In [ ]:
```
'This is a lab in the Introduction to Big Data and Spark Course at Berke
ley.'
```

In [1]:
```
import sys
import os
from test_helper import Test

baseDir = os.path.join('data')
inputPath = os.path.join('cs100', 'lab4', 'small')

ratingsFilename = os.path.join(baseDir, inputPath, 'ratings.dat.gz')
moviesFilename = os.path.join(baseDir, inputPath, 'movies.dat')
```

In [2]:
```
numPartitions = 2
rawRatings = sc.textFile(ratingsFilename).repartition(numPartitions)
rawMovies = sc.textFile(moviesFilename)

def get_ratings_tuple(entry):
    """ Parse a line in the ratings dataset
    Args:
        entry (str): a line in the ratings dataset in the form of UserI
D::MovieID::Rating::Timestamp
    Returns:
        tuple: (UserID, MovieID, Rating)
    """
    items = entry.split('::')
    return int(items[0]), int(items[1]), float(items[2])


def get_movie_tuple(entry):
    """ Parse a line in the movies dataset
    Args:
        entry (str): a line in the movies dataset in the form of MovieI
D::Title::Genres
    Returns:
        tuple: (MovieID, Title)
    """
    items = entry.split('::')
    return int(items[0]), items[1]


ratingsRDD = rawRatings.map(get_ratings_tuple).cache()
moviesRDD = rawMovies.map(get_movie_tuple).cache()

ratingsCount = ratingsRDD.count()
moviesCount = moviesRDD.count()

print 'There are %s ratings and %s movies in the datasets' % (ratingsCou
nt, moviesCount)
print 'Ratings: %s' % ratingsRDD.take(3)

print 'Movies: %s' % moviesRDD.take(3)

assert ratingsCount == 487650
assert moviesCount == 3883
assert moviesRDD.filter(lambda (id, title): title == 'Toy Story (199
5)').count() == 1
assert (ratingsRDD.takeOrdered(1, key=lambda (user, movie, rating): movi
e)
        == [(1, 1, 5.0)])
```

```
There are 487650 ratings and 3883 movies in the datasets
Ratings: [(1, 1193, 5.0), (1, 914, 3.0), (1, 2355, 5.0)]
Movies: [(1, u'Toy Story (1995)'), (2, u'Jumanji (1995)'), (3, u'Grumpi
er Old Men (1995)')]
```

In [3]:
```
tmp1 = [(1, u'alpha'), (2, u'alpha'), (2, u'beta'), (3, u'alpha'), (1,
u'epsilon'), (1, u'delta')]
tmp2 = [(1, u'delta'), (2, u'alpha'), (2, u'beta'), (3, u'alpha'), (1,
u'epsilon'), (1, u'alpha')]

oneRDD = sc.parallelize(tmp1)
twoRDD = sc.parallelize(tmp2)
oneSorted = oneRDD.sortByKey(True).collect()
twoSorted = twoRDD.sortByKey(True).collect()
print oneSorted
print twoSorted
assert set(oneSorted) == set(twoSorted)     # Note that both lists have
the same elements
assert twoSorted[0][0] < twoSorted.pop()[0] # Check that it is sorted by
the keys
assert oneSorted[0:2] != twoSorted[0:2]     # Note that the subset consi
sting of the first two elements does not match
```

```
[(1, u'alpha'), (1, u'epsilon'), (1, u'delta'), (2, u'alpha'), (2, u'be
ta'), (3, u'alpha')]
[(1, u'delta'), (1, u'epsilon'), (1, u'alpha'), (2, u'alpha'), (2, u'be
ta'), (3, u'alpha')]
```

In [4]:
```
def sortFunction(tuple):
    """ Construct the sort string (does not perform actual sorting)
    Args:
        tuple: (rating, MovieName)
    Returns:
        sortString: the value to sort with, 'rating MovieName'
    """
    key = unicode('%.3f' % tuple[0])
    value = tuple[1]
    return (key + ' ' + value)


print oneRDD.sortBy(sortFunction, True).collect()
print twoRDD.sortBy(sortFunction, True).collect()
```

```
[(1, u'alpha'), (1, u'delta'), (1, u'epsilon'), (2, u'alpha'), (2, u'be
ta'), (3, u'alpha')]
[(1, u'alpha'), (1, u'delta'), (1, u'epsilon'), (2, u'alpha'), (2, u'be
ta'), (3, u'alpha')]
```

In [5]:
```
oneSorted1 = oneRDD.takeOrdered(oneRDD.count(),key=sortFunction)
twoSorted1 = twoRDD.takeOrdered(twoRDD.count(),key=sortFunction)
print 'one is %s' % oneSorted1
print 'two is %s' % twoSorted1
assert oneSorted1 == twoSorted1
```

```
one is [(1, u'alpha'), (1, u'delta'), (1, u'epsilon'), (2, u'alpha'),
(2, u'beta'), (3, u'alpha')]
two is [(1, u'alpha'), (1, u'delta'), (1, u'epsilon'), (2, u'alpha'),
(2, u'beta'), (3, u'alpha')]
```

In [6]:
```
# TODO: Replace <FILL IN> with appropriate code
def getCountsAndAverages(IDandRatingsTuple):
    """ Calculate average rating
    Args:
        IDandRatingsTuple: a single tuple of (MovieID, (Rating1, Rating
2, Rating3, ...))
    Returns:
        tuple: a tuple of (MovieID, (number of ratings, averageRating))
    """
    movieID = IDandRatingsTuple[0]
    ratings = list(IDandRatingsTuple[1])
    num_ratings = len(ratings)
    sum_ratings = sum(ratings)
    avg_rating = float(sum_ratings) / float(num_ratings)
    return (movieID, (num_ratings, avg_rating))
```

In [7]:
```
# TEST Number of Ratings and Average Ratings for a Movie (1a)

Test.assertEquals(getCountsAndAverages((1, (1, 2, 3, 4))), (1, (4,
2.5)),
                            'incorrect getCountsAndAverages() with integ
er list')
Test.assertEquals(getCountsAndAverages((100, (10.0, 20.0, 30.0))), (100,
(3, 20.0)),
                            'incorrect getCountsAndAverages() with float
list')
Test.assertEquals(getCountsAndAverages((110, xrange(20))), (110, (20,
9.5)),
                            'incorrect getCountsAndAverages() with xrang
e')
```

```
1 test passed.
1 test passed.
1 test passed.
```

In [8]:
```
# TODO: Replace <FILL IN> with appropriate code

# From ratingsRDD with tuples of (UserID, MovieID, Rating) create an RDD with tuples of
# the (MovieID, iterable of Ratings for that MovieID)
movieIDsWithRatingsRDD = (ratingsRDD.map(lambda (user, movie, rating):
(movie, rating))
                          .groupByKey())


# Using `movieIDsWithRatingsRDD`, compute the number of ratings and average rating for each movie to
# yield tuples of the form (MovieID, (number of ratings, average rating))
movieIDsWithAvgRatingsRDD = movieIDsWithRatingsRDD.map(lambda (movie, rating):  getCountsAndAverages((movie, rating)))

# To `movieIDsWithAvgRatingsRDD`, apply RDD transformations that use `moviesRDD` to get the movie
# names for `movieIDsWithAvgRatingsRDD`, yielding tuples of the form
# (average rating, movie name, number of ratings)
movieNameWithAvgRatingsRDD = moviesRDD.join(movieIDsWithAvgRatingsRDD).map(lambda x:(x[1][1][1], x[1][0], x[1][1][0]))
```

In [9]:
```
# TEST Movies with Highest Average Ratings (1b)

Test.assertEquals(movieIDsWithRatingsRDD.count(), 3615,
                  'incorrect movieIDsWithRatingsRDD.count() (expected 361
5)')
movieIDsWithRatingsTakeOrdered = movieIDsWithRatingsRDD.takeOrdered(3)
Test.assertTrue(movieIDsWithRatingsTakeOrdered[0][0] == 1 and
                  len(list(movieIDsWithRatingsTakeOrdered[0][1])) == 993,
                  'incorrect count of ratings for movieIDsWithRatingsTakeO
rdered[0] (expected 993)')
Test.assertTrue(movieIDsWithRatingsTakeOrdered[1][0] == 2 and
                  len(list(movieIDsWithRatingsTakeOrdered[1][1])) == 332,
                  'incorrect count of ratings for movieIDsWithRatingsTakeO
rdered[1] (expected 332)')
Test.assertTrue(movieIDsWithRatingsTakeOrdered[2][0] == 3 and
                  len(list(movieIDsWithRatingsTakeOrdered[2][1])) == 299,
                  'incorrect count of ratings for movieIDsWithRatingsTakeO
rdered[2] (expected 299)')

Test.assertEquals(movieIDsWithAvgRatingsRDD.count(), 3615,
                  'incorrect movieIDsWithAvgRatingsRDD.count() (expected 3
615)')
Test.assertEquals(movieIDsWithAvgRatingsRDD.takeOrdered(3),
                  [(1, (993, 4.145015105740181)), (2, (332, 3.174698795180
723)),
                   (3, (299, 3.0468227424749164))],
                  'incorrect movieIDsWithAvgRatingsRDD.takeOrdered(3)')

Test.assertEquals(movieNameWithAvgRatingsRDD.count(), 3615,
                  'incorrect movieNameWithAvgRatingsRDD.count() (expected
3615)')
Test.assertEquals(movieNameWithAvgRatingsRDD.takeOrdered(3),
                  [(1.0, u'Autopsy (Macchie Solari) (1975)', 1), (1.0, u'B
etter Living (1998)', 1),
                   (1.0, u'Big Squeeze, The (1996)', 3)],
                   'incorrect movieNameWithAvgRatingsRDD.takeOrdered(3)')
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

In [10]:
```
# TODO: Replace <FILL IN> with appropriate code

# Apply an RDD transformation to `movieNameWithAvgRatingsRDD` to limit t
he results to movies with
# ratings from more than 500 people. We then use the `sortFunction()` he
lper function to sort by the

# average rating to get the movies in order of their rating (highest rat
ing first)
movieLimitedAndSortedByRatingRDD = (movieNameWithAvgRatingsRDD.filter(la
mbda x: x > (x[0], x[1], 500))).sortBy(sortFunction, False)
print 'Movies with highest ratings: %s' % movieLimitedAndSortedByRatingR
DD.take(20)
```

```
Movies with highest ratings: [(4.5349264705882355, u'Shawshank Redempti
on, The (1994)', 1088), (4.515798462852263, u"Schindler's List (1993)",
1171), (4.512893982808023, u'Godfather, The (1972)', 1047), (4.51046025
1046025, u'Raiders of the Lost Ark (1981)', 1195), (4.505415162454874,
u'Usual Suspects, The (1995)', 831), (4.457256461232604, u'Rear Window
(1954)', 503), (4.45468509984639, u'Dr. Strangelove or: How I Learned t
o Stop Worrying and Love the Bomb (1963)', 651), (4.43953006219765, u'S
tar Wars: Episode IV - A New Hope (1977)', 1447), (4.4, u'Sixth Sense,
The (1999)', 1110), (4.394285714285714, u'North by Northwest (1959)', 7
00), (4.379506641366224, u'Citizen Kane (1941)', 527), (4.375, u'Casabl
anca (1942)', 776), (4.363975155279503, u'Godfather: Part II, The (197
4)', 805), (4.358816276202219, u"One Flew Over the Cuckoo's Nest (197
5)", 811), (4.358173076923077, u'Silence of the Lambs, The (1991)', 124
8), (4.335826477187734, u'Saving Private Ryan (1998)', 1337), (4.326241
134751773, u'Chinatown (1974)', 564), (4.325383304940375, u'Life Is Bea
utiful (La Vita \ufffd bella) (1997)', 587), (4.324110671936759, u'Mont
y Python and the Holy Grail (1974)', 759), (4.3096, u'Matrix, The (199
9)', 1250)]
```

In [11]:
```
# TEST Movies with Highest Average Ratings and more than 500 Reviews (1c)

Test.assertEquals(movieLimitedAndSortedByRatingRDD.count(), 194,
                  'incorrect movieLimitedAndSortedByRatingRDD.count()')
Test.assertEquals(movieLimitedAndSortedByRatingRDD.take(20),
                [(4.5349264705882355, u'Shawshank Redemption, The (1994)', 1088),
                 (4.515798462852263, u"Schindler's List (1993)", 1171),
                 (4.512893982808023, u'Godfather, The (1972)', 1047),
                 (4.510460251046025, u'Raiders of the Lost Ark (1981)', 1195),
                 (4.505415162454874, u'Usual Suspects, The (1995)', 831),
                 (4.457256461232604, u'Rear Window (1954)', 503),
                 (4.45468509984639, u'Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1963)', 651),
                 (4.43953006219765, u'Star Wars: Episode IV - A New Hope (1977)', 1447),
                 (4.4, u'Sixth Sense, The (1999)', 1110), (4.394285714285714, u'North by Northwest (1959)', 700),
                 (4.379506641366224, u'Citizen Kane (1941)', 527), (4.375, u'Casablanca (1942)', 776),
                 (4.363975155279503, u'Godfather: Part II, The (1974)', 805),
                 (4.358816276202219, u"One Flew Over the Cuckoo's Nest (1975)", 811),
                 (4.358173076923077, u'Silence of the Lambs, The (1991)', 1248),
                 (4.335826477187734, u'Saving Private Ryan (1998)', 1337),
                 (4.326241134751773, u'Chinatown (1974)', 564),
                 (4.325383304940375, u'Life Is Beautiful (La Vita \ufffd bella) (1997)', 587),
                 (4.324110671936759, u'Monty Python and the Holy Grail (1974)', 759),
                 (4.3096, u'Matrix, The (1999)', 1250)], 'incorrect sortedByRatingRDD.take(20)')
```

```
1 test passed.
1 test passed.
```

In [12]:
```
trainingRDD, validationRDD, testRDD = ratingsRDD.randomSplit([6, 2, 2],
seed=0L)

print 'Training: %s, validation: %s, test: %s\n' % (trainingRDD.count(),
                                                    validationRDD.coun
t(),
                                                    testRDD.count())
print trainingRDD.take(3)
print validationRDD.take(3)
print testRDD.take(3)

assert trainingRDD.count() == 292716

assert validationRDD.count() == 96902
assert testRDD.count() == 98032

assert trainingRDD.filter(lambda t: t == (1, 914, 3.0)).count() == 1
assert trainingRDD.filter(lambda t: t == (1, 2355, 5.0)).count() == 1
assert trainingRDD.filter(lambda t: t == (1, 595, 5.0)).count() == 1

assert validationRDD.filter(lambda t: t == (1, 1287, 5.0)).count() == 1
assert validationRDD.filter(lambda t: t == (1, 594, 4.0)).count() == 1
assert validationRDD.filter(lambda t: t == (1, 1270, 5.0)).count() == 1

assert testRDD.filter(lambda t: t == (1, 1193, 5.0)).count() == 1
assert testRDD.filter(lambda t: t == (1, 2398, 4.0)).count() == 1
assert testRDD.filter(lambda t: t == (1, 1035, 5.0)).count() == 1
```

```
Training: 292716, validation: 96902, test: 98032

[(1, 914, 3.0), (1, 2355, 5.0), (1, 595, 5.0)]
[(1, 1287, 5.0), (1, 594, 4.0), (1, 1270, 5.0)]
[(1, 1193, 5.0), (1, 2398, 4.0), (1, 1035, 5.0)]
```

In [13]:
```
# TODO: Replace <FILL IN> with appropriate code
import math

def computeError(predictedRDD, actualRDD):
#     """ Compute the root mean squared error between predicted and actua
l
#     Args:
#         predictedRDD: predicted ratings for each movie and each user wh
ere each entry is in the form
#                       (UserID, MovieID, Rating)
#         actualRDD: actual ratings where each entry is in the form (User
ID, MovieID, Rating)
#     Returns:
#         RSME (float): computed RSME value
#     """

#     # Transform predictedRDD into the tuples of the form ((UserID, Movi
eID), Rating)
    predictedReformattedRDD = (predictedRDD.map(lambda (x, y, z):((x,
```

```
y), z)))
#
#    # Transform actualRDD into the tuples of the form ((UserID, MovieI
D), Rating)
    actualReformattedRDD = (actualRDD.map(lambda (x, y, z):((x, y), z)))

#    # Compute the squared error for each matching entry (i.e., the same
(User ID, Movie ID) in each
#    # RDD) in the reformatted RDDs using RDD transformtions - do not us
e collect()
    squaredErrorsRDD = (predictedReformattedRDD.join(actualReformattedRD
D)).map(lambda (x, y): (x[0], x[1], (y[0]-y[1])**2))

    # Compute the total squared error - do not use collect()
    totalError = squaredErrorsRDD.map(lambda x: x[2]).reduce(lambda a,
b: a + b)

    # Count the number of entries for which you computed the total squar
ed error
    numRatings = squaredErrorsRDD.count()

    # Using the total squared error and the number of entries, compute t
he RSME
    rmse = float(math.sqrt(float(totalError) / float(numRatings)))

    return rmse


# sc.parallelize turns a Python list into a Spark RDD.
testPredicted = sc.parallelize([
    (1, 1, 5),
    (1, 2, 3),
    (1, 3, 4),
    (2, 1, 3),
    (2, 2, 2),
    (2, 3, 4)])
testActual = sc.parallelize([
     (1, 2, 3),
     (1, 3, 5),
     (2, 1, 5),
     (2, 2, 1)])
testPredicted2 = sc.parallelize([
     (2, 2, 5),
     (1, 2, 5)])
testError = computeError(testPredicted, testActual)
print 'Error for test dataset (should be 1.22474487139): %s' % testError

testError2 = computeError(testPredicted2, testActual)
print 'Error for test dataset2 (should be 3.16227766017): %s' % testErro
r2

testError3 = computeError(testActual, testActual)
print 'Error for testActual dataset (should be 0.0): %s' % testError3
```

```
Error for test dataset (should be 1.22474487139): 1.22474487139
Error for test dataset2 (should be 3.16227766017): 3.16227766017
Error for testActual dataset (should be 0.0): 0.0
```

In [14]:
```
# TEST Root Mean Square Error (2b)
Test.assertTrue(abs(testError - 1.22474487139) < 0.00000001,
                'incorrect testError (expected 1.22474487139)')
Test.assertTrue(abs(testError2 - 3.16227766017) < 0.00000001,
                'incorrect testError2 result (expected 3.16227766017)')
Test.assertTrue(abs(testError3 - 0.0) < 0.00000001,
                'incorrect testActual result (expected 0.0)')
```

```
1 test passed.
1 test passed.
1 test passed.
```

In [15]:
```
# TODO: Replace <FILL IN> with appropriate code
from pyspark.mllib.recommendation import ALS

#You will end up with an RDD of the form: [(1, 1287), (1, 594), (1, 127
0)]
validationForPredictRDD = validationRDD.map(lambda x: (x[0], x[1]))
seed = 5L
iterations = 5
regularizationParameter = 0.1
ranks = [4, 8, 12]
errors = [0, 0, 0]
err = 0
tolerance = 0.02

minError = float('inf')
bestRank = -1
bestIteration = -1
for rank in ranks:
    model = ALS.train(trainingRDD, rank, seed=seed, iterations=iteration
s,
                      lambda_=regularizationParameter)
    predictedRatingsRDD = model.predictAll(validationForPredictRDD)
    error = computeError(predictedRatingsRDD, validationRDD)
    errors[err] = error
    err += 1
    print 'For rank %s the RMSE is %s' % (rank, error)
    if error < minError:
        minError = error
        bestRank = rank

print 'The best model was trained with rank %s' % bestRank
```

```
For rank 4 the RMSE is 0.892734779484
For rank 8 the RMSE is 0.890121292255
For rank 12 the RMSE is 0.890216118367
The best model was trained with rank 8
```

In [16]:
```
# TEST Using ALS.train (2c)
Test.assertEquals(trainingRDD.getNumPartitions(), 2,
                  'incorrect number of partitions for trainingRDD (expec
ted 2)')
Test.assertEquals(validationForPredictRDD.count(), 96902,
                  'incorrect size for validationForPredictRDD (expected
96902)')
Test.assertEquals(validationForPredictRDD.filter(lambda t: t == (1, 190
7)).count(), 1,
                  'incorrect content for validationForPredictRDD')
Test.assertTrue(abs(errors[0] - 0.883710109497) < tolerance, 'incorrect
errors[0]')
Test.assertTrue(abs(errors[1] - 0.878486305621) < tolerance, 'incorrect
errors[1]')
Test.assertTrue(abs(errors[2] - 0.876832795659) < tolerance, 'incorrect
errors[2]')
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

In [17]:
```
# TODO: Replace <FILL IN> with appropriate code


myModel = ALS.train(trainingRDD, bestRank, seed=seed, iterations=iterati
ons,
                    lambda_=regularizationParameter)

testForPredictingRDD = testRDD.map(lambda x: (x[0], x[1]))
predictedTestRDD = myModel.predictAll(testForPredictingRDD)

testRMSE = computeError(testRDD, predictedTestRDD)

print 'The model had a RMSE on the test set of %s' % testRMSE
```

```
The model had a RMSE on the test set of 0.891048561304
```

In [18]:
```
# TEST Testing Your Model (2d)
Test.assertTrue(abs(testRMSE - 0.87809838344) < tolerance, 'incorrect te
stRMSE')
```

```
1 test passed.
```

In [19]:
```
# TODO: Replace <FILL IN> with appropriate code

trainingAvgRating = trainingRDD.map(lambda x: x[2]).mean()
print 'The average rating for movies in the training set is %s' % trainingAvgRating


testForAvgRDD = testRDD.map(lambda x: (x[0], x[1], trainingAvgRating))

testAvgRMSE = computeError(testRDD, testForAvgRDD)
print 'The RMSE on the average set is %s' % testAvgRMSE
```

```
The average rating for movies in the training set is 3.57409571052
The RMSE on the average set is 1.12036693569
```

In [20]:
```
# TEST Comparing Your Model (2e)
Test.assertTrue(abs(trainingAvgRating - 3.57409571052) < 0.000001,
                'incorrect trainingAvgRating (expected 3.57409571052)')
Test.assertTrue(abs(testAvgRMSE - 1.12036693569) < 0.000001,
                'incorrect testAvgRMSE (expected 1.12036693569)')
```

```
1 test passed.
1 test passed.
```

You now have code to predict how users will rate movies!

In [21]:
```
print 'Most rated movies:'
print '(average rating, movie name, number of reviews)'
for ratingsTuple in movieLimitedAndSortedByRatingRDD.take(50):
    print ratingsTuple
```

```
Most rated movies:
(average rating, movie name, number of reviews)
(4.5349264705882355, u'Shawshank Redemption, The (1994)', 1088)
(4.515798462852263, u"Schindler's List (1993)", 1171)
(4.512893982808023, u'Godfather, The (1972)', 1047)
(4.510460251046025, u'Raiders of the Lost Ark (1981)', 1195)
(4.505415162454874, u'Usual Suspects, The (1995)', 831)
(4.457256461232604, u'Rear Window (1954)', 503)
(4.45468509984639, u'Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1963)', 651)
(4.43953006219765, u'Star Wars: Episode IV - A New Hope (1977)', 1447)
(4.4, u'Sixth Sense, The (1999)', 1110)
(4.394285714285714, u'North by Northwest (1959)', 700)
(4.379506641366224, u'Citizen Kane (1941)', 527)
(4.375, u'Casablanca (1942)', 776)
(4.363975155279503, u'Godfather: Part II, The (1974)', 805)
(4.358816276202219, u"One Flew Over the Cuckoo's Nest (1975)", 811)
(4.358173076923077, u'Silence of the Lambs, The (1991)', 1248)
(4.335826477187734, u'Saving Private Ryan (1998)', 1337)
(4.326241134751773, u'Chinatown (1974)', 564)
```

```
(4.325383304940375, u'Life Is Beautiful (La Vita \ufffd bella) (1997)',
587)
(4.324110671936759, u'Monty Python and the Holy Grail (1974)', 759)
(4.3096, u'Matrix, The (1999)', 1250)
(4.309457579972183, u'Star Wars: Episode V - The Empire Strikes Back (1
980)', 1438)
(4.30379746835443, u'Young Frankenstein (1974)', 553)
(4.301346801346801, u'Psycho (1960)', 594)
(4.296438883541867, u'Pulp Fiction (1994)', 1039)
(4.286535303776683, u'Fargo (1996)', 1218)
(4.282367447595561, u'GoodFellas (1990)', 811)
(4.27943661971831, u'American Beauty (1999)', 1775)
(4.268053855569155, u'Wizard of Oz, The (1939)', 817)
(4.267774699907664, u'Princess Bride, The (1987)', 1083)
(4.253333333333333, u'Graduate, The (1967)', 600)
(4.236263736263736, u'Run Lola Run (Lola rennt) (1998)', 546)
(4.233807266982622, u'Amadeus (1984)', 633)
(4.232558139534884, u'Toy Story 2 (1999)', 860)
(4.232558139534884, u'This Is Spinal Tap (1984)', 516)
(4.228494623655914, u'Almost Famous (2000)', 744)
(4.2250755287009065, u'Christmas Story, A (1983)', 662)
(4.216757741347905, u'Glory (1989)', 549)
(4.213358070500927, u'Apocalypse Now (1979)', 539)
(4.20992028343667, u'L.A. Confidential (1997)', 1129)
(4.204733727810651, u'Blade Runner (1982)', 845)
(4.1886120996441285, u'Sling Blade (1996)', 562)
(4.184615384615385, u'Braveheart (1995)', 1300)
(4.184168012924071, u'Butch Cassidy and the Sundance Kid (1969)', 619)
(4.182509505703422, u'Good Will Hunting (1997)', 789)
(4.166969147005445, u'Taxi Driver (1976)', 551)
(4.162767039674466, u'Terminator, The (1984)', 983)
(4.157545605306799, u'Reservoir Dogs (1992)', 603)
(4.153333333333333, u'Jaws (1975)', 750)
(4.149840595111583, u'Alien (1979)', 941)
(4.145015105740181, u'Toy Story (1995)', 993)
```

In [22]:
```
# TODO: Replace <FILL IN> with appropriate code
myUserID = 0

# Note that the movie IDs are the *last* number on each line. A common e
rror was to use the number of ratings as the movie ID.
myRatedMovies = [(0, 1088, 4.0),
(0, 1047, 4.0),
(0, 1195, 4.0),
(0, 1447, 5.0),
(0, 1110, 5.0),
(0, 805, 5.0),
(0, 811, 5.0),
(0, 564, 3.0),
(0, 1438, 3.0),
(0, 1039, 5.0),
(0, 1218, 5.0),
(0, 811, 3.0),

(5, 1775, 4.5),
(0, 817, 3.0),
(0, 600, 2.0),
(5, 633, 2.5),
(0, 860, 3.0),
(0, 662, 4.0),
(0, 549, 3.0),
(0, 539, 5.0),
(0, 1129, 4.0),
(0, 845, 5.0),
(0, 1300, 5.0),
(0, 983, 4.0),
(0, 750, 5.0),
(0, 941, 5.0)]
myRatingsRDD = sc.parallelize(myRatedMovies)
print 'My movie ratings: %s' % myRatingsRDD.take(10)
print(myRatingsRDD.count())
print(ratingsRDD.count())
```

```
My movie ratings: [(0, 1088, 4.0), (0, 1047, 4.0), (0, 1195, 4.0), (0,
1447, 5.0), (0, 1110, 5.0), (0, 805, 5.0), (0, 811, 5.0), (0, 564,
3.0), (0, 1438, 3.0), (0, 1039, 5.0)]
26
487650
```

In [23]:
```
# TODO: Replace <FILL IN> with appropriate code

trainingWithMyRatingsRDD = trainingRDD.union(myRatingsRDD)

print ('The training dataset now has %s more entries than the original t
raining dataset' %
       (trainingWithMyRatingsRDD.count() - trainingRDD.count()))
assert (trainingWithMyRatingsRDD.count() - trainingRDD.count()) == myRat
ingsRDD.count()
```

```
The training dataset now has 26 more entries than the original training
dataset
```

In [24]:
```
##### TODO: Replace <FILL IN> with appropriate code
myRatingsModel = ALS.train(trainingWithMyRatingsRDD, bestRank, seed=see
d, iterations=iterations,
                       lambda_=regularizationParameter)
```

In [25]:
```
# TODO: Replace <FILL IN> with appropriate code
predictedTestMyRatingsRDD = myRatingsModel.predictAll(testForPredictingR
DD)
testRMSEMyRatings = computeError(testRDD, predictedTestMyRatingsRDD)
print 'The model had a RMSE on the test set of %s' % testRMSEMyRatings
```

```
The model had a RMSE on the test set of 0.892003201254
```

In [26]:
```
# TODO: Replace <FILL IN> with appropriate code

# Use the Python list myRatedMovies to transform the moviesRDD into an R
DD with entries that are pairs of the form (myUserID, Movie ID) and that
does not contain any movies that you have rated.
myUnratedMoviesRDD = moviesRDD.map(lambda x: (0, x[0])).filter(lambda x:
x[1]!=[x[1] for x in myRatedMovies])
# Use the input RDD, myUnratedMoviesRDD, with myRatingsModel.predictAl
l() to predict your ratings for the movies
predictedRatingsRDD = myRatingsModel.predictAll(myUnratedMoviesRDD)
```

In [27]:
```
# TODO: Replace <FILL IN> with appropriate code

# Transform movieIDsWithAvgRatingsRDD from part (1b), which has the form
(MovieID, (number of ratings, average rating)), into and RDD of the form
(MovieID, number of ratings)
movieCountsRDD = movieIDsWithAvgRatingsRDD.map(lambda x: (x[0], x[1]
[0]))

# Transform predictedRatingsRDD into an RDD with entries that are pairs
of the form (Movie ID, Predicted Rating)
predictedRDD = predictedRatingsRDD.map(lambda x: (x[1], x[0]) )
#
# Use RDD transformations with predictedRDD and movieCountsRDD to yield
an RDD with tuples of the form (Movie ID, (Predicted Rating, number of r

atings))
predictedWithCountsRDD  = predictedRDD.join(movieCountsRDD).map(lambda
x: (x[0], (x[1][0], x[1][1])))

# moviesRDD.join(movieIDsWithAvgRatingsRDD)

# Use RDD transformations with PredictedWithCountsRDD and moviesRDD to y
ield an RDD with tuples of the form (Predicted Rating, Movie Name, numbe
r of ratings), for movies with more than 75 ratings
ratingsWithNamesRDD = (predictedWithCountsRDD.join(moviesRDD)).filter(la
mbda x: (x[1][0][0],x[1][1],x[1][0][1]>75))

predictedHighestRatedMovies = ratingsWithNamesRDD.takeOrdered(20, key=la
mbda x: -x[0])
print ('\n'.join(map(str, predictedHighestRatedMovies)))
```

```
(3952, ((0, 316), u'Contender, The (2000)'))
(3951, ((0, 35), u'Two Family House (2000)'))
(3950, ((0, 40), u'Tigerland (2000)'))
(3949, ((0, 214), u'Requiem for a Dream (2000)'))
(3948, ((0, 665), u'Meet the Parents (2000)'))
(3947, ((0, 46), u'Get Carter (1971)'))
(3946, ((0, 83), u'Get Carter (2000)'))
(3945, ((0, 38), u'Digimon: The Movie (2000)'))
(3944, ((0, 9), u'Bootmen (2000)'))
(3943, ((0, 75), u'Bamboozled (2000)'))
(3942, ((0, 26), u'Sorority House Massacre II (1990)'))
(3941, ((0, 20), u'Sorority House Massacre (1986)'))
(3940, ((0, 14), u'Slumber Party Massacre III, The (1990)'))
(3939, ((0, 24), u'Slumber Party Massacre II, The (1987)'))
(3938, ((0, 24), u'Slumber Party Massacre, The (1982)'))
(3937, ((0, 115), u'Runaway (1984)'))
(3936, ((0, 99), u'Phantom of the Opera, The (1943)'))
(3935, ((0, 18), u'Kronos (1973)'))
(3934, ((0, 27), u'Kronos (1957)'))
(3933, ((0, 13), u'Killer Shrews, The (1959)'))
```

In [ ]:
```
'This is a lab in the Introduction to Big Data and Spark Course at Berke
ley.'
```