Q1:

a)

$$\nabla_{u_i} RSE(U,V) = 2\left(u_i{}^T v_j - r_{ij}\right)v_j + 2\,\lambda\, u_i$$

$$\nabla_{v_j} RSE(U,V) = 2\left(u_i{}^T v_j - r_{ij}\right)u_i + 2\mu v_j$$

b)

$$l(u,v) = (u^T v - r)^2$$

$$\nabla l(u,v) = 2(uv - r)\begin{bmatrix} u \\ v \end{bmatrix}$$

$$\nabla^2 l(u,v) = \begin{bmatrix} 2v^2 & 4uv - 2r \\ 4uv - 2r & 2u^2 \end{bmatrix}$$

Suppose there's a vector $x \in R$ and at u=v=0 we have

$$\mathrm{x}^T \nabla^2 l(u,v) x = [x_1 \quad x_2]\begin{bmatrix} 0 & -2r \\ -2r & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$= -4r x_1 x_2$$

We don't know if it's greater or equal than 0. So l(u ,v) is not convex.

c)

If $\nabla_{u_i} RSE(U,V) = \nabla_{v_j} RSE(U,V) = 0$

For all i and j, u^* and v^* would minimize the RSE but since RSE is not convex, u^* and v^* can only be the local minima instead of global minima.

d)

if $u_i = v_j = 0$ then

$$\nabla_{u_i} RSE(U,V) = 2\left(u_i{}^T v_j - r_{ij}\right)v_j + 2\,\lambda\, u_i = 0$$

$$\nabla_{v_j} RSE(U,V) = 2\left(u_i{}^T v_j - r_{ij}\right)u_i + 2\mu v_j = 0$$

Q2:

a)

$$RSE(U^k, V^k) = \sum_{\delta_{ij} \in D} (\delta_{ij}{}^k)^2 + \lambda \sum_{i=1}^{n} ||u_i{}^k||_2{}^2 + \mu \sum_{i=1}^{n} ||v_j{}^k||_2{}^2$$

$$\nabla_{u_i} RSE(U^k, V^k) = 2\delta_{ij}{}^k v_j{}^k + 2\lambda u_i{}^k$$

$$\nabla_{v_j} RSE(U^k, V^k) = 2\delta_{ij}{}^k u_i{}^k + 2\mu v_j{}^k$$

b)

```
def swap((x,y)):
    """ Swap the elements of a pair tuple.
    """
    return (y,x)

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 predict(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*1.0



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*1.0
```

c)

```
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()
```

Since RSE is not convex so gradient descent would converge to a local minimum so the initial point can decide which local minimum to converge to. If we set all profiles to be zero. The gradient descent also converge to zero since zero is a stationary point so we would have no other information.

Q3:

a)

```
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 (i,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.

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

b)

```
def SE(joinedRDD):
    """ Receives as input a joined RDD 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,delta,ui,vj):delta*delta*1.0)\
                  .reduce(lambda x,y: x + y)
    return SE

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
    """
    normSq = profileRDD.map(lambda (i,ui): np.dot(ui,ui))\
                       .reduce(lambda x,y : x + y)
    return param * normSq * 1.0
```

c)

```
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.
    """
    adaptU = joinedRDD.map(lambda (i,j,delta,ui,vj) : (i,(ui,gradient_u(delta,ui,vj))))\
                .reduceByKey(lambda x,y :(x[0],x[1]+y[1]),numPartitions=N)\
                .mapValues(lambda (ui,gradDelta): ui - gamma * (gradDelta + 2* lam * ui * 1.0))
    return adaptU
```

```
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 (i,vi). The returned rdd contains exactly N partitions.
    """

    adaptV = joinedRDD.map(lambda (i,j,delta,ui,vj): (j,(vj,gradient_v(delta,ui,vj))))\
                .reduceByKey(lambda x,y:(x[0],x[1]+y[1]),numPartitions=N)\
                .mapValues(lambda (vj,grad): vj-gamma*(grad+2*mu*vj))
    return adaptV
```

Q4:

a)

```
cross_val_rmses = []
for k in folds:
    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 )
```

In the code, we can see that k-th fold can construct a training list of the other folds. Then we split the data into training RDD and testing RDD in N partitions. If the training list is empty, the training RDD and testing RDD would be the same. And after all we print the amount if data in training RDD and test RDD separately

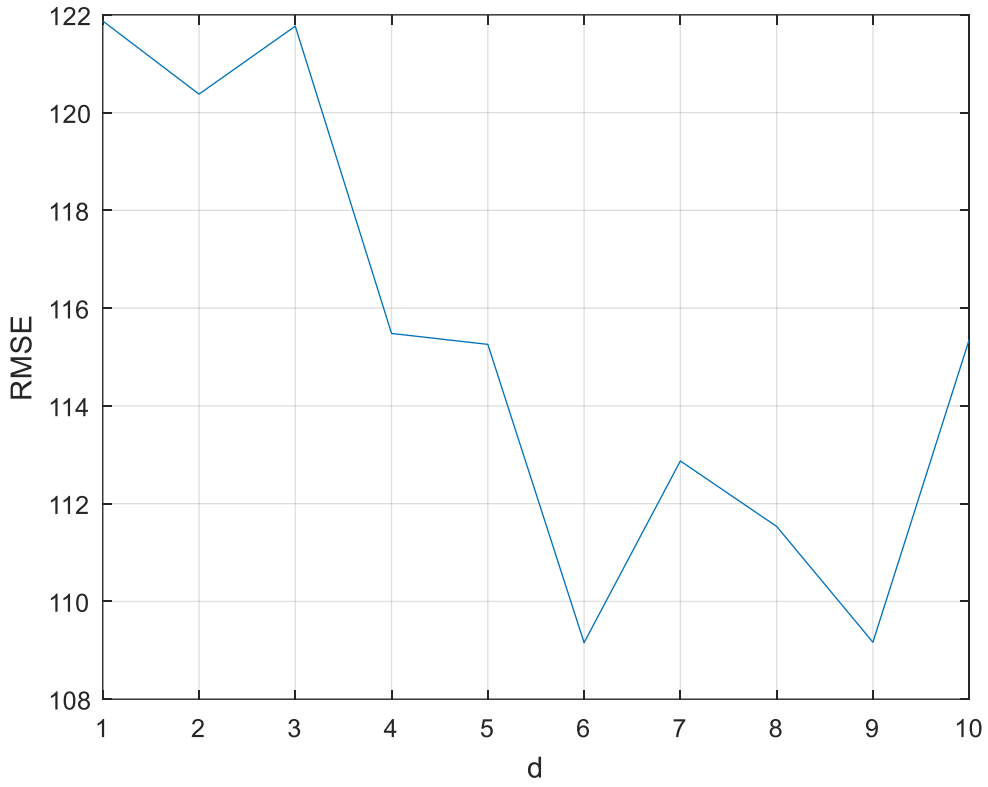
b)

The step size is defined as

$$\gamma_k = \frac{G}{k^p}$$

G is the gain and k p is gain exponent.

c)



When d is 6 we get the best result.

d)

gain = 0.1 power = 0.0

$$\gamma_k = \frac{G}{k^p} = 0.1$$

This is a constant step size which is much larger when gain is 0.001 and power is 0.2. It won't converge at the end and the RMSP is huge

```
5-fold cross validation error is: 13066647942735828644276216062948349669559835179285590711103286127927630015614586666758273001409088003107216622092288.000000
```

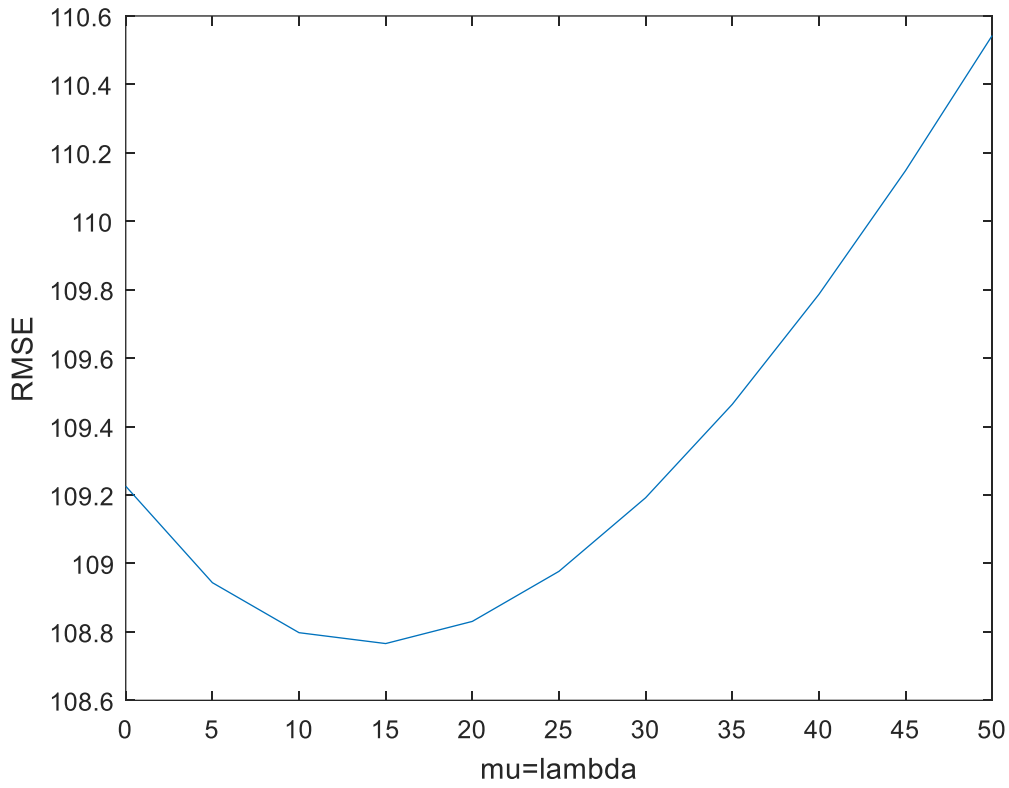
gain = 0.001 power = 1.0

$$\gamma_k = \frac{G}{k^p} = \frac{0.001}{k}$$

The step size is smaller than the step size when gain is 0.001 and power is 0.2. So It converges very slow. With 20 steps, it doesn't reduce the error significantly.

```
5-fold cross validation error is: 134.810291
```

e)


110.6
110.4
110.2
110
109.8
109.6
109.4
109.2
109
108.8
108.6
RMSE
0
5
10
15
20
25
30
35
40
45
50
mu=lambda