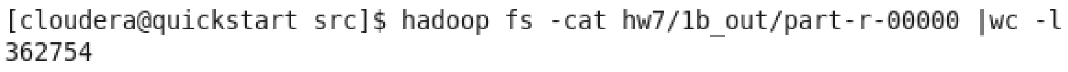
Homework 7

Problem 1: Computing Word Co-Occurrence (50%)

(a) See Java codes

(b) There are totally 362754 pairs:



```
[cloudera@quickstart src]$ hadoop fs -cat hw7/1b_out/part-r-00000 |wc -l
362754
```

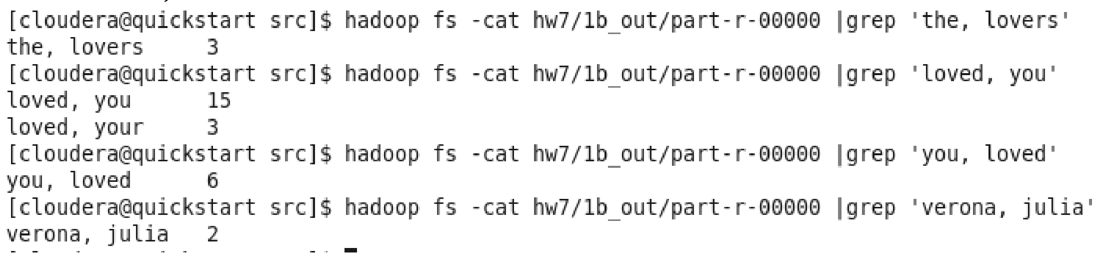
How often does the following word pairs occur:

– the, lovers: 3

– loved, you: 15

– you, loved: 6

– verona, Julia: 2



```
[cloudera@quickstart src]$ hadoop fs -cat hw7/1b_out/part-r-00000 |grep 'the, lovers'
the, lovers     3
[cloudera@quickstart src]$ hadoop fs -cat hw7/1b_out/part-r-00000 |grep 'loved, you'
loved, you      15
loved, your     3
[cloudera@quickstart src]$ hadoop fs -cat hw7/1b_out/part-r-00000 |grep 'you, loved'
you, loved      6
[cloudera@quickstart src]$ hadoop fs -cat hw7/1b_out/part-r-00000 |grep 'verona, julia'
verona, julia   2
```

(c) actual communication cost = Mapper input key-value pairs + Reducer input key-list-of-values pairs = 173126 + 362754 = 535880

(d) Two words may not be in one sentence even if one appears next to the other one. They can be the end of last sentence and the start of next sentence. And two words always show up together in sentence may not be neighbors.

Example1: “She is who I loved, you are not.” Although “loved you” are neighbors, they have little relationship.

Example2: “I really love you.” “I” and “love” appear together but that will not be counted.

(e) Stripes:

Communication cost = k-1.

Memory = 1 word + (k − 1 words and counts) * m, where m is the number of counts.

Pairs:

Communication cost = k (k-1) /2.

Memory = 2 ids + 1 count * m, where m is the number of counts.

Stripes approach is faster but cost more memory, while pairs approach is slower but requires fewer memory.

Problem 2: Collaborative Filtering - Similarity Measures (20%)

(a) $$\cos(r_x, r_y) = \frac{r_x{}^T . r_y}{\| r_x{}^T \| . \| r_y \|} = \frac{\sum_{i \in o_{xy}} r_{xi} r_{yi}}{\sqrt{\sum_{i \in o_{xy}} r_{xi}{}^2} \sqrt{\sum_{i \in o_{xy}} r_{yi}{}^2}}$$

If cosine similarity is normalized, then $\overline{r_x} = \overline{r_y} = 0$.

$$\cos(r_x, r_y) = \frac{\sum_{i \in o_{xy}} (r_{xi} - 0)(r_{yi} - 0)}{\sqrt{\sum_{i \in o_{xy}} (r_{xi} - 0)^2} \sqrt{\sum_{i \in o_{xy}} (r_{yi} - 0)^2}} = \frac{\sum_{i \in o_{xy}} (r_{xi} - \overline{r_x})(r_{yi} - \overline{r_y})}{\sqrt{\sum_{i \in o_{xy}} (r_{xi} - \overline{r_x})^2} \sqrt{\sum_{i \in o_{xy}} (r_{yi} - \overline{r_y})^2}} = p(x, y)$$

So the normalized cosine similarity measure corresponds to the Pearson correlation.

(b) From a quality perspective, by normalizing, we subtract the user’s mean rating so that we can remove bias.
From an implementation and data storage perspective, if we normalize the data, we need extra space and memory to calculate their summation and mean values.

(c) 1) Advantage: Jaccard similarity measurement could ignore values in the matrix and focus only on the sets of items rated. It works well when the utility matrix only reflected purchases.
2) Disadvantage: When utilities are more detailed ratings, the Jaccard distance loses important information.
3) We can use rounding data to calculate the Jaccard similarity. For example, we can set a fixed threshold such as the mean of the rating. Any rating that larger than the threshold can be considered as 1 while others can be considered as 0. In that case, Jaccard distance is intuitively correct.

Problem 3: Collaborative Filtering in MAPREDUCE (30%)

(a) Item-item similarity is more efficient if the number of query users is much bigger than the number of items.
Item-item similarity makes it easier to find similar items so it works better in practice.

(b)

	Movie1	Movie2	Movie3	Movie4	Movie5
User1	1	3	2		
User2		2	3		5
User3	1	2			

Job1:
Mapper output: (user1, [movie1, 1]), (user1, [movie2, 3]), (user1, [movie3, 2])
(user2, [movie2, 2]), (user2, [movie3, 3]), (user2, [movie5, 5])
(user3, [movie1, 1]), (user3, [movie2, 2])

Reducer input: (user1, < [movie1, 1], [movie2, 3], [movie3, 2]>)
(user2, < [movie2, 2], [movie3, 3], [movie5, 5]>)
(user3, < [movie1, 1], [movie2, 2]>)

Reducer output: ([movie1, movie2], [(1, 1), (3, 9), 3])
([movie2, movie3], [(3, 9), (2, 4), 6])
([movie1, movie3], [(1, 1), (2, 4), 2])

([movie2, movie3], [(2, 4), (3, 9), 6])
([movie2, movie5], [(2, 4), (5, 25), 10])
([movie3, movie5], [(3, 9), (5, 25), 15])

([movie1, movie2], [(1, 1), (2, 4), 2])

Job2:
Mapper output: ([movie1, movie2], [(1, 1), (3, 9), 3])
([movie2, movie3], [(3, 9), (2, 4), 6])
([movie1, movie3], [(1, 1), (2, 4), 2])

([movie2, movie3], [(2, 4), (3, 9), 6])
([movie2, movie5], [(2, 4), (5, 25), 10])
([movie3, movie5], [(3, 9), (5, 25), 15])

([movie1, movie2], [(1, 1), (2, 4), 2])

```
Reducer input: ([movie1, movie2], < [(1, 1), (3, 9), 3], [(1, 1), (2, 4), 2]>)
               ([movie2, movie3], < [(3, 9), (2, 4), 6], [(2, 4), (3, 9), 6]>)
               ([movie1, movie3], < [(1, 1), (2, 4), 2]>)
               ([movie2, movie5], < [(2, 4), (5, 25), 10]>)
               ([movie3, movie5], < [(3, 9), (5, 25), 15]>)
Reducer output: ([movie1, movie2], 0.98)
                ([movie2, movie3], 0.92)
                ([movie1, movie3], 1)
                ([movie2, movie5], 1)
                ([movie3, movie5], 1)
```