6.830 Term Project - Spring 2013

Machine Learning Algorithms for In-Database Analytics

Franck Dernoncourt & Sumaiya Nazeen

May 16, 2013

Abstract

Our project focused on extending the functionality of MADlib. MADlib is an open source machine learning and statistics library which works with Postgres or Greenplum to provide in-database analytics. Although some machine learning algorithms have been implemented in MADlib, there is room for additional contributions. We have implemented two different machine learning algorithms, symbolic regression with genetic programming and adaptive boosting for MADlib, and are in the process of contributing our code to the MADlib community codebase. We have also assessed the performance of our implementations and compared their performance with the same algorithms outside MADlib.

1 Introduction

Traditionally, large databases were mainly used for *data warehousing* i.e. accounting purposes in enterprises, supporting financial record-keeping and reporting at various levels of granularity. However, over the past decade, attitudes toward large databases have been changing quickly. Focus of large database usage has shifted from accountancy to analytics. Though the need for correct accounting and data warehousing practice still prevails, but it is becoming a shrinking fraction of the volume. The emerging trend rather focuses on supporting predictive analytics via statistical models and algorithms, for potentially noisy data.

In 2008, a group of data scientists came together to describe the emerging trend in database industry and developed a number of non-trivial analytics techniques implemented as simple SQL scripts [4]. They called it MAD, an acronym for *Magnetic* platform, *Agile* design patterns for modeling, loading and iterating over data, and *Deep* statistical models and algorithms for data analysis. This work eventually led to the development of a software framework - a library of analytic methods that can be installed and executed within a relational database engine that supports extensible SQL [9]. This library is known as MADlib.

MADlib is a free, open source library for in database analytic method available at `http://madlib.net`. It provides an evolving suite of SQL-based algorithms for machine learning, data-mining and statistics that run at scale within a database engine, with no need for data import/export to other tools. The goal of MADlib project is to eventually serve a role for scalable database systems that is similar to the CRAN library for R: a community repository of statistical methods supporting scalability and parallelism. At present, MADlib works with Postgres and Greenplum only and provides support for a limited set of analytic methods as shown in Table 1. The methods are implemented mostly in python, C++ and SQL. The project is open for contributions of both new methods, and ports to additional database platforms.

In this project, we aimed at contributing to the MADlib project by implementing two different machine learning algorithms - the first one is *Genetic Programming* which is a prediction algorithm and the second one is *Adaptive Boosting* which is a popular classification algorithm. Our goal was to implement those algorithms in python and SQL, incorporate them into MADLib and analyze their performance on a number of factors.

The rest of the report is organized as follows: in Section 2 we discuss the state of the art. Section 3, describes the background for each of the algorithms we implemented. We also discuss the MADlib implementations of the algorithms. In Section 4, we discuss the benchmarking setup and datasets. We also discuss our findings from benchmarking the algorithms on various factors. Finally, we conclude our report in Section 5.

Category	Method
Supervised Learning	Linear Regression Logistic Regression Naive Bayes Classification Decision Trees (C4.5) Support Vector Machines
Unsupervised Learning	k-Means Clustering SVD Matrix Factorization Latent Dirichlet Allocation Association Rules
Descriptive Statistics	Count-Min Sketch Flajolet-Martin Sketch Data Profiling Quantiles
Support Modules	Sparse Vectors Array Operations Conjugate Gradient Optimization

Table 1: Methods provided in MADlib v0.3 [9]

2 Related Works

The space of approaches that combines analytics with data has been growing rapidly. At a high level, there are two approaches:

1. Top-down language-based approach: This approach brings a statistical language to a data processing substrate.
2. Framework-based approach: This approach provides a framework to express statistical techniques on top of a data processing substrate.

Top-down approaches begin with a high-level statistical programming language like R or Matlab to specify machine learning algorithms. These high-level algorithms are then compiled down to the data infrastructure. Examples of such approach are System ML from IBM [8], Revolution Analytics [2] and SNOW [12].

Framework-based approaches provide a set of building blocks (individual machine learning algorithms) with library support for macro- and micro-programming to write the algorithms. Typically they provide a template to automate the common aspects of deploying an analytic task over a data substrate. There have been different framework based approaches for different data substrates. For example, MADlib provides a machine learning and statistics library for RDBMS [9]. Apache Mahout provides an open source machine learning library for Apache Hadoop [1]. SciDB advocates a completely rewritten DBMS engine for numerical computation [11]. GraphLab framework provides simplified support for programming parallel machine learning tasks [10]. Spark is a Scala-based domain-specific language (DSL) targeted at machine learning, providing access to the fault-tolerant, main-memory resilient distributed datasets [13]. ScalOps provides a Scala DSL for machine learning that is translated to Datalog, which is then optimized to run in parallel on the Hyracks infrastructure [3]. ScalOps bears more similarity with MADlib since it has its origins in Datalog and parallel relational algebra.

We limited our focus on understanding and extending MADlib. Currently, MADlib has much room for growth in multiple dimensions. The MADlib library supports only a limited number of machine learning algorithms as shown in Table 1. So, there is an open invitation to contribute additional statistical models and algorithmic methods, both textbook techniques and cutting-edge research. Also, there is the challenge of porting MADlib to DBMSs other than PostgreSQL and Greenplum. Since MADlib is open source, anyone can contribute to MADlib codebase following their guidelines.

3 Implementation of Algorithms

In this section, we discuss the implementation details of our algorithm. Before diving into the details of our implementations, first we look at the architecture of MADlib. Then we discuss the background and implementation details of genetic programming and adaptive boosting.

3.1 Understanding MADlib architecture

Performing linear algebra over matrices is at the heart of many statistical methods.And it is quite challenging for relational databases to operate over very large matrices. At a macroscopic scale, the matrices need to be partitioned intelligently so that the partitions can be fit into memory of a single machine and then there should be options for movement of these partitions in and out of database by using SQL constructs. At microscopic level, the database engine needs to be able to invoke efficient linear algebra functions on the data partitions. MADlib uses a number of techniques to address these challenges:

User-defined Aggregation

MADlib uses user-defined aggregates (UDA) as a basic building block in the macroscopic level whenever applicable. UDAs are the natural way in SQL to implement mathematical functions that take arbitrary number of tuples as input. A user-defined aggregate consists of three user-defined functions:

- A *transition function* that takes current transition state and a new tuple and combines them into a new transition state.
- An optional *merge function* that takes two transition states and computes a new combined transition state. This function is only needed for parallel execution.
- A *final function* that takes a transition state and transforms it into output value.

However, UDAs are not enough for implementing iterative algorithms which perform multiple passes over the data.

Driver functions for Multipass Iteration

Many statistical and machine learning algorithms are iterative in nature and do multiple passes over the dataset. They cannot be implemented as simple UDAs. MADlib implements complex iterative methods by writing driver user-defined functions (UDF) in Python to control iteration. The driver UDF stores any inter-iteration output into a temporary table and reuses the resulting table in subsequent iterations as needed. Final outputs are also stored in tables unless they are small, and can be queried as needed. As a result all data movement is done inside database engine and its bufferpool.

Support for Microprogramming

MADlib implements row-level UDFs (functions that are called multiple times per row) in C or C++. Thus, these functions can call open source libraries like LAPACK or Eigen. MADlib also includes a sparse matrix library written in C.

C++ abstraction layer for UDFs

MADlib provides a C++ abstraction layer for making it easier to write driver UDFs. UDFs can be written using standard C++ atomic types as well as vector and matrix types that are native to linear algebra operations. The C++ abstraction layer also provides memory management and exception handling support. Finally, it includes third-party libraries so that developers can write efficient code.

Python support for UDFs

MADlib is still in its early stages of development. It does not provide a full-fledged Python abstraction layer for writing driver UDFs. It only provides a database access layer via `plpy.py` module using classic PyGreSQL interface named `pg`.

Currently, most of the MADlib functions require to be cognizant of the schema of the input tables and produce output table with fixed schema.

3.2 Genetic Programming for symbolic regression

3.2.1 Background

Genetic programming is a subset of evolutionary algorithms, which is a family of optimization algorithms based on the principle of **Darwinian natural selection**. As part of natural selection, a given environment has a population of individuals that compete for survival and reproduction. The ability of each individual to achieve these goals determines their chance to have children, in other words to pass on their genes to the next generation of individuals, who for genetic reasons will have an increased chance of doing well, even better, in realizing these two objectives.

This principle of continuous improvement over the generations is taken by evolutionary algorithms to optimize solutions to a problem. In the **initial generation**, a **population** composed of different **individuals** is generated randomly or by other methods. An individual is a solution to the problem, more or less good: the quality of the individual in regards to the problem is called **fitness**, which reflects the adequacy of the solution to the problem to be solved. The higher the fitness of an individual, the higher it is likely to pass some or all of its genotype to the individuals of the next generation.

An individual is coded as a **genotype**, which can have any shape, such as a string (genetic algorithms), a vector of real (evolution strategies) or in our case a tree (genetic programming). Each genotype is transformed into a **phenotype** when assessing the individual, i.e. when its fitness is calculated. In some cases, the phenotype is identical to the genotype: it is called **direct coding**. Otherwise, the coding is called indirect. For example, suppose you want to optimize the size of a rectangular parallelepiped defined by its length, height and width. To simplify the example, assume that these three quantities are integers between 0 and 15. We can then describe each of them using a 4-bit binary number. An example of a potential solution may be to genotype 0001 0111 01010. The corresponding phenotype is a parallelepiped of length 1, height 7 and width 10.

During the transition from the old to the new generation are called **variation operators**, whose purpose is to manipulate individuals. There are two distinct types of variation operators:

- the **mutation operators**, which are used to introduce variations within the same individual, as genetic mutations;
- the **crossover operators**, which are used to cross at least two different genotypes, as genetic crosses from breeding.

The figure 1 summarizes how an evolutionary algorithm works.

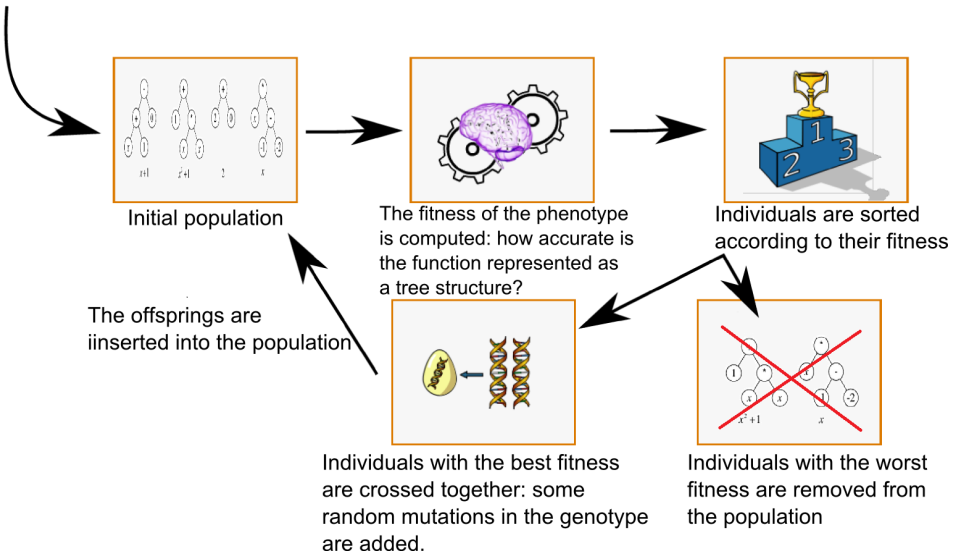

Figure 1: Functioning of an evolutionary algorithm: from an initial population of solutions (in symbolic regression, a solution is the tree representing a fraction), they are ranked according to their fitness, the worst ones are eliminated and the best ones are used to produce new solutions.

Genetic programming is a perfect suit for symbolic regression. The term "symbolic regression" represents the process during which measured data are fitted by suitable mathematical formula like x^2+C, $sin(x)+1/e^x$, etc. The figure 2 shows how a formula can the represented as a tree. This process is quite well known amongst mathematician and is used when some data of unknown process are obtained. The domain of symbolic regression is of functional nature, i.e. it consists of a function set like ($sin()$, $cos()$, $gamma()$, $MyFunction()$,...) and so called terminal set (t, x, p, ...). The final program is synthesized from a mixture of both sets, and can be quite complicated from a structural point of view. We plan to use genetic programming for symbolic regression in order to unravel unknown relation between some given attributes of a relation.

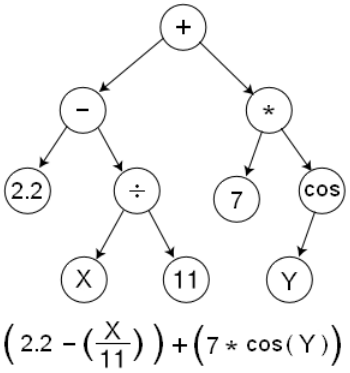

Figure 2: A function represented as a tree structure. Source: Wikipedia

3.2.2 Madlib Implementation

We implemented the symbolic regression using genetic programming within a Python module for MADlib that depends on the Python evolutionary computation framework DEAP [5]. The user can choose whether to store the data in memory, or retrieve by batch: this is a CPU vs. RAM trade-off.

Below is an example of a query that runs a symbolic regression, with 100 individuals per population size, 20 generations, and with a maximum size of the trees of 3. It takes 3 attributes as input (x1, x2 and x3) and one attribute as output (y1):

```
postgres=# SELECT * from madlib.gp('mock', '{x1, x2, x3}', '{y1}', 100, 20, 3);
```

3.3 Adaptive Boosting

3.3.1 Background

Boosting is one of the most important developments in classification methodology. Boosting works by iteratively applying a classification algorithm to re-weighted versions of the training data and then taking a weighted majority vote of the sequence of classifiers thus produced. For many classification algorithms, this simple strategy results in dramatic improvements in performance. While boosting has evolved over the years, we focus on the most commonly used version of boosting known as Adaptive Boosting (AdaBoost) for binary classification developed by Freund and Schapire [6].

Let us look at a concise description of AdaBoost in a two-class classification setting. We have training data $(x_1, y_1), \ldots, (x_n, y_n)$ with x_i a vector valued feature and $y_i = -1$ or 1. We define $F(x) = \sum_1^M c_m f_m(x)$ where each $f_m(x)$ is a classifier producing values 1 or -1 and c_m are constants; the corresponding prediction is sign$(F(x))$. The AdaBoost procedure trains the classifiers $f_m(x)$ on weighted versions of the training sample, giving higher weight to cases that are currently misclassified. This is done for a sequence of weighted samples, and then the final classifier is a linear combination of the classifiers from each stage. The pseudocode for AdaBoost is given in Figure 3.

AdaBoost

1. Start with weights $w_i = 1/N, i = 1, \ldots, N$.
2. Repeat for $m = 1, 2, \ldots, M$:
   (a) Fit the classifier $f_m(x) \in \{-1, 1\}$ using weights w_i on the training data.
   (b) Compute $\text{err}_m = E_w[1_{(y \neq f_m(x))}]$, $c_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (c) Set $w_i \leftarrow w_i \exp[c_m 1_{(y_i \neq f_m(x_i))}]$, $i = 1, 2, \ldots, N$, and renormalize so that $\sum_i w_i = 1$.
3. Output the classifier $\text{sign}[\sum_{m=1}^M c_m f_m(x)]$.

Figure 3: AdaBoost algorithm. E_w represents expectation over the training data with weights $w = (w_1, w_2, \ldots, w_N)$ and $I_{(S)}$ is the indicator of the set S [7].

We used "Stumps" as weak learners. Stumps are single-split trees with only two terminal nodes. Stumps are simple to implement, typically have low variance and success of boosting depends on variance reduction [7].

3.3.2 Madlib Implementation

The challenge in implementing (binary) Adaptive Boosting classification for MADlib is that, the algorithm is iterative in nature and in each iteration, it needs to make a pass over the entire training dataset as well as the weights data. Therefore, it cannot be implemented as a simple user-defined aggregate (UDA). We implemented AdaBoost as a driver UDF in Python. For performing Matrix arithmetic we used the `numpy` package. The control flow of the UDF is shown in Figure 4.

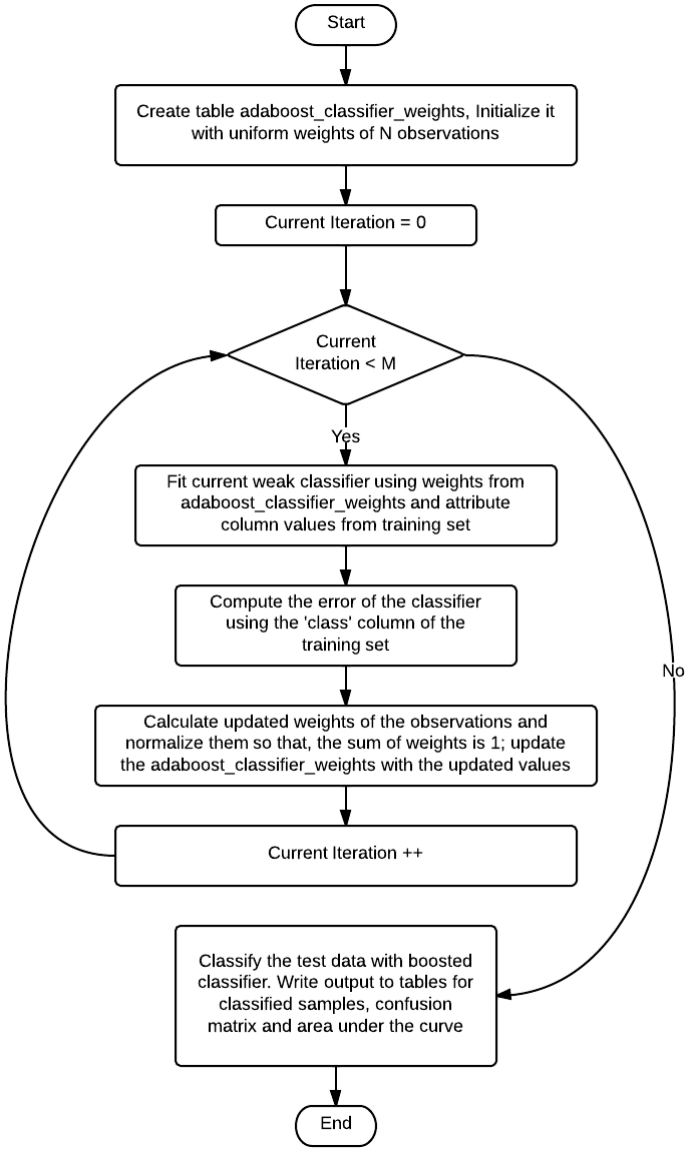

Figure 4: Control Flow of AdaBoost Implementation for MADlib.

The row-by-row, batched and in-memory versions differ only in how we access the training and test dataset and weights table. See Appendix E for documentation of usage of AdaBoost module.

4 Benchmark

In this section, we show the results of the benchmarks that we performed to assess the performance of our implementations in various settings. We also discuss the implications of the results that we found.

4.1 Setup

We ran the benchmark on PostgreSQL 9.1.9 and MADlib v0.3 on a single machine on a gigabit Ethernet cluster which runs Ubuntu 12.04 LTS and has 4GB RAM, a 10GB SATA HDD and a Core 2 Duo 2.4GHz CPU.

To benchmark the performance of GP for Symbolic Regression we used a synthetic dataset containing 100,000 rows, 3 inputs and 1 output. The function we used to generate the data is $x_1 * (x_2^2 + x_3)$.

To benchmark the performance of AdaBoost, we used BUPA liver disorder dataset which contains blood test results of 345 male individuals. This dataset is available at http://www.cs.huji.ac.il/~shais/datasets/ClassificationDatasets.html. We also used a synthetic dataset which consisted of 240000×11 matrix where first 10 columns are real valued random numbers and the last column indicates the class.

Each run of our algorithms inside MADlib was cold start meaning that we cleared the caches and restarted the database service (PostgreSQL) for every run.

4.2 Analysis of Results

Our benchmarking results are tabulated in Appendix A. Here, we discuss our understanding of the findings.

Effect of varying independent variables on runtime and memory usage.

The first focus of our analysis was the effect of the parameters of the symbolic regression and of AdaBoost on runtime and memory usage. Figure 5 shows that when we increase the value of our parameters, it increases the runtime proportionally as expected. By the same token, increasing the size of the data set cause PostgreSQL to use a higher amount of memory (Figure 9 all-in-memory execution).

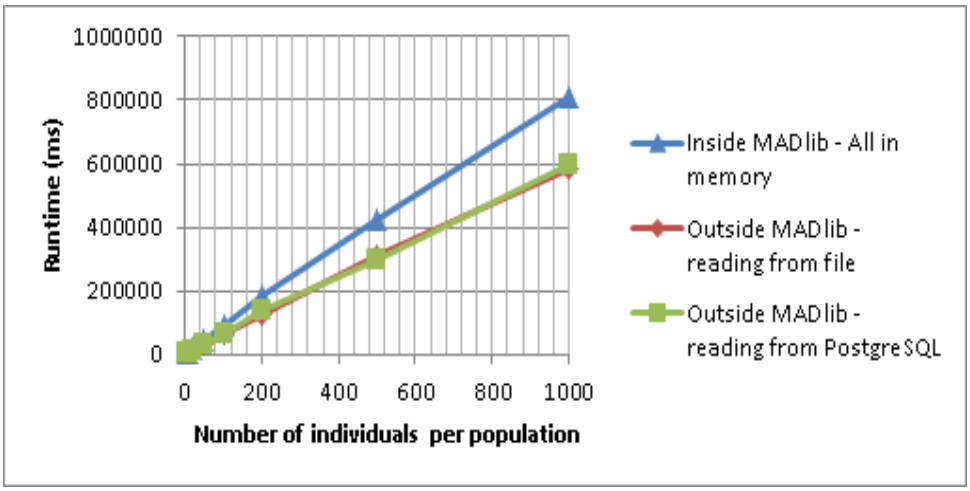

Figure 5: Runtimes of symbolic regression inside and outside MADlib.

Performance inside and outside MADlib

Figure 5 also compares the runtime between different execution environments: within MADlib, outside MADlib reading from a file and outside MADlib reading from the PostgreSQL database. The code used in all the cases was strictly identical in order to ensure a fair comparison between execution environments. Also, all the data was loaded in memory just before the function call.

First of all, in the case of the dataset that we used to perform symbolic regression, which contains 100,000 rows and of size 2686 KB, on average reading from a file takes 230 ms whereas it takes 510 ms reading

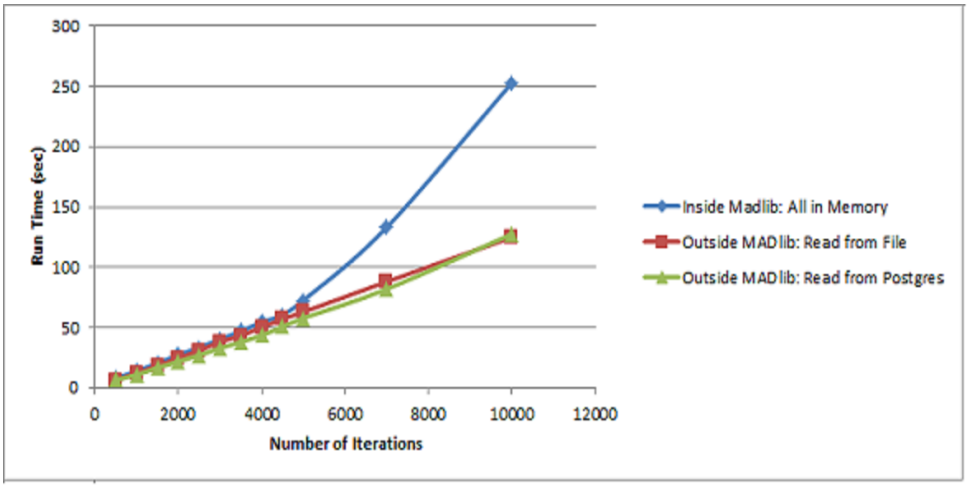

Figure 6: Runtimes of AdaBoost algorithm inside and outside MADlib.

from the PostgreSQL database. It takes 12.14 ms on average to read the BUPA dataset (345×7) from file where reading from PostgreSQL takes 12.68 ms.

The most intriguing result is the runtime within MADlib, which is significantly slower than the runtime outside MADlib. This result cannot be explained by the fact that querying the database makes running within MADlib slower since running outside MADlib reading from the PostgreSQL doesn't have the issue, and the difference of runtime increases when the number of iterations increases, which means the reason isn't a fixed cost.

To investigate this difference of runtime we bypassed MADlib and created PL/Python test case that narrows down the issue:

```
CREATE FUNCTION testcase (b integer)
  RETURNS float
AS $$
  import time
  start = time.time()
  a = 0
  for i in range(b):
    for ii in range(b):
      a = (((i+ii)%100)*149819874987)
  end = time.time()
  plpy.info("Time elapsed in Python: " + str((end - start)*1000) + ' ms')
  return a
$$ LANGUAGE plpythonu;
```

We compared the latter code with the following identical Python code:

```
import time
import sys

def testcase (b):
    a = 0
    for i in range(b):
        for ii in range(b):
            a = (((i+ii)%100)*149819874987) # keeping Python busy
    return a

def main():
    numIterations = int(sys.argv[1])
    start = time.time()
    print testcase(numIterations)
    end = time.time()
    print "Time elapsed in Python:"
```

```
    print str((end - start)*1000) + ' ms'

if __name__ == "__main__":
    main()
```

On our benchmark server, calling the PostgreSQL PL/Python function using *select * from testcase(20000);*, it takes on average 65 seconds, while when we call the usual Python script with 20000 as argument too it takes an average 48 seconds. The averages were computed running the queries and scripts 10 times. This result means that for some reason the CPython that is embedded in PostgreSQL 9.1 is slower than the Python 2.7.3 we use outside PostgreSQL.

We tested with PostgreSQL 9.2 (with Ubuntu 12.10 this time), and we still notice a runtime difference on my server (although overall 10% faster, probably due to some new version of CPython). We also tried using plpython3u, which implements PL/Python based on the Python 3 language variant. plpythonu that we used before is equivalent to plpython2u, which implements PL/Python based on the Python 2 language variant. Using plpython3u is far slower (88 seconds), but when running the Python script using python3 it is also slower (75 seconds), although still significantly faster than plpython3u.

Performance of batched execution.

So far our modules have loaded all the data in memory, then performed the computations on them. However, in some situation we might not have enough memory to store all the data. In older to circumvent this issue our modules allow to define the amount of memory the user wants to grant to the module. To do so, our GP implementation lets the user to choose the batch size, which corresponds to the number of rows the module can put in memory and the AdaBoost implementation lets the user to choose the number of batches (dataset will be divided into the number of partitions chosen by the user). At each iteration of the algorithm, we retrieve data in batches. Since we cannot store them in memory, it means we will have in total a significant amount of read/write accesses to the database. The smaller the batch size the higher the amount of read/write queries at each iteration will be.

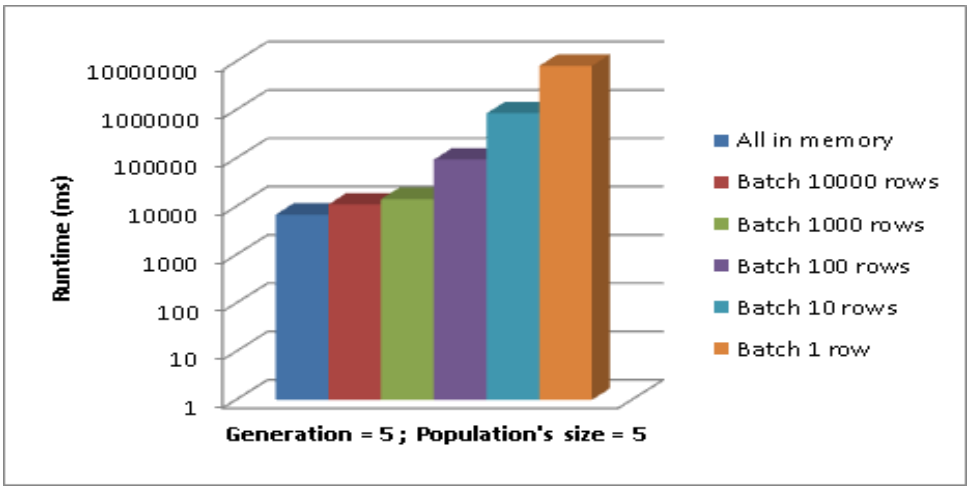

Figure 7: Runtimes of row-by-row, batched and all-in-memory execution of symbolic regression. The dataset contains 100,000 rows. Having a batch size of 10,000 rows means that at each iteration we do $100,000/10,000 = 10$ queries on the database

Figure 7 shows the effect of the batch size in the case of the symbolic regression. We fixed the parameters and only changed the batch size. As we can see, having a batch size of 10,000 rows or a batch size of 1,000 rows is a pretty good trade-off: for batch size = 10,000 rows, the runtime is 1.5 times bigger, and for batch size = 1,000 rows, the runtime a bit less than twice bigger. See Table 3 in Appendix A. Since the memory used is proportional to the batch size, it makes using large batches look attractive. However, if we further decrease the batch size, we see that it starts having a very negative impact on the runtime.

Figure 8 shows that, in case of AdaBoost, as we decrease the number of batches runtime decreases which is clearly useful in case we don't have enough memory to load the entire dataset. However, there is a significant difference between the runtimes of batched case vs all-in-memory case. This is because, in our all-in-memory implementation, we load the training and testing datasets once and then all weights

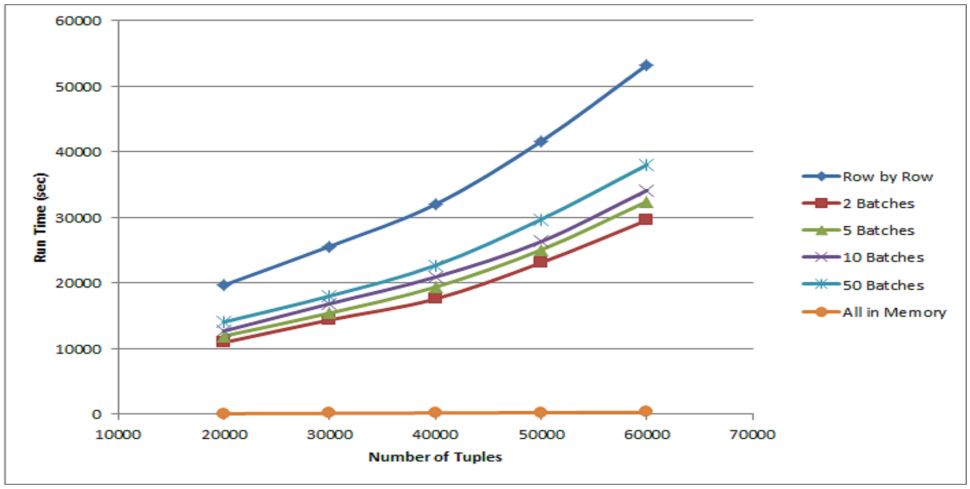

Figure 8: Runtimes of row-by-row, batched and all-in-memory execution of AdaBoost algorithm on synthetic dataset.

calculations are done in memory, whereas, in batched or row-by-row version, we create a weights table and in each iteration we read the weights table for fitting the classifier and also update the rows of the table with updated weights. Thus both batched and row-by-row version include significant amount of disk access which is time-consuming.

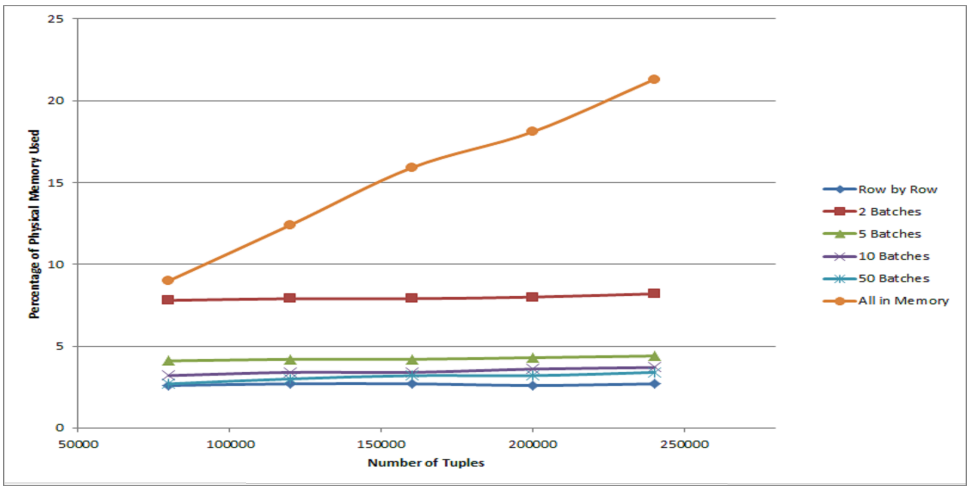

Figure 9: Memory usage by AdaBoost algorithm on synthetic dataset.

Table 6 shows the runtimes and Table 7 shows the memory usage of row-by-row execution, batched execution and all-in-memory execution of AdaBoost when it is run on a synthetic dataset of varying size. We can see that, batching gives us advantage in terms of memory usage.

5 Conclusions & Future Work

In this project, we aimed at contributing to MADlib by implementing two algorithms namely, Genetic Programming for Symbolic Regression and Adaptive Boosting for Binary Classification. We implemented the algorithms as MADlib modules, more specifically as Python driver UDFs inside PostgreSQL database. We tried to analyze the performance of these algorithms in a number of different settings. From the benchmarks we did, we initially found that, the runtime of both of the algorithms increases as we increase the number of independent variables (e.g., number of individuals in population or number of iterations for boosting). Running the algorithms using MADlib takes more time than bypassing MADlib altogether when data can be fit into memory. Running the algorithms using MADlib is advantageous when the dataset cannot be fit into memory. Batched execution is more efficient than row-by-row execution in terms of run time. As we increase batch size (i.e. decrease number of batches) we can see a decrease

in runtimes of the algorithms. Also, it is more efficient than reading the whole dataset into memory in terms of memory usage. However, we were faced with the issue that, the im-memory implementation of both the algorithms take more time inside MADlib which needs further investigation.

Our ultimate goal is to contribute our code to the MADlib codebase: we have already established contact with the core developers through the official MADlib mailing-list. In future, we plan to incorporate parallelism in these algorithms. Right now our implementation works only with PostgreSQL. We would like to add support for Greenplum, as MADlib is also compatible with it, and take advantage of their native parallelism.

References

[1] Apache Mahout. http://mahout.apache.org.

[2] Revolution Analytics. http://www.revolutionanalytics.com.

[3] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1151–1162. IEEE, 2011.

[4] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.

[5] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[6] Yoav Freund and Robert E Schapire. Experiments with a new boosting algorithm. In *Machine Learning International Workshop Then Conference*, pages 148–156. Morgan Kaufman Publishers, Inc., 1996.

[7] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *The annals of statistics*, 28(2):337–407, 2000.

[8] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

[9] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[10] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[11] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of SciDB. In *Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[12] Luke Tierney, Anthony J Rossini, and Na Li. Snow: A parallel computing framework for the R system. *International Journal of Parallel Programming*, 37(1):78–90, 2009.

[13] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

Appendix A: Tables showing Benchmarking Results

The following tables show benchmarking results for symbolic regression using genetic programming and classification using adaptive boosting.

Pop size	# of gen.	Runtimes (sec)		
		Outside MADlib from PostgreSQL	Outside MADlib from File	Inside MADlib
5	5	5.513	4.860	6.787
10	5	7.166	6.295	8.560
20	5	13.879	12.484	17.610
50	5	32.568	30.637	44.261
100	5	65.385	62.013	88.910
200	5	135.105	128.238	182.672
500	5	300.677	314.149	425.092
1000	5	597.274	582.635	810.041
5	10	7.888	7.669	10.523
5	20	11.245	11.137	16.282
5	50	25.119	23.975	35.220
5	100	47.355	46.355	69.156
5	200	92.778	95.063	119.511
5	500	226.639	239.827	293.664
5	1000	491.201	489.361	591.823
20	20	44.813	44.813	56.965

Table 2: Symbolic Regression runtimes on synthetic dataset.

Pop; Gen	Runtimes (sec)					
	All-in-Memory	10000-row batch	1000-row batch	100-row batch	10-row batch	Row-by-Row
5; 5	6.787	11.012	14.220	93.387	845.325	8153.139

Table 3: Symbolic Regression runtimes on synthetic dataset. Pop stands for population size. Gen stands for number of generations.

NumIter	Runtimes (sec)					
	All-in-Memory	2 Batches	5 Batches	10 Batches	50 Batches	Row-by-Row
50	1.04	96.72	99.23	149.96	213.31	314.75
100	2.09	147.90	236.93	527.41	770.55	1215.30
150	2.69	283.44	548.98	844.80	1746.58	2727.17
200	3.39	675.45	986.81	1107.41	3118.15	4334.23
250	4.18	1052.99	1728.76	2546.37	3118.154	6525.33
300	4.88	1566.79	2814.76	3926.46	6367.52	9041.89
350	5.72	2578.63	3789.65	5444.76	8567.15	11822.67
400	6.58	4548.88	5567.59	9720.97	12319.85	16878.20
450	7.58	7143.48	8869.80	12414.58	17613.34	24683.87
500	7.96	9776.36	11934.97	16977.72	23814.81	30417.70

Table 4: AdaBoost runtimes on BUPA liver disorder dataset (Row-by-Row vs. Batched vs. All-in-Memory execution).

NumIter	Runtimes (sec)		
	Outside MADlib from PostgreSQL	Outside MADlib from File	Inside MADlib
500	6.73	6.37	7.96
1000	10.85	12.23	14.01
1500	16.82	18.27	20.43
2000	21.87	24.56	27.50
2500	27.13	30.92	33.52
3000	32.95	38.25	40.27
3500	38.11	43.28	47.11
4000	43.51	50.45	54.48
4500	50.86	57.02	60.51
5000	57.86	62.97	72.63
7000	82.00	88.12	133.256
10000	127.60	124.88	252.98

Table 5: AdaBoost runtimes on BUPA liver disorder dataset (Inside vs. Outside MADlib).

NumTuples	Runtimes (sec)					
	All-in-Memory	2 Batches	5 Batches	10 Batches	50 Batches	Row-by-Row
20000	92.02	10938.07	11932.44	12702.27	14063.23	19688.52
30000	143.13	14379.26	15418.72	16838.87	18024.70	25595.08
40000	201.38	17579.04	19390.21	20911.01	22690.67	31993.85
50000	252.59	23106.67	25055.43	26324.06	29708.58	41592.01
60000	320.38	29576.54	32432.05	34126.77	38026.98	53237.77

Table 6: AdaBoost runtimes on synthetic dataset.

NumTuples	Memory Usage (%)					
	All-in-Memory	2 Batches	5 Batches	10 Batches	50 Batches	Row-by-Row
80000	9.0	7.8	4.1	3.2	2.7	2.6
120000	12.4	7.9	4.2	3.4	3.0	2.7
160000	15.9	7.9	4.2	3.4	3.2	2.7
200000	18.1	8.0	4.3	3.6	3.2	2.6
240000	21.3	8.2	4.4	3.7	3.4	2.7

Table 7: AdaBoost memory usage for synthetic dataset

Appendix B: Installing Madlib with PostgreSQL

The following instructions have been tested on Ubuntu 12.04 x64. They will not work with Ubuntu 12.04 x32 (MADlib does not have support for Ubuntu 32-bit). Also, MADlib doesn't work with GCC 4.7.*. Since, Ubuntu 12.10 ships with GCC 4.7.2 it might be an issue while installing MADlib on Ubuntu 12.10.

Install PostgreSQL packages:

```
$ sudo apt-get -y install \
  postgresql-9.1 libpq-dev \
  postgresql-server-dev-9.1 \
  postgresql-plpython-9.1
```

Download MADlib from http://madlib.net and copy .tar file to server:

```
$ wget https://github.com/madlib/madlib/zipball/v0.5.0
$ unzip v0.5.0
$ cd madlib-madlib-5fabd88
```

Build Madlib:

```
$ sudo apt-get -y install cmake
$ sudo apt-get -y install m4 gcc-4.6 g++-4.6 g++
$ ./configure
$ cd build/
$ make
$ make doc
$ make install
```

Connect to the database:

```
$ sudo su - postgres
$ psql
```

Add a password to a role:

```
postgres=# ALTER ROLE postgres WITH PASSWORD 'postgres';
```

Register Madlib with the PostgreSQL database

```
$ /usr/local/madlib/bin/madpack -p postgres -c \
  $USER@$HOST/postgres install
```

Test the installation by running install check procedure:

```
$ /usr/local/madlib/bin/madpack -p postgres -c \
  $USER@$HOST/$DATABASE install-check
```

Appendix C: Creating a module in Madlib

Create a new function testpy. We are going to create this section in any module called 'testmod'.

- Add new module folder: ./src/ports/postgres/modules/testmod/
- Create in this folder the following files:
  - __init__.py_in
  - testmod.py_in
  - testmod.sql_in
- Modify ./src/config/Modules.yml and add a new line " - name: testmod"
- recompile MADlib: with any user:
  - ./configure
  - make install # (no need for make clean) (in the MADlib build folder)
- re-register MADlib into Postgresql (with postgres user):
  - /usr/local/madlib/bin/madpack -p postgres -c $USER@$HOST/postgres reinstall

Appendix D: How to use Genetic Programming module in MADlib

Input

The input data is expected to be of the following form:

```
TABLE tableName (
    x1 DOUBLE PRECISION,
    ...
    xN DOUBLE PRECISION,
    y DOUBLE PRECISION
)
```

Usage

Perform Symbolic Regression using Genetic Programming:

```
postgres=# SELECT * FROM madlib.gp (
               'inputTableName',
               '{x1, ..., xN}', '{y}',
                numIndividuals,
                numGenerations,
                maxDepthOfTree
          );
```

Example

This is an example showing how to perform symbolic regression using the genetic programming module in MADlib.

Generate an artificial dataset using MATLAB that contains 100,000 rows:

```
x1 = 1:0.0001:(10+5000);
x2 = 10:0.0001:(19+5000);
x3 = 5:0.0001:(14+5000);
a =[x1; x2; x3; x1.*(x2.^2+x3)];
csvwrite('mock.csv', a(1:100000, :))
```

Import this dataset within PostgreSQL:

```
postgres=# CREATE TABLE mock (X1 real, X2 real, X3 real, Y1 real);
postgres=# COPY mock FROM '/root/mock.csv' DELIMITERS ',' CSV;
```

Run the symbolic regression, with 100 individuals per population size, 20 generations, and to maximum size of the trees of 3. We take 3 attributes as input (x1, x2 and x3) and one attribute as output (y1):

```
postgres=# SELECT * from madlib.gp('mock', '{x1, x2, x3}', '{y1}', 100, 20, 3);
```

The above query produces the following output:

```
           individual            |      fitness
---------------------------------+-------------------
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
 [mul, add, mul, x2, x2, x3, x1] | 0.000442927237356
(9 rows)
```

Each row corresponds to one individual, i.e. one formula. In this example, all rows correspond to the formula $x_1 * (x_2^2 + x_3)$, which is what we wanted to find. The fitness reflects how accurate each individual is. The lowest the fitness, the most accurate the formula is.

Appendix E: How to use AdaBoost module in MADlib

Currently our implementation supports only binary classification.

Input

The *training data* as well as *test data* is expected to be of the following form:

```
TABLE tableName (
    id INTEGER, // should be 1 indexed
    attribute1 DOUBLE PRECISION,
    ...
    attributeN DOUBLE PRECISION,
    class INTEGER // should be either 1 or -1
)
```

Usage

Perform AdaBoost classification loading the whole dataset into memory: (This type of execution is pretty fast when the dataset is small and fits into memory)

```
postgres=# SELECT * from madlib.adaboost_train_and_classify (
               'trainingSet', 'testSet',
               '{attribute1, ..., attributeN}',
               'class', numberOfIterations, pValue
           );
```

When the dataset cannot be fit into memory, the user can use either batched or row-by-row version of AdaBoost. Perform row-by-row version of AdaBoost classification:

```
postgres=# SELECT * from madlib.adaboostRow_train_and_classify (
               'trainingSet', 'testSet',
               '{attribute1, ..., attributeN}',
               'class', numberOfIterations, pValue
           );
```

Perform batched version of AdaBoost classification:

```
postgres=# SELECT * from madlib.adaboostBatch_train_and_classify (
               'trainingSet', 'testSet',
               '{attribute1, ..., attributeN}',
               'class', numberOfBatches,
                numberOfIterations, pValue
           );
```

Example

This is an over-simplified example of the in-memory execution of AdaBoost classification. Batched and row-by-row versions are similar. The training and test data:

```
postgres=# SELECT * from trainingSet;
 id | attr1 | attr2 | attr3 | class
----+-------+-------+-------+--------
  1 |    85 |    92 |    45 |     1
  2 |    85 |    64 |    59 |    -1
  3 |    86 |    54 |    33 |     1
  4 |    91 |    78 |    34 |     1
  5 |    87 |    70 |    12 |    -1
  6 |    98 |    55 |    13 |     1
(6 rows)

postgres=# SELECT * from testSet;
 id | attr1 | attr2 | attr3 | class
----+-------+-------+-------+--------
```

```
  1 |    95 |    82 |    15 |    -1
  2 |    86 |    54 |    54 |     1
  3 |    88 |    64 |    43 |     1
  4 |    81 |    70 |    46 |     1
  5 |    97 |    77 |    10 |    -1
  6 |    88 |    65 |    12 |    -1
(6 rows)
```

Perform AdaBoost classification:

```
postgres=# SELECT * FROM madlib.adaboost_train_and_classify (
               'trainingSet', 'testSet',
               '{attr1, attr2, attr3}',
               'class', 50, 0.05
           );
```

The above query produces the following output summary:

```
      result_table_name
-----------------------------
 adaboost_classifier_weights
 adaboost_classified_samples
 adaboost_confusion_matrix
 adaboost_AUC
(4 rows)
```

Check the contents of the above tables to get the classification results:

```
postgres=# SELECT * FROM adaboost_classifier_weights;

 id |     weight
----+-----------------
 0  | 0.0492970944955
 1  |  0.161430043833
 2  |  0.225351747846
 3  |  0.225351747846
 4  |  0.190980668725
 5  |  0.147588697256
(6 rows)

postgres=# SELECT * FROM adaboost_classified_samples;

 row_id |     score      | predicted_class
--------+----------------+-----------------
 0      |  12.3265804125 |               1
 1      | -12.2369378612 |              -1
 2      |  11.9033472397 |               1
 3      |  11.9033472397 |               1
 4      |  10.7171405001 |               1
 5      | -12.0688375328 |              -1
(6 rows)

postgres=# SELECT * FROM adaboost_confusion_matrix;

 tp | fp | fn | tn
----+----+----+----
  2 |  2 |  1 |  1
(1 row)

postgres=# SELECT * FROM adaboost_AUC;

      auc
----------------
 0.444444444444
(1 row)
```