Data Mining Assignment 1

G . K . Sudharshan (CS13M050)

March 10, 2014

1 Implementation : Design mapreduce algorithms to take a very large file of integers and produce as output: (a) The largest integer. (b) The same set of integers, but with each integer appearing only once. (c) The count of the number of distinct integers in the input.

Solution :

(a) The Largest Integer

- **Mapper** : It aggregates all values and gives it to a single reducer. This is done by emitting all values using same key.
- **Reducer** : It finds maximum by collecting all output from mapper/combiner.
- **Combiner** : It reduces effort on reducer by emitting just the maximum from mapper rather than all values.

Listing 1: Map for Largest Integer

```
public static class LargestIntegerMapper extends Mapper<Object, Text,
    IntWritable, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private IntWritable intValue = new IntWritable();
        public void map(Object key, Text value, Context context)
                        throws IOException, InterruptedException {
                try{
                        intValue.set(Integer.parseInt(value.toString())
                            );
                }catch(NumberFormatException e){

                }
                context.write(one, intValue);
        }
```

Listing 2: Reducer for Largest Integer

```
public static class LargestIntegerReducer extends Reducer<IntWritable,
    IntWritable, IntWritable, NullWritable> {
```

```
        private IntWritable result = new IntWritable();
        public void reduce(IntWritable key, Iterable<IntWritable>
            values, Context context) throws IOException,
            InterruptedException {
                int max=Integer.MIN_VALUE;
                for(IntWritable value : values){
                        if(value.get()>max){
                                max =value.get();
                        }
                }
                result.set(max);
                context.write(result,NullWritable.get());
        }
}
```

Listing 3: Combiner for Largest Integer

```
public static class LargestIntegerCombiner extends Reducer<IntWritable,
     IntWritable, IntWritable, IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(IntWritable key, Iterable<IntWritable>
            values, Context context) throws IOException,
            InterruptedException {
                int max=Integer.MIN_VALUE;
                for(IntWritable value : values){
                        if(value.get()>max){
                                max =value.get();
                        }
                }
                result.set(max);
                context.write(key, result);
        }
```

(b) The same set of integers, but with each integer appearing only once

- **Mapper** : It emits all values as keys to the reducer, all similar values are passed to the reducer with same key.
- **Reducer** : It merges all nodes with similar values.
- **Combiner** : It reduces the number of emits from a mapper.

Listing 4: Mapper for Distinct Values

```
public static class DistinctMapper extends Mapper<Object, Text,
    IntWritable, NullWritable> {
        private IntWritable intValue = new IntWritable();
        public void map(Object key, Text value, Context context)
                        throws IOException, InterruptedException {
                try{
                        intValue.set(Integer.parseInt(value.toString())
                            );
                }catch(NumberFormatException e){

                }
                context.write(intValue,NullWritable.get());
```

```
        }
    }
```

Listing 5: Reducer for Distinct Values

```
    public static class DistinctReducer extends Reducer<IntWritable,
        NullWritable, IntWritable, NullWritable> {
            private IntWritable intValue = new IntWritable();
            public void reduce(IntWritable key, Iterable<NullWritable>
                values, Context context) throws IOException,
                InterruptedException {
                    intValue.set(key.get());
                    context.write(intValue, NullWritable.get());
            }
    }
```

(c) The count of the number of distinct integers in the input

- Mapper, Reducer and Combiner works in similar fashion as in above problem, using the counters for counting the distinct values. One could use defaults reducer counter which is implicitlly counts number of reduce tasks.

Listing 6: Reducer to count Distinct Values

```
    public static class DistinctCountReducer extends Reducer<IntWritable,
        NullWritable, IntWritable, NullWritable> {
            int count =0;
            public void reduce(IntWritable key, Iterable<NullWritable>
                values, Context context) throws IOException,
                InterruptedException {
                    count++;
                    context.write(key, NullWritable.get());
            }
            @Override
            protected void cleanup(Context context) throws IOException,
                            InterruptedException {
                    context.getCounter(COUNTER_GROUP, COUNTER_DISTINCT).
                        increment(count);
                    super.cleanup(context);
            }
    }
```

Listing 7: Printing Counter Values

```
                    if(status){
                     Counter distinctCounter = job.getCounters().getGroup(
                         COUNTER_GROUP).findCounter(COUNTER_DISTINCT);
                     System.out.println("Unique Elements :: " +
                          distinctCounter.getValue());
                    }
```

2 Implementation : Average rating of the movies Write map, reduce and combine functions using Hadoop to find average rating of movies. Dataset: MovieLens 100K The full data set, 100000 ratings by 943 users on 1682 items. Each user has rated at least 20 movies. Users and items are numbered consecutively from 1. The data is randomly ordered. This is a tab separated list of user id | item id | rating | timestamp.

Solution :

- **Mapper** : It emits a rating variable for each movie reviews by a user.
- **Reducer** : It merges all reviews for a movie and computes the average.
- **Combiner** : Use of combiner would reduce the traffic at the reducers.

Listing 8: Mapper to find Movie Ratings

```
public static class AverageMovieRatingMapper extends Mapper<Object,
    Text, IntWritable, RatingWritable> {
        private IntWritable movieId = new IntWritable();
        private RatingWritable rating = new RatingWritable();
        public void map(Object key, Text value, Context context)
                        throws IOException, InterruptedException {
                try{
                        String[] values = value.toString().split("\t");
                        movieId.set(Integer.parseInt(values[1]));
                        rating.setCount(1);
                        rating.setRate(Integer.parseInt(values[2]));
                }catch(NumberFormatException e){

                }
                context.write(movieId, rating);
        }
}
```

Listing 9: Reducer to find Movie Ratings

```
public static class AverageMovieRatingReducer extends Reducer<
    IntWritable, RatingWritable, IntWritable, DoubleWritable> {
        private DoubleWritable result = new DoubleWritable();
        public void reduce(IntWritable key, Iterable<RatingWritable>
            values, Context context) throws IOException,
            InterruptedException {
                long sumrating=0;
                long count=0;
                for(RatingWritable value : values){
                        sumrating+=value.getRate()*value.getCount();
                        count+=value.getCount();
                }
                result.set((double)sumrating/count);
```

```
                context.write(key, result);
            }
        }
```

Listing 10: Combiner to find Movie Ratings

```
    public static class AverageMovieRatingCombiner extends Reducer<
        IntWritable, RatingWritable, IntWritable, RatingWritable> {
            private RatingWritable rating = new RatingWritable();
            public void reduce(IntWritable key, Iterable<RatingWritable>
                values, Context context) throws IOException,
                InterruptedException {
                    Map<Integer,Long> ratingMap = new TreeMap<Integer,Long
                        >();
                    for(RatingWritable value : values){
                            Long cnt = ratingMap.get(value.getRate());
                            if(cnt==null){
                                    cnt = value.getCount();
                            }else{
                                    cnt = cnt+value.getCount();
                            }
                            ratingMap.put(value.getRate(),cnt);
                    }
                    for(Integer rate : ratingMap.keySet()){
                            rating.setRate(rate);
                            rating.setCount(ratingMap.get(rate));
                            context.write(key, rating);
                    }
            }
    }
```

- Using a customized writable class Rating Variable which holds the count of ratings and sum of ratings.

3 Implementation : Suppose we want to use a mapreduce framework to compute minhash signatures. If the matrix is stored in chunks that correspond to some columns, then it is quite easy to exploit parallelism. Each Map task gets some of the columns and all the hash functions, and computes the minhash signature of its given columns. However, suppose the matrix were chunked by rows, so that a Map task is given the hash functions and a set of rows to work on. Design Map and Reduce functions to exploit mapreduce with data in this form.

Solution :

Assuming a set of random function of form : A*X+ B mod C. Here customized writable classes are implemented for convenience

- **Mapper** : It pops the hash value with each hash function for the document , if a particular shringle is present in the document.
- **Reducer** : It finds the min hash value with respect to each hash function for a given document.
- **Combiner** : Reducer itself can be used as a combiner reducing the workload at the reducer.

Listing 11: Mapper for finding MinHash Signatures

```
public static class MinHashMapper extends Mapper<LongWritable, Text,
    HashWritable, LongWritable> {
        private HashWritable hashKey = new HashWritable();
        private LongWritable longvalue = new LongWritable();
        public void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException {
                StringTokenizer tokenizer = new StringTokenizer(value.
                    toString()," ");
                int documentId = 0;
                long hashvalues[] = null;
                try{
                        hashvalues= computeHashValues(Integer.parseInt(
                            tokenizer.nextToken()));
                }catch(NumberFormatException e){
                        hashvalues= computeHashValues(0);
                }

                try{
                        while(tokenizer.hasMoreTokens()){
                                String token = tokenizer.nextToken();
                                if(Integer.parseInt(token)==1){
                                        for(int hashID=0;hashID<
                                            NUM_HASH_FUNCTION;hashID++)
                                            {
                                                hashKey.setDocumentId(
                                                    documentId);
                                                hashKey.setHashId(
                                                    hashID);
                                                longvalue.set(
                                                    hashvalues[hashID])
                                                    ;
                                                log.info(hashKey.
                                                    documentId+"--"+
                                                    hashKey.hashId+"=="
                                                    +longvalue.get());
                                                context.write(hashKey,
                                                    longvalue);
                                        }
                                }
                                documentId++;
                        }
                }catch(NumberFormatException e){

                }
        }
}
```

Listing 12: Reducer for Finding MinHash Signatures

```
public static class MinHashReducer extends Reducer<HashWritable,
    LongWritable, HashWritable, LongWritable> {
        private LongWritable longvalue = new LongWritable();
        public void reduce(HashWritable key, Iterable<LongWritable>
            values, Context context) throws IOException,
            InterruptedException {
                Long minvalue = Long.MAX_VALUE;
                for(LongWritable value : values){
                        if(value.get()<minvalue){
                                minvalue=value.get();
                        }
                }
                longvalue.set(minvalue);
                context.write(key, longvalue);
        }
}
```

4 Implementation : Suppose we wish to implement LSH by mapreduce. Specifically, assume chunks of the signature matrix consist of columns, and elements are keyvalue pairs where the key is the column number and the value is the signature itself (i.e., a vector of values). (a) Show how to produce the buckets for all the bands as output of a single mapreduce process. Hint : Remember that a Map function can produce several keyvalue pairs from a single element. (b) Show how another mapreduce process can convert the output of (a) to a list of pairs that need to be compared. Specifically, for each column i, there should be a list of those columns j $>$ i with which needs to be compared.

Solution :

Task 1: Finding the candidate pairs within bands.

- **Mapper** : It groups all documents which are similar within the band, i.e it emits document id for each bandid and hashvalue pair.
- **Reducer** : It concatenates all documents with same bandid and hashvalue pair.
- **Combiner** : Combiner can not be used here as the operation at reducer is not associative or commutative.

Listing 13: Mapper to compute Candidate Pairs within bands

```
public static class LSHMapper extends Mapper<LongWritable, Text,
    HashBandIdWritable,LongWritable> {
        private HashBandIdWritable outkey = new HashBandIdWritable();
```

```
        private LongWritable outvalue = new LongWritable();
        private long [] longarrvalues;
        private int numRows;

        @Override
        protected void setup(Context context) throws IOException,
                        InterruptedException {
                super.setup(context);
                this.numRows = context.getConfiguration().getInt(
                    PARAM_NUM_ROWS_PER_BLOCK, 5);
                this.longarrvalues = new long[numRows];
        }

        public void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException {
                StringTokenizer tokenizer = new StringTokenizer(value.
                    toString(),"\t");
                int hashId =0,bandId=0;
                if(tokenizer.hasMoreTokens()){
                        String token = tokenizer.nextToken();
                        try{
                                outvalue.set(Long.parseLong(token));
                        }catch(Exception e){
                                outvalue.set(Long.MAX_VALUE);
                        }
                }

                while(tokenizer.hasMoreTokens()){
                        String token = tokenizer.nextToken();
                        log.info(hashId);

                        try{
                                longarrvalues[hashId]=Long.parseLong(
                                    token);
                        }catch(NumberFormatException e){
                                longarrvalues[hashId]=0;
                        }

                        if(hashId==numRows-1){
                                outkey.setBandId(bandId);
                                outkey.setHashValue(Arrays.hashCode(
                                    longarrvalues));
                                context.write(outkey, outvalue);
                                hashId=0;
                                bandId++;
                        }else{
                                hashId++;
                        }
                }

        }
}
```

Listing 14: Reducer to compute Candidate Pairs within bands

```
public static class LSHReducer extends Reducer<HashBandIdWritable,
    LongWritable,HashBandIdWritable,Text> {
```

```
        private Text text = new Text();
        public void reduce(HashBandIdWritable key, Iterable<
            LongWritable> values, Context context) throws IOException,
            InterruptedException {
                StringBuilder builder = new StringBuilder();
                for(LongWritable value : values){
                        builder.append(value.get()).append("\t");
                }
                text.set(builder.toString());
                context.write(key,text);
        }
}
```

- **Task 2:** Finding the candidate pairs across bands.
- **Mapper** : It emits the candidate pairs acorss each bands, by emitting document id pair (i,j) such that i < j.
- **Reducer** : It removes the duplicate pairs.

Listing 15: Mapper to Compute Candidate Pairs Across Bands

```
public static class LSHMapper extends Mapper<LongWritable, Text,
    CandidatePairs,NullWritable> {
        private CandidatePairs outkey = new CandidatePairs();

        public void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException {
                StringTokenizer tokenizer = new StringTokenizer(value.
                    toString(),"\t");
                List<Long> docKeys = new ArrayList<Long>();
                //skipping first two columns
                if(tokenizer.hasMoreTokens()){
                        log.info(tokenizer.nextToken());
                }

                if(tokenizer.hasMoreTokens()){
                        log.info(tokenizer.nextToken());
                }

                while(tokenizer.hasMoreTokens()){
                        try{
                                long currKey = Long.parseLong(tokenizer
                                    .nextToken());
                                for(Long docKey:docKeys){
                                        if(currKey>docKey){
                                                outkey.setDocId1(docKey
                                                    );
                                                outkey.setDocId2(
                                                    currKey);
                                        }else{
                                                outkey.setDocId1(
                                                    currKey);
                                                outkey.setDocId2(docKey
                                                    );
                                        }
```

```
                                        context.write(outkey,
                                            NullWritable.get());
                                    }
                                    docKeys.add(currKey);
                                }catch(NumberFormatException e){

                                }
                            }

                        }
                }
```

Listing 16: Computing Candidate Pairs Across Bands

```
        public static class LSHReducer extends Reducer<CandidatePairs,
            NullWritable,CandidatePairs,NullWritable> {
                public void reduce(CandidatePairs key, Iterable<NullWritable>
                    values, Context context) throws IOException,
                    InterruptedException {
                        log.info("Part_2_reducer");
                        log.info(key);
                        context.write(key,NullWritable.get());
                }
        }
```

5 In Table 1 is a matrix with six rows.

Table 1: Input Matrix

Element	S1	S2	S3	S4
0	0	1	0	1
1	0	1	0	0
2	1	0	0	1
3	0	0	1	0
4	0	0	1	1
5	1	0	0	0

5.1 Compute the minhash signature for each column if we use the following three hash functions: h1(x) = 2x + 1 mod 6; h2(x) = 3x + 2 mod 6; h3(x) = 5x + 2 mod 6.

Solution :

- Initially if we compute hash values for each row we get as shown in Table 2,
- Using min-hash algorithm to compute minhas signatures.
- Table 3(g) corresponds the final minhash signature.

Table 2: Computing hash values

Element	S1	S2	S3	S4	2x+1 mod 6	3x+2 mod 6	5x+2 mod 6
0	0	1	0	1	1	2	2
1	0	1	0	0	3	5	1
2	1	0	0	1	5	2	0
3	0	0	1	0	1	5	5
4	0	0	1	1	3	2	4
5	1	0	0	0	5	5	3

Table 3: Computing minhash signature

(a) Initially

Element	S1	S2	S3	S4
h1	∞	∞	∞	∞
h2	∞	∞	∞	∞
h3	∞	∞	∞	∞

(b) Computing for Element 0

Element	S1	S2	S3	S4
h1	∞	1	∞	1
h2	∞	2	∞	2
h3	∞	2	∞	2

(c) Computing for Element 1

Element	S1	S2	S3	S4
h1	∞	1	∞	1
h2	∞	2	∞	2
h3	∞	1	∞	2

(d) Computing for Element 2

Element	S1	S2	S3	S4
h1	5	1	∞	1
h2	2	2	∞	2
h3	0	1	∞	0

(e) Computing for Element 3

Element	S1	S2	S3	S4
h1	5	1	1	1
h2	2	2	5	2
h3	0	1	5	0

(f) Computing for Element 4

Element	S1	S2	S3	S4
h1	5	1	1	1
h2	2	2	2	2
h3	0	1	4	0

(g) Computing for Element 5

Element	S1	S2	S3	S4
h1	5	1	1	1
h2	2	2	2	2
h3	0	1	4	0

5.2 Which of these hash functions are true permutations?

Solution : The hash function h3 = 5x + 2 mod 5 is a true permutation because it generates all possible hash values from 0 to 5 and 5is a co-prime

5.3 How close are the estimated Jaccard similarities for the six pairs of columns to the true Jaccard similarities?

Solution :

Table 4: Jacard Similarity for 6 pairs of documents

Jacard Similary	S1&S2	S1&S3	S1&S4	S2&S3	S2&S4	S3&S4
Estimated	1/3	1/3	2/3	2/3	2/3	2/3
Actual	0	0	1/4	0	1/4	1/4

6 One might expect that we could estimate the Jaccard similarity of columns without using all possible permutations of rows. For example, we could only allow cyclic permutations; i.e., start at a randomly chosen row r, which becomes the first in the order, followed by rows r + 1, r + 2, and so on, down to the last row, and then continuing with the first row, second row, and so on, down to row r - 1. There are only n such permutations if there are n rows. However, these permutations are not sufficient to estimate the Jaccard similarity correctly. Give an example of a two-column matrix where averaging over all the cyclic permutations does not give the Jaccard similarity.

Solution :

Table 5: Example for input with cyclic permuation not equal to Jaccard similarity

Element	S1	S2
a	0	0
b	1	1
c	0	1

Table 6: Finding min-hash signature for cyclic permutation starting at given position

Position	S1	S2
0	b	b
1	b	b
2	b	c

The Actual Jaccard's similarity is equal to 1/2 whereas the estimaed similarity equals 2/3.

7 For each of the (r, b) pairs in given below, compute the threshold, that is, the value of s for which the value of $1-(1-s^r)^b$ is exactly 1/2. How does this value compare with the estimate of $(1/b)^{1/r}$?

1. r = 3 and b = 10.
2. r = 6 and b = 20.
3. r = 5 and b = 50.

Solution :

Computing value of threshold similarity for the $1-(1-s^r)^b = 1/2$,

$$1-(1-s_{thalf}^r)^b = 1/2$$

$$(1-s_{thalf}^r) = (1/2)^{1/b}$$

$$s_{thalf}^r = 1-(1/2)^{1/b}$$

$$s_{thalf} = (1-(1/2)^{1/b})^{1/r}$$

Approximate threshold similarity where slope is steepest,

$$s_t = (1/b)^{1/r}$$

1. r = 3 and b = 10.

$$s_{thalf} = (1-(1/2)^{1/10})^{1/3}$$

$$s_{thalf} = 0.4061$$

$$s_t = (1/10)^{1/3}$$

$$s_t = 0.4642$$

2. r = 6 and b = 20.

$$s_{thalf} = (1-(1/2)^{1/20})^{1/6}$$

$$s_{thalf} = 0.5694$$

$$s_t = (1/20)^{1/6}$$

$$s_t = 0.6070$$

3. r = 5 and b = 50.

$$s_{thalf} = (1 - (1/2)^{1/50})^{1/5}$$

$$s_{thalf} = 0.4244$$

$$s_t = (1/50)^{1/5}$$

$$s_t = 0.4573$$

Steeper slope is always identified at a higher value of similarity compared to the similarity obtained $1 - (1 - s^r)^b = 1/2$.

8 Use the techniques explained in Section 1.3.5 to approximate the S-curve $1 - (1 - s^r)^b$ when s^r is very small.

Solution

Let us consider $\frac{1}{x} = s^r$,

$$1 - (1 - s^r)^b = 1 - (1 - \tfrac{1}{x})^{x(s^r b)}$$

Since value of s^r is very small, value of x be very large,limit as x goes to infinity of $(1 - \frac{1}{x})^x$ is 1/e.

$$1 - (1 - s^r)^b \approx 1 - (\tfrac{1}{e})^{s^r b} = 1 - e^{-s^r b}$$

when s^r is small and b is large.

The function for the probability of becoming a candidate pair approximates to a sigmoid function when value of b is very large. The slope of the s-curve can be controlled by the value of b.For higher values of b the S-curve would be steep. The Figure 1 shows the approximate function for r = 5 and s=50.

Figure 1: Plot of sigmoid function r = 5 and s= 50