DATASCI W261: Machine Learning at Scale

David Rose
david.rose@berkeley.edu
W261-1
Week 03
2015.09.18

HW3.0

- Merge sort is an algorithm that combines two or more lists that are in sorted order individually and outputs a single list in sorted order. The algorithm compares the first elements of each input list to determine the element with the lowest (or highest) value. This value is removed from the input list and appended to the output list. This process repeats until there are no remaining input values. The algorithm runs in O(m log n) time where m in the total number of elements in the combined input lists and n is the number of input lists. Hadoop uses a merge sort when preparing data to be sent to a reduce task to provide the reducer with a list of input sorted by key.
- A combiner function aggregates or otherwise combines output from a mapper function before it is sent to a reducer. It functions similarly to a reducer but with some constraints. In part the constraints are imposed because the combiner is not guaranteed to execute. Whether it does or not is a runtime decision by the Hadoop process.
    - One constraint is that the output of the combiner must match the signature of the output of the mapper.
    - Another is that the combiner will only operate on the output of the mapper tasks of a single node so that in a multi-node environment one combiner will not see all of the data but rather only the data that has been assigned to the map tasks on that particular node.
    - Combiners can be very effective in reducing the amount of bandwidth required for transmitting data from a map task to a reduce task.
    - An example usage would be in an item counting operation, whether it be words in a document or items in a shopping basket. A combiner can take the canonical map output, [key, 1], and produce for each key encountered a combined output, [key, n], where n is the sum of counts for the key.
- The Hadoop shuffle is the core operation of the Hadoop framework. The shuffle is reponsible for moving data from the map task output to the reduce task input.
    - Map side of shuffle:

        - map output is written to a circular memory buffer
        - when the buffer reaches a capacity threshold, data is spilled to disk
        - before being written to spill files, data is partitioned based on the reducer targets and sorted in memory by key
        - when map function completes, multiple spill files, if present, are merged into a single partitioned and sorted file.
    - Reduce side of shuffle:
        - the reducer receives notice from the job tracker that a map task is complete
        - the reducer begins copying the sorted files from the identified map nodes
        - the files are merged together, potentially in several rounds, maintaining the sort order
        - the files are fed to the reduce function

- The Apriori algorithm is used for gaining efficiency when deriving association rules. The algorithm is based on the insight that if an itemset I = $\{i_1..i_n\}$ is frequent (i.e., occurs greater than a pre-defined threshold in the data set), then all subsets of I are also frequent. In practice this is implemented by determining the frequent itemsets of length k = 1, removing the itemsets that are not frequent, and using the remaining itemsets to construct a candidate set of k + 1 items. This process is repeated iteratively until the itemset size of interest is reached or when there are no frequent itemsets remaining.
    - Although I can find no references to the use of the apriori algorithm in public health outbreak investigations, it could potentially be used to identify potential association rules between behaviors, events, symptoms, and diagnoses. Maybe.
- Confidence: a measure of a rule $(X \rightarrow Y)$ calculated as the ratio of the support of an itemset to the support of a chosen subset of the itemset, i.e., the left hand side of the rule:

$$confidence(X \rightarrow Y) = \frac{support(X \cup Y)}{support(X)}$$

- Lift: a measure of a rule $(X \rightarrow Y)$ calculated as the ratio of the confidence of the rule to the expected support of each constituent of the rule if the constituents were statistically independent:

$$lift(X \rightarrow Y) = \frac{confidence(X \rightarrow Y)}{support(Y)} = \frac{support(X \cup Y)}{support(X)support(Y)}$$

Type *Markdown* and LaTeX: α^2

In [193]:
```
# load data file into hdfs
!hdfs dfs -put -f ProductPurchaseData.txt /in
```

HW3.1

Mapper for exploratory data analysis

In [194]:
```
%%writefile map-3.1.py
#!/usr/bin/python
''' count some things
'''
from __future__ import print_function
import random
import sys
import time
random.seed()
for line in sys.stdin:
    # split line into tokens of items
    tokens = line.split()
    basketsize = len(tokens)
    # generate a basket key
    # this is probably overkill
    basketkey = ''.join([str(random.random()), str(time.time())])
    for token in tokens:
        print('\t'.join([token, str(basketsize), str(basketkey)]),

```

```
Overwriting map-3.1.py
```

Reducer for exploratory data analysis.

In [195]:
```
%%writefile reduce-3.1.py
#!/usr/bin/python
''' reducer aggregates some statistics
'''
from __future__ import print_function
import sys
items = {}
baskets = {}
maxbasket = 0
for line in sys.stdin:
    tokens = line.split()
    item = tokens[0]
    basketsize = int(tokens[1])
    basketkey = tokens[2]
    items[item] = 1
    # store size of each individual basket
    baskets[basketkey] = basketsize
    # keep running maximum of basket size
    maxbasket = max(maxbasket, basketsize)
# calculate some statistics
# mean basket size
meanbasket = sum(baskets.values())/float(len(baskets))
# standard deviation of basket size
std = (sum([(x - meanbasket)**2 for x in baskets.values()])
       / float(len(baskets)))**0.5
# emit results
results = ('total baskets: {}, unique items: {}, largest basket si
           + 'mean basket size: {:0.2f}, std deviation: {:0.2f}')
    .format(len(baskets), len(items), maxbasket, meanbasket, std)
print(results, file=sys.stdout)

```

```
Overwriting reduce-3.1.py
```

In [196]:
```
!chmod +x *.py
```

In [197]:

```
# HW3.1. Do some exploratory data analysis of this dataset.
# Report your findings such as number of unique products; largest
# basket, etc. using Hadoop Map-Reduce.
!printf "preparing output directory\n"
!hdfs dfs -rm -r -f -skipTrash /out/out-3.1
!printf "executing yarn task\n"
!yarn jar /usr/local/Cellar/hadoop/2.7.0/libexec/share/hadoop/tool
-files ./map-3.1.py,./reduce-3.1.py \
-mapper ./map-3.1.py -reducer ./reduce-3.1.py \
-input /in/ProductPurchaseData.txt -output /out/out-3.1
!printf "checking output directory: "
!hdfs dfs -ls /out/out-3.1
!printf "displaying results\n"
!hdfs dfs -cat /out/out-3.1/part-00000
```

```
preparing output directory
Deleted /out/out-3.1
executing yarn task
checking output directory: Found 2 items
-rw-r--r--   1 david supergroup          0 2015-09-21 17:50 /out/ou
t-3.1/_SUCCESS
-rw-r--r--   1 david supergroup        115 2015-09-21 17:50 /out/ou
t-3.1/part-00000
displaying results
total baskets: 31101, unique items: 12592, largest basket size: 37,
mean basket size: 12.24, std deviation: 5.03
```

HW3.2

In [198]:

```
%%writefile map-3.2.py
#!/usr/bin/python
''' map function to generate 2-itemsets

    implements in-memory combiner to aggregate counts of each
    2-itemset to reduce output file size
'''
from __future__ import print_function
import sys
itemsets = {}
magic = '*'
for line in sys.stdin:
    # split line into tokens of items
    items = line.split()
    for i in range(0, len(items)):
        item1 = items[i]
        # create fake itemset to keep count of 1-itemset
        # for use in reduce task
        itemset = ' '.join([item1, magic])
        if itemset not in itemsets:
            itemsets[itemset] = 0
        itemsets[itemset] += 1
        # create all possible combinations of 2-itemsets
        # for this basket
        for j in range(i + 1, len(items)):
            item2 = items[j]
            itemset = ' '.join(sorted([item1, item2]))
            if itemset not in itemsets:
                itemsets[itemset] = 0
            itemsets[itemset] += 1
for itemset in itemsets:
    results = '\t'.join([itemset, str(itemsets[itemset])])
    print(results, file=sys.stdout)

```

```
Overwriting map-3.2.py
```

In [199]:

```
%%writefile reduce-3.2.py
#!/usr/bin/python
''' reducer reads stream consisting of [2-itemset, count] pairs

    utilizes order inversion pattern to enable efficient
    stream processing

'''
from __future__ import print_function
import sys
# value for identifying left hand side of rule
magic = '*'
minsupport = 100
lhscurrent = ''
rhscurrent = ''
lhscount = 0
itemsetcount = 0
# loop through input
# when key changes, take action depending on which
# component of the key changes
for line in sys.stdin:
    tokens = line.split('\t')
    itemset = tokens[0]
    lhs = itemset.split()[0]
    rhs = itemset.split()[1]
    count = int(tokens[1])

    if not rhs == rhscurrent:
        if itemsetcount > minsupport and not rhscurrent == magic:
            # calculate the confidence for the current itemset
            confidence = itemsetcount / float(lhscount)
            print('{} -> {}\t{:0.8f}'
                  .format(lhscurrent, rhscurrent, confidence),
                file=sys.stdout)
        # reset loop values for right hand side
        itemsetcount = 0
        rhscurrent = rhs
    if not lhs == lhscurrent:
        # initialize the new lhs
        # reset loop values for left hand side
        lhscount = 0
        lhscurrent = lhs
    if rhs == magic:
        # increment support count for left hand side
        lhscount += count
    else:
        # increment support count for itemset
        itemsetcount += count
# process the last line
if itemsetcount > minsupport and not rhscurrent == magic and lhsco
    # calculate the confidence for the current itemset
    confidence = itemsetcount / float(lhscount)
    print('{} -> {}\t{:0.8f}'.format(lhscurrent, rhscurrent, confi
         file=sys.stdout)
```

```
55
56
```

```
Overwriting reduce-3.2.py
```

In [200]:
```
1 !chmod +x *.py
```

In [201]:
```
 1 # HW3.2.
 2
 3 !printf "preparing output directory\n"
 4 !hdfs dfs -rm -r -f -skipTrash /out/out-3.2
 5 !printf "executing yarn task\n"
 6 !yarn jar /usr/local/Cellar/hadoop/2.7.0/libexec/share/hadoop/tool
 7 -files ./map-3.2.py,./reduce-3.2.py \
 8 -mapper ./map-3.2.py -reducer ./reduce-3.2.py \
 9 -input /in/ProductPurchaseData.txt -output /out/out-3.2
10 !printf "checking output directory: "
11 !hdfs dfs -ls /out/out-3.2
12 !printf "displaying results\n"
13 !hdfs dfs -cat /out/out-3.2/part-00000 | sort -k 4nr -k 1 2>/dev/n
```

```
preparing output directory
Deleted /out/out-3.2
executing yarn task
checking output directory: Found 2 items
-rw-r--r--   1 david supergroup          0 2015-09-21 17:50 /out/ou
t-3.2/_SUCCESS
-rw-r--r--   1 david supergroup      41952 2015-09-21 17:50 /out/ou
t-3.2/part-00000
displaying results
DAI93865 -> FRO40251    1.00000000
ELE12951 -> FRO40251    0.99056604
DAI88079 -> FRO40251    0.98672566
DAI43868 -> SNA82528    0.97297297
DAI23334 -> DAI62779    0.95454545
```

HW3.3

NA

HW3.4

The approach outlined here relies on multiple invocations of hadoop, each with a single map and reduce function. Each invocation produces a file that will be made available through the hadoop shared cache architecture. This file will be used by the tasks in the next invocation.

- Round 1
  - Map1:
    - emit count (itemset, 1) for all 1-itemsets
  - Reduce1:
    - aggregate counts for all 1-itemsets
    - filter itemsets based on minimum support value
    - emit [itemset, count] for all frequent itemsets
  - file(s) produced by Reduce1 are combined, if more than 1, into file frequentitems1
- Round 2
  - Map2:
    - read in list of frequent 1-itemsets from file frequentitems1
    - emit count (itemset, 1) for all 2-itemsets iff the itemset contains at least one frequent 1-itemset
  - Reduce2:
    - aggregate counts for all 2-itemsets
    - filter itemsets based on minimum support value
    - emit [itemset, count] for all frequent itemsets
  - file(s) produced by Reduce2 are combined, if more than 1, into file frequentitems2
- Round 3
  - Map3:
    - read in list of frequent 2-itemsets from file frequentitems2
    - emit count (itemset, 1) for all 3-itemsets iff the itemset contains at least one frequent 2-itemset
  - Reduce3
    - aggregate counts for all 3-itemsets
    - filter itemsets based on minimum support value
    - read in list of frequent 2-itemsets from file frequentitems2 to make support levels for 2-itemsets available
    - for each frequent 3-itemset (X, Y, Z)
      - calculate confidence values for rules $((X, Y) \rightarrow Z), ((X, Z) \rightarrow Y), ((Y, Z) \rightarrow X)$ using the support values for the 3-itemsets and 2-itemsets, respectively.
        - e.g., $confidence((X, Y) \rightarrow Z) = \frac{support((X,Y) \cup Z)}{support((X,Y))}$
      - emit rule and confidence value

---

utility script for validating confidence levels

In [202]:

```
from __future__ import print_function
def confidence(itemA, itemB):
    with open('./ProductPurchaseData.txt', 'r') as fin:
        atotal = 0
        btotal = 0
        aonlytotal = 0
        bonlytotal = 0
        bothtotal = 0
        for line in fin:
            items = line.split()
            if itemA in items:
                atotal += 1
            if itemB in items:
                btotal += 1
            if itemA in items and not itemB in items:
                aonlytotal += 1
            if not itemA in items and itemB in items:
                bonlytotal += 1
            if itemA in items and itemB in items:
                bothtotal += 1
    print('LHS total: {}, LHS exclusive: {}, RHS total: {}, '
          + 'RHS exclusive: {}, both: {}' \
          .format(atotal, aonlytotal, btotal, bonlytotal, bothtota
    print('{} -> {}, confidence: {}'
          .format(itemA, itemB, bothtotal / float(atotal)))
    #print('{} -> {}, confidence: {}'.format(itemB, itemA, bothtot

```

In [203]:

```
confidence('DAI93865', 'FRO40251')
confidence('SNA98488', 'SNA99873')
```

```
LHS total: {}, LHS exclusive: {}, RHS total: {}, RHS exclusive: 208,
both: 0
DAI93865 -> FRO40251, confidence: 1.0
LHS total: {}, LHS exclusive: {}, RHS total: {}, RHS exclusive: 1, b
oth: 0
SNA98488 -> SNA99873, confidence: 1.0
```