Extracting single active user record from arff file

```
BufferedReader bufRead = new BufferedReader(
                                                    new FileReader(args[1]));

activeUsers = new Instances(bufRead);

Enumeration en=activeUsers.enumerateInstances();
Instance activeUser=(Instance)en.nextElement();
```

Extracting 5 nearest neighbors

```
LinearNNSearch kNN=new LinearNNSearch(dataset);

Instances neighbors=kNN.kNearestNeighbours(activeUser, 5);
double []  distances=kNN.getDistances();
```

Converting distances to similarities

```
double [] similarities=new double[distances.length];
for(int i=0;i<distances.length;i++)
{
        similarities[i]=1.0/distances[i];
}
```

Collect recommendations

Generate dictionary object (HashMap) to store recommendations of 5 neighbors for each book which is active user has not read

```
//to access each neighbor separately
Enumeration nInstances=neighbors.enumerateInstances();

//dictionary object:
//per each book – a list of recommendations: in this case 0 or 5
Map <String,List<Integer>> recommendations=
                                        new HashMap <String,List<Integer>>();

//Loop through all nearest neighbors
while(nInstances.hasMoreElements())
{
        Instance currNeighbor=(Instance)nInstances.nextElement();
        …
}
```

Add recommendations to the list

```
for(int i=0;i<currNeighbor.numAttributes();i++)
{
        if(activeUser.value(i)>0 ) //item is not ranked by the active user, but
                                   //ranked by a critique: 0 -yes, 1-no
        {
                //retrieve the name of the book
                String attrName=activeUser.attribute(i).name();
                List<Integer> lst=new ArrayList <Integer>();
                if(recommendations.containsKey(attrName))
                        lst=recommendations.get(attrName);

                 //read -we assume that ranked at max 5
                if( currNeighbor.value(i)<1)
                        lst.add(5);
                else
                        lst.add(0);         //add zero for this neighbor
        }
        recommendations.put(attrName, lst);
}
```

Class RecommendationRecord (to be sorted descending)

```
static class RecommendationRecord implements Comparable
<RecommendationRecord>
{
        public double score;
        public String attributeName;

        public int compareTo(RecommendationRecord other)
        {
                if(this.score>other.score)
                        return -1;
                if(this.score<other.score)
                        return 1;
                return 0;
        }
}
```

Collect weighted score and total similarity

```
List <RecommendationRecord> finalRanks
                    =new ArrayList <RecommendationRecord>();

Iterator <String> it=recommendations.keySet().iterator();
while(it.hasNext())
{
        String atrName=it.next();
        double totalImpact=0;
        double weightedSum=0;
        List <Integer> ranks=recommendations.get(atrName);
        for(int i=0;i<ranks.size();i++)
        {
                int val=ranks.get(i);

                totalImpact+=similarities[i];
                weightedSum+=(double)similarities[i]*val;
        }
        …
}
```

Compute score for each book

```
        …
        RecommendationRecord rec=new RecommendationRecord();
        rec.attributeName=atrName;
        rec.score=weightedSum/totalImpact;

        finalRanks.add(rec);
}

Collections.sort(finalRanks); //in descending order
```

Output top 3 recommendations

```
//print top 3 recommendations
System.out.println( finalRanks.get(0));
System.out.println( finalRanks.get(1));
System.out.println( finalRanks.get(2));
```