On Compiling Data Mining Tasks to PDDL *

Susana Fernández and **Fernando Fernández** and **Alexis Sánchez**
Tomás de la Rosa and **Javier Ortiz** and **Daniel Borrajo**
Universidad Carlos III de Madrid. Leganés (Madrid). Spain

David Manzano
Ericsson Research Spain
Madrid, Spain

Abstract

Data mining is a difficult task that relies on an exploratory and analytic process of large quantities of data in order to discover meaningful patterns and rules. It requires complex methodologies, and the increasing heterogeneity and complexity of available data requires some skills to build the data mining processes, or knowledge flows. The goal of this work is to describe data-mining processes in terms of Automated Planning, which will allow us to automatize the data-mining knowledge flow construction. The work is based on the use of standards both in data mining and automated-planning communities. We use PMML (Predictive Model Markup Language) to describe data mining tasks. From the PMML, a problem description in PDDL can be generated, so any current planning system can be used to generate a plan. This plan is, again, translated to a KFML format (Knowledge Flow file for the WEKA tool), so the plan or data-mining workflow can be executed in WEKA. In this manuscript we describe the languages, how the translation from PMML to PDDL, and from a plan to KFML are performed, and the complete architecture of our system.

Introduction

Currently, many companies are extensively using data-mining tools and techniques in order to answer questions as: who would be interested in a new service offer among your current customers? What type of service a given user is expecting to have? These questions, among others, arise nowadays in the telecommunication sector and many others. Data mining (DM) techniques give operators and suppliers an opportunity to grow existing service offers as well as to find new ones. Analysing the customer generated network data, operators can group customers into segments that share the same preferences and exhibit similar behaviour. Using this knowledge the operator can recommend other services to users with a high probability for uptake. The problem is not limited to operators. In every business sector, companies are moving towards the goal of understanding their customers' preferences and their satisfaction with the products and services to increase business opportunities.

Date mining is a difficult task that relies on an exploratory and analytic process of large quantities of data in order to discover meaningful patterns and rules. It requires a data mining expert able to compose solutions that use complex methodologies, including problem definition, data selection and collection, data preparation, or model construction and evaluation. However, the increasing heterogeneity and complexity of available data and data-mining techniques requires some skills on managing data-mining processes. The choice of a particular combination of techniques to apply in a particular scenario depends on the nature of the data-mining task and the nature of the available data.

In this paper, we present our current work that defines an architecture and tool based on Automated Planning that helps users (non necessarily experts on data-mining) on performing data-mining tasks. Only a standardized model representation language will allow for rapid, practical and reliable model handling in data mining. Emerging standards, such as PMML (Predictive Model Markup Language), may help to bridge this gap. The user can represent and describe in PMML data mining and statistical models, as well as some of the operations required for cleaning and transforming data prior to modelling. Roughly speaking, a PMML file is composed of three general parts: the data information, the mining build task and the model. But, during the complete data mining process, not all of them are available. The data part describes the resources and operations that can be used in the data mining process. The mining build task describes the configuration of the training task that will produce the model instance. It can be seen as the description of the sequence of actions executed to obtain the model, so from the perspective of planning, it can be understood as a plan. This plan would include the sequence of data-mining actions that should be executed over the initial dataset to obtain the final model that could be also added to the third part of the PMML. Therefore, a complete PMML file contains all the information describing a data mining task, from the initial dataset description to the final model, through the knowledge flow that generates that model. We use a PMML file containing only the data information, and leaving empty the other two parts, as an input to create PDDL (Planning Domain Definition Language) problem files. These files allow,

*This work has been partially supported by the Spanish MICINN under project TIN2008-06701-C03-03, the regional project CCG08-UC3M/TIC-4141 and the Automated User Knowledge Building (AUKB) project funded by Ericsson Research.

together with the PDDL domain file and a planner, to automatically create a plan or knowledge flow. This knowledge flow will be executed by a machine learning engine. In our case, we use one of the most used data mining tools, the WEKA system (Witten & Frank 2005). In WEKA, knowledge flows are described as KFML files. The results of this process can be evaluated, and new plans may be requested to the planning system. The plans generated by the planner for solving the translated PDDL problem are encoded in XML and can be directly added to the PMML mining building task. However, so far, stable versions of WEKA do not encode models in XML, so we leave empty the model part of the PMML file. Instead, the system returns to the user a result zip file containing all the model files together with statistical information files in WEKA format. In this paper, we describe the first implementation of such a tool, emphasizing the compilations between PMML and PDDL.

There has been previous work whose aim is also to apply planning techniques to aumotatize data mining tasks (Amant & Cohen 1997; Morik & Scholz 2003; Provost, Bernstein, & Hill 2005), but any of them use standard languages to represent the knowledge, as we have done in the work presented in this paper. Next section presents an introduction to data mining. The remainder sections describe the general architecture, the languages used in the application, the modelization of the DM tasks in PDDL, the implemented translators and the conclusions.

Introduction to KDD and Data Mining

Knowledge Discovery in Databases (KDD) concerns with the development of methods and techniques for analyzing data. The first part of the KDD process is related to the selection, preprocess and transformation of the data to be analyzed. The output of this part of the KDD process is a set of patterns or training data, described in terms of features, typically stored in a table. Data mining is the following step in the KDD process, and consists of applying data analysis and discovery algorithms that generate models that describe the data (Fayyad, Piatetsky-Shapiro, & Smyth 1996). The modelling task is sometimes seen as a machine learning or statistical problem where a function must be approximated. The KDD process finishes with the interpretation and evaluation of the models.

Learning algorithms are typically organized by the type of function that we want to approximate. If the output of the function is known "a priori" for some input values, we typically talk about supervised learning. In this case, we can use regression (when the approximated function is continuous), or classification (the function is discrete) (Mitchell 1997). There are many models for supervised learning. Some examples are neural networks, instance-based, decision and regression rules and trees, bayesian approaches, support vector machines, etc. When the output of the function to approximate is not known "a priori", then we talk about unsupervised learning. In this kind of learning, the training set is composed only of the list of input values. Then, the goal is to find a function which satisfies some learning objectives. Different learning objectives can be defined, like clustering, dimensionality reduction, association, etc, and they all may require different models and learning algorithms.

The evaluation of a data mining problem typically requires to apply the obtained model over data that were not used to generate the model. But depending on the amount of data, or the type of function modelled, different evaluation approaches may be used, such as split (dividing the whole data set in training and test sets), cross-validation, or leave-one-out (Mitchell 1997).

Therefore, In the data mining process, there are four main elements to define: the training data, the model or language representation, the learning algorithm, and how to evaluate the model. The number of combinations of those four elements is huge, since different methods and techniques, all of them with different parameter settings, can be applied. We show several examples of these operators in the paper. Because of that, the data mining process is sometimes seen as an expert process where data mining engineers transform original data, execute different mining operators, evaluate the obtained models, and repeat this process until they fit or answer the mining problem. Because of the complexity of the process, it is suitable as a planning problem that can be solved with automated approaches (Fernández *et al.* 2009).

Architecture

Figure 1 shows the general architecture of the approach. There are four main modules; each one can be hosted in a different computer connected through a network: the *Client*, the *Control*, the *Datamining* and the *Planner*. We have used the Java RMI (Remote Method Invocation) technology that enables communication between different servers running JVM's (Java Virtual Machine). The planner incorporated in the architecture is SAYPHI (De la Rosa, García-Olaya, & Borrajo 2007) and the DM Tool is WEKA. Although, any others could be used. Next, there is a description of each module.

The *Client* Module

It offers a control command console interface that provides access to all application functionalities and connects the user to the *Control* module using RMI. It captures the PMML file and sends it to the *Control* module. At this point, the PMML file only contains the information concerning the dataset and operations that can be executed in the data mining process. The other two parts are empty. Before performing any DM task, the corresponding dataset must be placed in the DM Tool host, through this module.

The *Control* Module

It is the central module of the architecture that interconnects and manages the planning and DM modules, serving requests from the user module. This module is also responsible of performing the conversions between the PDDL and PMML formats invoking the *PMML2PDDL* external application. It also transforms the output plan generated by the planner to a KFML format used by the WEKA Knowledge Flow. This KFML format transformation is performed

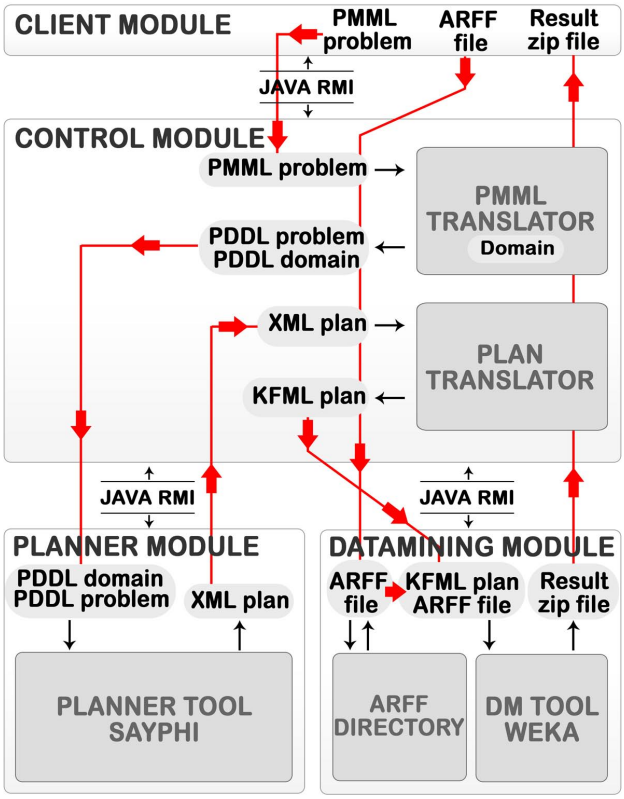

Figure 1: Overview of the architecture.

through the invocation of the external *PlanReader* application.

The input to the module is the user choice together with the required files to carry out that option. In case the user's option is to perform a DM request, the algorithm of Figure 2 is executed. First, the **PMML2PDDL** translator generates the PDDL problem file from the PMML file. Then, the planner is invoked through RMI to solve the translated problem. The returned plan is translated to a KFML file. Finally, the DM Tool is invoked to execute the translated KFML file. The *result* is a compressed file containing the model generated by the DM Tool and the statistics.

The *Datamining* Module

This module provides a Java wrapper that allows the execution of DM tasks in the WEKA DM Tool through Knowledge Flow plans. The Java wrapper is able to obtain the model output and the statistics generated as a result of the Knowledge Flow execution. This module also contains an *Arff* directory for managing the storage of the datasets that are necessary for the *PlanReader* and WEKA executions. The inclusion or removal of *arff* files are managed by the user through the options offered in the command control user interface.

DM-Request(*pmml-file*,*domain*):*result*

---

pmml-file: PMML file with the DM task
domain: PDDL domain

---

problem=**PMML2PDDL**(*pmml-file*)
plan=**RMI-Planner**(*domain*,*problem*)
kfml-file=**Plan2KFML**(*plan*)
result=**RMI-DMTool**(*kfml-file*)
return *result*

Figure 2: Algorithm for executing a DM request.

The input to the module is a KFML file and the output is a compressed file including the model and the statistics generated during the KFML execution.

The *Planner* Module

The planning module manages requests from the control module receiving the problem and the domain in PDDL format. It returns the result to the *Control* module in XML format ready for the conversion to a KFML format. Currently, planning tasks are solved by the planner SAYPHI, but the architecture could use any other planner that supports fluents and metrics. We have used SAYPHI because: i) it supports PDDL (requirements *typing* and *fluents*); ii) it incorporates several search algorithms able to deal with quality metrics; iii) it is implemented in Lisp allowing rapid prototyping of new functionalities; and iv) it is in continuous development and improvement in our research group.

Standard Languages of the Tool

This section describes the two languages used in this work (apart from PDDL). First, we describe PMML (Predictive Model Markup Language), an XML based language for data mining. Then, we describe KFML (Knowledge Flow for Machine Learning), another XML based language to represent knowledge flows in data mining defined in the WEKA tool (Witten & Frank 2005). The PDDL language is not explained since it is a well-known standard in the planning community. We use the PDDL 2.1 version (Fox & Long 2003) for handling classical planning together with numeric state variables.

The Predictive Model Markup Language (PMML)

PMML is a markup language for describing statistical and data mining tasks and models. It is based on XML, and it is composed of five main parts:[1]

- The header contains general information about the file, like the PMML version, date, etc.
- The data dictionary defines the meta-data, or the description of the input data or learning examples.

[1]The language is being developed by the Data Mining Group (DMG). See www.dmg.org for further information

Models	Functions
AssociationModel	
ClusteringModel	
GeneralRegressionModel	
MiningModel	
NaiveBayesModel	AssociationRules
NeuralNetwork	Sequences
RegressionModel	Classification
RuleSetModel	Regression
SequenceModel	Clustering
SupportVectorMachineModel	
TextModel	
TreeModel	

Table 1: Models and Functions defined in the PMML standard

- The transformation dictionary defines the functions applicable over the input data, like flattening, aggregation, computation of average or standard deviation, normalization, principal component analysis (PCA), etc. In our case, this knowledge defines the actions that can be applied over the data, which will be defined in the planning domain file.
- The mining build task describes the configuration of the training task that will produce the model instance. PMML does not define the content structure of this part of the file, so it could contain any XML value. This mining build task can be seen as the description of the sequence of actions executed to obtain the model, so from the perspective of planning, it can be understood as a plan. This plan would include the sequence of data-mining actions that should be executed over the initial dataset to obtain the final model.
- The model describes the final model generated after the data mining process, i.e. after executing the mining build task. There are different models that can be generated depending on the data analysis technique used, ranging from bayesian, to neural networks or decision trees. Depending on the type of model the description will use a different XML format.

The models implement some functions, and depending on the function, they may introduce different constraints over the data (which will have to be considered in the planning domain file). For instance, a TreeModel can be used both for classification or regression, but depending on the implementation itself, only one of such functions may be available. If, for instance, only regression has been implemented, then the class attribute of the dataset must be continuous if we want to apply the TreeModel. The complete list of models and functions defined in the PMML standard are defined in Table 1. Different models can implement different functions and, as introduced above, the applicability of the model to a function may depend on the implementation of the model itself and the advances of the state of the art, so they are not constrained "a priori".

A complete PMML file contains the information about the five parts. However, during the data mining process, not all the data is available. At the beginning, the PMML file contains only the header, the data dictionary and the transformation dictionary. In addition, it includes the different models that may be used to find a solution. This can be considered as the input of the data mining process. The mining build task contains the steps or process that should be followed to obtain a model using the data described in the PMML initial file. In our case, we generate this version of the PMML file from the result of planning by encoding the plans in XML. Finally, the model could be included after the data mining tool executes the data-mining workflow generated by the planner. In our case, we generate the model using the WEKA Knowledge Flow tool that receives as input a KFML file. The tool automatically translates the plan into the KFML file, but the model is not incorporated in the PMML file, because WEKA does not encode it yet in XML.

WEKA and the Knowledge Flow Files (KFML)

WEKA (Witten & Frank 2005) is a collection of machine learning algorithms to perform data mining tasks. It includes all the software components required in a data mining process, from data loading and filtering to advanced machine learning algorithms for classification, regression, etc. It also includes many interesting functionalities, like graphical visualization of the results. WEKA offers two different usages. The first one is using directly the WEKA API in Java. The second one consists of using the graphical tools offered: the Explorer permits to apply data mining algorithms from a visual interface; the Experimenter permits to apply different machine learning algorithms over different datasets in a batch mode; the Simple CLI, which permits to make calls to WEKA components through the command line; and the Knowledge Flow.

WEKA Knowledge Flow is a data-flow inspired interface to WEKA components. It permits to build a knowledge flow for processing and analysing data. Such knowledge flow can include most of the WEKA functionalities: load data, prepare data for cross-validation evaluation, apply filters, apply learning algorithms, show the results graphically or in a text window, etc. Knowledge flows are stored in KFML files, that can be given as input to WEKA.

A KFML file is an XML file including two sections. The first one defines all the components involved in the knowledge flow, as data file loaders, filters, learning algorithms, or evaluators. The second one enumerates the links among the components, i.e. it defines how the data flows in the data mining process, or how to connect the output of a component with the input of other components. WEKA Knowledge Flow permits to load KFML files, edit them graphically, and save them. KFML files can be executed both by using the graphical interface or the WEKA API.[2]

A knowledge flow can be seen as the sequence of steps that must be performed to execute a data mining process. From our point of view, the knowledge flow is not more

[2]The WEKA API can be used to execute KFML files only from version 3.6.

than the plan that suggest to perform a data mining process. Later, we will show how a plan generated in our architecture can be saved in the KFML format and executed through the WEKA API.

Modelling Data-Mining Tasks in PDDL

The PDDL domain file contains the description of all the possible DM tasks (transformations, training, test, visualization, ...). Each DM task is represented as a domain action. The PDDL problem files contain information for a specific dataset (i.e. dataset schema, the suitable transformation for the dataset, the planning goals, the user-defined metric, etc.). Domain predicates allow us to define states containing static information (i.e. possible transformations, available training or evaluation tasks, etc.) and facts that change during the execution of all DM tasks (e.g. `(preprocess-on ?d - DataSet)` when the dataset is pre-processed). Fluents in the domain allow us to define thresholds for different kinds of characteristics (e.g. an execution time threshold, or the readability of a model) and to store numeric values obtained during the execution (e.g. the total execution time, the mean-error obtained, etc.).

There are different kinds of actions in the domain file:

- Process Actions: for specific manipulations of the dataset. For instance, `load-dataset` or `datasetPreparation` for splitting the dataset or preparing it for cross-validation after finishing the data transformation. Preconditions verify the dataset and test mode availability with the predicates `available` and `can-learn`, respectively. Figure 3 shows the PDDL `datasetPreparation` action. It adds the effect `(eval-on)` for allowing the training and testing. We use fluents for estimating the execution time of actions, whose values are defined in the problem. We assume the execution time is proportional to the number of instances (in fact thousands of instances) and depends on the test mode. For example, we estimate that the preparation factor for splitting is 0.001 and for cross-validation is 0.005. The `(loaded ?d)` precondition is an effect of the `load-dataset` action.

```
(:action datasetPreparation
   :parameters (?d - DataSet ?t - TestMode)
   :precondition (and (loaded ?d)
                      (can-learn ?d ?t))
   :effect (and (eval-on ?d ?t)
                (not (preprocess-on ?d))
                (not (loaded ?d))
                (increase (exec-time) (* (thousandsofInstances)
                                         (preparationFactor ?t)))))
```

Figure 3: Example of PDDL process action.

- Transformation Actions: in the form of `apply-transformation-`<*filter*>. Preconditions verify field type constraints and whether the filter has been included as a DM task for the dataset or not. The action usually adds a fact indicating that the task has been done (e.g. the normalize transformation adds to the state the fact `(normalized DataSet)`). Figure 4 shows the attribute selection PDDL action. We assume this is the last transformation we can perform because afterwards we stop knowing the remaining attributes and the metric estimation would return an unknown value. Precondition `(known-fields)` checks it. Precondition `(transformation ?i AttributeSelection)` verifies the filter has been included in the PMML file and `(preprocess-on ?d)` prevents from applying the transformation before performing the pre-process and after preparing the data for splitting or cross-validation. Again, we assume an execution-time increment that is computed from several fluents whose values are defined in the problem.

```
(:action apply-transformation-AttributeSelection
   :parameters (?d - DataSet ?i - TransfInstance)
   :precondition (and (preprocess-on ?d)
                      (known-fields ?d)
                      (transformation ?i AttributeSelection))
   :effect (and (applied-instance ?d ?i)
                (applied-transformation ?d AttributeSelection)
                (not (known-fields ?d))
                (increase (exec-time)
                     (* (* (filter-time AttributeSelection)
                           thousandsofInstances))
                        (* (dataDictionaryNumberOfFields)
                           (dataDictionaryNumberOfFields))))))
```

Figure 4: PDDL action representing the attribute selection transformation.

- Training Actions: in the form of `train-`<*model-type*>, where <*model-type*> can be `classification`, `regression` or `clustering`. In each type, the action parameters indicate different models previously defined in the PMML file. There is a precondition for verifying if the model instance is learnable, predicate `learnable`, and another one to verify the model, predicate `is-model`. These actions add to the state that the selected option is a model for the dataset with the predicate `is-`<*model-type*>`-model`. Figure 5 shows the PDDL action for training in a classification task. Training tasks always increment errors, in case of classification they are the classification error, *percentage-incorrect*, and the readability of the learned model, *unreadability*. And, we assume values for these increments.

```
(:action train-classification
   :parameters (?mi - ModelInstance ?m - Model ?n -
     ModelName ?d - DataSet ?fi - FieldName  ?t - TestMode)
   :precondition (and (learnable ?mi)
          (is-model ?mi ?m)
          (implements ?m classification ?n)
          (is-field ?fi ?d)
          (dataDictionaryDataField-otype ?fi categorical)
          (eval-on ?d ?t))
   :effect (and (is-classification-model ?mi ?d ?fi)
         (not (preprocess-on ?d))
         (not (learnable ?mi))
         (increase (unreadability)
                   (model-unreadability ?m))
         (increase (percentage-incorrect)
                   (model-percentage-incorrect ?m))
         (increase (exec-time)
             (* (* (model-exec-time ?m)
                   (thousandsofInstances))
                (* (dataDictionaryNumberOfFields)
                   (dataDictionaryNumberOfFields))))))
```

Figure 5: PDDL action for training.

- Testing Actions: in the form of `test-`<*model-type*>.

These actions usually follow their corresponding training action. For instance, cross-validation or split. They are separated from the training actions, because they are needed when handling the process flow in the KFML. Testing actions add the fact that the learned model has been evaluated.

- Visualizing and Saving Actions: in the form of `visualize-<`*result-option*`>`. These actions are related to goals defined in the problem file. Action parameters indicate the preferred option depending on the learned model, the pre-defined options in the PMML file and the goal defined for the specific planning task. There is a precondition to verify if the visualization model is allowed for that model, predicate `allow-visualization-model`.

The domain actions represent specific DM tasks. A matching between domain actions and the DM tasks defined in the KFML file is required. To have an idea of the complexity of this planning domain, we have to take into account that WEKA implements more than 50 different transformation actions or filters, more than 40 training actions for classification and regression, and 11 training actions for classification. Each of these transformation and training actions can be parameterized, so the number of different instantiations of those actions is huge. Obviously, not all of them must be included in the PDDL files to generate a plan, and the user can define in the PMML which of them s/he is interested in using.

There are different ways to define the domain actions. For instance, in our case the actions `train-classification` and `train-regression` are coded separately because they correspond to different DM tasks in WEKA. However, they could have been merged in only one action, obtaining a more compact domain representation with an easier maintenance. With the current implementation we gain the opportunity of faster integration with other DM tools different than WEKA. The performance of the planner is not affected, since it performs the search using grounded actions, so both domain definitions will become equivalent in search time.

Information in the problem file is automatically obtained from the PMML file using the translator described in the next section. This information, specific to a dataset, is used by the planner to instantiate all the possible actions specified in the PMML file by the user and to handle constraints imposed to the planning task.

The whole domain contains the following actions. `load-dataset` is always the first action in all plans. Then, the plans can contain any number of filter actions such as `apply-transformation-DerivedField-2op`, `apply-transformation-normalize`, `apply-transformation-discretization` or `apply-transformation-AttributeSelection`. There could be no filter action, as well. After the attribute selection filter is applied no other filter is allowed. The following action is the `datasetPreparation` action that prepares the dataset for splitting or for a cross-validation. Then, there are three possible actions for training, `train-classification`, `train-regression` and `train-clustering`, another three for testing, `test-classification`, `test-regression` and `test-clustering`, and three more for visualizing the model `visualize-classification-model`, `visualize-regression-model` and `visualize-clustering-model`. The last action in all plans is the `visualize-result` for knowing the learning results (errors, execution time, ...).

Translators

PMML to PDDL

The PMML2PDDL translator automatically converts parts of a PMML file with the DM task information into a PDDL problem file. The problem file together with a domain file, that is assumed to stay fixed for all the DM tasks, are the inputs to the planner. The problem file contains the particular data for each DM episode, including the dataset description, the transformations and models available for that problem, and the possible preferences and constraints the user required. An example of preference is minimizing the total execution time. Obtaining an error less than a given threshold is an example of constraint.

A PDDL problem file is composed of four parts: the *objects*, the *inits*, the *goals* and the *metric*. *Inits* represents the initial state of the problem and there is one proposition for each fact in the state. There are some propositions common to all problems (static part of the problem specification) and others are translated from the PMML file (dynamic part). *Goals* represent the problem goals. So far, they are fixed for all the problems, and consist of visualizing the result, `(visualized-result result text)`, for saving the statistics required to analyze the generated DM models. *Metric* represents the formula on which a plan will be evaluated for a particular problem. A typical DM metric consists of minimizing the classification error, but others could be used, as minimizing execution time or maximizing the readability of the learned model. SAYPHI, as the planners based on the enhanced relaxed-plan heuristic introduced in Metric-FF (Hoffmann 2003), only work with minimization tasks. So, we transform maximizing the understandability of the learned model for minimizing *complexity*. The other preferences considered in our model are: *exec-time*, for minimizing the execution time; *percentage-incorrect*, for minimizing the classification error; and *mean-absolute-error*, for minimizing the mean absolute error in regression tasks and clustering. We can handle similar constraints.

The static part of the problem specification includes all the propositions concerning the predicates `is-model`, `learnable`, `allow-visualization`, `available` and `can-learn`, explained in the previous section, and the initial values of the fluents. The dynamic part includes the information about the dataset, the constraints and preferences, the transformations and the models available for the DM request. Figure 6 shows and example of PMML file representing the well-known *Iris* DM set, allowing only one transformation (a discretization) and two models (a neural network and a decision tree). In general, for most DM tasks, a user would include in the PMML file all WEKA DM techniques (there are plenty of them). In the example of PMML

file, the user also defines: two constraints, concerning the execution time and the complexity; and one preference, to minimize the percentage-incorrect value, or classification error. We obtain information from the following parts of the PMML: *Header, DataDictionary, TransformationDictionary* and *Models*. *Header* includes the constraints and the preferences. Constraints are translated by adding numerical goals to the PDDL problem. Preferences are translated by changing the metric of the problem. *DataDictionary* includes one field for each attribute in the dataset and they are translated into propositions in the initial state and objects of the problem. *TransformationDictionary* is translated by adding some objects and propositions in the initial state of the problem. Finally, each model defined in the *Models* part is translated by adding a proposition in the initial state.

Figure 6 shows an example of the translation from a PMML file to a PDDL problem file. It is easy to see how some information is translated. For instance, the tag *DataField* in the PMML file generates some predicates in the PDDL file, like `dataDictionaryDataField-type`, `dataDictionaryDataField-otype`, one for each attribute of the initial dataset. The transformation *Discretize* of the PMML file is translated by adding the predicate `(transformationInstance-type discretize continuous)`. When a model appears in the PMML, as *<NeuralNetwork modelName="nnmodel1" functionName="classification" algorithmName="Neural Net" activationFunction="radialBasis">*, it enables the planner to use neural networks. The parameters of that model are also defined in the PMML file, so the predicate `(implements NeuralNetwork classification nnmodel1)` and other related predicates are added to the problem file. Similarly, the preference described in the header of the PMML file, *<Constraint variable="percentage-incorrect" value="minimize"/>*, is translated by including a metric `(:metric minimize (percentage-incorrect))` in the problem file; while the constraints *<Constraint variable="exec-time" value="30"/ >* and *<Constraint variable="complexity" value="8"/ >* are translated by including the numerical goals `(< (exec-time) 30)` and `(< (complexity) 8)`.

The translator makes the following conversions from the PMML file to the PDDL problem file (*<variable>* represents the value of field *variable* in the PMML file):

1. Translates entry *<DataDictionary>* into the following propositions in the *inits*:
   (a) `(= (DataDictionaryNumberOfFields) <numberOfFields>)`
   (b) `(dataDictionaryDataField-type <name> <dataType>)`, for each *<DataField>* entry
   (c) `(dataDictionaryDataField-otype <name> <optype>)`, for each *<DataField>* entry
   (d) `(is-field <name> DataSet)`, for each *<DataField>* entry
   (e) Creates an object `<name>` of type $SchemaFieldName$ in *objects*
2. Adds the following proposition in the *inits* for each *<DefinedFunction>* entry:
   (a) `(transformationInstance-type <name> <opType>)`
   (b) `(transformation <name> valueX)`, where `valueX` is in *<DefinedFunction.Extension.WekaFilterOptions.option.value>* when *variable="filter" value*=valueX>
   (c) Create an object `<name>` of type $transfInstance$ in *objects*
3. Adds the following proposition in the *inits* for each *<DerivedField>* entry:
   (a) `(dataDictionaryDataField-type <name> <dataType>)`
   (b) `(dataDictionaryDataField-type <name> <optype>)`
   (c) `(derivedField-operation <name> <Apply.function>)`
   (d) `(derivedField-source <name> <Apply.FieldRef.field>)` for each *<FieldRef>* in each *<DerivedField>*
   (e) Create an object `<name>` of type $DerivedFieldName$ in the *objects*
4. Adds the following proposition in the *inits* for each *<modelName>* entry:
   `(implements <model> <functionName> <modelName>)`
5. Adds a numeric goal `(< (<Constraint variable>) <value>)` for each *<DataMiningConstraints>* entry
6. Translates entry *<DataMiningPreferences>* for `(:metric minimize (<Constraint variable>))` in *metric*.

In order to make the translation process easier and more general, we have defined an intermediate XML file, *translation.xml*, to drive the translation explained above. The translator interprets this file each time together with the input PMML file and generates a PDDL problem file. So, it is possible to include, modify or eliminate PMML entries from the translation process without changing the program, but modifying this file. This intermediate XML file permits to configure the information we need from the PMML file, so that in case the PMML file or the domain change we only have to change this file without modifying the translator. For example, the translation 1.(b) explained above is represented with the following entry in the *translation.xml* file:

```
<elements value="dataDictionaryDataField-type"
          where="DataDictionary/DataField">
    <field type="attribute">name</field>
    <field type="attribute">dataType</field>
    <format>(/0/ /1/ /2/)</format>
</elements>
```

Planning for DM Tasks

As we mentioned in the previous section, SAYPHI solves the planning task depending on the metric specified in the PMML file. SAYPHI has a collection of heuristic algorithms. For this application, the planner performs a Best-first Search that continues exploring nodes in order to find multiple solutions. Figure 7 shows an example of a best-cost solution plan when the translated PDDL indicates the percentage error as the problem metric. Likewise, Figure 8 shows a best-cost solution plan for the same problem, but using the execution time metric. In the first plan a *Neural Network* is preferred

```
<?xml version="1.0" encoding="UTF-8"?>
<PMML xmlns="http://www.dmg.org/PMML-3_2"
      version="3.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.dmg.org/PMML-3_2 pmml-3-2.xsd">
    <Header copyright="Copyright UC3M Ericsson LME 2009">
      <Extension>
          <DataSources DataSourceFormat="file">
            <DataFile location="iris.arff"/>
          </DataSources>
          <DataMiningConstraints constraints="yes">
            <Constraint variable="exec-time" value="30"/>
            <Constraint variable="complexity" value="8"/>
          </DataMiningConstraints>
          <DataMiningPreferences preferences="yes">
            <Constraint variable="percentage-incorrect"
                        value="minimize"/>
          </DataMiningPreferences>
      </Extension>
    </Header>

  <DataDictionary numberOfFields="5">
    <DataField name="sepallength" displayName="sepallength"
               optype="continuous" dataType="double"/>
    <DataField name="sepallwidth" displayName="sepallwidth"
               optype="continuous" dataType="double"/>
    <DataField name="petallength" displayName="petallength"
               optype="continuous" dataType="double"/>
    <DataField name="petalwidth" displayName="petalwidth"
               optype="continuous" dataType="double"/>
    <DataField name="class" displayName="class"
               optype="categorical" dataType="string">
        <Extension name="class" value="yes"/>
        <Value value="Iris-setosa" property="valid"/>
        <Value value="Iris-versicolor" property="valid"/>
        <Value value="Iris-virginica" property="valid"/>
    </DataField>
   </DataDictionary>

    <TransformationDictionary>

       <!--  WEKA Filters: -->
       <DefineFunction name="discretize" optype="continuous">
         <Extension>
           <WekaFilterOptions options="yes">
              <option variable="supervised" value="yes"/>
               <option variable="type" value="attribute"/>
           </WekaFilterOptions>
         </Extension>
         <ParameterField name="x" optype="continuous"/>
       </DefineFunction>

    </TransformationDictionary>

<!-- Models -->

<NeuralNetwork modelName="nnmodel1" functionName="classification"
algorithmName="Neural Net" activationFunction="radialBasis">
          <Extension>
            <WekaModelOptions options="yes">
              <option variable="model" value="functions"/>
            <option variable="algorithm" value="RBFNetwork"/>
            <option variable="num-clusters" value="2"/>
            </WekaModelOptions>
        </Extension>
</NeuralNetwork>

<TreeModel modelName="treemodel1" functionName="classification"
         algorithmName="C4.5">
         <Extension>
            <WekaModelOptions options="yes">
              <option variable="model" value="tree"/>
            <option variable="algorithm" value="J48"/>
            <option variable="unpruned" value="true" />
           </WekaModelOptions>
       </Extension>
</TreeModel>

</PMML>
```

(a) Example of PMML file encoding a DM request for the *Iris* domain. The PMML file has been simplified to eliminate non-relevant information for the purpose of generating the problem file.

```
(define (problem p1)
  (:domain mole)
(:objects  sepallength sepallwidth petallength
                    petalwidth class - SchemaFieldName
   discretize AttributeSelection - transfInstance)
  (:init

   ;; Common to all problems
   (is-model tree treeModel)
   (learnable tree)
   (is-model nn NeuralNetwork)
   (learnable nn)

   (allow-visualization-model treeModel graph)
   (allow-visualization-model treeModel text)
   (allow-visualization-model NeuralNetwork graph)
   (allow-visualization-model NeuralNetwork text)

   (allow-visualization-result result text)
   (allow-visualization-result result graph)
   (allow-visualization-result result2 text)
   (allow-visualization-result result2 graph)

   (= (exec-time) 0)
   (= (function-time-discretize) 0.05)

   (= (model-exec-time NeuralNetwork) 0.01)
   (= (model-exec-time treeModel) 0.001)

   (= (complexity) 0)
   (= (model-complexity NeuralNetwork) 5)
   (= (model-complexity treeModel) 9)

   (= (percentage-incorrect) 0)
   (= (model-percentage-incorrect NeuralNetwork) 5)
   (= (model-percentage-incorrect treeModel) 9)

   (= (mean-absolute-error) 0)
   (= (model-mean-absolute-error NeuralNetwork) 5)
   (= (model-mean-absolute-error treeModel) 9)

    (= (exec-time-test-regression) 5)
    (= (exec-time-test-clustering) 5)

   ;; depend on the PMML
   ;;PMML Inserted by the translator
   (dataDictionaryDataField-type sepallength double)
   (dataDictionaryDataField-type sepallwidth double)
   (dataDictionaryDataField-type petallength double)
   (dataDictionaryDataField-type petalwidth double)
   (dataDictionaryDataField-type class string)
   (dataDictionaryDataField-otype sepallength continuous)
   (dataDictionaryDataField-otype sepallwidth continuous)
   (dataDictionaryDataField-otype petallength continuous)
   (dataDictionaryDataField-otype petalwidth continuous)
   (dataDictionaryDataField-otype class categorical)
   (is-field sepallength initialDataSet)
   (is-field sepallwidth initialDataSet)
   (is-field petallength initialDataSet)
   (is-field petalwidth initialDataSet)
   (is-field class initialDataSet)
   (transformationInstance-type discretize continuous)
   (implements NeuralNetwork classification nnmodel1)
   (implements TreeModel classification treemodel1)
   (= (DataDictionaryNumberOfFields) 5)
   ;;PMML Inserted by the translator
   )
  (:goal (and (visualized-result result text)
              (< (complexity) 8)
              (< (exec-time) 30)
      ))
    (:metric minimize (percentage-incorrect))
)
```

(b) Problem file in PDDL generated from the PMML file shown in (a). Again, irrelevant information has been eliminated.

Figure 6: Simplified PMML and PDDL problem files.

```
0: (LOAD-DATASET INITIALDATASET)
1: (DATASETPREPARATION INITIALDATASET CROSS-VALIDATION)
2: (TRAIN-CLASSIFICATION NN NEURALNETWORK
          NNMODEL4 INITIALDATASET CLASS CROSS-VALIDATION)
3: (TEST-CLASSIFICATION NN INITIALDATASET NEURALNETWORK
          CLASS CROSS-VALIDATION RESU)
4: (VISUALIZE-RESULT NN INITIALDATASET CROSS-VALIDATION
          RESU TEXT)
```

Figure 7: An example plan minimizing the percentage error.

```
0: (LOAD-DATASET INITIALDATASET)
1: (DATASETPREPARATION INITIALDATASET SPLIT)
2: (TRAIN-CLASSIFICATION TREE TREEMODEL
        TREEMODEL1 INITIALDATASET CLASS SPLIT)
3: (TEST-CLASSIFICATION TREE INITIALDATASET
        TREEMODEL CLASS SPLIT RESU)
4: (VISUALIZE-RESULT TREE INITIALDATASET
        SPLIT RESU TEXT)
```

Figure 8: An example plan minimizing the execution time.

in the `train-classification` action because in the default definition it has a lower cost than applying the same action with another model. On the other hand, the second plan has two differences. The *Split* parameter is selected instead of *Cross-validation*, and *Tree Model* is preferred instead of *Neural Network*. Both differences arise because these actions are cheaper in terms of execution time. The best-cost solution using the readability metric coincides with the second plan, since the *Tree Model* is the parameter the minimizes the cost for the training action.

Most of the action costs rely on formulae affected by pre-defined constants for each parameter (e.g. a constant fluent defined in the initial state, such as `(= (model-percentage-incorrect NeuralNetwork) 5)`. These constant values were set by a data mining expert in order to reproduce common results depending on different executed tasks. Accordingly, the total cost for a solution is taken as an estimation, since the real cost in this application can not be known in advance. Sensing the environment after each action execution could establish real costs, but we have not included this process at the current state of the project.

Plan to KFML

This section describes the translator that generates the KFML files. Once the planner generates a plan, it has to be translated into a KFML file, so it can be executed by the WEKA Knowledge Flow. The translator reads a plan in an XML format. Figure 9 shows a plan (in a standard format, not XML) generated by SAYPHI using the PDDL problem shown in Figure 6.

The translator generates as output a new KFML file with an equivalent plan plus some new actions that the WEKA Knowledge Flow can execute. Each action in the PDDL Domain corresponds to one or many WEKA components. Therefore, the translator writes for each action in the plan the corresponding set of XML tags that represent the WEKA

```
0: (LOAD-DATASET INITIALDATASET)
1: (APPLY-TRANSFORMATION-ATTRIBUTESELECTION
          INITIALDATASET ATTRIBUTESELECTION4)
2: (DATASETPREPARATION INITIALDATASET CROSS-VALIDATION)
3: (TRAIN-CLASSIFICATION TREE TREEMODEL TREEMODEL1
          INITIALDATASET CLASS CROSS-VALIDATION)
4: (TEST-CLASSIFICATION TREE INITIALDATASET
          INITIALDATASET CLASS CROSS-VALIDATION RESULT)
5: (VISUALIZE-CLASSIFICATION-MODEL TREE
          TREEMODEL GRAPH INITIALDATASET CLASS)
6: (VISUALIZE-RESULT TREE INITIALDATASET
          CROSS-VALIDATION RESULT TEXT)
```

Figure 9: A example plan generated by SAYPHI.

```
...
<object class="weka.gui.beans.BeanInstance" name="2">
  <object class="int" name="id" primitive="yes">2</object>
  <object class="int" name="x" primitive="yes">450</object>
  <object class="int" name="y" primitive="yes">145</object>
  <object class="java.lang.String" name="custom_name">
    CrossValidationFoldMaker
  </object>
  <object class="weka.gui.beans.CrossValidationFoldMaker" name="bean">
    <object class="int" name="seed" primitive="yes">1</object>
    <object class="int" name="folds" primitive="yes">10</object>
  </object>
</object>
...
```

Figure 10: A part of the KFML file corresponding to the `DATASETPREPARATION` action in the solution plan presented in Figure 9.

component. For instance, Figure 10 shows a brief section of the KFML file generated from the plan in Figure 9. This part corresponds to the third action in the plan, i.e., the `DATAPREPARATION` action.

Actions appearing in a plan can correspond to one of these cases:

- The action corresponds to exactly one WEKA component. For instance, the `LOAD-DATASET` action corresponds to the `ArffLoader` component of the knowledge flow. Some additional information may be needed from the PMML file (e.g., the URI of the dataset to be loaded).
- The action corresponds to many WEKA components and the action parameters decide which component needs to be selected. For instance, in the `DATASETPREPARATION` action, the second parameter indicates the type of operation. Thus, `CROSS-VALIDATION` corresponds to the `CrossValidationFoldMaker` component in the knowledge flow, and `SPLIT` corresponds to the `TrainTestSplitMaker` component.
- The action corresponds to many WEKA components and the action parameters only specify the name of a technique or model. In these cases, the translator needs to extract that information from the PMML file in order to decide which KFML components should be selected. The information extracted from the PMML file includes the name of the component and the parameters that has to be written in the XML code. For instance, the `TREEMODEL` parameter of the `TRAIN-CLASSIFICATION` action corresponds to the J48 algorithm and some other component

properties defined in the PMML file (See Figure 6).

After writing all components into the KFML file, the translator connects them in the order specified by the plan using the linking primitives of KFML. Finally, the translator adds some extra components in order to save the information generated during the execution. That information are the learned models and the results of the plan execution. Figure 11 shows the knowledge flow diagram that the WEKA GUI presents when reading the KFML file generated for this example.

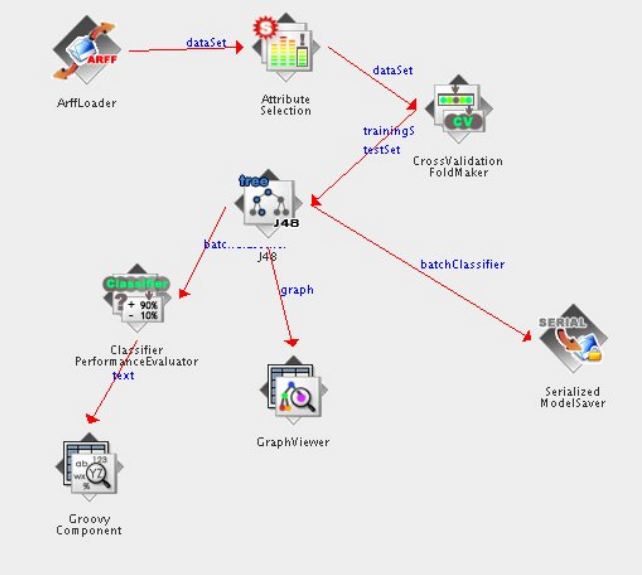

Figure 11: The knowledge flow diagram of the KFML file translated from the plan in Figure 9

Conclusions and Future Work

This paper presents a proposal for modelling data mining tasks by using automated planning, based on extensive use of standard representation languages. The contribution of the work is twofold: modelling the data mining task as an automated planning task and implementing translators able to compile any data mining episode represented in the standard language PMML into the planning standard language PDDL. We have defined a PDDL domain that contains actions to represent all the possible DM tasks (transformations, training, test, visualization, ...). The domain is assumed to stay fixed for all the DM episodes, but each action contains preconditions to control its activation. The PDDL problems are automatically translated from a PMML file representing a DM episode adding the propositions for activating the allowed actions in the particular DM episode. This model allows to deal with plan metrics as minimizing the total execution time or the classification error. During the planning process the metrics are computed from a planning point of view giving some estimated values for their increments experimented in the actions, but we cannot know their true value until a DM Tool executes them. Once the planner solves a problem the solution plan is translated into a KFML file to be executed by the Knowledge Flow of WEKA. The generated model and the statistics are returned to the user. We have implemented a distributed architecture to automate the process.

In the future, we would like to generate several plans to solve the same problem, to execute them in WEKA and to return all the models to the user. Probably, the best models according to the planner are not necessarily the best models according to the user. Thus, we would also like to apply machine learning for improving the plan generation process and the quality of the solutions.

References

Amant, R. S., and Cohen, P. R. 1997. Evaluation of a semi-autonomous assistant for exploratory data analysis. In *Proc. of the First Intl. Conf. on Autonomous Agents*, 355–362. ACM Press.

De la Rosa, T.; García-Olaya, A.; and Borrajo, D. 2007. Using cases utility for heuristic planning improvement. In *Case-Based Reasoning Research and Development: Proceedings of the 7th International Conference on Case-Based Reasoning*, 137–148. Belfast, Northern Ireland, UK: Springer Verlag.

Fayyad, U.; Piatetsky-Shapiro, G.; and Smyth, P. 1996. From data mining to knowledge discovery in databases. *AI Magazine* 17(3):37–54.

Fernández, F.; Borrajo, D.; Fernández, S.; and Manzano, D. 2009. Assisting data mining through automated planning. In Perner, P., ed., *Machine Learning and Data Mining 2009 (MLDM 2009)*, volume 5632 of *Lecture Notes in Artificial Intelligence*, 760–774. Springer-Verlag.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 61–124.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill.

Morik, K., and Scholz, M. 2003. The miningmart approach to knowledge discovery in databases. In *In Ning Zhong and Jiming Liu, editors, Intelligent Technologies for Information Analysis*, 47–65. Springer.

Provost, F.; Bernstein, A.; and Hill, S. 2005. Toward intelligent assistance for a data mining process: An ontology-based approach for cost-sensitive classification. *IEEE Transactions on Knowledge and Data Engineering* 17(4).

Witten, I. H., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd Edition, Morgan Kaufmann.