Ridge Regression

NAME

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

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)
```

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.

Detailed Description

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.

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.

SEE ALSO

linear_regression_model, linear_regression, lasso_regression, frovedis_sparse_data