Artificial Intelligence and Data Engineering

Cloud Computing - **Frodo** group

IMD Rating (Bloom Filter)
Hadoop, Spark implementation

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them.

The Lord of the Rings - **J. R. R. Tolkien**

Simone ***Bensi***
Paolo ***Falsini***
Tommaso ***Nocchi***

IMDb database

The file `title.ratings/data.tsv`[1] contains more than one million records, with the following fields:

`tconst`	**string**: alphanumeric unique identifier of the title.
`averageRating`	**double**: weighted average of all the individual user ratings.
`numVotes`	**int**: number of votes the title has received.

We have 10 rounded rating values, and we must compute 10 different bloom filters with the same false positive rate. In both Hadoop and Spark implementation we first build the Bloom filter, then we have a further step to test the resulting false positive rate.

In some situations, for example when you don't have access to the database, you can use the Bloom Filter to check if a title code has a requested rating average. Accessing to the Bloom Filter is quick and doesn't require too much memory, it can be done in the browser interface too. The price you must pay is a False positive rate: that's why we need to correctly calculate the appropriate Bloom Filter and then check it.

Bloom filter

A bloom filter can be seen as a classifier used to predict if an object is a member of a set. Given an element, the classifier prediction can be:

1	the element belongs to the set.
0	the element doesn't belong to the set.

The bloom filter classifier is trained adding the object which really belong to the set, after that phase the bloom filter can be used as a classifier. The bloom filter prediction will never consist in a false negative. To design a bloom filter for the titles rating (with a given false positive rate p), we need to estimate the number of keys n to be added to the bloom filter.

Bloom Filter Java class

The Bloom Filter corresponds to the java class `BloomFilter` (internally, each bit array is implemented using an array of WORD (32 bits)). The class exports the following methods:

- `public BloomFilter (int n_itemNumber, double p_falsePositiveRate)`
  creates an empty object to store `n_itemNumber` items and manage a false positive rate equal to `p_falsePositiveRate`
- `public void addItem(Text item)`
  add the item `item` to the `BloomFilter`
- `public boolean checkItem(Text item)`
  check if the title `item` in in the `BloomFilter`
- `public String getString()`
  return a string representing the `BloomFilter` [2]
- `public void setString(String integerList)`
  initialize the `Bloom Filter` with a previously saved string (see foot note)

[1] We manually removed the header line.
[2] The string representing the Bloom Filter is obtained converting each internal word into a string, then a `;` is used as separator.

Bloom Filter Python class[3]

The Bloom Filter corresponds to the python class `BloomFilter` (internally, each bit array is implemented using an array of bits. The class exports the following methods:

- `BloomFilter (n_itemNumber, p_falsePositiveRate)`
  creates an empty object to store `n_itemNumber` items and manage a false positive rate equal to `p_falsePositiveRate`
- `addItem(item)`
  add the item `item` to the `BloomFilter`
- `checkItem(item)`
  check if the title `item` in in the `BloomFilter`

[3] In order to obtain the same rounding behavior of Java we redefined the round procedure used in Python.

Hadoop Map-Reduce

In the Hadoop implementation we use three Map–Reduce jobs, coded in a single class `IMDRating`. The jar command is reported:

```
hadoop jar target/IMDRating-1.0-SNAPSHOT.jar IMDRating \
0.01 data.tsv count bloom check
```

The requested false positive rate is passed as the first parameter. The job configuration is used to make this value available in all the Map-Reduce stages.

```
Double fpr = (new Double(args[0]));
jobConfiguration.setDouble("fpr",fpr);
```

IMDRating count stage

To define the size of each Bloom Filter we must know the number of titles for each average rating, we design a Map-Reduce stage just for this.

```
IMDRating COUNT

Procedure MAP (Text t, Average a, numVotes v)
ra = round(a)
EMIT (ra (as Text), i = 1 (as Int))

Procedure REDUCER (ra a, Links [i₁,i₂, …iₘ])
for all n in Links do:
    count += 1
EMIT(a (as Text), count (as Int))
```

The (key, values) pairs of the Map–Reduce job have the following contents

Mapper input			Mapper output		Reducer input		Reduced output	
tt0000001	5.7	1882	06	1	01	1,1,1,...	01	2552
tt0000002	5.9	250	06	1	02	1,1,1,...	02	6640
tt0000003	6.5	1663	06	1	03	1,1,1,...	03	17817
tt0000004	9.8	163	10	1	04	1,1,1,...	04	43578
tt0000005	2.2	2487	02	1	05	1,1,1,...	05	102451
...			...		...		...	

The output file of this stage is a list of ten pairs (average rate, number of titles). This file is read in the main class and those pairs are stored in the job configuration to be used in the two subsequent stages.

```
while((line = br.readLine()) != null) {
    String[] tokens = line.split("\t");
    jobConfiguration.setInt(tokens[0], Integer.parseInt(tokens[1]));
}
```

IMDRating bloom stage

For every possible rating we build the Bloom Filter of the appropriate size, then we add all the selected titles to the Bloom filter. The output file of this stage is a list of 10 pairs (*average rating*, *string representing the Bloom Filter*).

```
IMDRating BLOOM

Procedure MAP (Text t, Average a, numVotes v)
ra = round(a)
EMIT (ra (as Text), t (as Text))

Procedure REDUCER (ra a, Links [t1,t2, …tm])
Bloom-Filter-Create (n_itemNumber, p_falsePositiveRate)
for all t in Links do:
    Bloom-Filter-AddItem(t)
EMIT(a (as Text), Bloom Filter (as Text))
```

The (key, values) pairs of the Map–Reduce job have the following contents

Mapper input			Mapper output		Reducer input		Reduced output	
tt0000001	5.7	1882	06	tt0000001	01	tt0000061,	01	BF(1)
tt0000002	5.9	250	06	tt0000002		tt0012001,	02	BF(2)
tt0000003	6.5	1663	06	tt0000003		tt0000401,	03	BF(3)
tt0000004	9.8	163	10	tt0000004		...	04	BF(4)
tt0000005	2.2	2487	02	tt0000005	02	tt0000005,	05	BF(5)
...			...			...	...	

The output file of this stage is a list of ten pairs (average rate, `BF(i)`). Each `BF(i)` string is a dump in decimal format of each word representing the Bloom Filter separated by `;` (semicolon). For example:

```
01      -1477355493;1719008440;439399423;...;8791
02      ...
...
```

This file is read in the Mapper setup of the check stage to restore the ten Bloom Filters. Doing so we can check and count the False Positive in a proper way.

Each Reducer of this stage receives a key (corresponding to the rating) an iterable containing the title codes (`tconst`). Before iterating the values, the Reducer gets from the configuration the number of title codes populating this rating (calculated in the count stage) and proceed to create the Bloom Filter of the appropriate size. We could have the number of title codes populating a certain rating, counting how many values there are in the Reducer iterable. However, we cannot iterate twice and we must cache those values while counting them so we can iterate a second time. We prefer to add a count stage at the beginning.

IMDRating check stage

For every possible rating, we select the titles that don't have that rating, we test them, and count the number of false positives. The ratio between the number of false positives and the tested titles gives the obtained false positive rate.

```
IMDRating CHECK

Procedure MAP (Text t, Average a, numVotes v)
for I in range (10)
    Bloom-Filter-Create (n_itemNumber(I), p_falsePositiveRate)
    Bloom-Filter-Initialize (String )
ra = round(a)
for I in range (10)
    if (I + 1 != ra && Bloom-Filter-CheckItem(ra, t)
        EMIT(a (as Text), 1)

Procedure REDUCER (ra a, Links [t₁,t₂, …tₘ])
for all n in Links do:
     count += 1

fpr = count / n_itemNumber(ra)
EMIT(a (as Text), fpr (as Double))
```

The (key, values) pairs of the Map–Reduce job have the following contents

Mapper input			Mapper output		Reducer input		Reduced output	
tt0000001	5.7	1882	10	1	01	1,1,1,...	01	0.009849948777695197
tt0000002	5.9	250	06	1	02	1,1,1,...	02	0.010216955115208785
tt0000003	6.5	1663	07	1	03	1,1,1,...	03	0.010058488094947571
tt0000004	9.8	163	08	1	04	1,1,1,...	04	0.009971343086156027
tt0000005	2.2	2487	09	1	05	1,1,1,...	05	0.010036027661975965
...			...		...		...	

The output file of this stage is a list of ten pairs (giving for each rating the false positive rate.

01	**0.009849948777695197**
02	**0.010216955115208785**
03	**0.010058488094947571**
04	**0.009971343086156027**
05	**0.010036027661975965**
06	**0.009797280857979816**
07	**0.009825030881604154**
08	**0.009849299117739263**
09	**0.00978369859788152**
10	**0.010080397854432524**

Spark

The Bloom Filter corresponds to the Python class `BloomFilter` (internally each bit array is implemented using an array of Boolean). The class exports the following methods:

- **`BloomFilter(items_count, fp_prob)`**
  creates an empty object to store `count` titles and to manage a false positive rate equal to `fpr`
- **`public addItem(t)`**
  add the title `t` to the `BitArray`
- **`public boolean checkItem(t)`**
  check if the title `t` in in the `BitArray`

The spark execution is obtained with the following command:

```
spark-submit IMDRating.py 0.01 data.tsv
```

Reading the file from the Hadoop distributed file system:

```
file = sc.textFile(sys.argv[2])
temp = file.map(lambda x: (x.split("\t")))
titles = temp.map(lambda x: (x[0],java_round(float(x[1]))))
```

Initializing the fpr parameter:

```
fpr = float(sys.argv[1])
```

Creating the Bloom Filters and populate them:

```
bloom_filter = [0 for _ in range(10)]
for rating in range(10):
    titles_collected = titles.filter(lambda x : x[1] == rating + 1).collect()
    bloom_filter[rating] = BloomFilter(len(titles_collected), fpr)
    for title in range(0, len(titles_collected)):
        bloom_filter[rating].add(titles_collected[title][0])
```

Testing the titles and getting the results:

```
for rating in range(10):
    titles_filtered = titles.filter(lambda x : x[1] != rating + 1)
    titles_fp = titles_filtered.filter(lambda x: bloom_filter[rating].check(x[0]) > 0)
    print("{0:02d} {1:12.10f}".format(rating + 1,
                                      titles_fp.count()/titles_filtered.count()))
```

The output of the pyspark code is the following:[4]

```
01    0.0100090392
02    0.0101379539
03    0.0100202553
04    0.0099148435
05    0.0101129053
06    0.0098527537
07    0.0100659188
08    0.0100194814
09    0.0100050532
10    0.0100080991
```

[4] We think the small differences between Hadoop and Spark are due to the implementation of the murmur hash function.