EECE 5698 Assignment3: Parallel Classification

Zifeng Wang

Problem 1

(1a) `SparseVector` is a subclass of standard python dictionary, which means it shares the same methods with the standard dictionary. Also, it implements new methods so that it extends the functionalities of standard dictionary.

(1b) Methods it shares with standard dictionary : has `key(), items(), keys()`, etc. Methods it has that are not present in a dictionary: `safeAccess(key), dot(other), __add__(other), __sub__(other), __mul__(s), __rmul__(s)`

(1c) The operations and outcomes are as below

```
In [1]: from SparseVector import SparseVector as SV

In [2]: sv1 = SV([(1,3), (5,8)])

In [3]: sv2 = SV([(1,10), (3,6)])

In [4]: sv1.dot(sv2)
Out[4]: 30

In [5]: sv2.dot(sv1)
Out[5]: 30

In [6]: 5 * sv1
Out[6]: {1: 15, 5: 40}

In [7]: sv1 * 5
Out[7]: {1: 15, 5: 40}

In [8]: 5 * sv2
Out[8]: {1: 50, 3: 30}

In [9]: sv2 * 5
Out[9]: {1: 50, 3: 30}
```

Problem 2 Here we prove 2(b) first

2(b)

For simplicity, prove 2(b) first

$$\begin{aligned}\nabla \ell(\beta; x, y) &= \frac{1}{1+e^{-y\beta^T x}} e^{-y\beta^T x}(-yx) \\ &= -\frac{e^{-y\beta^T x} e^{y\beta^T x}}{(1+e^{-y\beta^T x})e^{y\beta^T x}}(yx) \\ &= -\frac{yx}{1+e^{y\beta^T x}}\end{aligned}$$

2(a)

Then we prove 2(a), we could write the gradient in the form of components of β and x, where $\beta = [\beta_1, ..., \beta_d]^T, x = [x_1, ..., x_d]^T$. Let $i \in 1, ..., d$.

$$\frac{\partial \ell(\beta_i; x, y)}{\partial \beta_i} = -\frac{yx_i}{(1 + e^{y\beta^T x})^2}$$

So we have

$$\begin{aligned}\frac{\partial^2 \ell(\beta_i; x, y)}{\partial \beta_i \partial \beta_j} &= -\frac{yx_i}{(1 + e^{y\beta^T x})^2}(-e^{y\beta^T x})yx_j \\ &= \frac{y^2 e^{y\beta^T x}}{(1 + e^{y\beta^T x})^2} x_i x_j \\ &= k x_i x_j\end{aligned}$$

Where $k = \frac{y^2 e^{y\beta^T x}}{(1+e^{y\beta^T x})^2} > 0$ does not depend on i, j
So we have

$$\nabla^2 \ell(\beta; x, y) = kxx^T$$

Which is positive semi definite, because for $m \in \mathbb{R}^d$

$$m^T(kxx^T)m = k(mx^T)^2 \geq 0$$

So we have the loss is convex. From the definition of convex function

2(c)

From 2(a) we have

$$\frac{\partial \ell(\beta_j; x, y)}{\partial \beta_j}|_{x_j=0} = -\frac{0y}{(1 + e^{y\beta^T x})^2} = 0$$

Which means $\nabla \ell(\beta; x, y)$ does not rely on β_j given $x_j = 0$
Similarly we have

$$\begin{aligned}\ell(\beta; x, y) &= (1 + e^{-y\beta^T x}) \\ &= (1 + e^{-y\sum_{i=1}^d x_i \beta_i}) \\ &= (1 + e^{-y\sum_{i=1, i\neq j}^d x_i \beta_i} e^{-yx_j\beta_j})\end{aligned}$$

When $x_j = 0$, $e^{-yx_j\beta_j} = 1$, $\ell(\beta; x, y)$ does not rely on β_j as well.

2(d)

From the conclusions of previous problems, we get

$$\sum_{i=1}^{n} \ell(\beta; x_i, y_j)$$

is convex as it is just sum of convex functions $\ell(\beta; x_i, y_j)$.
Also, we have $\lambda \geq 0$ and we know all norms are convex, we have

$$\nabla^2(\lambda||\beta||_2^2) = 2\lambda I$$

So $\lambda||\beta||_2^2$ is convex as well. As a result we have

$$L(\beta) = \sum_{i=1}^{n} \ell(\beta; x_i, y_j) + \lambda||\beta||_2^2$$

is convex.

Problem 3

3(a) Here below are the function definitions of `logisticLoss`, `gradLogisticLoss` and `gradTotalLoss`

```
def logisticLoss(beta,x,y):
    """
        Given sparse vector beta, a sparse vector x, and a binary value y in {-1,+1}, compute the logistic loss

                l(β;x,y) = log( 1.0 + exp(-y * <β,x>) )

    The input is:
        - beta: a sparse vector β
        - x: a sparse vector x
            - y: a binary value in {-1,+1}

    """
    return np.log(1.0 + np.exp(-y * x.dot(beta)))
```

```
def gradLogisticLoss(beta,x,y):
    """
        Given a sparse vector beta, a sparse vector x, and
        a binary value y in {-1,+1}, compute the gradient of the logistic loss

            ∇l(B;x,y) = -y / (1.0 + exp(y <β,x> )) * x

    The input is:
        - beta: a sparse vector β
        - x: a sparse vector x
            - y: a binary value in {-1,+1}


    """
    return -y / (1.0 + np.exp(y * x.dot(beta))) * x
```

```
def gradTotalLoss(data,beta, lam = 0.0):
    """  Given a sparse vector beta and a dataset perform compute the gradient of regularized total logistic loss :

            ∇L(β) = Σ_{(x,y) in data}  ∇l(β;x,y)  + 2λ β

        Inputs are:
            - data: a python list containing pairs of the form (x,y), where x is a sparse vector and y is a binary value
            - beta: a sparse vector β
            - lam: the regularization parameter λ
    """
    gradLoss = SparseVector({})
    for (x, y) in data:
        gradLoss += gradLogisticLoss(beta, x, y)
    return gradLoss + 2.0*lam*beta
```

3(b) Here below is the final function definition of `test`

```
def test(data,beta):
    """ Output the quantities necessary to compute the accuracy, precision, and recall of the prediction of labels in a dataset under a given β.

        The accuracy (ACC), precision (PRE), and recall (REC) are defined in terms of the following sets:

                P = datapoints (x,y) in data for which <β,x> > 0
                N = datapoints (x,y) in data for which <β,x> <= 0

                TP = datapoints in (x,y) in P for which y=+1
                FP = datapoints in (x,y) in P for which y=-1
                TN = datapoints in (x,y) in N for which y=-1
                FN = datapoints in (x,y) in N for which y=+1

        For #XXX the number of elements in set XXX, the accuracy, precision, and recall of parameter vector β over data are defined as:

                ACC(β,data) = ( #TP+#TN ) / (#P + #N)
                PRE(β,data) = #TP / (#TP + #FP)
                REC(β,data) = #TP/ (#TP + #FN)

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

        The return values are
            - ACC, PRE, REC

    """
    P = 0
    N = 0
    TP = 0
    FP = 0
    TN = 0
    FN = 0
    for x, y in data:
        temp = x.dot(beta)
        if temp > 0:
            P += 1
            if y == 1:
                TP += 1
            elif y == -1:
                FP += 1
        else:
            N += 1
            if y == -1:
                TN += 1
            elif y == 1:
                FN += 1

    ACC = (TP + TN) / float(P + N)
    PRE = TP / float(TP + FP)
    REC = TP / float(TP + FN)
    return ACC, PRE, REC
```

3(c)

We choose λ from {0.0, 1.0, 10.0}

λ = 0.0

```
[wang.zife@c0013 Assignment3]$ python LogisticRegression.py mushrooms/mushrooms.train --testdata mushrooms/mushrooms.test --lam 0.0 --max_iter 20
Reading training data from mushrooms/mushrooms.train
Read 7416 data points with 117 features in total
Reading test data from mushrooms/mushrooms.test
Read 1000 data points with 115 features
Training on data from mushrooms/mushrooms.train with λ = 0.0 , ε = 0.1 , max iter =  20
k =  0  t =  3.35645294189      L(β_k) =  5140.379491033203     ||∇L(β_k)||_2 =  4273.548233026041      gamma =  0.000470184984576      ACC =  0.909    PRE =  0.855325914149   REC =  1.0
k =  1  t =  7.15586996078      L(β_k) =  2516.9939944922876    ||∇L(β_k)||_2 =  3275.986464902974      gamma =  0.000169266594447      ACC =  0.92     PRE =  0.892123287671   REC =  0.968401486989
k =  2  t =  10.0537118912      L(β_k) =  1536.6296089151033    ||∇L(β_k)||_2 =  802.3372840311758      gamma =  0.00362797056  ACC =  0.979    PRE =  0.965765765766   REC =  0.996282527881
k =  3  t =  13.5136380196      L(β_k) =  772.3391480787859     ||∇L(β_k)||_2 =  980.5492791241032      gamma =  0.000470184984576      ACC =  0.974    PRE =  0.981203007519   REC =  0.970260223048
k =  4  t =  16.9038770199      L(β_k) =  648.9689952939555     ||∇L(β_k)||_2 =  636.7301783321042      gamma =  0.000470184984576      ACC =  0.976    PRE =  0.972426470588   REC =  0.983271375465
k =  5  t =  20.1671419144      L(β_k) =  583.1293003296519     ||∇L(β_k)||_2 =  390.12150441871074     gamma =  0.00078364164096       ACC =  0.975    PRE =  0.981238273921   REC =  0.972118959108
k =  6  t =  23.5557079315      L(β_k) =  554.8636871007043     ||∇L(β_k)||_2 =  474.3895736703113      gamma =  0.000470184984576      ACC =  0.981    PRE =  0.977300552486   REC =  0.986988847584
k =  7  t =  26.8180449089      L(β_k) =  511.6589355365914     ||∇L(β_k)||_2 =  258.88738725459575     gamma =  0.00078364164096       ACC =  0.98     PRE =  0.981412639405   REC =  0.981412639405
k =  8  t =  30.066906929       L(β_k) =  486.9422819846264     ||∇L(β_k)||_2 =  276.7354829899021      gamma =  0.00078364164096       ACC =  0.984    PRE =  0.976277372263   REC =  0.994423791822
k =  9  t =  33.3298799992      L(β_k) =  467.33336504481593    ||∇L(β_k)||_2 =  297.33568125873046     gamma =  0.00078364164096       ACC =  0.979    PRE =  0.981378026071   REC =  0.979553903346
k =  10         t =  36.7253370285      L(β_k) =  449.7683227925332     ||∇L(β_k)||_2 =  319.5307352237093      gamma =  0.000470184984576      ACC =  0.984    PRE =  0.979779411765   REC =  0.990706319703
k =  11         t =  39.7053449154      L(β_k) =  425.92454429676246    ||∇L(β_k)||_2 =  166.72120140928928     gamma =  0.002176782336         ACC =  0.983    PRE =  0.988742964353   REC =  0.979553903346
k =  12         t =  43.0983970165      L(β_k) =  405.9943035852676     ||∇L(β_k)||_2 =  388.19925909616404     gamma =  0.000470184984576      ACC =  0.986    PRE =  0.979853479853   REC =  0.994423791822
k =  13         t =  46.0599400997      L(β_k) =  374.2509647417060?    ||∇L(β_k)||_2 =  147.16737512248403     gamma =  0.002176782336         ACC =  0.987    PRE =  0.994350282486   REC =  0.981412639405
k =  14         t =  49.4615418911      L(β_k) =  364.10371130630483    ||∇L(β_k)||_2 =  357.7097348748522      gamma =  0.000470184984576      ACC =  0.989    PRE =  0.985267034991   REC =  0.994423791822
k =  15         t =  52.2741019726      L(β_k) =  335.53335812252476    ||∇L(β_k)||_2 =  119.37562057442935     gamma =  0.00362797056  ACC =  0.989    PRE =  0.994371482176   REC =  0.985130111524
k =  16         t =  55.691030107       L(β_k) =  321.40210325045234    ||∇L(β_k)||_2 =  359.39323294505124     gamma =  0.000470184984576      ACC =  0.994    PRE =  0.992592592593   REC =  0.996282527881
k =  17         t =  58.0605199337      L(β_k) =  290.83776835918057    ||∇L(β_k)||_2 =  93.27830340?238        gamma =  0.01679616     ACC =  0.99     PRE =  0.994382022472   REC =  0.986988847584
k =  18         t =  61.3027141094      L(β_k) =  238.23480047134834    ||∇L(β_k)||_2 =  427.83733739578?6      gamma =  0.00078364164096       ACC =  0.997    PRE =  0.994454713494   REC =  1.0
k =  19         t =  64.4026780128      L(β_k) =  183.51233512776878    ||∇L(β_k)||_2 =  104.68994981492128     gamma =  0.001306069401?        ACC =  0.997    PRE =  0.994454713494   REC =  1.0
Algorithm ran for 20 iterations. Converged: False
Saving trained β in beta
```

λ = 1.0

```
[wang.zife@c0013 Assignment3]$ python LogisticRegression.py mushrooms/mushrooms.train --testdata mushrooms/mushrooms.test --lam 1.0 --max_iter 20
Reading training data from mushrooms/mushrooms.train
Read 7416 data points with 117 features in total
Reading test data from mushrooms/mushrooms.test
Read 1000 data points with 115 features
Training on data from mushrooms/mushrooms.train with λ = 1.0 , ε = 0.1 , max iter =  20
k =  0  t =  3.21237111092      L(β_k) =  5140.379491033203     ||∇L(β_k)||_2 =  4273.548233026041      gamma =  0.0004701849845 76      ACC =  0.909    PRE =  0.855325914149   REC =  1.0
k =  1  t =  6.8344906081       L(β_k) =  2521.031514908503     ||∇L(β_k)||_2 =  3276.344977453154      gamma =  0.0001692665944 47      ACC =  0.92     PRE =  0.892123287671   REC =  0.968401486989
k =  2  t =  9.58678293228      L(β_k) =  1541.034088057605     ||∇L(β_k)||_2 =  800.6911754008619      gamma =  0.0036279705 6  ACC =  0.979    PRE =  0.965765765766   REC =  0.996282527881
k =  3  t =  12.9323890209      L(β_k) =  800.6863496497471     ||∇L(β_k)||_2 =  1011.3003853022815     gamma =  0.0004701849845 76      ACC =  0.974    PRE =  0.981203007519   REC =  0.970260223048
k =  4  t =  16.258713007       L(β_k) =  674.8844560302292     ||∇L(β_k)||_2 =  665.9000981264079      gamma =  0.0004701849845 76      ACC =  0.975    PRE =  0.970642201835   REC =  0.983271375465
k =  5  t =  19.5909090042      L(β_k) =  606.7609904402055     ||∇L(β_k)||_2 =  407.56767081051424     gamma =  0.0004701849845 76      ACC =  0.98     PRE =  0.981412639405   REC =  0.981412639405
k =  6  t =  22.6482651234      L(β_k) =  566.9918768399507     ||∇L(β_k)||_2 =  258.89658750961775     gamma =  0.0013060694016         ACC =  0.984    PRE =  0.976277372263   REC =  0.994423791822
k =  7  t =  25.9726719856      L(β_k) =  537.8262514670596     ||∇L(β_k)||_2 =  432.03958572734217     gamma =  0.0004701849845 76      ACC =  0.98     PRE =  0.981412639405   REC =  0.981412639405
k =  8  t =  29.1943149567      L(β_k) =  498.52726352261834    ||∇L(β_k)||_2 =  206.87313843174843     gamma =  0.0013060694016         ACC =  0.984    PRE =  0.976277372263   REC =  0.994423791822
k =  9  t =  32.6094009876      L(β_k) =  480.7719639495731     ||∇L(β_k)||_2 =  347.81000518970666     gamma =  0.0004701849845 76      ACC =  0.982    PRE =  0.979704797048   REC =  0.986988847584
k =  10         t =  35.5933561325      L(β_k) =  453.1679500861842     ||∇L(β_k)||_2 =  162.82177927675863     gamma =  0.002176782336          ACC =  0.985    PRE =  0.976320582878   REC =  0.996282527881
k =  11         t =  39.0218451023      L(β_k) =  436.8865661464825     ||∇L(β_k)||_2 =  379.67929538500186     gamma =  0.0004701849845 76      ACC =  0.985    PRE =  0.981583793738   REC =  0.990706319703
k =  12         t =  41.9889609814      L(β_k) =  404.89582463799957    ||∇L(β_k)||_2 =  135.56679984387148     gamma =  0.002176782336          ACC =  0.985    PRE =  0.976320582878   REC =  0.996282527881
k =  13         t =  45.4102230072      L(β_k) =  390.9090013252125     ||∇L(β_k)||_2 =  295.1965329813116      gamma =  0.0004701849845 76      ACC =  0.99     PRE =  0.987084870849   REC =  0.994423791822
k =  14         t =  48.0886138333      L(β_k) =  369.3091644540126     ||∇L(β_k)||_2 =  108.30970241611934     gamma =  0.0060466176    ACC =  0.987    PRE =  0.976406533575   REC =  1.0
k =  15         t =  51.3794751167      L(β_k) =  348.4854989971825 6   ||∇L(β_k)||_2 =  390.5459580980805      gamma =  0.0007836416409 6       ACC =  0.992    PRE =  0.994402985075   REC =  0.990706319703
k =  16         t =  54.652310133       L(β_k) =  322.42985665689054    ||∇L(β_k)||_2 =  256.7968251280171      gamma =  0.0007836416409 6       ACC =  0.994    PRE =  0.990774907749   REC =  0.998141263941
k =  17         t =  57.9140601158      L(β_k) =  309.5841671934596     ||∇L(β_k)||_2 =  170.16428263905 41     gamma =  0.0007836416409 6       ACC =  0.995    PRE =  0.994434137291   REC =  0.996282527881
k =  18         t =  60.9546010494      L(β_k) =  301.3780633555753 6   ||∇L(β_k)||_2 =  114.12659985625002     gamma =  0.0013060694016         ACC =  0.995    PRE =  0.992606284658   REC =  0.998141263941
k =  19         t =  64.1429350376      L(β_k) =  297.65146407090236    ||∇L(β_k)||_2 =  154.96136506980739     gamma =  0.0007836416409 6       ACC =  0.995    PRE =  0.994434137291   REC =  0.996282527881
Algorithm ran for 20 iterations. Converged: False
Saving trained β in beta
```

λ = 10.0

```
[wang.zife@c0013 Assignment3]$ python LogisticRegression.py mushrooms/mushrooms.train --testdata mushrooms/mushrooms.test --lam 10.0 --max_iter 20
Reading training data from mushrooms/mushrooms.train
Read 7416 data points with 117 features in total
Reading test data from mushrooms/mushrooms.test
Read 1000 data points with 115 features
Training on data from mushrooms/mushrooms.train with λ = 10.0 , ε = 0.1 , max iter =  20
k =  0  t =  3.51568102837      L(β_k) =  5140.379491033203     ||∇L(β_k)||_2 =  4273.548233026041      gamma =  0.0004701849845 76      ACC =  0.909    PRE =  0.855325914149   REC =  1.0
k =  1  t =  7.39733791351      L(β_k) =  2557.3691986544427    ||∇L(β_k)||_2 =  3279.791419606979      gamma =  0.0001692665945 47      ACC =  0.921    PRE =  0.893653516295   REC =  0.968401486989
k =  2  t =  10.281635046       L(β_k) =  1580.4670069319575    ||∇L(β_k)||_2 =  786.7239775788153      gamma =  0.0036279705 6  ACC =  0.98     PRE =  0.965827338129   REC =  0.998141263941
k =  3  t =  13.9193339348      L(β_k) =  1064.089362549908     ||∇L(β_k)||_2 =  1329.3162391382662     gamma =  0.0002821109907 46      ACC =  0.976    PRE =  0.975925925926   REC =  0.979553903346
k =  4  t =  16.9486320019      L(β_k) =  826.6289929358925     ||∇L(β_k)||_2 =  322.7135610904411      gamma =  0.002176782336          ACC =  0.984    PRE =  0.976277372263   REC =  0.994423791822
k =  5  t =  20.4222068787      L(β_k) =  769.9113003889959     ||∇L(β_k)||_2 =  649.8966106405724      gamma =  0.0004701849845 76      ACC =  0.977    PRE =  0.981308411215   REC =  0.975836431227
k =  6  t =  23.901211977       L(β_k) =  704.5728340244931     ||∇L(β_k)||_2 =  295.29440558022776     gamma =  0.0004701849845 76      ACC =  0.982    PRE =  0.977941176471   REC =  0.988847583643
k =  7  t =  27.2272720337      L(β_k) =  687.5547576211729     ||∇L(β_k)||_2 =  171.56821254255854     gamma =  0.0007836416409 6       ACC =  0.981    PRE =  0.981447124304   REC =  0.983271375465
k =  8  t =  30.7138559818      L(β_k) =  677.5535789329613     ||∇L(β_k)||_2 =  198.237479476743       gamma =  0.0004701849845 76      ACC =  0.984    PRE =  0.979779411765   REC =  0.990706319703
k =  9  t =  33.893269062       L(β_k) =  668.0195579234075     ||∇L(β_k)||_2 =  130.72215992152115     gamma =  0.0013060694016         ACC =  0.983    PRE =  0.981515711645   REC =  0.986988847584
k =  10         t =  37.3623290062      L(β_k) =  659.5846528048683     ||∇L(β_k)||_2 =  232.29008733610144     gamma =  0.0004701849845 76      ACC =  0.986    PRE =  0.979853479853   REC =  0.994423791822
k =  11         t =  40.7058670521      L(β_k) =  649.5221675986354     ||∇L(β_k)||_2 =  130.5360271966 05      gamma =  0.0007836416409 6       ACC =  0.986    PRE =  0.983394833948   REC =  0.990706319703
k =  12         t =  44.1735640109      L(β_k) =  644.2783854745019     ||∇L(β_k)||_2 =  152.9467941417904      gamma =  0.0004701849845 76      ACC =  0.986    PRE =  0.979853479853   REC =  0.994423791822
k =  13         t =  47.3565020561      L(β_k) =  638.7149242589148     ||∇L(β_k)||_2 =  98.41017856559207      gamma =  0.0013060694016         ACC =  0.988    PRE =  0.987037037037   REC =  0.990706319703
k =  14         t =  50.8276350498      L(β_k) =  633.8041949411281     ||∇L(β_k)||_2 =  173.00810185909506     gamma =  0.0004701849845 76      ACC =  0.986    PRE =  0.979853479853   REC =  0.994423791822
k =  15         t =  54.1522438526      L(β_k) =  627.9060605562742     ||∇L(β_k)||_2 =  95.96541625553058      gamma =  0.0007836416409 6       ACC =  0.99     PRE =  0.987084870849   REC =  0.994423791822
k =  16         t =  57.4704430103      L(β_k) =  624.4949703130785     ||∇L(β_k)||_2 =  108.11140609196644     gamma =  0.0007836416409 6       ACC =  0.988    PRE =  0.981684981685   REC =  0.996282527881
k =  17         t =  60.9622499943      L(β_k) =  621.8652531701674     ||∇L(β_k)||_2 =  127.02344628818398     gamma =  0.0004701849845 76      ACC =  0.99     PRE =  0.987084870849   REC =  0.994423791822
k =  18         t =  64.1343040466      L(β_k) =  618.1779992597395     ||∇L(β_k)||_2 =  74.27450908799452      gamma =  0.0013060694016         ACC =  0.989    PRE =  0.983486238532   REC =  0.996282527881
k =  19         t =  67.6090679169      L(β_k) =  615.2920582868961     ||∇L(β_k)||_2 =  126.55220823233711     gamma =  0.0004701849845 76      ACC =  0.991    PRE =  0.988909426987   REC =  0.994423791822
Algorithm ran for 20 iterations. Converged: False
Saving trained β in beta
```

3(d)

We definitely want a model with (a) high precision and low recall, so that even if we could not find out all the edible mushrooms, we would minimize the possibility to misclassify poisonous mushroom as edible, which is extremely dangerous.

Problem 4

4(a)

```
def getAllFeaturesRDD(dataRDD):
    """ Get all the features present in grouped dataset dataRDD.

    The input is:
            - dataRDD containing pairs of the form (SparseVector(x),y).

        The return value is an list containing the union of all unique features present in sparse vectors inside dataRDD.
    """
    all_features = dataRDD.map(lambda (x,y): x)\
                          .reduce(add)

    return all_features.keys()
```

```
def totalLossRDD(dataRDD,beta,lam = 0.0):
    """  Given a sparse vector beta and a dataset in the form of RDD compute the regularized total logistic loss :

            L(β) = Σ_{(x,y) in data}  l(β;x,y)  + λ ||β ||_2^2

        Inputs are:
            - dataRDD: an RDD containing pairs of the form (x,y), where x is a sparse vector and y is a binary value
            - beta: a sparse vector β
            - lam: the regularization parameter λ

        The return value is a float number represents the total loss
    """
    loss = dataRDD.map(lambda (x,y): logisticLoss(beta, x, y))\
                  .reduce(add)
    return loss + lam * beta.dot(beta)
```

```
def gradTotalLossRDD(dataRDD,beta,lam = 0.0):
    """  Given a sparse vector beta and a dataset in the form of RDD, compute the gradient of regularized total logistic loss :

                ∇L(β) = Σ_{(x,y) in data}  ∇l(β;x,y)  + 2λ β

        Inputs are:
           - data: an RDD containing pairs of the form (x,y), where x is a sparse vector and y is a binary value
           - beta: a sparse vector β
           - lam: the regularization parameter λ

        The return value is a float number represents the gradient of total loss
    """
    gradLoss = dataRDD.map(lambda (x, y): gradLogisticLoss(beta, x, y))\
                      .reduce(add)
    return gradLoss + 2.0*lam*beta
```

```
def test(dataRDD,beta):
    """ Output the quantities necessary to compute the accuracy, precision, and recall of the prediction of labels in a dataset under a given β.

        The accuracy (ACC), precision (PRE), and recall (REC) are defined in terms of the following sets:

                P = datapoints (x,y) in data for which <β,x> > 0
                N = datapoints (x,y) in data for which <β,x> <= 0

                TP = datapoints in (x,y) in P for which y=+1
                FP = datapoints in (x,y) in P for which y=-1
                TN = datapoints in (x,y) in N for which y=-1
                FN = datapoints in (x,y) in N for which y=+1

        For #XXX the number of elements in set XXX, the accuracy, precision, and recall of parameter vector β over data are defined as:

                ACC(β,data) = ( #TP+#TN ) / (#P + #N)
                PRE(β,data) = #TP / (#TP + #FP)
                REC(β,data) = #TP/ (#TP + #FN)

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

        The return values are
             - ACC, PRE, REC

    """
    P_set = dataRDD.filter(lambda (x,y): x.dot(beta) > 0).cache()
    N_set = dataRDD.filter(lambda (x,y): x.dot(beta) <= 0).cache()
    P = P_set.count()
    N = N_set.count()
    TP = P_set.filter(lambda (x,y): y == 1).count()
    FP = P - TP
    TN = N_set.filter(lambda (x,y): y == -1).count()
    FN = N - TN
    ACC = (TP + TN) / float(P + N)
    PRE = TP / float(TP + FP)
    REC = TP / float(TP + FN)
    return ACC, PRE, REC
```

```
def train(dataRDD,beta_0,lam,max_iter,eps,test_data=None):
    """ Perform parallel logistic regression:

        to  minimize

                L(β) = Σ_{(x,y) in data}  l(β;x,y)  + λ ||β ||_2^2

    where the inputs are:
             - dataRDD: an rdd containing pairs of the form (x,y) as train data
             - beta_0: the starting vector β
             - lam:  is the regularization parameter λ
             - max_iter: maximum number of iterations of gradient descent
             - eps: upper bound on the l2 norm of the gradient
             - test_data: an rdd containing pairs of the form (x,y) as test data

    The function returns:
         -beta: the trained β,
         -gradNorm: the norm of the gradient at the trained β, and
         ├k: the number of iterations performed
    """
    k = 0
    gradNorm = 2*eps
    beta = beta_0
    start = time()
    while k<max_iter and gradNorm > eps:
        obj = totalLossRDD(dataRDD,beta,lam)

        grad = gradTotalLossRDD(dataRDD,beta,lam)
        gradNormSq = grad.dot(grad)
        gradNorm = np.sqrt(gradNormSq)

        fun = lambda x: totalLossRDD(dataRDD,x,lam)
        gamma = lineSearch(fun,beta,grad,obj,gradNormSq)

        beta = beta - gamma * grad
        if test_data == None:
            print 'k = ',k,'\tt = ',time()-start,'\tL(β_k) = ',obj,'\t||∇L(β_k)||_2 = ',gradNorm,'\tgamma = ',gamma
        else:
            acc,pre,rec = test(test_data,beta)
            print 'k = ',k,'\tt = ',time()-start,'\tL(β_k) = ',obj,'\t||∇L(β_k)||_2 = ',gradNorm,'\tgamma = ',gamma,'\tACC = ',acc,'\tPRE = ',pre,'\tREC = ',rec
        k = k + 1

    return beta,gradNorm,k
```

```python
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description = 'Parallel Logistic Regression.',formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('traindata',default=None, help='Input file containing (x,y) pairs, used to train a logistic model')
    parser.add_argument('--testdata',default=None, help='Input file containing (x,y) pairs, used to test a logistic model')
    parser.add_argument('--beta', default='beta', help='File where beta is stored (when training) and read from (when testing)')
    parser.add_argument('--lam', type=float,default=0.0, help='Regularization parameter λ')
    parser.add_argument('--max_iter', type=int,default=100, help='Maximum number of iterations')
    parser.add_argument('--eps', type=float, default=0.1, help='ε-tolerance. If the l2_norm gradient is smaller than ε, gradient descent terminates.')


    args = parser.parse_args()

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

    print 'Reading training data from',args.traindata
    traindata = readDataRDD(args.traindata, sc).cache()
    print 'Read',traindata.count(),'data points with',len(getAllFeaturesRDD(traindata)),'features in total'

    if args.testdata is not None:
        print 'Reading test data from',args.testdata
        testdata = readDataRDD(args.testdata, sc).cache()
        print 'Read',testdata.count(),'data points with',len(getAllFeaturesRDD(testdata)),'features'
    else:
        testdata = None

    beta0 = SparseVector({})

    print 'Training on data from',args.traindata,'with λ =',args.lam,', ε =',args.eps,', max iter = ',args.max_iter
    beta, gradNorm, k = train(traindata,beta_0=beta0,lam=args.lam,max_iter=args.max_iter,eps=args.eps,test_data=testdata)
    print 'Algorithm ran for',k,'iterations. Converged:',gradNorm<args.eps
    print 'Saving trained β in',args.beta
    writeBeta(args.beta,beta)
```

4(b) Results on mushrooms dataset

This is the previous result using LogisticRegression.py

```
[wang.zife@c0013 Assignment3]$ python LogisticRegression.py mushrooms/mushrooms.train --testdata mushrooms/mushrooms.test --lam 0.0 --max_iter 20
Reading training data from mushrooms/mushrooms.train
Read 7416 data points with 117 features in total
Reading test data from mushrooms/mushrooms.test
Read 1000 data points with 115 features
Training on data from mushrooms/mushrooms.train with λ = 0.0 , ε = 0.1 , max iter =  20
k =  0  t =  3.35645294189      L(β_k) =  5140.379491033203     ||∇L(β_k)||_2 =  4273.548233026041      gamma =  0.0004701849845 76      ACC =  0.909    PRE =  0.855325914149   REC =  1.0
k =  1  t =  7.15586996078      L(β_k) =  2516.9939944922876    ||∇L(β_k)||_2 =  3275.986464902974      gamma =  0.0001692665944 47      ACC =  0.92     PRE =  0.892123287671   REC =  0.968401486989
k =  2  t =  10.0537118912      L(β_k) =  1536.629608915103     ||∇L(β_k)||_2 =  802.3372840311758      gamma =  0.0036279705 6  ACC =  0.979    PRE =  0.965765765766   REC =  0.996282527881
k =  3  t =  13.5136380196      L(β_k) =  772.3391480787859     ||∇L(β_k)||_2 =  980.5492791241032      gamma =  0.0004701849845 76      ACC =  0.974    PRE =  0.981203007519   REC =  0.970260223048
k =  4  t =  16.9038770199      L(β_k) =  648.9689952939555     ||∇L(β_k)||_2 =  636.7301783321042      gamma =  0.0004701849845 76      ACC =  0.976    PRE =  0.972426470588   REC =  0.983271375465
k =  5  t =  20.1671419144      L(β_k) =  583.1293003296519     ||∇L(β_k)||_2 =  390.12150441871074     gamma =  0.0007836416409 6       ACC =  0.975    PRE =  0.981238273921   REC =  0.972118959108
k =  6  t =  23.5557079315      L(β_k) =  554.8636871007043     ||∇L(β_k)||_2 =  474.3895736703113      gamma =  0.0004701849845 76      ACC =  0.981    PRE =  0.977900552486   REC =  0.986988847584
k =  7  t =  26.8180449009      L(β_k) =  511.65898556365914    ||∇L(β_k)||_2 =  258.88738725459575     gamma =  0.0007836416409 6       ACC =  0.98     PRE =  0.981412639405   REC =  0.981412639405
k =  8  t =  30.066906929       L(β_k) =  486.9422819846264     ||∇L(β_k)||_2 =  276.73548293899021     gamma =  0.0007836416409 6       ACC =  0.984    PRE =  0.976277372263   REC =  0.994423791822
k =  9  t =  33.3298799992      L(β_k) =  467.33336504481593    ||∇L(β_k)||_2 =  297.33568125873046     gamma =  0.0007836416409 6       ACC =  0.979    PRE =  0.981378026071   REC =  0.979553903346
k =  10         t =  36.7253370285      L(β_k) =  449.7683227925332     ||∇L(β_k)||_2 =  319.5307352237093      gamma =  0.0004701849845 76      ACC =  0.984    PRE =  0.979779411765   REC =  0.990706319703
k =  11         t =  39.7053449154      L(β_k) =  425.92454429676246    ||∇L(β_k)||_2 =  166.721201409282 8     gamma =  0.0021767823 36         ACC =  0.983    PRE =  0.988742964353   REC =  0.979553903346
k =  12         t =  43.0983970165      L(β_k) =  405.9943035852676     ||∇L(β_k)||_2 =  388.19925909616404     gamma =  0.0004701849845 76      ACC =  0.986    PRE =  0.979853479853   REC =  0.994423791822
k =  13         t =  46.0599400997      L(β_k) =  374.2509647417060 7   ||∇L(β_k)||_2 =  147.16737512248403     gamma =  0.0021767823 36         ACC =  0.987    PRE =  0.994350282486   REC =  0.981412639405
k =  14         t =  49.4615418911      L(β_k) =  364.10371130630483    ||∇L(β_k)||_2 =  357.7097348748522      gamma =  0.0004701849845 76      ACC =  0.989    PRE =  0.985267034991   REC =  0.994423791822
k =  15         t =  52.2741019726      L(β_k) =  335.53335812252476    ||∇L(β_k)||_2 =  119.37562057442935     gamma =  0.0036279705 6  ACC =  0.989    PRE =  0.994371482176   REC =  0.985130111524
k =  16         t =  55.691090107       L(β_k) =  321.4021032504561     ||∇L(β_k)||_2 =  359.39323294505124     gamma =  0.0004701849845 76      ACC =  0.994    PRE =  0.992592592593   REC =  0.996282527881
k =  17         t =  58.0605199337      L(β_k) =  290.83776835018057    ||∇L(β_k)||_2 =  93.278303407238        gamma =  0.0167961 6     ACC =  0.99     PRE =  0.994382022472   REC =  0.986988847584
k =  18         t =  61.3027141094      L(β_k) =  238.23480047134834    ||∇L(β_k)||_2 =  427.83733773957826     gamma =  0.0007836416409 6       ACC =  0.997    PRE =  0.994454713494   REC =  1.0
k =  19         t =  64.4026780128      L(β_k) =  183.51233512776878    ||∇L(β_k)||_2 =  104.68994981492128     gamma =  0.0013060694016         ACC =  0.997    PRE =  0.994454713494   REC =  1.0
Algorithm ran for 20 iterations. Converged: False
Saving trained β in beta
```

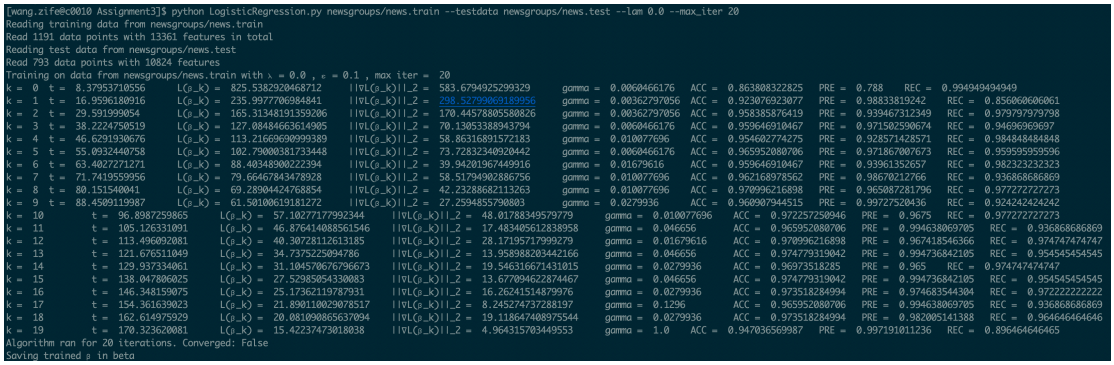
Here below is the result using parallel implementation

```
[wang.zife@c0010 Assignment3]$ python ParallelLogisticRegression.py mushrooms/mushrooms.train --testdata mushrooms/mushrooms.test --lam 0.0 --max_iter 20
2019-03-18 00:08:02 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Reading training data from mushrooms/mushrooms.train
Read 7416 data points with 117 features in total
Reading test data from mushrooms/mushrooms.test
Read 1000 data points with 115 features
Training on data from mushrooms/mushrooms.train with λ = 0.0 , ε = 0.1 , max iter =  20
k =  0  t =  3.60561108589      L(β_k) =  5140.379491032573     ||∇L(β_k)||_2 =  4273.548233026041      gamma =  0.0004701849845 76      ACC =  0.909    PRE =  0.855325914149   REC =  1.0
k =  1  t =  7.91716504097      L(β_k) =  2516.9939944922844    ||∇L(β_k)||_2 =  3275.9864649029773     gamma =  0.0001692665944 47      ACC =  0.92     PRE =  0.892123287671   REC =  0.968401486989
k =  2  t =  10.8794829845      L(β_k) =  1536.6296089151083    ||∇L(β_k)||_2 =  802.3372840311177      gamma =  0.0036279705 6  ACC =  0.979    PRE =  0.965765765766   REC =  0.996282527881
k =  3  t =  14.4739990234      L(β_k) =  772.3391480788059     ||∇L(β_k)||_2 =  980.5492791241635      gamma =  0.0004701849845 76      ACC =  0.974    PRE =  0.981203007519   REC =  0.970260223048
k =  4  t =  18.3914749622      L(β_k) =  648.9689952939462     ||∇L(β_k)||_2 =  636.7301783321427      gamma =  0.0004701849845 76      ACC =  0.976    PRE =  0.972426470588   REC =  0.983271375465
k =  5  t =  21.7866768837      L(β_k) =  583.1293003296535     ||∇L(β_k)||_2 =  390.12150441872 9      gamma =  0.0007836416409 6       ACC =  0.975    PRE =  0.981238273921   REC =  0.972118959108
k =  6  t =  25.2595560551      L(β_k) =  554.8636871007066     ||∇L(β_k)||_2 =  474.3895736703964      gamma =  0.0004701849845 76      ACC =  0.981    PRE =  0.977900552486   REC =  0.986988847584
k =  7  t =  28.584913969       L(β_k) =  511.65898556366611    ||∇L(β_k)||_2 =  258.88738725460325     gamma =  0.0007836416409 6       ACC =  0.98     PRE =  0.981412639405   REC =  0.981412639405
k =  8  t =  32.0210180283      L(β_k) =  486.9422819846284     ||∇L(β_k)||_2 =  276.73548298991193     gamma =  0.0007836416409 6       ACC =  0.984    PRE =  0.976277372263   REC =  0.994423791822
k =  9  t =  35.5006780624      L(β_k) =  467.3333650448169     ||∇L(β_k)||_2 =  297.3356812587419      gamma =  0.0007836416409 6       ACC =  0.979    PRE =  0.981378026071   REC =  0.979553903346
k =  10         t =  39.0885100365      L(β_k) =  449.76832279253426    ||∇L(β_k)||_2 =  319.53073522372 25     gamma =  0.0004701849845 76      ACC =  0.984    PRE =  0.979779411765   REC =  0.990706319703
k =  11         t =  42.1269369125      L(β_k) =  425.9245442967608     ||∇L(β_k)||_2 =  166.721201409291       gamma =  0.0021767823 36         ACC =  0.983    PRE =  0.988742964353   REC =  0.979553903346
k =  12         t =  45.8038110733      L(β_k) =  405.9943035852711     ||∇L(β_k)||_2 =  388.1992590961811      gamma =  0.0004701849845 76      ACC =  0.986    PRE =  0.979853479853   REC =  0.994423791822
k =  13         t =  49.0539588928      L(β_k) =  374.2509647417046     ||∇L(β_k)||_2 =  147.1673751224859      gamma =  0.0021767823 36         ACC =  0.987    PRE =  0.994350282486   REC =  0.981412639405
k =  14         t =  52.7122299671      L(β_k) =  364.1037113063075     ||∇L(β_k)||_2 =  357.70973487486793     gamma =  0.0004701849845 76      ACC =  0.989    PRE =  0.985267034991   REC =  0.994423791822
k =  15         t =  55.5561490059      L(β_k) =  335.5333581225248     ||∇L(β_k)||_2 =  119.37562057443033     gamma =  0.0036279705 6  ACC =  0.989    PRE =  0.994371482176   REC =  0.985130111524
k =  16         t =  58.8773890078      L(β_k) =  321.4021032504561     ||∇L(β_k)||_2 =  359.39323294506938     gamma =  0.0004701849845 76      ACC =  0.994    PRE =  0.992592592593   REC =  0.996282527881
k =  17         t =  61.3168818951      L(β_k) =  290.8377683501805     ||∇L(β_k)||_2 =  93.278303407238        gamma =  0.0167961 6     ACC =  0.99     PRE =  0.994382022472   REC =  0.986988847584
k =  18         t =  64.6839189529      L(β_k) =  238.23480047135536    ||∇L(β_k)||_2 =  427.8373373958149      gamma =  0.0007836416409 6       ACC =  0.997    PRE =  0.994454713494   REC =  1.0
k =  19         t =  67.7878470421      L(β_k) =  183.51233512776867    ||∇L(β_k)||_2 =  104.68994981492934     gamma =  0.0013060694016         ACC =  0.997    PRE =  0.994454713494   REC =  1.0
Algorithm ran for 20 iterations. Converged: False
Saving trained β in beta
```

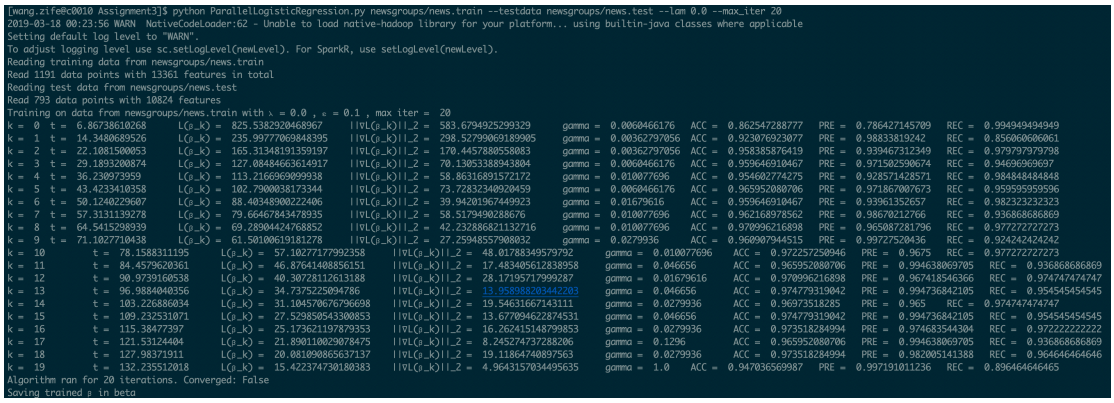
Here the LogisticRegression.py is a little bit faster (64.4 seconds V.S. 67.8 seconds), but I think the difference is not that outstanding. So the running times are comparable, maybe if I optimize the parallel version better, I could get better results. But since the dimensionality of the data is not that big, a simple for loop may outperform map-reduce procedure by not having communication and data transfer time etc.

4(c) Results on newsgroups dataset

This is the previous result using LogisticRegression.py



Here below is the result using parallel implementation



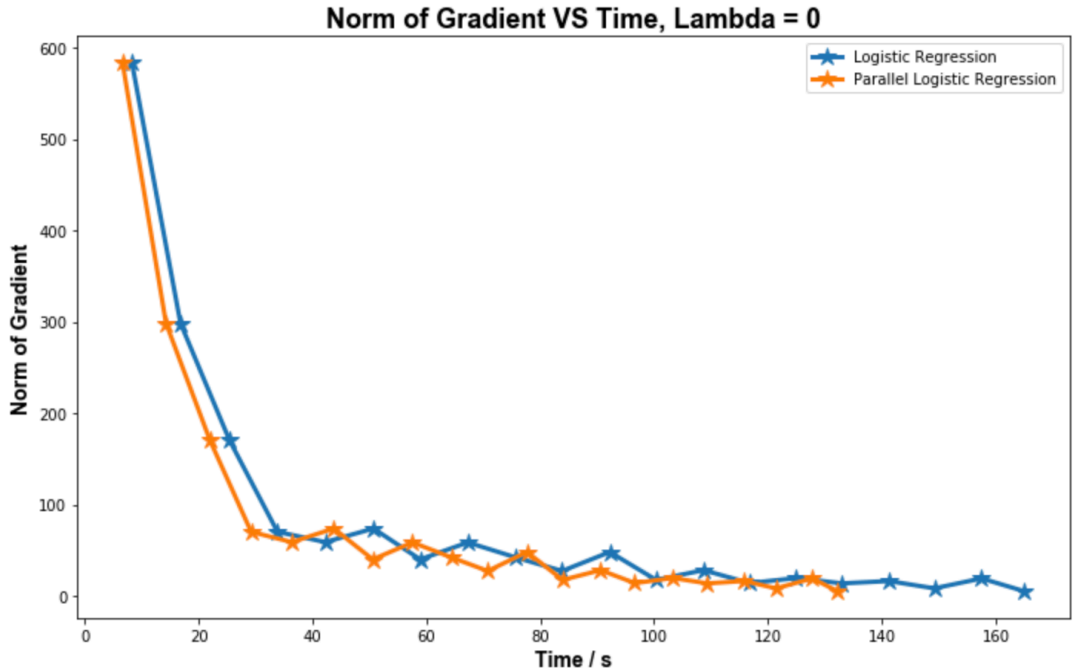
Here ParallelLogisticRegression.py is faster (132.2 seconds V.S. 170.3 seconds). The difference is considerable and it is because of the high dimensionality of the data make map-reduce more efficient than a single for loop.

Problem 5

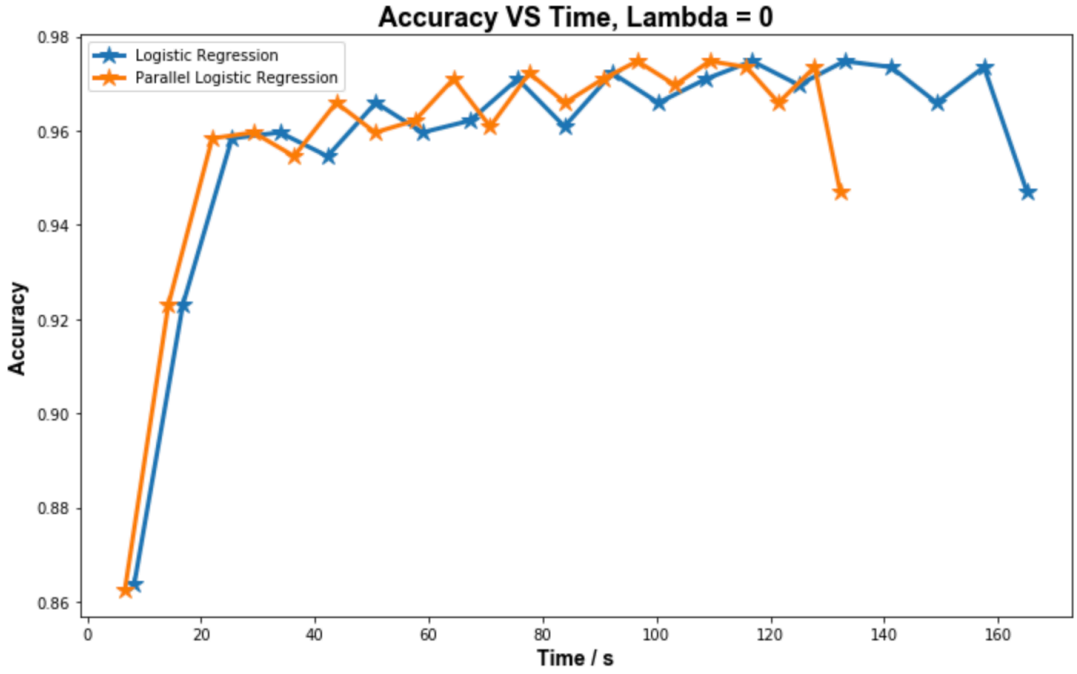
For convenience, I modified train() in both LogisticRegression.py and ParallelLogisticRegression.py to save certain variables for plotting. But I did not reflect the changes in the previous questions, so please refer to the final version attached.
Also plots are created on local machine using pyplot and Jupyter Notebook, an example snippet is below (all the plots follows this example except the parameters):

```
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import pickle
import os
root_path = '/Users/wangzifeng/Desktop/Courses@NEU/Parallel Programming/Homework/Ass3/'
time_lr = pickle.load(open(os.path.join(root_path, 'LR', 'time.pkl'),'rb'))
time_plr =pickle.load(open(os.path.join(root_path, 'PLR', 'time.pkl'),'rb'))
gradNorm_lr = pickle.load(open(os.path.join(root_path, 'LR', 'gradNorm.pkl'),'rb'))
gradNorm_plr =pickle.load(open(os.path.join(root_path, 'PLR', 'gradNorm.pkl'),'rb'))
acc_lr = pickle.load(open(os.path.join(root_path, 'LR', 'acc.pkl'),'rb'))
acc_plr =pickle.load(open(os.path.join(root_path, 'PLR', 'acc.pkl'),'rb'))
pre_lr = pickle.load(open(os.path.join(root_path, 'LR', 'pre.pkl'),'rb'))
pre_plr =pickle.load(open(os.path.join(root_path, 'PLR', 'pre.pkl'),'rb'))
rec_lr = pickle.load(open(os.path.join(root_path, 'LR', 'rec.pkl'),'rb'))
rec_plr =pickle.load(open(os.path.join(root_path, 'PLR', 'rec.pkl'),'rb'))

f = plt.figure(figsize=(12,7.2))

plt.plot(time_lr, gradNorm_lr, '-*', label = 'Logistic Regression', linewidth=3.0, markersize=12)
plt.plot(time_plr, gradNorm_plr, '-*', label = 'Parallel Logistic Regression', linewidth=3.0, markersize=12)

plt.legend()
plt.title("Norm of Gradient VS Time, Lambda = 0", fontsize='18', weight='bold', fontname='Arial')
plt.xlabel("Time / s", fontsize='14', weight='bold',fontname='Arial')
plt.ylabel("Norm of Gradient", fontsize='14',weight='bold',fontname='Arial')

plt.show()
```

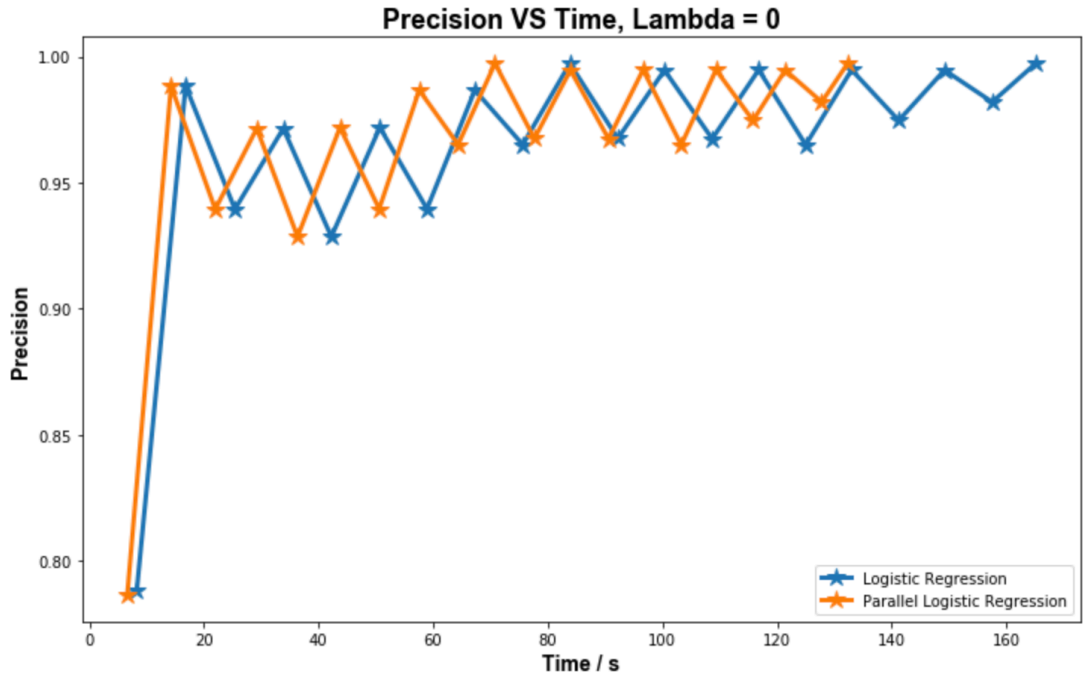
5(a) For each experiments, I run 20 iterations
Plot for norm of gradient vs time.



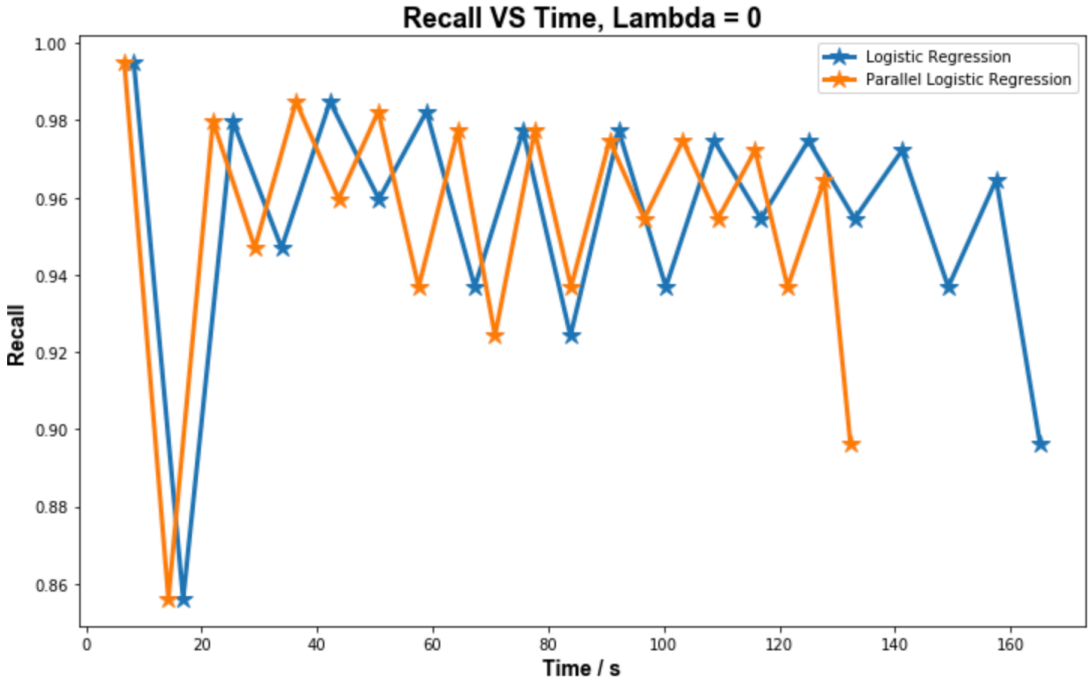
Plot for accuracy vs time



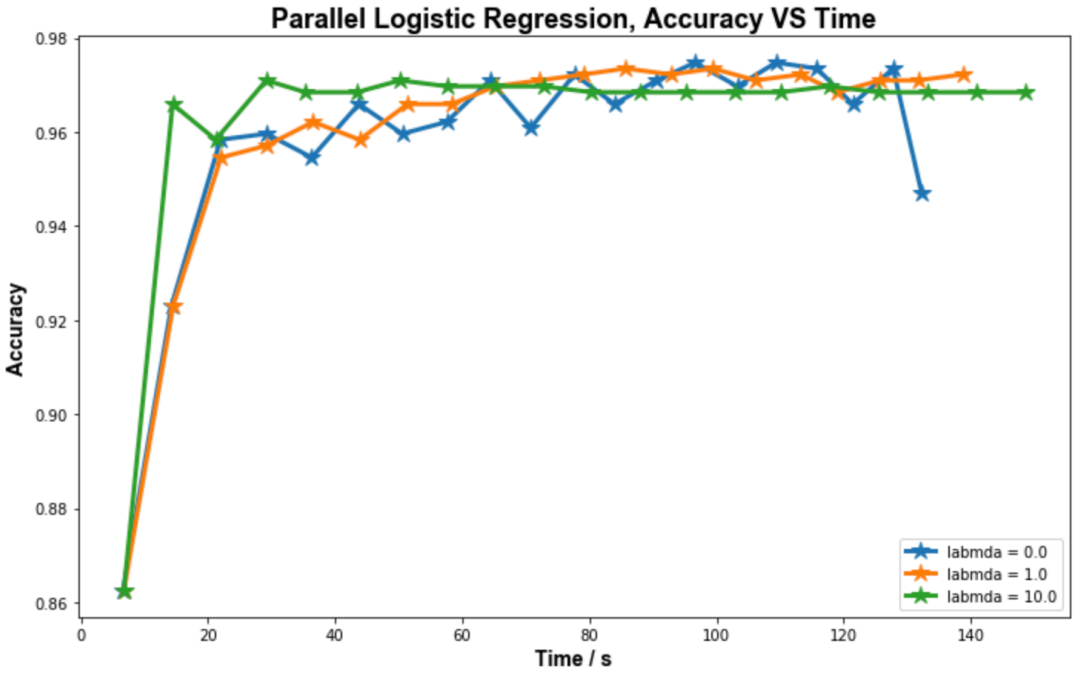
Plot for precision vs time



Plot for recall vs time



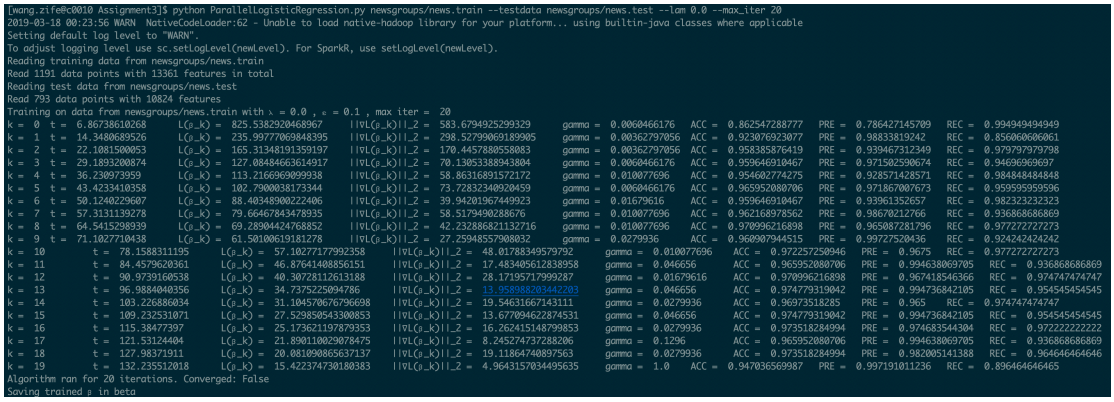
5(b) We choose λ from {0.0, 1.0, 10.0}, also for 20 epochs
For ParallelLogisticRegression.py



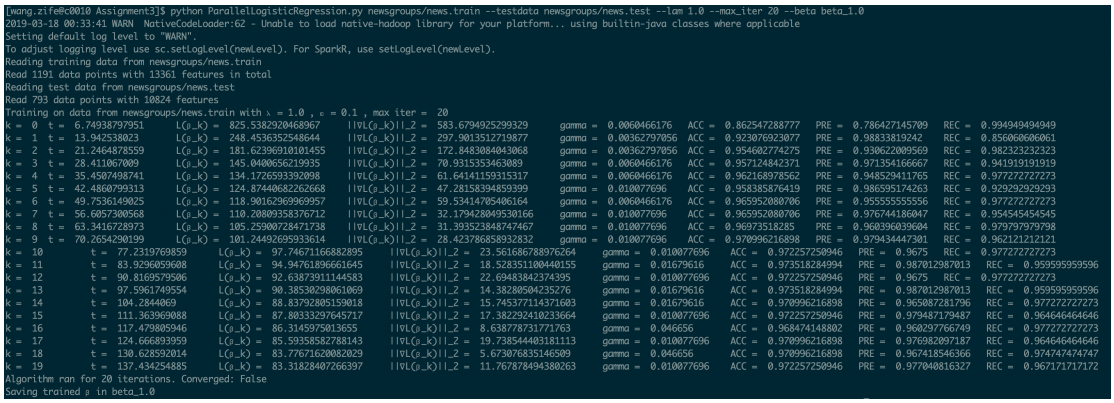
5(c)
From the plot in 5(b), we could see that the maximum accuracy is got when λ = 0.0, but still we need to verify which iteration is the best one (hard to tell from the plot), so I give the exact accuracy under each λ to find out which is the best.

The exact accuracy is as below (the numbers and corresponding iteration number should be the same between Logistic Regression and Parallel Logistic Regression.)

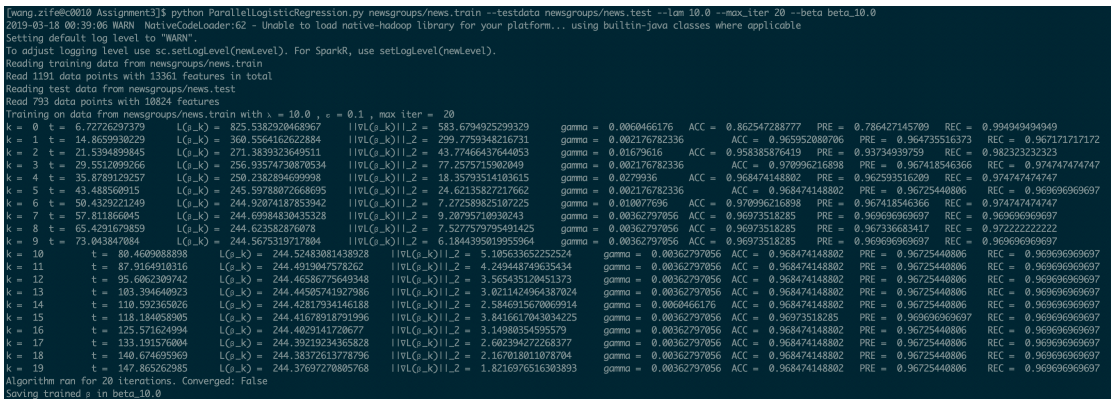
$\lambda = 0.0$



$\lambda = 1.0$



$\lambda = 10.0$



As a result, **λ = 0.0, k = 13 or λ = 0.0 k = 15** seems to be the best combination, they give exactly the same performance.
Here I just choose **λ = 0.0, k = 13** and find the corresponding beta below:

Features with 10 Most Positive Values

Feature	Value
doctor	0.739256993046
medic	0.654923809228
inform	0.64845288521
diseas	0.627412769331
treatment	0.594065145151
effect	0.588228768705
problem	0.542908430112
gordon	0.522539524406
peopl	0.509126766275
bank	0.505400995555

Features with 10 Most Negative Values

Feature	Value
basebal	-1.59643808354
game	-1.35316118574
team	-1.1894061731
player	-1.16455349865
win	-0.88657443609
plai	-0.885072905804
pitch	-0.79174957004
run	-0.780232214032
fan	-0.745623085766
yanke	-0.733704446503