Piecewise Regression Learning in CoReJava Framework

Juan Luo and Alexander Brodsky

***Abstract*—CoReJava (Constraint Optimization Regression in Java) is a framework which extends the programming language Java with built-in regression analysis, i.e., the capability to do parameter estimation for a function. CoReJava is unique in that functional forms for regression analysis are expressed as first-class citizens, i.e., as Java programs, in which some parameters are not a priori known, but need to be learned from training sets provided as input. Typical applications of CoReJava include calibration of parameters of computational processes, described as OO programs. If-then-else structures of Java language are naturally adopted to create piecewise functional forms of regression. Thus, minimization of the sum of least squared errors involves an optimization problem with a search space that is exponential to the size of learning set. In this paper, combinatorial restructuring algorithm is proposed to guarantee learning optimality and furthermore reduce the search space to be polynomial in the size of learning set, but exponential to the number of piece-wise bounds. Heaviside restructuring algorithm, which expresses the piecewise linear regression function using a unified functional format, instead of multiple pieces, is proposed to decrease the searching complexity further to be polynomial in both the size of learning set and the number of piece-wise bounds, while the learning outcome will be an approximation of the optimality.**

***Index Terms*—Combinatorial Restructuring, Heaviside Restructuring, Object-Oriented Programming, Piecewise Regression**

I. Introduction

Regression Analysis (RA) is a widely-used statistical technique for investigating and modeling the relationship between variables (see [1] for overview of RA). Given as input to regression learning is a parametric functional form, e.g., $f(x_1,x_2,x_3) = p_1x_1 + p_2x_2 + p_3x_3$, and a set of training examples, e.g., tuples of the form (x_1,x_2,x_3,f), where f is an experimental observation of the function f value for an input (x_1,x_2,x_3). Intuitively, the problem of regression analysis is to find the unknown parameters, e.g., p_1,p_2,p_3 which best approximate the training set. For example, the national housing price can be modeled as a function of such determinants as age of the house, the floor area of the house, neighborhood attributes, and location attributes. This functional form may have unknown parameters, reflecting the relationship between house price and a particular attribute of the house. This regression based on the national data set is an "overall" form of linear regression which is represented by line 'A' in Fig. 1. The regression analysis is applied to all of the data within your dataset. In other words, it gives you a measure of the global rather than local relationships between variables. However, it clearly misses some important local variations between Y and X.

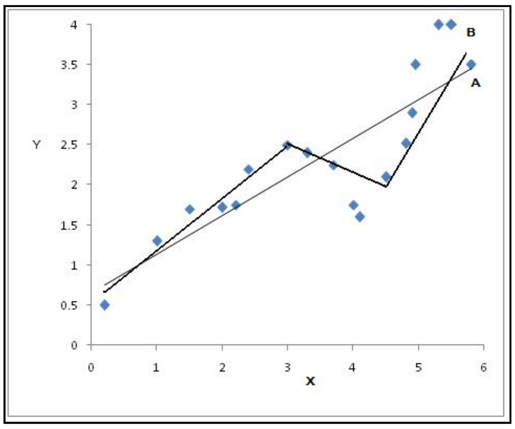


Figure 1: Global and Piecewise Linear Relationship

Piecewise Regression (PWR) are forms of data analysis that allows to evaluate how the relationship between a dependent variable and one or more explanatory variables changes according to the different value intervals in which the explanatory variable resides. So, instead of being a conventional and stationary model, for example, $y= p_0 + p_1x$ in the case of linear regression with a single explanatory variable, the piecewise linear regression model can be expressed as

$$f(p,x) = \begin{cases} f_1(p_1,x) & x < b_1 \\ f_2(p_2,x) & b_1 \le x < b_2 \\ \quad\ldots\ldots & \\ f_k(p_k,x) & b_{k-1} \le x \end{cases} \tag{1}$$

In the PWR expression (1), for different value intervals of the explanatory variable, a specific functional form f_i represents a "segment" of line in the overall problem. There exists a relationship between two variables, Y and X, which needs to be determined from the observed data. A global linear regression model will produce a relationship such as that depicted by line A. A local technique, such as linear "spline" function, depicted by line B, will give a more accurate picture of relationship between Y and X. This is obtained by essentially running three separate regressions over different ranges of the X variable with the constraint that the end points of the local regression lines meet at what are known as "knots".

In the CoReJava framework [2], the above spline functional form of regression analysis can be expressed as a Java program, in which the slopes and intercepts are not priori known, but can be learned from a given training set. One special thing for the knots in the Java program is that the values of knots are not priori known as well. All these parameters will be initialized as a special CoReJava data type

Manuscript received March 22, 2011.

Juan Luo is with the Dept. of Computer Science, George Mason University, Fairfax, VA 22030, USA (corresponding author to provide phone: 703-993-1531; e-mail: jluo2@gmu.edu).

Alexander Brodsky is with Dept. of Computer Science, George Mason University, Fairfax, VA 22030, USA (e-mail: Brodsky@gmu.edu)

– non-deterministic variable, i.e. only the value range of the variable is given during the initialization, instead of the value itself. As we can observe from the Fig. 1, the spline function is composed of three linear pieces. Correspondingly, the function will be represented with three "if" / "else" condition branches. Knots variables are involved in the "if" condition check, which in turn will increase the searching complexity of the problem. The regression learning in CoReJava framework actually involves two steps. First, it analyses the structure of the Java programs to automatically generate a constraint optimization problem from them, in which constraint variables correspond to all non-deterministic variables, and the objective function of the constraint optimization problem to be minimized is the summation of squares of errors w.r.t. the training set. Second, the generated optimization problem will be solved by the non-linear mathematical optimization solver. The searching complexity of non-deterministic variables in the optimization problem will be $O(2^{N\cdot M})$which is exponential in N (the size of the training set) and M (the number of condition checks in which non-deterministic variables are involved).

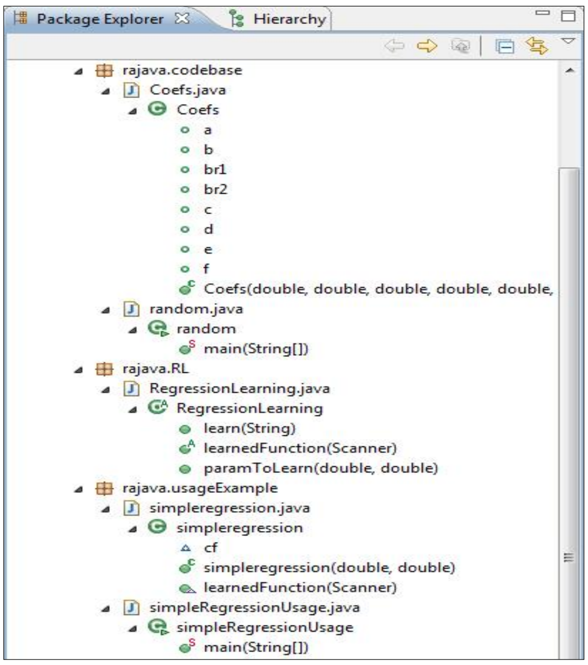
There is growing literature and techniques for examining local relationships in data sets. For example, the use of spline function [3] requires that the knots of piecewise linear model need to be given but the coefficients of the linear problem are left unknown. LOWESS regression in [4] can be used to fit segments of data without a global function. However, it fits data using curves and at the same time, it is so computationally intensive that it is practically impossible to be used. A connectionist model (three layer neural network) [5] is used to solve the piecewise linear regression problem heuristically with clustering and multi-category classification involved. Even though the neural network can show high accuracy on the available data set, it is not feasible to prove that it will show a good performance on all possible input combinations such as unseen data.

This paper proposes two different strategies for piecewise regression learning. One strategy is called combinatorial restructuring with the searching complexity $O(N^P)$, where N is the number of learning sets and P is the number of parameters in the function, i.e., only polynomial in N, as opposed to the original $O(2^{N\cdot M})$which is exponential in N. At the same time, it guarantees optimality of learning. However, for the combinatorial restructuring, the piecewise regression learning has to be repeated totally C_{N+1}^{k-1} number of times. The other strategy is called Heaviside restructuring with the searching complexity of $O(N^{P+K})$, where N is the number of learning set, P is the number of parameters in the function and K-1 is the number of "knots" in the function, i.e., polynomial in the size of data set, as opposed to the original $O(2^{N\cdot M})$, which is exponential in N. Heaviside restructuring is proposed to replace all "if" / "else" decision structures by a unified functional format in `learnedFunction` method. The "spline" function (piecewise functions) can be expressed as a single, continuous function by applying the Heaviside transformation [6]. The Heaviside restructuring decreases the search complexity to be polynomial in the number of learning set but the learning outcome will be an approximation of the optimal solution. Compared to the combinatorial restructuring, Heaviside restructuring improves the learning efficiency by trading with the approximation of optimal learning outcome. Finally, we conduct an experimental study to compare generic CoReJava learning, combinatorial restructuring, and Heaviside restructuring. It shows that both restructurings outperform generic CoReJava regression learning.

This paper is organized as follows. Section II is the overview of CoReJava framework which is exemplified by the implementation of regression learning on the spline function (see Fig. 1). Section III explains key ideas and algorithm of combinatorial restructuring for single-dimensional piecewise regression problem. As to higher dimensional piecewise surface regression model, the combinatorial restructuring cannot be simply reused for general models, however can be applied to solve a special class of the problems in the same domain. Section IV explains key ideas and algorithm for Heaviside restructuring of piecewise regression problem. An experimental study is performed to compare the results of different learning strategies in section V. Finally, we conclude and briefly outline our future work in Section VI.

II. Overview Of The CoreJava Framework Exemplifed By Regression On Spline Function

In this section, we exemplify the use and semantics of regression framework by implementing the spline function example which is described as line 'B' in Fig. 1. The spline function is the kind of estimates produced by a spline regression in which the slope varies for different ranges of the regressors. The spline function can be continuous but usually not differentiable.



```
Package Explorer    Hierarchy
rajava.codebase
  Coefs.java
    Coefs
      a
      b
      br1
      br2
      c
      d
      e
      f
      Coefs(double, double, double, double, double,
  random.java
    random
      main(String[])
rajava.RL
  RegressionLearning.java
    RegressionLearning
      learn(String)
      learnedFunction(Scanner)
      paramToLearn(double, double)
rajava.usageExample
  simpleregression.java
    simpleregression
      cf
      simpleregression(double, double)
      learnedFunction(Scanner)
  simpleRegressionUsage.java
    simpleRegressionUsage
      main(String[])
```

Figure 2: Regression Framework in Eclipse

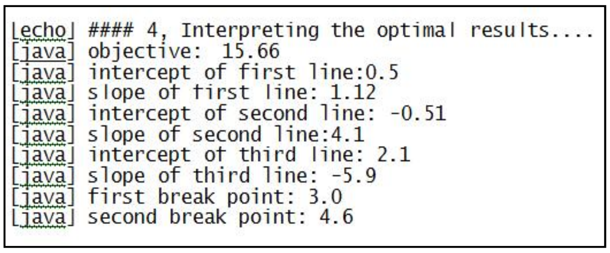
Fig. 2 shows a partially expanded CoReJava library of the regression framework for the spline function example. The three key packages in the example are (1) the existing codebase, that may have been developed not for the purpose of regression learning, (2) RL (Regression Learning), which is system package of the framework, and (3) the usageExample, which instantiates some classes in the codebase and then do as regression learning . The codebase contains all source code implementing the computational process. There is no regression learning related code in this package. The codebase in the example models the spline function with data members of two knots, br1 and br2 and three different set of intercepts and slopes, (a, b), (c, d) and (e, f), i.e., models the function

$$f(x) = \begin{cases} a + bx & x < br1 \\ c + dx & br1 \le x < br2 \\ e + fx & br2 \le x \end{cases} \quad (2)$$

In the `usageExample` package, let us assume that `Coefs` (parameters of spline function) is not priori known, and we would like to learn them using a given training set. To do that, the user can define its own learning behavior by extending an abstract class `RegressionLearning` which belongs to the system package RL. We use the class `simpleregression` to exemplify the user's learning behavior in the example. `Simpleregression` class has a `Coefs` object as data member, which we would like to learn. It defines a `constructor` and a `learnedFunction(Scanner scanner) method.` Method `learnedFunction` describes a functional form written using standard Java, while method `constructor` describes the set of adjustable parameters to that function. Note that the `constructor` method of `simpleregression` uses the method `paramToLearn(double min, double max)` to specify the learning parameters (coefficients and knots) of the `Coefs` object to be learned. The min and max here can specify a value range for the learning parameters to be chosen. The learning parameters are unknown in the constructor. `Simpleregression` class overrides the abstract method `learnedFunction(Scanner scanner) of its super class,` to represent the functional form used in learning. This method reads from the `scanner` object, as input, the X values of training examples and returns their Y values as output. Note that the output Y depends on the object `Coefs`, which contain unknown parameters to be learned. Both the `constructor` and `learnedFunction` methods can make use of an existing Java code base.

The parameter `scanner` is defined as a Java utility class `Scanner` and used to read a set of training. The `Scanner` object that encodes a set of training examples, in the form $(x_1,\ldots,x_n,f_n)$. In the spline function example, each data point is composed of pairs of X and Y values. The size of training set is decided by the information the user has collected. If the size of the table is not big enough, the model selection technique (cross validation) of machine learning will be used to filter the noise of learning sets [7].

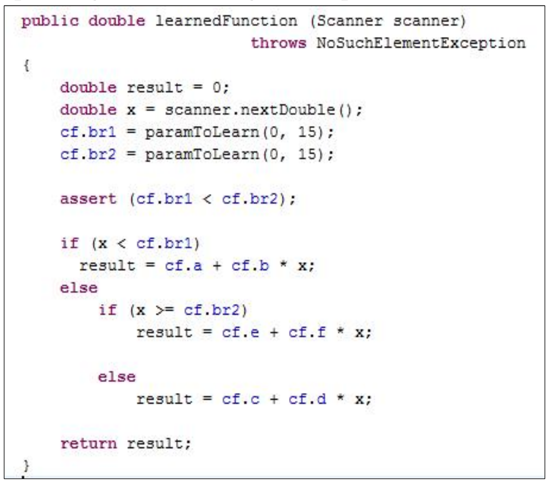
The `simpleRegressionUsage` class is used to show the actual execution of the spline funciton example. The `main` method is defined in this class. Two variables `min_Bound` and `max_Bound` are used to specify the range of coefficients to be chosen and assigned in the main method. Intuitively, the special regression semantics of the `simpleRegressionUsage constructor` is as follows. It constructs a `simpleRegressionUsage` object, in which all the invocations of `paramToLearn` method are replaced with actual values of type double, for which the function defined by the `learnedFunction` method would best approximate the set of training examples. After that`, the` main method will be executed as a regular Java program. The values of all parameters and knots will be printed out. The running result of spline function example, i.e., the values of parameters and knots is displayed in Fig. 3.



```
[echo] #### 4, Interpreting the optimal results....
[java] objective:  15.66
[java] intercept of first line:0.5
[java] slope of first line: 1.12
[java] intercept of second line: -0.51
[java] slope of second line:4.1
[java] intercept of third line: 2.1
[java] slope of third line: -5.9
[java] first break point: 3.0
[java] second break point: 4.6
```

Figure 3: Running Results

To implement regression learning, the compiler of CoReJava framework involves two steps. First, it analyses the structure of the `learnedFunction` method to automatically generate a constraint optimization problem, in which constraint variables correspond to `paramToLearn`, and the objective function to be minimized is the summation of squares of errors w.r.t. the training set, and then solves the optimization problem using the non-linear optimization solver AMPL / SNOPT [8]. AMPL is a comprehensive and powerful algebraic modeling language for linear and nonlinear optimization problems, in discrete or continuous variables. Second, regression framework constructs a regular `myLearning` object, in which all `paramsToLearn` are replaced with the optimal learned parameters, and then continues with regular Java semantics. The constructed optimization problem is solved by an external solver such as AMPL / SNOPT. That is why it inherits the solver's limitations, namely, it can only be solved when the resulting constraint domain is supported by the solver, and furthermore, may return only a local, rather than global, minimum of the corresponding non-linear regression problem.



```
public double learnedFunction (Scanner scanner)
                        throws NoSuchElementException
{
    double result = 0;
    double x = scanner.nextDouble();
    cf.br1 = paramToLearn(0, 15);
    cf.br2 = paramToLearn(0, 15);

    assert (cf.br1 < cf.br2);

    if (x < cf.br1)
      result = cf.a + cf.b * x;
    else
        if (x >= cf.br2)
            result = cf.e + cf.f * x;

        else
            result = cf.c + cf.d * x;

    return result;
}
```

Figure 4: Code Snippet for learnedFunction method

The `learnedFunction` in Fig. 4 represents the decision structure for the linear spline function of spatial housing prices model. In this method, the number of knots in the model is given as two however the thing not known is the exact values of the knots. That's why in the function cf.br1 and cf.br2 are assigned as non-deterministic value type within the range [0, 15]. The nested "if" and "else" decision structures are used to calculate the value of dependent variable Y according to the range in which the input value X lies. If it belongs to the first part of the spline function, i.e. X is less than the value of first knot, Y will be calculated using the first function $Y = a + bX$. Similarly, the value of Y can be calculated if X falls in the interval of the first knot and the second knot, or X falls in the range of the second knot and the end of input domain. In the "if" condition expression of the example, two knots are involved. These two variables are of non-deterministic data type. After the constraint optimization problem is generated for the `learnedFunction` method, a binary variable will be automatically generated for each non-deterministic variable in the conditional expression. Furthermore, each binary variable in the condition expression will increase the search space of the optimization problem by double. If given N data inputs and M "if" statements in the `learnedFunction`, there will be totally N*M of binary variables will be generated and then the searching space will be of complexity $O(2^{N \cdot M})$.

Due to the exponentially increasing complexity of searching space, it will take unreasonable amount of time to solve the above constraint problem. Most times, the solution for the regression learning even can't be found due to the limitation of the mathematical solver. Better strategy need to be proposed to decrease the complexity of searching space. Combinatorial restructuring is proposed first in section III and then Heaviside restructuring is proposed in section IV.

III. Combinatorial Restructuring

A. *Piecewise Regression Problem Definition*

We focus on regression learning of the piecewise function in (1) where $(p_1, p_2, \dots, p_k)$ are parameters of "case" functions and $(b_1, b_2, \dots, b_{k-1})$ are bound parameters which we call "knots". We assume a training data set $\{(x_n, f_n)\}, n = 1, .., N$ is given. The purpose of learning is to find the parameters $(p_1, p_2, \dots, p_k)$ which can best approximate the function set $(f_1, f_2, \dots, f_k)$, and at the same time find the values for the $(b_1, b_2, \dots, b_{k-1})$.

B. *Proposed Learning Approach*

For the generic least squared regression learning of CoReJava, the learning objective is

$$\min \sum_{i=1}^{N} (f(\hat{p}, \hat{b}, x_i) - y_i)^2 \quad (3)$$

The optimization would require N*M binary variables because of the piecewise form of the function $f(p, b, x)$, which corresponds to the search space of $O(2^{N \cdot M})$. To understand the idea of proposed combinatorial restructuring, we start with the simplest case, in which there is a single knot b_1, i.e.,

$$f(p_1, p_2, b_1, x) = \begin{cases} f_1(p_1, x) & x < b_1 \\ f_2(p_2, x) & b_2 \le x \end{cases} \quad (4)$$

Assume that without loss of generality, $(x_1, x_2, .., x_N)$ are sorted in an ascending order. In the combinatorial restructuring, instead of solving (3), we propose to solve N+1 problems, corresponding to the following value intervals of b_1, (-∞,x_1], $(x_1, x_2]$, …, $(x_{N-1}, x_N]$, and (x_N, ∞). Thus, the ith problem will be

$$min \sum_{i=1}^{N} (f(\hat{p}, \hat{b}, x_i) - y_i)^2 =$$

$$\min \left(\sum_{i=1}^{k} (f_1(\hat{p}, \hat{b}, x_i) - y_i)^2 + \sum_{i=k+1}^{N} (f_2(\hat{p}, \hat{b}, x_i) - \right.$$ *yi)2)* (5)

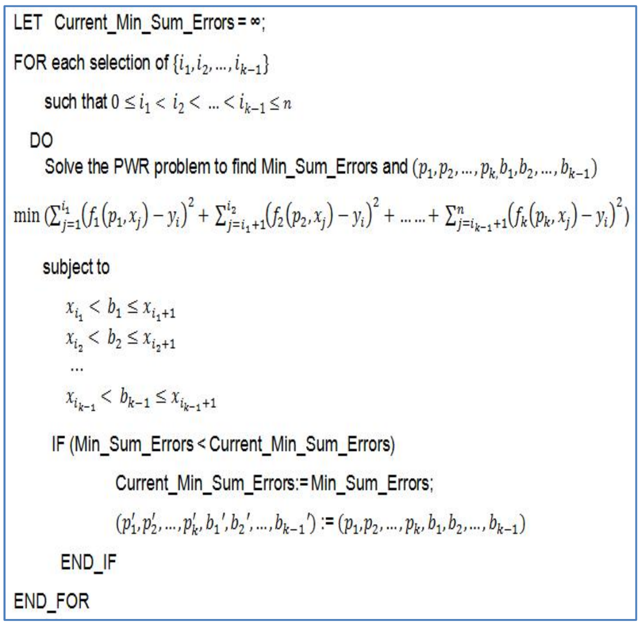
Note that in the above form, we do not need binary variables to represent the piecewise selection since b_1 can only satisfy one of the N+1 cases, it will be suffice to solve N+1 problem (5) and select the solution that gives the minimum. This approach will decrease the searching complexity from $O(2^{N \cdot M})$ to $O(N^P)$ where P is the number of parameters, N is the size of data inputs and M is the number of "if" branches in the `learnedFunction` of the regression learning.

Consider a more general function format (1), N data points in the training data set and $k - 1$ knots. Each knot has N+1 possible value intervals (-∞,x_1], $(x_1, x_2]$, …, $(x_{N-1}, x_N]$, and (x_N, ∞). The function definition assumes that $b_1 < b_2 < \cdots < b_{k-1}$. Let us index the intervals $1, \dots, N$. If $b_1 \epsilon (x_{i_1}, x_{i_1+1}]$, we say that b_1 resides in the interval i_1. Note that $x_0 = -\infty$, so if i_1=0, $-\infty < b_1 \le x_1$. More generally, if b_1 resides on interval i_1, b_2 resides on interval i_2, …, and b_{k-1} resides on interval i_{k-1}, we know that

$$\begin{aligned} x_{i_1} &< b_1 \le x_{i_1+1} \\ x_{i_2} &< b_2 \le x_{i_2+1} \\ &\dots \\ x_{i_{k-1}} &< b_{k-1} \le x_{i_{k-1}+1} \end{aligned} \quad (6)$$

The idea is that under the constraints in (6), for input points $\{x_j\}$ $j = 1, .., i_1$, $f(\hat{p}, \hat{b}, x_i) = f_1(p_1, x_j)$,…, and for input points $\{x_j\}$ $j = i_{k-1}, \dots, n$, $f(\hat{p}, \hat{b}, x_i) = f_k(p_k, x_j)$. Thus, under the constraint (6), the problem $min \sum_{i=1}^{N} (f(\hat{p}, \hat{b}, x_i) -$ *yi)2* can be rewritten as

$$\min_{p_1, .., p_k, b_1, .., b_{k-1}} \Big(\sum_{j=1}^{i_1} \big(f_1(p_1, x_j) - y_i\big)^2 + \sum_{j=i_1+1}^{i_2} \big(f_2(p_2, x_j) - y_i\big)^2 + \dots \dots + \sum_{j=i_{k-1}+1}^{n} \big(f_k(p_k, x_j) - y_i\big)^2\Big) \quad (7)$$



```
LET Current_Min_Sum_Errors = ∞;
FOR each selection of $\{i_1, i_2, \dots, i_{k-1}\}$
    such that $0 \le i_1 < i_2 < \dots < i_{k-1} \le n$
  DO
    Solve the PWR problem to find Min_Sum_Errors and $(p_1, p_2, \dots, p_k, b_1, b_2, \dots, b_{k-1})$
    $\min \big(\sum_{j=1}^{i_1} (f_1(p_1, x_j) - y_i)^2 + \sum_{j=i_1+1}^{i_2} (f_2(p_2, x_j) - y_i)^2 + \dots\dots + \sum_{j=i_{k-1}+1}^{n} (f_k(p_k, x_j) - y_i)^2\big)$
    subject to
      $x_{i_1} < b_1 \le x_{i_1+1}$
      $x_{i_2} < b_2 \le x_{i_2+1}$
      ...
      $x_{i_{k-1}} < b_{k-1} \le x_{i_{k-1}+1}$
    IF (Min_Sum_Errors < Current_Min_Sum_Errors)
        Current_Min_Sum_Errors:= Min_Sum_Errors;
        $(p_1', p_2', \dots, p_k', b_1', b_2', \dots, b_{k-1}') := (p_1, p_2, \dots, p_k, b_1, b_2, \dots, b_{k-1})$
    END_IF
END_FOR
```

Figure 5: Combinatorial Regression Learning Algorithm

This gives rise to our algorithm: enumerate all possible index selection $i_1, \dots, i_{k-1}$ for $b_1, .., b_{k-1}$ respectively, and for each selection solve the problem (7). This idea is summarized in the combinatorial restructuring algorithm in Fig. 5.

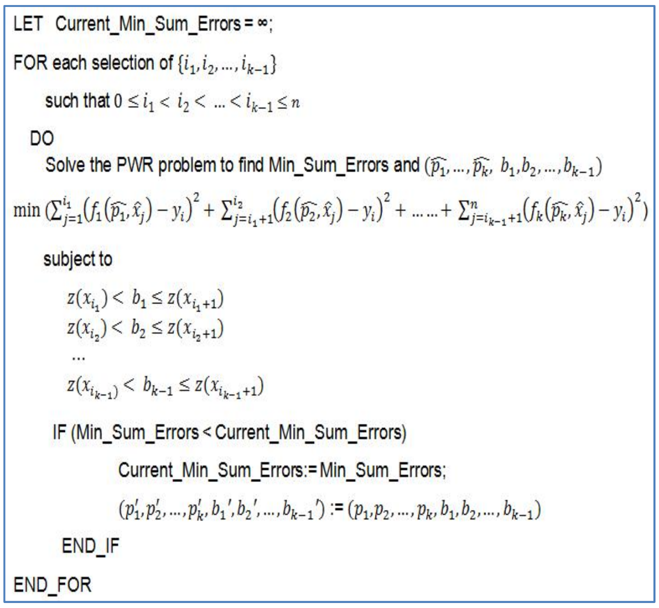
Claim: the combinatorial restructuring algorithm guarantees optimality of the regression problem (1), and requires solving C_{N+1}^{k-1} continuous optimization problems in the complexity of $O(N^p)$.

C. Piecewise Surface Regression Model

We now extend the functional form (1) to a multi-dimensional case, with the following form

$$\begin{cases} f_1(\widehat{p_1}, \hat{x}) & \mathrm{z}(\hat{x}) < b_1 \\ f_2(\widehat{p_2}, \hat{x}) & b_1 \le \mathrm{z}(\hat{x}) < b_2 \\ & \ldots . . \\ f_k(\widehat{p_k}, \hat{x}) & b_{k-1} \le \mathrm{z}(\hat{x}) \end{cases} \quad (8)$$

where $\hat{x} = (x_1, \ldots, x_n)$ and $\mathrm{z}(\hat{x}) = \sum_{i=1}^{n} w_i x_i$ where $\sum_{i=1}^{n} {w_i}^2 = 1$. Note that while this functional form is of multi-dimensional input space, its cases of piecewise functions are expressed with a fixed linear combination $\mathrm{z}(\widehat{x})$, i.e., the axis $(w_1, \ldots, w_n)$. For this case we will sort the learning set $\{(\widehat{x_j}, y_j)\}\, j = 1, .. N$ by the value $z(\widehat{x_j})$ and assume without loss of generality, that $z(\widehat{x_1}), \ldots . z(\widehat{x_n})$ is in increasing order. The combinatorial restructuring algorithm for this case is given in Fig. 6.



```
LET Current_Min_Sum_Errors = ∞;
FOR each selection of $\{i_1, i_2, \ldots, i_{k-1}\}$
    such that $0 \le i_1 < i_2 < \ldots < i_{k-1} \le n$
  DO
    Solve the PWR problem to find Min_Sum_Errors and $(\widehat{p_1}, \ldots, \widehat{p_k}, b_1, b_2, \ldots, b_{k-1})$
min $(\sum_{j=1}^{i_1}(f_1(\widehat{p_1}, \hat{x}_j) - y_i)^2 + \sum_{j=i_1+1}^{i_2}(f_2(\widehat{p_2}, \hat{x}_j) - y_i)^2 + \ldots\ldots + \sum_{j=i_{k-1}+1}^{n}(f_k(\widehat{p_k}, \hat{x}_j) - y_i)^2)$
    subject to
      $z(x_{i_1}) < b_1 \le z(x_{i_1+1})$
      $z(x_{i_2}) < b_2 \le z(x_{i_2+1})$
      ...
      $z(x_{i_{k-1}}) < b_{k-1} \le z(x_{i_{k-1}+1})$
     IF (Min_Sum_Errors < Current_Min_Sum_Errors)
            Current_Min_Sum_Errors:= Min_Sum_Errors;
            $(p_1', p_2', \ldots, p_k', b_1', b_2', \ldots, b_{k-1}') := (p_1, p_2, \ldots, p_k, b_1, b_2, \ldots, b_{k-1})$
     END_IF
END_FOR
```

Figure 6: Combinatorial Regression Learning Algorithm for a Multi-dimensional Case

We can have the similar claim for this case as to the piecewise function in (1). The combinatorial restructuring algorithm guarantees optimality of the regression problem (8), and requires solving C_{N+1}^{k-1}continuous optimization problems in the complexity of $O(N^p)$.

IV. Heaviside Restructuring Of Regression Learning

A. Problem Definition

Same to combinatorial restructuring, we focus on regression learning of the piecewise function in (1) where $(p_1, p_2, \ldots, p_k)$ are parameters of "case" functions and $(b_1, b_2, \ldots, b_{k-1})$ are bound parameters which we call "knots". We assume a training data set $\{(x_n, f_n)\}, n = 1, .., N$ is given. The purpose of learning is to find the parameters $(p_1, p_2, \ldots, p_k)$ which can best approximate the function set $(f_1, f_2, \ldots, f_k)$, and at the same time find the values for the $(b_1, b_2, \ldots, b_{k-1})$. The learning objective is expressed as (3), which can be rewritten as

$$\min_{p_1, .., p_k, b_1, .., b_{k-1}} \Big(\sum_{x_j < b_1}\big(f_1(p_1, x_j) - y_i\big)^2 + \sum_{b_1 \le x_j < b_2}\big(f_2(p_2, x_j) - y_i\big)^2 \ \ldots\ldots + \sum_{b_{k-1} < x_j}\big(f_k(p_k, x_j) - y_i\big)^2\Big) \quad (9)$$

In (9), both $(p_1, p_2, \ldots, p_k)$ and $(b_1, b_2, \ldots, b_{k-1})$ are non-deterministic variables to be searched during the optimization. The sum of least squared errors is composed of multiple itemized summations of least squared errors for every individual piece of the piecewise function $f(p, x)$. As described in Fig. 4, in `learnedFunction` method, a nested "if" and "else" statements are constructed to express every single piece of piecewise linear function. Given N data inputs and M "if" statements in `learnedFunction` method, the searching space will be of complexity $O(2^{N \cdot M})$. To lower the complexity, it is not necessarily to express the piecewise linear function in multiple pieces and each piece has a different functional form. However, it can be expressed as one unified function instead. Consequentially, we can remove "if" and "else" statements in the `learnedFunction` method. We call this approach Heaviside restructuring.

B. PWLR Expressed in Unified Functions

The piecewise linear function in (1) can be expressed a single, continuous unified function by applying Heaviside function (known as the Unit Step function), which is defined as:

$$U(x) = \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x \ge 0 \end{cases} \quad (10)$$

We can perform the transformation by switching on and switching off the appropriate functions at the right time. After transformation, the function in (1) can be rewritten as:

$$f(p, x) = f_1(p_1, x) - f_1(p_1, x) * U(x - b_1) + f_2(p_2, x) * U(x - b_1) - f_2(p_2, x) * U(x - b_2) + \ldots + f_k(p_k, x) * U(x - b_{k-1}) \quad (11)$$

The learning objective for the Heaviside restructuring can be expressed as:

$$\min_{p_1, .., p_k, b_1, .., b_{k-1}} \sum_{i=1}^{N} (f_1(p_1, x) - f_1(p_1, x) * U(x - b_1) + f_2(p_2, x) * U(x - b_1) - f_2(p_2, x) * U(x - b_2) + \ldots + f_k(p_k, x) * U(x - b_{k-1}) - y_i)^2 \quad (12)$$

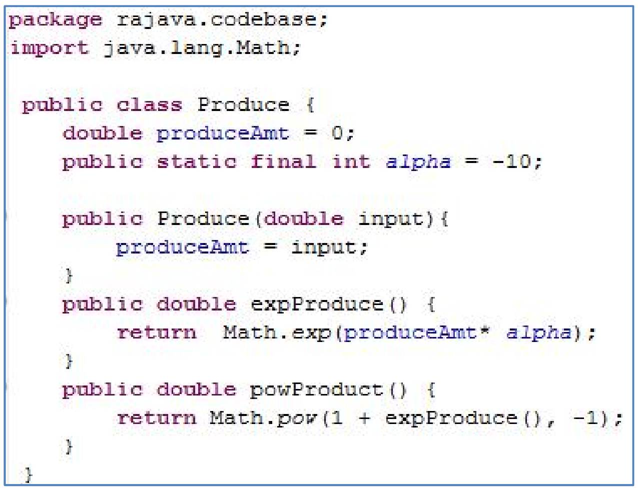
Given the number of "knots" as k-1, the total number of terms in the unified function will be 2k + 1. When the Heaviside function is converted to constraint optimization problem in CoReJava framework, it requires the function to be optimized is differentiable. A differentiable approximation of the Heaviside function is called sigmoid function, defined as:

$$\sigma(x) = \frac{1}{1+e^{-ax}} \quad (13)$$

Correspondingly equation (11) will be rewritten as:

$$f(p,x) = f_1(p_1,x) - f_1(p_1,x) * \sigma(x-b_1) + f_2(p_2,x) * \sigma(x-b_1) - f_2(p_2,x) * \sigma(x-b_2) + \cdots + f_k(p_k,x) * \sigma(x-b_{k-1}) \tag{14}$$

The variable α is a predefined constant which has been assigned as 10 in our experiments. However, the value of α can be changed for different experiment set-ups. The sigmoid function is implemented as a Java class `Produce` which is depicted in Fig. 7.



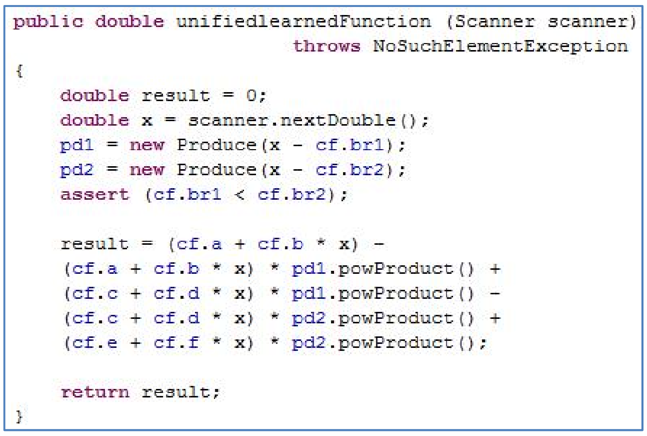
```
package rajava.codebase;
import java.lang.Math;

 public class Produce {
    double produceAmt = 0;
    public static final int alpha = -10;

    public Produce(double input){
        produceAmt = input;
    }
    public double expProduce() {
        return  Math.exp(produceAmt* alpha);
    }
    public double powProduct() {
        return Math.pow(1 + expProduce(), -1);
    }
 }
```

Figure 7: Class representation of sigmoid function

The `learnedFunction` is implemented a different way in the Heaviside restructuring. We do not need binary variables to represent the piecewise selection any more since nested "if" and "else" statements have been replaced by a single assignment statement. This approach will decrease the searching complexity from $O(2^{N \cdot M})$ where N is the number of data inputs and M is the number of "if" statements in the `learnedFunction`, to $O(N^{P+K})$ where P is the number of parameters and K-1 is the number of knots.



```
public double unifiedlearnedFunction (Scanner scanner)
                          throws NoSuchElementException
{
    double result = 0;
    double x = scanner.nextDouble();
    pd1 = new Produce(x - cf.br1);
    pd2 = new Produce(x - cf.br2);
    assert (cf.br1 < cf.br2);

    result = (cf.a + cf.b * x) -
    (cf.a + cf.b * x) * pd1.powProduct() +
    (cf.c + cf.d * x) * pd1.powProduct() -
    (cf.c + cf.d * x) * pd2.powProduct() +
    (cf.e + cf.f * x) * pd2.powProduct();

    return result;
}
```

Figure 8: Code Snippet for unifedlearnedFunction

Claim: the Heaviside restructuring algorithm decreases the searching complexity to polynomial degree while its learning outcome approximates optimal solution.

V. Experimental Study

The experiment is designed to compare different approaches of regression learning – generic CoReJava regression learning, combinatorial restructuring and Heaviside restructuring. Two matrices are adopted for evaluation. One is the execution time (in milliseconds), and the other is RMS (root mean square error) [10]. In our case, RMS error is equal to the sum of squared error differences between the observed dependent variable Y and the value $\hat{Y}$ which is returned from the `learnedFunction`. The quality of regression learning is evaluated by the RMS error. The lower the RMS errors, the better the learning quality.

To construct the training data sets, we generate four piecewise linear functions f1, f2, f3, and f4, which have 1, 2, 3 and 4 knots correspondingly. The number of sample data points generated for f1, f2, f3 and f4 are 50, 75, 100 and 125. For each function fi, sample data points (x_i, y_i) $i = 1,2,..,N$ according to the model $y = fi(x) + \varepsilon$, being ε a normal random variable with zero mean and standard deviation equal to 0.5. The experiments are run in windows vista with 1.60GHZ processor and 3GB memory. The supporting software is Java 6, AMPL/SNOPT and Eclipse 3.5.

Table I compares the execution time among different approaches. We can observe that for generic CoReJava regression learning, where non-deterministic knot variables are involved in the "if" condition, its execution time increase dramatically as the number of knots increase, from 456 ms for one knot to 13002 ms for four knots. However, the execution time for single running of combinatorial restructuring is trivial, around 10 ms compared to that of generic learning in CoReJava framework. Although the single running of combinatorial restructuring takes trivial time, the exhaustive search strategy for the combinatorial restructuring makes the situation worse. As the number of knots increase, all the possibilities of knots combination from the input data set increases dramatically as well. When the number of knots reaches four, it shows that the total running time for combinatorial restructuring reaches a substantial amount of time, around 10^8 milliseconds. This makes the combinatorial restructuring a less competitive strategy. In the view of execution time, Heaviside restructuring is a promising method in that it takes a little bit longer than a single run of combinatorial restructuring.

Table I: Execution Time in Milliseconds for Different Approaches

Approach / Number of knots	Generic CoReJava Regression Learning	Combinatorial restructuring		Heaviside Restructuring
		Each Run	No. of Runs	
K = 1 (50 data points)	456	9	49	14
K = 2 (75 data points)	1012	10	76 * 75 / 2=2850	14
K = 3 (100 data points)	4006	10	101 * 100 * 99 / 3 *2=166650	14
K = 4 (125 data points)	13002	10	126*125*124*123/4*3*2*1≈ 10^7	15

Table II summarizes the RMS error returned by each approach. We can observe for the generic CoReJava regression, the learning qualify is very poor with regarding to the RMS errors. For function f3 and f4, the RMS errors are 2544.346 and 15578.966, which actually means that feasible solution cannot be found by generic CoReJava regression learning due to the limitation of external optimization solver. However, the combinatorial restructuring guarantees optimality and locates the optimal feasible solution by minimizing the RMS error for piecewise linear functions, f1, f2, f3 and f4. That is why we prefer the combinatorial restructuring for optimal feasible solution by sacrificing the amount of execution time. RMS error for Heaviside restructuring is a little bit above the combinatorial restructuring but outperforms the generic CoRejava regression learning.

TABLE II: RMS ERROR COMPARISON AMONG DIFFERENT APPROACHES

Approach / Number of knots	Combinatorial Restructuring	Generic CoReJava Regression Learning	Heaviside Restructuring
K = 1 (50 data points)	13.947	366.826	18.227
K = 2 (75 data points)	17.940	758.216	21.904
K = 3(100 data points)	19.862	2544.346	23.647
K = 4 (125 data points)	30.317	15578.966	34.879

As the size of data set scales up, the execution time for combinatorial restructuring becomes very expensive. Combining both matrices (time and RMS), Heaviside restructuring is an efficient and applicable method for the piecewise regression learning problems.

VI. CONCLUSION AND FUTURE WORK

Spline functions (piecewise linear regression problems) are depicted as Java programs in CoReJava framework. It is initially solved by the generic regression learning in CoReJava framework. Due to exponentially increased searching complexity and limitation of external optimization solver, we propose a combinatorial restructuring which decreases the complexity of learning, at the same time guarantees the optimality. However, the exhaustive search strategy for the combinatorial restructuring makes the execution expensive. Heaviside restructuring furthermore decreases the searching complexity of learning to polynomial of the size of learning set and takes a little more execution time than a single run of combinatorial restructuring. It can't achieve the optimal solution due to the fact that it is the differentiable approximation of the piecewise functions. However, the RMS for Heaviside restructuring is close to the value of combinatorial restructuring.

Many research questions remain open. They include (1) decreasing the number of combinations of knots heuristically; (2) extending single-dimensional piecewise linear model to general higher-dimensional piecewise surface regression by clustering and classification; (3) adjusting the sigmoid function to achieve the better approximation of piecewise functions; and (4) special-purpose hybrid optimization algorithms suitable for learning that originates from a simulation process described as an OO program.

REFERENCES

[1] D. Montgometry, E. Peck and G. Vining, Introduciton to Linear Regression Analysis, 4th ed**.**, John Wiley & Sons Inc**,** 2007

[2] A. Brodsky, J. Luo and H. Nash, “CoReJava: Learning Functions Expressed as Object-Oriented Programs,” icmla, pp.368-375, Seventh International Conference on Machine Learning and Applications, 2008

[3] Wahba, G., “Spline models for observatinal data”, Philadelphia, SIAM, 1990

[4] Cleveland, S., “Robust locally weighted regresison and smoothing scatterplots”, Journal of the American Statistical Association, 74: pp 829-36, 1979.

[5] Ferrari-Trecate, G and Muselli M., “A New Learning Method for Piecewise Linear Regression”, Lecture Notes In Computer Science; Vol. 2415, pp 444-449, Proceedings of the International Conference on ANN, 2002.

[6] Arumugam, M and Scott, S,. “EMPRR: A High-Dimensional EM-Based Piecewise Regression Algorithm”. In Proceedings of The 2004 International Conference on Machine Learning and Applications (ICMLA '04), pages 264-271, Louisville, Kentucky, December 2004

[7] Bishop, C. Pattern Recognition and Machine Learning, Springer, (2006)

[8] http://www.ampl.com

[9] E. Alpaydin, Introduction to Machine Learning, MIT Press, 2004