Part II: Problem Solving

Author : Rishabh Goel (SBU ID : 112714848)

Q1.

Task : Multiply given matrices M and N using single pass matrix multiplication

Assumptions : size(M) = (I,J) ; size(N) = (J,K)

Algorithm : The Map Function: For each element m_{ij} of M, produce all the key-value pairs (i, k), (M, j, m_{ij}) for k = 1, 2, . . . up to the number of columns of N. Similarly, for each element n_{jk} of N, produce all the key-value pairs (i, k), (N, j, n_{jk}) for i = 1, 2, . . . up to the number of rows of M. M and N are really bits to tell which of the two matrices a value comes from.

The Reduce Function: Each key (i, k) will have an associated list with all the values (M, j, m_{ij}) and (N, j, n_{jk}), for all possible values of j. The Reduce function needs to connect the two values on the list that have the same value of j, for each j. An easy way to do this step is to sort by j the values that begin with M and sort by j the values that begin with N, in separate lists. The j^{th} values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied. Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

Spark Code:

// input to flatmap: t-> (matrix_name, ((i, j), v); t[0]-> matrix_name ; t[1] - > ((i,j),v) ; t[1][0] -> (i,j) ; t[1][0][0] -> i ; t[1][0][1] -> j ; t[1][1] -> v

```
rdd.flatmap(lambda t: (((t[1][0][0], k), (t[0], t[1][0][1], t[1][1]))
for k in range(K)) if t[0] == 'M'

else((i, t[1][0][1]), (t[0], t[1][0][0], t[1][1])) for i in range(I)))
```

// The Map function : for m_{ij} in 'M', produce (i, k), (M, j, m_{ij}) ; for n_{jk} in N produce (i, k), (N, j, n_{jk})

// output from flatmap: t-> ((i,k), (matrix_name, j, v); t[0]-> (i,k) ; t[1] - > (matrix_name,j,v)

```
.groupByKey() // grouping in the same list
```

// input to map: t-> ((i,k), [(matrix_name, j, v)]; t[0]-> (i,k) ; t[1] - > [(matrix_name,j,v)]

```
.map(lambda t: t, sum([t[1][i][2] * t[1][i + int(len(t[1]) / 2)][2]
    for i in range(int(len(sorted(t[1])) / 2))
  ]))
```

// The Reduce function : sort by matrix name and j , then product of first and second half of list, then sum of products

// output from map : ((i, k), v)

```
.collect()
```

Q2. + <code> => additions - <code> => subtractions

```
import tensorflow as tf
X = tf.constant(featuresZ_pBias, dtype=tf.float32, name="X")
y = tf.constant(price.reshape(-1,1), dtype=tf.float32, name="y")
############################################################
## Solve linreg as gradient descent problem:
## (partially follows HOML book, p. 237 - 239)

#first define learning parameters:
n_epochs = 10
learning_rate = 0.0001
+ alpha = 0.9 // hyperparameter for momentum
penalty = tf.constant(1.0, dtype=tf.float32, name="penalty")

#first define the parameters we are going to find optimal values
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.),
name = "beta")

+ momentum = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -
1., 1.), name = "momentum") // momentum vector

#then setup the prediction model's graph:
y_pred = tf.matmul(X, beta, name="predictions")

#Define the cost function for which we will calculate the gradients over beta
in order to minimize:
#                                    squared error     +      L2 regularization
penalizedCost = tf.reduce_sum(tf.square(y - y_pred)) + penalty *
tf.reduce_sum(tf.square(beta))

#add in gradient calculation
grads = tf.gradients(penalizedCost, [beta])[0]

#specify the operation to take for each epoch of training:
+ momentum_op = tf.assign(momentum, alpha*momentum + learning_rate*grads)
- training_op = tf.assign(beta, beta-learning_rate*grads)
+ training_op = tf.assign(beta, beta-momentum)


#initialize variables (i.e. beta)
init = tf.global_variables_initializer()


### everything above is just definitions of operations to carry out over
tensors
### now let's create a session to run the operations
with tf.Session() as sess:
 sess.run(init)
 for epoch in range(n_epochs):
  if epoch %10 == 0: #print debugging output
   print("Epoch", epoch, "; penalizedCost =", penalizedCost.eval())
  + sess.run(momentum_op)
  sess.run(training_op)
 #done training, get final beta:
 best_beta = beta.eval()
print(best_beta)
```

Q3. a) Let the two sets be S1 and S2, whose Jaccard similarity is 0. In minhashing, we estimate Jaccard similarity by creating a signature matrix using different hash functions and then taking the fraction of signatures in which the two sets agree. In the original characteristic matrix, there will be no rows which have 1 for both S1 and S2, since their Jaccard similarity is zero. Thus, we will always have different signatures for S1 and S2 for any particular hash function. We can safely say that h(S1) ≠ h(S2) for any hash function. Thus, the fraction of signatures in which the two sets agree will always be zero, making the Jaccard similarity estimate to be zero.

b) Approach 1 : Use m different hash functions (m is the length of signature matrix). For every hash function, do the following for every document key : hash all the words and get the minimum value of those hash values. This min hash value will go into the cell [hash_fn, doc] of the signature matrix. Assume hash_fns is an array of different hash functions.

```
rdd.flatmap( lambda t : ( (i,t[0]),min([hash_fns[i](t[1]) for i in
range(hash_fns)]) ))
```
// creating (hash_fn,doc), (min_hash)

```
.collect()
```

Approach 2 : If the number of words for every document is large, we can divide the words into batches and instead of using different hash functions for whole list of words, we use one hash function with different subset of words. We use sorting, so we have similar batch of words. We can experiment with different ways of selecting batches, one way is to just create batches of uniform size; another method is to create batches using prefixes. For eg, batch i defined by the first character of the words it contains (we can use either a 1 char prefix or 2 chars prefix, depending on size of our data)

Assume b-> no of batches

2.1 uniform length batches

```
rdd.flatmap( lambda t : ((i,t[0]),t[1][i*len(t[1])/b:(i+1)*len(t[1])/b] for i
in range(len(t[1]/b)))
```
// creating (batch_no,doc), (word sublist)

```
.map(lambda t : (t[0],min(hash_fn(t[1])))
```
// ((hash_fn,doc),min_hash)

```
.collect()
```

2.2 batches grouped by starting character

```
rdd.flatmap( lambda t : ((c,t[0]),t[1][i] for c in char_range(0,255) for i in
range(len(t[1]) if t[1][i][0]==c ))
```
// creating (batch_no,doc), (word sublist)

```
.map(lambda t : (t[0],min(hash_fn(t[1])))
```
// ((hash_fn,doc),min_hash)

```
.collect()
```

Q4. a)

Given: Jaccard similarity (s) of two sets, S1 and S2 = 0.75

Length of signature matrix (m) = 500

Band size (r) = 5

Therefore, Number of bands (b) = m/r = 500/5 = 100

We know, that the probability the minhash signatures for these documents agree in any one particular row of the signature matrix is equal to the Jaccard similarity (s) = 0.75

The probability that the two sets matched in at least 1 band can be calculated as follows :

1. The probability that the signatures agree in all rows of one particular band is $s^r = (0.75)^5 = 0.237$.

2. The probability that the signatures disagree in at least one row of a particular band is $1 - s^r = 1 - 0.237 = 0.763$.

3. The probability that the signatures disagree in at least one row of each of the bands is $(1 - s^r)^b = (0.763)^{100} = 1.79e^{-12}$

4. The probability that the signatures agree in all the rows of at least one band, and therefore meaning the two sets matched in at least one band, is $1 - (1 - s^r)^b = 1- (1.79e^{-12}) = 0.9999 \sim 1$.

Thus, we can see it is almost 100% probable that the two sets match in at least one band when they have a high Jaccard similarity of 0.75 .

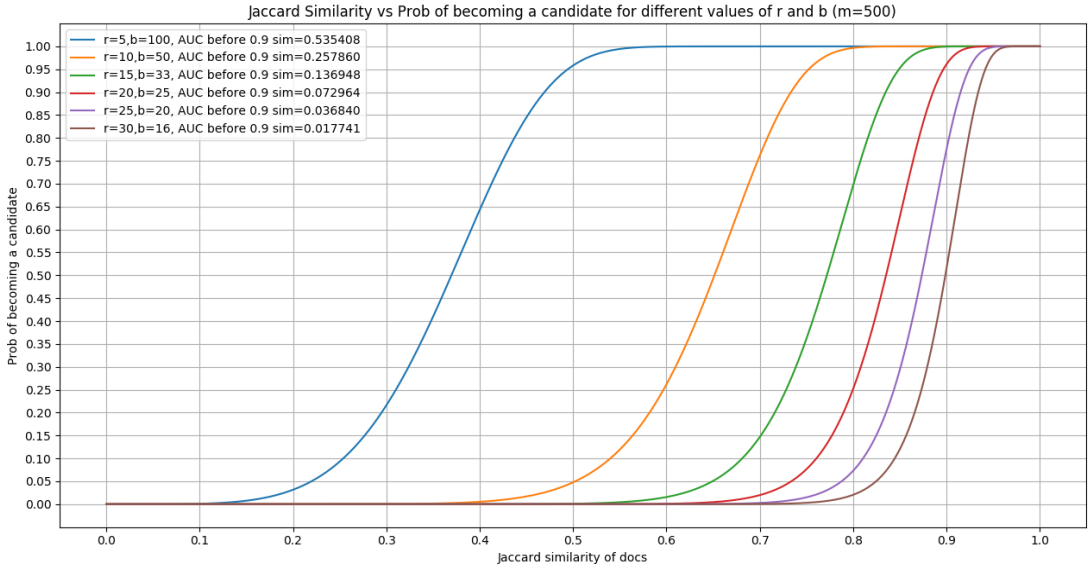
- Referenced MMds Book, 3.4.2

Q4. b)

From a), we know the probability for a pair of sets to be a candidate pair is calculated by $1 - (1 - s^r)^b$ (the symbols notation are same as a)).

First, we need to capture 99% of the pairs that have Jaccard similarity atleast 90%. So we can try different values of r and b , such that $1 - (1 - s^r)^b >= 0.99$. In my experiment , I have tried r = 5,10,15,20,25,30 and b = 500/r. As we can see in the plot, the curves for first 3 values of r can easily capture 99% of pairs with Jaccard similarity atleast 90%, but as we increase r further , i.e increase band size and decrease no of bands, we are giving less opportunities for a signature match (number of bands decreased) and making it more difficult for a given signature to match (band size increased , so need to match more rows). Thus, the values before r=15 seems to be serving our first goal.

Second, we need to keep the false positive rate upto 25%, i.e. sum of probabilities of pairs with Jaccard similarity less than 90% should be around 25%. So, we integrate the expression $1 - (1 - s^r)^b$ from 0 to our threshold (0.9) and check for which value of r and b, we are getting the area under curve around .25. According to my experiment, for r=10, b=50, area under curve is about .25. We observe here, for lower r value (5) , the false positive rate was higher, this makes sense because for a smaller band size , there are more chances of capturing pairs that may not have a high similarity , since we are comparing only few rows, also, the number of bands are higher, so we gave more opportunities to find a similar signature.



Thus we can safely say , for r=10 and b=50, we will be able to capture 99% of the pairs with Jaccard similarity atleast 90% , with a false positive rate of 25%.

Note : For creating these curves, I have taken discrete points with a minimal step value (since integrating it would be a tough ask), so it comes out to be a smooth curve, and calculated the area by summing up small rectangles below the curve.

Q4.c)

To speed up LSH, we could decrease r, thereby increasing b. By decreasing r, i.e band size , we are comparing fewer rows, which increases the chances of getting similar signature, so in this case, we will be able to encounter a similar signature sooner than for a bigger band size. This approach works, since we need to find only one band signature that matches for both sets, as soon as we find it, we need not move further. Although this approach will speed up LSH, it will lead to more false positives, as explained in part b). The plot in part b) supports this theory too, as r is decreasing, we see the area under curve before threshold to be increasing, i.e. false positive rate increases as we decrease r. However, the LSH speed will increase as explained above.