Smarter Email Marketing with the Markov Model

Markov Model?

State space

A finite set of states $S = \{S_1, S_2, S_3, \ldots\}$

Transition probabilities

A function f: $S \times S \to R$ such that:

- $0 \leq f(a, b) \leq 1$ for all $a, b \in S$
- $\sum_{b \in S} f(a, b) = 1$ for every $a \in S$

Initial distribution

A function g: $S \times R$ such that:

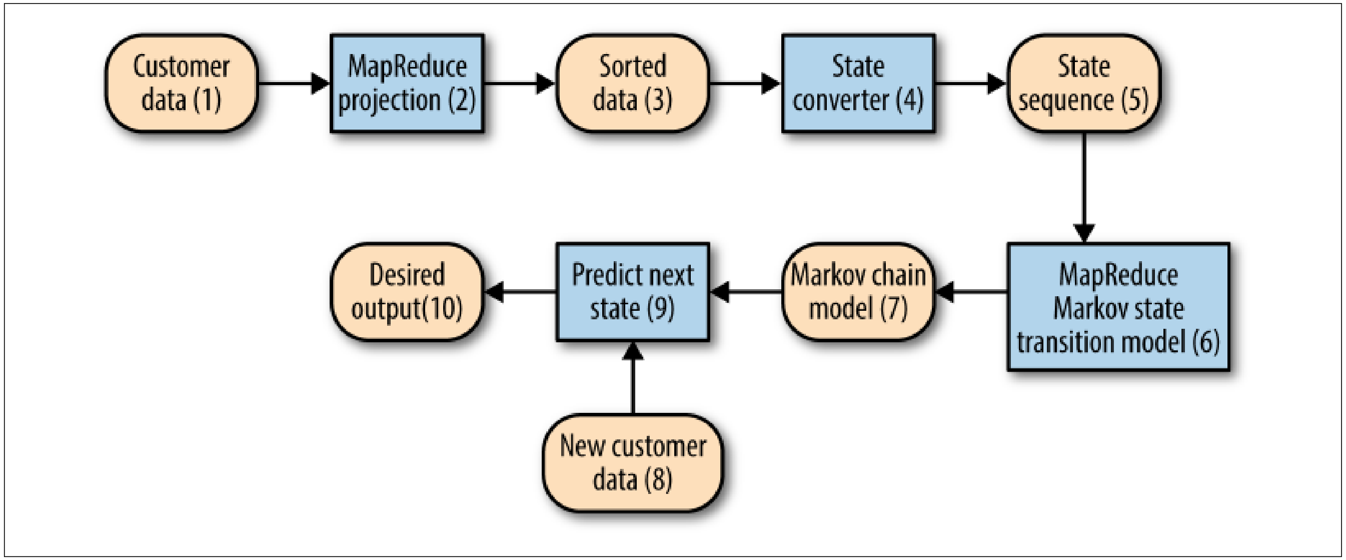
- $0 \leq g(a) \leq 1$ for every $a \in S$
- $\sum_{a \in S} g(a) = 1$

Today's weather	Tomorrow's weather	sunny	rainy	cloudy	foggy
`sunny`		0.6	0.1	0.2	0.1
`rainy`		0.5	0.2	0.2	0.1
`cloudy`		0.1	0.7	0.1	0.1
`foggy`		0.0	0.3	0.4	0.3

$$
\begin{aligned}
&P\left(S_2 = \texttt{cloudy}, S_3 = \texttt{foggy} \middle| S_1 = \texttt{sunny}\right)\\
&= P\left(S_3 = \texttt{foggy} \middle| S_2 = \texttt{cloudy}, S_1 = \texttt{sunny}\right) \times\\
&P\left(S_2 = \texttt{cloudy} \middle| S_1 = \texttt{sunny}\right)\\
&= P\left(S_3 = \texttt{foggy} \middle| S_2 = \texttt{cloudy}\right) \times\\
&P\left(S_2 = \texttt{cloudy} \middle| S_1 = \texttt{sunny}\right)\\
&= 0.1 \times 0.2\\
&= 0.02
\end{aligned}
$$

$$
\begin{aligned}
&P\left(S_3 = \texttt{foggy} \middle| S_1 = \texttt{foggy}\right) =\\
&P\left(S_3 = \texttt{foggy}, S_2 = \texttt{sunny} \middle| S_1 = \texttt{foggy}\right) +\\
&P\left(S_3 = \texttt{foggy}, S_2 = \texttt{cloudy} \middle| S_1 = \texttt{foggy}\right) +\\
&P\left(S_3 = \texttt{foggy}, S_2 = \texttt{rainy} \middle| S_1 = \texttt{foggy}\right) +\\
&P\left(S_3 = \texttt{foggy}, S_2 = \texttt{foggy} \middle| S_1 = \texttt{foggy}\right) +\\
&= P\left(S_3 = \texttt{foggy} \middle| S_2 = \texttt{sunny}\right) \times P\left(S_2 = \texttt{sunny} \middle| S_1 = \texttt{foggy}\right) +\\
&P\left(S_3 = \texttt{foggy} \middle| S_2 = \texttt{cloudy}\right) \times P\left(S_2 = \texttt{cloudy} \middle| S_1 = \texttt{foggy}\right) +\\
&P\left(S_3 = \texttt{foggy} \middle| S_2 = \texttt{rainy}\right) \times P\left(S_2 = \texttt{rainy} \middle| S_1 = \texttt{foggy}\right) +\\
&P\left(S_3 = \texttt{foggy} \middle| S_2 = \texttt{foggy}\right) \times P\left(S_2 = \texttt{foggy} \middle| S_1 = \texttt{foggy}\right)\\
&= 0.1 \times 0.0 +\\
&0.1 \times 0.4 +\\
&0.1 \times 0.3 +\\
&0.3 \times 0.3\\
&= 0.00 + 0.04 + 0.03 + 0.09\\
&= 0.16
\end{aligned}
$$

```
<customerID><,><transactionID><,><purchaseDate><,><amount>
```



Generating Time-Ordered Transactions with MapReduce

- customerID (Date1, Amount1);(Date2, Amount2);...(Date*N*, Amount*N*)
- Date1 ≤ Date2 ≤ ... ≤ Date*N*

- 시간 순으로 정렬 방법
  1. Reducer를 이용한 정렬
  2. Secondary sort를 이용한 정렬

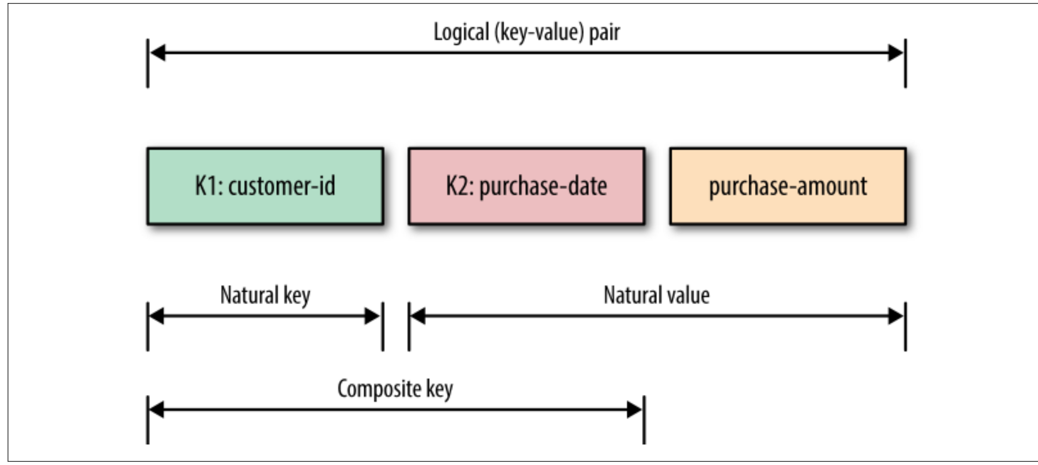
Hadoop Solution 1: Time-Ordered Transactions

`(purchase-date, amount)`

Class name	Description
`SortInMemoryProjectionDriver`	Driver class to submit jobs
`SortInMemoryProjectionMapper`	Mapper class
`SortInMemoryProjectionReducer`	Reducer class
`DateUtil`	Basic date utility class
`HadoopUtil`	Basic Hadoop utility class

Hadoop Solution 2: Time-Ordered Transactions

Key:(customer-id, purchase-date)
Value: (purchase-date, amount)



Class name	Description
`SecondarySortProjectionDriver`	Driver class to submit jobs
`SecondarySortProjectionMapper`	Mapper class
`SecondarySortProjectionReducer`	Reducer class
`CompositeKey`	Custom key to hold a pair of (`customer-id, purchase-date`), which is a combination of the natural key and the natural value we want to sort by
`CompositeKeyComparator`	How to sort `CompositeKey` objects; compares two composite keys for sorting
`NaturalKeyGroupingComparator`	Considers the natural key; makes sure that a single reducer sees a custom view of the groups (how to group `customer-id`)
`NaturalKeyPartitioner`	How to partition by the natural key (`customer-id`) to reducers; blocks all data into a logical group in which we want the secondary sort to occur on the natural value
`DateUtil`	Basic date utility class
`HadoopUtil`	Basic Hadoop utility class

Generating State Sequences

```
customer-id, State_1, State_2, ..., State_n
```

$\texttt{customer-id, State}_1\texttt{, State}_2\texttt{, ..., State}_n$

$\texttt{customer-id (Date}_1\texttt{, Amount}_1\texttt{);(Date}_2\texttt{, Amount}_2\texttt{);...(Date}_N\texttt{, Amount}_N\texttt{)}$

Time elapsed since last transaction	Amount spent compared to previous transaction
S: Small	L: Significantly less than
M: Medium	E: More or less same
L: Large	G: Significantly greater than

State name	Time elapsed since last transaction: amount spent compared to previous transaction
SL	Small: significantly less than
SE	Small: more or less same
SG	Small: significantly greater than
ML	Medium: significantly less than
ME	Medium: more or less same
MG	Medium: significantly greater than
LL	Large: significantly less than
LE	Large: more or less same
LG	Large: significantly greater than

```
$ cat transaction_sequence.txt
00VVD1E210,2012-06-18,87
00W6TWFW4S,2012-03-24,22,2012-05-22,80,2012-06-15,33
00W86Y0GFT,2012-02-15,141,2012-03-10,30,2012-03-25,49,2012-05-17,107
00W92K8A1W,2012-04-19,25
00W9W3Y3XH,2012-03-25,123
00XL1QERUO,2012-01-07,81,2012-05-10,154
00XPR1XW1P,2012-04-26,103
00Y1B0Y4CO,2012-03-10,81
00YR97DWWO,2012-07-15,118
00Z5SOHKED,2012-01-28,43,2012-02-25,27
00ZLLMHKND,2012-02-21,185,2012-04-02,63,2012-04-03,30

$ ./xaction_state.rb transaction_sequence.txt
00W6TWFW4S,ML,SG
00W86Y0GFT,SG,SL,ML
00XL1QERUO,LL
00Z5SOHKED,SG
00ZLLMHKND,MG,SG
```

Generating a Markov State Transition Matrix with MapReduce

9*9 transition matrix

Example 11-4. Markov state transition: map() function

```
/**
 * @param key is the Customer-ID, ignored
 * @param value is the sequence of states = {S1, S2, ..., Sn}
 * We assume value is an array of n states (indexed from 0 to n-1).
 */
map(key, value) {
    for (i=0, i < n-1, i++) {
       // value[i] denotes "from state"
       // value[i+1] denotes "to state"
       reducerKey = pair(value[i], value[i+1]);
       emit(reducerKey, 1);
    }
}
```

Example 11-5 defines the `combine()` function for our Markov state transition.

Example 11-5. Markov state transition: combine() function

```
/**
 * @param key is a Pair(state1, state2)
 * @param value is a list of integers (partial count of "state1" to "state2")
 */
combine(Pair(state1, state2) key, List<integer> values) {
    int partialSum = 0;
    for (int count : values) {
        partialSum += count;
    }
    emit(key, partialSum);
}
```

Example 11-6. Markov state transition: reduce() function

```
/**
 * @param key is a Pair(state1, state2)
 * @param value is a list of integers (partial count of "state1" to "
 */
reduce(Pair(state1, state2) key, List<integer> value) {
    int sum = 0;
    for (int count : value) {
        sum += count;
    }
    emit(key, sum);
}
```

```
# hadoop fs -cat /markov/state_transition_model/input/state_seq.txt | head
000IA1PHVZ,SG,SL,SG,SL,ML,MG,SG,SL,SG,SL,ML
000KH3DK15,SG,SL,SG,ML,SG,SL,SG,SL,SG,SL,SG,ML,SG,SL,SG
001KD25DTD,SG,SL,SG,SL,SG,SL,SG
00241F24T4,SG,SL,SG,SL,SG,SL,SG,ML,SG,SL,ML,SG,ML,SG
002C11GB8Y,SG,SL,SG,SL,SG,SL,SG,ML,SG,SL,SG,ML,SG
002SG5SKJT,SG,SL,SG,ML,SG,SL,SG
0030B44HD0,SG,SL,SG,SL,SG,SL,SG,SL,ML,SG
004ADRKOEW,SG,SL,SG,ML,MG,SG,SL,SG,LL
004MT1M5BY,SG,SL,SG,SL,SG,ML,SG,ML,SG,SL,ML
007DI3WJ5B,SL,SL,ML,MG,SG,SE,SL,SG,SG,SL,SG
```

```
# hadoop fs -cat /markov/state_transition_model/output/part*
LL,MG 2990
ME,SG 172
MG,LL 803
...
SL,SE 2099
LE,LG 2
LG,LE 1
MG,SG 19485
ML,SL 268
...
SL,ME 151
LG,SL 510
LL,SG 17062
...
SG,SG 5090
SL,SL 2772
```

Using the Markov Model to Predict the Next Smart Email Marketing Date

```
# cat StateTransitionTableBuilder.java
...
public class StateTransitionTableBuilder {
    ...
   public static void main(String[] args) {
      String hdfsDirectory = args[0];
      generateStateTransitionTable(hdfsDirectory);
   }
}

# export hdfsDir="/markov/state_transition_model/output"
# java StateTransitionTableBuilder $hdfsDir > model.txt
```

```
# Generate validation data
# ----------------------
./buy_xaction.rb 80000 30 .05 > validation.txt
head validation.txt
XURQDBEHME,1385141945,2013-01-01,98
3RT4PONSUP,1385141946,2013-01-01,53
4NYCEUD3YG,1385141947,2013-01-01,164
SF9KAY8F42,1385141948,2013-01-01,204
LKNCID1DRV,1385141949,2013-01-01,83
4EZJDVB4W1,1385141950,2013-01-01,116
ITJ39B3NX3,1385141951,2013-01-01,72
D8VVPAHG8I,1385141952,2013-01-01,124
21XHZJY561,1385141953,2013-01-01,103
F7LS37R08X,1385141954,2013-01-01,211
```

```
# Predict email marketing time
# ---------------------------
./mark_plan.rb validation.txt model.txt
XURQDBEHME, 2013-04-27
4NYCEUD3YG, 2013-04-14
SF9KAY8F42, 2013-04-07
LKNCID1DRV, 2013-04-30
4EZJDVB4W1, 2013-02-02
ITJ39B3NX3, 2013-04-27
D8VVPAHG8I, 2013-04-29
21XHZJY561, 2013-01-18
F7LS37R08X, 2013-02-14
...
```

Spark Solution

1. Handle input parameters.
2. Create a context object and convert the input into a JavaRDD<String>.
3. Convert the JavaRDD<String> into a JavaPairRDD<K,V>,
   - where K is a customerID and V is a Tuple2<purchaseDate, amount>.
4. Group transactions by customerID.
   - groupByKey() : the output of step 3,
   - result is a JavaPairRDD<K2,V2>, K2 = customerID , V2 = Iterable<Tuple2<purchaseDate, Amount>>.
5. Create a Markov state sequence:

   State$_1$, State$_2$, ..., State$_N$
   - mapValues() transformation to the JavaPairRDD<K2,V2> and generate a JavaPairRDD<K4, V4>.
   - (K2, V2) => (K3, V3) K2 = K3 = K4 = customerID, V3 = sorted V2 (order is based on purchaseDate)
   - we use V3 to create a Markov state sequence (as V4).
6. Generate a Markov state transition with the following input/output:
   1. *Input* : JavaPairRDD<K4, V4> pairs.
   2. *Output :* A matrix of states {S1, S2, S3, ...}
7. Emit the final output.

Step 1: Handle input parameters

```java
// Step 1: handle input parameters
if (args.length != 1) {
    System.err.println("Usage: SparkMarkov <input-path>");
    System.exit(1);
}
final String inputPath = args[0];
System.out.println("inputPath:args[0]="+args[0]);
```

Step 2: Create Spark context object and convert Input into RDD

```
// Step 2: create Spark context object (ctx) and convert input into
//         JavaRDD<String>,
// where each element is an input record
JavaSparkContext ctx = new JavaSparkContext();
JavaRDD<String> records = ctx.textFile(inputPath, 1);
records.saveAsTextFile("/output/2");

// You may optionally partition RDD
// public JavaRDD<T> coalesce(int N)
// Return a new RDD that is reduced into N partitions.
// JavaRDD<String> records = ctx.textFile(inputPath, 1).coalesce(9);
```

```
$ hadoop fs -cat /output/2/part* | head -3
V31E55G4FI,1381872898,2013-01-01,123
301UNH7I2F,1381872899,2013-01-01,148
PP2KVIR4LD,1381872900,2013-01-01,163
```

Step 3: Convert RDD into JavaPairRDD

```
// Step 3: convert JavaRDD<String> into JavaPairRDD<K,V>, where
//    K: customerID
//    V: Tuple2<purchaseDate, Amount> : Tuple2<Long, Integer>
//    PairFunction<T, K, V>
//    T => Tuple2<K, V>
JavaPairRDD<String, Tuple2<Long,Integer>> kv = records.mapToPair(
    new PairFunction<
                    String,              // T
                    String,              // K
                    Tuple2<Long,Integer> // V
                   >() {
    public Tuple2<String,Tuple2<Long,Integer>> call(String rec) {
        String[] tokens = StringUtils.split(rec, ",");
        if (tokens.length != 4) {
        // not a proper format
        return null;
        }
        // tokens[0] = customer-id
        // tokens[1] = transaction-id
        // tokens[2] = purchase-date
        // tokens[3] = amount
        long date = 0;
        try {
            date = DateUtil.getDateAsMilliSeconds(tokens[2]);
        }
        catch(Exception e) {
            // ignore for now -- must be handled
        }
        int amount = Integer.parseInt(tokens[3]);
        Tuple2<Long,Integer> V = new Tuple2<Long,Integer>(date, amount);
        return new Tuple2<String,Tuple2<Long,Integer>>(tokens[0], V);
    }
});
kv.saveAsTextFile("/output/3");
```

```
$ hadoop fs -cat /output/3/part* | head
(V31E55G4FI,(1357027200000,123))
(301UNH7I2F,(1357027200000,148))
(PP2KVIR4LD,(1357027200000,163))
(AC57MM3WNV,(1357027200000,188))
(BN020INHUM,(1357027200000,116))
(UP8R2SOR77,(1357027200000,183))
(VD91210MGH,(1357027200000,204))
(COI4OXHET1,(1357027200000,78))
(76S34ZE89C,(1357027200000,105))
(6K3SNF2EG1,(1357027200000,214))
```

Step 4: Group transactions by customerID

```
// Step 4: group transactions by customerID. Apply groupByKey()
//         to the output of step 2; result will be
//         JavaPairRDD<K2,V2>, where
//           K2: customerID
//           V2: Iterable<Tuple2<purchaseDate, Amount>>
JavaPairRDD<String, Iterable<Tuple2<Long,Integer>>> customerRDD =
    kv.groupByKey();
customerRDD.saveAsTextFile("/output/4");
```

```
$ hadoop fs -cat /output/4/part* | head -3
(0IROUCA5O2,[(1361347200000,86), (1362643200000,30), (1362816000000,45),
             (1364886000000,27), (1366009200000,40), (1366182000000,28),
             (1369724400000,115), (1370502000000,32), (1371970800000,42),
             (1372575600000,32), (1374649200000,43)])
(4N0B1U5HVG,[(1358668800000,81), (1359446400000,33), (1363071600000,98),
             (1365750000000,50), (1366614000000,29), (1367218800000,48),
             (1369378800000,30), (1369810800000,41), (1370674800000,28),
             (1373353200000,107)])
(3KJR1907D9,[(1361088000000,105), (1362211200000,26), (1366182000000,103),
             (1366182000000,28), (1370415600000,111), (1373266800000,61),
             (1373439600000,34)])
```

Step 5: Create a Markov state sequence

```java
// Step 5: Create Markov state sequence: State1, State2, ..., StateN. Apply
//         mapValues() to JavaPairRDD<K2,V2> and generate JavaPairRDD<K4, V4>.
//         First convert (K2, V2) into (K3, V3) pairs [K2 = K3 = K4].
//         V3 is a sorted V2 (order is based on purchaseDate);
//         i.e., a sorted transaction sequence.
//         Then use V3 to create Markov state sequence (as V4).
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// Pass each value in the key-value pair RDD through a map function without
// changing the keys; this also retains the original RDD's partitioning.
JavaPairRDD<String, List<String>> stateSequence = customerRDD.mapValues(
    new Function<
                Iterable<Tuple2<Long,Integer>>, // input
                List<String>                    // output ("state sequence")
               >() {
    public List<String> call(Iterable<Tuple2<Long,Integer>> dateAndAmount) {
        List<Tuple2<Long,Integer>> list = toList(dateAndAmount);
        Collections.sort(list, TupleComparatorAscending.INSTANCE);
        // now convert sorted list (by date) into a state sequence
        List<String> stateSequence = toStateSequence(list);
        return stateSequence;
    }
});
stateSequence.saveAsTextFile("/output/5");
```

```
$ hadoop fs -cat /output/5/part* | head
(0IROUCA5O2,[SG, SL, SG, SL, SG, ML, SG, SL, SG, SL])
(4N0B1U5HVG,[SG, ML, MG, SG, SL, SG, SL, SG, ML])
(3KJR1907D9,[SG, ML, SG, ML, MG, SG])
(8555DQOK14,[SG, ML, LL])
(J6VXOTY7IA,[SG, ML, SG, SL, SG, ML, SG])
(T29M0VFTO4,[SG, SL, SG, SL, ML, SG, SL, SG, SL, SG, SL, SG, SL, SG])
(J0BO64O93C,[SG, SL, SG, SL, ML, SG, SG, SL, SG, SL, SG, SL, ML, SG, SL])
(NT58RT7KK4,[MG, SG, SL, SG, SL, SG, SL, SG, SL, SG])
(HBD6YAC69Y,[SG, SL, SG, SL, SG, SL, SG, SL, ML, MG])
(1BNFI5D3Z1,[SG, SL, SG, SL, SG, SL, SG, SL])
```

Step 6: Generate a Markov state transition matrix

```java
JavaPairRDD<Tuple2<String,String>, Integer> model = stateSequence.flatMapToPair(
    new PairFlatMapFunction<
                            Tuple2<String, List<String>>, // T
                            Tuple2<String,String>,        // K
                            Integer                       // V
                           >() {
    public Iterable<Tuple2<Tuple2<String,String>, Integer>>
        call(Tuple2<String, List<String>> s) {
        List<String> states = s._2;
        if ( (states == null) || (states.size() < 2) ) {
            return Collections.emptyList();
        }

        List<Tuple2<Tuple2<String,String>, Integer>> mapperOutput =
            new ArrayList<Tuple2<Tuple2<String,String>, Integer>>();
        for (int i = 0; i < (states.size() -1); i++) {
            String fromState = states.get(i);
            String toState = states.get(i+1);
            Tuple2<String,String> k = new Tuple2<String,String>(fromState,
                                                                 toState);
            mapperOutput.add(new Tuple2<Tuple2<String,String>, Integer>(k, 1));
        }
        return mapperOutput;
    }
});
model.saveAsTextFile("/output/6.1");
```

```
$ hadoop fs -cat /output/6.1/part* | head
((SG,SL),1)
((SL,SG),1)
((SG,SL),1)
((SL,SG),1)
((SG,ML),1)
((ML,SG),1)
((SG,SL),1)
((SL,SG),1)
((SG,SL),1)
((SG,ML),1)
```

```java
// combine/reduce frequent patterns (fromState, toState)
JavaPairRDD<Tuple2<String,String>, Integer> markovModel =
    model.reduceByKey(new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer i1, Integer i2) {
        return i1 + i2;
    }
});
markovModel.saveAsTextFile("/output/6.2");
```

```
$ hadoop fs -cat /output/6.2/part*
((SL,LL),7890)
((SG,LL),11140)
...
((MG,SG),19769)
((LL,MG),2885)
...
((SG,SL),254532)
((SG,ML),50112)
...
((ML,LL),2450)
((ML,SG),66275)
```

Step 7: Emit final output

```
// Step 7: emit final output
// convert markovModel into "<fromState><,><toState><TAB><count>"
// Use map() to convert JavaPairRDD into JavaRDD:
// <R> JavaRDD<R> map(Function<T,R> f)
// Return a new RDD by applying a function to all elements of this RDD.
JavaRDD<String> markovModelFormatted = markovModel.map(
    new Function<Tuple2<Tuple2<String,String>, Integer>, String>() {
    public String call(Tuple2<Tuple2<String,String>, Integer> t) {
        return t._1._1 + "," + t._1._2 + "\t" + t._2;
    }
});
markovModelFormatted.saveAsTextFile("/output/6.3");
```

```
$ export hdfsDir=/output/6.3
$ java org.dataalgorithms.chap11.statemodel.ReadDataFromHDFS $hdfsDir
INFO : path=hdfs://hnode01319.nextbiosystem.net:8020/output/6.3/part-00000
INFO : line=SL,LL 7890
INFO : line=SL,MG 209
INFO : line=SG,LL 11140
...
INFO : line=ML,LL 2450
INFO : line=ML,SG 66275
INFO : list=[{SL,LL,7890},
             {SL,MG,209},
             {SG,LL,11140},

             ...,
             {ML,LL,2450},
             {ML,SG,66275}]
```

Helper method

Example 11-16. toList() method

```
static List<Tuple2<Long,Integer>> toList(Iterable<Tuple2<Long,Integer>> iterable) {
    List<Tuple2<Long,Integer>> list = new ArrayList<Tuple2<Long,Integer>>();
    for (Tuple2<Long,Integer> element: iterable) {
        list.add(element);
    }
    return list;
}
```

Example 11-18. Comparator class

```
static class TupleComparatorAscending implements
   Comparator<Tuple2<Long, Integer>>, Serializable {
   final static TupleComparatorAscending INSTANCE = new TupleComparatorAscending();
   public int compare(Tuple2<Long, Integer> t1, Tuple2<Long, Integer> t2) {
        // return -t1._1.compareTo(t2._1);     // sorts RDD elements descending
        return t1._1.compareTo(t2._1);          // sorts RDD elements ascending
   }
}
```

Example 11-17. toStateSequence() method

```
/**
 * @param list : List<Tuple2<Date,Amount>>
 * list = [T2(Date1,Amount1), T2(Date2,Amount2), ..., T2(DateN,AmountN)
 * where Date1 <= Date2 <= ... <= DateN
 */
static List<String> toStateSequence(List<Tuple2<Long,Integer>> list) {
   if (list.size() < 2) {
     // not enough data
     return null;
   }
   List<String> stateSequence = new ArrayList<String>();
   Tuple2<Long,Integer> prior = list.get(0);
   for (int i = 1; i < list.size(); i++) {
      Tuple2<Long,Integer> current = list.get(i);

      long priorDate = prior._1;
      long date = current._1;
      // one day = 24*60*60*1000 = 86400000 milliseconds
      long daysDiff = (date - priorDate) / 86400000;

      int priorAmount = prior._2;
      int amount = current._2;
      int amountDiff = amount - priorAmount;

      String dd = null;
      if (daysDiff < 30) {
         dd = "S";
      }
      else if (daysDiff < 60) {
         dd = "M";
      }
      else {
        dd = "L";
      }

      String ad = null;
      if (priorAmount < 0.9 * amount) {
         ad = "L";
      }
      else if (priorAmount < 1.1 * amount) {
         ad = "E";
      }
      else {
        ad = "G";
      }

      String element = dd + ad;
      stateSequence.add(element);
      prior = current;
   }
```