Linear SVM

NAME

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

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)

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.

Detailed Description

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.

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.

SEE ALSO

svm_model, logistic_regression, frovedis_sparse_data