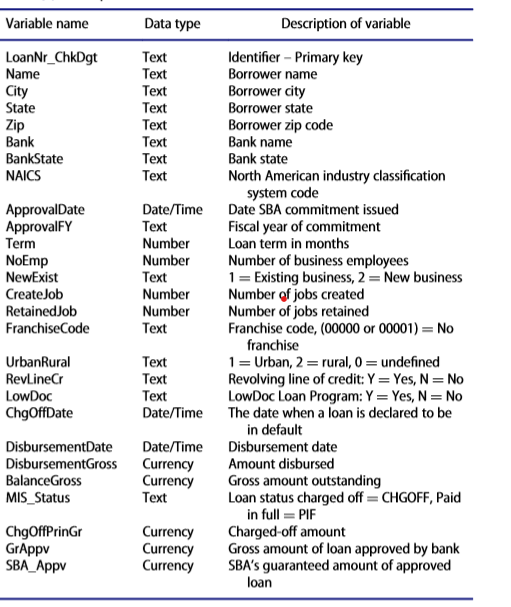
**Final Project**

**Background:**

The dataset is from the U.S. Small Business Administration. The organization was founded in 1953 on the principle of promoting and assisting small enterprises in the U.S. credit market. Researches show that small businesses have been an incubator of job creation in the United States; to that end, fueling small business growth has social beneﬁts by creating job opportunities and reducing unemployment. One-way SBA assists these small business enterprises is through a loan guarantee program which is designed to encourage banks to grant loans to small businesses. SBA acts much like an insurance provider to reduce the risk for a bank by taking on some of the risk through guaranteeing a portion of the loan. In the case that a loan goes into default, SBA then covers the amount they guaranteed. banks will incur some losses if a small business defaults on its SBA-guaranteed loan. The objective of this analysis is to determine the risk indicators for the probability of small businesses’ delinquency on loan from the dataset provided by SBA.

**Implication:**

The analysis is the importance bedrock for model development by data scientists to make good predictions for the likelihood of delinquency of small businesses.

**Dataset Description:**

This dataset is from the U.S. Small Business Administration (SBA) and provides historical data from 1987 through 2014. This large data set contains 27 variables and 899,164 observations. Each observation represents a loan that was guaranteed to some degree by the SBA. Included is a variable [MIS\_Status] which indicates if the loan was paid in full or defaulted/charged off.

**First Analysis**

**Problem Statement**: Find the statistic of loans issued (min, max, average, median) for each year.

**Summary of Implementation:**

1. Created a CustomWritable class that has count, average fields.
2. In the Mapper, instantiate a CustomWritable object and set the loan amount of each record to the Average field of the object. Then set the count field to 1. Mapper output key is the year and Mapper output value is the CustomWritable object.
3. In the reducer, declare a min variable with maximum value of Integer, a max, a sum, a count and an average variable with a value of 0. Median variable is declared Null. The next step is iterate over the array of values and perform series of if/else statement to find the maximum, minimum, sum and count. Then calculate the average. The last step using the Array list data structure with Collection.sort to determine the median value.

**Code Screenshots:**

public class CustomWritable implements Writable {  
   
 int count;  
 Double average;

………

………

}

public class MapperClass extends Mapper<LongWritable, Text, IntWritable, CustomWritable> {  
 CustomWritable loanInfo = new CustomWritable();  
 IntWritable yearIssued = new IntWritable();  
  
 @Override  
 protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
 String line = value.toString();  
 String[] record = line.split(",");  
 String year = record[9];  
 String loanAmount = record[25];

if (loanAmount != null && year != null && isNumeric(loanAmount) && isNumeric(year)) {  
 loanInfo.setAverage(Double.*parseDouble*(loanAmount));  
 loanInfo.setCount(1);  
 yearIssued.set(Integer.*parseInt*(year));  
 context.write(yearIssued, loanInfo);  
 }  
 }

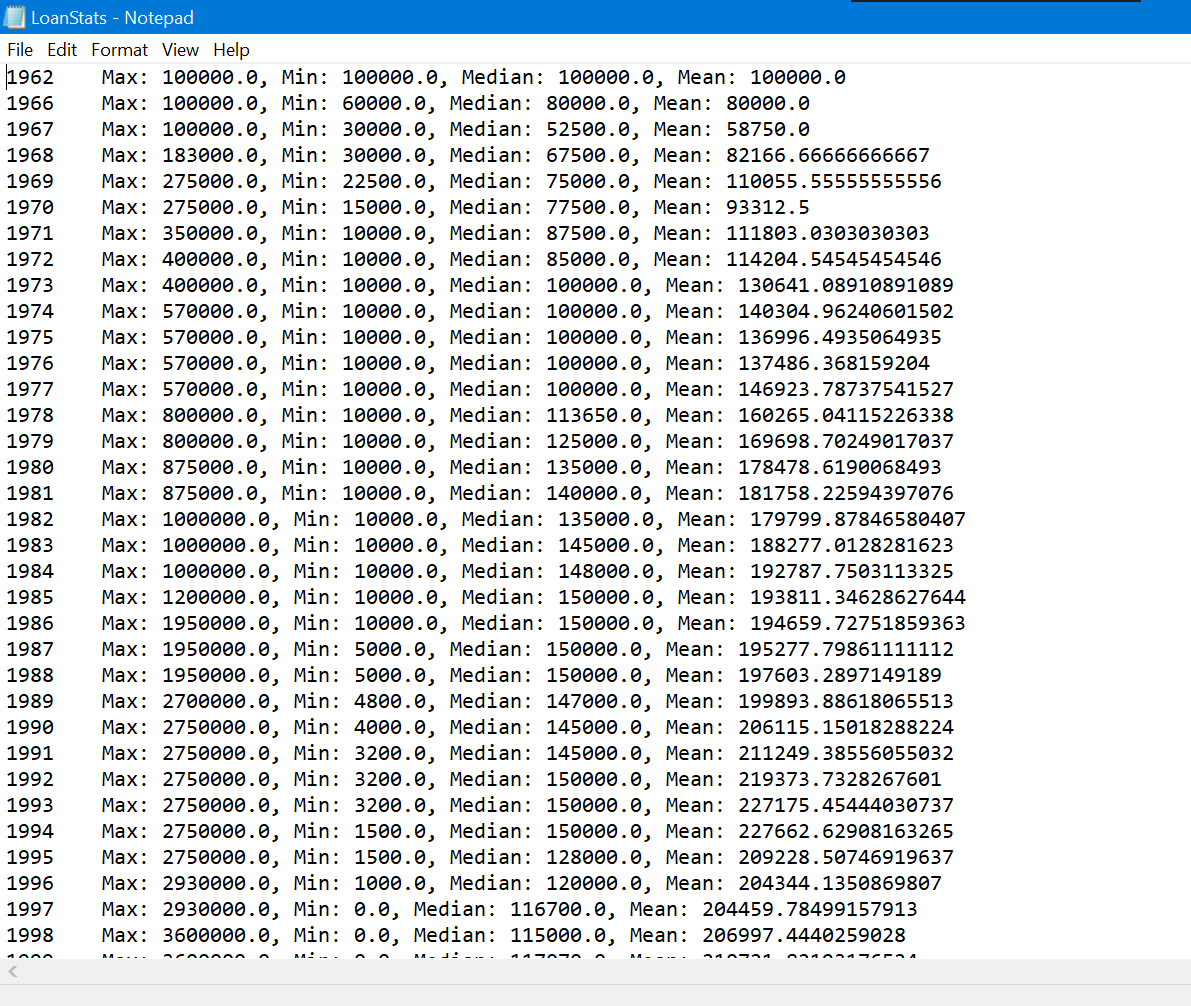
boolean isNumeric(String number) {  
 if (number == null) {  
 return false;  
 }  
 try {  
 Double.*parseDouble*(number);  
 return true;  
 } catch (Exception e) {  
 return false;  
  
 }  
 }  
}

public class ReducerClass extends Reducer<IntWritable, CustomWritable,IntWritable, Text> {  
 double min=Integer.*MAX\_VALUE*;  
 double max= 0;  
 double sum=0;  
 int count = 0;  
 double median;  
 double mean;

CustomWritable loanStat = new CustomWritable();  
 Text result = new Text();  
 ArrayList<Double> loans = new ArrayList();

@Override  
 protected void reduce(IntWritable key, Iterable<CustomWritable> values, Context context) throws IOException, InterruptedException {  
 for(CustomWritable current : values){  
 sum += current.getAverage();  
 count += current.getCount();  
 loans.add(current.getAverage());  
  
 if(current.getAverage()<min){  
 min=current.getAverage();  
 }  
 if(current.getAverage()>max){  
 max=current.getAverage();  
 }  
 }  
 mean = sum/count;  
 Collections.*sort*(loans);  
 if(loans.size()%2==0){  
 median = (loans.get((loans.size()-1)/2) +loans.get((loans.size()/2)))/2;  
 }  
 else {  
 median = loans.get(loans.size()/2);  
 }  
 result.set("Max: "+ max + ", Min: " + min + ", Median: "+ median + ", Mean: " + mean );  
 context.write(key,result);

Here is the output file.



**Second Analysis**

**Problem Statement**: Find the default rate grouped by state, city and industry in the accending order.

**Summary of Implementation:**

I perform two mapreduce program sequentially. The first mapreduce program will calculate the default rate by the common text key (State, City, Industry). Then the sorting in the acending order will be handled by the second mapreduce program.

**Code Screenshots:**

First mapreduce program:

public class CustomWritable implements Writable {  
 Double loanAmount;  
 Double chargeOffAmount;

public class Mapper extends org.apache.hadoop.mapreduce.Mapper<LongWritable, Text,Text, CustomWritable> {  
 CustomWritable ckw = new CustomWritable();  
 Text textKey = new Text();  
  
 @Override  
 protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
 String line = value.toString();  
 String[] record = line.split(",");  
 String city = record[2];  
 String state = record[3];  
 String industry = record[7];  
 String loanAmount = record[25];  
 String chargeOffAmount = record[24];

if(isNumeric(loanAmount) && isNumeric(chargeOffAmount)) {  
 textKey.set("State: "+ state + ", City: " + city + ", Industry: " + industry);  
 ckw.setLoanAmount(Double.*parseDouble*(loanAmount));  
 ckw.setChargeOffAmount(Double.*parseDouble*(chargeOffAmount));  
 context.write(textKey, ckw);  
 }  
 }  
  
 boolean isNumeric(String number) {  
 if (number == null) {  
 return false;  
 }  
 try {  
 Double.*parseDouble*(number);  
 return true;  
 } catch (Exception e) {  
 return false;  
  
 }  
 }

public class Reducer extends org.apache.hadoop.mapreduce.Reducer<Text, CustomWritable,Text, DoubleWritable> {  
 Double sumLoan=0.0;  
 Double sumChargeOff=0.0;  
 Double defaultRate;  
 DoubleWritable dw = new DoubleWritable();  
 @Override  
 protected void reduce(Text key, Iterable<CustomWritable> values, Context context) throws IOException, InterruptedException {  
 for(CustomWritable current : values){  
 sumLoan +=current.getLoanAmount();  
 sumChargeOff +=current.getChargeOffAmount();  
 }  
 defaultRate = sumChargeOff/sumLoan;  
 dw.set(defaultRate);  
 context.write(key,dw);  
 }  
}

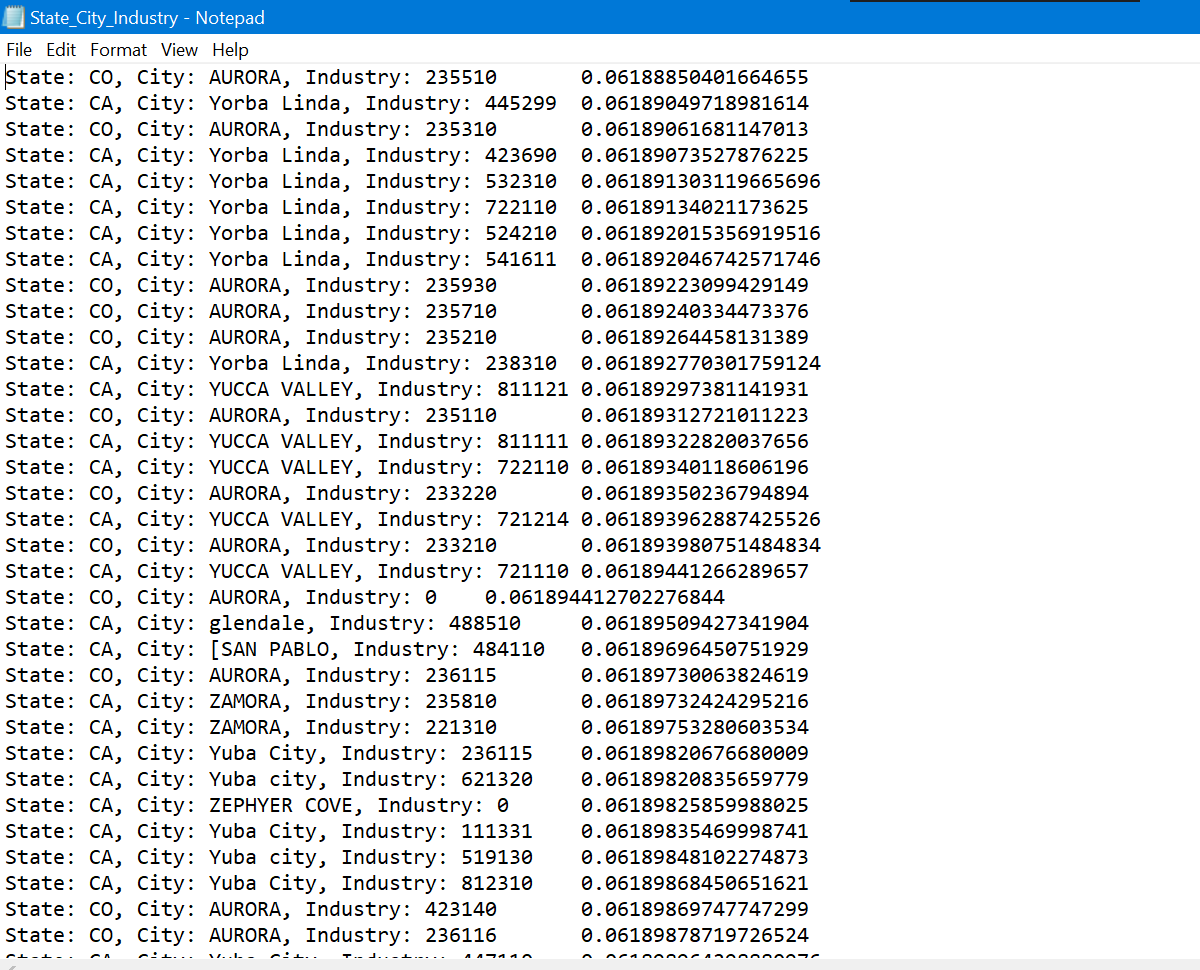
Second mapreduce program:

public class SortingMapper extends Mapper<LongWritable, Text, DoubleWritable, Text> {  
 DoubleWritable defaultRate = new DoubleWritable();  
 @Override  
 protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
 String line = value.toString();  
 String[] record = line.split("\t");  
 String default\_rate = record[record.length-1];

if(default\_rate !=null ){  
 defaultRate.set(Double.*parseDouble*(default\_rate));  
 context.write(defaultRate, value);  
 }  
 }  
}

public class SortingReducer extends Reducer<DoubleWritable, Text, NullWritable,Text> {  
 @Override  
 protected void reduce(DoubleWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException {  
 for(Text t : values) {  
 context.write(NullWritable.*get*(), t);  
 }  
 }  
}

Here is the output file:



**Third Analysis:**

**Problem Statement**: Find the top ten state which have the lowest default rates:

**Summary of Implementation:**

This analysis is performed by two mapreduce programs in sequence. The first mapreduce program will calculate the default rate by state. Then the second one will select the top ten which yield the lowest default rate using the treemap algorithm. The combiner is used to reduce the number of record transferred to a reducer.

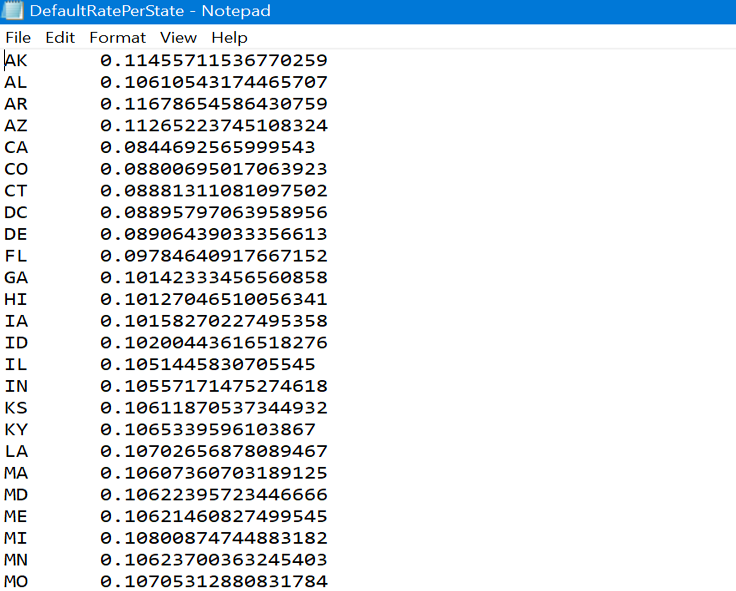
**Code Screenshots:**

First mapreduce program:

public class DefaultRateWritable implements Writable {  
 double loanAmount;  
 double chargeoffAmount;

public class DefaultRateMapperClass extends Mapper<LongWritable, Text, Text, DefaultRateWritable> {  
 DefaultRateWritable drw = new DefaultRateWritable();  
 Text stateKey = new Text();  
 @Override  
 protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
 String line = value.toString();  
 String[] record = line.split(",");  
 String loanStatus = record[23].trim();  
 double loanAmount = isNumeric(record[25])?Double.*parseDouble*(record[25]):0;  
 double chargeOffAmount = isNumeric(record[24])?Double.*parseDouble*(record[24]):0;  
  
 String state = (record[3] != null && record[3] !="")?record[3]:"Null";  
 stateKey.set(state);  
  
 if(loanStatus.equalsIgnoreCase("P I F")){  
 drw.setloanAmount(loanAmount);  
 drw.setchargeoffAmount(0);  
 context.write(stateKey,drw);  
 }  
 else if(loanStatus.equalsIgnoreCase("CHGOFF")) {  
 drw.setloanAmount(loanAmount);  
 drw.setChargeoffAmount(chargeOffAmount);  
 context.write(stateKey,drw);  
 }  
 }  
 boolean isNumeric(String number) {  
 if (number == null) {  
 return false;  
 }  
 try {  
 Double.*parseDouble*(number);  
 return true;  
 } catch (Exception e) {  
 return false;  
  
 }  
 }

public class DefaultRateReducerClass extends Reducer<Text, DefaultRateWritable,Text,Text> {  
 double sumLoan=0.0;  
 double sumChargeOff=0.0;  
 double defaultRate;  
 Text result = new Text();  
 @Override  
 protected void reduce(Text key, Iterable<DefaultRateWritable> values, Context context) throws IOException, InterruptedException {  
 for(DefaultRateWritable current: values){  
 sumLoan += current.amountPaid;  
 sumChargeOff += current.outstandingBalance;  
 }  
 defaultRate = sumChargeOff/sumLoan;  
 result.set(Double.*toString*(defaultRate));  
 context.write(key,result);

Output file of the first mapreduce program: 

Second mapreduce program:

public class TopTenMapper extends Mapper<LongWritable, Text, NullWritable,Text> {  
 TreeMap<Double,Text> top10 = new TreeMap<Double, Text>();  
  
 @Override  
 protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
 String line = value.toString();  
 String[] record = line.split("\t");  
 if(record.length ==2) {  
 Double defaultRate = new Double(Double.*parseDouble*(record[1]));//?  
  
 top10.put(defaultRate, value);  
 }  
 if (top10.size()>10){  
 top10.remove(top10.firstKey());  
 }  
 }  
  
 @Override  
 protected void cleanup(Context context) throws IOException, InterruptedException {  
 for(Text t : top10.values()){  
 context.write(NullWritable.*get*(),t);  
 }  
 }  
}

public class TopTenReducer extends Reducer<NullWritable, Text, NullWritable,Text> {  
 TreeMap<Double,Text> top10 = new TreeMap<Double,Text>() ;  
 @Override  
 protected void reduce(NullWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException {  
 for(Text value: values){  
 String line = value.toString();  
 String[] record = line.split("\t");  
 if(record.length ==2 ) {  
 double defaultRate = Double.*parseDouble*(record[1]);  
 top10.put(defaultRate, value);  
 }  
 if(top10.size()>10){  
 top10.decendingMap().remove(top10.firstKey());  
 }  
 }  
 }  
  
 @Override  
 protected void cleanup(Context context) throws IOException, InterruptedException {  
 for(Text t: top10.values()){ //?  
 context.write(NullWritable.*get*(),t);

public class DriverClass {  
 public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  
 Configuration conf =new Configuration();  
 Job job = Job.*getInstance*(conf,"DefaultRate");  
 job.setJarByClass(DriverClass.class);  
  
 FileInputFormat.*addInputPath*(job,new Path(args[0]));  
 FileOutputFormat.*setOutputPath*(job,new Path(args[1]));  
  
 job.setMapOutputValueClass(DefaultRateWritable.class);  
 job.setMapOutputKeyClass(Text.class);  
 job.setOutputKeyClass(Text.class);  
 job.setOutputValueClass(Text.class);  
  
 job.setMapperClass(DefaultRateMapperClass.class);  
 job.setReducerClass(DefaultRateReducerClass.class);  
  
  
 if (!job.waitForCompletion(true)) {  
 System.*exit*(1);  
 }  
  
  
 Job job2 = Job.*getInstance*(conf,"job2");  
 job2.setJarByClass(DriverClass.class);  
 job2.setMapperClass(TopTenMapper.class);  
 job2.setReducerClass(TopTenReducer.class);  
 job2.setNumReduceTasks(1);  
  
 job2.setMapOutputKeyClass(NullWritable.class);  
 job2.setMapOutputValueClass(Text.class);  
  
 job2.setOutputKeyClass(NullWritable.class);  
 job2.setOutputValueClass(Text.class);  
  
 FileInputFormat.*addInputPath*(job2,new Path(args[1] + "/part-r-00000"));  
 FileOutputFormat.*setOutputPath*(job2,new Path(args[2]));  
  
 System.*exit*(job2.waitForCompletion(true)?0:1);  
 }  
}

Here is the final output file:

