README.md

Distributed Coordinate Descent

1.Coordinate descent algorithm

Coordinate descent is a minimization method where the function is minimized one coordinate at a time. The core principle for coordinate descent algorithm is to minimize function F(X) by minimizing one direction at a time. The simplest implementation of same is using a cyclic coordinate descent algorithm. The same has been implemented below.

2. Distributed Coordinate descent algorithm

Following is the implementation of the distributed coordinate descent algorithm as per the paper below. MPI is used for communication among different processors executing the task. For verification of the same KDD cup 98 challenge dataset is used. Here objective is to use regression methods to predict the value of possible donation. Furthermore the dataset has lot of features but many can be dropped.

Rendle, Steffen, et al. Robust large-scale machine learning in the cloud. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2016.

3. Project Structure

```
drwxrwxr-x  8 foo foo  4096 Jun  1 14:11 ./
drwxrwxr-x 10 foo foo  4096 May 21 10:32 ../
drwxrwxr-x  3 foo foo  4096 May 21 10:32 Data/
-rw-rw-r--  1 foo foo  2007 Jun  1 13:32 kddcup98.py
-rw-rw-r--  1 foo foo   120 Jun  1 14:11 kddcup.sh
drwxrwxr-x  3 foo foo  4096 May  5 22:27 Logger/
drwxrwxr-x  2 foo foo  4096 Jun  1 12:52 Output/
drwxrwxr-x  3 foo foo  4096 May 10 17:57 PerformanceMetric/
-rw-rw-r--  1 foo foo 12604 Jun  1 14:05 README.md
drwxrwxr-x  3 foo foo  4096 May 21 12:47 Regression/
-rw-rw-r--  1 foo foo 95049 Jun  1 14:11 timeit.log
-rw-rw-r--  1 foo foo 27452 May 21 11:05 Untitled.ipynb
drwxrwxr-x  3 foo foo  4096 May 14 15:58 Utils/
```

3.1 Logger

This module includes script for storing the time taken for execution.

```
import time
import logging

logger = logging.getLogger('timeLogger')
logger.setLevel(logging.DEBUG)
fh = logging.FileHandler('timeit.log')
fh.setLevel(logging.DEBUG)
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
logger.addHandler(fh)
logger.addHandler(ch)


def timeit(f):
      def timed(*args, **kw):
            ts = time.time()
            result = f(*args, **kw)
```

```
            te = time.time()
            logger.info('Function:%r  Execution Time: %2.4f sec' %(f.__name__, te-ts))
            return result
      return timed
```

For MPI processes it is necessary to calculate time taken by some function to execute. For this the function is wrapped between timers and difference between end and start time is the total execution time by that function. This increases unnecessary code foot print which is reduced by this module. The above method aids in implementation of **Decorator** design pattern. The function that needs to be timed should be decorated as shown below

```
@timeit
def foo():
      return 0
```

Each time this function is called its execution time is calculated and stored in timeit.log file.

3.2 Utils

This module includes all frequently used helper methods.

3.2.1 Data Reader

```
import scipy.sparse as sp
import numpy as np

from sklearn.datasets import load_svmlight_file
from os import  listdir
from os.path import basename, join
from Logger.TimeIt import timeit
"""This module incudes common
utility functions for IO"""

@timeit
def listAllFiles(basePath):
      pathList = []
      for file in listdir(basePath):
            pathList.append(join(basePath, file))
      return pathList

@timeit
def svmLightToNPVectors(pathList, featureCount, dType, includeBias = False):
      csrMatrixList = []
      y = []
      for file in pathList:
            output = load_svmlight_file(file, n_features = featureCount, dtype = dType)
            csrMatrixList.append(output[0])
            y.extend(output[1])
      x = sp.vstack(csrMatrixList)
      if includeBias:
            x = sp.hstack((np.ones((x.shape[0], 1)), x))
      return x.tocsr(), np.array(y).reshape(-1, 1)
```

There are two methods in data reader module. First method listAllFiles takes input as base path and provides output as list of all the files in the base path.
Second method converts multiple files in SVM light format to a single numpy array. Additionally this method also adds bias to the data if required.
The shape of x will be (totalDocuments, totalFeatures) and shape of y will be (totalDocuments, 1).

3.2.2 Preprocessing

```
import numpy as np

from random import shuffle
from Logger.TimeIt import timeit

"""
This module includes all the methods
for data preprocessing
```

```
"""
@timeit
def splitDataSet(data, testSetSize):
	testSetSplit = round(len(data)*testSetSize)
	shuffle(data)
	return data[testSetSplit:], data[:testSetSplit]

@timeit
def distributedMaxScaler(data, communicator = None, root = None):
	maximum = data.max(axis = 0)

	if communicator != None:
		globalMaximum = communicator.gather(maximum, root)
		if communicator.Get_rank() == root:
			globalMaximum = np.vstack(globalMaximum)
			globalMaximum = globalMaximum.max(axis = 0).reshape(1, -1)
		maximum = communicator.bcast(globalMaximum, root)

	scaledData = (data - maximum)
	return scaledData


@timeit
def distributedMinMaxScaler(data, communicator = None, root = None):
	minimum = data.min(axis = 0)
	maximum = data.max(axis = 0)
	if communicator != None:
		globalMinimum = communicator.gather(minimum, root)
		globalMaximum = communicator.gather(maximum, root)
		if communicator.Get_rank() == root:
			globalMaximum = np.vstack(globalMaximum)
			globalMaximum = globalMaximum.max(axis = 0).reshape(1, -1)
			globalMinimum = np.vstack(globalMinimum)
			globalMinimum = globalMinimum.min(axis = 0).reshape(1, -1)
		maximum = communicator.bcast(globalMaximum, root)
		minimum = communicator.bcast(globalMinimum, root)

	scaledData = (data - minimum)/(maximum - minimum)
	return scaledData
```

This module includes methods for data preprocessing. The first method is splitDataSet. This method takes input as a numpy array and provides output as a split in train and test set. Testset size is provided as percentage.
The second method is distributedMaxScaler. This method takes input as data and optionally communicator and root can also be provided if the data scaler is to be executed in distributed environment. If communicator is provided then the maximum data points are calculated within the workers and sent to root. Root will calculate the maximum data point value across the workers and sent it to all workers.
Scaling algorithm : *data - max(data)*
Third method is distributedMinMaxScaler which is also a data scaling method. It works in a similar way as the previous method.
Scaling algorithm : *(data - minimum)/(maximum - minimum)*

3.3 Performance Metric

```
import numpy as np
import math

from Logger.TimeIt import timeit

def SquaredError(yTrue, yPrediction):
	return np.sum((yTrue - yPrediction) ** 2)

def MSE(yTrue, yPrediction):
	return SquaredError(yTrue, yPrediction)/len(yTrue)

def RMSE(yTrue, yPrediction):
	return math.sqrt(MSE(yTrue, yPrediction))

@timeit
def distributedMSE(root, communicator, yTrue, yPrediction):
	localSquaredError = SquaredError(yTrue, yPrediction)
	assimilatedSquaredError = communicator.gather([localSquaredError, len(yTrue)])
```

```
        if communicator.Get_rank() == root:
                globalSquaredSum = 0
                globalCount = 0
                for localSquaredError in assimilatedSquaredError:
                        globalSquaredSum = globalSquaredSum + localSquaredError[0]
                        globalCount = globalCount + localSquaredError[1]
                return (globalSquaredSum/globalCount)


@timeit
def distributedRMSE(root, communicator, yTrue, yPrediction):
        globalMSE = distributedMSE(root, communicator, yTrue, yPrediction)
        if communicator.Get_rank() == root:
                return math.sqrt(globalMSE)
```

This module includes methods to calculate the performance of the gradient descent methods. The first method calculates Squared Error which is sum of square of residuals of actual value and prediction. Second method is MSE which averages the squared error. Third method calculates RMSE which is root of MSE.
Next two methods calculate MSE and RMSE but for distributed environment. All workers calculate local errors and send it to root. Root calculates the global error using the local errors calculated by other workers.

3.4 Distributed Coordinate Descent Algorithm

```
@timeit
def distributedCoordinateDescent(trainSetX, trainSetY,
alpha, beta, maxEpochs, communicator = None, tolerance = 1.0e-10):
        cache = {}
        yPrediction = prediction(trainSetX, beta)
        prevRMSE = 0

        for i in range(0, maxEpochs):
                for coordinate in range(beta.shape[0]):

                        denominator = 0
                        featureVector = trainSetX[:, coordinate].reshape(-1, 1)
                        if coordinate not in cache:
                                denominator =  np.dot(featureVector.T, featureVector)
                                cache[coordinate] = denominator
                        else:
                                denominator = cache[coordinate]


                        numerator = np.dot(featureVector.T,
                      (trainSetY - yPrediction + featureVector * beta[coordinate]))

                        if communicator != None:
                                communicator.barrier()
                                aggregate = communicator.allreduce([numerator, denominator], op = MPI.SUM)
                                numerator = aggregate[0]/communicator.Get_size()
                                denominator = aggregate[1]/communicator.Get_size()

                        betaOld = beta[coordinate][0]

                        beta[coordinate][0] = (1 - alpha) * beta[coordinate] +
                            alpha * (numerator/(denominator + 0.001))

                        yPrediction = yPrediction + featureVector *
                                      (beta[coordinate][0] - betaOld)


                RMSE = Performance.RMSE(trainSetY, yPrediction)
                if i%20 == 0 and communicator != None:
                        """Test convergence among all workers"""
                        localConvergence = False
                        if tolerance > math.fabs(prevRMSE - RMSE):
                                     localConvergence = True
                                     print("Converged locally")
                        globalConvergenceList = communicator.allgather(localConvergence)
                        communicator.barrier()
                        globalConvergence = True
                        for localConvergence in globalConvergenceList:
                                if localConvergence == False:
```

```
                                """Atleast one worker has not converged"""
                                globalConvergence = False
                                break
                        if globalConvergence == True:
                                print("Converged globally")
                                return beta
                elif i%20==0:
                        """Test convergence locally"""
                        if tolerance > math.fabs(prevRMSE - RMSE):
                                print("Converged")
                                return beta

                print(str(i)+ " "+ str(RMSE))
                prevRMSE = RMSE
                print("Epoch completed")
        return beta
```

The above is the implementation for distributed coordinate descent algorithm. Each processor will select one coordinate and minimize along that direction. However as each worker has a subsample of the dataset the final step take the average of all the step calculated by each parallel processors.

Here a step length controller is used. Usual implementation of coordinate descent do not need any step length controller however as the final step is aggregate of all the processors, a step length controller is required. Currently a constant step length is used however it wont be very difficult to perform a line search for step length.

4. Execution

4.1 Parallel execution

```
import sys
import numpy as np
import os
import pandas as pd

from mpi4py import MPI
from Utils import Preprocessing
from PerformanceMetric import Performance
from Regression import CD

"""Command line arguments"""
inputPath = sys.argv[1]
outputPath = sys.argv[2]
maxEpochs = int(sys.argv[3])

"""MPI variables"""
comm = MPI.COMM_WORLD
processorCount = comm.Get_size()
currentRank = comm.Get_rank()
root = 0


"""For storing the assigned task to the processor"""
processorTasks = []
xTrain = None
yTrain = None
xTest = None
yTest = None


if currentRank == root:
        if not os.path.exists(outputPath):
                os.makedirs(outputPath)
        trainData = pd.read_csv(os.path.join(inputPath, "Train", "cup98LRN.txt"))
        """Selected Features"""
        featureList = ['STATE', 'PVASTATE', 'MDMAUD', 'CLUSTER', 'GENDER', 'HIT',
        'DATASRCE', 'MALEMILI', 'MALEMILI', 'VIETVETS', 'WWIIVETS', 'LOCALGOV', 'STATEGOV',
        'FEDGOV','CARDPROM', 'NUMPROM', 'RAMNTALL', 'NGIFTALL', 'CARDGIFT', 'AVGGIFT',
    'TARGET_D']
        trainData = trainData[featureList]
        trainData = pd.get_dummies(trainData)
        processorTasks = np.array_split(trainData, processorCount)
```

```
trainData = comm.scatter(processorTasks, root)

"""Feature cleanup"""
features = np.array(trainData.loc[:, trainData.columns != 'TARGET_D'])
target = np.array(trainData['TARGET_D']).reshape(-1, 1)


"""Split in train and test set"""
totalData = list(range(0, len(trainData)))
trainSetIndices, testSetIndices = Preprocessing.splitDataSet(totalData, 0.3)


"""Min max scaler"""
features = Preprocessing.distributedMinMaxScaler(features, comm, root)

"""Add bias"""
features = np.insert(features, -1, 1, axis = 1)


modelParams = None
if currentRank == root:
       modelParams = np.random.uniform(1, 5, (features.shape[1], 1))


modelParams = comm.bcast(modelParams, root)


startTime = MPI.Wtime()
"""Perform coordinate descent"""
CD.distributedCoordinateDescent(features[trainSetIndices],
 target[trainSetIndices], 0.9, modelParams, maxEpochs, comm)


endTime = MPI.Wtime()
print(endTime - startTime)
```

The root process will read all the data and distribute among all the workers. Each processor will then scale the data using a distributedMinMaxScaler. Furthermore each processor will have a split of train and test data. Root worker will calculate initial model parameters/beta and then share it will all the other workers. Finally each worker will execute distributedCoordinateDescent method.

4.2 Executing the code

For executing the code call the following script.

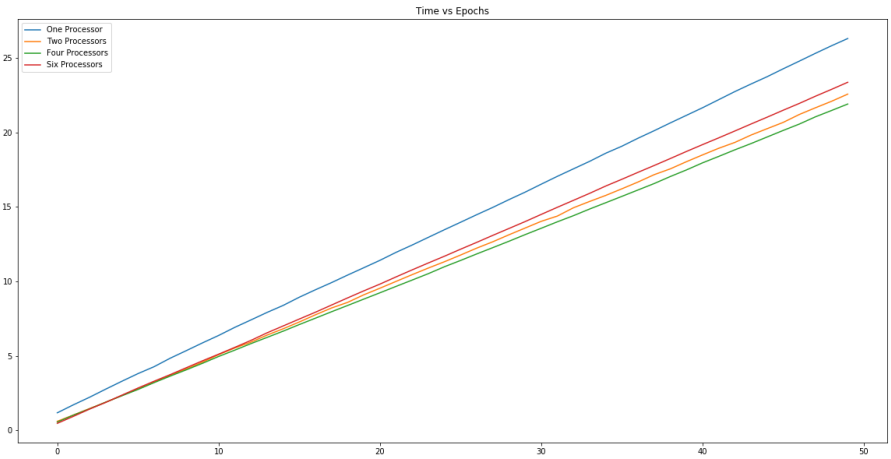
```
foo@bar:~$ sh kddcup.sh {ProcessorCount} {InputPath} {OutputPath} {MaxIterations}
```

The above shell script includes following code

```
if test "$#" -ne 4; then
    echo "Illegal number of parameters"
else
    mpiexec -n $1 python3 kddcup98.py $2 $3 $4
fi
```

5. Performance and Results

5.1 Epoch vs Time

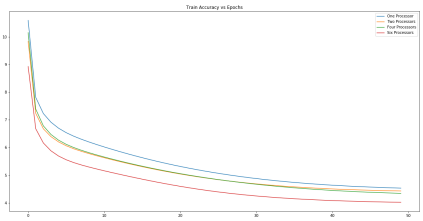
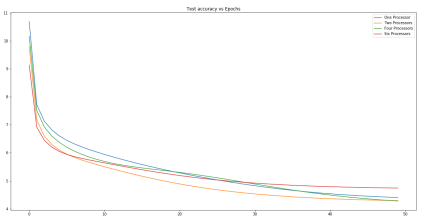


The above graph shows the time taken by each processors to execute the epochs. It can be seen that the time taken for execution is more for single processor and as the processor count increases the performance increases. Running with six processors degrades the performance little, however it is still better than single processor execution.

No of processors	Time taken for one epoch
1	1.18 +/- 0.05
2	0.61 +/- 0.05
4	0.54 +/- 0.08
6	0.48 +/- 0.02

Here one eopch is one pass over all the coordinates.

5.2 RMSE vs epochs

Train Set	Test Set
	

5.2 RMSE vs Time

Train Set	Test Set