CS5425 Assignment1 Report

Name: Guo Shijia ID: E0337953

Task1:

something need to specify it:

Step1: when generate the word count tokens, I do some clean job, each token was been extracted from the first char to the last char, then transfer to lowercase(also remove the single character).

Example: “(-Jack%” → “jack”

The implement detail:

```
public String tokenClean(String token) {
    int length = token.length();
    int start = 0;
    int end = length-1;
    for(; start < length; start++) {
    char c = token.charAt(start);
    if((c >= 'A'&& c<='Z')||(c >='a' && c<='z')) {
        break;
        }
    }
    for(; end >= 0; end--) {
        char c = token.charAt(end);
        if((c >= 'A'&& c<='Z')||(c >='a' && c<='z')) {
            break;
        }
    }
    if(start > end) {
        return "";
    }
    return token.substring(start, end + 1).toLowerCase();
}
```

Step 2: generate the word count for each files

Step 3: extract the common words, reduce by the word, only the key contains 2 values will be write to the file, keep the small value.

Step 4: we use key to sort the record. We use the frequency of the words as the key, then implement our comparator class to sort the record descending by the key.

```
// rewrite the key sort comparator, to make sure the IntWritable as key, the order is reverse
public static class ReverseIntWritableComparator extends WritableComparator {
   public ReverseIntWritableComparator() {
     super(IntWritable.class);
   }
   @Override
   public int compare(byte[] b1, int s1, int l1,
                      byte[] b2, int s2, int l2) {
     int thisValue = readInt(b1, s1);
     int thatValue = readInt(b2, s2);
     return (thisValue<thatValue ? 1 : (thisValue==thatValue ? 0 : -1));
   }
}
```

Task2:

Step1: Aggregate the user’s item_scores records, in order to generate the item cooccurrence matrix, the key is user_id, value is all the item_score pairs belong to this user.

Step2: use the output of step1 to generate the item cooccurrence matrix, use 2 loops to achieve this. The output of this step like: key: item1:item2 value: number

Step3_1: to represent user's item_score pairs in another way in order to simplify the matrix calculation, we change output of step1 to this format:
key: item_id value: user_id, scores.
Step3_2: to represent item cooccurrence matrix in another way, we use the item_id as the key, the format change to this: key: item_id value: item_id,number
Step4_1: in mapper phase, we just take the output of step3_1 and step3_2 as input, in order to distinguish different source, we add prefix to the value(U stand for user_score pairs, I stand for item_number pairs). Use flag to distinguish source.

```
protected void setup(Context context) throws IOException, InterruptedException {
    FileSplit split = (FileSplit) context.getInputSplit();
    flag = split.getPath().getParent().getName();// data set
}

@Override
public void map(LongWritable key, Text values, Context context) throws IOException, InterruptedException {
    String[] tokens = Recommend.DELIMITER.split(values.toString());
    Text k = new Text(tokens[0]);
    Text v = new Text();
    if(flag.equalsIgnoreCase("step3_1")) {
        v.set("U,"+tokens[1]+","+tokens[2]);
    }
    if(flag.equalsIgnoreCase("step3_2")) {
        v.set("I,"+tokens[1]+","+tokens[2]);
    }
    context.write(k, v);
}
```

In reduce phase, we got 2 lists, for a specific item, we got its all user scores and all coocurrence items. For all the users who have scores for this specific item, we can calculate the recommendation scores of this item contribute the its coocurrence items. It can represent like this:
the recommendation scores that the item A contribute to item B for user C = scores(C give A) * number(item A and item B coocurrence number)

```
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOEx
        Map<String,Float> user_scores = new HashMap<String,Float>();
        Map<String,Integer> item_conoccurrence_num = new HashMap<String,Integer>();
       for(Text item : values) {
            String[] tokens = Recommend.DELIMITER.split(item.toString());
            if(tokens[0].equalsIgnoreCase("U")) {
                user_scores.put(tokens[1],Float.valueOf(tokens[2]));
            }
            if(tokens[0].equalsIgnoreCase("I")) {
                item_conoccurrence_num.put(tokens[1],Integer.valueOf(tokens[2]));
            }
       }
       Iterator<String> iter = user_scores.keySet().iterator();
       while(iter.hasNext()) {
            String user_id = iter.next();
            float scores = user_scores.get(user_id);
            Iterator<String> iter_item = item_conoccurrence_num.keySet().iterator();
            while(iter_item.hasNext()) {
                String item_id = iter_item.next();
                int num = item_conoccurrence_num.get(item_id);
                float res = num * scores;
                Text k = new Text(user_id);
                Text v = new Text(item_id + "," + res);
                context.write(k, v);
            }
       }
```

Step4_2: in the previous step, we calculate the scores that one item contribute to another item, actually, one item' s recommendation scores need sum up all coocurrence items contributions. We simply aggregate the scores in this step.

```
        public void reduce(Text key, Iterable<Text> values, Context context) thro

            Map<String,Float> item_scores = new HashMap<String,Float>();
            for(Text item : values) {
               String[] tokens = Pattern.compile("[,]").split(item.toString());
               String item_id = tokens[0];
               Float scores = Float.parseFloat(tokens[1]);
               if(item_scores.containsKey(item_id)) {
                   item_scores.put(item_id, scores + item_scores.get(item_id));
               }else {
                  item_scores.put(item_id, scores);
               }
            }
            Iterator<String> iter = item_scores.keySet().iterator();
            while(iter.hasNext()) {
                String item_id = iter.next();
                float scores = item_scores.get(item_id);
                Text v = new Text(item_id+","+scores);
                context.write(key,v);
            }

        }
    }
```

Step5: In the last step, we do the sorting and filtering, base on the recommendation rules, if user already buy one item, we need to filter it out from the recommendations. In map phase, we use original data file and recommend scores as input, aggregate by the user_id, use same trick to distinguish the 2 different sources. In reduce phase, we use provided method to sort the output. And item already buy will be filter from the recommendation.

```
public void reduce(Text key, Iterable<Text> values, Context context) throw
    Set<String> user_history = new HashSet<String>();
    HashMap<String,Float> recommendation = new HashMap<String,Float>();

   for(Text item : values) {
       String[] tokens = Recommend.DELIMITER.split(item.toString());
       if(tokens[0].equalsIgnoreCase("H")) {
          user_history.add(tokens[1]);
       }
       if(tokens[0].equalsIgnoreCase("R")) {
          recommendation.put(tokens[1],Float.parseFloat(tokens[2]));
       }
   }
   for(String item: user_history) {
       recommendation.remove(item);
   }
   List<Entry<String,Float>> list = new LinkedList<Entry<String,Float>>();
   list=SortHashMap.sortHashMap(recommendation);
   StringBuffer recommend_val = new StringBuffer();
   recommend_val.append("The user id is: " + key.toString()+"\n");
   for(Entry<String,Float> ilist : list){
    recommend_val.append(ilist.getKey());
    recommend_val.append("\t");
    recommend_val.append(ilist.getValue());
    recommend_val.append("\n");
   }
   if(key.toString().equalsIgnoreCase("953"))
       context.write(key, new Text(recommend_val.toString()));
}
```