Data Science with Linear Programming

Nantia Makrynioti[1, 2] **Nikolaos Vasiloglou**[1] **Emir Pasalic**[1] **Vasilis Vassalos**[2]

[1]LogicBlox
{nantia.makrynioti, nikolaos.vasiloglou, emir.pasalic}@logicblox.com
[2]Department of Informatics, Athens University of Economics and Business
{vassalos}@aueb.gr

Abstract

The standard process of data science tasks is to prepare features inside a database, export them as a denormalized data frame and then apply machine learning algorithms. This process is not optimal for two reasons. First, it requires denormalization of the database that can convert a small data problem into a big data problem. The second problem is that it assumes that the machine learning algorithm is disentangled from the relational model of the problem. That seems to be a serious limitation since the relational model contains very valuable domain expertise. In this paper we explore the use of convex optimization and specifically linear programming as a data science tool that can express most of the common machine learning algorithms and at the same time it can be natively integrated inside a declarative database. We are using SolverBlox, a framework that accepts as an input Datalog code and feeds it into a linear programming solver. We demonstrate the expression of three common machine learning algorithms, Linear Regression, Factorization Machines and Spectral Clustering, and present use case scenarios where data processing and modelling of optimization problems can be done step by step inside the database.

1 Introduction

As data science becomes more and more prevalent in the industry, mathematical modelling languages, such as R and Matlab, remain popular, but users also seek other solutions, which will provide a declarative framework for defining machine learning algorithms, as well as allow them to work on data stored in relational databases. In the following sections, we describe how the "say what you want to do and not how to do it" concept of declarative paradigm can be applied to data science problems via linear programming and LogiQL (an extended version of Datalog used in the LogicBlox database [Aref *et al.*, 2015]).

Recent systems for large-scale declarative machine learning, e.g. SystemML [Ghoting *et al.*, 2011], Mahout Samsara [Lyubimov and Palumbo, 2016] and TensorFlow [Abadi *et al.*, 2016], focus on supporting a number of linear algebra operators, which are common in building machine learning models, and techniques for optimizing plans consisting of these operators. However, in real world problems data is not given as a matrix or tensor, but as relational tables. These systems ignore the relational nature of data and require conversion to matrices. Apart the tedious process of exporting/importing data between a database and a machine learning system, denormalization also results in losing important domain information embedded in the relational representation. Moreover, the computation of the optimal parameters of the model should still be described and implemented by the user, which diverges from the concept of declarative programming. By defining machine learning models inside a database with a declarative language, such as LogiQL, casting them as linear programs and then delegating their solution to an appropriate solver, the user needs only to define the model and the objective function, as well as any constraints that may be useful to the task. Moreover, the machine learning algorithm workflow can include database queries. For example the user might want to apply different regularization to data points that satisfy a complicated data constraint, which is not possible to compute once the data is flattened.

We argue that linear programming is a convenient interface of non-probabilistic machine learning to logical programming. At a syntax level possible nonlinearities could be expressed either with language directives or with logical predicates, which a compiler will be able to rewrite to mathematical expressions that can be passed to a lower level solver. For example several nonlinearities can be automatically relaxed with rewritings and be processed by branch and bound solvers or they can be converted to nonconvex equivalents. To unify linear programming with logical programming, we use the SolverBlox framework and demonstrate that we can express different classes of machine learning problems, which can then be exported to the exact same format (.lp Gurobi) and be handled by external solvers. SolverBlox can also be extended to support more classes of optimization problems, such as quadratic or SAT problems, which would eliminate the need for rewritings.

The contributions of the paper are summarized below:

- A suitable interface for expressing deterministic machine learning algorithms on relational data, where the user is able to define her model in a declarative manner

and get the solution by the system.

- A unified framework that combines data processing with expressing machine learning tasks, which allows users to build different models more easily, resulting in a more interactive way for doing data science.
- Implementation of three machine learning algorithms as linear programs and experiments with two use case scenarios, which demonstrate the ability to create more stable models using constraints.

2 Related Work

Large-scale systems for declarative machine learning, such as SystemML and Tensorflow, and big data frameworks, like Spark, require denormalized data, which have already been pre-joined to a universal relation, in order to be stored in matrices. Our work on expressing machine learning models with LogiQL and SolverBlox applies on relational data and therefore it is closer to languages that aim at expressing optimization problems in the relational context. RELOOP [Mladenov *et al.*, 2016], LBJava [Rizzolo and Roth, 2007], WOLFE [Singh *et al.*, 2015] and Saul [Kordjamshidi *et al.*, 2015] introduce domain-specific languages embedded in imperative languages, such as Java and Python, which enable users to express relational optimization problems. In these systems, relational data are either queried via common APIs or by using a set of abstractions that represent the relational data model. To a similar direction, the work by Kersting et al. [Kersting *et al.*, 2017] introduces a logical programming variant of AMPL, which parametrizes arithmetic expressions with logical variables and enables to index expressions by querying a knowledge base. By modelling the optimization problem and then sending it to a solver, both these papers and our work follow the declarative paradigm of "model+solver", which is prominent in machine learning [Geffner, 2014], [Sra *et al.*, 2011]. Linear programming has been proven to be very powerful since it can express deep learning networks [Bastani *et al.*, 2016].

The main difference between the aforementioned work and our approach is that the systems above are still separate from the relational database engine, which means that the latter is not aware of the optimization process. Using SolverBlox, we demonstrate that the user can express optimization problems where the data lives and take advantage of LogiQL incremental evaluation. As a result each time data change in the LogicBlox database, the grounded instance matrix is not recomputed from scratch, but only the appropriate elements are modified. The solver is also triggered automatically to provide a new solution based on the updated data.

3 SolverBlox and LogiQL

LogiQL is a declarative language derived from Datalog [Maier and Warren, 1988], a deductive database query language [Abiteboul *et al.*, 1995]. In LogiQL, rules based on first order logic are used to specify declaratively *what* the computation should produce (i.e., the logical structure of a program's result), rather than step-by step algorithmic details of how to compute it. A further level of abstraction, SolverBlox, is a framework for expressing linear and mixed integer programs with LogiQL.

With SolverBlox, the user can define predicates in LogiQL to represent the objective function and the constraints of the linear program, as well as other business logic. The SolverBlox framework goes beyond the usual semantics of Datalog by allowing the programmer access to a *second order existential quantifier* [Mancini, 2004] (the quantifier is second order because it quantifies not over primitive values, but over predicates): for a predicate `x[i, j]` which corresponds to the (indexed) variable x_{ij} in a mathematical programming problem, the programmer need not specify a set of rules that describe its computation in LogiQL. Rather, she asserts that a predicate `x[i, j]=v` exists such that it (a) maximizes the objective function and (b) satisfies all the model constraints, expressed as a set of *integrity constraints* [Ramakrishnan and Ullman, 1995] over `x`. It is at that point up to the SolverBlox implementation to find a predicate that satisfies those conditions.

Under the hood, the SolverBlox implementation creates a collection of LogiQL predicates that represent the typical triple of a vector c, a matrix A and a vector b used in a linear programming instance, as well as rules for populating these predicates.

```
matrix[row,col]=v -> int(row), int(col), float(v).
bound[row]=v -> int(row), float(row).
objective[col]=v -> int(col), float(v).
solution[col]=v -> int(col), float(v).
```

This is a compile-rewrite step that results in a pure LogiQL program P', which has the intended semantics of finding suitable values for predicates marked with `lang:solver:variable` such that the objective function is maximized. The program P' also contains a call to a foreign function interface that at runtime marshals the contents of the predicates to an external solver, invokes the solver, and populates the predicate that stores the solution.

The process of going from a constraint based specification in LogiQL (as in our examples below) to the concrete problem instance (c,A,b) is called grounding. The automatic synthesis of a pure LogiQL program that translates constraints over variable predicates into a representation that can be consumed by the solver is highly benefited by the LogiQL evaluation engine in terms of query optimization and parallelization. Moreover, when the data in the LB database change, the program P' is automatically re-executed by the system (part of standard LogiQL semantics) to update the instance (predicates matrix, bound, etc.), and re-invoke the external solver, updating the solution predicate. It's important to note that these updates are evaluated incrementally, which means that the grounding logic modifies only the parts of the instance matrix that are affected by the changes of the input. The present system has the capacity to execute this process for any linear and mixed-integer programming problem expressed with the syntax described in section 4. More details of the process of grounding in SolverBlox can be found in [Aref *et al.*, 2015].

4 Machine Learning Algorithms in SolverBlox

In this section we describe how we implement a number of machine learning algorithms as linear programs using SolverBlox. We include algorithms for two tasks, regression and clustering.

4.1 Linear Regression

When the objective function is the least absolute error, L_1 Linear Regression can be naturally expressed as a linear program, as displayed below.

$$\min \sum_{i=1}^{n} \varepsilon_i \quad subject\ to\ -\varepsilon_i \leq (\hat{y}_i - y_i) \leq \varepsilon_i \tag{1}$$

On the other hand, L_2 Linear Regression that minimizes least square error cannot be solved with linear programming, as the objective function is quadratic. An approximation of L_2 Linear Regression can be defined as a linear program by multiplying the absolute error with an estimation of the error, as it is displayed below.

$$error = \sum_{i=1}^{n} w_i |\hat{y}_i - y_i| \tag{2}$$

where w_i is the estimation of the error, e.g. the average error from previous iterations of L_1 Regression or just the error of the previous iteration. The equivalent linear program of this approximative algorithm is the following:

$$\min \sum_{i=1}^{n} \varepsilon_i \quad subject\ to\ -\varepsilon_i \leq w_i(\hat{y}_i - y_i) \leq \varepsilon_i \tag{3}$$

We now present how we express and solve these linear programs with LogiQL and SolverBlox. We showcase our implementation of Linear Regression on a data science problem from the retail domain, where we aim to predict the future demand of a SKU (stock-keeping unit) based on historical sales. Predictions are generated for each SKU, at each store and on each date of the forecast horizon.

The main components of SolverBlox programs are extensional predicates (EDB), intensional predicates (IDB) and language pragmas. An EDB predicate stores values of the extensional database, that is, values that the user explicitly imported into the database. On the other hand, the values of an IDB predicate are computed via IDB rules, i.e. logical implications that specify what values the predicate should contain, based on the values of other EDB or IDB predicates.

In the code snippet below, we first define the types and arity of a number of predicates. Apart from primitive types, one can also define and use entities as types, similar to classes in Java or structs in C. `sku`, `store` and `day` are all entity predicates.

```
observables(sku,str,day) -> sku(sku), store(str), day(day).

sku_coeff[sku]=v -> sku(sku), float(v).
brand_coeff[br]=v -> string(br), float(v).

Prediction[sku, str, day] = v -> sku(sku), store(str), day(day),
      float(v).
```

We then move on to defining IDB predicates for generating dot products between features and coefficients, as well as sums of these products and predictions for every sku-store-day combination in the input data. In this case, features are binary so dot products are basically equal to the value of the corresponding coefficient. Values of dot products and sums are indexed based on whether they concern SKU features, store features, day features or a combination of these. The `error` predicate stores the difference between the target value and the prediction, whereas `totalError` is the sum of all individual errors. To denote that `totalError` is the objective function which we want to minimize, we use the language annotation `lang:solver:minimal('totalError).` Variables of the linear program are defined by using the pragma `lang:solver:variable('name_of_predicate).` The absolute value of the error is expressed with two constraints that bound the error between two values. In SolverBlox, constraints are rules which consist of a conjunction of predicates on the left side and a mathematical expression between LogiQL variables (here we use the term "variable" to denote a LogiQL variable, not a variable of the linear program) on the right side.

```
sku_coeffF[sku]=v <- unique_skus(sku), sku_coeff[sku]=v.

brand_coeffF[sku]=v <- brand_coeff[br]=v, unique_skus(sku), brand[
      sku]=br.

store_coeffF[str]=v <- unique_stores(str), store_coeff[str]=v.

sum_of_sku_features[sku]=v <- unique_skus(sku), sku_coeffF[sku]=v1,
       brand_coeffF[sku]=v2, brandType_coeffF[sku]=v3, codeUB_coeffF
      [sku]=v4, subfamily_coeffF[sku]=v5, v=v1+v2+v3+v4+v5.

 Target[sku, str, day] = v <-
      observables(sku,str,day),
      sum_of_sku_features[sku]=z1,
      sum_of_store_features[str]=z2,
      sum_of_day_features[day]=z3,
     sum_of_sku_store_day_features[sku,str,day]=z4,
      v=z1+z2+z3+z4.

//IDB predicate of error between prediction and actual value
error[sku, str, day] += Target[sku, str, day] - total_sales[sku,
      str, day].

//IDB predicate of objective function
totalError[] += abserror[sku, str, day] <- observables(sku, str,
      day).
lang:solver:minimal('totalError).

//Constraints that bound error between two values
observables(sku, str, day),
        abserror[sku, str, day]=v1,
    error[sku, str, day]=v2
    ->
    v1>=v2.

    observables(sku, str, day),
     abserror[sku, str, day]=v1,
    error[sku, str, day]=v2,
    w=0.0f-v2
      ->
    v1>=w.
```

The code will then be translated to a format supported by a solver, e.g the Gurobi linear programming solver [1]. The solver will compute the solution of the problem and send it back to SolverBlox, which will populate the corresponding predicates. As a result, the user will be able to print the values of the model parameters or access parts of the solution via

[1] http://www.gurobi.com/products/gurobi-optimizer

queries in the same way as with any relation in the LogicBlox database.

4.2 Factorization Machines

In this section we implement Factorization Machines [Rendle, 2010], another regression algorithm, whose model function involves interactions between features. An important element of this algorithm is that the coefficient of an interaction is computed as a product between two vectors, which makes for a non-linear function.

$$\begin{gathered} y = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{m} < v_i, v_j > x_i \cdot x_j \\ where\ n = number\ of\ features, \\ k = size\ of\ v, \\ w_0 \in R,\ w \in R^n,\ V \in R^{nxk}\ and \\ < v_i, v_j >= \sum_{f=1}^{k} v_{i,f} \cdot v_{j,f} \end{gathered} \tag{4}$$

A simple way to create a linear model with feature interactions is to introduce a single weight for each one of them, as it is done with features. However, this would increase the parameters of the model considerably, especially with large amounts of training data.

Below, we present a linear approximation of the initial model that needs a reduced number of parameters for interactions. We start by assigning each interaction to a fixed number of buckets based on one or more hash functions. Then, we only need to learn weights for the buckets, whose number should be smaller than the number of interactions. The idea is that similar interactions would be assigned to the same bucket and as a result they would have the same weight. As this approach can result in conflicts, i.e. interactions that involve totally different features may be placed in the same bucket, we can use k hash functions and take the sum of bucket weights to compute the final coefficient of an interaction. The rewritten model function would be the following:

$$y = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{l=1}^{k} \sum_{i=1}^{n} \sum_{j=i+1}^{m} bc_l[i,j] \cdot x_i x_j \tag{5}$$

where $bc_l[i,j]$ is the bucket coefficient of the l hash function, for the interaction between features i and j.

Another approach to circumvent the non-linearity of the dot product between interaction coefficients in the original algorithm is to keep one of the vectors fixed and solve for the other. In this case, we keep alternating between vectors at each iteration of the optimization process, so that another one becomes an unknown variable and the previous one is inserted to the set of known predicates. The process is repeated until the LP is solved for every vector. Below we present the implementation of the first method, which uses hashing. Currently SolverBlox does not support dynamic modification of solver variables, so the method that alternates between vectors would be inefficient.

Building on our previous example for product demand, let's say that we want to add an interaction between a SKU and a month. This is a useful interaction for seasonal products, which can sell more during summer or winter. Predicate `sku_monthOfYear_bucket` stores the number of the bucket where each combination of sku-month has been placed (in this example we use 100 buckets). The next predicate, `sku_monthOfYear_interaction`, defines that each interaction will be multiplied with the coefficient of its bucket. In case of binary features, the product between an interaction and a coefficient is equal to the value of the coefficient.

```
sku_monthOfYear_bucket[sku, moy] = v <- observables(sku,_,day),
      monthOfYear[day]=moy, sku_id[sku]=n1, month_id[moy]=n2, n=n1+
      n2, string:hash[n]=z, int:mod[z, 100]=v.

sku_monthOfYear_interaction[sku, day]=v <- observables(sku,_,day),
      monthOfYear[day]=moy, sku_monthOfYear_bucket[sku, moy]=z3,
      bucket_coeff[z3]=v.
```

Using absolute error as the objective function, the rest of the linear program is similar to the one described for Linear Regression.

4.3 Clustering

The last use case we present regards clustering tasks. K-means is probably the most popular clustering algorithm, but due to the euclidean norm it is difficult to rewrite it as a linear program. So, instead of finding centroids, we use another objective function, whose purpose is to minimize intra-cluster distances, i.e. distances between data points of a cluster [2]. By doing so, we can rewrite clustering as an integer program using the following objective function:

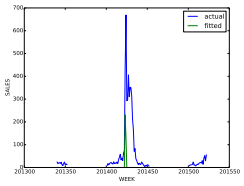
$$\min \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=1}^{m} x_{k,i} * x_{k,j} * d_{i,j},\ x \in \{0,1\} \tag{6}$$

If two points belong to the same group, then the variable x for both i and j will be equal to 1 and the distance between the two data points will be added to the total cost of the group. On the other hand if two points are in different clusters, at least one of the two x variables will be zero and the distance will not be added to the total cost. This objective function is still non linear, as we have introduced a product between two unknown variables, i.e. x variables. We can observe that the result of the product between x variables is the same with the one produced by the boolean AND operator. Therefore, we can replace it with a single predicate and generate the same result using constraints, as shown below.

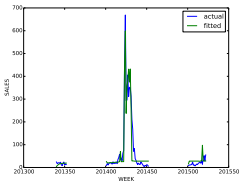
$$\begin{gathered} \min \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=1}^{m} y_{i,j,k} * d_{i,j},\ y \in \{0,1\} \\ y_{i,j,k} \geq x_{i,k} + x_{j,k} - 1 \\ y_{i,j,k} \leq x_{i,k} \\ y_{i,j,k} \leq x_{j,k} \end{gathered} \tag{7}$$

The implementation of this linear program in SolverBlox is as follows:

[2] This approach is very similar to spectral clustering. The problem can be relaxed as an LP and it can be significantly optimized if we only use nearest neighbor distances instead of all distances

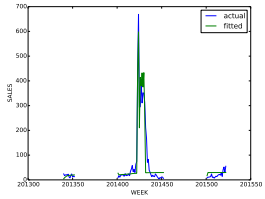


(a) LR

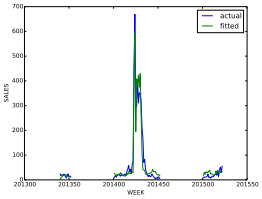


(b) LR-L1Reg-biasConst

Figure 1: Fit of SKU 9 - 1st and 2nd model



(a) LR-L1Reg-softBiasConst-predConst



(b) FM-L1Reg-softBiasConst-predConst

Figure 2: Fit of SKU 9 - 3rd and 4th model

```
lang:solver:variable(`instance_to_group).
lang:solver:variable(`y).

//The objective function is defined as the sum of distances
      multiplied by a binary variable denoting whether two points
      belong to the same cluster.
total_grouping[] += y[k, x1, x2]*squared_dist[x1, x2] <- INSTANCE(
      x1), INSTANCE(x2), GROUP(k).
lang:solver:minimal(`total_grouping).

//Constraints
//Variable will be binary, takes only 0 or 1
INSTANCE(x), GROUP(k), instance_to_group[k, x]=v -> bloxopt_binary(
      v).

INSTANCE(x1), INSTANCE(x2), GROUP(k), y[k, x1, x2]=v,
      instance_to_group[k, x1]=z1, instance_to_group[k, x2]=z2, w=z1
      +z2-1.0f -> v>=w.
INSTANCE(x1), INSTANCE(x2), GROUP(k), y[k, x1, x2]=v,
      instance_to_group[k, x1]=z1 -> v<=z1.
INSTANCE(x1), INSTANCE(x2), GROUP(k), y[k, x1, x2]=v,
      instance_to_group[k, x2]=z1 -> v<=z1.
INSTANCE(x1), INSTANCE(x2), GROUP(k), y[k, x1, x2]=v -> v>=0.
```

Data points are stored in the predicate `INSTANCE` and the distance between them is computed based on their feature values and is stored in the predicate `squared_dist`. The membership between each cluster and each data point as well as the assignment of two data points in the same cluster designate the LP variables. Apart from the constraints needed to model the clustering algorithm as an LP problem, we can also set additional constraints to specify that each data point belongs to a single cluster or that each cluster must have at least one member.

5 Experiments

5.1 Regression

We first present experiments with the Linear Regression implementation described above. The data consist of historical sales from five SKUs and are divided into training (136509 observations) and test set (9100 observations) based on two separate time periods. The quality of results is measured in terms of WAPE (Weighted Absolute Percent Error) and bias. The feature set includes similar features to the ones mentioned in section 4.1, such as the brand of the SKU or the month when a sale took place. We present results from four models, which we will describe step by step. All results are displayed in Table 1, whereas Figures 1 and 2 show the fit curve of training predictions from SKU #9 (gaps between lines correspond to periods with no sales observations).

The first model is a linear regression model with the aforementioned features (LR). We observe that WAPE for all SKUs on training data is close to 100 with the bias being negative and also close to 100. This behaviour can be explained by the nature of the data, which contain a lot of observations with zero sales, forcing the solver to push coefficients to zero. Although WAPE is minimized, minimization of bias is not considered, resulting in a useless model. The user can easily make the solver take bias into account by adding a constraint to the linear program that requires bias for every SKU aggregated over all stores and days to be equal to zero. However, the addition of the bias constraint creates a colinearity issue, which becomes evident by the inconsistencies in WAPE of test data between runs of the same linear program and the large values of coefficients. To address this issue we move on to adding L_1 regularization to the model (LR-L1Reg-biasConst). Table 1 shows the improvement of WAPE and bias metrics for all SKUs, which is also shown by the fit in Figure 1b. We omit bias values on training data for this model, as they are all zero due to the constraint.

We can also replace the bias constraint with a softer one, that restricts SKU bias between two parameters, whose sum of values is added as a second regularization term to the objective function (LR-L1Reg-softBiasConst-predConst). Softer constraints allow the user to restrict more variables of the problem, such as defining that sales predictions must be greater or equal to zero by adding the appropriate constraint. These modifications help the model even further as WAPE for two of the SKUs is reduced, whereas bias on test data is highly improved for all SKUs as displayed in section "LR-L1Reg-softBiasConst-predConst" of Table 1. By adding some interactions between features, we can create a Factorization Machines model (FM-L1Reg-softBiasConst-predConst). Results show that interactions between SKU and date features, as well as SKU and promotion features have a positive effect. In general data science is all about adding more degrees of freedom (interactions) and at the same time guiding the solver with constraints to solutions that make more sense. For example we can also add a bias constraint per store aggregated over all SKUs and days, which is usually a client demand. It is well known that in machine learning there are several models that can minimize the objective but not all of them generalize well, or are interpretable. Adding more constraints leads to better models.

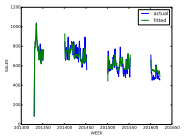
Aggregated forecasting

In many problems users are interested in forecasts at an aggregated level. For example retailers many times prefer to monitor the total sales of a family of products at every store per day, including sales from every SKU of that family. In-

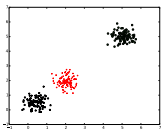
Table 1: Regression experiments on 5 SKUs (ids: 8, 3, 9, 6, 26)

Model	SKU id	8	3	9	6	26
LR	WAPE on training	99.99	99.99	93.43	99.97	99.99
	BIAS on training	-99.99	-99.99	-93.43	-99.97	-99.99
	WAPE on test	99.99	99.62	88.16	97.86	99.99
	BIAS on test	-99.99	-80.68	-88.16	-84.45	-99.99
LR-L1Reg-biasConst	WAPE on training	95.7	67.73	36.26	62.77	102.6
	WAPE on test	67.07	111.26	49.78	106.9	86.46
	BIAS on test	-36.54	90.33	-1.6	68.68	3.47
LR-L1Reg-softBiasConst-predConst	WAPE on training	98.07	63.74	34.44	59.99	104.8
	BIAS on training	-1.96	0	0	0	0
	WAPE on test	71.56	75.33	50.89	73.29	87.15
	BIAS on test	-36.92	5.99	-0.5	0.97	6.13
FM-L1Reg-softBiasConst-predConst	WAPE on training	94.09	62.35	31.28	58.71	99.07
	BIAS on training	-2.75	0.01	0.004	0.007	0.13
	WAPE on test	70.77	71.02	47.18	70.07	80.82
	BIAS on test	-36.74	8.03	-0.04	3.44	6.21

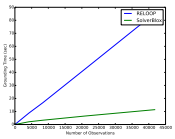
stead of generating sales at SKU-Store-Day level and then aggregating over all SKUs of a particular family, by expressing models as linear programs it's very easy to generate predictions at the Subfamily-Store-Day level directly and as a result decrease significantly computation time in case of large datasets. We built a Factorization Machines model with similar features as before and ran it on a much larger dataset of 2033354 observations. The model now generates 60346 forecasts at the Subfamily-Store-Day level regarding a single family of SKUs. Figure 3a demonstrates that aggregated forecasts fit actual sales well.



(a) Fit of aggregated forecasts for a family of products



(b) Clustering of 300 2-d points



(c) Grounding time for RELOOP and SolverBlox

Figure 3: Aggregated forecasting, clustering and time performance

5.2 Clustering

In this section we present an experiment with the SolverBlox clustering implementation. We use a dataset of 300 2-dimensional data points, which have been generated by three Gaussian distributions. The result of the clustering is displayed in Figure 3b. We can observe that points which have been generated by the same Gaussian distribution are also assigned to the same cluster. Since the membership of a point to a cluster is a binary variable, this is a Mixed Integer Programming problem. Using a LP relaxation to make this problem easier, results in membership values being equal to 0.5 for the majority of the points. As a result, these values do not help us determine where to assign each data point by rounding.

5.3 Time Performance

Finally, we present an experiment between SolverBlox and RELOOP [Mladenov *et al.*, 2016] that compares the time needed for grounding in both frameworks. We used a publicly available dataset that predicts the age of an abalone from physical measurements [3]. The dataset contains 4177 observations, but we replicated those in order to create larger datasets of 2, 5 and 10 times the original size. For RELOOP we used the Block Grounder. Figure 3c shows the time needed for both systems to generate the grounding extension of the linear program. We observe that in SolverBlox grounding time increases much slower than in RELOOP.

6 Discussion

In this paper we demonstrated the value of blending machine learning algorithms and relational databases in order to accelerate and improve data science tasks. Although the experiments presented in this paper were done on small scale datasets, we have applied them to larger datasets (millions of records and tenths of attributes) from major retailers. Despite the fact that most of the grounding is happening inside the database, it is possible that the LP matrices fed to the solver can be very big. This is something that Kersting et al [Kersting *et al.*, 2017] has also addressed with specialized lifting techniques. This technique compresses the LP generated matrix. Recent advances in the database technology [Abo Khamis *et al.*, 2016] have indicated that it is possible to exploit functional dependencies between features and accelerate grounding and significantly compress the LP matrix.

Another important aspect of integrating optimization in a declarative database is extending to higher order convex programming, like Quadradic and Semidefinite programming. Integration of higher order solvers can extend the expressivity of machine learning algorithms but it is not trivial.

[3] https://archive.ics.uci.edu/ml/datasets/abalone

References

[Abadi *et al.*, 2016] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[Abiteboul *et al.*, 1995] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Abo Khamis *et al.*, 2016] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 13–28, New York, NY, USA, 2016. ACM.

[Aref *et al.*, 2015] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1371–1382, New York, NY, USA, 2015. ACM.

[Bastani *et al.*, 2016] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2613–2621, 2016.

[Geffner, 2014] Hector Geffner. Artificial intelligence: From programs to solvers. *AI Commun.*, 27(1):45–51, 2014.

[Ghoting *et al.*, 2011] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.

[Kersting *et al.*, 2017] Kristian Kersting, Martin Mladenov, and Pavel Tokmakov. Relational linear programming. *Artif. Intell.*, 244:188–216, 2017.

[Kordjamshidi *et al.*, 2015] Parisa Kordjamshidi, Dan Roth, and Hao Wu. Saul: Towards declarative learning based programming. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1844–1851, 2015.

[Lyubimov and Palumbo, 2016] Dmitriy Lyubimov and Andrew Palumbo. *Apache Mahout: Beyond MapReduce*. CreateSpace Independent Publishing Platform, USA, 1st edition, 2016.

[Maier and Warren, 1988] David Maier and David S. Warren. *Computing with Logic: Logic Programming with Prolog*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1988.

[Mancini, 2004] Toni Mancini. *Declarative constraint modelling and specification-level reasoning*. PhD thesis, Universita' degli Studi di Roma "La Sapienza", 2004.

[Mladenov *et al.*, 2016] Martin Mladenov, Danny Heinrich, Leonard Kleinhans, Felix Gonsior, and Kristian Kersting. RELOOP: A python-embedded declarative language for relational optimization. In *Declarative Learning Based Programming, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 13, 2016.*, 2016.

[Ramakrishnan and Ullman, 1995] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23(2):125 – 149, 1995.

[Rendle, 2010] Steffen Rendle. Factorization machines. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 995–1000, Washington, DC, USA, 2010. IEEE Computer Society.

[Rizzolo and Roth, 2007] Nicholas Rizzolo and Dan Roth. Modeling discriminative global inference. In *Proceedings of the First IEEE International Conference on Semantic Computing (ICSC 2007), September 17-19, 2007, Irvine, California, USA*, pages 597–604, 2007.

[Singh *et al.*, 2015] Sameer Singh, Tim Rocktäschel, Luke Hewitt, Jason Naradowsky, and Sebastian Riedel. WOLFE: an nlp-friendly declarative machine learning stack. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 61–65, 2015.

[Sra *et al.*, 2011] Suvrit Sra, Sebastian Nowozin, and Stephen J. Wright. *Optimization for Machine Learning*. The MIT Press, 2011.