EECE 5698 Assignment 4: Matrix Factorization

Zifeng Wang

Problem 1

1(a)

We can calculate the gradients as below

$$\begin{aligned}\nabla_{u_i} \operatorname{RSE}(U, V) &= \nabla_{u_i}\Big(\sum_{(i,j,r_{ij})\in\mathcal{D}} \left(u_i^\top v_j - r_{ij}\right)^2 + \lambda \sum_{i=1}^{n} \|u_i\|_2^2 + \mu \sum_{j=1}^{n} \|v_j\|_2^2\Big) \\ &= \sum_{j=1}^{m} 2\left(u_i^\top v_j - r_{ij}\right) v_j + 2\lambda u_i\end{aligned}$$

$$\begin{aligned}\nabla_{v_j} \operatorname{RSE}(U, V) &= \nabla_{v_j}\Big(\sum_{(i,j,r_{ij})\in\mathcal{D}} \left(u_i^\top v_j - r_{ij}\right)^2 + \lambda \sum_{i=1}^{n} \|u_i\|_2^2 + \mu \sum_{j=1}^{n} \|v_j\|_2^2\Big) \\ &= \sum_{i=1}^{n} 2\left(u_i^\top v_j - r_{ij}\right) u_i + 2\mu v_j\end{aligned}$$

1(b)

$$\begin{aligned}\nabla^2 \ell(u, v) &= \nabla^2\big(\left(u^\top v - r\right)^2\big) \\ &= \nabla^2((uv - r)^2) \\ &= \begin{bmatrix} 2v^2 & 2(2uv - r) \\ 2(2uv - r) & 2u^2 \end{bmatrix}\end{aligned}$$

$$\nabla^2 \ell(0, 0) = \begin{bmatrix} 0 & -2r \\ -2r & 0 \end{bmatrix}$$

It's eigenvalues are $2r, -2r$, due to the noise, $r \neq 0$, so the eigenvalues could not be all non-negative. So $\nabla^2 \ell(0, 0)$ is not positive semi-definite. Thus $\ell(u, v)$ is not convex.

1(c)

No. Because RSE is not convex. Gradient equals to zero doesn't mean global minimum.

1(d) Derive from results in 1(a)

$$
\begin{aligned}
\nabla_{u_i} \mathrm{RSE}(U,V) &= \sum_{j=1}^{m} 2\left(u_i^\top v_j - r_{ij}\right) v_j + 2\lambda u_i \\
&= \sum_{j=1}^{m} 2\left(0^\top 0 - r_{ij}\right) 0 + 2\lambda 0 \\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
\nabla_{v_j} \mathrm{RSE}(U,V) &= \sum_{i=1}^{n} 2\left(u_i^\top v_j - r_{ij}\right) u_i + 2\mu v_j \\
&= \sum_{i=1}^{n} 2\left(0^\top 0 - r_{ij}\right) 0 + 2\mu 0 \\
&= 0
\end{aligned}
$$

Problem 2

2(a)

$$
\begin{aligned}
\mathrm{RSE}(U^k, V^K) &= \sum_{(i,j,r_{ij})\in\mathcal{D}} \left({u_i^k}^\top v_j^k - r_{ij}^k\right)^2 + \lambda \sum_{i=1}^{n} \left\|u_i^k\right\|_2^2 + \mu \sum_{j=1}^{n} \left\|v_j^k\right\|_2^2 \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} {\delta_{ij}^k}^2 + \lambda \sum_{i=1}^{n} \left\|u_i^k\right\|_2^2 + \mu \sum_{j=1}^{n} \left\|v_j^k\right\|_2^2
\end{aligned}
$$

$$
\begin{aligned}
\nabla_{u_i} \mathrm{RSE}(U^k, V^k) &= \sum_{j=1}^{m} 2\left({u_i^k}^\top v_j^k - r_{ij}^k\right) v_j^k + 2\lambda u_i^k \\
&= \sum_{j=1}^{m} 2\delta_{ij}^k v_j^k + 2\lambda u_i^k
\end{aligned}
$$

$$
\begin{aligned}
\nabla_{v_j} \mathrm{RSE}(U^k, V^k) &= \sum_{i=1}^{n} 2\left({u_i^k}^\top v_j^k - r_{ij}^k\right) u_i^k + 2\mu v_j^k \\
&= \sum_{i=1}^{n} 2\delta_{ij}^k u_i^k + 2\mu v_j^k
\end{aligned}
$$

2(b) The implementation is as follow:

```python
def predict(u,v):
    """ Given a user profile uprof and an item profile vprof, predict the rating given by the user to this item

        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 np.dot(u, v)

def pred_diff(r,u,v):
    """ Given a rating, a user profile u and an item profile v, compute the difference between the prediction and actual rating

        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
          - δ =  <u,v> - r
    """
    return np.dot(u, v) - r
```

```python
def gradient_u(delta,u,v):
    """ Given a user profile u and an item profile v, and the difference in rating predictions δ, compute the gradient

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

             of the square error loss:

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

        Inputs are:
           -δ: 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
```

```python
def gradient_v(delta,u,v):
    """ Given a user profile u and an item profile v, and the difference in rating predictions δ, compute the gradient

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

        of the square error loss:

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

        Inputs are:
           -δ: 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
```

2(c)

```python
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):i).distinct(numPartitions = N)
    numUsers = V.count()
    randRDD = RandomRDDs.normalVectorRDD(sparkContext, numUsers, d,numPartitions=N, seed=seed)
    V = V.zipWithIndex().map(swap)
    randRDD = randRDD.zipWithIndex().map(swap)
    return V.join(randRDD,numPartitions = N).values()
```

We initialized the user and item profiles to random values instead of zero because we want to get rid of stuck at saddle point. If we initialize all profiles to be zero vectors, the RSE and its gradients will stay zero no matter how many iterations go, we will not get closer to our optimal solution.

Problem 4

4(a)

```python
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,δij,ui,vj)

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

    """
    U_V = U.cartesian(V).map(lambda (i_u, j_v): ((i_u[0], j_v[0]), (i_u[1], j_v[1])))
    R = R.map(lambda (i, j, rij) : ((i, j), rij))
    joined = R.join(U_V, numPartitions = N).map(lambda (ij, v) : (ij[0], ij[1], pred_diff(v[0],v[1][0],v[1][1]), v[1][0], v[1][1]))
    return joined
```

4(b)

```python
def SE(joinedRDD):
    """ Receives as input a joined RDD as well as a λ and a μ and computes the MSE:

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

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

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

```python
def normSqRDD(profileRDD,param):
    """ Receives as input an RDD of profiles (e.g., U) as well as a parameter (e.g., λ) and computes the square of norms:
        λ Σ_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 λ>0

        The return value is:
        λ Σ_i ||ui||_2^2
    """

    sum_u_norm = profileRDD.map(lambda (i, u): np.dot(u, u)).reduce(add)
    return param * sum_u_norm
```

In __main__, they are used as below to calculate the regularized objective function.

```python
obj = SE(joinedRDD) + normSqRDD(U,args.lam) + normSqRDD(V,args.lam)
```

4(c)

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

        ui = ui - γ  ∇_ui RegSE(R,U,V)

        where

        RegSE(R,U,V) = Σ_{i,j in R} (<ui,vj>-rij)^2 + λ Σ_i ||ui||_2^2 + μ Σ_j ||vj||_2^2


        Inputs are
         -joinedRDD: an RDD with tuples of the form (i,j,δij,ui,vj), where δij = <ui,vj> - rij
         -gamma: the gain γ
         -lam: the regularization parameter λ
         -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.
    """
    grad_u = joinedRDD.map(lambda (i,j,sig,ui,vj): (i, gradient_u(sig,ui,vj))).reduceByKey(add, numPartitions=N)
    U = joinedRDD.map(lambda (i,j,sig,ui,vj): (i, ui)).reduceByKey(lambda x, y: (x+y)/2, numPartitions=N)
    new_U = U.join(grad_u, numPartitions=N).mapValues(lambda (ui, grad_u): ui-gamma*(grad_u+2*lam*ui))
    return new_U
```

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

        ui = ui - γ  ∇_ui RegSE(R,U,V)

        where

        RegSE(R,U,V) = Σ_{i,j in R} (<ui,vj>-rij)^2 + λ Σ_i ||ui||_2^2 + μ Σ_j ||vj||_2^2


        Inputs are
         -joinedRDD: an RDD with tuples of the form (i,j,δij,ui,vj), where δij = <ui,vj> - rij
         -gamma: the gain γ
         -mu: the regularization parameter μ
         -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.
    """
    grad_v = joinedRDD.map(lambda (i,j,sig,ui,vj): (j, gradient_v(sig,ui,vj))).reduceByKey(add, numPartitions=N)
    V = joinedRDD.map(lambda (i,j,sig,ui,vj): (j, vj)).reduceByKey(lambda x, y: (x+y)/2, numPartitions=N)
    new_V = V.join(grad_v, numPartitions=N).mapValues(lambda (vj, grad_v): vj-gamma*(grad_v+2*mu*vj))
    return new_V
```

Problem 5

5(a)

This part is for reading data for each fold if **`args.folds`** is specified, otherwise, read all data in a single fold.

```python
folds = {}

if args.output is None:
    for k in range(args.folds):
        folds[k] = readRatings(args.data+"/fold"+str(k),sc)
else:
    folds[0] = readRatings(args.data,sc)
```

This part constructs the training and testing set for each code.

```python
train_folds = [folds[j] for j in folds if j is not k ]

if len(train_folds)>0:
    train = train_folds[0]
    for fold in  train_folds[1:]:
        train=train.union(fold)
    train.repartition(args.N).cache()
    test = folds[k].repartition(args.N).cache()
    Mtrain=train.count()
    Mtest=test.count()

    print("Initiating fold %d with %d train samples and %d test samples" % (k,Mtrain,Mtest) )
else:
    train = folds[k].repartition(args.N).cache()
    test = train
    Mtrain=train.count()
    Mtest=test.count()
    print("Running single training over training set with %d train samples. Test RMSE computes RMSE on training set" % Mtrain )
```

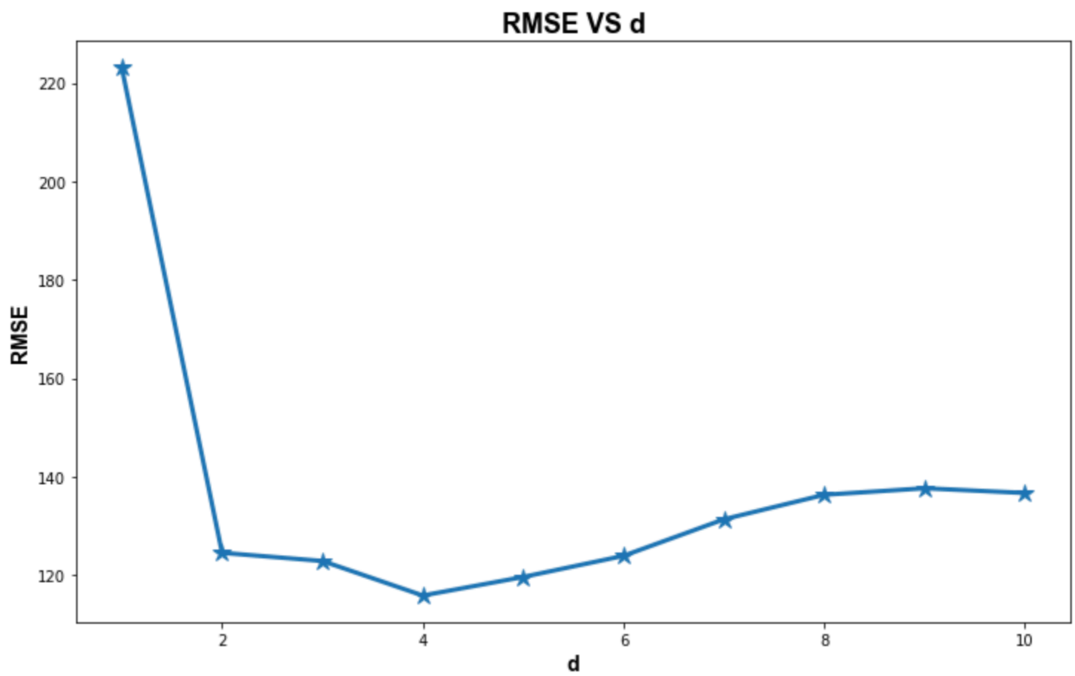
Explaination of what each operation does is in the comments in red below:

```
# specify training folds as all folds except the test fold
train_folds = [folds[j] for j in folds if j is not k ]
if len(train_folds)>0:
    # the next 3 lines merge the folds inside train_folds into one
single list
    train = train_folds[0]
    for fold in  train_folds[1:]:
        train=train.union(fold)
    # repartition the train data and cache it
    train.repartition(args.N).cache()
    # repartition the train data and cache it
    test = folds[k].repartition(args.N).cache()
    # calculate number of training samples
    Mtrain=train.count()
    # calculate number of test samples
    Mtest=test.count()
    # print info
    print("Initiating fold %d with %d train samples and %d test
samples" % (k,Mtrain,Mtest) )
else:
    # use all data as training set and repartition & cache it
    train = folds[k].repartition(args.N).cache()
    # use training set as test set
    test = train
    # calculate number of training samples
    Mtrain=train.count()
    # calculate number of test samples
    Mtest=test.count()
    # print info
    print("Running single training over training set with %d train
samples. Test RMSE computes RMSE on training set" % Mtrain )
```

5(b) step size is set as below, i is the number of iteration, **`args.gain`** and **`args.power`** are user specified parameters:

```
gamma = args.gain / i**args.power
```

5(c) Plot the RMSE w.r.t. d as below:



We can see that b = 4 is the optimal value

5(d)

For **`--gain 0.1 --pow 0.0 --maxiter 5`**, we have the observation below

```
Iteration: 1    Time: 9.074272  Objective: 1013500.042176   TestRMSE: 165.467171
Iteration: 2    Time: 20.203857 Objective: 1923873332.578143    TestRMSE: 8309.034806
Iteration: 3    Time: 30.992711 Objective: 313778766604297948233728.000000 TestRMSE: 237055796220.879547
Iteration: 4    Time: 42.134367 Objective: 3150685195168790513492671456715833560386074221357824092681809681396531 2.000000   TestRMSE: 89057963239239285497093242161725 44.000000
Iteration: 5    Time: 52.153823 Objective:
441117766146899700022882063213317902977853435020712318016277617757962366893939561331282586665009631749594721796654567048124778888591540742279500909207131119584134440495140133027181490420645404248757598072 0
259072.000000 TestRMSE: 647479734501589103123742685416032237194709102261733344799834268784317605303536830854251197230519156736 0.000000
Initiating fold 1 with 40 train samples and 10 test samples
Training set contains 10 users and 10 items
Iteration: 1    Time: 8.352871  Objective: 934540.007357    TestRMSE: 187.817433
Iteration: 2    Time: 18.836408 Objective: 2025824149.155913    TestRMSE: 5114.998322
Iteration: 3    Time: 30.346551 Objective: 760917655932644697833472 0.000000 TestRMSE: 92883984755.931259
Iteration: 4    Time: 40.665033 Objective: 643519204970746786523486264267281499590551982001203286696283228006776832.000000  TestRMSE: 1844953462334471705429503534694400 0.000000
Iteration: 5    Time: 51.119073 Objective:
411058600748409639819099388828376717444569768332208557527124714290689831978914870974543988990656478550648918000066569058297616183806627202320981074380252016485845210179451243076852664497225766928687844 00
3906846720.000000 TestRMSE: 346909988383956470957912460851437146070401703435586043333816324662343417879708829296026842928459806670848.000000
Initiating fold 2 with 40 train samples and 10 test samples
Training set contains 10 users and 10 items
Iteration: 1    Time: 8.361678  Objective: 1135618.639875   TestRMSE: 123.156506
Iteration: 2    Time: 19.509996 Objective: 2594881440.592070    TestRMSE: 6211.310434
Iteration: 3    Time: 30.373434 Objective: 946313925860785276741222 4.000000 TestRMSE: 109927540214.046951
Iteration: 4    Time: 41.694653 Objective: 870112131350369054872560550861952904857232391701940177941329561123815424.000000  TestRMSE: 1742619732197773759144526854121062 4.000000
Iteration: 5    Time: 52.736168 Objective:
930550704333429817631273193023319739539684295026117391612497319272412422732014276508587961828632834869787494773446119106859996677876548055785721363194033851384785996934925695383550538344914678127543862 83
2197009408.000000 TestRMSE: 267057601896676384009176661582435831817012882486044892648550415808410556283935027958663309538302925733888.000000
Initiating fold 3 with 40 train samples and 10 test samples
Training set contains 10 users and 10 items
Iteration: 1    Time: 8.356851  Objective: 1007655.537988   TestRMSE: 167.223907
Iteration: 2    Time: 18.974245 Objective: 5111830477.907906    TestRMSE: 8403.427325
Iteration: 3    Time: 29.763850 Objective: 1084505163823511199012618 24.000000   TestRMSE: 568305429493.300415
Iteration: 4    Time: 40.451817 Objective: 17401876042254243731220040136706345177656148004940064070636066487004875 98080.000000  TestRMSE: 121396762312757194534570201351979008 0.000000
Iteration: 5    Time: 51.705304 Objective:
806116425814361195090724626344250381270366580996403095511436697094108327009122971110442426314339948053864692972047233564175040131803959999532587626750891470873974844084205429871170047370265072641450167840
33749936217262129152.000000   TestRMSE: 531897030278947679431312571371944175848358057137793717832864984835549650549041648488417678259124776052153384 96.000000
Initiating fold 4 with 40 train samples and 10 test samples
Training set contains 10 users and 10 items
Iteration: 1    Time: 9.126647  Objective: 1057955.040949   TestRMSE: 151.439377
Iteration: 2    Time: 19.999344 Objective: 5019020205.414135    TestRMSE: 1720.935899
Iteration: 3    Time: 30.767982 Objective: 1083297297436629757235036 16.000000   TestRMSE: 56454004136.552391
Iteration: 4    Time: 42.049717 Objective: 17637932733210290493656315471147043882621790502157049600300679544204238 52032.000000  TestRMSE: 580103058500592722402955533892976 64.000000
Iteration: 5    Time: 53.351197 Objective:
839682832426387617009851230804780353198534879962365736471317305433148861501797224108816547945471265367054688549102390931206995462702845157850732678844740644261908676214930853645368391986546029802853379 64
6467185273314109030 4.000000   TestRMSE: 126074452646853528135059896637891025025972270631614787363819297297458111399878482511071949041458413286981632 0.000000
5-fold cross validation error is: 108902135993501844158896594785683023339005507160964376223227172546982239817692881841218509971705835857190584 32.000000
```

We can see that the step size is too big, so the RMSE is 'exploding'.

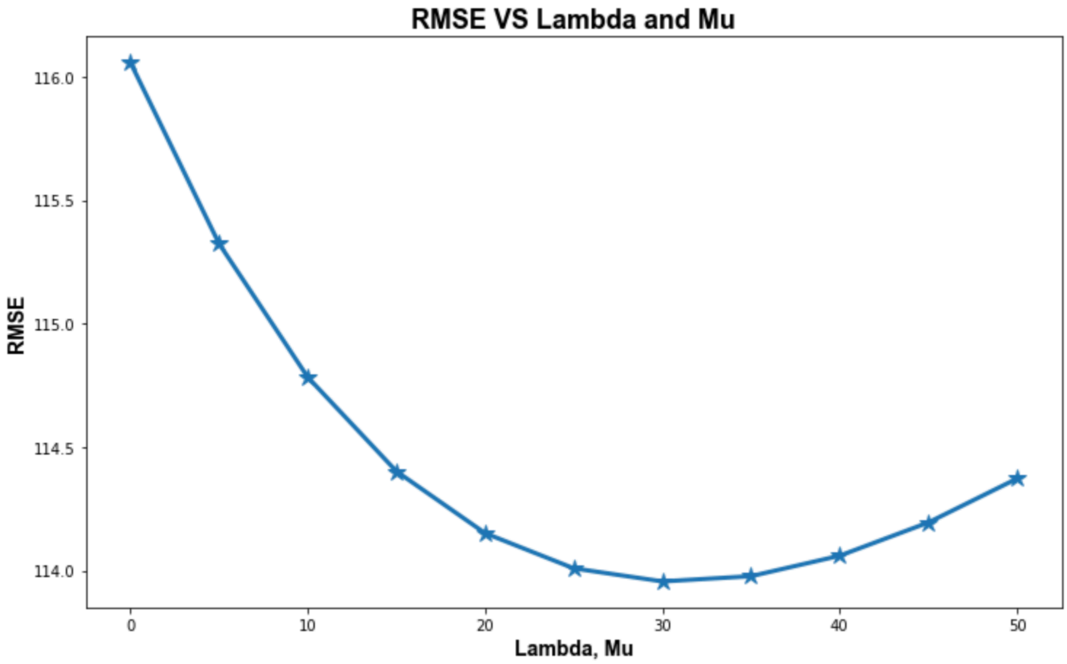
For **`--gain 0.0001 --pow 1.0 --maxiter 20`**, we have the observation below (for simplicity I just capture screen for 1 fold, the rest are similar):

```
Training set contains 10 users and 10 items
Iteration: 1    Time: 9.031763  Objective: 1013500.042176   TestRMSE: 165.467171
Iteration: 2    Time: 20.364278 Objective: 1010755.181194   TestRMSE: 165.464027
Iteration: 3    Time: 31.279617 Objective: 1009408.319779   TestRMSE: 165.453452
Iteration: 4    Time: 43.331643 Objective: 1008501.947583   TestRMSE: 165.443073
Iteration: 5    Time: 53.420159 Objective: 1007813.094133   TestRMSE: 165.433514
Iteration: 6    Time: 63.803859 Objective: 1007254.467580   TestRMSE: 165.424752
Iteration: 7    Time: 76.152883 Objective: 1006782.808386   TestRMSE: 165.416680
Iteration: 8    Time: 86.259500 Objective: 1006373.486731   TestRMSE: 165.409194
Iteration: 9    Time: 97.272816 Objective: 1006011.123790   TestRMSE: 165.402207
Iteration: 10   Time: 107.417636    Objective: 1005685.461192    TestRMSE: 165.395651
Iteration: 11   Time: 117.996144    Objective: 1005389.307289    TestRMSE: 165.389468
Iteration: 12   Time: 129.741703    Objective: 1005117.420821    TestRMSE: 165.383611
Iteration: 13   Time: 139.845720    Objective: 1004865.861072    TestRMSE: 165.378044
Iteration: 14   Time: 150.207647    Objective: 1004631.588413    TestRMSE: 165.372734
Iteration: 15   Time: 160.763873    Objective: 1004412.207617    TestRMSE: 165.367655
Iteration: 16   Time: 170.849454    Objective: 1004205.796721    TestRMSE: 165.362785
Iteration: 17   Time: 181.421509    Objective: 1004010.789316    TestRMSE: 165.358104
Iteration: 18   Time: 192.244272    Objective: 1003825.891393    TestRMSE: 165.353595
Iteration: 19   Time: 202.592768    Objective: 1003650.021234    TestRMSE: 165.349245
Iteration: 20   Time: 214.696185    Objective: 1003482.265060    TestRMSE: 165.345041
```

We can see that the step size is too small, so the optimization process is too slow.

5(e)

Here as the searching space is continuous and so big, I just let $\lambda = \mu = 0:5:50$ to search for the optimal value. However, a finer search may be conducted further around the optimal value I find.



From the figure, we can see that $\lambda = \mu = 30$ is approximately the optimal solution.

Bonus:
The method is similar as above.

- Set $\lambda = \mu = 0$, and set d = 1:10 to evaluate the least RMSE. If we can observe a first decreasing and then increasing curve, we take the minimum value of d, either do finer search or fix that value. Otherwise, give d a wider range to search.
- For investigating λ and μ, I use a greedy search. I search λ in range [1,50] first, find the optimal value for λ. Then fix the optimal value for λ, search μ in range [1,50]
- As it is too time consuming to do all this experiments, I just offer the general method here, more work needs to be done to find the exact optimal triplet.