Data Mining Tasks in Apache Spark

Link to Github Repository :
https://github.com/VrajSoni/Data-Mining-Tasks-in-Apache-Spark

1.K Means Clustering

Explanation

The csv file is loaded as a dataframe and stored in a variable named ‘data’. This implementation uses a 2D list of Double Accumulators of size in K x Features in order to calculate the new clusters.(K is the number of Clusters the user wishes to have).

The iterations start after picking K random rows from the dataframe as Centroids. In each iteration we broadcast the current list of Centroids. Now, for each row of the dataframe, we find the euclidean distance of this vector from all the centroids. The row(data point) is then assigned to the minimum distanced centroid and it’s feature values are added to the respective centroid’s double accumulator variable. After each row is assigned a cluster, new centroids are calculated from the average of these stored double accumulators.

The new Centroids are then compared to previous Centroid, if they are the same then the iterations are terminated, otherwise the iterations are continued further.

Pseudocode

```
dataRDD <- Load data as RDD
Features <-  Number of features on each vector of the sample
Sum <- 2D(K x Features) list of Double Accumulators
Iterations <- Number of Iterations
C <- Select random K rows from dataRDD
Cnt <- Integer Accumulator

For iter = 1 to Iterations
     Centroids <- Broadcast(C)
     Foreach(s in sum)
```

```
          s <- 0
     END FOR
     Cnt <- 0
     Foreach(row in data)
          Distances <- Initialize a list of Doubles
          For j = 1 to K
               dist <- 0.0
               For f= 1 to Features
                    dist += (Centroids[j][f] - row[f]) *
(Centroids[j][f] - row[f])
               END FOR
               dist <- square_root(dist)
               APPEND dist in Distances
               END FOR

          Index <- Index of minimum Distance
     END FOR
          For j = 1 to Features
               Sum[Index][j] += row[j]
          END FOR

     newCentroids <- List of Rows
     For j = 1 to Clusters
          list <- Initialize a list of Doubles
          For f = 1 to Features
               APPEND Average of Sum[j][f] in list
          APPEND list to newCentroids as Row
          END FOR
     END FOR
     IF newCentroids = C
          BREAK
     ELSE
          C = newCentroids
END FOR
Print(Centroids)
```

2. Multi Linear Regression(Using Gradient Descent)

Explanation

The csv file is loaded as a dataframe and stored in a variable named ‘data’. The A Double accumulator deriv is used to calculate the derivative for every omega.

The omega values are initialized with a value of 1 and the iterations are performed. In each iteration, first the current values for omega are broadcast.Firstly,we accumulate the predicted values for each row of the data frame using a list of Double Accumulators. This list of predicted values is broadcast as well. For each of the omega, the derivative value is calculated using these predicted values, learning rate and the number of rows in the data frame. These new values of omega after each iteration are stored and then broadcast in the next iteration for further use.

Pseudocode

```
dataRDD <- Load data as RDD
Iterations <- Number of Iterations
Features <-  Number of features on each vector of the sample
M <- Number of Rows in data
W <- 1D(Size = K + 1) List with all values as 1
Alpha <- Learning Rate for Gradient Descent
Deriv <- Double Accumulator
Cnt <- Atomic Integer

For iter = 1 to iterations
     Omega = Broadcast(W)
     Predicted <- List(Size = M) of Double Accumulators
     Cnt <- 0

     Foreach(row in data)
          For w = 1 to Features + 1
               IF w != 0
                    predicted[Cnt] += Omega[w] * row[w-1]
               ELSE
                    predicted[Cnt] += Omega[w]
          Increment Cnt
       END FOR
```

```
     END FOR

     pred = Broadcast(predicted)
     For w = 1 to Features + 1
          Cnt <- 0
          Deriv <- 0
          Foreach(row in data)
               Xij <- 1
               IF w != 0
                    Xij <- row[w-1]
               Error <- pred[Cnt] - row[Features]
               Deriv += Error * Xij
               Increment Cnt
          END FOR
     END FOR
          W[w] = Omega[w] - (Alpha * Deriv)/M
END FOR
     Print(W)
```

3. KNN Algorithm

Explanation

For this algorithm a training dataset is required in order to find the category of the desired inputs. For every point in the test dataset, we will require a map to store the (distance,category) pair in a hashmap to determine that the point at given distance is of which category. We will find the euclidean distance of each test data point with every train data point and store it in the broadcasted hashmap.

A counts array is used to store the count of each category of the k nearest neighbours. The index of maximum count in the counts array will determine the category of the test data point.

Pseudocode

```
trainData <- Load the training data as RDD
rows <- Number of rows in the dataset
cols <- Number of features in a vector
k <- Number of nearest neighbours to include in majority
voting process
totalCategories <- The total categories in the dataset
cnt <- Atomic Integer
testData <- The points whose category is to be determined

For i = 0 to testData.size
     mp <- Hashmap to store Distance,Category pairs
     distanceMap <- Broadcast(mp)
     final int index <- i
     Foreach(row in trainData)
          dist <- 0.0
          For j = 1 to cols
               testVal <- testData[index][j]
               trainVal <- row[j]
               dist <- dist + (testVal - trainVal)*(testVal -
trainVal)
          End For
     dist <- square root of dist
     store (dist,row[cols]) in distanceMap
     End Foreach
```

```
     counts <- Array to store count of all categories of the
k nearest neighbours
     mp <- value of distanceMap
     Iterator <- iterator to iterate through mp
     temp <- k
     Foreach(entry in iterator)
          IF temp > 0 :
               counts[entry.value()]++;
          temp<- temp - 1
     End Foreach
     category <- get the index of maximum count from counts
array
     Print("Category of point" i "is" category)
End For
```