Data Types - RDD-based API

MLlib supports local vectors and matrices stored on a single machine, as well as distributed matrices backed by one or more RDDs. Local vectors and local matrices are simple data models that serve as public interfaces. The underlying linear algebra operations are provided by Breeze. A training example used in supervised learning is called a “labeled point” in MLlib.

Local vector

A local vector has integer-typed and 0-based indices and double-typed values, stored on a single machine. MLlib supports two types of local vectors: dense and sparse. A dense vector is backed by a double array representing its entry values, while a sparse vector is backed by two parallel arrays: indices and values. For example, a vector `(1.0, 0.0, 3.0)` can be represented in dense format as `[1.0, 0.0, 3.0]` or in sparse format as `(3, [0, 2], [1.0, 3.0])`, where `3` is the size of the vector.

Scala **Java** **Python**

MLlib recognizes the following types as dense vectors:

- NumPy’s `array`
- Python’s list, e.g., `[1, 2, 3]`

and the following as sparse vectors:

- MLlib's `SparseVector`.
- SciPy's `csc_matrix` with a single column

We recommend using NumPy arrays over lists for efficiency, and using the factory methods implemented in `Vectors` to create sparse vectors.

Refer to the `Vectors` Python docs for more details on the API.

```
import numpy as np
import scipy.sparse as sps
from pyspark.mllib.linalg import Vectors

# Use a NumPy array as a dense vector.
dv1 = np.array([1.0, 0.0, 3.0])
# Use a Python list as a dense vector.
dv2 = [1.0, 0.0, 3.0]
# Create a SparseVector.
sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])
# Use a single-column SciPy csc_matrix as a sparse vector.
sv2 = sps.csc_matrix((np.array([1.0, 3.0]), np.array([0, 2]), np.array([0, 2])), shape = (3, 1))
```

Labeled point

A labeled point is a local vector, either dense or sparse, associated with a label/response. In MLlib, labeled points are used in supervised learning algorithms. We use a double to store a label, so we can use labeled points in both regression and classification. For binary classification, a label should be either `0` (negative) or `1` (positive). For multiclass classification, labels should be class indices starting from zero: `0, 1, 2, ...`.

Scala **Java** **Python**

A labeled point is represented by `LabeledPoint`.

Refer to the `LabeledPoint` Python docs for more details on the API.

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense feature vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse feature vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

Sparse data

It is very common in practice to have sparse training data. MLlib supports reading training examples stored in `LIBSVM` format, which is the default format used by `LIBSVM` and `LIBLINEAR`. It is a text format in which each line represents a labeled sparse feature vector using the following format:

```
label index1:value1 index2:value2 ...
```

where the indices are one-based and in ascending order. After loading, the feature indices are converted to zero-based.

Scala **Java** **Python**

`MLUtils.loadLibSVMFile` reads training examples stored in LIBSVM format.

Refer to the `MLUtils` Python docs for more details on the API.

```
from pyspark.mllib.util import MLUtils

examples = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
```

Local matrix

A local matrix has integer-typed row and column indices and double-typed values, stored on a single machine. MLlib supports dense matrices, whose entry values are stored in a single double array in column-major order, and sparse matrices, whose non-zero entry values are stored in the Compressed Sparse Column (CSC) format in column-major order. For example, the following dense matrix

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{pmatrix}$$

is stored in a one-dimensional array `[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]` with the matrix size `(3, 2)`.

Scala **Java** **Python**

The base class of local matrices is `Matrix`, and we provide two implementations: `DenseMatrix`, and `SparseMatrix`. We recommend using the factory methods implemented in `Matrices` to create local matrices. Remember, local matrices in MLlib are stored in column-major order.

Refer to the `Matrix` Python docs and `Matrices` Python docs for more details on the API.

```
from pyspark.mllib.linalg import Matrix, Matrices

# Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
dm2 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])

# Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
```

Distributed matrix

A distributed matrix has long-typed row and column indices and double-typed values, stored distributively in one or more RDDs. It is very important to choose the right format to store large and distributed matrices. Converting a distributed matrix to a different format may require a global shuffle, which is quite expensive. Four types of distributed matrices have been implemented so far.

The basic type is called `RowMatrix`. A `RowMatrix` is a row-oriented distributed matrix without meaningful row indices, e.g., a collection of feature vectors. It is backed by an RDD of its rows, where each row is a local vector. We assume that the number of columns is not huge for a `RowMatrix` so that a single local vector can be reasonably communicated to the driver and can also be stored / operated on using a single node. An `IndexedRowMatrix` is similar to a `RowMatrix` but with row indices, which can be used for identifying rows and executing joins. A `CoordinateMatrix` is a distributed matrix stored in coordinate list (COO) format, backed by an RDD of its entries. A `BlockMatrix` is a distributed matrix backed by an RDD of `MatrixBlock` which is a tuple of `(Int, Int, Matrix)`.

Note

The underlying RDDs of a distributed matrix must be deterministic, because we cache the matrix size. In general the use of non-deterministic RDDs can lead to errors.

RowMatrix

A `RowMatrix` is a row-oriented distributed matrix without meaningful row indices, backed by an RDD of its rows, where each row is a local vector. Since each row is represented by a local vector, the number of columns is limited by the integer range but it should be much smaller in practice.

Scala **Java** **Python**

A `RowMatrix` can be created from an `RDD` of vectors.

Refer to the `RowMatrix` Python docs for more details on the API.

```
from pyspark.mllib.linalg.distributed import RowMatrix

# Create an RDD of vectors.
rows = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])

# Create a RowMatrix from an RDD of vectors.
mat = RowMatrix(rows)

# Get its size.
m = mat.numRows()  # 4
n = mat.numCols()  # 3
```

```
# Get the rows as an RDD of vectors again.
rowsRDD = mat.rows
```

IndexedRowMatrix

An `IndexedRowMatrix` is similar to a `RowMatrix` but with meaningful row indices. It is backed by an RDD of indexed rows, so that each row is represented by its index (long-typed) and a local vector.

Scala Java **Python**

An `IndexedRowMatrix` can be created from an `RDD` of `IndexedRow`s, where `IndexedRow` is a wrapper over `(long, vector)`. An `IndexedRowMatrix` can be converted to a `RowMatrix` by dropping its row indices.

Refer to the `IndexedRowMatrix` Python docs for more details on the API.

```
from pyspark.mllib.linalg.distributed import IndexedRow, IndexedRowMatrix

# Create an RDD of indexed rows.
#   - This can be done explicitly with the IndexedRow class:
indexedRows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
                              IndexedRow(1, [4, 5, 6]),
                              IndexedRow(2, [7, 8, 9]),
                              IndexedRow(3, [10, 11, 12])])
#   - or by using (long, vector) tuples:
indexedRows = sc.parallelize([(0, [1, 2, 3]), (1, [4, 5, 6]),
                              (2, [7, 8, 9]), (3, [10, 11, 12])])

# Create an IndexedRowMatrix from an RDD of IndexedRows.
mat = IndexedRowMatrix(indexedRows)

# Get its size.
m = mat.numRows()  # 4
n = mat.numCols()  # 3
```

```
# Get the rows as an RDD of IndexedRows.
rowsRDD = mat.rows

# Convert to a RowMatrix by dropping the row indices.
rowMat = mat.toRowMatrix()
```

CoordinateMatrix

A `CoordinateMatrix` is a distributed matrix backed by an RDD of its entries. Each entry is a tuple of `(i: Long, j: Long, value: Double)`, where `i` is the row index, `j` is the column index, and `value` is the entry value. A `CoordinateMatrix` should be used only when both dimensions of the matrix are huge and the matrix is very sparse.

Scala **Java** **Python**

A `CoordinateMatrix` can be created from an `RDD` of `MatrixEntry` entries, where `MatrixEntry` is a wrapper over `(long, long, float)`. A `CoordinateMatrix` can be converted to a `RowMatrix` by calling `toRowMatrix`, or to an `IndexedRowMatrix` with sparse rows by calling `toIndexedRowMatrix`.

Refer to the `CoordinateMatrix` Python docs for more details on the API.

```
from pyspark.mllib.linalg.distributed import CoordinateMatrix, MatrixEntry

# Create an RDD of coordinate entries.
#   - This can be done explicitly with the MatrixEntry class:
entries = sc.parallelize([MatrixEntry(0, 0, 1.2), MatrixEntry(1, 0, 2.1), MatrixEntry(6, 1, 3.7)])
#   - or using (long, long, float) tuples:
entries = sc.parallelize([(0, 0, 1.2), (1, 0, 2.1), (2, 1, 3.7)])

# Create an CoordinateMatrix from an RDD of MatrixEntries.
mat = CoordinateMatrix(entries)

# Get its size.
```

```
m = mat.numRows()  # 3
n = mat.numCols()  # 2

# Get the entries as an RDD of MatrixEntries.
entriesRDD = mat.entries

# Convert to a RowMatrix.
rowMat = mat.toRowMatrix()

# Convert to an IndexedRowMatrix.
indexedRowMat = mat.toIndexedRowMatrix()

# Convert to a BlockMatrix.
blockMat = mat.toBlockMatrix()
```

«

BlockMatrix

A `BlockMatrix` is a distributed matrix backed by an RDD of `MatrixBlock`s, where a `MatrixBlock` is a tuple of `((Int, Int), Matrix)`, where the `(Int, Int)` is the index of the block, and `Matrix` is the sub-matrix at the given index with size `rowsPerBlock` x `colsPerBlock`. `BlockMatrix` supports methods such as `add` and `multiply` with another `BlockMatrix`. `BlockMatrix` also has a helper function `validate` which can be used to check whether the `BlockMatrix` is set up properly.

Scala **Java** **Python**

A `BlockMatrix` can be created from an `RDD` of sub-matrix blocks, where a sub-matrix block is a `((blockRowIndex, blockColIndex), sub-matrix)` tuple.

Refer to the `BlockMatrix` Python docs for more details on the API.

```
from pyspark.mllib.linalg import Matrices
from pyspark.mllib.linalg.distributed import BlockMatrix

# Create an RDD of sub-matrix blocks.
```

```
blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
                         ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])

# Create a BlockMatrix from an RDD of sub-matrix blocks.
mat = BlockMatrix(blocks, 3, 2)

# Get its size.
m = mat.numRows() # 6
n = mat.numCols() # 2

# Get the blocks as an RDD of sub-matrix blocks.
blocksRDD = mat.blocks

# Convert to a LocalMatrix.
localMat = mat.toLocalMatrix()

# Convert to an IndexedRowMatrix.
indexedRowMat = mat.toIndexedRowMatrix()

# Convert to a CoordinateMatrix.
coordinateMat = mat.toCoordinateMatrix()
```

«