STA 250, Final Project

Longphi Nguyen

December 11, 2013

Introduction

Companies are constantly trying to attract customers to buy from them. And one way of attracting customers is by having advertisements for various items. However, to give the same ad to everyone is naive. So, many online stores tailor ads based on a customer's rating on an item and shopping history. This is in an effort to improve the likelihood of interesting a customer with a new product. Unfortunately, popular online stores will often have hundreds of millions of customers and millions of items, which causes typical calculations to be computationally infeasible. One way to get around this is by using a divide-and-conquer approach known as MapReduce. This paper will explain how chained MapReduce (i.e. a sequence of MapReduces) was used to calculate the Pearson's correlation between two items. The correlation is then used to suggest items to a user.

Methodology

To explain the methodology, I will use an example. Suppose the data is in the form *(user, item, rating)*. Example data is given below, which is sorted for convenience.

user	item	rating
1	A	1
1	B	-1
1	C	1
1	D	3
2	A	2
2	B	-2
3	A	3
3	D	4
3	B	-3
4	B	6
4	D	1
4	D	2

The first MapReduce combines the data for a user such that each row contains all (item, rating) pairs for that user. The example becomes:

user	{(item rating)}
1	(A,1), (B,-1), (C,1), (D,3)
2	(A,2), (B,-2)
3	(A,3), (D,4), (B,-3)
4	(B,6), (D,1), (D,2)

This becomes convenient because we can now determine whether a user has rated two items. We can then calculate the correlation for all users who rated those two items. For example, user 1, 2, and 3 have rated items A and B. When the correlation between the items is high, we can say that they are correlated (i.e. users rate both have similarly).

The correlation is calculated by:

$$corr(\mathbf{x},\mathbf{y}) = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2\sum_{i=1}^{n}(y_i-\bar{y})^2}},$$

$$\text{where } \mathbf{x} = (x_1,...,x_n),\ \mathbf{y} = (y_1,...,y_n)$$

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i,\ \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

A more convenient form does not require $\bar{x}$ and $\bar{y}$ to be calculated ahead of time.

$$corr(\mathbf{x},\mathbf{y}) = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}\sqrt{n\sum_{i=1}^{n} y_i^2 - (\sum_{i=1}^{n} y_i)^2}} \tag{1}$$

Using the result from the first MapReduce, the second MapReduce calculates the correlation as explained above. For ease of visualization, the result is put into matrix form, where each row corresponds to a user, and each column corresponds to an item. This gives:

user/item	A	B	C	D
1	1	-1	1	3
2	2	-2		
3	3	-3		4
4		6		1.5

For how the 1.5 was obtained, see the first paragraph of the **Results** section. The correlation between item A and B forms vectors $\mathbf{a} = (1,2,3)$ and $\mathbf{b} = (-1,-2,-3)$, which were obtained by taking the rows that rated both items. And, clearly, $correlation(\mathbf{a},\mathbf{b}) = -1$. We can do this for all pair-wise combinations of $\{A,B,C,D\}$. Note: we do not need to calculate correlation for all permutations because correlation is symmetric. The result is as follows.

item pair	correlation
(A,B)	-1.0
(A,C)	0.0
(A,D)	1.0
(B,C)	0.0
(B,D)	-0.98
(C,D)	0.0

The final MapReduce sorts these correlations with respect to an item, so that we can recommend the most similarly rated item. That is, since (A,D) has the largest correlation (keeping A constant), compared to (A,B) and (A,C), then we can recommend D as the most relevant item to A. After sorting in increasing correlation, we obtain the following.

item pair	correlation
(A,B)	-1.0
(A,C)	0.0
(A,D)	1.0
(B,D)	-0.98
(B,C)	0.0
(C,D)	0.0

Then, for people who rated item A, we recommend items $D > C$. Similarly, for people who rated item B, we recommend items $C > D$. However, note that from the above table, it is difficult to give suggestions items C and D (and even B). See the **Possible Extensions** (1) and **Self Criticisms** sections.

Results

My code follows closely to Marcel Caraciolo's [1] code, with some improvements. We both come to the same output (except in special cases discussed next). Looking at the example data, we see that user 4 has rated item D twice. This can come up in cases where a user may give an item a second chance. Marcel's code treats both ratings as different items, which will cause ambiguity in the MapReduce steps and outputs. My code treats the item ID's as unique. So to deal with this situation, the average of the two ratings is taken (i.e. $\frac{1+2}{2} = 1.5$). This is a reasonable assumption since stores have unique item ID's for their items. To take the average, a *combiner* function was added between the first *mapper* and *reducer* step.

Furthermore, there were several minor optimizations, such as removing unecessary output from the *mappers* and *reducers* of Marcel's code. This led to a 88 second improvement (from 850.8 seconds to 762.82 seconds) in the computation time using the $100,000$ lines of data used by Marcel. A note on these computation times is given in the **Possible Extensions** (2) section.

Conclusions

Since the initial problem of calculating pair-wise correlation is computationally infeasible, MapReduce can help alleviate some of the troubles by reducing the problem into smaller steps. A chained MapReduce makes calculating the pair-wise correlation simple and efficient. But since sites need to constantly calculate these correlations, speeding up Marcel's code can save a lot of time for larger data sets. As such, we can give item suggestions to customers at a faster rate.

Though the example used was for online stores, the methodology is useful for many other applications. For example, Marcel used an example of movie relevance.

Possible Extensions

(1) The symmetry of correlation could be used to list all permutations of the item pairs (e.g. for $\{A, B, C, D\}$), rather than just the combinations. This would make it easier to find related items to item D. However, as is, we would need to iterate over the (item1, D) item pairs, order the correlations, and then find the items with the largest correlations. I did not do this because I could not find a simple way to implement it without using all permutations (rather than combinations) in the second MapReduce – which defeats the purpose of using the symmetry of correlation.

(2) The computation times obtained for Marcel's and my code were done locally. The times will inevitably be faster when done under Hadoop or other. However, I could not get Hadoop running, so this was what I could get.

(3) A clear flaw in calculating correlations is that we do not take advantage of a user's information. For example, it may be plausible that a person who rates many times has more credibility. As such, it may be reasonable to give their rating more weight compared to others.

(4) There are several other measures of similarity. Marcel discusses and implements some, such as a cosine, regularized correlation, and Jaccard measure. Depending on the data, some of these may be more preferable than the simple correlation.

(5) The computation time was calculated for the overall process. However, I wanted to determine the time it took for each mapper and reducer, to understand which process took the longest. I could not find a way to do this using

MRJob.

Self Criticisms

I took this opportunity to both help learn Python and MRJob, since it may help to improve an old project of mine. However, I may not have been adventurous enough since my code (written from scratch) turned out to be similar to Marcel's with some tweaks. It may have been a good opportunity to try to translate Marcel's approach directly into Java for even more speedup. That way, I could learn Java and see the computational speedup when doing so. Or, I could have tried working on my old project using MapReduce. My old project took word counts and calculated the pair-wise correlation, but was done naively. The ideas used here would be perfect for the old project. It would be interesting to see the speedup compared to the half-hour of calculations needed in the naive approach.

Also, my code is not all clear. For example, rather than giving my mapper and reducer functions descriptive names, they were simply called *mapper*1, *reducer*1, *mapper*2, *reducer*2, *mapper*3, *reducer*3. Although it makes it clear as to what it is (a mapper/reducer), it does not make it clear as to what it does. To compensate, I tried to provide comments, but they may not be completely clear.

Appendix

Code (final.py)

```
from mrjob.job import MRJob
from math import sqrt
import time

try:
    from itertools import combinations
except ImportError:
    from metrics import combinations


PRIOR_COUNT = 10
PRIOR_CORRELATION = 0


class SemicolonValueProtocol(object):
    def write(self, key, values):
        return ','.join(str(v) for v in values) #key, values


class Similarities(MRJob):

    OUTPUT_PROTOCOL = SemicolonValueProtocol

    def steps(self):
        return [
            self.mr(mapper=self.mapper1, combiner=self.combiner1, reducer=self.reducer1),
            self.mr(mapper=self.mapper2, reducer=self.reducer2),
            self.mr(mapper=self.mapper3, reducer=self.reducer3)]

#
```

```
# ~~==| MapReduce1 |==~~
#
# Create a (user, {(item, rating)}) tuple, where the user has rated the item
    def mapper1(self, key, line):
        user, item, rating = line.split('|')
        yield (user, item), float(rating)

    # If any user rated an item more than once, take the average of the ratings.
    def combiner1(self, user_item, ratings):
        n=0 # get length of ratings
        ratingSum=0
        for rating in ratings:
            n+=1
            ratingSum+=rating

        mean=ratingSum/n
        yield user_item[0], (user_item[1], mean)

    def reducer1(self, user, pairs):
        pairsList = [] # A list of (item, rating from this user) pairs

        for item, rating in pairs:
            pairsList.append((item, rating))

        yield user, pairsList

#
# ~~==| MapReduce2 |==~~
#
# Calculate correlation of ratings between (item1, item2).
# Create (item1, item2, correlation) tuples.
    def mapper2(self, user, pairs):
        for pair1, pair2 in combinations(pairs, 2):
            yield (pair1[0], pair2[0]), # (item1, item2)
                  (pair1[1], pair2[1]) # (rating1, rating2)

    def reducer2(self, item_pair, rating_pair):
        sum1, sum2, sum1_sq, sum2_sq, sum12 = 0.0, 0.0, 0.0, 0.0, 0.0
        n=0

        for rating1, rating2 in rating_pair:
            sum1 += rating1
            sum2 += rating2
            sum1_sq += rating1*rating1
            sum2_sq += rating2*rating2
            sum12 += rating1*rating2
            n+=1

        # Calculate correlation, using the form without means to avoid having to
        # calculate the means via a map/reduce before this one.
        denominator = sqrt(n*sum1_sq - sum1*sum1) * sqrt(n*sum2_sq - sum2*sum2)
        if denominator:
            correlation = (n*sum12 - sum1*sum2)/denominator
        else: # The denominator isn't valid, so set correlation=0
            correlation = 0.0
        yield item_pair, correlation

#
# ~~==| MapReduce3 |==~~
#
# For (item1, item*) pairs, sort by correlation. Where item* = (all items != item1) s.t.
# (item*, item1) correlation has not already been calculated.
# Create (item1, item*, correlation) tuples, sorted by increasing correlation.
    def mapper3(self, item_pair, correlation):
        yield (item_pair[0], correlation), item_pair[1]

    def reducer3(self, item1_corr, item2List):
        for item2 in item2List:

if __name__ == '__main__':
    t0=time.clock()
    Similarities.run()
    print time.clock()-t0
```

References

[1] Marcel Caraciolo, *Introduction to Recommendations with Map-Reduce and mrjob*. August 23, 2012, *http://aimotion.blogspot.com/2012/08/introduction-to-recommendations-with.html*

[2] MRJob documentation. *http://pythonhosted.org/mrjob/*