Unified Architecture for Python in-RDBMS Analytics*

Zhixiang Meng
University of Waterloo
Cheriton School of Computer Science
z9meng@uwaterloo.ca

Mengyun Zheng
University of Waterloo
Cheriton School of Computer Science
m24zheng@uwaterloo.ca

ABSTRACT

In this report, we replicate a unified architecture for in-RDBMS analytics, Bismarck, proposed by Feng.etl[8] in Python. Our main contribution in this work is to extend the C implemented system to a Python version(PyBismarck) which also supports Mini-Batch Gradient Descent(MBGD) aggregate computation. In our experiments, PyBismarck with MBGD demonstrated the possibility of leveraging open-source Python libraries in in-RDBMS analytics which would enable fast deployment of advanced machine learning models.

Categories and Subject Descriptors

H.2 [**Database Management**]: [Miscellaneous]

General Terms

Analytics, Incremental Gradient Descent, Mini-Batch Gradient Descent, PostgreSQL, User-Defined Aggregate, Python

1. INTRODUCTION

There has been an arms race among database vendors to offer sophisticated in-database analytics, due to the increasing use of statistical data analysis in enterprise applications.[8] This race, however, is blocked by the challenge of incorporating new statistical techniques into the existing RDSMS which leads to a time-consuming development process.

In this project, we attempted to replicate Bismarck, which is a unified architecture integrating many data analytics tasks (e.g. SVM and LMF) formulated as Incremental Gradient Descent (IGD) into an RDBMS proposed by Feng.etl[8]. We used Python to take advantages of advanced third-party packages and tools supported in Python. Therefore, we proposed PyBismarck which is based on similar architecture and supports similar functions. We extended gradient descent aggregate computation from IGD to Mini-Batch Gradient Descent (MBGD). Moreover, we enabled the system to import open-source packages for more sophisticated models available in Python.

The rest of the report is organized as follows: In Section 2, we will review the work has done by Feng.etl [8].In Section 3, we introduce three Gradient Descent methods and provide more details about gradient computation in Logistic Regression and Support Vector Machine. Then we introduce the general architecture of PyBismarck and implementation in detail in Section 4. In Section 5, we conduct the experiments based on Bismarck's source code to verify the impact of data ordering and necessity of parallelism. We then compare different gradient descent methods implemented in PyBismarck and validate that PyBismarck is able to integrate Python third-party packages into an RDBMS supporting more advanced analytics. The final section contains concluding remarks and future work.

2. BACKGROUND

The bottleneck of developing a system providing sophisticated in-database analytics is that each new data analytics technique requires ad hoc steps, such as new memory requirements and new data access methods, which result in little code reuse across different algorithms and thus slowing down the development process. Feng.etl [8] propose a solution to take a step towards a unified architecture for in-RDBMS analytics to solve such a problem and accelerate the process. In their work[8], they leverage the insights from mathematical programming literature that many common data analytics tasks can be framed as convex programming problems and develop a system called Bismarck which provides a single systems-level abstraction to implement a large class of existing and next-generation analytics techniques. They observe that IGD has a data-access pattern that is essentially identical to the data access pattern of SQL aggregation functions, which allows them to implement and integrate all the convex data analytics techniques in RDBMS by formulating IGD using aggregate features available in almost every commercial and open-source database system.

Comparing with the existing tools like MADlib [7], Oracle Data Mining [2], and Microsoft SQL Server Data Mining [1] which also provide SQL-like interface for end-user to proceed statistical analytics, Bismarck provides a single architecture abstraction for end-user to unify the in-RDBMS implementations of all the convex data analytics technique, such as

*This effort is part of the CS848, Advanced Topics in Databases: DB for ML, ML for DB (Spring 2019), class project. For coding details, we make it an open source `https://github.com/zxmeng/PyBismarckn`

Logistic Regression, SVM, LMF, etc. More specifically, end-user is able to specify a task, such as SVM, and corresponding hyperparameters and training data for such task. The Incremental Gradient Descent Aggregate component in the Bismarck will run the task in-RDMBS with a data access pattern like a SQL aggregate query.

In order to speed up the pure incremental gradient descent aggregation computation, they also explore the impact of data ordering and parallelizing gradient computation. IGD computation conventionally requires shuffling data before each iteration starts. From their experimental results obtained from Bismarck and the benchmark on a broad range of models, they found that shuffling only once before the first pass has a slightly lower convergence rate than shuffling for each pass but also reduce the runtime per iteration. Therefore, shuffling once has better overall performance than shuffling always. Moreover, since IGD can be parallelized in a shared-memory environment[15] leveraging the standard user-defined aggregation features in every RDBMS, they adopt Atomic Incremental Gradient scheme in Bismarck and obtain almost linear speed-ups.

3. METHODOLOGY

In this project, our main purpose is to replicate the system Bismarck proposed by Feng [8] in Python, which supports in-database analytics. As Feng.etl [8] stated in their work, incremental gradient descent (IGD) has essentially the same data-access pattern as SQL aggregation functions. Such observation inspired them to implement and integrate the convex data analytic techniques in RDBMS by formulating IGD using aggregate features available in almost every commercial and open-source database system. In this section, we will review the gradient methods and introduce more algorithmic details in implementing and aggregating Logistic Regression and Support Vector Machine as examples.

3.1 Gradient Methods

3.1.1 Batch Gradient Descent

To elaborate gradient descent method[9][13], we take a basic linear regression model for example. Suppose we have a data set with tuples as $x_1, y_1, ..x_n, y_n$, where $x's$ are d-dimensional vectors in $\mathbb{R}^d$. Then our goal is to find optimal weights $w \in \mathbb{R}^d$ for model:

$$h(x) = \sum_{i=1}^{n} w_i x_i = w^T x \tag{1}$$

Considering least-square cost function, the objective function can be expressed as follows:

$$J(w) = \frac{1}{2} \sum_{i=1}^{n} (h(x_i) - y_i)^2 \tag{2}$$

We want to choose w so as to minimize $J(w)$. Gradient Descent is a search algorithm that starts with some "initial guess" for w, and then repeatedly updates w to make $J(w)$ smaller, until w converges to a value that minimizes $J(w)$. And it is guaranteed to converge to the globally optimal solution.[14]. In each iteration, the j^th entry in w will be updated as:

$$w_j \leftarrow w_j - \alpha \frac{\partial}{\partial w_j} J(w) \tag{3}$$

Algorithm 1 Batch Gradient Descent

```
w ← arbitrary vector
for iteration ← 1, .., T do
    Shuffle the training set
    for for j^th entry in w do
        w_j ← w_j + α ∂/∂w_j J(w)
    end for
end for
```

where α is learning rate, and the gradient can be derived as:

$$\frac{\partial}{\partial w_j} J(w) = (h(x) - y) x_j \tag{4}$$

It can be expressed in pseudo-code as in Algorithm 1.

However, this method requires to scan through the entire training set before updating, so it is also called batch gradient descent.

In order to extend to all convex data analytics techniques, we generalize the objective function as follows:

$$\min_{w \in \mathbb{R}^d} \sum_{i=1}^{n} J(w, x_i) \tag{5}$$

Here, J_w expressed any abstract cost function including regularization term like $L1$ norm (Lasso) and $L2$ norm (Ridge)

3.1.2 Incremental Gradient Descent

The improvement in incremental gradient descent (also stochastic gradient) comparing with conventional gradient descent method is that it updates the parameters each time it encounters a training example according to the gradient of the error with respect to that single training example only while run through the data set. That is to say, we can access one tuple from a table in the database and update the weights accordingly right away. Such a data-access pattern is essentially identical to the data access pattern of SQL aggregation functions. The idea is expressed in pseudo-code as in Algorithm 2

Often, incremental gradient descent gets w close to the minimum faster than batch gradient descent. However that it may never converge to the minimum, and the parameters w will keep oscillating around the minimum of $J(w)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum[9][13]. Also, if we slowly let the learning rate α decrease to zero as the algorithm runs, it is also possible to ensure that the parameters will converge to the global minimum rather than merely oscillate around the minimum. For these reasons, particularly when the training set is large, incremental gradient descent is often preferred over batch gradient descent.[9][13] Moreover, even if the order of training example is a fixed, arbitrary order, the algorithm also converges.[8][10][11][12][16]

3.1.3 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent (MBGD) is a compromise between batch gradient descent (BGD) and incremental gradient descent(IGD). It simply separates the training set into

Algorithm 2 Incremental Gradient Descent

```
w ← arbitrary vector
for iteration ← 0, .., T do
    Shuffle the training set
    for i^th tuple in data table do
        for for j^th entry in w do
            w_j ← w_j + α ∂/∂w_j J(w)
        end for
    end for
end for
```

Algorithm 3 Mini-Batch Gradient Descent

```
B ← batch size
w ← arbitrary vector
for iteration ← 0, .., T do
    Shuffle the training set
    for t^th batch do
        for for j^th entry in w do
            w_j ← w_j + α ∂/∂w_j J(w)
        end for
    end for
end for
```

small batches and performs an update of model weights for each of these batches. The idea can be expressed in pseudocode as in Algorithm 3

Therefore, the model update frequency is higher than BGD which allows for a more robust convergence, avoiding local minima. The batched updates provide a computationally more efficient process than IGD. The batching allows both the efficiency of not having all training data in memory and algorithm implementations.

3.2 Logistic Regression

The hypothesis logistic model is:

$$h(x) = g(w^T \cdot x) = \frac{1}{1 + e^{-w^T \cdot x}} \tag{6}$$

The objective is to maximize the log likelihood which can be expressed as follows, where $y_i \in$ {-1,1}

$$l(w) = \sum_{i=1}^{n} log(1 + e^{-y_i w^T \cdot x_i}) + \mu \|\overrightarrow{w}\|_1 \tag{7}$$

For simplicity , we firstly reduce the regularization term $\mu\|\overrightarrow{w}\|_1$. Then the gradient for one training example x_i, y_i is:

$$\frac{\partial}{\partial w_j} l(w) = (h(x) - y_i)x_j \tag{8}$$

which follow the Generalized Linear Models format. Follow Feng's[8] coding style, we implement IGD for logistic regression as Algorithm 4. The MBGD method computes the gradient by applying similar aggregate function but on a batch of datasets. More details of MBGD implementation will be provided in Section 4.2

Algorithm 4 IGD for Logistic Regression

```
INITIALIZE
for iteration ← 0, .., T do
    for Each tuple x_i, y_i in the data table do
        IGD Aggregate
    end for
    Update parameters in model table
    Loss Aggregate
end for
function INITIALIZE(W, α, μ, T)
    w ← zero-vector
    alpha ← learning rate;
    μ ← regularization term hyper-parameter
    T ← number of iterations
end function
function IGD AGGREGATE(x_i, y_i, w, μ, α)
    sig ← 1/(1 + exp(w^T x_i y_i))
    c ← α * y * sig
    for j ← 0, ..d do
        w_j ← x_{i,j} * c
    end for
    u ← μ * α
    L1 Regularization(w, u)
end function
function L1 REGULARIZATION(w_j, u)
    for j ← 0, ..d do
        if w_j > u then
            w_j ← w_j − u
        end if
        if w_j < u then
            w_j ← w_j + u
        else
            w_j ← w_j
        end if
    end for

end function
function LOSS AGGREGATE(w, x, y,)
    loss ← 1 + exp(−w^t · x · y)
end function
```

Algorithm 5 IGD and Loss Aggregate for SVM

```
function GRADIENT DESCENT AGGREGATE(x_i, y_i, w, μ, α)
    u ← μ * α
    for j ← 0, ..d do
        if y_i * w_j * x_ij ⩽ 1 then
            w_j ← α * y_i * x_i,j
        end if
        L1 Regularization(w_j, u)
    end for
end function
function LOSS AGGREGATE(w, x, y,)
    loss ← max(0, 1 − w^T · x · y))
end function
```

3.3 SVM

The objective function for SVM classification is as follows:

$$J(w) = \sum_{i=1}^{n} max(0, 1 - y_i * w^T \cdot x_i) + \mu \|\overrightarrow{w}\|_1 \qquad (9)$$

However, $max(0, 1 - y_i * w^T \cdot x_i)$ (Hinge loss) is not differentiable with respect to w. We compute sub-gradient instead and also ignore the regularization term for now.

$$\frac{\partial}{\partial w_j} J(w) = \begin{cases} \alpha * y_i \cdot x_i & \text{if } y_i * w_j \cdot x_{ij} \leqslant 1 \\ 0 & \text{otherwise} \end{cases} \qquad (10)$$

The pseducode for computing gradient is shown in Algorithm 5.

4. IMPLEMENTATION

Why Python Recently, Python has become one of the most popular language in Machine Learning and Data Analytics. With lots of open-source libraries and frameworks being available, such as scikit-learn, Tensorflow, Pytorch and so on, it is much easier to develop sophisticated models in Python.

Therefore, we intend to replicate the Bismarck[8] using Python, which could enable more powerful machine learning models to be deployed for in-RDBMS analytics. PyBismarck, our Python version of Bismarck, has similar features and functions with Bismarck but only support two analytical tasks (Logistic Regression and SVM) so far.

We implement Mini-Batch Gradient Descent(MBGD) aggregation computation in RDBMS in similar manner with Bismarck. To take advantages of using Python, we then replace the gradient computation component and loss computation component with built-in models and functions in scikit-learn [6]. For comparison purpose, we also implement out-RDBMS MBGD to perform the same tasks and with the same tools.

All the algorithms presented in this section are implemented over PostgreSQL[5] with extension PL/Python [4] which is a procedural language allows PostgreSQL functions to be written in the Python.

4.1 PyBismarck IGD Implementation

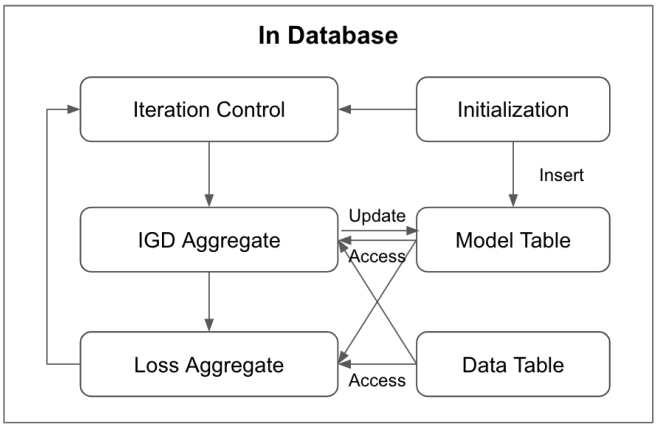

Figure 1: Overview of In-RDBMS IGD Implementation

The high-level architecture of PyBismarck is presented in **Figure** 1 which is a replication of Bismarck using Python. The PyBismarck takes the task, data set specified by end-user and allows any modification on the other parameter setting such as learning rate, iteration,etc. Then the system will initialize the model with user-given values. More specifically, a model table will be created based on the dimension of data set to contain all the parameters(mu, learning rate, and weights for model) to be updated during the IGD computation. After the initialization, the IGD aggregates component starts to look at each tuple in the data table in one iteration. With one access to a tuple, the gradient will be computed based the analytical task and then will be used to update the parameters contained in linear model table. After one iteration, the current weight will be read and used to compute loss in Loss Aggregate component.

For different analytical tasks, the objective function and gradient varies. Currently, PyBismarck support Logistic Regression and SVM only. In section 3, we have reviewed the IGD computation theoretically and Alogithm 3,5 shows more more details about how the IGD Aggregate component updates the parameters.

4.2 Mini-Batch Gradient Descent Implementation

Since IGD requires to update weights of models so frequently, the training process is more computationally expensive than BGD. The Mini-Batch Gradient Descent seeks the balance between BGD and IGD.

The General architecture is shown in **Figure** 2. The main logic of PyBismarck MBGD version is similar to the IGD version, while some modifications are required to enable the computation of gradients. As we know, Postgresql does not allow the functions to take multiple rows of data as input. To enable the calculation of the mini-batch gradient, we use "SFUNC" to define a transit function scanning the tuples within each batch and storing them in the internal state. Then, a function defined by "FINALFUNC" will be called to process the data batch in the internal state. With a similar idea, we implemented the loss aggregate function.

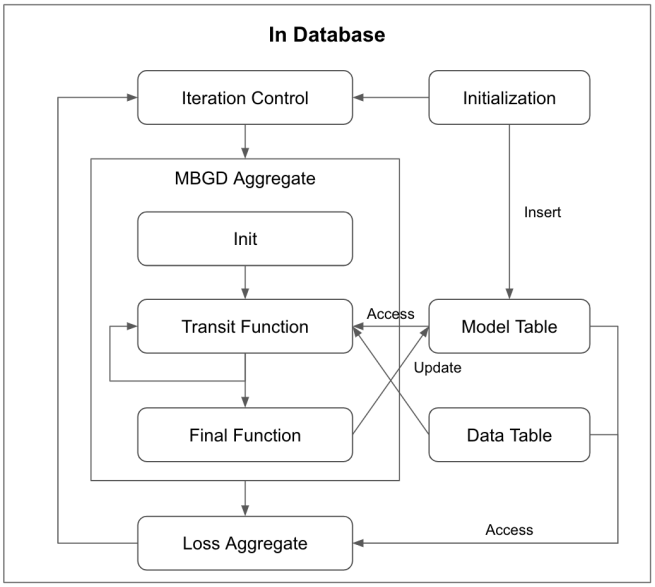

Figure 2: Overview of In-RDBMS Mini-Batch Gradient Descent Implementation

Algorithm 6 Mini-Batch GD

```
INITIALIZE
for iteration ← 0, .., T do
    for t^th batch with data (X^{t}, Y^{t}) do
        states ← []
        for Each tuple data x_i, y_i in the batch do
            state ← Transit(x_i, y_i)
        end for
        Final Aggregate(state)
    end for
    Update parameters in model table
    Loss Aggregate(state)
end for
function TRANSIT(x_i, y_i)
    state ← state + [x_i + y_i]
end function
function FINAL AGGREGATE(state)
    Gradient computation with batch data state
end function
function LOSS AGGREGATE(teststate)
    Loss computation with partial batch data teststate
end function
```

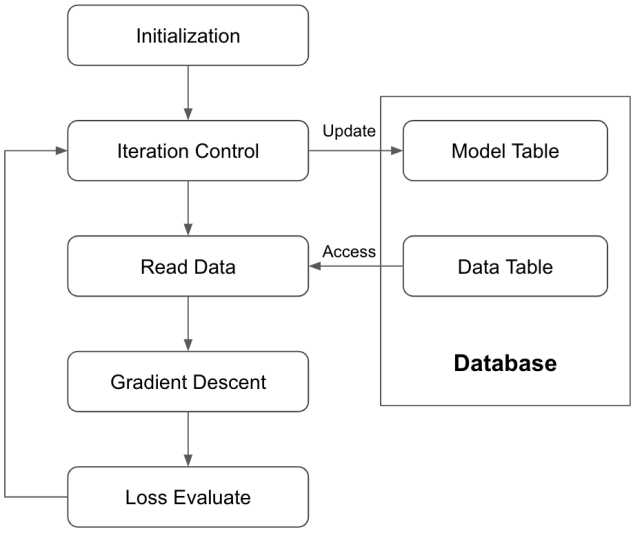

Figure 3: Overview of Out-RDBMS Mini-Batch Gradient Descent Implementation

4.3 Mini-Batch Gradient Descent Out-RDBMS Implementation

We want to verify how much the advantages the in-database analytics may bring based on a unified architecture comparing with the out-database analytics. As a comparison group, we implement out-RDMS version to accomplish the same tasks and with scikit-learn package.

To avoid direct reading the whole large data table from RDBMS which consumes to much time and memory, the system will access a batch of data from RDBMS each time and process Mini-Batch Gradient Descent(MBGD) computation to updates parameters following the same aggregate manner as in section 4.2 by using scikit-learn package built-in models and functions. The General architecture is shown in **Figure** 3.

5. EXPERIMENTS

In our experiments, we first verify the main statements made from Bismarck's experimental results [8] by running their source code[3]. Then, we conduct experiments on the same data set with PyBismarck. A series of experiments are intended to show the impact of implementing language on the in-RDBMS analytics and the benefits may bring by using in-RDBMS analytical tools

5.1 Data Source

In this project, we use the same two publicly available real-world datasets as in Bismarck's paper.[8] as shown in Table 1. The Forest dataset is dense with all feature vectors stored in one column for a tuple. Since the DBLife dataset is sparse, it stores a list of (key, value) tuple where the key is the position in feature vector containing a value and the rest of positions are all zeroes.

Table 1: Dataset Statistics

Dataset	Dimension	Examples	Size
Forest (Dense)	54	581k	77 M
DBLife (Sparse)	41k	16k	2.7 M

5.2 Replicate Experimental Results from Bismarck

In the Feng.etl's work[8], the main contribution regarding their empirical studies is that the order data is stored and parallelization of computation on a single node multicore RDBMS are two key factors influencing performance. In this section, we verify how the data order and parallelism impact performance by running the source code on the same datasets.

5.2.1 Impact of Data Ordering

As stated in the paper[8], the order of data stored affects performance, for example, if all of the positive examples come before the negative examples, the resulting convergence rate may be much slower than if the data were randomly ordered. A common approach to gradient descent methods is to shuffle training example before each iteration begins. Based on their experiments, they find that across a broad range of models, while shuffling one has a slightly slower convergence rate than shuffling on each pass. The reason behind this is that the lack of expensive reshuffling reduces the overhead caused by shuffling and thus allows to run more iterations in the same amount of time. Thus, we also choose to shuffle once before the first iteration instead of shuffling always.

To show the impact of ordering, we conduct experiments on the same datasets ordered by labels in four different methods, which are random shuffle:shuffle once before the first iteration begins, DESC order: all positive examples come before negative examples and no shuffle after, ASC order: all negative examples come before positive examples and no shuffle after, non-shuffle: the first iteration starts with original order. We compare the average running time for each iteration and the number of iterations to converge. From the experimental results, we find that the order does not affect the running time of iteration and random shuffle even needs more time sine the overhead caused by shuffling. However, data ordering does influence the convergence rate. As shown in Table 3, the datasets in descending order of labels require the almost two times iteration than in random order to converge. Ascending order of labels has slightly better performance but also converge slower than the shuffle order. Since the original datasets do not distribute in any pattern, the performance of non-shuffle is similar to shuffle once.

Table 2: Running Time for Each Iteration of Bismarck

Dataset	Task	Random Shuffle	DESC Order	ASC Order	Non Shuffle
Forest (Dense)	LR SVM	1.2 s 1.08 s	1.15 s 0.98 s	1.15 s 0.99 s	1.06 s 0.92 s
DBLife (Sparse)	LR SVM	0.025 s 0.022 s	0.024 s 0.019 s	0.023 s 0.02 s	0.02 0.018 s

Table 3: Number of Iterations before Convergence of Bismarck

Dataset	Task	Random Shuffle	DESC Order	ASC Order	Non Shuffle
Forest (Dense)	LR SVM	4 13	4 39	4 14	4 13
DBLife (Sparse)	LR SVM	48 61	70 87	64 87	47 62

5.2.2 Impact of Memory Sharing

In Feng's work[8], they state that the shared memory version converges faster than the shared-nothing version. We replicate the experiments and get the results as shown in Table 4. We find that memory sharing does result in slightly faster performance for each model on each dataset, but the improvement is not that obvious. For total of 100 iterations, the benefit could be brought by parallelism is less than 10 seconds in total.

Table 4: Running Time Comparison for Shared Memory

Dataset	Task	Pure UDA	Shared Memory
Forest (Dense)	LR SVM	1.2 s 1.08 s	1.15 s 1.0 s
DBLife (Sparse)	LR SVM	0.025 s 0.022 s	0.024 s 0.02 s

5.3 Baseline Comparison

In this section, we intend to compare the performance of PyBismarck with Bismarck. Then we will analyze the impact of the different gradient descent methods and hyper-parameter choosing on performance. Finally, we compare the MBGD version of PyBismarck with out-RDBMS Python implementation of MBGD.

5.3.1 Impact of Implementing Language

We first compared the end-to-end runtimes of PyBismarck and Bismarck on LR and SVM. The results are summarized in Table 5. Overall, we see that Bismarck has faster performance overall tasks against PyBismarck, especially on the sparse dataset. We can conclude the advantage of utilizing C instead of Python when implementing aggregate functions in RDBMS.

Table 5: Running Time Comparison for Bismarck and PyBismarck

Dataset	Task	Bismarck	PyBismarck IGD
Forest (Dense)	LR SVM	1.2 s 1.08 s	89.3 s 80.9 s
DBLife (Sparse)	LR SVM	0.025 s 0.022 s	295.8 s 281.7 s

5.3.2 Impact of Gradient Descent Methods

To improve the execution time of PyBismarck, we implemented an MBGD version as well, to reduce the iterations of data processing. The results are as shown in Table 6. We used out-RDBMS MGDB implemented as a baseline in the comparison. We found that MBGD could accelerate the

training process and in-RDBMS implementation was faster than out-RDBMS.

Although it was still slower than C-implemented Bismarck, PyBismarck with MBGD already made great progress. Moreover, compared with Bismarck, it is much easier to deploy a new and advanced model using PyBismarck. It is also easier to control and tune the training procedure.

Table 6: Running Time Comparison for PyBismarck IGD and MBGD

Dataset	Task	PyBismarck IGD	PyBismarck MBGD	MBGD Out-RDBMS
Forest (Dense)	LR	89.3 s	19.2 s	39.7 s
	SVM	80.9 s	18.84 s	38.8 s

5.3.3 Impact of Batch Size

When we experimented with MBGD, we also found that the performance was highly dependent on the batch size. Generally speaking, the larger the batch size was, the faster the training process would be, as shown in Table 7.

Table 7: Running Time with Different Batch Sizes

Batch Size	50000	100000	500000
Forest (Dense) LR	43.2 s	19.2 s	7.9 s

6. CONCLUSIONS AND DISCUSSION

In this project, we experimented with Bismarck [8] and validated their results. We found that the order of data did matter in the training process, and shared memory could help improve the execution time a little. We also found that the performance of Bismarck is not that much stable. In some cases, we find the range of oscillation even increases with iteration grows. We think that is caused by the one-shuffle operation. Moreover, for in-RDBMS analytics, it's easy to apply the supported models on the training data but it is hard to investigate the change of loss and prediction performance on test data

To extend Bismarck to more advanced machine learning models, we developed PyBismarck in Python based on their work. We implemented the IGD version and MBGD version of PyBismarck and compared it with out-RDBMS Python implementation of MBGD. We found that PyBismarck can replicate the function of Bismarck but the execution time would be much slower. We improved the execution time by implementing MBGD as weights update method, and it would be dependent on the batch size.

Our work of PyBismarck MBGD demonstrated the possibility of leveraging open-source Python libraries in in-RDBMS analytics which would enable fast deployment of advanced machine learning models. In this project, we only tested LR and SVM with small datasets. In the future, more work needs to be done to investigate more into the pros and cons of in-RDBMS and out-RDBMS analytics, and the performance of deploying advanced models.

References

[1] Microsoft sql server 2008 r2 data mining. (2019).

[2] Oracle data mining.

[3] Bismarck source code. `http://i.stanford.edu/hazy/victor/bismarck-download/`, 2019.

[4] Pl/python extension. `https://www.postgresql.org/docs/11/plpython.html`, 2019.

[5] Postgresql 11.4 documentation. `https://www.postgresql.org/docs/`, 2019.

[6] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[7] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: New analysis practices for big data. *Proc. VLDB Endow.*, 2(2):1481–1492, Aug. 2009.

[8] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336, New York, NY, USA, 2012. ACM.

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

[10] D. Karimi and R. K. Ward. A hybrid stochastic-deterministic gradient descent algorithm for image reconstruction in cone-beam computed tomography. *Biomedical Physics & Engineering Express*, 2(1):015008, feb 2016.

[11] Z.-Q. Luo. On the convergence of the lms algorithm with adaptive learning rate for linear feedforward networks. *Neural Comput.*, 3(2):226–245, June 1991.

[12] O. Mangasarian and M. Solodov. Serial and parallel backpropagation convergence via nonmonotone perturbed minimization. *Optimization Methods and Software*, 4(2):103–116, 1994.

[13] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[14] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM J. on Optimization*, 19(4):1574–1609, Jan. 2009.

[15] F. Niu, B. Recht, C. Re, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 693–701, USA, 2011. Curran Associates Inc.

[16] L. Zhi-Quan and T. Paul. Analysis of an approximate gradient projection method with applications to the backpropagation algorithm. *Optimization Methods and Software*, 4(2):85–101, 1994.