PolyAnalyst Data Analysis Technique and Its Specialization for Processing Data Organized as a Set of Attribute Values

Mikhail V. Kiselev[1], Sergei M. Ananyan[2], Sergei B. Arseniev[1]

[1]Megaputer Intelligence Ltd., 38 B.Tatarskaya, Moscow 113184, Russia
megaputer@glas.apc.org
[2]IUCF, Indiana University, 2401 Sampson Lane, Bloomington, IN 47405
sananyan@indiana.edu

Abtract. The data analysis techniques of the PolyAnalyst data mining system [2] are based on the automated synthesis of functional programs treated as the multi-dimensional non-linear regression models. This approach provides the system with two valuable properties: 1) it can discover in data the hidden relations that might be of a great variety of forms, 2) it can explore arbitrarily complexly structured data if the corresponding data access primitives are provided. The paper contains a formal description of the final version of the basic PolyAnalyst mechanisms, which are utilized in the general case, as well as in a particular case of data organized as a set of attribute values (SAV), which is the most common format for data explored by KDD methods.

1 Introduction

A great variety of methods proposed for the automated discovery of numerical relations in data can be ordered with respect to some measure of generality of the functional relations found or, alternatively, measure of their computational complexity. On the one pole are the fast algorithms discovering very narrow classes of relations, such as, for example, the linear regression. On the opposite pole are the methods based on an extensive search in wide sets of all possible relations. The latter methods are able to discover and formalize complex non-linear dependencies but the price to pay for their generality is a very large computational time. As examples of this kind of systems one can mention FAHRENHEIT [7], ARE [5], or 49er [6]. These systems construct more involved formulae that express the relations in data by combining simpler formulae using the functional composition mechanism, and eventually finding a sufficiently accurate form for the relation. Probably the most extreme position in this row is occupied at present by the PolyAnalyst data mining system [2; 3; 4], whose main module formulates and tests hypotheses about the sought relation in the form of programs automatically written in a simple internal functional language (henceforth this main module of the system will be referred to as simply

PolyAnalyst). The internal programming language has a sufficient expressive power to formalize any relation which can be expressed in an algorithmic form if a necessary set of functional primitives is provided.

Since the search for the best regression model in a set of functional programs is a very difficult problem, the logical structure of the PolyAnalyst system is quite complex. The system includes the following components, which are described in detail in the present paper:

- internal programming language;
- mechanism preventing creation of trivial and equivalent programs;
- program evaluation module;
- strategy of search in space of functional programs.

PolyAnalyst implements a very general mechanism for generating and testing the regression models. For example, this mechanism can work with data that have an arbitrary structure because the programs generated by PolyAnalyst access the explored data via the special data access primitives, which can provide an access to specific elements of vectors, matrices, lists, and other arbitrarily complex data structures. However, in the case when the database records are represented as sets of scalar values, some of the PolyAnalyst algorithms can be implemented much simpler. An exact definition of this data format and the corresponding modification of the PolyAnalyst techniques specialized for the exploration of data represented as sets of scalar values, are the subject of the last section of the paper.

2 PolyAnalyst internal programming language

We start the description by considering the PolyAnalyst internal programming language, which is used for formulating hypotheses about the relations in data. This language is a functional programming language in the sense defined in [1]. Similar to any other programming language, the PolyAnalyst internal language includes three principal components: data types, functional primitives, and control structures, which can also be considered as the methods for constructing more complex programs from the simpler programs.

The data types of the PolyAnalyst internal language form the two classes: universal data types, which are defined for all application domains, and user-defined domain-specific data types. The former class includes only two data types, namely, *boolean*, denoted as **L**, and *numerical* (real number), denoted as **N**. The **N** type is the only data type containing an infinite number of values. All other data types, including **L** and all user-defined types, involve finite sets of values. To clarify the term "user-defined data types" it is necessary to explain our understanding of the concept of the structure of the explored data. We consider the records of the analyzed database as sets of mappings from some sets to other sets. Say, the data represented as a two-dimensional matrix of numerical values can be considered as a mapping from the direct product of the sets of the vertical and horizontal positions to **N**: $PosX \times PosY \rightarrow \mathbf{N}$. Generally speaking, any data structure can be thought of as a mapping from some sets of keys

determining the access to individual values, to the sets of these values, so that such an approach does not limit our ability to work with arbitrarily complexly organized data.

In our formalism the properties of every data type are determined by two characteristics. The first characteristic describes the ordering properties, namely, whether the relation “greater than” or the operator “next” are defined for the data of this type. The second characteristic is called enumerability. It determines whether the instruction "For each value from the data type X perform the following actions ..." makes sense for the considered data type.

The functional primitives of the PolyAnalyst internal language are at the same time the simplest programs of this language, so that prior to describing them it is necessary to explain how the programs synthesized by the system are represented. A program P is considered as an object having a certain set (possibly empty) of inputs **in**(P) and one output. Every input $\alpha \in$ **in**(P) is marked by its data type DT(α) and also by some other attributes to be discussed below. The data type of the program output is denoted as DT(P). Every input α of the program can be assigned some value $p(\alpha)$ in accordance with its data type. The set of all possible mappings from the set of inputs **in**(P) to the set of their values will be denoted as *EVIN*(P). For every $p \in$ *EVIN*(P) the value P(p) returned by the program can be computed as a result of the sequence of operations determined by its internal structure. If the program contains the so called data access primitives (see below) then a database record for which the program is evaluated should be specified. In that case the output value of the program depends on the database record number i explicitly: P(p, i).

The functional primitives which can be included in the programs created by PolyAnalyst, are also broken into two classes: universal and user-defined primitives. The first class includes various operations defined on the universal data types **L** and **N**. These are the boolean operations AND, OR, and NOT, which are represented as primitives with two (AND, OR) or one (NOT) inputs of the type **L** and an output of the type **L**. As a generalization of the numerical relational operators a ternary primitive *inr* with the prototype *inr*:**L**(x:**N**, y:**N**, z:**N**) is used. The value of this primitive is the value of the proposition $y \leq x < y + z$. In addition to these primitives the universal primitives include a so called TF-commutator *if*:**N**(b:**L**, x:**N**, y:**N**). If b = 1 then the value of this primitive is x otherwise it is y. The user-defined primitives are divided to primitives generated automatically for defined data types, data access primitives, and special primitives. For example, for each data type ***T*** a primitive expressing the equality relation with the prototype **L**(***T***, ***T***) and the TF-commutator with the prototype ***T***(**L**, ***T***, ***T***) are created automatically. The primitives implementing the "greater than" relation and the "next" operation are produced if the respective data types are described as ordered. The prototypes and the bodies of the data access primitives are determined by the structure of the analyzed database records. For example, if the records are organized as two-dimensional matrices containing the values of the type *D*, then PolyAnalyst creates a primitive with the prototype *D(PosX, PosY)* where *PosX* and *PosY* are the data types representing the horizontal and vertical positions in the matrix. And finally, the subgroup of special primitives includes the primitives corresponding to the operations specific for the explored application domain. For example, the calculation of the sine function might be required only in a narrow class of the application areas. Therefore a primitive implementing the

calculation of sine should be defined explicitly by its body and prototype when necessary.

As it has been mentioned before, the functional primitives are considered as the simplest programs. In order to create more complex programs from the simpler programs, several production methods (or control structures) are used. The PolyAnalyst internal language has two basic types of the production methods: functional composition and iteration/recursion.

1. **Functional composition**. A program created using the functional composition is defined by the quadruplet $P_{FC} = <P_{up}, \Pi_{down}, A \subset \mathbf{in}(P_{up}), m{:}A \xrightarrow{m} \Pi_{down}>$, where Π_{down} is a set of programs (it must be non-empty) and $DT(m(\alpha)) = DT(\alpha)$. A new program P_{FC} has the following syntactic characteristics: $DT(P_{FC}) = DT(P_{up})$, $\mathbf{in}(P_{FC}) = \bigcup_{P \in \Pi_{down}} \mathbf{in}(P) \cup \mathbf{in}(P_{up}) \backslash A$. The semantics of this construction is quite obvious. To determine the value of P_{FC} for given input values the values returned by the programs comprising Π_{down} are calculated. Then every input α of P_{up}, which belongs to A, is assigned the value of $m(\alpha)$ and P_{up} is evaluated. Its output value becomes the output value of P_{FC}.

2. **Iteration/recursion**. In contrast with a simple and clear functional composition, the production method dedicated to creating iterative and recursive constructions is very complex. Due to the space limitation, we provide here only its formal definition without supplying any additional comments or examples concerning this production method. The most general form of this construction is expressed by the following twelve components: $P_{iter} = <P_{pred}, P_{ord}, P_{cond}, \Pi_{act}, \Omega, A_{pred} \subset \mathbf{in}(P_{pred}), A_{ord} \subset \mathbf{in}(P_{ord}), A_{iter} \subset \mathbf{in}(P_{cond}) \cup \bigcup_{P \in \Pi_{act}} \mathbf{in}(P), m_{pred}{:}A_{pred} \xrightarrow{m_{pred}} \Omega, m_{ord}{:}A_{ord} \xrightarrow{m_{ord}} \Omega, m_{iter}{:}A_{iter} \xrightarrow{m_{iter}} \Omega \cup \Pi_{act}, \boldsymbol{out} \in \Omega \cup \Pi_{act}>$, where P_{xxx} are programs, Π_{act} is a set of programs, and Ω is a set of loop variables. From the syntactic point of view, the loop variables are the objects which have a single attribute - their data type (this type should be enumerable). The iterative/recursive construction is syntactically correct if the following additional conditions hold: $DT(P_{pred}) = DT(P_{cond}) = \mathbf{L}$, $DT(P_{ord}) = \mathbf{N}$, $DT(m_{xxx}(\alpha)) = DT(\alpha)$ for all m_{xxx}. A special pseudo-program without inputs denoted as $\Im$ may be substituted in place of some components of the considered construction. The output value of this pseudo-program always equals to 1. For example, if $\Omega = \varnothing$, then P_{pred} and P_{ord} should be $\Im$. The prototype of P_{iter} is defined as $DT(P_{iter}) = DT(\boldsymbol{out})$, $\mathbf{in}(P_{iter}) = \bigcup_{P \in \Pi_{t}} \mathbf{in}(P) \cup (\mathbf{in}(P_{pred}) \backslash A_{pred}) \cup (\mathbf{in}(P_{ord}) \backslash A_{ord}) \cup (\mathbf{in}(P_{cond}) \backslash A_{iter})$. The semantics of this construction is determined by the following algorithm for its evaluation. (Note that the mappings m_{xxx} describe the method of passing the values to the inputs of programs included in the construction.)

1. If $\Omega \neq \varnothing$, create a list **LOOPVAR** of all combinations of the possible loop variable values for which the value of P_{pred} equals to 1. The method of passing the values of the loop variables to the inputs of P_{pred} is determined by the mapping m_{pred}.

2. If $\Omega \neq \varnothing$, sort the list **LOOPVAR** in the order of ascending values returned by P_{ord} for the combinations of the loop variable values from **LOOPVAR**. The **LOOPVAR** list can be considered as a matrix LV[i,ω], where i is the variable value combination number, and ω is the loop variable.

3. $i \leftarrow 1$.

4. Calculate the values of all the programs from the set Π_{act}. The values of their inputs are determined by the following rule. If $m_{iter}(\alpha) \in \Omega$, then the value of the input α equals to LV[1,$m_{iter}(\alpha)$], otherwise it is equal to the value of the respective input of P_{iter}.

5. Evaluate P_{cond}.

6. If $\Omega \neq \varnothing$ and i = <number of rows of LV>, or the value of P_{cond} equals to 0, stop the computation. Take the value of ***out*** as the value of the whole construction P_{iter}.

7. $i \leftarrow i + 1$.

8. Calculate the values of all programs from the set Π_{act}. The values of their inputs are taken from the loop variables, the outputs of the programs that belong to Π_{act}, or from the inputs of P_{iter} in accordance with the mapping m_{iter}. For example, if $m_{iter}(\alpha) = \omega \in \Omega$, then LV[$i$,$\omega$] should be taken as the value of α.

9. Go to step 5.

Passing the values between the components of this production is depicted schematically in Fig. 2. In addition to the described general form, the iteration/recursion production method has several special forms which are not considered here.

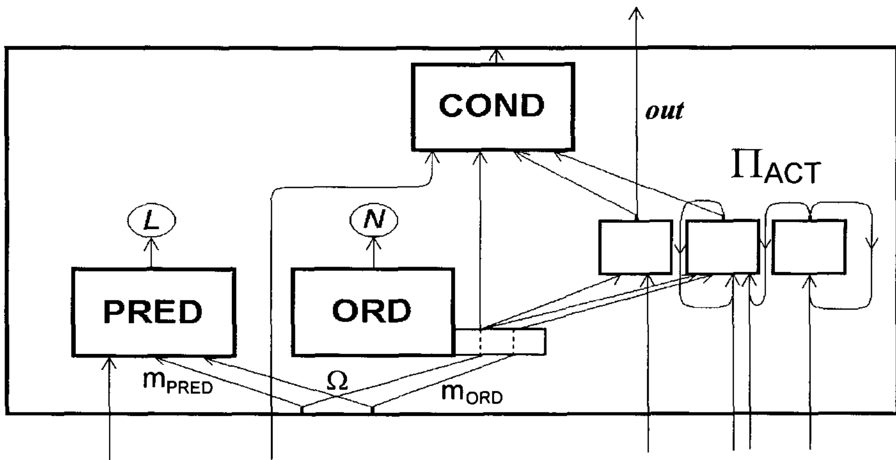

Fig. 1 Iteration/recursion production method.

Although the two discussed production methods are sufficient to provide the PolyAnalyst internal language with the expressive power of a universal programming language, an important special case of the functional composition was selected as a third production method. This mechanism is used for representing numerical dependencies. It is based on the inclusion of the programs returning numerical values in the form of rational expressions (i.e. polynomial divided by polynomial). The reasons for singling out this form of numerical relations, as well as the details of the implementation of this production method are described in [3] and will not be discussed in the present paper.

3 Prevention of building trivial and equivalent programs

Applying the production methods to the existing programs we obtain new programs. However, one cannot guarantee that the new programs will be "good" programs from the semantic point of view. The synthesized programs should satisfy the following three requirements to be considered as semantically correct:

1. *Dependence on all inputs.* This requirement is applicable if $\mathbf{in}(\mathrm{P}) \neq \varnothing$.
$\forall \alpha \in \mathbf{in}(\mathrm{P}) \exists p, q \in EVIN(\mathrm{P}): \mathrm{P}(p) \neq \mathrm{P}(q) \wedge \forall \beta \in \mathbf{in}(\mathrm{P}) \setminus \{\alpha\}: p(\beta) = q(\beta)$.

2. *Dependence on a database record.* This requirement is applicable to the programs containing data access primitives. $\exists p \in EVIN(\mathrm{P}) \exists i \exists j: \mathrm{P}(p, i) \neq \mathrm{P}(p, j)$.

3. *Inequivalence to existing programs.* For all existing programs Q for which there exists an isomorphism $f: \mathbf{in}(\mathrm{P}) \xrightarrow{f} \mathbf{in}(\mathrm{Q})$ such that $\mathrm{DT}(f(\alpha)) = \mathrm{DT}(\alpha)$ and for all such isomorphisms: $\exists p \in EVIN(\mathrm{P}): \mathrm{P}(p) \neq \mathrm{Q}(f[p])$, where $f[p]$ is defined by the equality $f[p](\alpha) \overset{def}{=} p(f^{-1}(\alpha))$.

In order to sift out the programs not obeying these conditions the following mechanisms are utilized:

a. Inputs and outputs of the programs are marked by an additional flag called "consistency type" so that inputs and outputs of certain combinations of consistency types cannot be connected in the process of creating new programs. For example, the equivalence NOT(a<b) ⇔ a=b ∨ a>b is eliminated by declaring the input of the NOT primitive and the output of the LESS_THAN primitive as inconsistent.

b. The symmetry properties of the program inputs are considered. Only one representative of symmetrically equivalent productions is selected. For example, OR(P(...), ...) is retained, while OR(...,P(...)) is rejected.

c. Similarly, the requirement that some inputs cannot be connected to the same output is taken into account. This helps in avoiding the equivalencies like OR(P(...), P(...)) ⇔ P(...).

d. For the commutative productions a certain fixed order of their application is selected. This measure eliminates the equivalencies of the kind: P(NOT*(x), y, z, w*) = IF(Q*(x, y), z, w*), where P(*x, y, z, w*) = IF(OR*(x, y), z, w*), Q(*x, y*) = OR(NOT*(x), y*).

e. Direct tests are applied to check whether the requirements 1 and 2 hold.

f. For each constructed program PolyAnalyst calculates a special value called the input-output signature, which depends on the values of its inputs and the respective returned values. The procedure of calculating this signature guarantees that the programs equivalent in the sense defined in requirement 3 have the same signature values. The procedures with equal signatures are tested for satisfying the requirement 3.

4 Evaluation of the constructed programs

As it has been mentioned, the created programs are considered as solutions of some problem. For example, if the problem of the discovery of a numerical relation is being solved, then the programs constructed by PolyAnalyst are treated as regression functions and are evaluated in terms of standard error of the respective regression models. Strictly speaking, not every program is considered as a potential solution: the programs, which involve no data access primitives, and even some classes of programs including such primitives serve only as components for other programs. The module performing the evaluation of programs can realize an arbitrary functional ***Ev***[P], which should be minimized. This feature provides the system with a great flexibility: the ability to solve a wide class of KDD and optimization problems. It should be noted that each program P represents in general a set of mappings from a set of database records to DT(P), parametrized by the values of its inputs. Therefore for an individual program P the problem of finding the best values for its inputs must be solved. Depending on the functional ***Ev***[P] and the form of the program P, the methods of combinatorics, numerical optimization, or other approaches may be used to solve this problem.

5 GT-search

The last but not least part of the PolyAnalyst system is a module that chooses the production methods, and components for building new programs. The generation of new programs is performed by two processes. The first process, with a lower priority, implements the full search in the space of programs in the order of increasing complexity (which approximately equals the number of primitives constituting the program). The other process, which has a higher priority, creates new programs using the so called generalizing transformations (GT) [3]. The application of GT to an existing program yields a new program called a GT-derivative. The program P′ is called a GT-derivative of the program P (that is denoted as $P \overset{GT}{\succ} P'$) iff:

1. There exists an isomorphism $f\colon \mathbf{in}(\mathsf{P}) \xrightarrow{f} I \subset \mathbf{in}(\mathsf{P}')$ such that DT(f(α)) = DT(α);

2. There exists an isomorphism $F\colon EVIN(\mathsf{P}) \xrightarrow{F} E \subset EVIN(\mathsf{P}')$ such that $q = F(p)$ => $q(f(\alpha)) = p(\alpha)$ and

$\forall \alpha \in \mathbf{in}(\mathsf{P}') \setminus I \forall p, q \in E\colon p(\alpha) = q(\alpha) \wedge \forall p \in E\colon \mathsf{P}'(p) = \mathsf{P}(F^{-1}(p))$.

For any program there exists a large number of classes of transformations of the program structure, which lead to the creation of a GT-derivative of this program. The success of the utilization of the GT-search for navigating in the space of programs is based on an obvious fact that $\mathsf{P} \overset{GT}{\succ} \mathsf{P}' \Rightarrow \boldsymbol{Ev}[\mathsf{P}'] \le \boldsymbol{Ev}[\mathsf{P}]$. This property makes it possible to organize a GT-based search in the space of programs in the following way.

When the process of the full search finds a program for which *Ev*[P] is sufficiently small, this program becomes a parent for a generation of programs created from this program with the help of GT. If one of these programs demonstrates a significant decrease of ***Ev***, this program in turn is taken as a starting point for building new programs with the help of GT and so on. By utilizing this approach the system can build rather complex programs over a reasonable period of time. This description of the GT-search concludes the discussion of the internal PolyAnalyst mechanisms employed in a general case. The next section is devoted to discussing a special case when the data has a "set of attribute values" (SAV) structure.

6 Specialization of PolyAnalyst mechanisms for the case of data represented as a set of attribute values (SAV)

First of all, let us furnish a precise definition of the SAV data format. From the point of view of PolyAnalyst a data format is defined completely by specifying the data access primitives, a set of possible data types, and a set of the user-defined primitives. The term "set of attribute values" implies that each considered database record constitutes a set of scalar values of different types. Therefore for every position of the data record a data access primitive with no inputs is generated. The data type of the output of this primitive matches the data type of the attribute value in the respective position. Also an additional data type is introduced for every position of the record, which contains unordered non-numerical values. Beside the access primitives, the only class of the user-defined primitives which can be introduced for a domain of that kind is the class of equality primitives for additional data types. It can be easily shown that this set of data types and functional primitives leaves very few possibilities for employing the iteration/recursion production method for building new programs. For this reason, and also because the functional composition method is much easier to implement, it is reasonable to use in the SAV application domains only the functional composition method. Furthermore, in this case the majority of programs are generated utilizing an important subclass of the functional composition method, namely the production of rational expressions.

Beside the internal language, the other component of the PolyAnalyst system, which is influenced greatly by the assumption of the SAV data organization, is the GT-search mechanism. Since in this case the **rational** production method plays a very important role, the following two kinds of generalizing transformations applicable to rationals are used most often:

1. Let us denote the rational expression subjected to GT as $\mathrm{P} = A/B$ where A and B are polynomials. If Q is a program returning a numerical value, while C and D are polynomials, then as it can be readily seen, the program $\mathrm{R} = \dfrac{\mathrm{Q} * A + C}{\mathrm{Q} * B + D}$ is a GT-derivative of P.

2. If we multiply any term in A or B by a construction IF(Q, *a*, *b*), where Q is a program returning a boolean value, while *a* and *b* are inputs of the type **N**, then the

new rational expression will be a GT-derivative of P. This is true because if $a = b = 1$ then IF(Q, a, b) $\equiv 1$.

It should be noted that the basic commercially available version of PolyAnalyst utilizes only these two classes of GT.

7 Conclusion

The history of successful utilization of PolyAnalyst in various fields including banking, marketing, manufacturing, and many other fields proves the efficiency of applying the automated program synthesis techniques to KDD problems. The universality of the described approach is achieved due to the absence of any inherent limitations on the structure of analyzed data, as well as on the procedure of evaluating the built programs in accordance with arbitrary criteria implemented in the PolyAnalyst's program evaluation module. The GT search and the mechanisms suppressing the generation of trivial and equivalent programs solve, or at least soften the problem of combinatorial growth of the number of the generated programs. The assumption of the SAV data structure allows one to make important simplifications, which increase the performance of the system even further.

References

1. Backus, J. (1978) Can programming be liberated from the von Neumann style? *Commun. ACM*, v.21, pp 613-541.
2. Kiselev, M.V. (1994) PolyAnalyst - a machine discovery system inferring functional programs, In: *Proceedings of AAAI Workshop on Knowledge Discovery in Databases'94*, Seattle, pp. 237-249.
3. Kiselev, M.V., Arseniev, S.B. (1996) Discovery of numerical dependencies in form of rational expressions, in; *Proceedings of ISMIS'96 (Ninth International Symposium on Methodologies for Intelligent Systems) poster session*, Zakopane, Poland, pp. 134-145.
4. Kiselev, M.V., Ananyan, S. M., and Arseniev, S. B. (1997) Regression-Based Classification Methods and Their Comparison with Decision Tree Algorithms, In: *Proceedings of 1st European Symposium on Principles of Data Mining and Knowledge Discovery*, Trondheim, Norway, Springer, pp 134-144.
5. Shen Wei-Min (1990) Functional Transformations in AI Discovery Systems, *Artif.Intell.*, v.41, pp 257-272.
6. Zembowicz, R., Zytkow, J. M. (1992) Discovery of Equations: Experimental Evaluation of Convergence, In: *Proceedings of AAAI-92*, AAAI Press, Menlo Park, CA, pp 70-75.
7. Zytkow J.M., Zhu J. (1991) Application of Empirical Discovery in Knowledge Acquisition, In: *Proceedings of Machine Learning - EWSL-91*, pp 101-117.