HW 2

Xianlong Zhang

Question 0:

The help(PR.readData) will print as following:

```
Help on function readData in module ParallelRegression:

readData(input_file, spark_context)
    Read data from an input file and return rdd containing pairs of the form:
                    (x,y)
    where x is a numpy array and y is a real value. The input file should be a
    'comma separated values' (csv) file: each line of the file should contain x
    followed by y. For example, line:

    1.0,2.1,3.1,4.5

    should be converted to tuple:

    (array(1.0,2.1,3.1),4.5)
(END)
```

The help(PR.f) will print as following:

```
Help on function f in module ParallelRegression:

f(x, y, beta)
    Given vector x containing features, true label y,
    and parameter vector β, return the square error:

                f(β;x,y) =  (y - <x,β>)^2
(END)
```

The two parts are printed because they are in the annotation part under readData() and f() functions.

The Figure 1 is a part of small.test, which is set of numbers, and Figure 2 is an element of resulting RDD, the RDD is a tuple of those elements as format (x, y), x is an array.

```
-1.5460516864,-0.440503936352,0.193226092738,0.0799273949016,-2.04398013779,-0.18226367364
1,0.3315720525,0.136656957145,0.416246480333,-0.189843572572,0.250982384557,0.333345941303
,1.91983395996,-0.865812750011,1.09553347009,-0.331235635898,-0.941938716028,0.97487994833
1,2.2618574226,0.629385712349,-0.693263667343,1.367929047,1.11236515445,-1.85539872736,0.8
55406531137,-0.0580555589141,-1.06576904541,-1.01761117901,0.689704218716,0.162443041459,-
0.284098184552,-1.15820351086,0.447166978403,-0.994884295296,0.425119792687,-0.41264188668
6,0.296842048713,0.465319023514,0.308523916837,0.440023122441,-0.162562787075,-0.296333392
969,1.17969164605,-0.78981949926,-0.697921885789,-1.29640209363,0.323771712368,-1.04674736
218,0.811261370827,0.2516793425,27.2647885495
```

Figure 1

```
(array([-1.54605169, -0.44050394,  0.19322609,  0.07992739, -2.04398014,
       -0.18226367,  0.33157205,  0.13665696,  0.41624648, -0.18984357,
        0.25098238,  0.33334594,  1.91983396, -0.86581275,  1.09553347,
       -0.33123564, -0.94193872,  0.97487995,  2.26185742,  0.62938571,
       -0.69326367,  1.36792905,  1.11236515, -1.85539873,  0.85540653,
       -0.05805556, -1.06576905, -1.01761118,  0.68970422,  0.16244304,
       -0.28409818, -1.15820351,  0.44716698, -0.9948843 ,  0.42511979,
       -0.41264189,  0.29684205,  0.46531902,  0.30852392,  0.44002312,
       -0.16256279, -0.29633339,  1.17969165, -0.7898195 , -0.69792189,
       -1.29640209,  0.32377171, -1.04674736,  0.81126137,  0.25167934]), 27.2647885495)
```

Figure 2

Question 1:

(a)

The modified code is as following:

```
def predict(x,beta):
    """ Given vector x containing features and parameter vector β,
        return the predicted value:

                        y = <x,β>

    """
    return np.dot(x, beta)
```

(b)

```
>>> import numpy as np
>>> import ParallelRegression as PR
>>> x = np.array([np.cos(t) for t in range(5)])
>>> beta = np.array([np.sin(t) for t in range(5)])
>>> PR.predict(x, beta)
0.43121883997110472
```

Question 2:

(a)

$$f(\beta; x, y) = (y - \beta^T x)^2 = (y - \sum \beta_i x_i)^2$$

$$\nabla f(\beta; x, y) = \frac{\partial (y - \sum \beta_i x_i)^2}{\partial \beta} = 2 * (y - \sum \beta_i x_i) * (\frac{\partial (-\sum \beta_i x_i)}{\partial \beta}) = -2 * (y - \beta^T x) * x$$

(b)

```
def f(x,y,beta):
    """ Given vector x containing features, true label y,
        and parameter vector β, return the square error:

                f(β;x,y) =  (y - <x,β>)^2

    """
    return np.square(y - np.dot(x, beta))
```

(c)

```
def localGradient(x,y,beta):
    """ Given vector x containing features, true label y,
        and parameter vector β, return the gradient ∇f of f:

                ∇f(β;x,y) =  -2 * (y - <x,β>) * x

        with respect to parameter vector β.

        The return value is  ∇f.
    """
    return -2.0 * (y - np.dot(x, beta)) * x
```

(d)

```
>>> import ParallelRegression as PR
>>> import numpy as np
>>> x = np.array([np.cos(t) for t in range(5)])
>>> beta =  np.array([np.sin(t) for t in range(5)])
>>> estimatedRes = PR.estimateGrad(lambda beta: PR.f(x, y, beta), beta, 0.000001)
>>> calRes = PR.localGradient(x, y, beta)
>>> estimatedRes
array([-1.13756132, -0.61462725,  0.47339313,  1.12617914,  0.74356078])
>>> calRes
array([-1.13756232, -0.61462754,  0.47339296,  1.12617816,  0.74356035])
```

We set $\delta = 0.000001$ small enough, and from results above, we find they are nearly identical, so it's correct since localGradient agrees with the estimate produced by estimateGrad.

Question 3:

(a)

```
def F(data,beta,lam = 0):
    """  Compute the regularized mean square error:

             F(β) = 1/n Σ_{(x,y) in data}    f(β;x,y)  + λ ||β ||_2^2
                  = 1/n Σ_{(x,y) in data} (y- <x,β>)^2 + λ ||β ||_2^2

         where n is the number of (x,y) pairs in RDD data.

         Inputs are:
            - data: an RDD containing pairs of the form (x,y)
            - beta: vector β
            - lam:  the regularization parameter λ


         The return value is F(β).

    """
    sum, count = data.map(lambda (x, y) : (f(x, y, beta), 1))\
                     .reduce(lambda x, y : (x[0] + y[0], x[1] + y[1]))
    return (1. / count) * sum + lam * np.sum(np.square(beta))
```

(b)

```
def gradient(data,beta,lam = 0):
    """ Compute the gradient  ∇F of the regularized mean square error
                F(β) = 1/n Σ_{(x,y) in data} f(β;x,y) + λ ||β ||_2^2
                     = 1/n Σ_{(x,y) in data} (y- <x,β>)^2 + λ ||β ||_2^2

        where n is the number of (x,y) pairs in data.

        Inputs are:
             - data: an RDD containing pairs of the form (x,y)
             - beta: vector β
             - lam:  the regularization parameter λ


        The return value is an array containing ∇F.
    """
    gradSum, count = data.map(lambda (x, y) : (localGradient(x, y, beta), 1))\
                    .reduce(lambda x, y: (x[0] + y[0], x[1] + y[1]))
    return (1. / count) * gradSum + lam * 2 * beta
```

(c)

```
>>> import ParallelRegression as PR
>>> import numpy as np
>>> lam = 1.0
>>> beta =  np.array([np.sin(t) for t in range(50)])
>>> data = PR.readData("data/small.test", sc)

>>> grad = PR.gradient(data, beta, lam)

>>> grad
array([ 4.39945352,  2.86050932,  4.50982604, -1.85170994, -6.19540742,
       -3.84463142, -2.80309467,  3.58914049,  5.23949536,  3.20436283,
       -1.13257134, -4.00822426, -3.97652387,  1.62077657,  8.53774366,
        3.76708615, -3.56305719, -6.7752212 , -7.21234235,  2.42757438,
        2.0236413 ,  5.2434361 , -1.23645832, -1.93962279, -4.89414425,
        0.33872292,  2.67334964,  5.74150933,  3.16590298, -0.2020467 ,
       -1.27129553, -1.11336197,  2.52013787,  1.58421622, -0.24287307,
       -2.71253267, -1.66387657, -1.93504448,  6.16984068,  2.62113744,
        5.2980497 ,  2.08366788, -6.20624771,  0.16714202, -0.13615823,
        2.54079899, -0.05016347,  1.36447249, -5.48897702, -1.47100687])



>>> esitmatedGrad = PR.estimateGrad(lambda beta: PR.F(data, beta, lam), beta, 0.0000001)

>>> estimatedGrad
array([ 4.39945325,  2.8605092 ,  4.50982498, -1.85171075, -6.19540742,
       -3.84463135, -2.8030945 ,  3.58914178,  5.23949552,  3.204363  ,
       -1.13257101, -4.00822387, -3.976524  ,  1.62077697,  8.53774281,
        3.76708499, -3.56305804, -6.77522166, -7.21234244,  2.42757437,
        2.02364106,  5.24343591, -1.23645805, -1.93962251, -4.89414447,
        0.33872311,  2.67335054,  5.74150818,  3.1659016 , -0.20204823,
       -1.27129567, -1.11336249,  2.52013706,  1.58421642, -0.24287431,
       -2.71253271, -1.66387736, -1.9350432 ,  6.16984096,  2.62113701,
        5.29804993,  2.08366771, -6.20624803,  0.16714296, -0.13615818,
        2.5407985 , -0.05016318,  1.36447284, -5.48897674, -1.47100707])
```

Question 4:

(a)

The code I modified is as following:

```
def test(data,beta):
    """ Compute the mean square error

                MSE(β) =  1/n Σ_{(x,y) in data} (y- <x,β>)^2

        of parameter vector β over the dataset contained in RDD data, where n is the size of RDD data.

        Inputs are:
             - data: an RDD containing pairs of the form (x,y)
             - beta: vector β

        The return value is MSE(β).

    """
    return F(data, beta)
```

(b)

The train() function I modified is as following:

```
    startT = time()
    beta = beta_0 * 1.0
    for k in range(max_iter):
        F_val = F(data, beta, lam)
        F_grad = gradient(data, beta, lam)
        F_norm = np.linalg.norm(F_grad)
        if F_norm < eps:
            print "Present iteration: ", k + 1, ", time used: ", (endT - startT),\
            ", present value: ", F_val, ", norm of regularized MSE: ", F_norm
            return beta, F_norm, k + 1
        endT = time()
        print "Present iteration:", k + 1, ", time used:", (endT - startT),\
        ", present value:", F_val, ", norm of regularized MSE:", F_norm
        gamma = lineSearch(lambda x : F(data, x, lam), beta, F_grad)
        beta = beta - gamma * F_grad * 1.0
    return beta, F_norm, k + 1
```

(c)

The following is the result we get from execution:

```
Reading training data from data/small.train
Training on data from data/small.train with λ = 0.0 , ε = 0.01 , max iter =  100
Present iteration: 1 , time used: 0.149458885193 , present value: 220.564648376 , norm of regularized MSE: 10.7326121454
Present iteration: 2 , time used: 0.532584905624 , present value: 196.950858491 , norm of regularized MSE: 5.21787375028
Present iteration: 3 , time used: 0.994854927063 , present value: 192.202962818 , norm of regularized MSE: 0.981922719761
Present iteration: 4 , time used: 1.34191083908 , present value: 192.030079227 , norm of regularized MSE: 0.695480877873
Present iteration: 5 , time used: 1.72033691406 , present value: 191.9525186 , norm of regularized MSE: 0.168629000552
Present iteration: 6 , time used: 1.99393677711 , present value: 191.947034509 , norm of regularized MSE: 0.126634425216
Present iteration: 7 , time used: 2.35749983788 , present value: 191.944461359 , norm of regularized MSE: 0.0342334207718
Present iteration: 8 , time used: 2.63192486763 , present value: 191.944201927 , norm of regularized MSE: 0.0250179753358
Present iteration:  9 , time used:  2.63192486763 , present value:  191.944098559 , norm of regularized MSE:  0.00750286425664
Algorithm ran for 9 iterations. Converged: True
Saving trained β in beta_small_0.0
Reading test data from data/small.test
Reading beta from beta_small_0.0
Computing MSE on data data/small.test
MSE is: 255.121147746
```

The part in the following screenshot has the effect of "--silent". We can see that there are two modes –silent and –verbose. If we choose –silent the sc.setLogLevel will be ERROR, and some trival information won't appear.

```
verbosity_group.add_argument('--verbose', dest='verbose', action='store_true')
verbosity_group.add_argument('--silent', dest='verbose', action='store_false')
parser.set_defaults(verbose=True)

args = parser.parse_args()

sc = SparkContext(appName='Parallel Ridge Regression')

if not args.verbose :
    sc.setLogLevel("ERROR")
```

Question 5:

The table of the resulting MSE values are as following, and from the table we can find that when $\lambda = 8.0$, the smallest MSE can be got, which is 3967.17658354.

$\lambda = 0.0$	$\lambda = 1.0$	$\lambda = 2.0$	$\lambda = 3.0$	$\lambda = 4.0$	$\lambda = 5.0$	$\lambda = 6.0$
4151.49371162	4000.09930202	3977.8384858	3971.3275161	3968.86820762	3967.83265427	3967.38615943
$\lambda = 7.0$	$\lambda = 8.0$	$\lambda = 9.0$	$\lambda = 10.0$	$\lambda = 11.0$	$\lambda = 12.0$	$\lambda = 13.0$
3967.21396808	3967.17658354	3967.2071355	3967.27230693	3967.35461275	3967.44420838	3967.53515038
$\lambda = 14.0$	$\lambda = 15.0$	$\lambda = 16.0$	$\lambda = 17.0$	$\lambda = 18.0$	$\lambda = 19.0$	$\lambda = 20.0$
3967.6248401	3967.7117152	3967.79442828	3967.87275232	3967.94720755	3968.0177581	3968.08323632

Question 6:

The average running time for different partitions are as following:

N=1	N=2	N=4	N=8	N=16	N=32	N=64
61.845	38.330	24.210	16.133	13.464	15.399	18.821

From table above, when N = 16, the smallest mean convergence time can be got, which is 13.464, so we can conclude that adding more partitions not always decrease the convergence time for the reason that every partition will be assigned a task, if there are too many tasks, the data size is too small in each partition, or sometimes workload is not well balanced, which would cause the cost of switching thread and make some tasks wait. It's a very inefficient use of resources.