3. Classifying Users

The first of the three problems you must solve requires you to label, or classify, all users as either children or adults. You know that parental control events label a user, and you know that the only parental control events that you saw were those that label the account as a child account. You also know that only adult accounts are able to perform account control operations like changing the password. You should be able to use that information to label some of the accounts and then use those known labels to discover the unknown labels.

A natural approach would be to use a classifier, such as logistic regression or naïve Bayes. But before you can get to your classifier, you have to decide what the features are. Each session contains a set of actions and timings, so one approach would be to try to learn the labels based on behavior profiles, such as how many movies are watched or whether feedback is given. While that may be possible, you'll find it hard to tease out the signal because having parental controls enabled does not guarantee anything about the user's behavior. The main thing that setting parental controls tells us is that some actions (account operations, as noted before) and some content is disallowed for that account.

Given that some content is disallowed, the content viewed should be a stronger source of signal. Some content will only be accessible to accounts without parental controls enabled, and some content will presumably be much more likely to be viewed by accounts that have parental controls enabled. To create features from the content viewed, we can count each item of content as a boolean feature, which fits well with either the logistic regression or naïve Bayes approach.

Before you begin building a classifier, however, think for a moment about the structure of this problem. In this data set, there are roughly 2000 users, each of which viewed zero or more content items, and over 8500 content items, each of which was viewed by zero or more users. This structure is clearly a bipartite graph, and as such, a graph-oriented approach will be very direct and natural. The general idea is that the known labels from the users will propagate to the content items they viewed, and from there to other users who viewed the same content items. The same would go for ratings and reviews as well. The one hitch is that you don't know (and shouldn't expect) that there's a strict separation between the items viewed by accounts with parental controls enabled and those viewed by accounts without parental controls enabled. You therefore have to be a little careful about propagating the labels. Instead of propagating absolute labels, you could let each node propagate influence. This approach is known as SimRank and feels like a better place to start than a classifier. You can always fall back to a classifier later if SimRank doesn't work out. A good source for more information about simrank is Jeh and Widom 2001 (http://ilpubs.stanford.edu:8090/508/1/2001–41.pdf) or chapter 5 in Mining Massive Data Sets.

Step 1. Extract content items played

The first thing you'll need to do is extract the content items played, rated, or reviewed by every user. Even though you've now significantly reduced the size of the data file you're working with, you still want to use Hadoop so that your approach is scalable. For the same reasons as before, you'll write a streaming job using Python. For the map phase, the idea is to extract from the JSON every item that is played, rated, or reviewed by a user. In the reduce phase, you'll aggregate them by user and dump them out with one user per record. One other thing you'll need to carry through the script is the known labels assigned in the cleaning step. You'll extract those from the JSON in the mapper and output them as part of the keys in the reducer.

1. Write a mapper

For the mapper, we want to use a key that will let us aggregate by user but also sort by dates. To do that, we'll output a compound key that is composed of the user ID, the session start date, and the session end date. Because Hadoop will sort the keys lexicographically, we'll need to convert the dates into seconds since the epoch. Because the keys are sorted without regard for the fact that the keys are compound, the order of the key components will be the order of precedence for sorting.

Create a file called `kid_map.py` with the following contents:

```
#!/usr/bin/python

import json
import sys

def main():
   # Read all lines from stdin
   for line in sys.stdin:
      data = json.loads(line)

      # Collect all items touched
      items = set()
      items.update(data['played'].keys())
      items.update(data['rated'].keys())
      items.update(data['reviewed'].keys())

      # Generate a comma-separated list
      if items:
         itemstr = ','.join(items)
      else:
         itemstr = ','

      # Emit a compound key and compound value
      print "%s,%010d,%010d\t%s,%s" % (data['user'], long(data['start']), long(data['end']),
data['kid'], itemstr)

if __name__ == '__main__':
   main()
```

2. Write a reducer

For the reducer, you have to be careful with the way labels are aggregated and output. First, note that the key from the mapper contains the start and end timestamps, so the keys received by the reducer are aggregated by the user and then sorted by the creation time (and the user). This gives you a time sequence order for the setting of the labels. If you see a label, and you've not seen a label before, then you adopt that label. If you see a label, and you've seen the same label before, nothing changes. If you see a label, and you've seen a conflicting label before, you want to treat this user as two different users: the first label user and the second label user. If you never see a label for a user, then that user will remain unlabeled. When you output each user record, append an 'a' to user IDs for users that are labeled as adults, and append a 'k' to user IDs for users that are labeled as kids. For unlabeled users, just use the user ID as is.

Create a file called `kid_reduce.py` with the following contents:

```
#!/usr/bin/python

import sys

def main():
    current = None

    # Read all lines from stdin
    for line in sys.stdin:
        # Decompose the compound key and value
        key, value = line.strip().split('\t')
        user, start, end = key.split(',')
        kid, itemstr = value.split(',', 1)

        # Create a data record for this user
        data = {}
        data['user'] = user
        data['kid'] = str2bool(kid)
        data['items'] = set(itemstr.split(","))

        if not current:
            current = data
        else:
            if current['user'] != user or \
                (data['kid'] != None and current['kid'] != None and data['kid'] != current['kid']):
                # If this is a new user or we have new information about whether the user is a kid
                # that conflicts with what we knew before, then print the current record and start
                # a new record.
                dump(current)
                current = data
            else:
                if data['kid'] != None and current['kid'] == None:
```

```
               # If we just found out whether the user is a kid, store that
               current['kid'] = data['kid']

            # Store the items
            current['items'].update(data['items'])

   # Print the record for the last user.
   dump(current)

"""
Emit the data record
"""
def dump(data):
   # Remove any empty items
   try:
      data['items'].remove('')
   except KeyError:
      pass

   # If there are still items in the record, emit it
   if len(data['items']) > 0:
      # Annotate the session ID if we know the user is an adult or child
      if data['kid'] == True:
         data['user'] += 'k'
      elif data['kid'] == False:
         data['user'] += 'a'
      print "%s\t%s" % (data['user'], ",".join(data['items']))

"""
Translate a string into a boolean, but return None if the string doesn't parse.
"""
def str2bool(str):
   b = None

   if str.lower() == 'true':
      b = True
   elif str.lower() == 'false':
      b = False

   return b
if __name__ == '__main__':
   main()
```

Notice that the map task outputs a compound key, but you only want to aggregate by the first part. Fortunately, the simple sort applied during the shuffle and sort phase will accomplish both the aggregation by user ID and the ordering by time stamps. In MapReduce, this approach would usually be called a secondary sort, but because we're using Hadoop streaming, it doesn't require any additional code or configuration to happen.

3. Run the job

Run the job with the following commands:

```
$ hadoop jar $STREAMING -mapper kid_map.py -file kid_map.py -reducer kid_reduce.py -file kid_reduce.py
-input clean -output kid
…
13/11/18 19:23:57 INFO streaming.StreamJob: Job complete: job_201311081846_0022
13/11/18 19:23:57 INFO streaming.StreamJob: Output: kid
$ hadoop fs -cat kid/part-00000 | head -4
10108881    9107,16316
10142325    9614
10151338a   34645
10151338k   38467,33449,26266
cat: Unable to write to output stream.
```

The job runs and gives you the expected output.

Step 2. Prepare the SimRank algorithm

The next step is to prepare to run the SimRank algorithm. SimRank requires an adjacency matrix and a list of nodes in the teleport set. Start with the teleport sets. You'll need one for the adults, and one for the kids. Because you have that information succinctly in the output from the previous job, you can extract the information directly with command line tools without using Hadoop. We also want to divide our labels into training and test sets. In the following, use 80%/20%. Run the following commands:

```
$ hadoop fs -cat kid/part-\* | cut -f1 | grep a > adults
$ expr `wc -l adults | awk '{ print $1 }'` / 5
20
$ tail -n +21 adults | hadoop fs -put - adults_train
$ head -20 adults | hadoop fs -put - adults_test
$ hadoop fs -cat kid/part-\* | cut -f1 | grep k > kids
$ expr `wc -l kids | awk '{ print $1 }'` / 5
24
$ tail -n +25 kids | hadoop fs -put - kids_train
$ head -24 kids | hadoop fs -put - kids_test
```

In the above commands, the `expr` command is used to perform math operations and output the results. All math operations are done using integer arithmetic. The backtick (`) operators execute the command they surround and output the command's output.

Step 3. Build an adjacency matrix

Next, you'll build an adjacency matrix. The most efficient way to build and store the matrix is as a series of sparse columns, where each column is represented as the column label followed by all of the row labels where the matrix has non-zero entries in that column. The value in each cell is then the inverse of the number of non-zero entries in the column. (To be more efficient, we could even replace column and row labels with ids.) Interestingly, the format of the data in the kids directory is exactly that sparse columnar format, except that data only includes columns for users. To have a proper adjacency matrix, we'll also need columns for the content items. Fortunately, that's pretty easy to do with a quick MapReduce job.

1. Write a mapper

For the mapper, we just need to output every content item with every user who viewed it. Create a file called `item_map.py` with the following contents:

```
#!/usr/bin/python
import sys
def main():
   # Read all lines from stdin
   for line in sys.stdin:
      key, value = line.strip().split('\t')
      items = value.split(',')

      # Emit every item in the set paired with the user ID
      for item in items:
         print "%s\t%s" % (item, key)
if __name__ == '__main__':
   main()
```

2. Write a reducer

For the reducer, you need only aggregate all of the users for each content item. Create a file called `item_reduce.py` with the following contents:

```
#!/usr/bin/python
import sys
def main():
   last = None

   # Read all lines from stdin
   for line in sys.stdin:
      item, user = line.strip().split('\t')
      if item != last:
         if last != None:
            # Emit the previous key
            print "%s\t%s" % (last, ','.join(users))

         last = item
         users = set()
      users.add(user)
   # Emit the last key
   print "%s\t%s" % (last, ','.join(users))
if __name__ == '__main__':
   main()
```

3. Run the job

Run the job with the following commands:

```
$ hadoop jar $STREAMING -mapper item_map.py -file item_map.py -reducer item_reduce.py -file
item_reduce.py -input kid -output item
...
13/11/20 11:42:43 INFO streaming.StreamJob:  map 100%  reduce 100%
13/11/20 11:42:45 INFO streaming.StreamJob: Job complete: job_201311081846_0038
13/11/20 11:42:45 INFO streaming.StreamJob: Output: item
$ hadoop fs -cat item/part-\* | head
10081e1 85861225,78127887,83817844,67863534,79043502
10081e10    10399917
10081e11    58912004
10081e2 58912004
10081e3 10399917
10081e4 10399917
10081e5 10399917
10081e7 10399917
10081e8 58912004
10081e9 58912004
cat: Unable to write to output stream.
```

You now have an adjacency matrix and can implement the SimRank algorithm. There are not any well-tested packages for SimRank on Hadoop, but SimRank is pretty simple to implement. The implementation below consists of a simple MapReduce job to do the matrix math (power iteration) and a shell script wrapper to manage the iterations.

Step 4. Implement the SimRank algorithm

Start with the mapper for the MapReduce job. All the mapper needs to do is read in the current SimRank vector and compute the matrix product with the adjacency matrix. First read in the SimRank vector from HDFS and store it in a dictionary, getting columns from the adjacency matrix as input records. For each non-zero entry in a column, multiply by the corresponding entry in the SimRank vector and emit the result with the row label as the key.

1. Write a mapper

Create a new file called `simrank_map.py` with the following contents:

```
#!/usr/bin/python

import sys

def main():
   if len(sys.argv) < 2:
      sys.stderr.write("Missing args: %s\n" % ":".join(sys.argv))
      sys.exit(1)

   v = {}

   # Read in the vector
   with open(sys.argv[1]) as f:
      for line in f:
         (key, value) = line.strip().split("\t")
         v[key] = float(value)

   # Now read the matrix from the mapper and do the math
   for line in sys.stdin:
      col, value = line.strip().split("\t")
      rows = value.split(',')

      for row in rows:
         try:
            # Add the product to the sum
            print "%s\t%.20f" % (row, v[col] / float(len(rows)))
         except KeyError:
            # KeyError equates to a zero, which we don't need to output.
            pass
if __name__ == '__main__':
   main()
```

2. Write a reducer

For the reducer, sum up the intermediate values for each row, add in the teleport contribution, and emit the final sum as the value for that row in the new SimRank vector. To know what the contribution from the teleport set is, you'll need to read in the training set. The tricky part is to make sure you only add the teleport contribution to rows that should get it, and that you don't forget to output rows that have a teleport contribution but no contribution from the adjacency matrix. Start with beta value of 0.8. Create a new file called `simrank_reduce.py` with the following contents:

```
#!/usr/bin/python

import sys

BETA = 0.8

def main():
   if len(sys.argv) < 2:
      sys.stderr.write("Missing args: %s\n" % ":".join(sys.argv))
      sys.exit(1)

   # These are the known labels in the data and hence the sources of rank
   with open(sys.argv[1]) as f:
      sources = set(f.read().splitlines())

   # Calculate contribution from the teleport set
   teleport = (1.0 - BETA) / float(len(sources))

   # For every row of data, sum it up, multiply by BETA and
   # add the teleport contribution
   last = None
   sum = 0.0

   for line in sys.stdin:
      row, value = line.strip().split("\t")

      if row != last:
         if last != None:
            dump(last, sum, sources, teleport)

         last = row
         sum = 0.0

      # Add the product to the sum
      sum += float(value)
   # Print the last value
   dump(last, sum, sources, teleport)

   # For any source row that we didn't see data for,
   # print just the teleport contribution
   for row in sources:
      print "%s\t%.20f" % (row, teleport)

"""
Print the sum for the row times BETA, with the teleport
contribution added if it's a source row.
"""
def dump(last, sum, sources, teleport):
   if last in sources:
      print "%s\t%.20f" % (last, sum * BETA + teleport)
      sources.remove(last)
   else:
      print "%s\t%.20f" % (last, sum * BETA)

if __name__ == '__main__':
   main()
```

3. Write a utility to test convergence

Now, instead of running the job directly, create a shell script that you can iterate until the SimRank vector converges. Before you can do that, however, you'll need to write a quick utility to test convergence in the SimRank vector by summing up the changes from the iteration. Create a file called `simrank_diff.py` with the following contents:

```
#!/usr/bin/python

import subprocess
import sys

def main():
   if len(sys.argv) < 4:
      sys.stderr.write('Usage: python simrank_diff.py v1 v2 threshold')
      exit(-1)

   v = {}

   # Read the first file from HDFS
   with subprocess.Popen(['hadoop', 'fs', '-cat', sys.argv[1]], stdout=subprocess.PIPE).stdout as f:
      for line in f:
         row, value = line.strip().split('\t')
         v[row] = float(value)

   diff = 0

   # Read the second file from HDFS
   with subprocess.Popen(['hadoop', 'fs', '-cat', sys.argv[2]], stdout=subprocess.PIPE).stdout as f:
      for line in f:
         row, value = line.strip().split('\t')

         try:
            diff += abs(v[row] - float(value))
         except KeyError:
            diff += float(value)

   # If the amount of difference exceeds our threshold, exit with 1
   if diff < float(sys.argv[3]):
      exit(1)
   else:
      exit(0)

if __name__ == "__main__":
   main()
```

4. Write a shell script

Now you can write the shell script. You'll need to first create the initial SimRank vector and then run your power iteration job until it converges. Since you're just comparing the adult vector to the kid vector, you don't need much precision. For this data set, you can call a total change of less than 0.01 between iterations converged. In a larger data set, you'd pick a larger number.

Create a new file called `simrank.sh` with the following contents:

```
#!/bin/bash

# Store the training file name
s=$1
# Count the number of lines
n=`hadoop fs -cat $s | wc -l | awk '{ print $1 }'`
# Calculate the inverse number of lines
e=`perl -e "print 1 / $n;"`
# Get my user id
u=`id -u -n`

# Remove bits from previous runs
rm v

# Build the initial SimRank vector
for line in `hadoop fs -cat $s`
do
   echo -e "$line\t$e" >> v
done

# Upload it to HDFS
hadoop fs -rm v
hadoop fs -put v

i=0
j=0

# Iterate until we converge (more or less)
while :
do
   # Update the counters
   i=$j
   j=`expr $i + 1`
   # Run the job
   echo Beginning pass $j
   hadoop jar $STREAMING -files "hdfs:///user/$u/v,hdfs:///user/$u/$s" \
                         -mapper "simrank_map.py v" -file simrank_map.py \
                         -reducer "simrank_reduce.py $s" -file simrank_reduce.py \
                         -input kid -input item -output simrank$j

   # If this isn't the first pass, test for convergence
   if [ $i != 0 ]
   then
      # Since we don't need perfect results, we'll converge at 0.01
      python simrank_diff.py simrank$i/part-00000 simrank$j/part-00000 0.01

      if [ $? = 1 ]
      then
         exit
      fi
   fi

   # Prepare the SimRank vector for the next pass
   echo Copying vector for next round
   hadoop fs -rm v
   hadoop fs -cp simrank$j/part-00000 v
done
```

5. Run the job

Test the job on the adult training set. Make the script executable by any user with `chmod a+x` command and run the job. Note that it will take 24 iterations for SimRank to converge, so this step may take several minutes.

```
$ chmod a+x simrank.sh
$ ./simrank.sh adults_train
rm: v: No such file or directory
rm: `v': No such file or directory
Beginning pass 1
/tmp/hadoop-training/hadoop-unjar2754804698549065604/] []
/var/folders/ll/xl67db9x2rs47q5fs4b255rr0000gp/T/streamjob270418292900895757.jar tmpDir=null
13/11/25 15:47:58 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments.
Applications should implement Tool for the same.
13/11/25 15:47:58 INFO mapred.FileInputFormat: Total input paths to process : 2
13/11/25 15:47:59 INFO streaming.StreamJob: getLocalDirs(): [/tmp/hadoop-training/mapred/local]
13/11/25 15:47:59 INFO streaming.StreamJob: Running job: job_201311252212_0020
13/11/25 15:47:59 INFO streaming.StreamJob: To kill this job, run:
13/11/25 15:47:59 INFO streaming.StreamJob: /usr/bin/hadoop job  -
Dmapred.job.tracker=192.168.56.101:8021 -kill job_201311252212_0020
13/11/25 15:47:59 INFO streaming.StreamJob: Tracking URL: http://192.168.56.101:50030/jobdetails.jsp?
jobid=job_201311252212_0020
13/11/25 15:48:00 INFO streaming.StreamJob:  map 0%  reduce 0%
13/11/25 15:48:04 INFO streaming.StreamJob:  map 67%  reduce 0%
13/11/25 15:48:08 INFO streaming.StreamJob:  map 100%  reduce 0%
13/11/25 15:48:11 INFO streaming.StreamJob:  map 100%  reduce 100%
13/11/25 15:48:12 INFO streaming.StreamJob: Job complete: job_201311252212_0020
13/11/25 15:48:12 INFO streaming.StreamJob: Output: simrank1
Copying vector for next round
Deleted v
Beginning pass 2
…
$ hadoop fs -cat simrank24/part\* | head
10081e1 0.00001761487440064474
10081e10    0.00000278115643514088
10081e11    0.00000278115643514088
10081e2 0.00001761487440064474
10081e3 0.00001761487440064474
10081e4 0.00001761487440064474
10081e5 0.00001761487440064474
10081e7 0.00000278115643514088
10081e8 0.00000278115643514088
10081e9 0.00000278115643514088
cat: Unable to write to output stream.
```

That successfully gives you the final SimRank vector. Store it in any local directory as you'll need it later. Use the `hadoop fs -mv` command. Then, run the job again for the kid training set. Again, SimRank will need 24 iterations to converge, so this step could take some time.

```
$ hadoop fs -mv simrank24 adult_final
$ hadoop fs -rm -R simrank\*
$ ./simrank.sh kids_train
…
$ hadoop fs -mv simrank24 kid_final
```

Step 5. Interpret and compare the SimRank vectors

Now, the last thing is a way to interpret the results: to compare the final SimRank vector from the adult training set with the final SimRank vector from the kid training set. There are two steps you need to complete for the comparison. First, normalize the two vectors so that they represent an equal influence from a labeled node of either type. Second, compare each entry and assign a label based on the larger value.

You could accomplish both the normalization and comparison in Hive using a join query, but deduping the user ids isn't straightforward. It's possible, but it will result in some very complicated queries. The same is true for Pig. It will turn out to be easier to do the final processing with two simple MapReduce jobs. The first job normalizes the adult SimRank data, and the second assigns labels to users.

Step 6. Normalize the SimRank data

For the first job, you simply need to go through every record and multiply it by a factor of the number of adults over the number of kids. Thinking ahead to the next stage, you should also make every adult score negative. By making all adult scores negative in this job, you can simply add the kid and adult scores together in the next job and label the records based on whether the sum is positive or negative.

The above can be accomplished with a simple map-only job.

1. Write a map-only Job

Create a new file called `adult_map.py` with the following contents:

```
#!/usr/bin/python

import sys

def main():
   if len(sys.argv) < 3:
      sys.stderr.write("Missing args: %s\n" % sys.argv)

   # Calculate conversion factor
   num_adults = float(sys.argv[1])
   num_kids = float(sys.argv[2])
   factor = -num_adults / num_kids

   # Apply the conversion to every record and emit it
   for line in sys.stdin:
      key, value = line.strip().split('\t')

      print "%s\t%.20f" % (key, float(value) * factor)

if __name__ == "__main__":
    main()
```

You'll run this job below after you construct the second job.

2. Write a reduce-only Job

The second job adds together the adult and child SimRank scores and assigns a label based on the results. If the sum is more than zero, the user is a kid. Otherwise, the user is an adult. If you specify the `KeyValueTextInputFormat` as the input format, you can accomplish this operation with a reduce-only job. Create a new file called `combine_reduce.py` with the following contents:

```
#!/usr/bin/python

import re
import sys

def main():
   last = None
   last_root = None
   sum = 0.0
   p = re.compile(r'^(\d{7,8})[ak]?$')

   # Read all the lines from stdin
   for line in sys.stdin:
      key, value = line.strip().split('\t')
      m = p.match(key)

      # Ignore anything that's not a user ID
      if m:
         if key != last:
            # Dump the previous user ID's label if it's not
            # the same real user as the current user.
            if last != None and last_root != m.group(1):
               dump(last, sum)
            last = key
            last_root = m.group(1)
            sum = 0.0

         sum += float(value)

   dump(last, sum)
def dump(key, sum):
   # sum is between -1 and 1, so adding one and truncating gets us 0 or 1.
   # We want ties to go to adults, though.
   if sum != 0:
      print "%s\t%d" % (key, int(sum + 1))
   else:
      print "%s\t0" % key
if __name__ == "__main__":
    main()
```

3. Run both jobs

Run both jobs and get your final results with the following commands:

```
$ hadoop fs -cat adults_train | wc -l
      84
$ hadoop fs -cat kids_train | wc -l
     96
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -input adult_final -output adult_mod -mapper
"adult_map.py 84 96" -file adult_map.py
...
$ hadoop jar $STREAMING -D mapred.textoutputformat.separator=, -input adult_mod -input kid_final -
output final -reducer combine_reduce.py -file combine_reduce.py -inputformat
org.apache.hadoop.mapred.KeyValueTextInputFormat
…
```

Notice that our jobs set the output format to use commas as the delimiter, which saves you from having to convert to a CSV later.

4. Test your results against the hold-out test set

Because all of the user IDs in the adult set end with ‘a’ and all of the IDs in the kid set end with ‘k’, you can test your results against the hold-out data very simply:

```
$ hadoop fs -cat final/part\* | grep a, | grep -v ,0
$ hadoop fs -cat final/part\* | grep k, | grep -v ,1
```

By grepping for the adult entries (those with an “a” in them) and then grepping for those that do not include “,0” we find all mislabeled adult entries. The same goes for the child entries.

From the fact that both commands returned no output, you can see that you get perfect score on the hold-out set.

5. Re-run the job with combined test and hold-out data

Put the hold-out data back into the mix and run it again to get your final results.

Before you re-run the job, however, you should make one small change to the last job in order to save an extra step. The change removes ‘a’ and ‘k’ from the user IDs before it emits them. Open `combine_reduce.py` and modify it as follows (changes in bold red):

```
#!/usr/bin/python

import re
import sys

def main():
   last = None
   last_root = None
   sum = 0.0
   p = re.compile(r'^(\d{7,8})[ak]?$')

   # Read all the lines from stdin
   for line in sys.stdin:
      key, value = line.strip().split('\t')
      m = p.match(key)

      # Ignore anything that's not a user ID
      if m:
         if key != last:
            # Dump the previous user ID's label if it's not
            # the same real user as the current user.
            if last != None and last_root != m.group(1):
               dump(last_root, sum)
            last = key
            last_root = m.group(1)
            sum = 0.0

         sum += float(value)

   dump(last_root, sum)

def dump(key, sum):
   # sum is between -1 and 1, so adding one and truncating gets us 0 or 1.
   # We want ties to go to adults, though.
   if sum != 0:
      print "%s,%d" % (key, int(sum + 1))
   else:
      print "%s,0" % key

if __name__ == "__main__":
    main()
```

Now, rerun the whole chain with the following commands:

```
$ hadoop fs -rm -R adult_final adult_mod kid_final final simrank\*
$ hadoop fs -cat kid/part-\* | cut -f1 | grep a  | hadoop fs -put - adults_all
$ hadoop fs -cat kid/part-\* | cut -f1 | grep k  | hadoop fs -put - kids_all
$ ./simrank.sh adults_all
…
$ hadoop fs -mv simrank24 adult_final
$ hadoop fs -rm -R simrank\*
$ ./simrank.sh kids_all
…
$ hadoop fs -mv simrank24 kid_final
$ hadoop fs -cat adults_all | wc -l
     104
$ hadoop fs -cat kids_all | wc -l
     120
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -input adult_final -output adult_mod -mapper
"adult_map.py 104 120" -file adult_map.py
...
$ hadoop jar $STREAMING -input adult_mod -input kid_final -output final -reducer combine_reduce.py -
file combine_reduce.py -inputformat org.apache.hadoop.mapred.KeyValueTextInputFormat
…
$ hadoop fs -getmerge final Task1Solution.csv
```

Using the scoring methodology from the challenge, this solution scores an accuracy of 99.64%, mislabeling a total of 9 records.

Navigation
Table of Contents
Solution Kit Introduction
Project Introduction