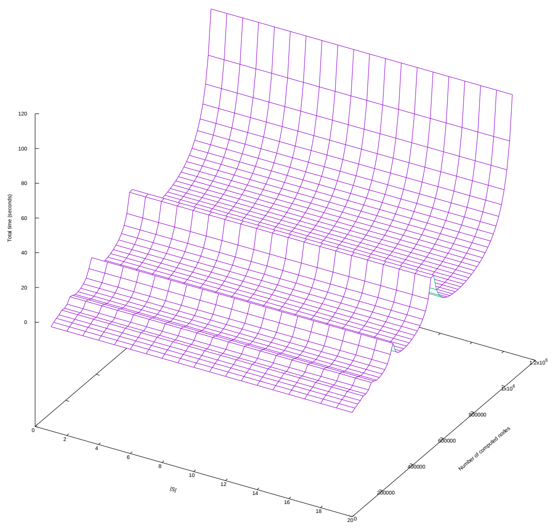
Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/

Volume VV, Issue II *Submitted:* 2017-01-11
MMMMMM YYYY *Accepted:* yyyy-mm-dd

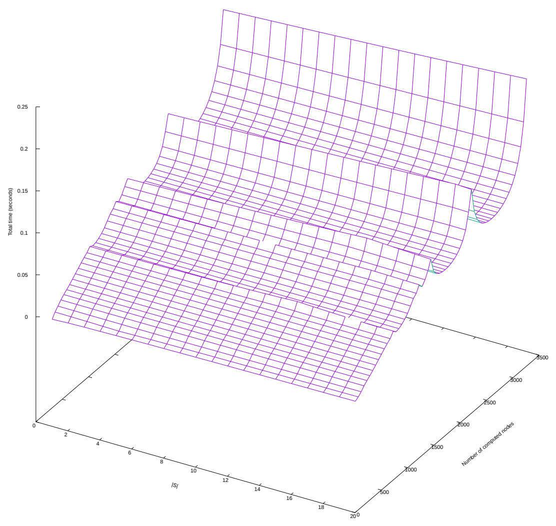
Instance		Total time (sec)				Cost function time (sec)			
$\lvert S\rvert$	$2^{\lvert S\rvert}$	UBB	ES	SFS	SFFS	UBB	ES	SFS	SFFS
1	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
5	32	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00
6	64	0.01	0.01	0.00	0.01	0.00	0.00	0.00	0.01
7	128	0.01	0.01	0.00	0.01	0.00	0.01	0.00	0.00
8	256	0.02	0.02	0.01	0.01	0.01	0.01	0.00	0.01
9	512	0.04	0.04	0.01	0.02	0.02	0.03	0.00	0.02
10	1024	0.05	0.08	0.01	0.02	0.03	0.06	0.00	0.01
11	2048	0.12	0.18	0.01	0.04	0.07	0.12	0.00	0.04
12	4096	0.29	0.34	0.01	0.05	0.19	0.23	0.00	0.04
13	8192	0.42	0.73	0.01	0.05	0.27	0.47	0.00	0.04
14	16384	0.88	1.46	0.01	0.06	0.61	0.93	0.00	0.04
15	32768	1.74	2.93	0.01	0.10	1.13	1.85	0.00	0.08
16	65536	4.79	6.39	0.01	0.10	2.78	3.67	0.01	0.08
17	131072	5.41	12.13	0.01	0.09	3.31	7.31	0.00	0.07
18	262144	13.17	24.29	0.01	0.07	8.17	14.57	0.01	0.06
19	524288	26.84	49.38	0.01	0.15	16.05	29.12	0.01	0.13
20	1048576	75.96	104.37	0.01	0.22	44.51	58.26	0.01	0.19

Instance		# Computed notes				# Best solution			
$\lvert S\rvert$	$2^{\lvert S\rvert}$	UBB	ES	SFS	SFFS	UBB	ES	SFS	SFFS
1	2	2.0	2.0	2.0	6.0	10	10	10	10
2	4	3.8	4.0	4.0	11.2	10	10	10	10
3	8	6.9	8.0	6.1	13.4	10	10	9	9
4	16	13.2	16.0	9.8	32.9	10	10	8	8
5	32	26.3	32.0	13.2	59.0	10	10	7	7
6	64	49.1	64.0	16.6	97.5	10	10	5	5
7	128	92.0	128.0	21.6	79.9	10	10	4	5
8	256	175.0	256.0	27.8	127.6	10	10	3	3
9	512	372.1	512.0	33.0	260.8	10	10	2	2
10	1024	607.0	1024.0	34.5	190.2	10	10	4	4
11	2048	1312.8	2048.0	46.9	662.6	10	10	2	2
12	4096	3257.7	4096.0	59.4	741.0	10	10	0	0
13	8192	4792.4	8192.0	56.1	667.5	10	10	2	2
14	16384	10807.0	16384.0	69.5	722.2	10	10	1	1
15	32768	20250.5	32768.0	83.4	1369.5	10	10	1	2
16	65536	49513.3	65536.0	96.3	1439.2	10	10	2	2
17	131072	59277.8	131072.0	82.5	1202.0	10	10	1	1
18	262144	147266.8	262144.0	97.9	1095.5	10	10	0	0
19	524288	290012.4	524288.0	114.8	2201.3	10	10	2	2
20	1048576	802959.2	1048576.0	152.5	3257.3	10	10	0	0

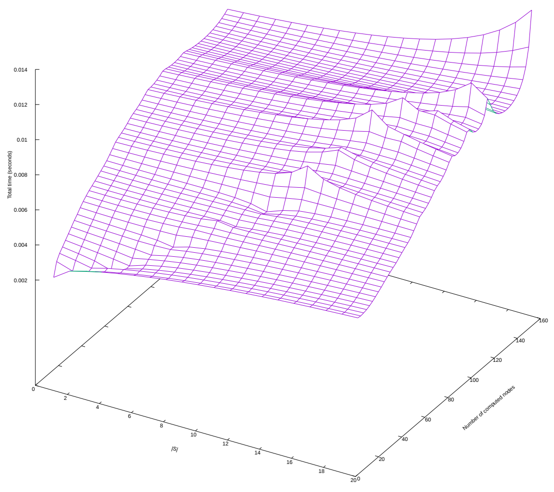
Table 1: Table created in the first example of benchmarking experiment, a feature selection procedure on synthetic NP-hard instances. For each instance size, the four algorithms (ES, SFFS, SFS and UBB) are compared against each other in four criteria: total required computational time, computational time required to compute all calls of the cost function, number of computed nodes (i.e., number of times the cost function was computed) and number of times the algorithm had a best solution (i.e., found a global minimum).



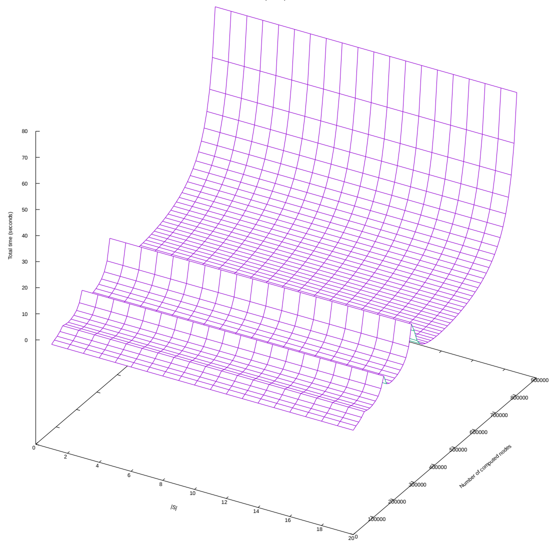
(a)



(b)



(c)



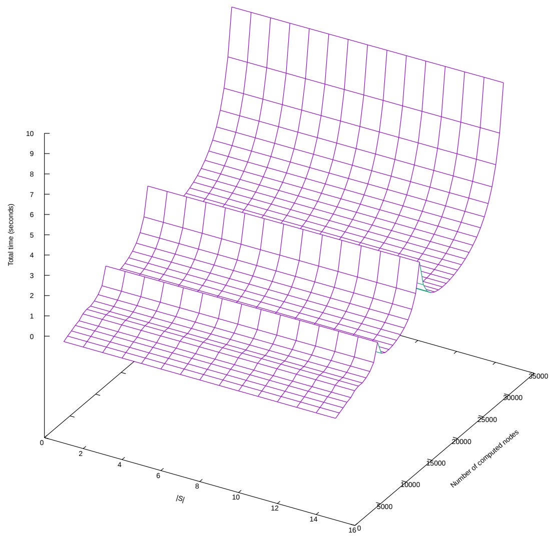
(d)

Figure 2: 3D graphs depicting the performance in the first example of benchmarking experiment, a feature selection procedure on synthetic NP-hard instances. The evaluated algorithms are ES (Figure 2a), SFFS (Figure 2b), SFS (Figure 2c) and UBB (Figure 2d). For each graph, it is plotted a surface showing the required computational time and the number of cost function calls as a function of instance size.

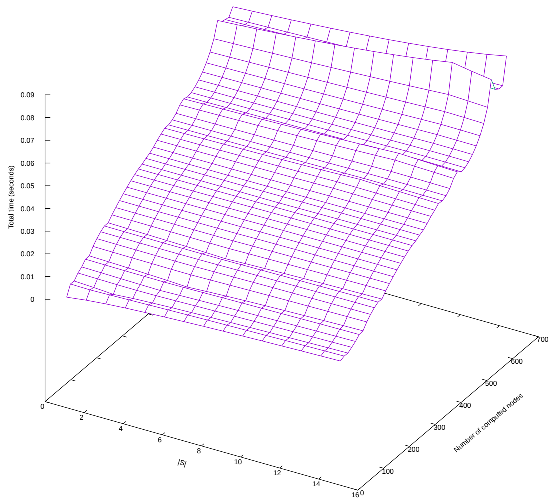
Instance		Total time (sec)				Cost function time (sec)			
$\|S\|$	$2^{\|S\|}$	UBB	ES	SFS	SFFS	UBB	ES	SFS	SFFS
1	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	8	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00
4	16	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00
5	32	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00
6	64	0.01	0.01	0.00	0.01	0.01	0.01	0.00	0.01
7	128	0.02	0.02	0.00	0.02	0.01	0.01	0.00	0.02
8	256	0.03	0.04	0.01	0.03	0.03	0.03	0.00	0.03
9	512	0.07	0.08	0.01	0.03	0.06	0.07	0.00	0.03
10	1024	0.13	0.16	0.01	0.04	0.12	0.15	0.01	0.04
11	2048	0.28	0.38	0.01	0.04	0.26	0.35	0.01	0.04
12	4096	0.51	0.85	0.02	0.05	0.46	0.78	0.01	0.05
13	8192	0.96	2.01	0.01	0.09	0.87	1.85	0.01	0.08
14	16384	1.63	4.21	0.01	0.07	1.48	3.90	0.01	0.07
15	32768	2.20	9.53	0.02	0.08	2.00	8.90	0.01	0.08

Instance		# Computed notes				# Best solution			
$\|S\|$	$2^{\|S\|}$	UBB	ES	SFS	SFFS	UBB	ES	SFS	SFFS
1	2	2.0	2.0	2.0	8.0	2	2	2	2
2	4	4.0	4.0	4.0	16.0	2	2	2	2
3	8	8.0	8.0	7.0	38.0	2	2	2	2
4	16	16.0	16.0	11.0	43.0	2	2	2	2
5	32	31.0	32.0	15.5	60.0	2	2	2	2
6	64	61.0	64.0	21.0	94.5	2	2	2	2
7	128	121.0	128.0	27.0	168.5	2	2	2	2
8	256	233.0	256.0	35.0	296.0	2	2	2	2
9	512	455.0	512.0	41.5	321.5	2	2	1	1
10	1024	895.5	1024.0	49.5	369.5	2	2	1	2
11	2048	1720.5	2048.0	56.5	390.0	2	2	1	2
12	4096	2848.5	4096.0	66.5	455.5	2	2	1	2
13	8192	4817.0	8192.0	74.0	596.5	2	2	0	2
14	16384	7947.0	16384.0	81.5	624.0	2	2	0	2
15	32768	11164.5	32768.0	89.0	651.5	2	2	0	2

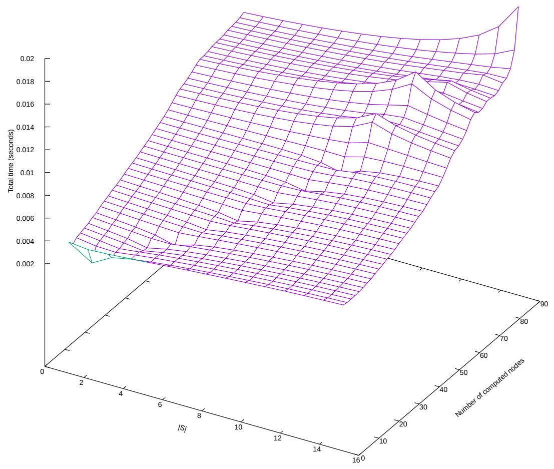
Table 2: Table created in the second example of benchmarking experiment, a feature selection procedure on the Zoo Data Set. For each instance size, the four algorithms (ES, SFFS, SFS and UBB) are compared against each other in four criteria: total required computational time, computational time required to compute all calls of the cost function, number of computed nodes (i.e., number of times the cost function was computed) and number of times the algorithm had a best solution (i.e., found a global minimum).



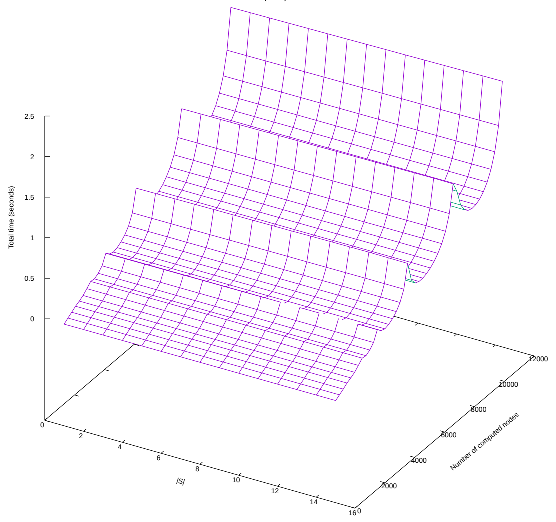
(a)



(b)



(c)



(d)

Figure 3: 3D graphs depicting the performance in the second example of benchmarking experiment, a feature selection procedure on the Zoo Data Set. The evaluated algorithms are ES (Figure 3a), SFFS (Figure 3b), SFS (Figure 3c) and UBB (Figure 3d). For each graph, it is plotted a surface showing the required computational time and the number of cost function calls as a function of instance size.