Towards a Framework for Designing Full Model Selection and Optimization Systems

Quan Sun, Bernhard Pfahringer and Michael Mayo

Department of Computer Science
The University of Waikato
Hamilton, New Zealand
`{qs12,bernhard,mmayo}@cs.waikato.ac.nz`

Abstract. People from a variety of industrial domains are beginning to realise that appropriate use of machine learning techniques for their data mining projects could bring great benefits. End-users now have to face the new problem of how to choose a combination of data processing tools and algorithms for a given dataset. This problem is usually termed the Full Model Selection (FMS) problem. Extended from our previous work [10], in this paper, we introduce a framework for designing FMS algorithms. Under this framework, we propose a novel algorithm combining both genetic algorithms (GA) and particle swarm optimization (PSO) named GPS (which stands for **G**A-**P**SO-FM**S**), in which a GA is used for searching the optimal structure for a data mining solution, and PSO is used for searching optimal parameters for a particular structure instance. Given a classification dataset, GPS outputs a FMS solution as a directed acyclic graph consisting of diverse data mining operators that are available to the problem. Experimental results demonstrate the benefit of the algorithm. We also present, with detailed analysis, two model-tree-based variants for speeding up the GPS algorithm.

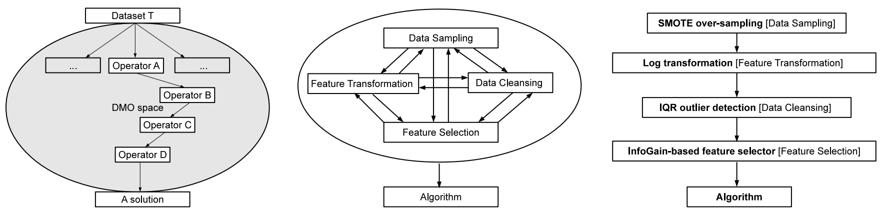
1 Introduction

Machine learning users now have to face the new problem of how to choose a combination of data processing tools and algorithms. The goal is usually defined as maximizing or minimizing a quantitative measure. In classification problems, the goal could be optimising the classification accuracy, the Lift score or the ROC area (AUC); in regression problems the goal could be optimising RMSE (root mean squared error), MAE (mean absolute error), or any proper loss function.

Sometimes the final goal might be a combination of multiple goals. Traditionally, these problems are addressed separately in the feature selection, model or parameter selection and the meta-learning fields. A practical data mining problem consists of many sub-problems which presents an extremely large search space that could be a very time-consuming task for humans to explore manually. Therefore, strategies and methods that can help people to choose, or that could automatically suggest, an optimised data mining solution is useful. In this paper, we propose a framework which can be used for designing new FMS algorithms, and we also present a novel FMS algorithm which is a realization and an application of the proposed framework.

1.1 Data Mining in the DMO Space and Framework

We first define the DMO space and discuss potential approaches for searching the space. We here attempt to define a search space that consists of all data mining actions (operators) that are available to a given dataset for a user-specified goal, such as a set of outlier filters, a set of feature selection methods, a set of data transformation techniques and a set of base learning algorithms. In this sense, we call the subject of interest "the space of data mining operators (DMO)", or simply "the DMO space" [10].


Dataset T
...
Operator A
...
Operator B
DMO space
Operator C
Operator D
A solution
Data Sampling
Feature Transformation
Data Cleansing
Feature Selection
Algorithm
SMOTE over-sampling [Data Sampling]
Log transformation [Feature Transformation]
IQR outlier detection [Data Cleansing]
InfoGain-based feature selector [Feature Selection]
Algorithm

(a) An illustration of the DMO space

(b) A graphical representation of the DMO template used by GPS

(c) A graphical representation of a DMO solution template instance

Fig. 1. A full model defined by the GPS algorithm

In this search space, a data mining solution is abstracted as a directed acyclic graph (DAG) consisting of DMOs that are connected based on some relations: see Figure 1 (a) for an illustration. For simplicity, in Figure 1 (a) we consider that an optimal data mining solution is given by a DAG defined by four DMOs (A, B, C and D) for dataset T. The DMO space is represented by the largest oval, which consists of all DMOs applicable to T. The directed arrows represent the relationships (action rules) in the DAG. If Operator A is an outlier filter, Operator B is a feature reduction method, Operator C is a decision tree algorithm, and Operator D is a post-processing method, the DAG can be interpreted as follows: given a dataset T, in an optimal solution we first use the outlier detection method (DMO A) to remove outliers, and then we employ the feature selection method (DMO B) to remove useless features, and then build a decision tree model (DMO C), and finally, we use a probability calibration method (DMO D) to calculate the model outputs. This is a very large search space because in theory there exists an arbitrary number of DMOs (including an arbitrary number of link directions, node orders and arrangements). Therefore, the next question is how to search in this space?

In practice, due to the resources at hand, usually we do not search in an infinite DMO space, and, moreover, we can make the DMO space a finite space by

defining the DMOs that are to be included. For example, given a dataset T, and given we have one outlier detection algorithm, two feature selection methods, three classification algorithms and that the goal is to build a model that gives the lowest classification error on T, typically, we can define the following *node* type DMO objects:
$DMO_{filter}, DMO_{no-filter}, DMO_{feature-selection-1}$,
$DMO_{feature-selection-2}$, $DMO_{no-feature-selection}$,
$DMO_{algorithm-1}, DMO_{algorithm-2}, DMO_{algorithm-3}$.

Given these DMOs, if we want to preprocess the data, we can define some *function* type DMOs that output a new data object. For example:
data $\Longleftarrow \quad DMO_{preprocessing-1}(DMO_{filter}, DMO_{feature-selection-1})$
data $\Longleftarrow \quad DMO_{preprocessing-2}(DMO_{no-filter}, DMO_{feature-selection-1})$
data $\Longleftarrow \quad DMO_{preprocessing-3}(DMO_{filter}, DMO_{feature-selection-2})$
...

where $DMO_{preprocessing-1,2,3}$ are *function* type DMOs. We can also define more complex *function* type DMOs which take *function* and *node* type DMOs as inputs and output a solution. For example:
solution $\Longleftarrow \quad DMO_{build-model}(DMO_{preprocessing-1}, DMO_{algorithm-1})$
solution $\Longleftarrow \quad DMO_{build-model}(DMO_{preprocessing-2}, DMO_{algorithm-2})$
solution $\Longleftarrow \quad DMO_{build-model}(DMO_{preprocessing-3}, DMO_{algorithm-1})$
...

where $DMO_{build-model}$, and $DMO_{preprocessing-1,2,3}$ are all *function* type DMOs. In this way, we are free to define which, and what kind of, DMOs are to be added to the DMO space.

To meet the data mining goal, we could simply search all the DMO function-object relations (paths) in the space. Therefore, the solution which has the lowest classification or regression error could be the output of a grid-search-like exhaustive search. One advantage of an exhaustive search in a finite DMO space is that the user controls the search complexity. Another advantage is that the DMO relations can be designed by a data mining expert and then shared and reused. For example, if an expert designed a good DMO search space for an unbalanced binary classification problem, she can probably share it with her colleagues or reuse it for a new project.

However, the disadvantage is also obvious because the search complexity grows dramatically as the number of DMOs increases, with the result that if the search space is too large, due to computational and time costs the user may have to terminate the search before all DMOs are explored. To overcome this problem, we may need to think about questions such as how promising DMOs can be automatically defined/generated for a given dataset.

In the previous examples, we have defined some DMOs by hand. One could generate DMOs simply by generating all possible DMO combinations of different types, but doing so would create an extremely large (even infinite) search space, and the problem becomes intractable. We here propose a semi-automatic method to solve this problem.

Firstly, we define some DMO functions, and add these functions to the DMO search space as we did on previous page. Secondly, we define some templates (rules) for searching. Here are two examples:

$$\begin{aligned} &\mathit{solution} \Longleftarrow \\ &\quad DMO_{chain-search}(DMO_{[filter]}, DMO_{[feature-selection]}, \\ &\quad DMO_{[tree-model]}) \qquad (1) \\ &\mathit{solution} \Longleftarrow \\ &\quad DMO_{chain-search}(DMO_{random-topology-search}(DMO_{[filter]}, \\ &\quad DMO_{[feature-selection]}), DMO_{[tree-model]}) \qquad (2) \end{aligned}$$

Template (1) is a chain solution. Here a chain solution means the order (such as from left to right) of each DMO does matter. A "[...]" is a placeholder for a certain type of DMO object: in this example, the [filter] placeholder can be substituted by any filter-type DMO. The [feature-selection] placeholder follows the same rule, and the [tree-model] placeholder can be substituted only by a "tree" type model. In template (2), we can see a new DMO function called "random-topology-search", which means that the order of the DMOs will be changed automatically during the search. So we can see that template (1) is actually a subset of template (2). Once we have a set of DMO objects added, and a DMO template defined, then we have a finite DMO space.

So far, we have defined a DMO space that consists of *node* type DMOs, *function* type DMOs and DMO templates. In the template part of the search space, we will have to make a decision on what kind of search strategy to use when searching for substitution DMOs for placeholders. We here consider only cases where an exhaustive search (in the case of too many DMOs permutations) is not feasible, and we are particularly interested in a search method that optimises a problem by iteratively trying to improve a candidate DMO with regard to a given measure of quality (the goal metric). These methods are usually referred to as a "heuristic search", such as the best-first search, the local search (using neighborhood relation) and the population-based evolutionary algorithms.

1.2 Related Work

The PSMS system proposed in [3], is an application of Particle Swarm Optimization (PSO) to the problem of full model selection for binary classification problems. In total, 3 feature transformation objects, 13 feature selection objects and 10 classifier objects are used in the PSMS system. A full PSMS model is defined as a 16-dimensional particle position. For the details of the PSMS system, we refer the reader to [3]. Based on the experimental results in [3], the PSMS system shows promising results when it is compared with the Pattern Search (PS) strategy [9] for the FMS problem. The system also showed competitive performance compared with other search strategies in a model selection competition.

From the system architecture point of view, PSMS assumes a full model has three components: feature transformation, feature selection, and learning algorithm. In the DMO framework, we can define the following DMO template for the search space covered by the PSMS system:
solution $\Longleftarrow$
$DMO_{chain-search}($
$DMO_{random-topology-search}(DMO_{[feature-transformation]},$
$DMO_{[feature-selection]}),$
$DMO_{[algorithm]})$
We can see that the search space covered by the above DMO template is a simplified presentation of a full model, because a full model may have other components, such as data cleansing and data sampling. Extended from our previous work [10], in the next section, we introduce a novel search strategy for the FMS problem, which covers five data mining components, namely, data cleansing, data sampling, feature transformation, feature selection and algorithm DMOs.

2 The GPS Search Strategy

In this section, we propose a novel algorithm for searching a FMS solution in the DMO space. The algorithm combines both genetic algorithm (GA) [6] and particle swarm optimization (PSO) [7], in which GA is used for searching the optimal template instance of a DMO template, and PSO is used for searching the optimal parameter set for a particular template instance. The motivation is that GA is usually considered a good strategy for combinational optimization problems, whereas PSO is usually considered good at numerical optimization.

The proposed algorithm is named as GPS (**GA**-**P**SO FM**S**). It can be seen as a realization and an application of the DMO framework. Before introducing the GPS algorithm, we first define a DMO template. Here, we assume a FMS solution consists of five DMOs:
$DMO_{[data-cleansing]}$,
$DMO_{[data-sampling]}$,
$DMO_{[feature-transformation]}$,
$DMO_{[feature-selection]}$, and
$DMO_{[algorithm]}$.
Then, a DMO template for the FMS problem covered by GPS is defined as:
solution $\Longleftarrow$
$DMO_{chain-search}($
$DMO_{random-topology-search}($
$DMO_{[data-cleansing]}, DMO_{[data-sampling]},$
$DMO_{[feature-transformation]}, DMO_{[feature-selection]}),$
$DMO_{[algorithm]})$ (3)
Graphically, this template can be represented as Figure 1 (b). The four DMOs at the top can be performed in any order, then followed by an *Algorithm* DMO. Figure 1 (c) shows a solution instance of the DMO template, which can be interpreted as: given a dataset, we firstly apply the data sampling technique,

Algorithm 1 Pseudocode of the GPS strategy for searching a FMS solution

```
procedure GPS($T$,$P$,$M$,$W$,$G$)
    Input:
    $T$ (number of generations for GA), $P$ (population size for GA), $M$ (number of evolutions for PSO), $W$ (swarm size for PSO), $G$ (goal metric)

    Get $P$ random template instances based on template (3).
    Populate template instances with objects in the DMO pools (Table 2)
    for $i \leftarrow 1$ to $T$ do
        Use a standard PSO procedure PSO($M$,$W$,$G$,$I$) to search for the optimal parameters for each template instance $I$ (optimising the goal metric G), and assign an evaluation score to each template instance $I$. This procedure is similar to the PSMS system [3].
        Do crossover // single point crossover among the top 20% template instances.
        Do mutation // randomly choose 30% template instances from the population, and randomly change one DMO in each template instance.
        Replace the worst $N$ template instances with the $N$ new template instances generated in above two steps, here we use $N = (20\% + 30\%) \times P$ .
        $solution_{best} \leftarrow population_{best}$
    end for
    return $solution_{best}$
end procedure
```

SMOTE [2], followed by applying log-transformation, then, we do IQR outlier detection, and then use information gain based feature selection; finally, an AdaBoost.M1 [4] model is built based on the transformed data. We call such a solution a "DMO solution template instance", shortened to "template instance".

For each of the five DMOs we have defined in template (3), we have a pool of data mining tools available. For this research, the filters and algorithms in the WEKA [5] machine learning package are used. Table 2 shows the tools that are included in the GPS system.

Algorithm 1 shows the pseudocode of the GPS algorithm. The basic steps of the system are: firstly a initial population of DMO template instances is randomly generated based on a predefined template (e.g., template (3) and Figure 1 (b)), the placeholders of each template instance are randomly populated with the objects in the pools of DMOs (e.g., Figure 1 (c)). Then for each GA iteration (generation), PSO is used for searching an optimal parameters for each template instance (similar to the PSMS system). The population of template instances is then sorted by their PSO-based evaluation scores. After the PSO optimization procedures are done, typical GA operators, such as crossover and mutation, can be applied for generating new template instances which are used for replacing the template instances with relatively low evaluation scores. The above procedure is repeated T times, where T is the number of GA generations. Finally, the template instance with the best evaluation score is returned as the GPS solution.

Table 1. Data sets: basic characteristics

Original data sets			Final binary data sets
Data set with release year	#Insts	Atts:Classes	Class distribution (#Insts)
Adult 96	48,842	14:2	23% vs 77% (10,000)
Chess 94	28,056	6:18	48% vs 52% (8,747)
Connect-4 95	67,557	42:3	26% vs 74% (10,000)
Covtype 98	581,012	54:7	43% vs 57% (10,000)
KDD09 Customer Churn 09	50,000	190:2	8% vs 92% (10,000)
Localization Person Activity 10	164,860	8:11	37% vs 63% (10,000)
MAGIC Gamma Telescope 07	19,020	11:2	35% vs 65% (10,000)
MiniBooNE Particle 10	130,065	50:2	28% vs 72% (10,000)
Poker Hand 07	1,025,010	11:10	45% vs 55% (10,000)
UCSD FICO Contest 10	130,475	334:2	9% vs 91% (10,000)

3 Comparing GPS to PSMS and Other Learning Systems

We experiment with ten classification problems. All of them are real-world datasets which can be downloaded from the UCI repository, the UCSD data mining contest repository and the KDD Cup repository. These data sets were selected because they are large and come from different research and industrial areas. To speed up the experiments, all five multi-class datasets were converted to binary problems by retaining only the two largest classes from each. After this conversion to binary problems, for datasets that are larger than 10,000 instances, a subset of 10,000 instances is randomly selected for experiments. Table 1 shows the basic properties of the original and the final datasets.

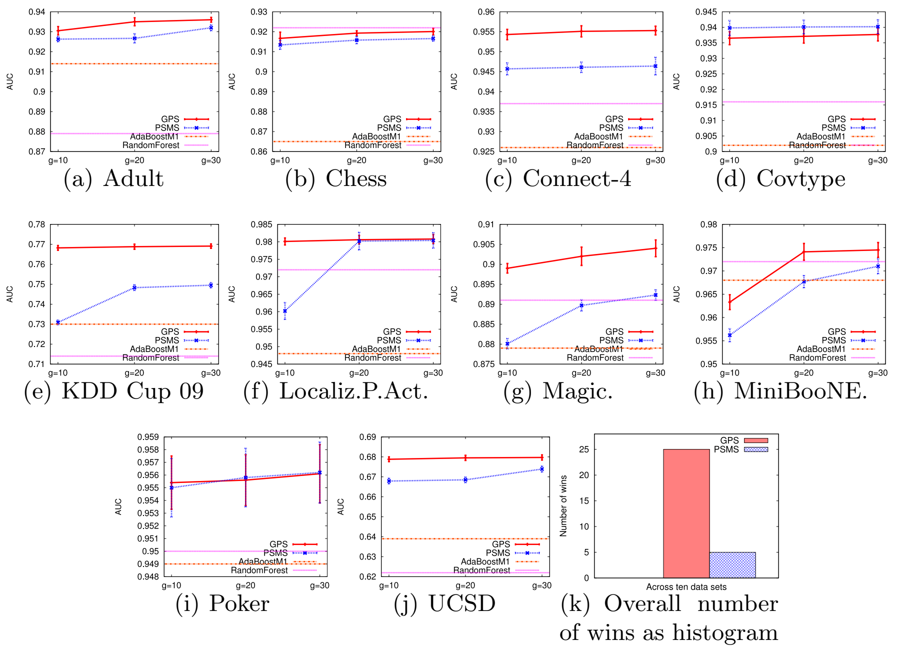
To test the performance of the GPS algorithm, we implemented a variant[1] of the PSMS system proposed in [3] with the DMO pools defined in Table 2. The two systems are set to optimise the AUC performance[2] and are tested under 30 configurations (3 experiments per dataset): for GPS, the population size for GA and the swarm size for PSO are both set to 10, and the number of PSO evolutions is set to 10; for PSMS, the swarm size is set to 10.

For each dataset, three experiments were conducted. Let g be the number of GA generations for GPS; when g=10, the number of PSO evolutions for PSMS is set to 1000; when g=20, the number of PSO evolutions for PSMS is set to 2000; when g=30, the number of PSO evolutions for PSMS is set to 3000. So, for each experiment, the training cost for both systems is roughly the same. The objective functions of both GPS and PSMS are based on the respective training set AUC performance obtained from 3-fold cross validation of a particular template instance. The AUC performance of two popular ensemble learning algorithms, AdaBoost.M1 [4] with 1,000 decision stumps, and Random Forest [1] with 1,000 unpruned random trees are also reported as baseline performance.

Figure 2 (a) to Figure 2 (j) show the comparison results based on the AUC performance obtained from 5 times 3-fold cross validation. Figure 2 (k) gives a summary in terms of number of wins. Overall, on the 10 datasets, the GPS algorithm wins 83% (25 wins) of the 30 experiments. The results demonstrate the benefit of combining GA and PSO for the FMS problem. Also, we can see that the best performance of both GPS and PSMS outperform AdaBoost.M1 and Random Forest on 9 out of the 10 datasets, which indicates the advantage

[1] In our implementation, the dimensionality of each particle is adapted automatically based on the number of parameters of a particular DMO

[2] The balanced error rate (BER) was used in the original PSMS system



(a) Adult (b) Chess (c) Connect-4 (d) Covtype

(e) KDD Cup 09 (f) Localiz.P.Act. (g) Magic. (h) MiniBooNE.

(i) Poker (j) UCSD (k) Overall number of wins as histogram

Fig. 2. A comparison of AUC performance between GPS and PSMS under 30 different configurations; the number of PSO evolutions for GPS is set to 10; x-axis g is the number of GA generations for GPS; when g=10, the number of PSO evolutions for PSMS is set to 1000; when g=20, the number of PSO evolutions for PSMS is set to 2000; when g=30, the number of PSO evolutions for PSMS is set to 3000

of a full model over the single algorithm model. Another interesting pattern is that the GPS algorithm outperforms the baseline algorithms with big margin on datasets with a relatively imbalanced class distribution.

4 Speeding Up the GPS System

The training complexity of the GPS algorithm depends on the base learners found and evaluated during the search. The main cost for GPS is the cost for estimating a base learner's performance (e.g., cross validation). The algorithm searches for a full model consisting of many data mining operators. Therefore, although GPS is powerful in modeling, the user may have to wait for several hours, or even days on relatively large data. For example, on the reduced version of the KDD Cup 2009 data (with 50,000 data points and 190 numeric attributes), the GPS system took about six hours to complete on an AMD 2.8G PC with 16G RAM (number of GA generations, number of PSO evolutions, GA's population size and PSO's swarm size were all set to 10, and 3-fold cross validation was used in the objective function). Therefore, in this section, we present a strategy for speeding up the GPS algorithm. Before introducing the new algorithm, we first review the model tree idea.

Table 2. WEKA algorithms and filters that are used as the DMO objects

Data Sampling	Data Cleansing	Feature Trans.	Feature Sel.
SMOTE oversampling	NumericCleaner	Normalize	CfsSubsetEval
Resample with replacement	RemoveUseless	Standardize	InfoGainAttributeEval
Resample no replacement	ReplaceMissingValues	Center	GainRatioAttributeEval
Do nothing	Do nothing	AddNoise	OneRAttributeEval
		Discretize	PrincipalComponents
		NominalToBinary	ChiSquaredAtt.Eval
		NumericTransform	Do nothing
		Do nothing	

Algorithm	HyperParameters
Bagging with Random Tree	num.Bagging.Iterations, num.Atts., depth.Tree
Bagging with REPTree	num.Bagging.Iterations, num.Folds., depth.Tree
AdaBoost.M1 with DecisionStump	num.Boosting.Iterations , useResample
LogitBoost with DecisionStump	num.Boosting.Iterations , useResample
Bagging with J48 Decision Tree	num.Bagging.Iterations , prune , conf.
RotationForest with REPTree	num.Iterations, Percentage.removed, projection

A model tree [11] is a decision tree system that uses linear models at the leaves instead of using discrete class labels for classification tree or mean as the prediction for regression tree. Model trees inherit the advantageous scalable feature of decision tree systems since the training data is stored in a tree structure. Some variants that have been designed based on the model tree idea show promising results, such as the logistic model tree [8].

We here propose a novel GPS-based model tree algorithm named the Full Model Tree, because GPS builds a full model on a given dataset. The idea is that instead of training the GPS algorithm on the full training data, we build GPS models at the leaves of a tree structure. In the second set of experiments in this paper, we compare GPS to Full Model Tree with two different tree structures, namely, the perfect binary tree and the random binary tree based on the following definitions.

Definition 1: A perfect binary tree is a binary tree with all leaf nodes at the same depth. All internal nodes have degree 2.

Definition 2: A random binary tree is a binary tree formed by inserting nodes one at a time according to a random mechanism.

Next, we show that theoretically when the above two binary tree structures are used, and if the tree height is greater than zero and the training complexity of GPS is worse than linear, then GPS-based Full Model Tree is faster than GPS when training on the same data.

Assume the running time of the normal GPS algorithm (GPS-0) for training its model on a dataset of n data points is $O(f(n))$, and that for the GPS-based Full Model Tree is $O(g(n))$. Based on our preliminary experiments, we found that the *empirical* training complexity of GPS is worse than linear on most of the datasets we have tested, so here we consider the case for $f(n) > n^1, n > 1$.

Theorem 1. *For a perfect-binary-tree-based GPS Full Model Tree T with height $h \geq 1$. If GPS-0's empirical training complexity is worse than linear, such as $f(n) > n^1, n > 1$, then we have $g(n) < f(n)$.*

Proof. Let l be the number of leaf nodes of T, we have $l = 2^h$, $(l \geq 2)$. Let k be the number of data points at each leaf of T, we have $k = n/l$. Then, we have

$g(n) = l \times f(k)$ and $f(n) = f(k \times l)$. Let $f(n) = n^x$, so we have $x > 1$.

$$\begin{aligned} f(n) - g(n) &= f(k \times l) - l \times f(k) = (k \times l)^x - l \times k^x \\ &= k^x \times l^x - k^x \times l = k^x \times (l^x - l) = k^x \times l^{x-1} > 0. \end{aligned}$$

Theorem 2. *For a random-binary-tree-based GPS Full Model Tree T with height $h \geq 1$. If GPS-0's empirical training complexity is worse than linear, such as $f(n) > n^1, n > 1$, then we have $g(n) < f(n)$.*

Proof. Let l be the number of leaf nodes of T, $h \geq 1$ so we have $l \geq 2$. Let k_i be the number of data points at leaf i of T, we have $\sum_{i=1}^{l} k_i = n$. Let $f(n) = n^x$, we have $x > 1$. Then, we have $g(n) = \sum_{i=1}^{l} g(k_i) = \sum_{i=1}^{l} {k_i}^x$ and $f(n) = n^x = (\sum_{i=1}^{l} k_i)^x$. Therefore, $f(n) - g(n) = \{(\sum_{i=1}^{l} k_i)^x - \sum_{i=1}^{l} {k_i}^x\} > 0$.

The two theorems state that theoretically the two Full Model Tree variants are faster than GPS in the case that the training complexity of GPS is worse than linear. Results above are also applicable to memory consumption stating that the two Full Model Tree variants are supposed to be more memory efficient than GPS if the training complexity in terms of memory of GPS is worse than linear. The results also imply that if GPS's training complexity is linear or better, then theoretically the Full Model Tree variants will not speed up the original GPS algorithm. Next, we describe how the GPS-based Full Model Trees are built.

When growing a perfect binary tree, firstly the algorithm checks if the tree height is equal to a user-specified value h. If tree height $= h$ or the current data contains only one class, then make a leaf node and build a GPS model, else the best variable is selected for splitting. Here, the *best* is based on the information gain measure of a variable. For numeric variables, we examine information gain using the median of a variable as the splitting point; for nominal variables, we balance the number of data points from distinct categorical values. For instance, imagine a nominal variable has two distinct categorical values A and B; if the data we are to split has 100 data points, where 80 of them belong to A, and 20 of them belong to B, then we randomly select 30 data points from A, and put them into B. If the nominal variable has three distinct categorical values, say A with 60 data points, B with 30 data points, and C with 10 data points, then we merge B and C first, and then balance A and BC by randomly moving 10 data points from A to BC. The same balancing strategy is also applicable to a nominal variable having more than three distinct categorical values. In this way the amount of data from the current node is roughly equally split for its two child nodes.

When growing a random binary tree, firstly the algorithm checks if the tree height is equal to a user-specified value h. If tree height $= h$ or the current data contains only one class, then make a leaf node and build a GPS model, else the algorithm randomly chooses one of the best K variables for splitting. Here, the *best* is based on the information gain measure of a variable, where K is a user-specified value. For numeric variables, the best splitting point is found by trying all possible splitting points between two neighbored numbers (the splitting point

Table 3. Performance and runtime of the GPS and the Full Model Tree algorithms; A "⊖" indicates that in terms of AUC, the GPS algorithm is significantly better than the respective algorithm; A "◊" indicates that in terms of runtime, the GPS algorithm is significantly slower than the respective algorithm; level of significance 0.05

Dataset	**GPS**	**FMT-perfect**	**FMT-random**	**GPS**	**FMT-perf.**	**FMT-rand.**
	AUC			Runtime (mins)		
Adult	0.94 ± 0.002	0.93 ± 0.003	0.93 ± 0.002 ⊖	45 ± 6	37 ± 4 ◊	48 ± 11
Connect-4.	0.95 ± 0.002	0.95 ± 0.002	0.95 ± 0.003 ⊖	91 ± 5	77 ± 9 ◊	74 ± 14 ◊
KDD Cup.	0.77 ± 0.002	0.77 ± 0.002	0.76 ± 0.003 ⊖	178 ± 9	157 ± 11 ◊	189 ± 8
Mini.B.E.	0.98 ± 0.002	0.98 ± 0.002 ⊖	0.97 ± 0.003 ⊖	124 ± 7	123 ± 9	135 ± 12
UCSD.	0.68 ± 0.003	0.68 ± 0.002 ⊖	0.67 ± 0.002 ⊖	487 ± 16	417 ± 19 ◊	476 ± 17

with the highest gain will be selected); for nominal variables, the data is split between the majority categorical value and the other categorical values.

Next, we examine both the predictive performance and the runtime of the two Full Model Tree variants (one uses the perfect binary tree structure, the other uses the random binary tree structure, namely, FMT-perfect and FMT-random, respectively) to the original GPS algorithm.

We use five medium size datasets for this experiment. Table 1 shows the properties of these datasets. The original *KDDCup09* dataset has 50,000 data points, 190 numeric variables and 40 categorical variables. To speed up the experiment, the 40 categorical variables were removed from the data because some variables have thousands of distinct values. We set the height of both FMT-perfect and FMT-random to 3. So, for FMT-perfect, there will be $2^3 = 8$ leaf GPS models to be built, and each leaf will have $n/8 \pm 1$ data points where n is the total number of training data points. The K value for FMT-random is set to $log(M)+1$, where M is the number of variables. For the GPS algorithm, the number of generations for GA, the population size for GA, the number of evolutions for PSO, and the swarm size for PSO are all set to 10. The objective function of GPS is based on 2-fold cross validation.

Table 3 shows the comparison results based on 5 times 3-fold cross validation. The AUC performance and the runtime are reported. For the AUC performance, we can see that GPS significantly outperforms FMT-random on all datasets, indicating that FMT-random is not good enough to be used as a GPS alternative. The GPS algorithm significantly outperforms FMT-perfect on two datasets; for the other three datasets, the performance of GPS and FMT-perfect has no significant difference. This indicates that for these three datasets, FMT-perfect can be used as a GPS alternative. In terms of runtime, the FMT-random algorithm is significantly faster than GPS only on the *Connect-4.* dataset. One reason could be that the number of data points at the leaf nodes of FMT-random are not the same, so the empirical training complexity of FMT-random varies at each leaf. We can see that FMT-perfect is faster than FMT-random on all datasets because usually the number of leaf nodes of FMT-random is less than that for FMT-perfect. The results show that FMT-perfect is significantly faster then GPS on 4 out of 5 datasets, indicating that FMT-perfect is a viable approach for speeding up GPS. Overall, on 3 out of 5 datasets, namely, *Adult*, *Connect-4.*, and *KDD-Cup09*, the perfect-binary-tree-based Full Model Tree could significantly speed up the GPS algorithm without sacrificing GPS's predictive power.

5 Conclusions

We proposed a framework (in the DMO space setting) which can be used for designing new FMS (full model selection) algorithms, and a novel FMS algorithm which can be seen as a realization and an application of the framework. Our experiments on ten real-world problems show that the GPS algorithm performs very competitively with PSMS, the state-of-the-art PSO-based FMS algorithm. We also examined the feasibility of using the model tree idea for speeding up the GPS algorithm. Our experimental results suggest that using the perfect binary tree as the internal tree structure for GPS-based Full Model Tree is a viable approach when the empirical training complexity of GPS is worse than linear. The techniques described in this paper could probably also be applied to regression and label ranking problems, but this needs to be verified in a future study. Another future work direction is to compare the performance of the GPS systems to fine-tuned base-level ensemble algorithms. The 5-DMO template (3) defined in Section 2 is only one of many possible templates for practical data mining solutions, in future research we will also investigate methods for optimizing alternative templates simultaneously in a cloud environment.

References

1. Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001)
2. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. J. Artif. Int. Res. 16, 321–357 (June 2002)
3. Escalante, H.J., Montes, M., Sucar, L.E.: Particle swarm model selection. Journal of Machine Learning Research 10, 405–440 (2009)
4. Freund, Y., Schapire, R.: Experiments with a new boosting algorithm. In: Thirteenth International Conference on Machine Learning. pp. 148–156 (1996)
5. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.: The weka data mining software: An update. SIGKDD Explorations 11(1) (2009)
6. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press, Cambridge, MA, USA (1992)
7. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Neural Networks, 1995. Proceedings., IEEE International Conference on. vol. 4, pp. 1942–1948. IEEE (1995)
8. Landwehr, N., Hall, M., Frank, E.: Logistic model trees. Mach. Learn. 59, 161–205 (May 2005)
9. Momma, M., Bennett, K.P.: A pattern search method for model selection of support vector regression. In: Proceedings of the SIAM International Conference on Data Mining. SIAM (2002)
10. Sun, Q., Pfahringer, B., Mayo, M.: Full model selection in the space of data mining operators. In: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion. pp. 1503–1504. ACM (2012)
11. Wang, Y., Witten, I.H.: Induction of model trees for predicting continuous classes. In: Poster papers of the 9th European Conference on Machine Learning. Springer (1997)