MorpheusPy: Factorized Machine Learning with NumPy

Side Li Arun Kumar
Universtiy of California, San Diego
La Jolla, California
{s7li,arunkk}@eng.ucsd.edu

ABSTRACT

Factorizing machine learning (ML) over relational databases is an approach to avoid storage wastage and runtime inefficiency in the ML computations. Some recent works generalized factorized ML to any ML computations expressible in the formal language of linear algebra (LA). They achieved this by creating a framework of algebraic rewrite rules to push individual LA operations through joins. Thus, any ML algorithm expressible as a vectorized LA program can be automatically "factorized." However, they only implemented the prototype in R while recently Python has become the dominating library in ML analytics. In this paper, We explore how to extend the idea of factorized LA to Python and NumPy. We compare implementations in R and Python. To avoid overhead, we introduce low-level C++ rewrites for some LA operators. We show that with NumPy automatic LA rewrite can also achieve order-of-magnitude speedups given a few optimizations. We also experiment with sparse matrices whose result shed insights on the impact of sparsity on LA rewrites.

1 INTRODUCTION

Many relational datasets in the real-world are multi-table, but most machine learning (ML) toolkits expect the training dataset to be a single table with all the features. This forces data scientists to often perform joins to materialize a single table before ML that concatenates features from all base tables. Alas, such joins could introduce redundancy in the data due to repetition of records, which could blow the data up in size. This could lead to both storage wastage and runtime inefficiency due to redundancy in the ML computations. In recent work, Kumar et al. [5] introduced the paradigm of "factorized ML" to mitigate this issue by showing how to decompose ML computations and push them through joins to the base tables. This idea enabled them to run "ML over joins" and avoid the need to always materialize joins. However, their work was limited to a only one class of ML algorithms and required a manual rewrite of the ML implementation, which led to a daunting development overhead for data scientists interested in using this idea for other ML algorithms. Consequently, to mitigate this development overhead, Chen et al.[2] generalized factorized ML to any ML computations expressible in the formal language of linear algebra (LA). They achieved this by creating a framework of algebraic rewrite rules to push individual LA operations (e.g., matrix-vector multiplication) through joins. Thus, any ML algorithm expressible as a vectorized LA program can be automatically "factorized." They prototyped factorized LA in R [7], which is a popular environment for ML analytics, and demonstrated substantial runtime speedups on real-world datasets.

Increasingly, Python and ML systems in Python such as TensorFlow [1] and Scikit-Learn [6] are becoming more popular than R for ML analytics tasks, especially in Web and software companies. NumPy is the underlying LA library in Python but TensorFlow's architecture and integration of NumPy is quite different from R. [8] Thus, in this project, we explore how to extend the idea of factorized LA to this new and important ML environment. In summary, we examine rewrite rules in Python and experiment with performance result. To further optimize performance, we make changes to the previous factorized data structure.

Outline. The rest of the paper is organized as follows: In Section 2, we explain the differences between NumPy implementation and previous R prototype. In Section 3, we illustrate the implemented details of operators. In Section 4, we introduce a few data prep operations developed along with NumPy. In Section 5, we validate NumPy implementation with synthetic and real data sets. In Section 6, we discuss limitations of MorpheusPy.

2 KEY DIFFERENCE

R and Python are two different languages with different principles and domain usages. The difference leads us to additional considerations in the implementation. Originally, NumPy is a library designed to be "R" in Python. However, it has diverged from R in the way to accommodate Python's features. Some designs work perfectly in R would cause significant performance penalty in NumPy. As such, we propose a sightly different design for MorpheusPy.

2.1 Represent K matrix as Array

In our previous work, we have a sparse matrix K to store join information between entity and attribute tables. This K matrix plays a critical role in rewriting operators, by which we avoid redundant computation. Note that K as a complementary matrix also brings overhead in space and runtime. Nonetheless, the performance benefit from factorized rewriting dwarfs the overhead in R implementation as reported by our previous work.

Unfortunately, NumPy does not have native implementations of sparse matrices. The most viable option on the plate would be sparse matrices provided by SciPy [4]. According to our early stage of experiments, this library causes much more overhead compared to R implementation especially in matrix multiplication operators such as LMM and RMM. As such, we decide to use NumPy arrays to represent the join information. For example, if the i-th row of entity matrix is joined with the j-th row of attribute matrix, the element stored in K array at index i will just be j.

With this form of representation, we have gained a few benefits. The first one is most obvious: the array representation saves space. Without additional data structure, we only store values linearly in memory. Another improvement is from performance, since K array also helps avoid unnecessary computation. Given K as a sparse matrix, matrix-matrix multiplication would still be expensive because of dimensions of K. However, with K as an array we can use each element to directly slice matrices which only involves

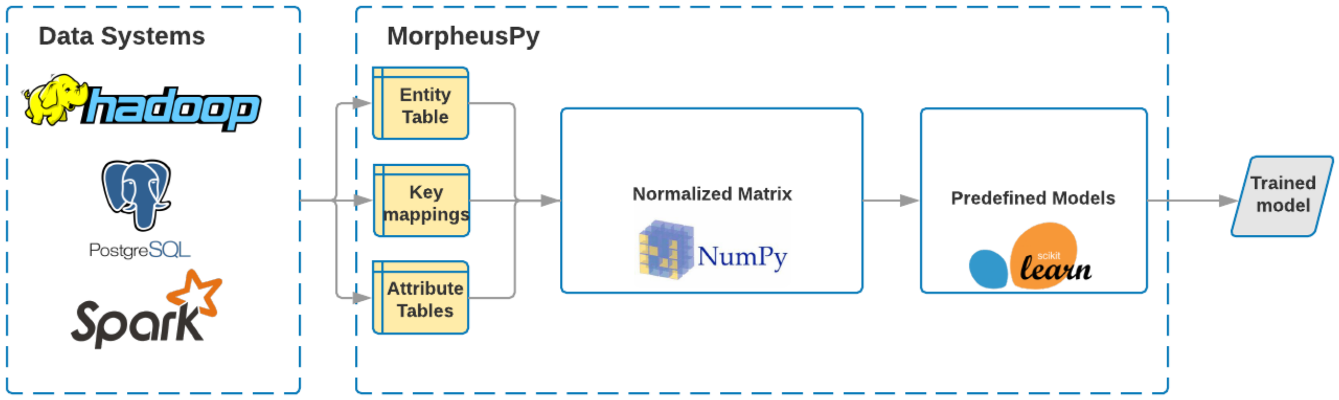

Figure 1: MorpheusPy Overview

memory fetching and most straightforward calculation. Lastly, it is easier to prepare K array which saves efforts during the stage of data collection.

2.2 Consideration of matrix sparsity

In our previous work, we only considered dense matrix and accordingly runtime analysis. However, a lot of data sets are in the format of sparse matrices. In SciPy library, there are multiple sparse matrix implementations as described in Table 1:

Table 1: List of SciPy Sparse Matrices

Sparse Matrix Type	Description
bsr_matrix	Block Sparse Row matrix
coo_matrix	A sparse matrix in COOrdinate format
csc_matrix	Compressed Sparse Column matrix
csr_matrix	Compressed Sparse Row matrix
dia_matrix	Sparse matrix with DIAgonal storage
dok_matrix	Dictionary Of Keys based sparse matrix
lil_matrix	Row-based linked list sparse matrix

Optimizations on sparse matrix can be a potentially significant amount of work, so we only consider COO in this project. From our observations, the more sparse input matrices are, the less speedup we can achieve from factorized rewriting because the overhead from the library and language will be relatively more significant. To further theorize the impact of sparsity, we bring in a new factor called p in addition to runtime complexity analysis of the previous work.

2.3 Low-level rewrite in C++

In our NumPy implementation, we get our hands dirty with low-level optimizations in C++[9]. Initially, we have a version of pure high-level python version implemented with only high-level NumPy APIs that produces a significantly worse result than those reported in our previous work. This performance degrade motivates us to dive into low-level optimization to minimize overhead caused by the language and LA library. We only rewrite the part that is inherently slow if implemented in Python. Further detail of C++ rewrite is summarized in figure 2 and will be discussed in section 3.

3 IMPLEMENTATION

MorpheusPy is an implementation of normalized matrix (NM) using APIs provided by NumPy. Our class has S as entity matrix, R as a list of attribute matrix, and K as a list of join information arrays. Speaking of matrices, we support NumPy matrix, ndarray as well as SciPy's COO sparse matrix. Our NM is a subclass of NumPy matrix, by which NM can be directly used with many existing algorithms that support NumPy. We follow the design principle in constructing the class as well as defining universal functions [10].

3.1 Factorized Operators

3.1.1 Element-wise Scalar Operators. Scalar operators in MorpheusPy is same as R prototype. The rewrite rules are:

$$T \oslash x \rightarrow (S \oslash x, K, R \oslash x); x \oslash T \rightarrow (x \oslash S, K, x \oslash R)$$

$$f(T) \rightarrow (f(S), K, f(R))$$

We perform functions only on entity and attribute matrices.

3.1.2 Left Matrix Multiplication (LMM). In previous work, we have a rewrite rule as below:

$$TX \rightarrow SX[1:d_S,] + K(RX[d_S+1:d,])$$

Define $RX[d_S+1:d,]$ as V, and $SX[1:d_S,]$ as W. Note that K is a sparse matrix, and therefore KV is a matrix-matrix multiplication. This step is essential because lazy valuations of matrices happen here, yet it is pure overhead brought by our rewrite rules. The fundamental purpose of this step is to expand a matrix of dimension $n_R \times d_X$ to a matrix of dimension $n_S \times d_X$. In Morpheus, we opt to store K as arrays, so we define a new helper function *expand_add* to evaluate this step:

$$TX \rightarrow expand\_add(SX[1:d_S,], RX[d_S+1:d,], K)$$

$$\rightarrow expand\_add(W, V, K)$$

In fact, *expand_add* finds the corresponding i-th row in V and adds the row to j-th row of W. The helper function is implemented in

C++ directly considering memory access with low-level language is very fast. In C++, each column of V is scanned to favor memory layout pattern because V in most cases has fewer columns than rows.

3.1.3 Right Matrix Multiplication (RMM). RMM in previous work has rewrite rules as:

$$XT \to [XS, (XK)R]$$

Similar to LMM, XK is an expensive matrix-matrix multiplication. Essentially, this step compresses a matrix of dimension $n_X \times n_S$ to a matrix of dimension $n_X \times n_R$. In MorpheusPy, we rewrite this part in a function called *group* in C++:

$$XT \to [XS, group(X, K)R]$$

group creates an empty matrix($n_X \times n_R$), and loops through each row of X to add the i-th row to the corresponding j-th row in the empty matrix according to K array. The entire process is mostly memory access and simple arithmetic calculations.

3.1.4 Aggregation Operators. In our previous work, aggregation rewrite rules are:

$$rowSums(T) \to rowSums(S) + KrowSums(R)$$
$$colSums(T) \to [colSums(S), colSums(K)R]$$
$$sum(T) \to sum(S) + colSums(K)rowSums(R)$$

Since in MorpheusPy we opt to use K as arrays, we no longer have in-built colSum function for K. Hence, we have a sightly new design (define X as a $1 \times n_R$ vector):

$$rowSums(T) \to rowSums(S) + rowSums(R)[K]$$
$$colSums(T) \to [colSums(S), group(X, K)R]$$
$$sum(T) \to sum(S) + group(X, K)rowSums(R)$$

We create a vector to pass into group function which will compress the vector. Essentially, we create colSum on the fly directly with arrays.

3.1.5 Cross-product. The previous work has rewrite rules as follow:

$$P = R^T(K^T S)$$
$$\begin{bmatrix} crossprod(S) & P^T \\ P & crossprod((diag(colSums(K)))^{\frac{1}{2}}R) \end{bmatrix}$$

Cross product in NumPy is different from R's in the sense that there is no optimized *crossprod* function. In other words, we have to manually invoke cross product by first transpose and then matrix multiplication. Also, $crossprod((diag(colSums(K)))^{\frac{1}{2}}R)$ is expensive since diag creates a very high dimension matrix that will also be multiplied with R. As such, MorpheusPy has modified rewrite rules in accordance to performance and K array (X is a $1 \times n_R$ matrix) :

$$P = R^T group\_left(S, K)$$
$$\begin{bmatrix} S^T S & P^T \\ P & crossprod(multiply(group(X, K)R)) \end{bmatrix}$$

First of all, we implement $K^T S$ in C++ as a function called *group_left*, because this part essentially compresses S matrix. We also completely rewrite $(diag(colSums(K)))^{\frac{1}{2}}R$. *group* is equivalent to *colSums* on K matrix. Creating a diagonal matrix and then square root is equivalent to mapping and sqrt each row in R matrix to a new row based on last step's result. This remapping is very fast in C++, as each entry is mapped at a granularity of memory addresses. Also, we optimize parts for sparse matrix by looking at coordinates.

3.1.6 Matrix Inverse Operators. Matrix inverse in MorpheusPy is the same as R implementation because we merely call factorized operators that we have specified in the previous section:

$$ginv(T) \to ginv(crossprod(T))T^T, if d < n$$
$$ginv(T) \to T^T ginv(crossprod(T^T)), o/w$$

Table 2: C++ Operators

Operator	Main Usage	Description
expand_add	LMM	expand_add(W, V, K) finds the corresponding i-th row in V and adds it to j-th row of W. i-j mapping is stored in K with d_V greater than d_W.
group	RMM	group(E, X, K) finds the i-th row of X to the corresponding j-th row of E (empty matrix). i-j mapping is stored in K with n_X greater than n_E. It is equivalent to XK.
group_left	crossprod	group_left(E, S, K) finds the i-th row of S to the corresponding j-th row of E (empty matrix). i-j mapping is stored in K with n_S greater than n_E. It is equivalent to $K^T S$.

3.2 Runtime Complexity Analysis

We present a new runtime complexity analysis with sparsity and k array overhead being considered. The result is shown in Table 3. One fact we can easily see is that the arithmetic overhead brought by K grows proportionally to d_X and n_S. In previous work, we have shown that the speed-ups converge to $1 + FR$ as TR approaches infinity. However, we have learned in MorpheusPy that sparsity will affect the result dramatically. Assume all S and K have uniform sparsity p. Then, with TR going to infinity, most operators converge to a speedup of $(1 + FR)\frac{ed_S}{ed_S+1}$. The more features in entity matrix, the better speedup we have achieved. The more sparse matrices are, the more overhead we have in comparison to the total amount of arithmetic calculation.

4 DATA PREPARATION OPERATIONS

In addition to operators, we now have helpful utility functions implemented by a factorized means.

4.1 Mean Centering

We implement mean centering as follow:

$$matrix - mean(matrix)$$

mean is calculated independently with entity matrix S and attribute matrices R. With the help of K array, we employ similar rewrite

Table 3: Arithmetic computations of the standard algorithms and factorized ones

Operator	Standard	Factorized with sparsity
Scalar Op	$n_S(d_S + d_R)p_T$	$n_S d_S p_S + n_R d_R p_R$
Aggregation		
LMM	$d_X n_S(d_S + d_R)p_T$	$d_X(n_S d_S p_S + n_R d_R p_R) + d_X n_S$
RMM	$n_X n_S(d_S + d_R)p_T$	$n_X(n_S d_S p_S + n_R d_R p_R) + n_X n_S$
crossprod	$(d_S + d_R)^2 p_T^2 n_S$	$d_S^2 p_S^2 n_S + d_R^2 p_R^2 n_R$ $+ d_S d_R n_R p_S p_R$

ideas as described in section 3 to expand R to calculate the mean. Mean can either be the mean of the entire matrix, or the mean of columns. The rewrite rule of mean is described as follow:

Mean:

$$sum(matrix)/(matrix.shape[0] * matrix.shape[1])$$

Column Mean:

$$[mean(S, axis = 0), mean(R[K], axis = 0)]$$

The scalar subtraction is then handled by our normalized operator.

4.2 Normalization and Standardization

Normalization and standardizations are two approaches to transform the matrix into a consistent fashion. We implement these two functions with the help of mean_centering, min, max and std as follow:

Normalization:

$$mean\_centering(matrix)/(max(matrix) - min(matrix))$$

Standardization:

$$mean\_centering(matrix)/std(matrix)$$

max and min are trivial to implement as we simply iterate over every entry in each factorized matrix to find maximum and minimum values. *Std* is implemented with the help of *mean* and factorized scalar operators as: $sqrt(mean((matrix - mean(matrix))^2))$.

4.3 Imputation

Imputation is an approach to replace missing values in input tables. In MorpheusPy, we implement a simple imputation, that is, replacing missing values with means. We depend on NumPy's native helper functions about nan to find nan positions in matrices, to create masks in imputation.

5 PERFORMANCE

To measure the performance of MorpheusPy in comparsion to R prototype, we redo experiments on synthetic datasets and real-world datasets.

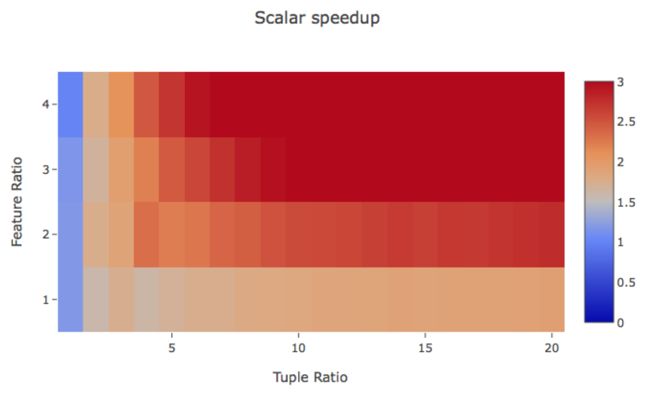

Figure 2: Speed-ups of Scalar operator

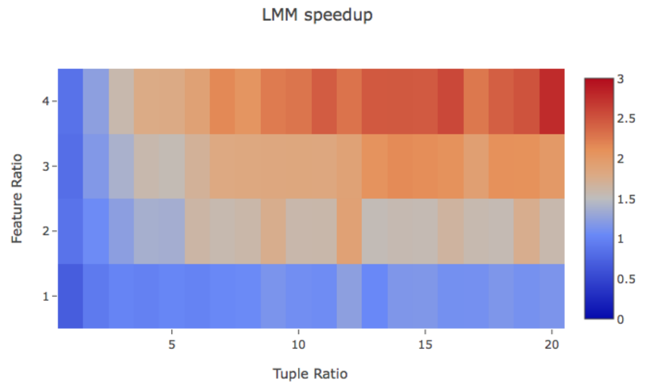

Figure 3: Speed-ups of LMM operator

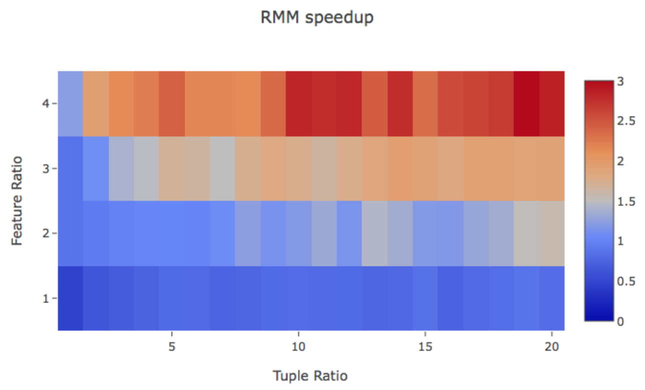

Figure 4: Speed-ups of RMM operator

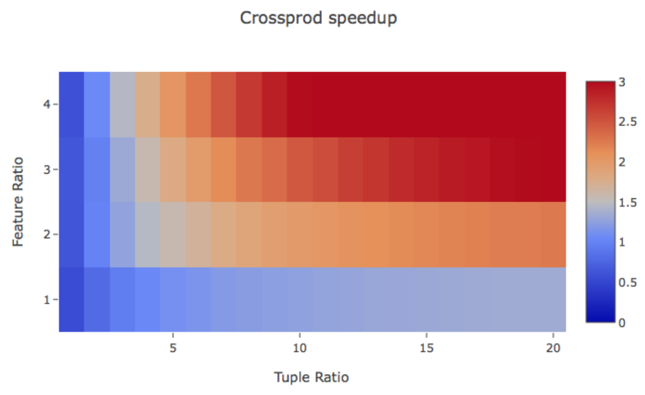

Figure 5: Speed-ups of Crossprod operator

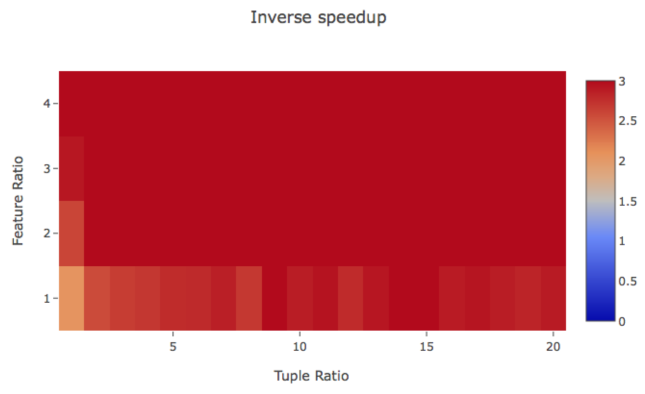

Figure 6: Speed-ups of Inverse operator

5.1 Experiment setup

All experiments were run on CloudLab [3]. All experiments were conducted on a machine with 14 Intel Xeon E5-2683 2.0GHz cores, 224GB RAM and 3TB disk with Ubuntu 16.04 LTS as the OS. Our code is implemented in Python 2.7, and we use NumPy 1.13, SciPy 1.1 as dependency libraries. Our C++ is compiled with gcc 5.4.0 20160609.

5.2 Operator-level Result on Synthetic Data

In the experiment, we fix n_R as 10^6 and d_S as 20. We only vary feature ratio and tuple ratio. We conduct each experiment on operators for ten times and subtract min/max value to get a stable result. Also, we generate a synthetic dataset for each run to avoid memory caching problem. We present the result as heat maps in Figure 2, 3, 4, 5 and 6.

Overall, on dense synthetic data, MorpheusPy achieves operator speedups close to these of R prototype. For scalar (add) operator, MorpehusPy speedups increase smoothly by tuple ratio and feature ratio with a maximum speedup of 3x. Speedups are significant when tuple ratios are large. LMM shares a similar speedup distribution with scalar, though the speedups are more variable in proportion to tuple/feature ratios. For RMM, it also has a similar speedup distribution as RMM. One interesting difference is that RMM speedups are only substantial when feature ratio is higher than 2. The same phenomenon applies to Crossprod; speedups are more considerable when feature ratios are greater than 2. For Inverse, even with feature ratio = 1 and tuple ratio = 1, the speedup is close to 3x. For larger feature/tuple ratios, speedups are also much more substantial.

5.3 ML Algorithm-level Results on Synthetic Data

We examine common ML algorithms including linear regression(LS), logistic regression(LR), K Means and GNMF. Algorithms are implemented following SciKit Learn style and can be easily extended to more complicated models. In this experiment, we keep the same configuration from last section by fixing n_R as 10^6 and d_S as 20. We present the result in Figure 7, 8, 9 and 10.

Overall, we find MorpheusPy sees significant speedups in ML algorithms when both feature and tuple ratios in synthetic datasets are large. For LS, speedups are only higher than two if feature ratio is greater than 2. Tuple ratios seem to have less effect on speedups as speedups for LS are always around 3x when feature ratio is greater than 2. For LR, the distribution of speedups is similar to that of LS except that the maximum speedup is only about 2x. For K Means and GNMF, we find speedups are proportional to tuple ratio and feature ratio.

5.4 ML Algorithm-level Results on Real Datasets

We also test MorpeusPy with simple ML algorithms on seven real-world datasets (Movie, Yelp, Expedia, Walmart, LastFM, Book and Flights). All datasets are in a sparse matrix format, so we import them as SciPy's COO sparse matrix to be consistent. We present the speedups in comparison to R prototype in table 4, and the wall clock runtime of factorized implementations in table 5.

Overall, we find that MorpheusPy sees significantly lower speedups than MorpheusR for LS on the first 4 datasets but comparable speedups on the remaining 3. However, for LR, except on Flights, MorpheusPy speedups are lower than MorpheusR. But interestingly, for K-Means, except on Expedia, MorpheusPy speedups are all higher than MorpheusR. The situation is more mixed across datasets for GNMF. We suspect that the difference in speedups boils down to language and NumPy library overhead. To further verify our intuition, we conduct fine-grained experiment on one dataset with all four algorithms.

Meanwhile, we find that MorpheusPy has a faster absolute runtime for all algorithms in all datasets. For LS and LR, MorpheusPy achieves 3x to even 12x reduction in absolute runtime, except that interestingly it only has 2x speedup in runtime cost for Expedia. For K Means and GNMF, we still see a significant decrease in total runtime in MorpheusPy, with most having a 3x in reduction.

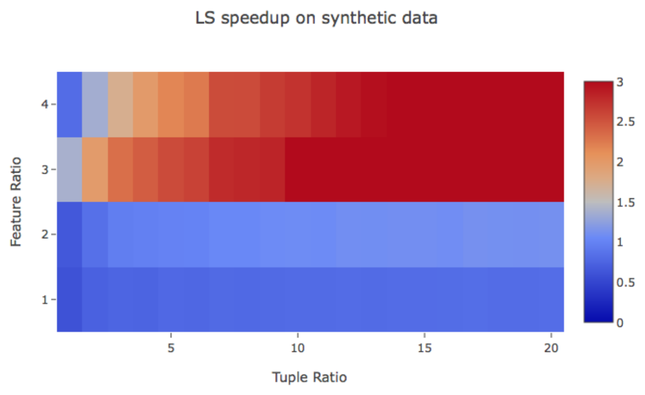

Figure 7: Speed-ups of Linear Regression

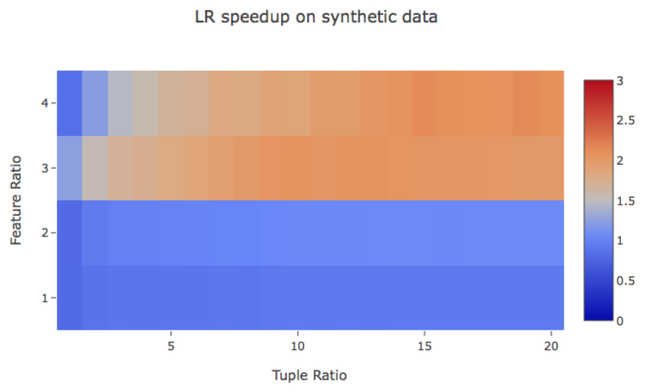

Figure 8: Speed-ups of Logistic Regression

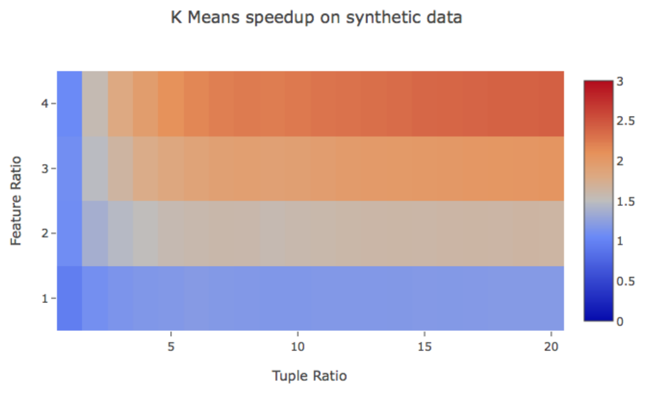

Figure 9: Speed-ups of K Means

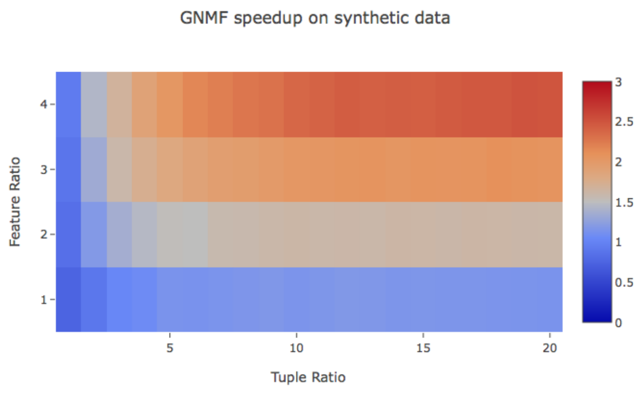

Figure 10: Speed-ups of GNMF

Table 4: ML Algorithm-level Speedups

MorpheusPy				
	LS	LR	K Means	GNMF
Movie	19.4	8.7	8.5	5.8
Yelp	11.7	6.2	6.1	5.1
Expedia	3.0	4.0	3.8	2.9
Walmart	4.9	3.1	4.1	4.2
LastFM	10.2	3.2	3.4	3.7
Book	5.6	2.2	2.1	2.4
Flights	4.4	4.0	5.3	4.7
MorpheusR				
	LS	LR	K Means	GNMF
Movie	36.3	30.3	6	8
Yelp	36.4	30.1	6.1	12
Expedia	22.2	14	4.5	5.9
Walmart	10.9	9.8	2.0	2.8
LastFM	11.0	8.7	2.3	3.4
Book	5.2	3.9	1.3	1.4
Flights	4.4	3.4	1.8	2.0

5.5 Case Study: Movie Dataset

We further decompose four algorithms into consecutive steps, and measure the runtime cost of each step with both materialized and normalized approach.

5.5.1 Linear Regression. Algorithm 1 and 2 are original and stepped linear regression. We decompose a single line of Python code into five phases in order to investigate the performance of MorpheusPy at a fine granularity. Line 3 and 7 are LMM/RMM that can be normalized, which we should expect to see substantial

Table 5: ML Algorithm-level Absolute Runtime (second)

	LS	LR	K Means	GNMF
MorpheusPy				
Movie	0.3	0.7	3.2	3.0
Yelp	0.2	0.3	1.2	0.9
Expedia	2.1	2.6	9.0	7.3
Walmart	0.1	0.4	1.2	1.2
LastFM	0.2	0.3	1.2	1.0
Book	0.2	0.3	1.1	0.9
Flights	0.1	0.1	0.3	0.3
MorpheusR				
Movie	1.9	2.3	8.6	9.0
Yelp	0.6	0.7	2.4	1.9
Expedia	3.3	3.9	13.2	13.9
Walmart	1.2	1.5	5.0	4.9
LastFM	0.8	0.9	3.3	2.9
Book	0.8	0.9	3.7	2.8
Flights	0.3	0.4	1.0	0.8

speedups. Line 9 and 11 should be trivial as matrices have very low dimensions. We run the algorithm for 20 iterations with initial *gamma* = 0.000001 and random *w*. We take average of the runtime cost of each step in the loop. The result of stepped runtime cost is presented in Figure 11.

Algorithm 1: Linear Regression

```
1 def linear_regression(X, y, w, iterations, gamma):
2     for _ in range(iterations):
3         w -= gamma * (X.T * (X * w - y))
```

Algorithm 2: Linear Regression in Steps

```
1 def linear_regression(X, y, w, iterations, gamma):
2   for _ in range(iterations):
3       tmp0 = X * w
4       # X * w - y
5       tmp1 = tmp0 - y
6       # X.T * (X * w - y)
7       tmp2 = X.T * tmp1
8       # gamma * (X.T * (X * w - y))
9       tmp3 = gamma * tmp2
10      # w -= gamma * (X.T * (X * w - y))
11      w -= tmp3
```

Overall, the stepped result meets our expectation that the runtime costs of Line 3 and 7 have been reduced significantly. Note that

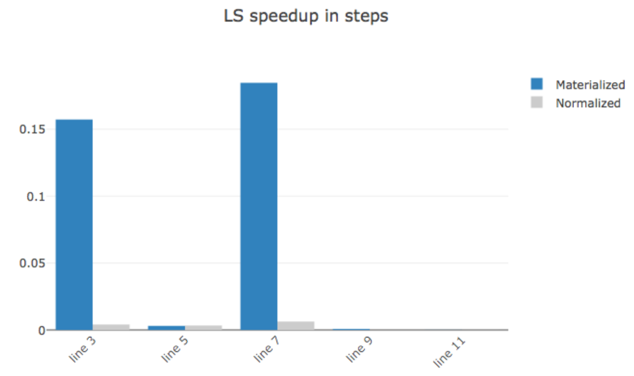

Figure 11: Runtime cost of Linear Regression

the runtime costs of Line 3 and 7 still dominate the total runtime of the algorithm. Only if we could further reduce the LMM cost of Line 3 and 7 can we achieve the speedup of 30x in R. As such, we conclude that LMM with sparse matrices is the bottleneck that results in worse speedup in linear regression.

5.5.2 Logistic Regression. Algorithm 3 and 4 are original and stepped logistic regression. We decompose a single line of Python code into seven phases in order to investigate the performance of MorpheusPy at a fine granularity. Line 3 and 11 are LMM/RMM that can be normalized, which we should expect to see substantial speedups. Line 5, 7, and 9 are parts that will not be normalized, which should also account for part of total runtime cost. We run the algorithm for 20 iterations with initial *gamma* = 0.000001 and random *w*. We take average of the runtime cost of each step in the loop. The result of stepped runtime cost is presented in Figure 12.

Algorithm 3: Logistic Regression

```
1 def linear_regression(X, y, w, iterations, gamma):
2     for _ in range(iterations):
3         w -= gamma * (X.T * (y / (1 + np.exp(X * w))))
```

Interestingly, in Normalized approach, the runtime cost of Line 5 account for more than $\frac{1}{2}$ of the total runtime cost. It means that even if we take out the runtime cost of all other steps, the maximum speedup we can achieve is at most 15x. Line 5 is simply an exponential function in NumPy, which obviously implies that exponential in NumPy is costly bringing too much overhead. Hence, we attribute that the worse speedup in logistic regression to the overhead of NumPy.

5.5.3 K Means. Algorithm 5 and 6 are original and stepped K Means. We decompose the algorithm into fifteen phases in order to investigate the performance of MorpheusPy at a fine granularity. Line 7 and 13 are scalar operations. Line 18 and 32 are LMM/RMM that can be normalized, which we should expect to see substantial speedups. Line 20, 22, 29, and 30 are parts that will not be normalized, which should also account for part of total runtime cost. We

Algorithm 4: Logistic Regression in Steps

```
1  def linear_regression(X, y, w, iterations, gamma):
2    for _ in range(iterations):
3      tmp0 = X * w
4      # np.exp(X * w)
5      tmp1 = np.exp(tmp0)
6      # 1 + np.exp(X * w)
7      tmp2 = 1 + tmp1
8      # y / (1 + np.exp(X * w))
9      tmp3 = y / tmp2
10     # X.T * (y / (1 + np.exp(X * w)))
11     tmp4 = X.T * tmp3
12     # gamma * (X.T * (y / (1 + np.exp(X * w))))
13     tmp5 = gamma * tmp4
14     # w -= gamma * (X.T * (y / (1 + np.exp(X * w))))
15     w -= gamma * tmp5
```

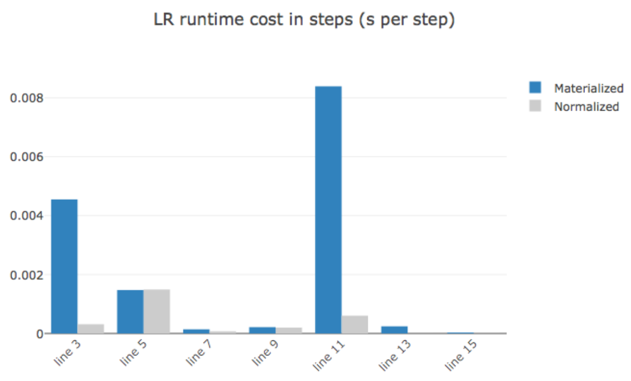

Figure 12: Runtime cost of Logistic Regression

run the algorithm for 20 iterations with initial *center_num* = 5. Because the algorithm has parts in a loop and parts outside a loop, we take the summation of the runtime cost of each line. The result of stepped runtime cost is presented in Figure 13.

Algorithm 5: K Means

```
1  def k_means(data, iterations, center_num):
2    all_one = np.matrix([1] * data.shape[0]).T
3    all_one_k = np.matrix([1] * center_num)
4    all_one_c = np.matrix([1] * k_center.shape[0]).T
5
6    t2 = (np.power(data, 2)).sum(axis=1) * all_one_k
7    t22 = data * 2
8    ya = None
9
10   for _ in range(iterations):
11     dist = t2 - t22 * k_center + all_one * np.power(
        k_center, 2).sum(axis=0)
12     ya = (dist == (np.amin(dist) * all_one_k))
13     k_center = (data.T * ya) / (all_one_c * ya.sum(axis
        =0))
14
15   return k_center, ya
```

Algorithm 6: K Means in Steps

```
1  def k_means(data, k_center, center_num):
2    all_one = np.matrix([1] * data.shape[0]).T
3    all_one_k = np.matrix([1] * center_num)
4    all_one_c = np.matrix([1] * k_center.shape[0]).T
5
6    # np.power(data, 2)
7    tmp0 = data.power(2) if sparse.issparse(data) else np.
        power(data, 2)
8    # (np.power(data, 2)).sum(axis=1)
9    tmp1 = tmp0.sum(axis=1)
10   # (np.power(data, 2)).sum(axis=1) * all_one_k
11   t2 = tmp1.dot(all_one_k)
12
13   t22 = data * 2
14   ya = None
15
16   for _ in range(iterations):
17     # t22 * k_center
18     tmp0 = t22 * k_center
19     # all_one * np.power(k_center, 2).sum(axis=0)
20     tmp1 = all_one * np.power(k_center, 2).sum(axis=0)
21     # dist = t2 - t22 * k_center + all_one * np.power(
        k_center, 2).sum(axis=0)
22     dist = t2 - tmp0 - tmp1
23
24     # np.amin(dist)
25     tmp2 = np.amin(dist)
26     # np.amin(dist) * all_one_k)
27     tmp3 = tmp2 * all_one_k
28     # ya = (dist == (np.amin(dist) * all_one_k))
29     ya = (dist == tmp3)
30
31     # data.T * ya
32     tmp4 = data.T * ya
33     # all_one_c * ya.sum(axis=0)
34     tmp5 = all_one_c * ya.sum(axis=0)
35     # k_center = (data.T * ya) / (all_one_c * ya.sum(axis
        =0))
36     k_center = tmp4 / tmp5
```

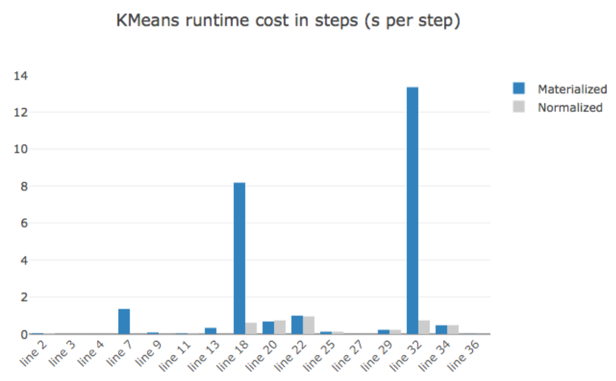

Figure 13: Runtime cost of K Means

Overall, the stepped result meets our expectation that LMM/RMM costs have been reduced significantly. One interesting point is that in materialized approach Line 7 imposes a noticeable portion of

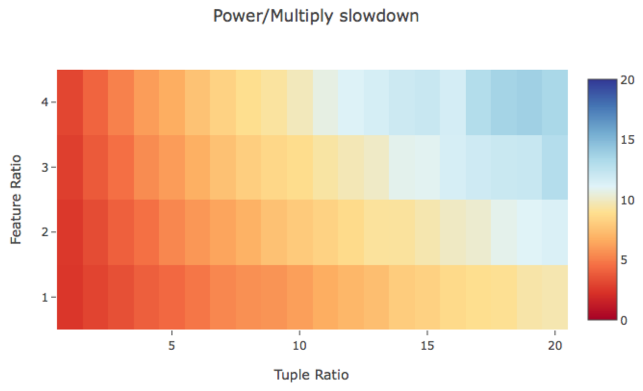

Figure 14: Slowdowns of Power operator compared to Multiply Operator

runtime cost while in normalized approach Line 7 has only trivial impact on the total runtime cost. Line 7 and 13 are scalar operations but np.power is obviously more expensive for large sparse datasets. It is interesting that two scalar operators power and multiply have significantly different runtime cost. We therefore run another small experiment to validate this effect by comparing the slowdown of power operator on synthetic dataset. The result in presented in Figure 14. The figure clearly demonstrates that in SciPy's implementation power on COO matrix is more expensive than multiplication. As such, we conclude that the speedup of power operator gives us advantage at getting a better speedup in k means.

5.5.4 GNMF. Algorithm 7 and 8 are original and stepped GNMF. We decompose two lines of Python code into ten phases in order to investigate the performance of MorpheusPy at a fine granularity. Line 3 and 12 are LMM/RMM that can be normalized, which we should expect to see substantial speedups. Other lines are parts that will not be normalized. We run the algorithm for 20 iterations with initial $components = 5$ and random w, h. We take average of the runtime cost of each step in the loop. The result of stepped runtime cost is presented in Figure 15.

Algorithm 7: GNMF

```
1 def GNMF(X, w, h):
2   for _ in range(iterations):
3     h = np.multiply(h, (w.T * X) / (w.T * w * h))
4     w = np.multiply(w, (X * h.T) / (w * (h * h.T)))
```

Similar to Logistic Regression, in Normalized approach, the runtime cost of Line 3, 12 only account for about $\frac{1}{2}$ of the total runtime cost. It means that even if we take out the runtime cost of these two operations, the maximum speedup we can achieve is just about 8x. On the other hand, line 15, 17 and 19 are expensive NumPy matrix multiplication which might bottleneck the total runtime of GNMF. Hence, overall achieve a comparable result as R implementation.

Algorithm 8: GNMF in Steps

```
1  def GNMF(X, w, h):
2    for _ in range(iterations):
3      tmp0 = w.T * X
4      tmp1 = w.T * w
5      # w.T * w * h
6      tmp2 = tmp1 * h
7      # (w.T * X) / (w.T * w * h)
8      tmp3 = tmp0 / tmp2
9      # np.multiply(h, (w.T * X) / (w.T * w * h))
10     h = np.multiply(h, tmp3)
11
12     tmp4 = X * h.T
13     tmp5 = h * h.T
14     # w * (h * h.T)
15     tmp6 = w * tmp5
16     # (X * h.T) / (w * (h * h.T))
17     tmp7 = tmp4 / tmp6
18     # np.multiply(w, (X * h.T) / (w * (h * h.T)))
19     w = np.multiply(w, tmp7)
```

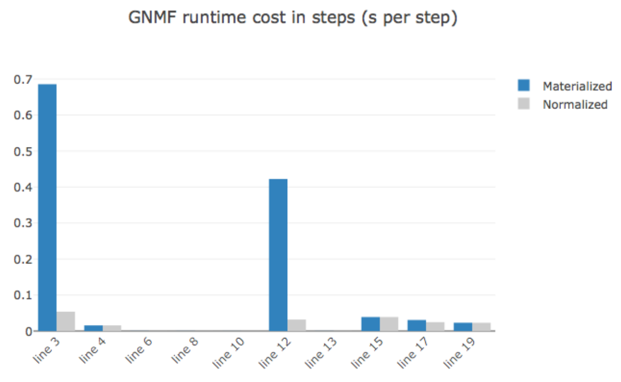

Figure 15: Runtime cost of GNMF

6 LIMITATIONS

In this section, we conclude limitations and lessons we learned for MorpheusPy implementation.

6.1 Poor Integration with Existing ML Library

Though Morpheus is designed to be a high-level wrapper that can be easily used with any existing machine learning library, it conflicts with performance optimizations of these libraries. As we have known from real-world experience and other online documentation, Python is imperative and therefore very slow. A lot of approaches such as Cython and PyPy are invented to improve Python's performance in general. For example, to optimize the performance, many SciKit learn algorithms directly employ Cython or even C level programming to manipulate memory, which means high-level wrapper is skipped. As a result, MorpheusPy's high-level rewrite will be ignored by SciKit Learn. To fully exploit performance improvement of Morpheus, we might need to develop another set of machine learning libraries.

For users interested in using MorpheusPy, we have implemented four basic algorithms including Linear Regression, Logistic Regression, K Means, and GNMF. All algorithms extend SciKit Learn's BaseEstimator, providing a similar API semantics as standard SciKit Learn algorithms.

6.2 No Support of Slicing

The idea of Morpheus is to perform lazy evaluations as a whole to avoid redundant runtime computation. To slice parts of a normalized matrix, we need to execute a dynamic join on the fly which may bring further performance overhead. As such, we decide not to implement slicing, more specifically *__getitem__* function, in MorpheusPy.

6.3 Improvement in Performance

As shown in Section 5, NumPy and SciPy have imposed lots of overhead on MorpheusPy implementation. Operators might be further optimized in C++. However, due to time constraint, we will not explore further in this project.

7 CONCLUSION

Factorized ML techniques have been shown to be beneficial to ML performance. Our previous prototype in R achieves the speedup of orders of magnitude for some machine learning algorithms. We extend the idea to the single-node environment of Python and NumPy as MorpheusPy. MorpheusPy implements all basic operators, as well as reimplements parts that cause a significant amount of overhead. We found interesting difference between the speedups obtained by MorpheusPy and MorpheusR for different ML algorithms. We verified that these differences boiled down to differences in the overheads of individual operator implementations in these two environments. Nevertheless, we found many cases where MorpheusPy gave substantial speedups, even over an order of magnitude in some cases. Meanwhile, we developed data preparation operations in the factorized fashion, which can serve as data cleaning utility. Our work also prepares us for the future implementation of graph-based ML frameworks such as TensorFlow.

All of our codes and the datasets are publicly released on the project webpage: https://adalabucsd.github.io/morpheus.html.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. 2017. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1214–1225. https://doi.org/10.14778/3137628.3137633

[3] R. Ricci. E. Eide and C. Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:the magazine of USENIX* 39, 6 (2014), 36–38. https://www.usenix.org/publications/login/dec14/ricci

[4] E. Jones, T. Oliphant, P. Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. (2001–). http://www.scipy.org/ [Online; accessed 2018-03-05].

[5] A. Kumar, J. Naughton, and J. M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1969–1984. https://doi.org/10.1145/2723372.2723713

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[7] R Core Team. 2018. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[8] S. C. Colbert S. Walt and G. Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (March 2011), 22–30.

[9] The Scipy community 2017. *NumPy C API*. The Scipy community. https://docs.scipy.org/doc/numpy-1.13.0/reference/c-api.html [Online; accessed 2018-03-05].

[10] The Scipy community 2017. *Subclassing ndarray*. The Scipy community. https://docs.scipy.org/doc/numpy-1.13.0/user/basics.subclassing.html [Online; accessed 2018-03-05].