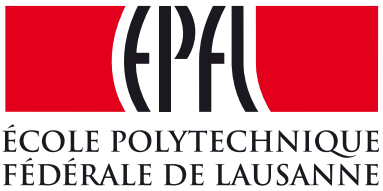


MACHINE LEARNING AND OPTIMIZATION LABORATORY, EPFL

Sparse Linear Algebra in the Deeplearning4j Framework

Audrey LOEFFEL

EPFL Professor:
Martin JÄGGI

Skymind Supervisor:
François GARILLOT

August 14, 2017

Acknowledgements

I would like to thank Professor Jäggi to have supervised from the EPFL. I would like to thank François Garillot for the help, the supervison and the feedbacks.
I would like to thank Raver for the precious help he provided. I would like to thank Ed, Susan, Slin, Natalie and Chris for welcoming me in the San Francisco team. Always a great ambiance and good place to work.

Lausanne, 12 Mars 2011 D. K.

Abstract

In this work we present an implementation of sparse multi-dimensional array and a set of operations for the Deeplearning4j ecosystem. We study different sparse storage formats and the manner they can be implement into the library to be compliant with the existing API. We also present the integration of the accelerated linear algebra operations with the BLAS routines and the native library of DL4J.

...

Contents

List of Figures

List of Tables

Introduction

Linear algebra forms the backbone of many machine learning algorithms, especially in deep learning. It provides structures such as vectors and matrices to hold the data and parameters to perform operations between them. Linear Algebra makes matrix operations fast and easy especially when training on GPUs.

Many of machine learning applications tend to deal with sparse dataset. Natural language processing applications, as stated by the Zips'f law, the frequency of any word in inversely proportional to its rank in the frequency table. ...

Typically in neural networks we have set of input organized into a vector on which we apply a collection of weights stored in matrices.

1 Sparse Data and Formats

Definition

Data is said sparse when it is contains only a few non-zero values. That kind of dataset is really common in Machine Learning applications and can be an high influence on the computation. Common use-cases include recommendation systems, clickstream dataset,...

The sparsity of a dataset is defined by :

$$sparsity = \frac{\text{\# non-zero values}}{\text{\# values}} \tag{1.1}$$

Conversely when a dataset has only a few null values, the data are said dense. The density of the dataset is defined by the inverse of the sparsity:

$$density = \frac{1}{\text{sparsity}} \tag{1.2}$$

Using dense methods and data structure with sparse data dismisses all optimization and performance gains we could achieve using sparse linear algebra.

The Advantages of Sparse Data

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. Machine learning and more specifically deep learning algorithms rely on linear algebra. Sparsity is a very useful property in Machine Learning. Some algorithms can have fast optimization, fast evaluation of the model, statistical robustness or other computational advantages.

A lot of machine learning applications are using sparse dataset such as recommender system.

Natural language processing algorithm tends to have sparse dataset as stated by the Zipf's law [Zip()].

Sparse Data are very common in Machine Learning

In Machine Learning it's very common to deal with sparse dataset. We can encounter them in any kind of applications: Natural Language Processing, Retrieving Systems, Recommender Systems, etc.

Given the possible optimization that sparse dataset allows and the high number of people that could take advantages of it, it becomes important to add the support of sparse data in the DL4J suite of libraries.

A Real Case of Sparse Dataset

In 2008 Netflix launched a contest, the Netflix Grand Prize [net()], to improve their recommender system model and to increase the accuracy of predictions. They published an sample dataset made with the ratings of anonymous Netflix customers. The dataset had more than 100 millions sampled ratings and it contained about $m = 480'186$ users and $m = 17'770$ movies [Koren(2009)]. If stored as a dense matrix, it would need to store $8'532'905'220$ values in memory. That corresponds to a sparsity $\cong \frac{100'000'000}{8'532'905'220} = 0.011719338$.

Storing more than 8 trillions 64-bit floating-point numbers needs more than 64 gigabyte of memory which quickly become unmanageable even for the world's fastest supercomputers.

Solution: Sparse Representation and Operations

Even if the compression of sparse data is very efficient when storing as dense data, we could gain in performance during the processing by using a sparse representation with sparse operations.

In sparse operations only the values are accessed and processed while in dense operations, every cell is accessed even if it contains a zero value.

To reduce the size of the memory needed and to gain in performance, we must store the data into a sparse format. There are different kind of formats which each of them is more suitable to different aspects (Storage vs computation).

Formats

There are many methods for storing sparse data, each of them presents different advantages and disadvantages.

Matrices

Coordinates Format

It is the simplest method to encode a sparse array. The coordinates and the value of each non-zero entry are stored in arrays. Typically each element is encoded in a tuple (row, column, value)

$$A_{(M\times N)} = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 3 \\ 1 & 0 & 4 \\ 0 & 0 & 0 \end{bmatrix} \quad \rightarrow \quad \begin{aligned} Values_{(1\times NNZ)} &= \begin{bmatrix} 2 & 3 & 1 & 4 \end{bmatrix} \\ Rows_{(1\times NNZ)} &= \begin{bmatrix} 0 & 1 & 2 & 2 \end{bmatrix} \\ Columns_{(1\times NNZ)} &= \begin{bmatrix} 1 & 2 & 0 & 2 \end{bmatrix} \end{aligned}$$

Figure 1.1: A matrix stored in COO format

This format provides an easy and fast way to retrieve a value and to insert a new non-zero element. It's also fast and simple to convert into a dense format.

But this format is not the most efficient regarding the memory consumption.

Compressed Row Format

The Compressed Row and the Compressed Column formats are the most general format to store a sparse array. They don't store any unnecessary element conversely to the COO format. But it requires more steps to access a element than the COO format.

Similarly to the row-major ordering explained in section 3.1, this format stores the values by row and use pointers to differentiate each row.

Each non-zero element of a row are stored contiguously in the memory. Each row are also contiguously stored.

The format, described by the Intel MKL Sparse Library [mkl(a)], requires four arrays:

Values	All the nonzero values are store contiguously in an array. The array size is NNZ.
Column pointers	This array keeps the column position for each values.
Beginning of row pointers	Each pointer i points to the first element of the row i in the values array. The array size is the number of rows of the array.
End of row pointers	Each pointer i points to the first element in the values array that does not belong to the row i . The array size is the number of rows of the array.

$$A_{(N\times M)} = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 \\ 0 & 0 & 2 & 1 \end{bmatrix} \rightarrow \begin{aligned} Values_{(1\times NNZ)} &= \begin{bmatrix} 2 & 3 & 1 & 4 & 2 & 1 \end{bmatrix} \\ Columns_{(1\times NNZ)} &= \begin{bmatrix} 1 & 2 & 0 & 2 & 2 & 3 \end{bmatrix} \\ pointersB_{(1\times N)} &= \begin{bmatrix} 0 & 1 & 2 & 2 & 4 \end{bmatrix} \\ PointersE_{(1\times N)} &= \begin{bmatrix} 1 & 2 & 2 & 4 & 6 \end{bmatrix} \end{aligned}$$

Figure 1.2: A matrix stored in CSR format

Compressed Column Format

The Compressed Column Format is similar to CSR but it compresses columns instead of rows. The values are stored in a column-major ordering, as explained in section 3.1.

Given a matrix $N \times M$, the pointers arrays will have a size M.

Tensors - Multi-dimensional arrays

A tensor is a multi-dimensional array. The order of the tensor is its number of dimension. It corresponds to the number of indexes needed to index a value. Matrices, vectors and scalars can be represented as tensors where the order is equals to 2, 1 and 0 respectively.

This generalization allows a more generic implementation of a n-dimensional array in the Nd4j library.

Coordinates Format

The COO format can easily be extended to encode tensors by storing an array of indexes instead the row and column coordinates.

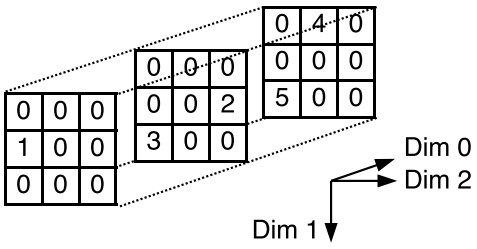


Figure 1.3: A $3 \times 3 \times 3$ tensor

The tensor, illustrated in figure 1.3, of order $K = 3$ with shape $3 \times 3 \times 3$ which has the following non-zero values :

value	indexes
1	0 1 0
2	1 1 2
3	1 2 0
4	2 0 1
5	2 2 0

It can be encoded with one values array and one indexes array :

$$Values_{(1\times NNZ)} = \begin{bmatrix} 1, & 2, & 3, & 4, & 5 \end{bmatrix}$$
$$Indexes_{(NNZ\times K)} = \begin{bmatrix} [0,1,0], & [1,1,2], & [1,2,0], & [2,0,1], & [2,2,0] \end{bmatrix}$$

Figure 1.4: A tensor stored in COO format

2 The Deeplearning4j Library

Deeplearning4j is a open-source Deep Learning library for the JVM. It runs on distributed CPU's and GPU's.

–> TODO

Architecture of the library

The library is composed by several sub-libraries:

Deeplearning4j provides the tools to implement neural networks and build computation graphs

Nd4j is the mathematical back-end of Deeplearning4j. It provides the data structures for the n-dimensional arrays and allow Java to access the native libraries via JavaCPP and the Java Native Interface.

Libnd4j is the computing library that provides native operations on CPU and GPU. It's written in C++ and Cuda.

Datavec provides the operations for the data processing such that data ingestion, normalization and transformation into feature vectors.

The Importance of Nd4j in the Library

Nd4j is at the base of the Deeplearning4j library, it provides data storage, manipulations, and operations. It gives the atomic pieces needed to build more complex deep learning systems. Nd4j stands for N-Dimensional Arrays for Java and is basically a scientific computing library for the JVM. It features n-dimensional array object and the support of CPU and GPU via Cuda.

The APIs provided by the library are essentially wrappers for the different version of BLAS

(Basic Linear Algebra Subprogram).

BLAS is a specification that defines the low-level routines for linear algebra operations (for vectors and matrices). There exist several libraries implementing those subroutines in C or Fortran for dense or sparse formats. In Nd4j the BLAS subroutines can directly be called from Java thanks to JavaCPP, that internally uses the Java Native Interface (JNI) to call native routines from the JVM environment. This architecture allows the library to benefit from the advantages of the native side.

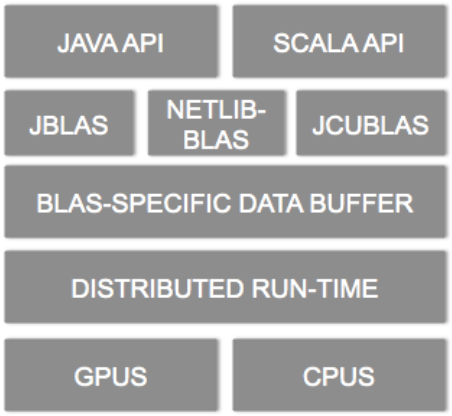


Figure 2.1: Nd4j architecture

Nd4j needs a Sparse Representation

In Deeplearning4j, Sparse Data are treated as dense and use the dense operations of BLAS and Libnd4j to perform computations. With a new sparse representation we can gain in storage space and computation speed.

Currently there is no library in the JVM ecosystem that supports the sparse operations. Despite the fact that the Breeze (a numerical processing library for Scala) library [bre()] support sparse vectors and sparse CSR matrices but no sparse tensor. Moreover it is only support the operations of the first level but not accelerated.

Nd4j will be the first JVM library to support the sparse tensors with their accelerated operations.

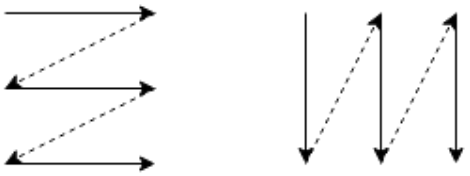
3 Structure of an Multi-dimensional Array

The new sparse array have to be compliant with the API and inter-operable with the current dense array implementation.

Storing an Array

A dense array is stored as a single contiguous block of memory, flatten in a one-dimensional array. Arrays are stored off-heap (outside the JVM environment). The reasons behind this design decision are numerous: better performance (avoid the interruptions of the garbage collector), better interoperability with BLAS libraries, and to avoid the disadvantages of the JVM such as the limited size of arrays due to the integer indexing (limited to $2^{31}-1 \cong 2.14$ billion elements)

There are two methods to store a multi-dimensional array into a linear memory space: row-major order (C) or column-major order (Fortran). Figure 3.1 shows how a two-dimensional array is stored according to the order.



$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \Rightarrow \begin{matrix} C-Order = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & a_{23} \end{bmatrix} \\ F-Order = \begin{bmatrix} a_{11} & a_{21} & a_{12} & a_{22} & a_{31} & a_{13} \end{bmatrix} \end{matrix}$$

Figure 3.1: Comparison between C-order and F-order

The data is accessed via strides which define how to index over contiguous block of data. For each dimension it defines by how many values two consecutive elements are separated. In the

case of matrix A defined in figure 3.1, the strides would be $(3,1)$ in case of C-order and $(1,3)$ in case of F-order. Strides $(3,1)$ means that each row is separated by 3 values and each column is separated by 1 value.

Data Buffer

The DataBuffer is a storage abstraction which provides optimal storage and retrieval depending on the backend. The data is stored off-heap through JavaCPP. It is basically a wrapper around a pointer and an indexer with utility methods to access and modify the data. The pointer points to the allocated memory space and the indexes provides an easy-to-use and efficient way to access a multi-dimensional memory space.

The implementation of the databuffer depends on the data type, because of the length needed to store a single value (int -> 32 bits, long -> 64bits, float -> 32bits, double -> 64bits, assuming a 64bit-architecture).

// TODO speak about memory copy, host and device memory, etc

Parameters of an Array

The information about the shape of the array are grouped in a DataBuffer object called ShapeInformation. It groups these following information:

Rank	The number of dimension of the array.
Shape	The shape of the array.
Strides	Provides information about the logical layout of the array for each dimension.
Offset	Provides the position of the first value in memory that belongs to the array or view (explained in section 3.2.
ElementWiseStride	Indicates how two contiguous elements are physically separated in memory.
Order	C or F order

Views

The data in memory can be shared by multiple NDArrays. An NDArray can refer to a subset of another NDArray. We say that such an array is a view of the original array. This is a powerful concept that avoid the unnecessary copy of the data which is a very expensive operation.

Since the memory space is shared, changes to array will impact the content of the other arrays. Figure **??** illustrates how a view shares its data with its original array.

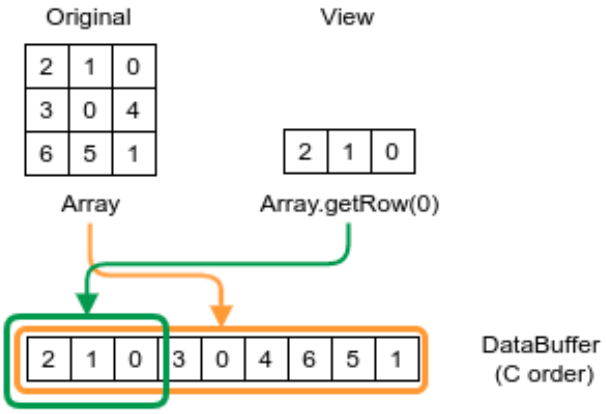


Figure 3.2: View shares memory with the original array

Indexes

NDArrays can be accessed through a combination of indexes. There are many way to index an array. For each dimension of the original array, we can specify if we want everything, an interval or a specific value of it. It gives a lot of flexibility to access sub-part of a array.

The indexes provide an efficient mechanism to access Tensors along dimension. A tensor along a dimension (TAD) is a view of an actual tensor with a lower rank.

Operations

Nd4j defines five types of operations:

1. Scalar: element-wise operations such as addition or multiplication where a scalar is applying on an ndarray.

2. Transform: in-place operations such as logarithm or cosine. There are usually executed in an element-wise manner but it's not always the case; some can be applied along a dimension.

3. Accumulation: Also called Reduction. Operations applied over the whole ndarray or along a dimension such as the sum of all values.

4. Index Accumulation: Similar to accumulation operations but return an index instead of a value. A classic operation is getting the maximum or minimum value along a dimension.

5. Broadcast: Some of the more useful operations are vector operations, such as *addRowVector* which add a row vector to every row of a matrix.

4 Implementation

Hierarchy of Arrays

The API of an array is defined by an interface called *INDArray* which has a dense implementation for each backend: *NDArray* class for the CPU and *JCublasNDArray* class for the GPU. But since most of the operations and methods are shared between the two backends, they are implemented in an abstract class called *BaseNDArray*.

Adding sparse representations asked two questions:

1. What can be shared with the dense arrays ?
2. What can be between the different sparse arrays and what are format-specific ?

To answer those questions, we need to go a little bit deeper in the implementation.

The dense implementation includes methods that are inherently related to the way dense array is internally made, and other methods are related to the generic parameters such as the shape or are utility method.

The first type of methods is not useful in case of sparse. Dense and sparse arrays are not built in the same manner. Methods such as *getStrides* are not relevant in the sparse context. Reciprocally the sparse array will need methods which will be irrelevant in the dense context.

We encounter the same situation between the different sparse formats. Some will need utility methods that the other ones won't need.

However some methods should be reusable such as traversal operations which operate in an element-wise manner.

But everything has to be defined in the *INDArray* interface. To avoid code duplication, everything than can be shared should be implemented in the higher level of the hierarchy.

The methods that are not compatible with a type of array will simply throw an unsupported operation exception.

The drawback brought with this solution is that we always need to verify the type of the array before doing any operations.

// TODO update schema

Figure 4.1: Arrays hierarchy in Nd4j

Limitations and Constraints

Nd4j has been made in the perspective of dense arrays. The design has been thought and optimized regarding the dense implementation which brings some constraints to implement the sparse representation

Storing off-heap

The data, encapsulated into DataBuffer object, is stored off-heap. Several reasons drive this design decision.

1. The size of the memory that can be allocated within the JVM is limited to 2 billion. We can not fit matrices into the JVM.
2. We want to use Cuda to accelerate the computation on the GPU. Cuda is not compatible with the JVM environment, the data has to be stored on the native side.
3. Java execution is always sequential, with C++ we can have cheap parallel executions.
4. At C++ level we can have SIMD (Single instruction, multiple data) instructions that allow to execute an instruction in parallel on different blocks of the same data.

Workspaces

Workspaces provides some contiguous memory buffer. No garbage collection mecanism occurs within them, the programmer has to take care of closing the workspace to avoid memory leaks.

DataBuffers have a fixed length

Once allocated a *DataBuffer* can not be reallocated. It is not a issue when manipulating dense arrays, we never need to add a new value in a dense array, we only update the existing values.

However we need to be able to add new values in the sparse buffers. Each time a zero value become a non-zero value we have to put the value and its indexes in memory.

To reallocate a buffer we need to:

1. Create a new *Pointer* to a bigger memory space and replace the current pointer of the *DataBuffer* .
2. Create a new *Indexer* with the new *Pointer* and replace the current indexer of the *DataBuffer*.
3. Copy the content from the old *Pointer* to the new space.
4. The buffer has two length attributes: *length* which count the number of values in the buffer and *underlyingLength* which reflects the actual allocated size in memory. After the reallocation the *underlyingLength* is increased to the new size, but the *length* only increases at the insertion of a new value.

We over-allocate the buffer in order to limit the amount of reallocation operations.

Another possibility would have been to create a new *DataBuffer* with the content of the old buffer plus the new value. But changing the DataBuffer object in a array would have broken the view mechanism since the views would still point to the old *DataBuffer* reference. Moreover this solution doe not allow the over-allocation.

Alternatives

-> TODO The implementation of CSR and COO would have been easier with managed memory. We could have stored the arrays in Java ArrayList and we woudn't have to take care of the memory, reallocation, etc

CSR Matrices

Structure

The representation uses 4 data buffers to encore a CSR matrix. One for the non-zero values, one for the columns indexes and two for the row pointers (to the beginning and to the end of each row).

Put a value

To insert a new value or to update an existing non-null value, we need to identify where the values of row we want ot insert to are located in the values buffer and in the columns buffer. The beginning and end of rows pointers give us the range of indexes.

While iterating over the range of values, if we find a value with the same column index than the one we want to insert to, we can update the values and nothing needs to be changed in the three other buffer.

However if there is no value with that column, we need to insert a new one at the correct position. Then we need to update the end of row pointer for this row. Finally each row pointers that come after need to be increment by one.

– ADD pseudo code for putscalar of csr?

Get a Sub-Array

1. First, we need to resolve the index to get the new shape of the array, the new rank, etc. In the case we use the resolution of the dense array but we are only interested in

   - the shape: an array with two elements containing the new shape of the sub-array.
   - the offsets: an array with two elements that indicate the first row and the first column that belongs to the sub-array.
   - the offset: indicates the position in the data buffer of the first element that belongs to the sub-array.

2. Sometimes the offsets are equal to zeros while having a non-null offset. In this case we need to override the offsets.

```
1        offsets[0] = (int) resolution.getOffset() / shape()[1];
2        offsets[1] = (int) resolution.getOffset() % shape()[1];
```

3. With the offsets we can now compute the first and the last position of each dimension.

```
1       long firstRow = offsets[0];
2       long lastRow = firstRow + shape[0];
3       long firstColumn = offsets[1];
4       long lastColumn = firstElement + shape[1];
```

4. Now we have the bounds for each dimension, we can reconstruct the new beginning and end of row pointers et the columns indexes.

Limits of this format

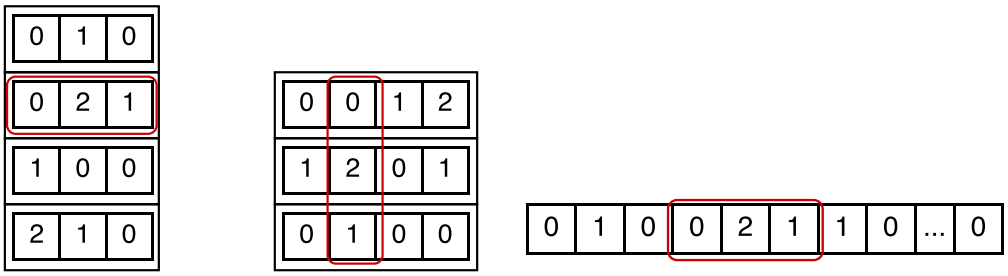
This formats only works with two dimensions and cannot be extended to tensors. Therefor it makes it difficult to be compatible with the API. Moreover the operations to get or put values aren't straightforward. Several step are necessary before accessing the value.

COO Tensors

Naive implementation

Based on the description in 1.5.2 the COO encoding needs one data buffer to store all the non-null values and one for the indexes of each values.

An easy solution would have been to store the indexes into a multi-dimension array of *DataBuffers*: One buffer for each value, or one buffer for each dimension. Due to the native constraints that makes hardly manageable to have such arrays (Difficulty to pass the array to the native side and Cuda side), we choose to flatten the indexes into one buffer.



(a) each index is stored contiguously

(b) Each dimension is stored contiguously

(c) Flattened Indexes

Figure 4.2: Illustration of the different possible datastructure for storing the indexes [0, 2, 1] of a value v

But this implementation makes difficult to be compliant with the API. It brings several issues:

- The key of views is the sharing of their data. In case of COO format, views have to share the data buffer and the indexes buffer. Without the indexes it is not possible to add a value in the original array by adding it in a view. If we only put the new value in the shared value buffer without updating the indexes, the original array would have a value buffer bigger than its indexes buffer and there would be an offset between the values and the indexes.

  Even when sharing both buffers, how would we know which value is included in the view and which is not?

- The coordinates of a value in a view are not necessary the sames of the same value in the original array.

They can be offset if dimension is partially included in the view. Figure 4.3 shows a matrix an a view (in red). The value $v_i = 5$ would have the coordinates [1, 1] in the original array while it has the coordinates [0, 0] in the view. How can the indexes be translated between views?

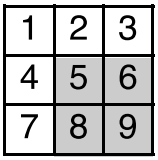


Figure 4.3: A 3 × 3 matrix with a 2 × 2 view in grey

- A view can have a lower or higher rank than its original array. How can the view indexes be stored if they do not have the same dimensionality?

Reverse coordinates

An important and frequently used method of the API is to get the value of a coordinates. Since the indexes are flatten into one buffer, we need to find it into buffer and get the underlying index.

Since the indexes buffer is lexicographically sorted, a binary search can be applied. But instead of comparing a value, we need to compare a sub-array of the buffer with the target indexes. An additional step is necessary before the comparison: We use the index range of the value buffer (from 0 to NNZ) and for each index we need to get the bounds of the sub-array and extract it.

tensor contractions are not equal depending on the dimensions because the time to access the elements is not the same because the values are sorted along the first dimension -> binary search, skip list, tree ...

More parameters are needed to define the tensors

Most of the issues cited in section 4.4.1 are due to the support of the different types of indexes. Each index implements the interface *INDArrayIndex* and extends from *NDArrayIndex*. They provide a very efficient and powerful mechanism to access part of a array but they introduce some constraints when implementing views for COO.

All Indexes

All indexes are the most straightforward of the library. They are used to collect all the elements of a dimension.

Interval Indexes

Interval indexes takes an subpart of a dimension containing in an interval. They don't modify the rank

The grey sub-array in figure 4.3 is the result of the operation :

```
1     myArray.get(NDArrayIndex.interval(1, 3), NDArrayIndex.interval(1, 3));
```

After the resolution of the indexes, we obtain offsets equal to $[1,1]$ with an offset equal to 4, which mean that the first row and the first column are not included in the view and the first element in memory that belongs to the view is at position 4.

To be able to identify what values belong to the view and what values do not, we need to define the bounds of each dimension. For this example, we have

$(idx_{row}, idx_{colum}) \in view$ if $idx_{row} \in [1,3[$ and $idx_{column} \in [1,3[$

We only need to store the lower bound i_{lower} of the interval $idx_{dim} \in [i_{lower}, i_{upper}[$, we can easily compute $i_{upper} = i_{lower} + shape[dim]$. The lower bounds are stored in the *sparseOffset* array.

Point Index

Point indexes take one unique element of a dimension. They reduce the rank of the array.

Assuming we have a tensor with a shape equals to $(2 \times 3 \times 3)$ as shown in figure 4.4. We want to take a slice along the first dimension, assuming that the first dimension being the pages, the second the rows and the third the columns. We use the following operation:

```
1     myTensor.get(NDArrayIndex.point(0),
2         NDIndexArray.all(),
3         NDIndexArray.all());
```

The result is a view with a shape equals to (3×3), as shown in grey in figure 4.4

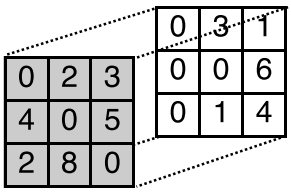


Figure 4.4: A $2 \times 3 \times 3$ tensor with a slice along the first dimension in grey

The coordinates of the resulting matrix have only two dimensions instead of three. We have an issue when trying to access a value given a pair of coordinates in the view context. There is no direct matching between these coordinates and those who actually are in the indexes buffer.

Each value of the tensor has its coordinates in the form (idx_0, idx_1, idx_2). But in the resulting view the values are defined with only two coordinates: (idx_1, idx_2), the first dimension is fixed with a value equal to 0.

To translate the view coordinates to the original coordinates which are actually stored into the shared buffer, we need to keep track of the unused dimension and its value. The solution is to add an additional parameter array that keeps track of the status of each dimension. The array is called *flags* and it can contains either 0, which means *active*, or 1 which means *fixed*.

In this example the flags array would be equal to [1, 0, 0] because the first dimension is fixed at position 0. The value 0 would be keep into the *sparseOffset* array. And the indexes translation would be : T(idx_1, idx_2) = $(sparseOffset[0], idx_1, idx_2)$

Specified Index

Specified index will take specific values of a dimension, depending on the position in the indexes array when calling $myArray.get(indexes...)$. This index takes an array in parameter with the values we want to select.

Assuming we have an 3 × 5 matrix, as shown in figure 4.5a, on which we call the following operation to select the first, the third and the fourth columns :

```
1      myMatrix.get(NDIndexArray.all(),
2          new SpecifiedIndex(0, 2, 3));
```

Instead of returning a view of $myMatrix$, it returns a new array containing the data from the original matrix. The data is copied, not shared. The data of the view that would result from this operation is not easily indexable in memory with stride and offsets and it would be difficult to represent it. Therefor the figure 4.5b is an independent 3 × 3 matrix resulting from the above operation.

0	2	3	1	2
4	0	5	0	0
2	8	0	1	0

(a) A 3 × 5 matrix with selected columns in grey

0	3	1
4	5	0
2	0	1

(b) A 2×2 matrix created from the column of the matrix in figure 4.5a

Figure 4.5: Example of a specified index behavior on a matrix

New Axis Index

New axis indexes are used to add a new dimension to the array. The new dimension always has a length equal to 1. It can be prepended or inserted in the middle of the dimensions. Since the rank is higher, more coordinates are needed to access a value. However the shared indexes buffers is limited to the original rank. Similarly to the point index, we need a new parameter array that keep tracks of the position of the new dimensions to be able to translate the coordinates from view to original context.

Assuming we have a 3×3 matrix: calling *myArray.get(NDArrayIndex.newAxis(), NDArrayIndex.all())* will prepend a new dimension. The view is a $1 \times 3 \times 3$ tensor with a hidden dimension parameter array equal to [0].

Sparse Indexes Translation

To translate a view coordinates to the underlying coordinates, we have to take into accounts: the sparse offsets, the hidden dimensions and the flags. With those three parameters we have everything need for translating.

While iterating over the view dimensions, three situations can happen :

1. The current dimension is hidden: We do nothing
2. The current dimension is fixed: We put the corresponding offset into the result array. Since several fixed dimensions can occur in a row, we need to repeat until we reach an active dimension.
3. The current dimension is active: We sum the view coordinate with the sparse offset.

This method is described in detail in algorithm 1

Computations of the the Parameters

Computation of the Sparse Offsets

The sparse offsets are computed from the offset. The offset gives us the position of the first element in the array. We want to reverse it the coordinates of this values which will be the sparse offset.

1. For each dimension except the innermost one, we divide the offset by the shape of that dimension. Rounded to the lower integer, this quotient gives us the sparse offset for the dimension. Then we need to remove the number of elements that are in the same dimension but at a lowest value (if the element is in the 4th row, we want to remove all the elements of the previous rows).

Algorithm 1 Translate the indexes from view to underlying context

```
procedure TRANSLATE(int[] viewIndexes)
        ▷ underlyingRank, hiddenDimensions and sparseOffsets are class fields

    result ← new int[underlyingRank]
    idxPhy ← 0, hidden ← 0

    for idxVir = 0 to viewIndexes.length do
                    ▷ The current dimension is hidden, it does not appear in the result
        if hidden < hiddenDimension.length & hiddenDimension[hidden] ==
idxVir then
            hidden ← hidden + 1
        else
            while idxPhy < underlyingRank & isFixed(idxPhy) do
                result[idxPhy] ← sparseOffsets[idxPhy] ▷ If the dimension is fixed, the
coordinate takes the value of the offset
                idxPhy ← idxPhy + 1
            end while
                    ▷ If the dimension is not fixed, we add the offset to the coordinate
            if idxPhy < underlyingRank &!isFixed(idxPhy) then
                result[idxPhy] ← sparseOffsets[idxPhy] + viewIndexes[idxVir]
                idxPhy ← idxPhy + 1
            end if
        end if
    end for
end procedure
```

2. We iterate until the last element. In this case we have the offsets set up up to the row dimension. To find which column the value is in, we need to take the modulo of the remaining offset by the number of columns. There is only one operation per dimension, then for a n-order tensor, the complexity of this step if O(n).

3. We have the sparse offsets given a set of indexes. But we might be computing the sparse offsets of a view which is already offset from an original array. We need to merge the new sparse offsets with those of the current array. The final result is the sum of the existing offset and the freshly computed offset.

   During the merge we should particularly be careful with fixed dimensions of the current array because they are absent from the sparse offset resolution explained above. The result does not contain any information about them, we need to add these fixed dimensions with their current offset to final offsets array.

Algorithm 2 Calculate the sparse offsets

procedure CREATESPARSEOFFSETS(int $offset$)
▷ $rank$, $underlyingRank$ (the rank of the original array and the number of dimension of the indexes buffer), $shape$ and $sparseOffsets$ are class fields of the array instance

```
newOffsets ← new int[rank]                          ▷ Compute the new offsets
for i = 0 to (rank − 2) do
    nbElements ← ∏_{j=i+1}^{rank} shape[j]
    newOffsets[i] ← ⌊offset ÷ nbElements⌋
    offset ← offset − newOffsets[i] × nbElements
end for
newOffsets[rank − 1] ← offset mod shape[rank − 1]

finalOffsets ← new int[underlyingRank] ▷ Merge with sparseOffsets of this array
active ← 0
for i = 0 to underlyingRanke do
    if flags[i] == 0 then        ▷ If the dimension is already inactive in the current array
        finalOffsets[i] ← sparseOffsets[i]
    else
        finalOffsets[i] ← newOffsets[active] + sparseOffsets[i]
        i ← i + 1
    end if
end for
return finalOffsets
end procedure
```

$sparseOffsets = [1,1,0,0,0] flags = [1,0,0,0,0]$

Figure 4.6: Results from the resolution

$offsets = [0,0] shape = [2,5] offset = 15$

Figure 4.7: Results from the resolution

Example of the Sparse Offset Computation

Let's take an tensor $myTensor$ with a shape equal to $[2,4,4,5]$ on which we call the following operation :

```
1     myTensor.get(NDIndexArray.all(),
2     NDIndexArray.point(0),
3     NDIndexArray.point(3),
4     NDIndexArray.all());
```

$myTensor$ is actually a view of a 5-order tensor and has a the following parameters:

Assuming we name the dimension as [book, page, row, column], we are taking all the column of the last row of the first pages of each book.

The indexes resolution returns the following parameters:

First step is to iterate over the dimension:

Number of element in one book: $numElement = 4 \times 4 \times 5 = 60$ Book offset = $\lfloor offset/numElement \rfloor = \lfloor 15/60 \rfloor = 0$ Then we update the offset: $offset = offset - 0 * 60$

Number of element in one page = $numElement = 4 \times 5 = 20$ Page offset = $\lfloor offset/numElement \rfloor = \lfloor 15/20 \rfloor = 0$ Then we update the offset: $offset = 15 - 0 * 20$

Number of element in one row = $numElement = 5$ Page offset = $\lfloor offset/numElement \rfloor = \lfloor 15/5 \rfloor = 3$ Then we update the offset: $offset = 15 - 3 * 5 = 0$

Finally we reach the last dimension: Column offset = $offset \mod numElement = 0 \mod 5 = 0$

We get an temporary offsets array equal to $newOffsets = [0,0,3,0]$. Now we need to merge with the existing offsets of $myTensor$.

Its first dimension is fixed, so we copy its sparse offset : $finalOffest[0] = myTensor.sparseOffset[0] = 1$ Its second dimension is active and there is already an non-zero offset. The offset is equal to $finalOffset[1] = newOffset[0] + myTensor.sparseOffset[1] = 0 + 1 = 1$ Its second di-

mension is active. The offset is equal to $finalOffset[2] = newOffset[1] + myTensor.sparseOffset[2] = 0+0 = 0$ Its third dimension is active. The offset is equal to $finalOffset[3] = newOffset[2] + myTensor.sparseOffset[3] = 3+0 = 0$ Its third dimension is active. The offset is equal to $finalOffset[4] = newOffset[3] + myTensor.sparseOffset[4] = 0+0 = 0$

We finally get the final sparseOffset : $[1,1,0,3,0]$

Computation of the Flags

the *Flags* array determines which dimension is active and which one is hidden from the point of view of the array. The dimension can only be reduced using a *point* index, which means the flags can be computed during the offset resolution. An interval with a length equals 1 does not reduced the dimensionality of the array. We fill the flags array while iterating over the indexes array.

Computation of the Hidden Dimensions

The resolution returns an array containing the position of the *newAxis* indexes in the indexes array. The array may already have some hidden dimensions. In this case we need to adapt the result with the current hidden dimension.

Final Implementation

The final representation is encoded with 5 *DataBuffers*:

1. Values: Stored the value linearly.
2. Indexes: Stored the flattened indexes of each values.
3. Flags: Define which dimensions are active and which are fixed.
4. Sparse Offsets: Define the bound of each dimension.
5. Hidden Dimensions: Keep track of the position in the shape array of the hidden dimension.

The indexes and values are sorted in the lexicographic order in order to make the search by indexes easily via binary search. The sort is ensured before performing any operations.

The flags, the offsets and the hidden dimensions arrays are grouped in one *DataBuffer*, similarly to the *ShapeInformation* buffer.

Algorithm 3 Calculate the hidden dimensions

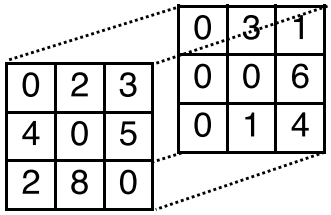
```
procedure CREATEHIDDENDIMENSIONS(int[] newAxis)
                                    ▷ hiddenDimensions is a class field of the array

    if newAxis is empty or null then return hiddenDimensions
    end if

    if hiddenDimensions is empty then return newAxis
    end if

    size ← newAxis.length + hiddenDimensions.length          ▷ Merge both arrays
    newArray ← new int[size], newArrayIdx ← 0, newDim ← 0

    for (oldDim = 0 to hiddenDimensions.length) do
        while ((oldDim ≥ hiddenDimensions.length ‖ newAxis[newDim] ≤
hiddenDimensions[i]) && newDim < newAxis.length) do
            newArray[i] ← newAxis[newDim] + oldDim
            newArrayIdx ← newArrayIdx + 1
            newDim ← newDim + 1
        end while
        newArray[i] ← hiddenDimensions[oldDim] + newDim
        newArrayIdx ← newArrayIdx + 1
    end forreturn newArray
end procedure
```

Example

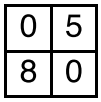
Figure 4.8a shows a tensor T with a shape $(2 \times 3 \times 3)$ and Figure 4.8b shows the view resulting from the operation below:

```
T_view = T.get(NDArrayIndex.newAxis(),
    NDArrayIndex.point(0),
    NDArrayIndex.interval(1, 3),
    NDArrayIndex.interval(1, 3));
```

The first index prepends a new dimension and increases the rank, the second takes the first page of the tensor and reduces the rank. Then the next ones select the sub-array in the first page. The parameters of T_{view} are listed in figure 4.9



(a) A tensor T with shape $(2 \times 3 \times 3)$



(b) A view of tensor T with shape $(1 \times 2 \times 2)$

Figure 4.8: A tensor and a view of it

$$Shape = \begin{bmatrix} 1 & 2 & 2 \end{bmatrix} \quad Values = \begin{bmatrix} 5 & 8 \end{bmatrix}$$
$$Indexes = \begin{bmatrix} 0 & 1 & 2 & 0 & 2 & 1 \end{bmatrix} \quad Flags = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$
$$SparseOffsets = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \quad HiddenDimension = \begin{bmatrix} 0 \end{bmatrix}$$

Figure 4.9: Parameters defining the view T_{view} in 4.8b

Put a Value

Several steps are needed to put a new value in a COO array:

1. Translate the indexes into the underlying context.
2. Verify if the value of this indexes is already non null. In this case we can either remove the entry if the new value is 0, or replace the value in the buffer. Nothing else need to be done.
3. The new value is 0: the indexes and the value are not added in the buffers.
4. It is a new non-null value: we need to insert the value and the indexes in the buffers. But first we need to ensure that the buffers have enough spaces for the new entries. If

not, we need to reallocate them. The new entries are added at the end of the buffers; the sort are not maintained at insertion to avoid sorting the array too often when it is not needed.

Get a Sub-Array

The new parameters make the task easier. Once the indexes resolved and the parameters computed, a new array is created with the current values and indexes buffers and the new parameters.

However if one of the indexes is a *SpecifiedIndex* we need to create a new array and copy the data. In this case the indexes resolution is translate any type of indexes to specified index:

Assuming we have a $3 \times 3 \times 3 \times 3$ 4-order tensor. The indexes array of the following operation:

```
1      myTensor.get(NDIndexArray.all(),
2          NDIndexArray.point(1),
3          NDIndexArray.interval(0, 2),
4          new SpecifiedIndex(0,2));
```

would become : [SpecifiedIndex(0,1,2), SpecifiedIndex(1), SpecifiedIndex(0,1), SpecifiedIndex(0,2)].

Then we iterate over all the included combination of coordinates and add the elements into a new empty array.

5 Operations

Basic Linear Algerba Subprograms (BLAS)

As introduced in section 2.2, BLAS defines the low-level routines for linear algebra operations (for vectors and matrices). Nd4j supports OpenBlas and Intel MKL as the CPU's libraries and Cuda (Cublas) as the GPU's library.

Those libraries are using a dense format only. In case of sparse we need to support two additional libraries: Intel MKL Sparse Blas and CuSparse. To be able to call the MKL sparse methods which are in the native side, we need to add its presets. The presets are generated by JavaCPP and define the signatures of each routines for which we need to bind it into a wrapper method in the java side.

BLAS is organized into three sets of routines called *levels*:

Level 1 performs scalar, vector and vector-vector operations (dot product, norm,...).

Level 2 performs matrix-vector operations (matrix-vector multiplication (gemv)).

Level 3 performs matrix-matrix operations (general matrix multiplication (gemm)).

All the operations are accessible through a BLAS wrapper of *Nd4j*. They can be directly called via static methods or by calling them through an higher abstraction in *INDArray* objects.

This wrapper only wraps the methods of the dense BLAS library, this is why we need a specific wrapper for sparse BLAS library. However we want the different implementations to be transparent for the users, thus we use the dense wrapper as entry point and then redirect the call to the sparse wrapper internally. Thereby the user does not need to know the type of array is manipulating.

A wrapper contains the implementations of each level and Lapack, which is a linear equation system resolver library. The *Levels* interface define the set of operations for each level and then

the implementations gather the the parameter and link to the JavaCPP presets and the native side.

Similarly the sparse architecture has to have the same architecture: One sparse wrapper which contains the sparse implementation for each level.

// TODO explain the architecture in nd4j?

Backends

Nd4j can run on the CPU or the GPU.

It supports interchangeable backends. The API is the same, the differences between the backends are transparent for the user. Each backend is parameter-able

– TODO :

In-Place Routines

Some of the operations are performed in-place, that means the input matrix is modified during the operations.

We might have an issue when the number of non-null values increases. We could run out of space in the different buffers because the memory can not be reallocated from the native side and because we can not know how many new values will be added. To avoid this problem we have to ensure having enough space before performing the operations.

The solution is to allocate the maximum size that the routine could use to the *DataBuffers*. At first it may look like inefficient and counterproductive according to the benefits we want to gain with the sparse implementation. However such operations are never executed on the whole array but only one a small sub-part of it.

Example

To illustrate this issue let's take an array $myArray$ with a size equals to $(100'000 \times 100'000 \times 100)$. The following operation take a slice of the array $myArray$ (which corresponds in a row in this example) and assign 1.0 to each value.

```
1      myArray.get(NDIndexArray.point(0),
2          NDIndexArray.point(0),
3          NDIndexArray.all())
4          .assign(1.0),
```

Despite the huge size of the array, we only add at most 100 new values. If the buffers have not the capacity to get this amount of new values, we need to reallocate them prior the operation.

Level 1 Routines

Sparse BLAS Level 1 routines in Intel MKL library [mkl(b)] is a set of functions that perform vector operations on sparse vectors stored in a compressed format.

The vector is represented in the compressed form by two arrays, *values* and *indexes*. Figure 5.1 shows how a vector is represented in that format.

$$V_{(N\times 1)} = \begin{bmatrix} a_{k_0} \\ 0 \\ a_{k_1} \\ \dots \\ a_{k_{NNZ}} \end{bmatrix} \quad \rightarrow \quad \begin{aligned} Value &= \begin{bmatrix} a_{k_0} & a_{k_1} & \dots & a_{k_{NNZ}} \end{bmatrix} \\ Indexes &= \begin{bmatrix} k_0 & k_1 & \dots & k_{NNZ} \end{bmatrix} \end{aligned}$$

Figure 5.1: A vector v and its compressed form representation

The *values* array can be easily obtained from the CSR and COO format since it corresponds to the *values* buffer in both representations.

In case of CSR vector the *indexes* array corresponds to the column indexes buffer and can also be easily obtained.

However a value in a COO vector is always defined by two coordinates. It is due to a Nd4j singularity where the lowest possible rank for an array is 2, even in case of vector. The *indexes* array has to be extract from the *indexes* buffer depending whether it is a row vector or a column vector.

Once we get the two arrays, the procedure to call the BLAS routines is identical regardless of the underlying sparse format of the array. That mean we only need one implementation. The logic behind the arrays getters are implemented in the format-specific classes.

Level 2 and Level 3 Routines

The level 2 routines perform operations between a sparse matrix and dense vectors whereas level 3 ones perform between a sparse matrix and dense matrices.

Intel MKL library supports several sparse matrix storage formats such as COO and CSR and each type has its own implementation of the routines.

For now only the general matrix-vector product is implemented (Gemv) for the COO format. All the input parameters of the COO routines can be extracted by a *SparseCOOGemvParameters* object which makes the future implementation of the level 2 routines easier.

Gemv Routine

This routine computes the matrix-vector product of a COO matrix and a dense vector. It can either compute $y = A * x$ or $y = A^T * x$ according to the parameter received in argument.

All the parameters needed to call the BLAS routine are extracted and computed according to the type of sparse format. In case of COO, we need the an array of values, an array with the row indexes and an array with the column indexes. Since the indexes are flattened into one buffer, we split have to it into the two arrays.

Libnd4j

Most of the operations are implemented into libNd4j. Basically we used gemv and gemm from blas ..

6 Results

Storing a huge data set

Storing a huge sparse data set (example of gloVe algo dataset?) is now possible with COO while it wasn't with dense. -> compare the theoretical size since dense is not possible to instance

Operations performance

level 1

compare 2-3 operations, which ones?

level 2

Gemv

7 Future

Sparse Representation

API compliant

Implement the methods of INDArray tobe used like the dense array. reshape, operations, etc..

More Supported Sparse Format

support csr, csc for matrix and conversion between them. Slice of a COO tensor to CSR vector, etc

– import data directly in sparse format – conversion from dense

..?

optimizing the tensor contraction

-> skip list or other datastructure -> sort along different dimensions

Operations

-> blas level 2 and 3 -> libnd4j -> assign, etc -> support tensors and op on contraction

Optimization

Nd4j is built with the idea to avoid the JVM environement for storing the data. It is based on the postulate that the data is usually huge and does not fit into the memory. However with the new sparse implementation, we can store huge datasets into a reasonable size of memory (as

long as it has a high sparsity)

Perhaps we should consider storing spars array on-heap.

More complexe format like storing by block (sparse-> on-heap, dense -> off-heap), or full sparse on-heap with managed memory.

8 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

A An appendix

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliography

[Zip()] Zipf's law. https://en.wikipedia.org/wiki/Zipf

[bre()] Breeze - a numerical processing library for scala. https://github.com/scalanlp/breeze. [Online; accessed 14-August-2017].

[mkl(a)] Intel MKL sparse format. https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-csr-matrix-storage-format, a. [Online; accessed 8-August-2017].

[mkl(b)] Intel MKL sparse level 1 routines. https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-level-1-routines, b. [Online; accessed 13-August-2017].

[net()] Netflix grand prize. http://www.netflixprize.com. [Online; accessed 8-August-2017].

[Koren(2009)] Yehuda Koren. 1 the bellkor solution to the netflix grand prize, 2009.