Part II, Question 1:

Assumptions: M & N are two matrices with dimensions i,j and j,k respectively and the dimensions of the 2 matrices are accessible globally.
Input Format: (matrix_name, ((i, j, v))

```
#Reading from DFS
records = sc.textFile("dfs:...") //sc is spark context

def mapping(k,v):
    #initialize an empty list
    l=list ()
    if k=='M':
        #making combination of M row with all N cols
        for col in range (0, k):
            #appending the pair in list l
            l.append((v[0], col),('M',v[1],v[2]))  #((i,k),('M',j,v))
    else:
        #making combination of N column with all M rows
        for row in range (0, i):
            #appending the pair in list l
            l.append((row,[1]), ('N',v[1],v[2]))  #((k,j),('N',j,v))
    return l #return the list of all mappings

def combine(k,v):
    for keys in k:
        listM=list()
        listN=list()
        if(v[0][1]=='M'):
            listM.append((v[1],v[2])) #separating values in M
        else:
            listN.append((v[1],v[2])) #separating values in N

    sorted (listM, key=lambda x: x[0]) #sorting the list by j value in M
    sorted (listN, key=lambda x: x[0]) #sorting the list by j value in N
    i=0
    ans=0
    while i<len(listN):
        ans=ans+listN[i][1]*listM[i][1] #adding element wise product of the lists M,N
        i=i+1
    return ((k[0],k[1],ans));  #returning in the form (row,col,value)

#spark transformation starts here
res=records.flatMap(lambda x: mapping(k,v)) #used flatmap as we want to return all the combinations
.groupByKey() #grouping the result by Key-> we will get all the i,j elements together.
.map(lambda k,v: combine(k,v)) #doing list element wise multiplication and addition(sum of product)
```

Part II, Question 2:

Main idea: Everything remains the same as demo code, only the training operation is changed with the introduction of momentum optimization.

#momentum vector is introduced and initialized to zero. Initialized to zero as the acceleration in the beginning is zero and gradually increases/decreases speed later.
#momentum vector acts as acceleration so when the local minima is far, acceleration becomes high and when the local minima is close, the acceleration reduces.
momentum_vector = tf.zeros(shape=[featuresZ_pBias.shape[1], dtype="float32")

#specify the operation to take for each epoch of training: [below equations were taken from HOML, Geron, 2017]
m<- βm + ηΔJ(θ) -> update the momentum vector by adding learning rate*grads, this acts like acceleration to find the local minima to minimize the error. [β=momentum, m=momentum vector, η=learning rate, J(θ)=cost function]
#**θ<- θ – m** -> update the weights by subtracting the new momentum vector, this finds the next better(adjusts) weights to minimize the error.
momentum_vector = momentum*momentum_vector + learning_rate*grads
training_op = tf.assign(beta, beta-momentum_vector)

#β initialized to 0.9 as it's a typical momentum value [Reference: HOML, Geron, 2017]
momentum= 0.9

Part II, Question 3(a):

Jaccard similarity: It is defined as the intersection of the two sets over the union.

Min Hashing: It is a technique where instead of looking for the similar shingles, we compare the first occurrence of any element in that set (helps in reducing the size of matrix).

If Jaccard similarity is zero => S1 ∩ S2=0 which implies that neither of the sets had any common element.

Jaccard similarity = 0 as there is no row which has all 1s (implying the element was present in both S1 and S2.)

There are 3 possibilities that can happen-

1. Type X rows have 1 in both columns.
2. Type Y rows have 1 in one of the columns and 0 in the other.
3. Type Z rows have 0 in both columns

Jaccard Similarity: Sim (S1, S2) =x/(x+y) – (1)

We check in S1, S2, the row number where the first 1 appeared. The probability that we shall meet a type X row before we meet a type Y row is x/ (x + y). if we see a Type X row (where the row is all 1s) then we can say that h(S1) =h(S2). However, if we see the a row(other than type Z) of type Y then h(S1)≠h(S2) which implies that probability that h(S1)=h(S2) =x/(x+y) which is same as Jaccard Similarity given in eq. (1). [Reference]

Part II, Question 3(b):

Assumptions:
Each record is in the format (set_name, (elem1, elem2, ...))
There are 500 Hash Functions chosen from h_0 to h_{499}

Main Idea: For each element in a set compute 500 hash values and take the minimum hash value.

```
#Reading record from hdfs
records = sc.textFile("hdfs:...")

def map(k,v):
        l=list ()
        for element in v:
                minimum=int('inf') #setting minimum to be maximum value
                hash_i=int('inf')
                for I in range (0,500):
                        if h_i(element) < minimum:
                                hash_i=i
                                minimum= h_i(element)
```

```
                l.append((hash_i , k), minimum)
        return l
```

#output after mapping will be as ((hash$_i$, S), minVal), this gives the signature matrix using minhashing in an efficient manner.

```
records.flatMap(lambda k,v: map(k,v))
```

Part II, Question 4(a):

The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is **$P= 1 – (1 – s^r)^b$** [Reference : mmds]

Given:
r=5 (band size)
b=number of bands
sample size=500
Jaccard Similarity=0.75

Sample size=r*b
500=5*b

⇨ b=100

$P= 1 – (1 – s^r)^b$
$= 1 – (1 – 0.75^5)^{100}$
$= 1 – (1 - 0.2373046875)^{100}$
$= 1- (0.7626953125)^{100}$
$= 1 – (1.7183345553283 * 10^{-12})$ [which is very close to 1]

Part II, Question 4(b):

Given:
Sample size=500
Jaccard similarity(s)=90%

Sample size=b*r
500=b*r
500/b=r ----- (1)

Probability that 2 sets matched in at least 1 band-
$P= 1 – (1 – s^r)^b$
$0.99= 1 – (1-s^{500/b})^b$ -------(2)

As per the textbook [HOML, Geron, 2017], the threshold t is given by the formula-
$(1/b)^{(1/r)}$ -------- (3)
The above value must be increased to decrease the false positives.

With eq. (3) in mind, r and b must be chosen. With b=28,r=18, the probability that 2 sets matched in at least 1 band is given by-
$P= 1 – (1 – s^r)^b$
$= 1 – (1 – 0.9^{18})^{28}$

$= 1 – (0.8499053647)^{28}$
$= 1 - 0.010528729687194$
$= 0.98947127031$ (which is close to 0.99 value we wish to obtain)

With b=28, r=18, the threshold $t= (1/28)^{(1/18)}$
$= 0.83100250243647$

We can now calculate the False Positive rate from the given research paper which comes out to be 3.6% which is lower than 25% as stated in the question.

Part II, Question 4(c):

We need to speed up the LSH approach in 3(b) by allowing a greater number of false positives.

Main idea: Instead of calculating the Hash values for all the elements in the set, we will choose a number m smaller than total elements in the set which will help in speeding up the process. Also, instead of 500 Hash functions, we can limit the number of hash functions to around half, say 300 which will also help in speeding up. The thing to note with this approach is that this may result in increased number of false positives.

```
def map(k,v):
        l=list ()
        i=0
        while i < m: #iterating over only m elements
                i=i+1
                minimum=int('inf') #setting minimum to be maximum value
                hash_i=int('inf')
                for I in range (0,300):
                        if h_i(element) < minimum:
                                hash_i=i
                                minimum= h_i(element)
                l.append((hash_i , k), minimum)
        return l
```