Manual of Frovedis Spark API

Contents

Chapter 1

Introduction

This manual contains Spark API documantation. If you are new to Frovedis, please read the tutorial_spark first.

Currently we only provide part of the API documentation. We are still updating the contents.

- exrpc
  - FrovedisSparseData
- Matrix
  - FrovedisBlockcyclicMatrix
  - pblas_wrapper
  - scalapack_wrapper
  - [arpack_wrapper]
  - getrf_result
  - gesvd_result
- Machine Learning
  - LinearRegressionModel
  - Linear Regression
  - Lasso Regression
  - Ridge Regression
  - LogisticRegressionModel
  - Logistic Regression
  - SVMModel
  - Linear SVM
  - MatrixFactorizationModel
  - Matrix Factorization using ALS
  - kmeans

Chapter 2

FrovedisSparseData

2.1 NAME

FrovedisSparseData - A data structure used in modeling the in-memory sparse data of frovedis server side at client spark side.

2.2 SYNOPSIS

import com.nec.frovedis.exrpc.FrovedisSparseData

2.2.1 Constructors

FrovedisSparseData (`RDD[Vector]` data)

2.2.2 Public Member Functions

Unit load (`RDD[Vector]` data)
Unit loadcoo (`RDD[Rating]` data)
Unit debug_print()
Unit release()

2.3 DESCRIPTION

FrovedisSparseData is a pseudo sparse structure at client spark side which aims to model the frovedis server side sparse data (basically crs matrix).

Note that the actual sparse data is created at frovedis server side only. Spark side FrovedisSparseData contains a proxy handle of the in-memory sparse data created at frovedis server, along with number of rows and number of columns information.

2.3.1 Constructor Documentation

2.3.1.1 FrovedisSparseData (`RDD[Vector]` data)

It accepts a spark-side RDD data of sparse or dense Vector and converts it into the frovedis server side sparse data whose proxy along with number of rows and number of columns information are stored in the constructed FrovedisSparseData object.

For example,

```
// sample input matrix file with elements in a row separated by whitespace
val data = sc.textFile(input)
// parsedData: RDD[Vector]
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))
// conversion of spark data to frovedis side sparse data
val fdata = new FrovedisSparseData(parsedData)
```

2.3.2 Pubic Member Function Documentation

2.3.2.1 Unit load (`RDD[Vector]` data)

This function can be used to load a spark side sparse data to a frovedis server side sparse data (crs matrix). It accepts a spark-side RDD data of sparse or dense Vector and converts it into the frovedis server side sparse data whose proxy along with number of rows and number of columns information are stored in the target FrovedisSparseData object.

For example,

```
// sample input matrix file with elements in a row separated by whitespace
val data = sc.textFile(input)
// parsedData: RDD[Vector]
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))

val fdata = new FrovedisSparseData() // an empty object
// conversion of spark data to frovedis side sparse data
fdata.load(parsedData)
```

2.3.2.2 Unit loadcoo (`RDD[Rating]` data)

This function can be used to load a spark side Rating matrix (COO data) to a frovedis server side sparse data (crs matrix). It accepts a spark-side `RDD[Rating]` object and converts it into the frovedis server side sparse data whose proxy along with number of rows and number of columns information are stored in the target FrovedisSparseData object.

For example,

```
// sample input matrix file with rows of COO triplets (i,j,k)
val data = sc.textFile(input)
// ratings: RDD[Rating]
val ratings = data.map(_.split(',') match { case Array(user, item, rate) =>
                      Rating(user.toInt, item.toInt, rate.toDouble)
               })
```

```
val fdata = new FrovedisSparseData() // an empty object
// conversion of spark coo data to frovedis side sparse (crs) data
fdata.loadcoo(ratings)
```

2.3.2.3 Unit debug_print()

It prints the contents of the server side sparse data on the server side user terminal. It is mainly useful for debugging purpose.

2.3.2.4 Unit release()

This function can be used to release the existing in-memory data at frovedis server side.

Chapter 3

FrovedisBlockcyclicMatrix

3.1 NAME

FrovedisBlockcyclicMatrix - A data structure used in modeling the in-memory
blockcyclic matrix data of frovedis server side at client spark side.

3.2 SYNOPSIS

import com.nec.frovedis.matrix.FrovedisBlockcyclicMatrix

3.2.1 Constructors

FrovedisBlockcyclicMatrix (`RDD[Vector]` data)

3.2.2 Public Member Functions

Unit load (`RDD[Vector]` data)
Unit load (String path)
Unit loadbinary (String path)
Unit save (String path)
Unit savebinary (String path)
RowMatrix to_spark_RowMatrix (SparkContext sc)
Vector to_spark_Vector ()
Matrix to_spark_Matrix ()
Unit debug_print()
Unit release()

3.3 DESCRIPTION

FrovedisBlockcyclicMatrix is a pseudo matrix structure at client spark side which aims to model the frovedis server side `blockcyclic_matrix<double>` (see manual of frovedis blockcyclic_matrix for details).

Note that the actual matrix data is created at frovedis server side only. Spark side FrovedisBlockcyclicMatrix contains a proxy handle of the in-memory matrix data created at frovedis server, along with number of rows and number of columns information.

3.3.1 Constructor Documentation

3.3.1.1 FrovedisBlockcyclicMatrix (`RDD[Vector]` data)

It accepts a spark-side `RDD[Vector]` and converts it into the frovedis server side blockcyclic matrix data whose proxy along with number of rows and number of columns information are stored in the constructed FrovedisBlockcyclicMatrix object.

For example,

```
// sample input matrix file with elements in a row separated by whitespace
val data = sc.textFile(input)
// parsedData: RDD[Vector]
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))
// conversion of spark data to frovedis blockcyclic matrix
val fdata = new FrovedisBlockcyclicMatrix(parsedData)
```

3.3.2 Pubic Member Function Documentation

3.3.2.1 Unit load (`RDD[Vector]` data)

This function can be used to load a spark side dense data to a frovedis server side blockcyclic matrix. It accepts a spark `RDD[Vector]` object and converts it into the frovedis server side blockcyclic matrix whose proxy along with number of rows and number of columns information are stored in the target FrovedisBlockcyclicMatrix object.

For example,

```
// sample input matrix file with elements in a row separated by whitespace
val data = sc.textFile(input)
// parsedData: RDD[Vector]
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))

val fdata = new FrovedisBlockcyclicMatrix() // an empty object
// conversion of spark data to frovedis blockcyclic matrix
fdata.load(parsedData)
```

3.3.2.2 Unit load (String path)

This function is used to load the text data from the given file in the target server side matrix instance. Note that the file must be placed at server side at the given path.

3.3.2.3 Unit loadbinary (String path)

This function is used to load the little-endian binary data from the given file in the target server side matrix instance. Note that the file must be placed at server side at the given path.

3.3.2.4 Unit save (String path)

This function is used to save the target matrix as text file with the filename at the given path. Note that the file will be saved at server side at the given path.

3.3.2.5 Unit savebinary (String path)

This function is used to save the target matrix as little-endian binary file with the filename at the given path. Note that the file will be saved at server side at the given path.

3.3.2.6 RowMatrix to_spark_RowMatrix (SparkContext sc)

This function is used to convert the target matrix into spark RowMatrix. Note that this function will request frovedis server to send back the distributed data at server side blockcyclic matrix in the rowmajor-form and the spark client will then convert the distributed chunks received from frovedis server to spark distributed RowMatrix.

The SparkContext object "sc" will be required while converting the frovedis data to spark distributed RowMatrix.

3.3.2.7 Vector to_spark_Vector ()

This function is used to convert the target matrix into spark Vector form. Note that this function will request frovedis server to send back the distributed data at server side blockcyclic matrix in the rowmajor-form and the spark client will then convert the received rowmajor data from frovedis server into spark non-distributed Vector object.

3.3.2.8 Matrix to_spark_Matrix ()

This function is used to convert the target matrix into spark Matrix form. Note that this function will request frovedis server to send back the distributed data at server side blockcyclic matrix in the column-major form and the spark client will then convert the received column-major data from frovedis server into spark Matrix object.

3.3.2.9 Unit debug_print()

It prints the contents of the server side distributed matrix data on the server side user terminal. It is mainly useful for debugging purpose.

3.3.2.10 Unit release()

This function can be used to release the existing in-memory data at frovedis server side.

Chapter 4

pblas_wrapper

4.1 NAME

pblas_wrapper - a frovedis module provides user-friendly interfaces for commonly used pblas routines in scientific applications like machine learning algorithms.

4.2 SYNOPSIS

import com.nec.frovedis.matrix.PBLAS

4.2.1 Public Member Functions

```
Unit PBLAS.swap (FrovedisBlockcyclicMatrix v1, FrovedisBlockcyclicMatrix v2)
Unit PBLAS.copy (FrovedisBlockcyclicMatrix v1, FrovedisBlockcyclicMatrix v2)
Unit PBLAS.scal (FrovedisBlockcyclicMatrix v, Double al)
Unit PBLAS.axpy (FrovedisBlockcyclicMatrix v1,
          FrovedisBlockcyclicMatrix v2, Double al = 1.0)
Double PBLAS.dot (FrovedisBlockcyclicMatrix v1, FrovedisBlockcyclicMatrix v2)
Double PBLAS.nrm2 (FrovedisBlockcyclicMatrix v)
Unit PBLAS.gemv (FrovedisBlockcyclicMatrix m, FrovedisBlockcyclicMatrix v1,
          FrovedisBlockcyclicMatrix v2, Boolean trans = false,
          Double al = 1.0, Double be = 0.0)
Unit PBLAS.ger (FrovedisBlockcyclicMatrix v1, FrovedisBlockcyclicMatrix v2,
         FrovedisBlockcyclicMatrix m, Double al = 1.0)
Unit PBLAS.gemm (FrovedisBlockcyclicMatrix m1, FrovedisBlockcyclicMatrix m2,
          FrovedisBlockcyclicMatrix m3, Boolean trans_m1 = false,
          Boolean trans_m2 = false, Double al = 1.0, Double be = 0.0)
Unit PBLAS.geadd (FrovedisBlockcyclicMatrix m1, FrovedisBlockcyclicMatrix m2,
          Boolean trans = false, Double al = 1.0, Double be = 1.0)
```

4.3 DESCRIPTION

PBLAS is a high-performance scientific library written in Fortran language. It provides rich set of functionalities on vectors and matrices. The computation loads of these functionalities are parallelized over the available processes in a system and the user interfaces of this library is very detailed and complex in nature. It requires a strong understanding on each of the input parameters, along with some distribution concepts.

Frovedis provides a wrapper module for some commonly used PBLAS subroutines in scientific applications like machine learning algorithms. These wrapper interfaces are very simple and user needs not to consider all the detailed distribution parameters. Only specifying the target vectors or matrices with some other parameters (depending upon need) are fine. At the same time, all the use cases of a PBLAS routine can also be performed using Frovedis PBLAS wrapper of that routine.

This scala module implements a client-server application, where the spark client can send the spark matrix data to frovedis server side in order to create blockcyclic matrix at frovedis server and then spark client can request frovedis server for any of the supported PBLAS operation on that matrix. When required, spark client can request frovedis server to send back the resultant matrix and it can then create equivalent spark data (Vector, Matrix, RowMatrix etc., see manuals for FrovedisBlockcyclicMatrix to spark data conversion).

The individual detailed descriptions can be found in the subsequent sections. Please note that the term “inout”, used in the below section indicates a function argument as both “input” and “output”.

4.3.1 Detailed Description

4.3.1.1 swap (v1, v2)

Parameters
v1: A FrovedisBlockcyclicMatrix with single column (inout)
v2: A FrovedisBlockcyclicMatrix with single column (inout)

Purpose
It will swap the contents of v1 and v2, if they are semantically valid and are of same length.

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.3.1.2 copy (v1, v2)

Parameters
v1: A FrovedisBlockcyclicMatrix with single column (input)
v2: A FrovedisBlockcyclicMatrix with single column (output)

Purpose
It will copy the contents of v1 in v2 (v2 = v1), if they are semantically valid and are of same length.

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.3.1.3 scal (v, al)

Parameters
v: A FrovedisBlockcyclicMatrix with single column (inout)
al: A double parameter to specify the value to which the input vector needs to be scaled. (input)

Purpose
It will scale the input vector with the provided “al” value, if it is semantically valid. On success, input vector “v” would be updated (in-place scaling).

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.3.1.4 axpy (v1, v2, al=1.0)

Parameters
v1: A FrovedisBlockcyclicMatrix with single column (input)
v2: A FrovedisBlockcyclicMatrix with single column (inout)
al: A double parameter to specify the value to which "v1" needs to be scaled (not in-place scaling) [Default: 1.0] (input/optional)

Purpose
It will solve the expression v2 = al*v1 + v2, if the input vectors are semantically valid and are of same length. On success, "v2" will be updated with desired result, but "v1" would remain unchanged.

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.3.1.5 dot (v1, v2)

Parameters
v1: A FrovedisBlockcyclicMatrix with single column (input)
v2: A FrovedisBlockcyclicMatrix with single column (input)

Purpose
It will perform dot product of the input vectors, if they are semantically valid and are of same length. Input vectors would not get modified during the operation.

Return Value
On success, it returns the dot product result of the type double. If any error occurs, it throws an exception.

4.3.1.6 nrm2 (v)

Parameters
v: A FrovedisBlockcyclicMatrix with single column (input)

Purpose
It will calculate the norm of the input vector, if it is semantically valid. Input vector would not get modified during the operation.

Return Value
On success, it returns the norm value of the type double. If any error occurs, it throws an exception.

4.3.1.7 gemv (m, v1, v2, trans=false, al=1.0, be=0.0)

Parameters
m: A FrovedisBlockcyclicMatrix (input)
v1: A FrovedisBlockcyclicMatrix with single column (input)
v2: A FrovedisBlockcyclicMatrix with single column (inout)
trans: A boolean value to specify whether to transpose "m" or not [Default: false] (input/optional)
al: A double type value [Default: 1.0] (input/optional)
be: A double type value [Default: 0.0] (input/optional)

Purpose
The primary aim of this routine is to perform simple matrix-vector multiplication.
But it can also be used to perform any of the below operations:

```
(1) v2 = al*m*v1 + be*v2
(2) v2 = al*transpose(m)*v1 + be*v2
```

If trans=false, then expression (1) is solved. In that case, the size of "v1" must be at least the number of columns in "m" and the size of "v2" must be at least the number of rows in "m".
If trans=true, then expression (2) is solved. In that case, the size of "v1" must be at least the number of rows in "m" and the size of "v2" must be at least the number of columns in "m".

Since "v2" is used as input-output both, memory must be allocated for this vector before calling this routine, even if simple matrix-vector multiplication is required. Otherwise, this routine will throw an exception.

For simple matrix-vector multiplication, no need to specify values for the input parameters "trans", "al" and "be" (leave them at their default values).

On success, "v2" will be overwritten with the desired output. But "m" and "v1" would remain unchanged.

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.3.1.8 ger (v1, v2, m, al=1.0)

Parameters
v1: A FrovedisBlockcyclicMatrix with single column (input)
v2: A FrovedisBlockcyclicMatrix with single column (input)
m: A FrovedisBlockcyclicMatrix (inout)
al: A double type value [Default: 1.0] (input/optional)

Purpose
The primary aim of this routine is to perform simple vector-vector multiplication of the sizes "a" and "b" respectively to form an axb matrix. But it can also be used to perform the below operations:

```
m = al*v1*v2' + m
```

This operation can only be performed if the inputs are semantically valid and the size of "v1" is at least the number of rows in matrix "m" and the size of "v2" is at least the number of columns in matrix "m".

Since "m" is used as input-output both, memory must be allocated for this matrix before calling this routine, even if simple vector-vector multiplication is required. Otherwise it will throw an exception.

For simple vector-vector multiplication, no need to specify the value for the input parameter "al" (leave it at its default value).

On success, "m" will be overwritten with the desired output. But "v1" and "v2" will remain unchanged.

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.3.1.9 gemm (m1, m2, m3, trans_m1=false, trans_m2=false, al=1.0, be=0.0)

Parameters
m1: A FrovedisBlockcyclicMatrix (input)
m2: A FrovedisBlockcyclicMatrix (input)
m3: A FrovedisBlockcyclicMatrix (inout)
trans_m1: A boolean value to specify whether to transpose "m1" or not [Default: false] (input/optional)
trans_m2: A boolean value to specify whether to transpose "m2" or not [Default: false] (input/optional)
al: A double type value [Default: 1.0] (input/optional)
be: A double type value [Default: 0.0] (input/optional)

Purpose
The primary aim of this routine is to perform simple matrix-matrix multiplication.
But it can also be used to perform any of the below operations:

```
(1) m3 = al*m1*m2 + be*m3
(2) m3 = al*transpose(m1)*m2 + be*m3
(3) m3 = al*m1*transpose(m2) + be*m3
(4) m3 = al*transpose(m1)*transpose(m2) + be*m3
```

(1) will be performed, if both “trans_m1” and “trans_m2” are false.

(2) will be performed, if trans_m1=true and trans_m2 = false.

(3) will be performed, if trans_m1=false and trans_m2 = true.

(4) will be performed, if both “trans_m1” and “trans_m2” are true.

If we have four variables nrowa, nrowb, ncola, ncolb defined as follows:

```
if(trans_m1) {
  nrowa = number of columns in m1
  ncola = number of rows in m1
}
else {
  nrowa = number of rows in m1
  ncola = number of columns in m1
}

if(trans_m2) {
  nrowb = number of columns in m2
  ncolb = number of rows in m2
}
else {
  nrowb = number of rows in m2
  ncolb = number of columns in m2
}
```

Then this function can be executed successfully, if the below conditions are all true:

```
(a) "ncola" is equal to "nrowb"
(b) number of rows in "m3" is equal to or greater than "nrowa"
(b) number of columns in "m3" is equal to or greater than "ncolb"
```

Since “m3” is used as input-output both, memory must be allocated for this matrix before calling this routine, even if simple matrix-matrix multiplication is required. Otherwise it will throw an exception.

For simple matrix-matrix multiplication, no need to specify the value for the input parameters “trans_m1”, “trans_m2”, “al”, “be” (leave them at their default values).

On success, “m3” will be overwritten with the desired output. But “m1” and “m2” will remain unchanged.

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.3.1.10 geadd (m1, m2, trans=false, al=1.0, be=1.0)

Parameters
m1: A FrovedisBlockcyclicMatrix (input)

m2: A FrovedisBlockcyclicMatrix (inout)
trans: A boolean value to specify whether to transpose “m1” or not [Default: false] (input/optional)
al: A double type value [Default: 1.0] (input/optional)
be: A double type value [Default: 1.0] (input/optional)

Purpose
The primary aim of this routine is to perform simple matrix-matrix addition. But it can also be used to perform any of the below operations:

```
(1) m2 = al*m1 + be*m2
(2) m2 = al*transpose(m1) + be*m2
```

If trans=false, then expression (1) is solved. In that case, the number of rows and the number of columns in “m1” should be equal to the number of rows and the number of columns in “m2” respectively.
If trans=true, then expression (2) is solved. In that case, the number of columns and the number of rows in “m1” should be equal to the number of rows and the number of columns in “m2” respectively.

If it is needed to scale the input matrices before the addition, corresponding “al” and “be” values can be provided. But for simple matrix-matrix addition, no need to specify values for the input parameters “trans”, “al” and “be” (leave them at their default values).

On success, “m2” will be overwritten with the desired output. But “m1” would remain unchanged.

Return Value
On success, it returns nothing. If any error occurs, it throws an exception.

4.4 SEE ALSO

scalapack_wrapper

Chapter 5

scalapack_wrapper

5.1 NAME

scalapack_wrapper - a frovedis module provides user-friendly interfaces for commonly used scalapack routines in scientific applications like machine learning algorithms.

5.2 SYNOPSIS

import com.nec.frovedis.matrix.ScaLAPACK

5.3 WRAPPER FUNCTIONS

GetrfResult ScaLAPACK.getrf (FrovedisBlockcyclicMatrix m)
Int ScaLAPACK.getri (FrovedisBlockcyclicMatrix m, Long ipivPtr)
Int ScaLAPACK.getrs (FrovedisBlockcyclicMatrix m1, FrovedisBlockcyclicMatrix m2,
    Long ipivPtr, boolean trans = false)
Int ScaLAPACK.gesv (FrovedisBlockcyclicMatrix m1, FrovedisBlockcyclicMatrix m2)
Int ScaLAPACK.gels (FrovedisBlockcyclicMatrix m1, FrovedisBlockcyclicMatrix m2,
    Boolean trans = false)
GesvdResult ScaLAPACK.gesvd (FrovedisBlockcyclicMatrix m,
    Boolean wantU = false, Boolean wantV = false)

5.4 DESCRIPTION

ScaLAPACK is a high-performance scientific library written in Fortran language. It provides rich set of linear algebra functionalities whose computation loads are parallelized over the available processes in a system and the user interfaces of this library is very detailed and complex in nature. It requires a strong understanding on each of the input parameters, along with some distribution concepts.

Frovedis provides a wrapper module for some commonly used ScaLAPACK subroutines in scientific applications like machine learning algorithms. These wrapper interfaces are very simple and user needs not to consider all the detailed distribution parameters. Only specifying the target vectors or matrices with some other parameters (depending upon need) are fine. At the same time, all the use cases of a ScaLAPACK routine can also be performed using Frovedis ScaLAPACK wrapper of that routine.

This scala module implements a client-server application, where the spark client can send the spark matrix data to frovedis server side in order to create blockcyclic matrix at frovedis server and then spark client can request frovedis server for any of the supported ScaLAPACK operation on that matrix. When required, spark client can request frovedis server to send back the resultant matrix and it can then create equivalent spark data (Vector, Matrix, RowMatrix etc., see manuals for FrovedisBlockcyclicMatrix to spark data conversion).

The individual detailed descriptions can be found in the subsequent sections. Please note that the term "inout", used in the below section indicates a function argument as both "input" and "output".

5.4.1 Detailed Description

5.4.1.1 getrf (m)

Parameters
m: A FrovedisBlockcyclicMatrix (inout)

Purpose
It computes an LU factorization of a general M-by-N distributed matrix, "m" using partial pivoting with row interchanges.

On successful factorization, matrix "m" is overwritten with the computed L and U factors. Along with the return status of native scalapack routine, it also returns the proxy address of the node local vector "ipiv" containing the pivoting information associated with input matrix "m" in the form of GetrfResult. The "ipiv" information will be useful in computation of some other routines (like getri, getrs etc.)

Return Value
On success, it returns the object of the type GetrfResult as explained above. If any error occurs, it throws an exception explaining cause of the error.

5.4.1.2 getri (m, ipivPtr)

Parameters
m: A FrovedisBlockcyclicMatrix (inout)
ipiv: A long object containing the proxy of the ipiv vector (from GetrfResult) (input)

Purpose
It computes the inverse of a distributed square matrix using the LU factorization computed by getrf(). So in order to compute inverse of a matrix, first compute it's LU factor (and ipiv information) using getrf() and then pass the factored matrix, "m" along with the "ipiv" information to this function.

On success, factored matrix "m" is overwritten with the inverse (of the matrix which was passed to getrf()) matrix. "ipiv" will be internally used by this function and will remain unchanged.

For example,

```
val res = ScaLAPACK.getrf(m) // getting LU factorization of "m"
ScaLAPACK.getri(m,res.ipiv()) // "m" will have inverse of the initial value
```

Return Value
On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

5.4.1.3 getrs (m1, m2, ipiv, trans=false)

Parameters
m1: A FrovedisBlockcyclicMatrix (input)
m2: A FrovedisBlockcyclicMatrix (inout)
ipiv: A long object containing the proxy of the ipiv vector (from GetrfResult) (input)
trans: A boolean value to specify whether to transpose "m1" [Default: false] (input/optional)

Purpose
It solves a real system of distributed linear equations, AX=B with a general distributed square matrix (A) using the LU factorization computed by getrf(). Thus before calling this function, it is required to obtain the factored matrix "m1" (along with "ipiv" information) by calling getrf().

For example,

```
val res = ScaLAPACK.getrf(m1) // getting LU factorization of "m1"
ScaLAPACK.getrs(m1,m2,res.ipiv())
```

If trans=false, the linear equation AX=B is solved.
If trans=true, the linear equation transpose(A)X=B (A'X=B) is solved.

The matrix "m2" should have number of rows >= the number of rows in "m1" and at least 1 column in it.

On entry, "m2" contains the distributed right-hand-side (B) of the equation and on successful exit it is overwritten with the distributed solution matrix (X).

Return Value
On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

5.4.1.4 gesv (m1, m2)

Parameters
m1: A FrovedisBlockcyclicMatrix (inout)
m2: A FrovedisBlockcyclicMatrix (inout)

Purpose
It solves a real system of distributed linear equations, AX=B with a general distributed square matrix, "m1" by computing it's LU factors internally. This function internally computes the LU factors and ipiv information using getrf() and then solves the equation using getrs().

The matrix "m2" should have number of rows >= the number of rows in "m1" and at least 1 column in it.

On entry, "m1" contains the distributed left-hand-side square matrix (A), "m2" contains the distributed right-hand-side matrix (B) and on successful exit "m1" is overwritten with it's LU factors, "m2" is overwritten with the distributed solution matrix (X).

Return Value
On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

5.4.1.5 gels (m1, m2, trans=false)

Parameters
m1: A FrovedisBlockcyclicMatrix (input)
m2: A FrovedisBlockcyclicMatrix (inout)
trans: A boolean value to specify whether to transpose "m1" [Default: false] (input/optional)

Purpose
It solves overdetermined or underdetermined real linear systems involving an M-by-N distributed matrix (A) or its transpose, using a QR or LQ factorization of (A). It is assumed that distributed matrix (A) has full rank.

If trans=false and M >= N: it finds the least squares solution of an overdetermined system.
If trans=false and M < N: it finds the minimum norm solution of an underdetermined system.
If trans=true and M >= N: it finds the minimum norm solution of an underdetermined system.
If trans=true and M < N: it finds the least squares solution of an overdetermined system.

The matrix "m2" should have number of rows >= max(M,N) and at least 1 column.

On entry, "m1" contains the distributed left-hand-side matrix (A) and "m2" contains the distributed right-hand-side matrix (B). On successful exit, "m1" is overwritten with the QR or LQ factors and "m2" is overwritten with the distributed solution matrix (X).

Return Value
On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

5.4.1.6 gesvd (m, wantU=false, wantV=false)

Parameters
m: A FrovedisBlockcyclicMatrix (inout)
wantU: A boolean value to specify whether to compute U matrix [Default: false] (input)
wantV: A boolean value to specify whether to compute V matrix [Default: false] (input)

Purpose
It computes the singular value decomposition (SVD) of an M-by-N distributed matrix.

On entry "m" contains the distributed matrix whose singular values are to be computed.

If wantU = wantV = false, then it computes only the singular values in sorted oder, so that sval(i) >= sval(i+1). Otherwise it also computes U and/or V (left and right singular vectors respectively) matrices.

On successful exit, the contents of "m" is destroyed (internally used as workspace).

Return Value
On success, it returns the object of the type GesvdResult containing the singular values and U and V components (based on the requirement) along with the exit status of the native scalapack routine. If any error occurs, it throws an exception explaining cause of the error.

5.5 SEE ALSO

pblas_wrapper, arpack_wrapper, getrf_result, gesvd_result

Chapter 6

getrf_result

6.1 NAME

getrf_result - a structure to model the output of frovedis wrapper of scalapack getrf routine.

6.2 SYNOPSIS

import com.nec.frovedis.matrix.GetrfResult

6.2.1 Public Member Functions

Unit release()
Long ipiv()
Int stat()

6.3 DESCRIPTION

GetrfResult is a client spark side pseudo result structure containing the proxy of the in-memory scalapack getrf result (node local ipiv vector) created at frovedis server side.

6.3.1 Public Member Function Documentation

6.3.1.1 Unit release()

This function can be used to release the in-memory result component (ipiv vector) at frovedis server.

6.3.1.2 Long ipiv()

This function returns the proxy of the node_local "ipiv" vector computed during getrf calculation. This value will be required in other scalapack routine calculation, like getri, getrs etc.

6.3.1.3 Int stat()

This function returns the exit status of the scalapack native getrf routine on calling of which the target result object was obtained.

Chapter 7

gesvd_result

7.1 NAME

gesvd_result - a structure to model the output of frovedis singular value decomposition methods.

7.2 SYNOPSIS

import com.nec.frovedis.matrix.GesvdResult

7.2.1 Public Member Functions

`SingularValueDecomposition[RowMatrix,Matrix]` to_spark_result(SparkContext sc)
Unit save(String svec, String umat, String vmat)
Unit savebinary(String svec, String umat, String vmat)
Unit load_as_colmajor(String svec, String umat, String vmat)
Unit load_as_blockcyclic(String svec, String umat, String vmat)
Unit loadbinary_as_colmajor(String svec, String umat, String vmat)
Unit loadbinary_as_blockcyclic(String svec, String umat, String vmat)
Unit debug_print()
Unit release()
Int stat()

7.3 DESCRIPTION

GesvdResult is a client spark side pseudo result structure containing the proxies of the in-memory SVD results created at frovedis server side. It can be used to convert the frovedis side SVD result to spark equivalent data structures.

7.3.1 Public Member Function Documentation

7.3.1.1 `SingularValueDecomposition[RowMatrix,Matrix]` to_spark_result(SparkContext sc)

This function can be used to convert the frovedis side SVD results to spark equivalent result structure (`SingularValueDecomposition[RowMatrix,Matrix]`). Internally it uses the SparkContext object while performing this conversion.

7.3.1.2 save(String svec, String umat, String vmat)

This function can be used to save the result values in different text files at server side. If saving of U and V components are not required, “umat” and “vmat” can be null, but “svec” should have a valid filename.

7.3.1.3 savebinary(String svec, String umat, String vmat)

This function can be used to save the result values in different little-endian binary files at server side. If saving of U and V components are not required, “umat” and “vmat” can be null, but “svec” should have a valid filename.

7.3.1.4 load_as_colmajor(String svec, String umat, String vmat)

This function can be used to load the target result object with the values in given text files. If loading of U and V components are not required, “umat” and “vmat” can be null, but “svec” should have a valid filename.

If “umat” and/or “vmat” filenames are given, they will be loaded as frovedis distributed column major matrix.

7.3.1.5 load_as_blockcyclic(String svec, String umat, String vmat)

This function can be used to load the target result object with the values in given text files. If loading of U and V components are not required, “umat” and “vmat” can be null, but “svec” should have a valid filename.

If “umat” and/or “vmat” filenames are given, they will be loaded as frovedis distributed blockcyclic matrix.

7.3.1.6 loadbinary_as_colmajor(String svec, String umat, String vmat)

This function can be used to load the target result object with the values in given little-endian binary files. If loading of U and V components are not required, “umat” and “vmat” can be null, but “svec” should have a valid filename.

If “umat” and/or “vmat” filenames are given, they will be loaded as frovedis distributed column major matrix.

7.3.1.7 loadbinary_as_blockcyclic(String svec, String umat, String vmat)

This function can be used to load the target result object with the values in given little-endian binary files. If loading of U and V components are not required, “umat” and “vmat” can be null, but “svec” should have a valid filename.

If “umat” and/or “vmat” filenames are given, they will be loaded as frovedis distributed blockcyclic matrix.

7.3.1.8 Unit debug_print()

This function can be used to print the result components at server side user terminal. This is useful in debugging purpose.

7.3.1.9 Unit release()

This function can be used to release the in-memory result components at frovedis server.

7.3.1.10 Int stat()

This function returns the exit status of the scalapack native gesvd routine on calling of which the target result object was obtained.

Chapter 8

LinearRegressionModel

8.1 NAME

LinearRegressionModel - A data structure used in modeling the output of the frovedis server side linear regression algorithms at client spark side.

8.2 SYNOPSIS

import com.nec.frovedis.mllib.regression.LinearRegressionModel

8.2.1 Public Member Functions

Double predict (Vector data)
`RDD[Double]` predict (`RDD[Vector]` data)
Unit save(String path)
Unit save(SparkContext sc, String path)
LinearRegressionModel LinearRegressionModel.load(String path)
LinearRegressionModel LinearRegressionModel.load(SparkContext sc, String path)
Unit debug_print()
Unit release()

8.3 DESCRIPTION

LinearRegressionModel models the output of the frovedis linear regression algorithms, e.g., linear regression, lasso regression and ridge regression. Each of the trainer interfaces of these algorithms aim to optimize an initial model and output the same after optimization.

Note that the actual model with weight parameter etc. is created at frovedis server side only. Spark side LinearRegressionModel contains a unique ID associated with the frovedis server side model, along with some generic information like number of features etc. It simply works like a pointer to the in-memory model at frovedis server.

Any operations, like prediction etc. on a LinearRegressionModel makes a request to the frovedis server along with the unique model ID and the actual job is served by the frovedis server. For functions which returns some output, the result is sent back from frovedis server to the spark client.

8.3.1 Pubic Member Function Documentation

8.3.1.1 Double predict (Vector data)

This function can be used when prediction is to be made on the trained model for a single sample. It returns with the predicted value from the frovedis server.

8.3.1.2 `RDD[Double]` predict (`RDD[Vector]` data)

This function can be used when prediction is to be made on the trained model for more than one samples distributed among spark workers.

It is performed by all the worker nodes in parallel and on success the function returns a `RDD[Double]` object containing the distributed predicted values at worker nodes.

8.3.1.3 LinearRegressionModel LinearRegressionModel.load(String path)

This static function is used to load the target model with data in given filename stored at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data. On success, it returns the loaded model.

8.3.1.4 LinearRegressionModel LinearRegressionModel.load(SparkContext sc, String path)

This is Spark like static API provided for compatibility with spark code. But the "sc" parameter is simply ignored in this case and internally it calls the above load() method as "LinearRegressionModel.load(path)".

8.3.1.5 Unit save(String path)

This function is used to save the target model with given filename. Note that the target model is saved at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data.

8.3.1.6 Unit save(SparkContext sc, String path)

This is Spark like API provided for compatibility with spark code. But the "sc" parameter is simply ignored in this case and internally it calls the above save() method as "save(path)".

8.3.1.7 Unit debug_print()

It prints the contents of the server side model on the server side user terminal. It is mainly useful for debugging purpose.

8.3.1.8 Unit release()

This function can be used to release the existing in-memory model at frovedis server side.

8.4 SEE ALSO

logistic_regression_model, svm_model

Chapter 9

Linear Regression

9.1 NAME

Linear Regression - A regression algorithm to predict the continuous output without any regularization.

9.2 SYNOPSIS

import com.nec.frovedis.mllib.regression.LinearRegressionWithSGD

LinearRegressionModel
LinearRegressionWithSGD.train (`RDD[LabeledPoint]` data,
        Int numIter = 1000,
        Double stepSize = 0.01,
        Double miniBatchFraction = 1.0)

import com.nec.frovedis.mllib.regression.LinearRegressionWithLBFGS

LinearRegressionModel
LinearRegressionWithLBFGS.train (`RDD[LabeledPoint]` data,
        Int numIter = 1000,
        Double stepSize = 0.01,
        Int histSize = 10)

9.3 DESCRIPTION

Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function given by the **squared loss**:

```
L(w;x,y) := 1/2(wTx-y)^2
```

Where the vectors x are the training data examples and y are their corresponding labels which we want to predict. w is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. The method is called linear since it can be expressed as a function of wTx and y. Linear regression does not use any regularizer.

The gradient of the squared loss is: (wTx-y).x

Frovedis provides implementation of linear regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form **min f(w)** is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense nxn approximation to the inverse Hessian (n being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

This module provides a client-server implementation, where the client application is a normal Apache Spark program. Spark has its own mllib providing the Linear Regression support. But that algorithm is slower when comparing with the equivalent Frovedis algorithm (see frovedis manual for ml/linear_regression) with big dataset. Thus in this implementation, a spark client can interact with a frovedis server sending the required spark data for training at frovedis side. Spark RDD data is converted into frovedis compatible data internally and the spark ML call is linked with the respective frovedis ML call to get the job done at frovedis server.

Spark side call for Linear Regression quickly returns, right after submitting the training request to the frovedis server with a dummy LinearRegressionModel object containing the model information like number of features etc. with a unique model ID for the submitted training request.

When operations like prediction will be required on the trained model, spark client sends the same request to frovedis server on the same model (containing the unique ID) and the request is served at frovedis server and output is sent back to the spark client.

9.3.1 Detailed Description

9.3.1.1 LinearRegressionWithSGD.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
minibatchFraction: A double parameter containing the minibatch fraction (Default: 1.0)

Purpose
It trains a linear regression model with stochastic gradient descent with minibatch optimizer, but without any regularizer at frovedis server. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached.

For example,

```
val data = sc.textFile("./sample")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)
```

```
// training a linear regression model with default parameters using SGD
val model = LinearRegressionWithSGD.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(parsedData) // manual creation of frovedis sparse data
val model2 = LinearRegressionWithSGD.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LinearRegressionModel object containing a unique model ID for the training request along with some other general information like number of features etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

9.3.1.2 LinearRegressionWithLBFGS.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
histSize: An integer parameter containing the gradient history size (Default: 10)

Purpose
It trains a linear regression model with LBFGS optimizer, but without any regularizer at frovedis server. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached.

```
val data = sc.textFile("./sample")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a linear regression model with default parameters using LBFGS
val model = LinearRegressionWithLBFGS.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(parsedData) // manual creation of frovedis sparse data
val model2 = LinearRegressionWithLBFGS.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LinearRegressionModel object containing a unique model ID for the training request along with some other general information like number of features etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

9.4 SEE ALSO

linear_regression_model, lasso_regression, ridge_regression, frovedis_sparse_data

Chapter 10

Lasso Regression

10.1 NAME

Lasso Regression - A regression algorithm to predict the continuous output with L1 regularization.

10.2 SYNOPSIS

import com.nec.frovedis.mllib.regression.LassoWithSGD

LinearRegressionModel
LassoWithSGD.train (`RDD[LabeledPoint]` data,
        Int numIter = 1000,
        Double stepSize = 0.01,
        Double regParam = 0.01,
        Double miniBatchFraction = 1.0)

import com.nec.frovedis.mllib.regression.LassoWithLBFGS

LinearRegressionModel
LassoWithLBFGS.train (`RDD[LabeledPoint]` data,
        Int numIter = 1000,
        Double stepSize = 0.01,
        Double regParam = 0.01,
        Int histSize = 10)

10.3 DESCRIPTION

Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function given by the **squared loss**:

```
L(w;x,y) := 1/2(wTx-y)^2
```

Where the vectors x are the training data examples and y are their corresponding labels which we want to predict. w is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. The method is called linear since it can be expressed as a function of wTx and y. Lasso regression uses L1 regularization to address the overfit problem.

The gradient of the squared loss is: (wTx-y).x
The gradient of the regularizer is: sign(w)

Frovedis provides implementation of lasso regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form **min f(w)** is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense nxn approximation to the inverse Hessian (n being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

This module provides a client-server implementation, where the client application is a normal Apache Spark program. Spark has its own mllib providing the Lasso Regression support. But that algorithm is slower when comparing with the equivalent Frovedis algorithm (see frovedis manual for ml/lasso_regression) with big dataset. Thus in this implementation, a spark client can interact with a frovedis server sending the required spark data for training at frovedis side. Spark RDD data is converted into frovedis compatible data internally and the spark ML call is linked with the respective frovedis ML call to get the job done at frovedis server.

Spark side call for Lasso Regression quickly returns, right after submitting the training request to the frovedis server with a dummy LinearRegressionModel object containing the model information like number of features etc. with a unique model ID for the submitted training request.

When operations like prediction will be required on the trained model, spark client sends the same request to frovedis server on the same model (containing the unique ID) and the request is served at frovedis server and output is sent back to the spark client.

10.3.1 Detailed Description

10.3.1.1 LassoWithSGD.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
regParam: A double parameter containing the regularization parameter (Default: 0.01)
minibatchFraction: A double parameter containing the minibatch fraction (Default: 1.0)

Purpose
It trains a linear regression model with stochastic gradient descent with minibatch optimizer and with L1 regularizer at frovedis server. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached.

For example,

```
val data = sc.textFile("./sample")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a linear regression model with default parameters
// using SGD optimizer and L1 regularizer
val model = LassoWithSGD.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(parsedData) // manual creation of frovedis sparse data
val model2 = LassoWithSGD.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LinearRegressionModel object containing a unique model ID for the training request along with some other general information like number of features etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

10.3.1.2 LassoWithLBFGS.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
regParam: A double parameter containing the regularization parameter (Default: 0.01)
histSize: An integer parameter containing the gradient history size (Default: 10)

Purpose
It trains a linear regression model with LBFGS optimizer and with L1 regularizer at frovedis server. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached.

For example,

```
val data = sc.textFile("./sample")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a linear regression model with default parameters
// using LBFGS optimizer and L1 regularizer
val model = LassoWithLBFGS.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(parsedData) // manual creation of frovedis sparse data
val model2 = LassoWithLBFGS.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LinearRegressionModel object containing a unique model ID for the training request along with some other general information like number of features etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

10.4 SEE ALSO

linear_regression_model, linear_regression, ridge_regression, frovedis_sparse_data

Chapter 11

Ridge Regression

11.1 NAME

Ridge Regression - A regression algorithm to predict the continuous output with L2 regularization.

11.2 SYNOPSIS

import com.nec.frovedis.mllib.regression.RidgeRegressionWithSGD

LinearRegressionModel
RidgeRegressionWithSGD.train (`RDD[LabeledPoint]` data,
    Int numIter = 1000,
    Double stepSize = 0.01,
    Double regParam = 0.01,
    Double miniBatchFraction = 1.0)

import com.nec.frovedis.mllib.regression.RidgeRegressionWithLBFGS

LinearRegressionModel
RidgeRegressionWithLBFGS.train (`RDD[LabeledPoint]` data,
    Int numIter = 1000,
    Double stepSize = 0.01,
    Double regParam = 0.01,
    Int histSize = 10)

11.3 DESCRIPTION

Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function given by the **squared loss**:

```
L(w;x,y) := 1/2(wTx-y)^2
```

Where the vectors x are the training data examples and y are their corresponding labels which we want to predict. w is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. The method is called linear since it can be expressed as a function of wTx and y. Ridge regression uses L2 regularization to address the overfit problem.

The gradient of the squared loss is: (wTx-y).x
The gradient of the regularizer is: w

Frovedis provides implementation of ridge regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form **min f(w)** is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense nxn approximation to the inverse Hessian (n being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

This module provides a client-server implementation, where the client application is a normal Apache Spark program. Spark has its own mllib providing the Ridge Regression support. But that algorithm is slower when comparing with the equivalent Frovedis algorithm (see frovedis manual for ml/ridge_regression) with big dataset. Thus in this implementation, a spark client can interact with a frovedis server sending the required spark data for training at frovedis side. Spark RDD data is converted into frovedis compatible data internally and the spark ML call is linked with the respective frovedis ML call to get the job done at frovedis server.

Spark side call for Ridge Regression quickly returns, right after submitting the training request to the frovedis server with a dummy LinearRegressionModel object containing the model information like number of features etc. with a unique model ID for the submitted training request.

When operations like prediction will be required on the trained model, spark client sends the same request to frovedis server on the same model (containing the unique ID) and the request is served at frovedis server and output is sent back to the spark client.

11.3.1 Detailed Description

11.3.1.1 RidgeRegressionWithSGD.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
regParam: A double parameter containing the regularization parameter (Default: 0.01)
minibatchFraction: A double parameter containing the minibatch fraction (Default: 1.0)

Purpose
It trains a linear regression model with stochastic gradient descent with minibatch optimizer and with L2 regularizer at frovedis server. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached.

For example,

```
val data = sc.textFile("./sample")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a linear regression model with default parameters
// using SGD optimizer and L2 regularizer
val model = RidgeRegressionWithSGD.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(parsedData) // manual creation of frovedis sparse data
val model2 = RidgeRegressionWithSGD.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LinearRegressionModel object containing a unique model ID for the training request along with some other general information like number of features etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

11.3.1.2 RidgeRegressionWithLBFGS.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
regParam: A double parameter containing the regularization parameter (Default: 0.01)
histSize: An integer parameter containing the gradient history size (Default: 10)

Purpose
It trains a linear regression model with LBFGS optimizer and with L2 regularizer at frovedis server. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached.

For example,

```
val data = sc.textFile("./sample")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a linear regression model with default parameters
// using LBFGS optimizer and L2 regularizer
val model = RidgeRegressionWithLBFGS.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(parsedData) // manual creation of frovedis sparse data
val model2 = RidgeRegressionWithLBFGS.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LinearRegressionModel object containing a unique model ID for the training request along with some other general information like number of features etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

11.4 SEE ALSO

linear_regression_model, linear_regression, lasso_regression, frovedis_sparse_data

Chapter 12

LogisticRegressionModel

12.1 NAME

LogisticRegressionModel - A data structure used in modeling the output of the frovedis server side logistic regression algorithm at client spark side.

12.2 SYNOPSIS

import com.nec.frovedis.mllib.classification.LogisticRegressionModel

12.2.1 Public Member Functions

Double predict (Vector data)
`RDD[Double]` predict (`RDD[Vector]` data)
Unit save(String path)
Unit save(SparkContext sc, String path)
LogisticRegressionModel LogisticRegressionModel.load(String path)
LogisticRegressionModel LogisticRegressionModel.load(SparkContext sc, String path)
Unit debug_print()
Unit release()

12.3 DESCRIPTION

LogisticRegressionModel models the output of the frovedis logistic regression algorithm, the trainer interface of which aims to optimize an initial model and outputs the same after optimization.

Note that the actual model with weight parameter etc. is created at frovedis server side only. Spark side LogisticRegressionModel contains a unique ID associated with the frovedis server side model, along with some generic information like number of features etc. It simply works like a pointer to the in-memory model at frovedis server.

Any operations, like prediction etc. on a LogisticRegressionModel makes a request to the frovedis server along with the unique model ID and the actual job is served by the frovedis server. For functions which returns some output, the result is sent back from frovedis server to the spark client.

12.3.1 Pubic Member Function Documentation

12.3.1.1 Double predict (Vector data)

This function can be used when prediction is to be made on the trained model for a single sample. It returns with the predicted value from the frovedis server.

12.3.1.2 `RDD[Double]` predict (`RDD[Vector]` data)

This function can be used when prediction is to be made on the trained model for more than one samples distributed among spark workers.

It is performed by all the worker nodes in parallel and on success the function returns a `RDD[Double]` object containing the distributed predicted values at worker nodes.

12.3.1.3 LogisticRegressionModel LogisticRegressionModel.load(String path)

This static function is used to load the target model with data in given filename stored at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data. On success, it returns the loaded model.

12.3.1.4 LogisticRegressionModel LogisticRegressionModel.load(SparkContext sc, String path)

This is Spark like static API provided for compatibility with spark code. But the “sc” parameter is simply ignored in this case and internally it calls the above load() method as “LogisticRegressionModel.load(path)”.

12.3.1.5 Unit save(String path)

This function is used to save the target model with given filename. Note that the target model is saved at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data.

12.3.1.6 Unit save(SparkContext sc, String path)

This is Spark like API provided for compatibility with spark code. But the “sc” parameter is simply ignored in this case and internally it calls the above save() method as “save(path)”.

12.3.1.7 Unit debug_print()

It prints the contents of the server side model on the server side user terminal. It is mainly useful for debugging purpose.

12.3.1.8 Unit release()

This function can be used to release the existing in-memory model at frovedis server side.

12.4 SEE ALSO

linear_regression_model, svm_model

Chapter 13

Logistic Regression

13.1 NAME

Logistic Regression - A classification algorithm to predict the binary output with logistic loss.

13.2 SYNOPSIS

import com.nec.frovedis.mllib.classification.LogisticRegressionWithSGD

LogisticRegressionModel
LogisticRegressionWithSGD.train(`RDD[LabeledPoint]` data,
    Int numIter = 1000,
    Double stepSize = 0.01,
    Double miniBatchFraction = 1.0,
    Double regParam = 0.01)

import com.nec.frovedis.mllib.classification.LogisticRegressionWithLBFGS

LogisticRegressionModel
LogisticRegressionWithLBFGS.train(`RDD[LabeledPoint]` data,
    Int numIter = 1000,
    Double stepSize = 0.01,
    Int histSize = 10,
    Double regParam = 0.01)

13.3 DESCRIPTION

Classification aims to divide the items into categories. The most common classification type is binary classification, where there are two categories, usually named positive and negative. Frovedis supports binary classification algorithm only.

Logistic regression is widely used to predict a binary response. It is a linear method with the loss function given by the **logistic loss**:

```
L(w;x,y) := log(1 + exp(-ywTx))
```

Where the vectors x are the training data examples and y are their corresponding labels (Frovedis considers negative response as -1 and positive response as 1, but when calling from Spark interface, user should pass 0 for negative response and 1 for positive response according to the Spark requirement) which we want to predict. w is the linear model (also called as weight) which uses a single weighted sum of features to make a prediction. Frovedis Logistic Regression supports ZERO, L1 and L2 regularization to address the overfit problem. But when calling from Spark interface, it supports the default L2 regularization only.

The gradient of the logistic loss is: -y(1 - 1 / (1 + exp(-ywTx))).x
The gradient of the L1 regularizer is: sign(w)
And The gradient of the L2 regularizer is: w

For binary classification problems, the algorithm outputs a binary logistic regression model. Given a new data point, denoted by x, the model makes predictions by applying the logistic function:

```
f(z) := 1 / 1 + exp(-z)
```

Where z = wTx. By default (threshold=0.5), if f(wTx) > 0.5, the response is positive (1), else the response is negative (0).

Frovedis provides implementation of logistic regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form **min f(w)** is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense nxn approximation to the inverse Hessian (n being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

This module provides a client-server implementation, where the client application is a normal Apache Spark program. Spark has its own mllib providing the Logistic Regression support. But that algorithm is slower when comparing with the equivalent Frovedis algorithm (see frovedis manual for ml/logistic_regression) with big dataset. Thus in this implementation, a spark client can interact with a frovedis server sending the required spark data for training at frovedis side. Spark RDD data is converted into frovedis compatible data internally and the spark ML call is linked with the respective frovedis ML call to get the job done at frovedis server.

Spark side call for Logictic Regression quickly returns, right after submitting the training request to the frovedis server with a dummy LogicticRegressionModel object containing the model information like threshold value etc. with a unique model ID for the submitted training request.

When operations like prediction will be required on the trained model, spark client sends the same request to frovedis server on the same model (containing the unique ID) and the request is served at frovedis server and output is sent back to the spark client.

13.3.1 Detailed Description

13.3.1.1 LogisticRegressionWithSGD.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)

minibatchFraction: A double parameter containing the minibatch fraction (Default: 1.0)
regParam: A double parameter containing the regularization parameter (Default: 0.01)

Purpose
It trains a logistic regression model with stochastic gradient descent with minibatch optimizer and with default L2 regularizer. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached. After the training, it returns the trained logistic regression model.

For example,

```
val data = MLUtils.loadLibSVMFile(sc, "./sample")
val splits = data.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a logistic regression model with default parameters using SGD
val model = LogisticRegressionWithSGD.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(data) // manual creation of frovedis sparse data
val model2 = LogisticRegressionWithSGD.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LogisticRegressionModel object containing a unique model ID for the training request along with some other general information like threshold (default 0.5) etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

13.3.1.2 LogisticRegressionWithLBFGS.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
histSize: An integer parameter containing the gradient history size (Default: 1.0)
regParam: A double parameter containing the regularization parameter (Default: 0.01)

Purpose
It trains a logistic regression model with LBFGS optimizer and with default L2 regularizer. It starts with an

initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached. After the training, it returns the trained logistic regression model.

For example,

```
val data = MLUtils.loadLibSVMFile(sc, "./sample")
val splits = data.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a logistic regression model with default parameters using LBFGS
val model = LogisticRegressionWithLBFGS.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(data) // manual creation of frovedis sparse data
val model2 = LogisticRegressionWithLBFGS.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a LogisticRegressionModel object containing a unique model ID for the training request along with some other general information like threshold (default 0.5) etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

13.4 SEE ALSO

logistic_regression_model, linear_svm, frovedis_sparse_data

Chapter 14

SVMModel

14.1 NAME

SVMModel - A data structure used in modeling the output of the frovedis server side linear SVM (Support Vector Machine) algorithm, at client spark side.

14.2 SYNOPSIS

import com.nec.frovedis.mllib.classification.SVMModel

14.2.1 Public Member Functions

Double predict (Vector data)
`RDD[Double]` predict (`RDD[Vector]` data)
Unit save(String path)
Unit save(SparkContext sc, String path)
SVMModel SVMModel.load(String path)
SVMModel SVMModel.load(SparkContext sc, String path)
Unit debug_print()
Unit release()

14.3 DESCRIPTION

SVMModel models the output of the frovedis linear SVM (Support Vector Machine) algorithm, the trainer interface of which aims to optimize an initial model and outputs the same after optimization.

Note that the actual model with weight parameter etc. is created at frovedis server side only. Spark side SVMModel contains a unique ID associated with the frovedis server side model, along with some generic information like number of features etc. It simply works like a pointer to the in-memory model at frovedis server.

Any operations, like prediction etc. on a SVMModel makes a request to the frovedis server along with the unique model ID and the actual job is served by the frovedis server. For functions which returns some output, the result is sent back from frovedis server to the spark client.

14.3.1 Pubic Member Function Documentation

14.3.1.1 Double predict (Vector data)

This function can be used when prediction is to be made on the trained model for a single sample. It returns with the predicted value from the frovedis server.

14.3.1.2 `RDD[Double]` predict (`RDD[Vector]` data)

This function can be used when prediction is to be made on the trained model for more than one samples distributed among spark workers.

It is performed by all the worker nodes in parallel and on success the function returns a `RDD[Double]` object containing the distributed predicted values at worker nodes.

14.3.1.3 SVMModel SVMModel.load(String path)

This static function is used to load the target model with data in given filename stored at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data. On success, it returns the loaded model.

14.3.1.4 SVMModel SVMModel.load(SparkContext sc, String path)

This is Spark like static API provided for compatibility with spark code. But the "sc" parameter is simply ignored in this case and internally it calls the above load() method as "SVMModel.load(path)".

14.3.1.5 Unit save(String path)

This function is used to save the target model with given filename. Note that the target model is saved at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data.

14.3.1.6 Unit save(SparkContext sc, String path)

This is Spark like API provided for compatibility with spark code. But the "sc" parameter is simply ignored in this case and internally it calls the above save() method as "save(path)".

14.3.1.7 Unit debug_print()

It prints the contents of the server side model on the server side user terminal. It is mainly useful for debugging purpose.

14.3.1.8 Unit release()

This function can be used to release the existing in-memory model at frovedis server side.

14.4 SEE ALSO

linear_regression_model, logistic_regression_model

Chapter 15

Linear SVM

15.1 NAME

Linear SVM (Support Vector Machines) - A classification algorithm to predict the binary output with hinge loss.

15.2 SYNOPSIS

import com.nec.frovedis.mllib.classification.SVMWithSGD

SVMModel
SVMWithSGD.train(`RDD[LabeledPoint]` data,
        Int numIter = 1000,
        Double stepSize = 0.01,
        Double regParam = 0.01,
        Double miniBatchFraction = 1.0)

import com.nec.frovedis.mllib.classification.SVMWithLBFGS

SVMModel
SVMWithLBFGS.train(`RDD[LabeledPoint]` data,
        Int numIter = 1000,
        Double stepSize = 0.01,
        Double regParam = 0.01,
        Int histSize = 10)

15.3 DESCRIPTION

Classification aims to divide items into categories. The most common classification type is binary classification, where there are two categories, usually named positive and negative. Frovedis supports binary classification algorithms only.

The Linear SVM is a standard method for large-scale classification tasks. It is a linear method with the loss function given by the **hinge loss**:

```
L(w;x,y) := max{0, 1-ywTx}
```

Where the vectors x are the training data examples and y are their corresponding labels (Frovedis considers negative response as -1 and positive response as 1, but when calling from Spark interface, user should pass 0 for negative response and 1 for positive response according to the Spark requirement) which we want to predict. w is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. Linear SVM supports ZERO, L1 and L2 regularization to address the overfit problem. But when calling from Spark interface, it supports the default L2 regularization only.

The gradient of the hinge loss is: -y.x, if ywTx < 1, 0 otherwise.
The gradient of the L1 regularizer is: sign(w)
And The gradient of the L2 regularizer is: w

For binary classification problems, the algorithm outputs a binary svm model. Given a new data point, denoted by x, the model makes predictions based on the value of wTx.

By default (threshold=0), if wTx >= 0, then the response is positive (1), else the response is negative (0).

Frovedis provides implementation of linear SVM with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form **min f(w)** is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense nxn approximation to the inverse Hessian (n being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

This module provides a client-server implementation, where the client application is a normal Apache Spark program. Spark has its own mllib providing the Linear SVM support. But that algorithm is slower when comparing with the equivalent Frovedis algorithm (see frovedis manual for ml/linear_svm) with big dataset. Thus in this implementation, a spark client can interact with a frovedis server sending the required spark data for training at frovedis side. Spark RDD data is converted into frovedis compatible data internally and the spark ML call is linked with the respective frovedis ML call to get the job done at frovedis server.

Spark side call for Linear SVM quickly returns, right after submitting the training request to the frovedis server with a dummy SVMModel object containing the model information like threshold value etc. with a unique model ID for the submitted training request.

When operations like prediction will be required on the trained model, spark client sends the same request to frovedis server on the same model (containing the unique ID) and the request is served at frovedis server and output is sent back to the spark client.

15.3.1 Detailed Description

15.3.1.1 SVMWithSGD.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
regParam: A double parameter containing the regularization parameter (Default: 0.01)
minibatchFraction: A double parameter containing the minibatch fraction (Default: 1.0)

Purpose
It trains an svm model with stochastic gradient descent with minibatch optimizer and with default L2

regularizer. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached. After the training, it returns the trained svm model.

For example,

```
val data = MLUtils.loadLibSVMFile(sc, "./sample")
val splits = data.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training a svm model with default parameters using SGD
val model = SVMWithSGD.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(data) // manual creation of frovedis sparse data
val model2 = SVMWithSGD.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with an SVMModel object containing a unique model ID for the training request along with some other general information like threshold (default 0.0) etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

15.3.1.2 SVMWithLBFGS.train()

Parameters
data: A `RDD[LabeledPoint]` containing spark-side distributed sparse training data
numIter: An integer parameter containing the maximum number of iteration count (Default: 1000)
stepSize: A double parameter containing the learning rate (Default: 0.01)
regParam: A double parameter containing the regularization parameter (Default: 0.01)
histSize: An integer parameter containing the gradient history size (Default: 1.0)

Purpose
It trains an svm model with LBFGS optimizer and with default L2 regularizer. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved (default convergence tolerance value is 0.001) or maximum iteration count is reached. After the training, it returns the trained svm model.

For example,

```
val data = MLUtils.loadLibSVMFile(sc, "./sample")
val splits = data.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)
val tvec = test.map(_.features)

// training an svm model with default parameters using LBFGS
val model = SVMWithLBFGS.train(training)
// cross-validation of the trained model on 20% test data
model.predict(tvec).collect.foreach(println)
```

Note that, inside the train() function spark side sparse data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(data) // manual creation of frovedis sparse data
val model2 = SVMWithLBFGS.train(fdata) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with an SVMModel object containing a unique model ID for the training request along with some other general information like threshold (default 0.0) etc. But it does not contain any weight values. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

15.4 SEE ALSO

svm_model, logistic_regression, frovedis_sparse_data

Chapter 16

MatrixFactorizationModel

16.1 NAME

MatrixFactorizationModel - A data structure used in modeling the output of the frovedis server side matrix factorization using ALS algorithm, at client spark side.

16.2 SYNOPSIS

import com.nec.frovedis.mllib.recommendation.MatrixFactorizationModel

16.2.1 Public Member Functions

Double predict (Int uid, Int pid)
`RDD[Rating]` predict (`RDD[(Int, Int)]` usersProducts)
`Array[Rating]` recommendProducts(Int uid, Int num)
`Array[Rating]` recommendUsers(Int pid, Int num)
Unit save(String path)
Unit save(SparkContext sc, String path)
MatrixFactorizationModel MatrixFactorizationModel.load(String path)
MatrixFactorizationModel MatrixFactorizationModel.load(SparkContext sc, String path)
Unit debug_print()
Unit release()

16.3 DESCRIPTION

MatrixFactorizationModel models the output of the frovedis matrix factorization using ALS (alternating least square) algorithm, the trainer interface of which aims to optimize an initial model and outputs the same after optimization.

Note that the actual model with user/product features etc. is created at frovedis server side only. Spark side MatrixFactorizationModel contains a unique ID associated with the frovedis server side model, along with some generic information like rank value etc. It simply works like a pointer to the in-memory model at frovedis server.

Any operations, like prediction etc. on a MatrixFactorizationModel makes a request to the frovedis server along with the unique model ID and the actual job is served by the frovedis server. For functions which returns some output, the result is sent back from frovedis server to the spark client.

16.3.1 Pubic Member Function Documentation

16.3.1.1 Double predict (Int uid, Int pid)

This method can be used on a trained model in order to predict the rating confidence value for the given product id, by the given user id.

"uid" should be in between 1 to M, where M is the number of users in the given data. And "pid" should be in between 0 to N, where N is the number of products in the given data.

16.3.1.2 `RDD[Rating]` predict (`RDD[(Int, Int)]` usersProducts)

This method can be used to predict the rating confidence values for a given list of pair of some user ids and product ids.

In the list of pairs, "uid" should be in between 1 to M and "pid" should be in between 1 to N, where M is the number of users and N is the number of products in the given data.

It is performed by all the worker nodes in parallel and on success the function returns a `RDD[Rating]` object containing the distributed predicted ratings at worker nodes.

16.3.1.3 `Array[Rating]` recommendProducts(Int uid, Int num)

This method can be used to recommend given "num" number of products for the user with given user id in sorted order (highest scored products to lowest scored products).

"uid" should be in between 1 to M, where M is the number of users in the given data. On success, it returns an array containing ratings for the recommended products by the given user.

16.3.1.4 `Array[Rating]` recommendUsers(Int pid, Int num)

This method can be used to recommend given "num" number of users for the product with given product id in sorted order (user with highest scores to user with lowest scores).

"pid" should be in between 1 to N, where N is the number of products in the given data. On success, it returns an array containing ratings for the recommended users the given product.

16.3.1.5 MatrixFactorizationModel MatrixFactorizationModel.load(String path)

This static function is used to load the target model with data in given filename stored at frovedis server side at specified location (filename with relative/aboslute path) as little-endian binary data. On success, it returns the loaded model.

16.3.1.6 MatrixFactorizationModel MatrixFactorizationModel.load(SparkContext sc, String path)

This is Spark like static API provided for compatibility with spark code. But the "sc" parameter is simply ignored in this case and internally it calls the above load() method as "MatrixFactorizationModel.load(path)".

16.3.1.7 Unit save(String path)

This function is used to save the target model with given filename. Note that the target model is saved at frovedis server side at specified location (filename with relative/aboslute path) as little-endian binary data.

16.3.1.8 Unit save(SparkContext sc, String path)

This is Spark like API provided for compatibility with spark code. But the "sc" parameter is simply ignored in this case and internally it calls the above save() method as "save(path)".

16.3.1.9 Unit debug_print()

It prints the contents of the server side model on the server side user terminal. It is mainly useful for debugging purpose.

16.3.1.10 Unit release()

This function can be used to release the existing in-memory model at frovedis server side.

Chapter 17

Matrix Factorization using ALS

17.1 NAME

Matrix Factorization using ALS - A matrix factorization algorithm commonly used for recommender systems.

17.2 SYNOPSIS

import com.nec.frovedis.mllib.recommendation.ALS

```
MatrixFactorizationModel
ALS.trainImplicit (RDD[Rating] data,
        Int rank,
        Int iterations = 100,
        Double lambda = 0.01,
        Double alpha = 0.01,
        Long seed = 0)
```

17.3 DESCRIPTION

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix. Frovedis currently supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries.

Like Apache Spark, Frovedis also uses the alternating least squares (ALS) algorithm to learn these latent factors. The algorithm is based on a paper "Collaborative Filtering for Implicit Feedback Datasets" by Hu, et al.

This module provides a client-server implementation, where the client application is a normal Apache Spark program. Spark has its own mllib providing the ALS support. But that algorithm is slower when comparing with the equivalent Frovedis algorithm (see frovedis manual for ml/als) with big dataset. Thus in this implementation, a spark client can interact with a frovedis server sending the required spark data for training at frovedis side. Spark RDD data is converted into frovedis compatible data internally and the spark ML call is linked with the respective frovedis ML call to get the job done at frovedis server.

Spark side call for ALS.trainImplicit() quickly returns, right after submitting the training request to the frovedis server with a dummy MatrixFactorizationModel object containing the model information like rank etc. with a unique model ID for the submitted training request.

When operations like prediction will be required on the trained model, spark client sends the same request to frovedis server on the same model (containing the unique ID) and the request is served at frovedis server and output is sent back to the spark client.

17.3.1 Detailed Description

17.3.1.1 ALS.trainImplicit()

Parameters
data: A `RDD[Rating]` containing spark-side distributed rating data
rank: An integer parameter containing the number of latent factors (also known as rank)
iterations: An integer parameter containing the maximum number of iteration count (Default: 100)
lambda: A double parameter containing the regularization parameter (Default: 0.01)
alpha: A double parameter containing the learning rate (Default: 0.01)
seed: A Long parameter containing the seed value to initialize the model structures with random values

Purpose
It trains a MatrixFactorizationModel with alternating least squares (ALS) algorithm. It starts with initializing the model structures of the size MxF and NxF (where M is the number of users and N is the products in the given rating matrix and F is the given rank) with random values and keeps updating them until maximum iteration count is reached. After the training, it returns the trained MatrixFactorizationModel model.

For example,

```
// -------- data loading from sample rating (COO) file at Spark side--------
val data = sc.textFile("./sample")
val ratings = data.map(_.split(',') match { case Array(user, item, rate) =>
                    Rating(user.toInt, item.toInt, rate.toDouble)
              })

// Build the recommendation model using ALS with default parameters
val model = ALS.trainImplicit(ratings,4)
println("Rating: " + model.predict(1,2)) // predict the rating for 2nd product by 1st user
```

Note that, inside the trainImplicit() function spark side COO rating data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the “data” parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData() // an empty object
fdata.loadcoo(ratings) // manual creation of frovedis sparse data
val model2 = ALS.trainImplicit(fdata,4) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a MatrixFactorizationModel object containing a unique model ID for the training request along with some other general information like rank etc. But it does not contain any user/product components. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

17.4 SEE ALSO

matrix_factorization_model, frovedis_sparse_data

Chapter 18

KMeansModel

18.1 NAME

KMeansModel - A data structure used in modeling the output of the frovedis server side kmeans clustering algorithm at client spark side.

18.2 SYNOPSIS

import com.nec.frovedis.mllib.clustering.KMeansModel

18.2.1 Public Member Functions

Int predict (Vector data)
`RDD[Int]` predict (`RDD[Vector]` data)
Int getK()
Unit save(String path)
Unit save(SparkContext sc, String path)
KMeansModel KMeansModel.load(String path)
KMeansModel KMeansModel.load(SparkContext sc, String path)
Unit debug_print()
Unit release()

18.3 DESCRIPTION

KMeansModel models the output of the frovedis kmeans clustering algorithm.

Note that the actual model with centroid information is created at frovedis server side only. Spark side KMeansModel contains a unique ID associated with the frovedis server side model, along with some generic information like k value etc. It simply works like a pointer to the in-memory model at frovedis server.

Any operations, like prediction etc. on a KMeansModel makes a request to the frovedis server along with the unique model ID and the actual job is served by the frovedis server. For functions which returns some output, the result is sent back from frovedis server to the spark client.

18.3.1 Pubic Member Function Documentation

18.3.1.1 Int predict (Vector data)

This function can be used when prediction is to be made on the trained model for a single sample. It returns with the predicted value from the frovedis server.

18.3.1.2 `RDD[Int]` predict (`RDD[Vector]` data)

This function can be used when prediction is to be made on the trained model for more than one samples distributed among spark workers.

It is performed by all the worker nodes in parallel and on success the function returns a `RDD[Int]` object containing the distributed predicted values at worker nodes.

18.3.1.3 Int getK()

It returns the number of clusters in the target model.

18.3.1.4 KMeansModel KMeansModel.load(String path)

This static function is used to load the target model with data in given filename stored at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data. On success, it returns the loaded model.

18.3.1.5 KMeansModel KMeansModel.load(SparkContext sc, String path)

This is Spark like static API provided for compatibility with spark code. But the “sc” parameter is simply ignored in this case and internally it calls the above load() method as “KMeansModel.load(path)”.

18.3.1.6 Unit save(String path)

This function is used to save the target model with given filename. Note that the target model is saved at frovedis server side at specified location (filename with relative/absolute path) as little-endian binary data.

18.3.1.7 Unit save(SparkContext sc, String path)

This is Spark like API provided for compatibility with spark code. But the “sc” parameter is simply ignored in this case and internally it calls the above save() method as “save(path)”.

18.3.1.8 Unit debug_print()

It prints the contents of the server side model on the server side user terminal. It is mainly useful for debugging purpose.

18.3.1.9 Unit release()

This function can be used to release the existing in-memory model at frovedis server side.

Chapter 19

kmeans

19.1 NAME

kmeans - A clustering algorithm commonly used in EDA (exploratory data analysis).

19.2 SYNOPSIS

import com.nec.frovedis.mllib.clustering.KMeans

```
KMeansModel
KMeans.train (RDD[Vector] data,
        Int k,
        Int iterations = 100,
        Long seed = 0,
        Double epsilon = 0.01)
```

19.3 DESCRIPTION

Clustering is an unsupervised learning problem whereby we aim to group subsets of entities with one another based on some notion of similarity. K-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters (K).

This module provides a client-server implementation, where the client application is a normal Apache Spark program. Spark has its own mllib providing the KMeans support. But that algorithm is slower when comparing with the equivalent Frovedis algorithm (see frovedis manual for ml/kmeans) with big dataset. Thus in this implementation, a spark client can interact with a frovedis server sending the required spark data for training at frovedis side. Spark RDD data is converted into frovedis compatible data internally and the spark ML call is linked with the respective frovedis ML call to get the job done at frovedis server.

Spark side call for KMeans.train() quickly returns, right after submitting the training request to the frovedis server with a dummy KMeansModel object containing the model information like k value etc. with a unique model ID for the submitted training request.

When operations like prediction will be required on the trained model, spark client sends the same request to frovedis server on the same model (containing the unique ID) and the request is served at frovedis server and output is sent back to the spark client.

19.3.1 Detailed Description

19.3.1.1 KMeans.train()

Parameters
data: A `RDD[Vector]` containing spark-side data points
k: An integer parameter containing the number of clusters
iterations: An integer parameter containing the maximum number of iteration count (Default: 100)
seed: A long parameter containing the seed value to generate the random rows from the given data samples (Default: 0)
epsilon: A double parameter containing the epsilon value (Default: 0.01)

Purpose
It clusters the given data points into a predefined number (k) of clusters.
After the successful clustering, it returns the KMeansModel.

For example,

```
// -------- data loading from sample kmeans data file at Spark side--------
val data = sc.textFile("./sample")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))
val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
val training = splits(0)
val test = splits(1)

// Build the cluster using KMeans with default parameters
val model = KMeans.train(training,2)

// Evaluate the model on test data
model.predict(test).foreach(println)
```

Note that, inside the train() function spark data is converted into frovedis side sparse data and after the training, frovedis side sparse data is released from the server memory. But if the user needs to store the server side constructed sparse data for some other operations, he may also like to pass the FrovedisSparseData object as the value of the "data" parameter. In that case, the user needs to explicitly release the server side sparse data when it will no longer be needed.

For example,

```
val fdata = new FrovedisSparseData(parsedData) // manual creation of frovedis sparse data
val model2 = KMeans.train(fdata,2) // passing frovedis sparse data
fdata.release() // explicit release of the server side data
```

Return Value
This is a non-blocking call. The control will return quickly, right after submitting the training request at frovedis server side with a KMeansModel object containing a unique model ID for the training request along with some other general information like k value etc. But it does not contain any centroid information. It simply works like a spark side pointer of the actual model at frovedis server side. It may be possible that the training is not completed at the frovedis server side even though the client spark side train() returns with a pseudo model.

19.4 SEE ALSO

kmeans_model, frovedis_sparse_data