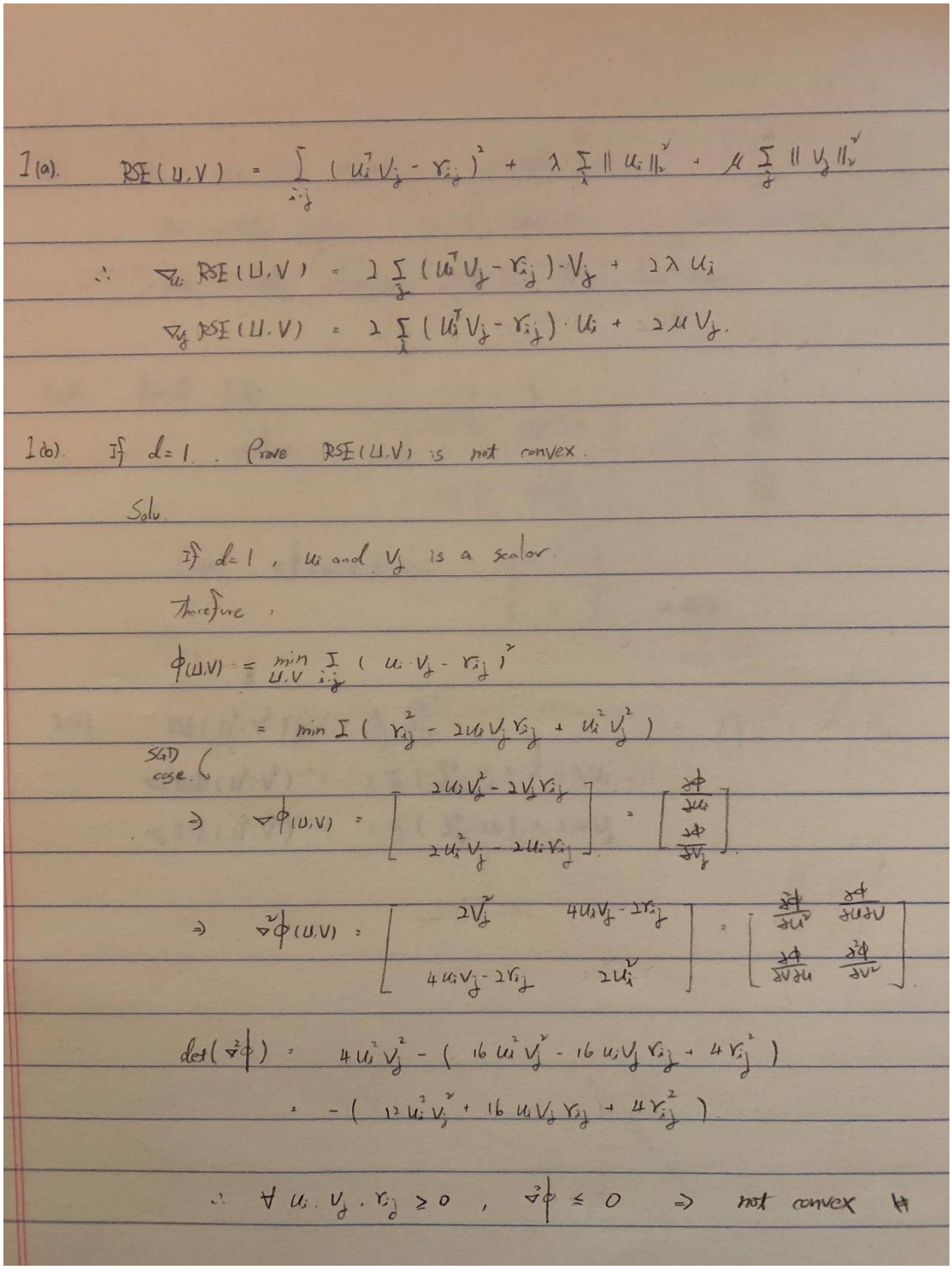
ST: Parallel Analysis Assignment #4

Kuan-Chih Lee

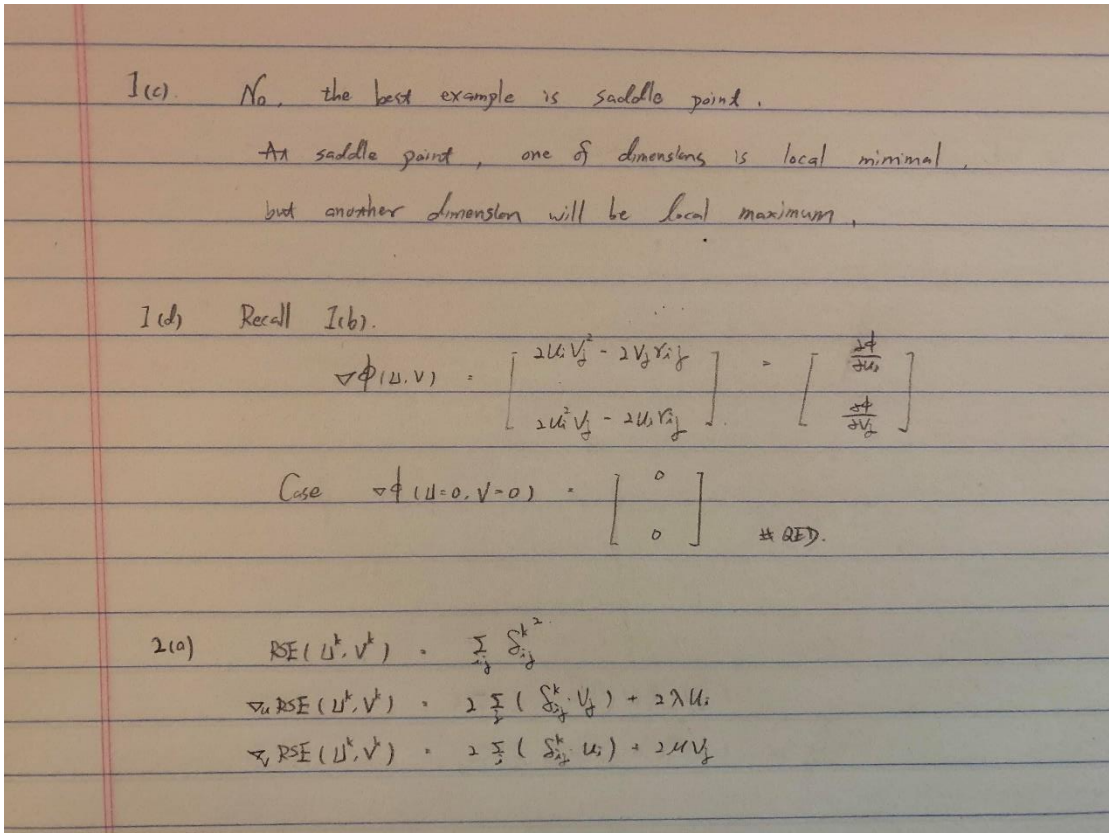
Instructor: Prof. Stratis Ioannidis

Question 1:

1a. to 1b.



1c. to 2a.



Question 2:

2b.

```
def predict(u,v):
    """ Given a user profile uprof and an item profile vpro

        Inputs are:
           -u: user profile, in the form of a numpy array
           -v: item profile, in the form of a numpy array

        The return value is
           - the inner product <u,v>
    """
    return u.T.dot(v)

def pred_diff(r,u,v):
    """ Given a rating, a user profile u and an item profil

        Inputs are:
           -r: the rating a user gave to an item
           -u: user profile, in the form of a numpy array
           -v: item profile, in the form of a numpy array

        The return value is the difference
          - 都 =  <u,v> - r
    """
    return predict(u,v) - r
```

```
def gradient_u(delta,u,v):
    """ Given a user profile u and an item profile v, and th

            ?  u l(u,v)   = 2 (<u,v> - r ) v

           of the square error loss:

            l(u,v) = (<u,v> - r)^2

        Inputs are:
           -帮: the difference  <u,v> - r
           -u: user profile, in the form of a numpy array
           -v: item profile, in the form of a numpy array

        The return value is
          - The gradient w.r.t. u
    """
    return 2*delta*v

def gradient_v(delta,u,v):
    """ Given a user profile u and an item profile v, and th

            ?  v l(u,v)= 2 (<u,v> - r) u

        of the square error loss:

            l(u,v) = (<u,v> - r)^2

        Inputs are:
           -帮: the difference  <u,v> - r
           -u: user profile, in the form of a numpy array
           -v: item profile, in the form of a numpy array

        The return value is
          - the gradient w.r.t. v
    """
    return 2*delta*u
```

2c.

```
def generateItemProfiles(R,d,seed,sparkContext,N):
    """ Generate the item profiles from rdd R and store them in an RDD containing tuples of the form
         (j,vj)
       where v is a random np.array of dimension d.

       The random uis are generated using normalVectorRDD(), a function in RandomRDDs.

       Inputs are:
            - R: an RDD that contains the ratings in (user, item, rating) form
            - d: the dimension of the user profiles
            - seed: a seed to be used for in generating the random vectors
            - sparkContext: a spark context
            - N: the number of partitions to be used during joins, etc.

       The return value is an RDD containing the item profiles
    """
    V = R.map(lambda (i,j,rij):j).distinct(numPartitions = N)
    numItems = V.count()
    randRDD = RandomRDDs.normalVectorRDD(sparkContext, numItems, d, numPartitions=N, seed=seed)
    V = V.zipWithIndex().map(swap)
    randRDD = randRDD.zipWithIndex().map(swap)
    return V.join(randRDD,numPartitions = N).values()
```

Why do we initialize the user and item profiles to random values? What would happen if we initialized all profiles to be zero vectors?

Ans:

If we initialize user and item profiles with zero vectors, the model will lose the capability of training. Recall question 2(b), the gradient, as well as regularization term, is the product of user or item profile.

Question 4:

4a.

```
def joinAndPredictAll(R,U,V,N):
    """ Receives as inputs the ratings R, the user profiles U, and the items V, and constructs a joined RDD.

        Inputs are:
         - R: an RDD containing tuples of the form (i,j,rij)
         - U: an RDD containing tuples of the form (i,ui)
         - V: an RDD containing tuples of the form (j,vj)
         - N: the number of partitions to be used during joins, etc.

        The output is a joined RDD containing tuples of the form:

        (i,j,帮ij,ui,vj)

        where
          帮ij = <u,v>-rij
        is the prediction difference.

    """
    return R.map(lambda (i,j,rij): (i, (rij,j))).join(U, numPartitions = N) \
            .map(lambda (i, ((rij,j),ui)): (j, (i,ui,rij))).join(V, numPartitions = N) \
            .map(lambda (j, ((i,ui,rij),vj)): (i, j, pred_diff(rij,ui,vj), ui, vj))
```

4b.

```
def SE(joinedRDD):
    """ Receives as input a joined RDD as well as a 弇 and a 卷 and computes the MSE:

        SE(R,U,V) = 峉_{i,j in data} (<ui,vj>-rij)^2

        The input is
         -joinedRDD: an RDD with tuples of the form (i,j,帮ij,ui,vj), where 帮ij = <ui,vj> - rij is the prediction difference.

        The output is the SE.
    """
    return joinedRDD.map(lambda (i, j, delta, ui, vj): delta**2).reduce(add)

def normSqRDD(profileRDD,param):
    """ Receives as input an RDD of profiles (e.g., U) as well as a parameter (e.g., 弇) and computes the square of norms:
        弇 峉_i ||ui||_2^2

        The input is:
          -profileRDD: an RDD of the form (i,u), where i is an index and u is a numpy array
          -param: a scalar 弇>0

        The return value is:
        弇 峉_i ||ui||_2^2
    """
    return param * profileRDD.map(lambda (i, x): np.dot(x.T, x)).reduce(add)
```

SE is loss function, and normSqRDD is regularization term

4c.

```
def adaptU(joinedRDD,gamma,lam,N):
    """ Receives as input a joined RDD
        as well as a gain 吊, and regularization parameters 弁 and 卷,  and constructs a new RDD of user profiles of the form

        ui = ui - 吊  ?  ui RegSE(R,U,V)

        where

        RegSE(R,U,V) = 峯_{i,j in R} (<ui,vj>-rij)^2 + 弁 峯_i ||ui||_2^2 + 卷 峯_j ||vj||_2^2


        Inputs are
         -joinedRDD: an RDD with tuples of the form (i,j,帤ij,ui,vj), where 帤ij = <ui,vj> - rij
         -gamma: the gain 吊
         -lam: the regularization parameter 弁
         -N: the number of partitions to be used in reduceByKey operations

        The return value  is an RDD with tuples of the form (i,ui). The returned rdd contains exactly N partitions.
    """
    U_t0 = joinedRDD.map(lambda (i, j, delta, ui, vj): (i, ui)).reduceByKey(lambda x,y: x).repartition(N)
    grad = joinedRDD.map(lambda (i, j, delta, ui, vj): (i, gradient_u(delta,ui,vj))).reduceByKey(add).repartition(N)
    return U_t0.join(grad).map(lambda (i, (ui,g)): (i, ui - gamma*(g+2*lam*ui)))
```

```
def adaptV(joinedRDD,gamma,mu,N):
    """ Receives as input a joined RDD
        as well as a gain 吊, and regularization parameters 弁 and 卷,  and constructs a new RDD of user profiles of the form

        ui = ui - 吊  ?  ui RegSE(R,U,V)

        where

        RegSE(R,U,V) = 峯_{i,j in R} (<ui,vj>-rij)^2 + 弁 峯_i ||ui||_2^2 + 卷 峯_j ||vj||_2^2


        Inputs are
         -joinedRDD: an RDD with tuples of the form (i,j,帤ij,ui,vj), where 帤ij = <ui,vj> - rij
         -gamma: the gain 吊
         -mu: the regularization parameter 卷
         -N: the number of partitions to be used in reduceByKey operations

        The return value  is an RDD with tuples of the form (j,vj). The returned rdd contains exactly N partitions.
    """
    V_t0 = joinedRDD.map(lambda (i, j, delta, ui, vj): (j, vj)).reduceByKey(lambda x,y: x).repartition(N)
    grad = joinedRDD.map(lambda (i, j, delta, ui, vj): (j, gradient_v(delta,ui,vj))).reduceByKey(add).repartition(N)
    return V_t0.join(grad).map(lambda (j, (vj,g)): (j, vj - gamma*(g++2*mu*vj)))
```

Question 5:

5a.

```
265     folds = {}
266
267     if args.output is None:
268         for k in range(args.folds):
269             folds[k] = readRatings(args.data+"/fold"+str(k),sc)
270     else:
271         folds[0] = readRatings(args.data,sc)
272
273     cross_val_rmses = []
274     for k in folds:
275         train_folds = [folds[j] for j in folds if j is not k ]
276
277         if len(train_folds)>0:
278             train = train_folds[0]
279             for fold in  train_folds[1:]:
280                 train=train.union(fold)
281             train.repartition(args.N).cache()
282             test = folds[k].repartition(args.N).cache()
283             Mtrain=train.count()
284             Mtest=test.count()
285
286             print("Initiating fold %d with %d train samples and %d test samples" % (k,Mtrain,Mtest) )
287         else:
288             train = folds[k].repartition(args.N).cache()
289             test = train
290             Mtrain=train.count()
291             Mtest=test.count()
292             print("Running single training over training set with %d train samples. Test RMSE computes RMSE on training set" % Mtrain )
```

Data collection:

Line 267:

If args.output is not None, cross validation is skipped, and U,V are trained over entire dataset and store it in files output_U and output_V.'

Line 268-269

Read k in k-fold by args.folds. And then, read data from files under arg.data/ folder

and store in RDD in a dictionary called fold.

K-fold CV started (Line 274)

Line 275:

Store different folds, except the fold picked in Line 274, in a list called train_folds.

Line 279-280:

Pop out repeated data within different folds.

Line 281:

Training data will be the combination of k-1 folds

Testing data will be the fold picked at Line 274

5b.

Take question 5c as an example.

Original gamma is 0.001 (args.gain) and decay parameter is 0.2 (args.pow).

So, the first iteration adjusted gamma will be gamma / 1^0.2.

Second iteration adjusted gamma will be gamma / 2^0.2

20^{th} iteration adjusted gamma will be gamma / 20^0.2

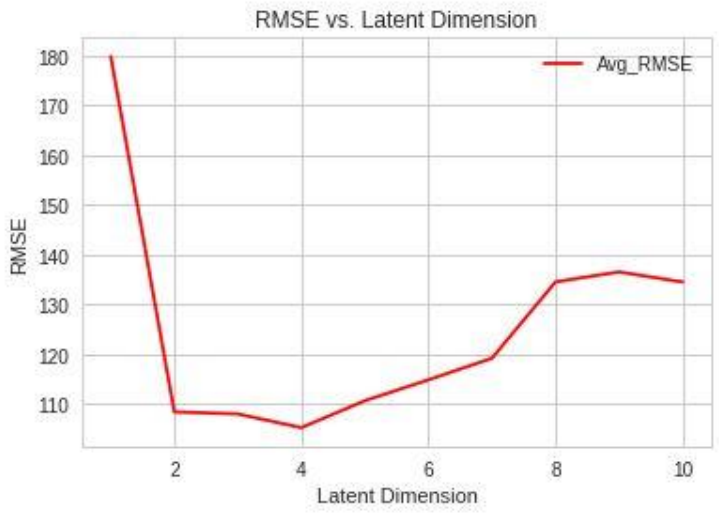
Formula:

$$\gamma \leftarrow \frac{\gamma}{\text{epoch}^{pow}}$$

5c.

```
Best Latent 4, Best Regul 0, RMSE 105.232654
```

```
df = pd.read_csv("A4_plotCSV_1", names=["Latent", "Regul", "Avg_RMSE"])
plt.plot(df.Latent, df.Avg_RMSE, c='r')
plt.xlabel('Latent Dimension')
plt.ylabel('RMSE')
plt.title('RMSE vs. Latent Dimension')
plt.legend()
plt.show()
```



5d.

It’s not converged.

--gain 0.1 --pow 0.0 --maxiter 5

```
Initiating fold 4 with 40 train samples and 10 test samples
Training set contains 10 users and 9 items
Iteration: 1    Time: 1.557365  Objective: 1065416.059115       TestRMSE: 143.052712
Iteration: 2    Time: 4.907680  Objective: 4708796697.763937    TestRMSE: 4176.174975
Iteration: 3    Time: 8.185757  Objective: 7161332518230058591689l136.000000    TestRMSE: 70913861720.276382
Iteration: 4    Time: 11.428958 Objective: 5195009925435386575249799808l161227863139038901056258893741999176607491686l4.000000   TestRMSE: 683525349844142
640118390540331909l2.000000
Iteration: 5    Time: 14.813233 Objective: 2188729706741136163878591325714597005903192508601082047023367090636843747224230683678241531105155786446956861054082635746268835911366713062716019367052749637106999365020302026512673449663698523520589857328835664244102922240.000000 TestRMSE: 661887532119253848128774118569879852183068741501027648603949412404971274611759794754605785364638390147874816.000000
Latent 4, regularization 0, average error is: 1234933905130418524594015384431999426810813019140498722692760436573862985325091498057307126389656332613451776.000000
```

Because of small and constant learning rate, it converges very slowly.

--gain 0.0001 --pow 1.0 --maxiter 5

```
Iteration: 1    Time: 1.797256  Objective: 1013406.331867       TestRMSE: 165.467171
Iteration: 2    Time: 5.374592  Objective: 1010664.098955       TestRMSE: 165.463864
Iteration: 3    Time: 9.070005  Objective: 1009317.129034       TestRMSE: 165.453203
Iteration: 4    Time: 12.596313 Objective: 1008410.193909       TestRMSE: 165.442764
Iteration: 5    Time: 15.780084 Objective: 1007720.662149       TestRMSE: 165.433156
Iteration: 6    Time: 19.033258 Objective: 1007161.333454       TestRMSE: 165.424353
Iteration: 7    Time: 22.474347 Objective: 1006688.979356       TestRMSE: 165.416245
Iteration: 8    Time: 25.869146 Objective: 1006278.981426       TestRMSE: 165.408726
Iteration: 9    Time: 29.121583 Objective: 1005915.964738       TestRMSE: 165.401710
Iteration: 10   Time: 32.425149 Objective: 1005589.671713       TestRMSE: 165.395127
Iteration: 11   Time: 35.812656 Objective: 1005292.910156       TestRMSE: 165.388918
Iteration: 12   Time: 39.320272 Objective: 1005020.437703       TestRMSE: 165.383038
Iteration: 13   Time: 42.554505 Objective: 1004768.312337       TestRMSE: 165.377449
Iteration: 14   Time: 46.000265 Objective: 1004533.493091       TestRMSE: 165.372118
Iteration: 15   Time: 49.216399 Objective: 1004313.583448       TestRMSE: 165.367019
Iteration: 16   Time: 52.451824 Objective: 1004106.660237       TestRMSE: 165.362130
Iteration: 17   Time: 55.668734 Objective: 1003911.155930       TestRMSE: 165.357431
Iteration: 18   Time: 59.002831 Objective: 1003725.775495       TestRMSE: 165.352905
Iteration: 19   Time: 62.320994 Objective: 1003549.436279       TestRMSE: 165.348538
Iteration: 20   Time: 65.534573 Objective: 1003381.223651       TestRMSE: 165.344318
```

5e.

The result shows that without any regularization, we can obtain the best average RMSE by cross-validation.

```
Best Latent 4, Best Regul 0, RMSE 108.866230
```

```
df = pd.read_csv("A4_plotCSV_2", names=["Latent", "Regul", "Avg_RMSE"])
plt.plot(df.Regul, df.Avg_RMSE, c='r')
plt.xlabel('Regularization')
plt.ylabel('RMSE')
plt.title('RMSE vs. Regularization')
plt.legend()
plt.show()
```