1. Using the Spark distribution and data you downloaded in hw1, follow the step by step instructions at the following URL:


**Note:** To get the examples to run, you must first replace the file machine-learning/scala/build.sbt in your Spark distribution with the one from the following link:

https://web.stanford.edu/~rezab/dao/hw/build.sbt

Now, answer the following questions:

(a) Set the rank to 8, and print the factor for the movie “Saving Private Ryan (1998)”

(b) Set the rank to 5, and print the factor for the movie “Alien (1979)”

Submit your code and answers.

**Solution.** The easiest way to get the factor for a movie is to perform a lookup by movie id on the productFeatures RDD in the model object. You can find the movie ids by reading the data files or by manipulating the movies map in the starter code.

(a) var rank = 8; val lambda = 1.0; val numIter = 20;
    val sprModel = ALS.train(training, rank, numIter, lambda)
    println(sprModel.productFeatures.lookup(2028).last.deep.mkString("\n"))

On the medium dataset with my ratings this gives the following output (your numbers will be different):

0.6873751422019242, 0.6718855758972344, 0.6834398363332329, 0.6893283557452138, 0.6876830429230224, 0.6782168366822537, 0.6792120139113983, 0.6688776672478713

(b) rank = 5
    val alienModel = ALS.train(training, rank, numIter, lambda)
    println(alienModel.productFeatures.lookup(1214).last.deep.mkString("\n"))

Output:

0.8177888909453307, 0.8243880686528039, 0.8126616775938632, 0.820525503879746, 0.8233517412785794

One common mistake I saw was collecting the productFeatures RDD locally and then filtering it for the desired factors. In our case this is possible because we are working with a small dataset, but in an industry setting this matrix could be too large to collect locally. Regardless, collecting the matrix before filtering it is unnecessarily wasteful.
I also saw solutions that created new RDDs and joined them with `productFeatures` to find the desired factors. This works but is non-idiomatic — joins are generally used for combining two tables of data instead of combining one piece of data with a table of data. In our case `filter` and `lookup` are more appropriate.

2. Using the Spark distribution and data you already downloaded in hw1, follow the step by step instructions at the following URL, skipping to section 4.1 in the tutorial: https://databricks-training.s3.amazonaws.com/graph-analytics-with-graphx.html#getting-started-again

Go to section 4.1 in the tutorial, and answer the following questions:

(a) What are websites with the top 10 pageranks in the wikipedia dataset?
(b) What are websites with the top 10 indegrees in the wikipedia dataset?

Submit your code and answers.

**Solution.**

(a) The code is given at the link, but the top 10 pageranks were

University of California, Berkeley: 1321.1117543121227 
Berkeley, California: 664.8841977233989 
Uc berkeley: 162.5013274339786 
Berkeley Software Distribution: 90.47860388486127 
Lawrence Berkeley National Laboratory: 81.9040939642022 
George Berkeley: 81.85226118458043 
Busby Berkeley: 47.87199821801991 
Berkeley Hills: 44.76406979519929 
Xander Berkeley: 30.32407534728813 
Berkeley County, South Carolina: 28.908336483710315

(b) We can adapt the tutorial code for this:

```scala
val in = graph.inDegrees.cache()
val titleAndIn = graph.outerJoinVertices(in) {
    (v, title, degOpt) => (degOpt.getOrElse(0), title)
}
titleAndIn.vertices.top(10) {
    Ordering.by((entry: (VertexId, (Int, String))) => entry._2._1)
}.foreach(t => println(t._2._2 + ": " + t._2._1))
```

yielding:

University of California, Berkeley: 7387 
Berkeley, California: 3900 
Uc berkeley: 989 
Lawrence Berkeley National Laboratory: 438 
Berkeley Software Distribution: 407 
George Berkeley: 403 
Busby Berkeley: 232
3. Give the pseudocode for finding the shortest path between a given source and a destination using the Pregel paradigm. Assume each vertex has a unique ID and each edge has a weight of 1.

Solution:

The basic idea for our algorithm is as follows:

1: Initialize distance attribute of every node $i$ to be $d_i = n + 1$
2: Set the distance attribute of the source node to be $d_s = 0$
3: iterate until the target node has $d_t < n + 1$
4: Every node $i$ passes $d_i + 1$ to each of its neighbors
5: Every node updates $d_i = \min\{d_i, \text{values received}\}$
6: end iterating

Consider that there is no need for a node to pass messages unless its distance value has changed since it last passed a message. Hence we can further improve this algorithm by reducing the number of messages transmitted:

1: Initialize distance attribute of every node $i$ to be $d_i = n + 1$
2: Initialize the change attribute of every node $i$ to be $c_i = 0$
3: For the source node set $d_s = 0$ and $c_s = 1$
4: iterate until the target node has $d_t < n + 1$
5: Every node $i$ with $c_i = 1$ passes $d_i + 1$ to each of its neighbors
6: Every node updates $d_i := \min\{d_i, \text{values received}\}$
7: Every node updates $c_i = 1$ if $d_i$ changed during the previous step, else $c_i = 0$
8: end iterating

Further improvements are also possible. Consider for example, running a distance process from both the source and the target node simultaneously. So each node stores four values:

(a) distance from source vertex
(b) indicator for changes in distance from source
(c) distance from target vertex
(d) indicator for changes in distance from target

We then stop the algorithm once any node has a distance to both the source and the target. We return the sum of these distances.

While you can design graphs for which this algorithm produces no improvement over the basic algorithm (even though it requires more memory) it is often found that this two way search significantly reduces the number of vertices that must pass messages
(for example, on an infinite 2D grid we half the number of vertices that ever pass a message).