Report on Big Data project: (Analysis on trending youtube videos dataset)

Monica Gondolini - Mat. 855202
Eugenio Cavina - Mat. 833890

July 16, 2019

Contents

1 Introduction

1.1 Dataset description

- This dataset is a daily record of the top trending YouTube videos for different region. To determine the trending videos, YouTube uses a combination of factors including likes, share, comments and views. In the dataset there are also some videos categories for each region with a mapping between category ID and category name (can be different between region).
- Trending YouTube Video Statistics (`https://www.kaggle.com/datasnaek/youtube-new`).
- Direct links (you need to be logged in to download)
  - CAvideos.csv: (`https://www.kaggle.com/datasnaek/youtube-new/downloads/CAvideos.csv/115`)
  - GBvideos.csv: (`https://www.kaggle.com/datasnaek/youtube-new/downloads/GBvideos.csv/115`)
  - USvideos.csv: (`https://www.kaggle.com/datasnaek/youtube-new/downloads/USvideos.csv/115`)
  - US_category_id.json: (`https://www.kaggle.com/datasnaek/youtube-new/downloads/USvideos.csv/115`)

1.1.1 File description

The trending videos **csv** file fields are the same for each region and are the following (underlined fields are used in our analyses):

- video_id
- trending_date
- title
- channel_title
- category_id
- publish_time
- tags
- views
- likes
- dislikes
- comment_count

- thumbnail_link
- comments_disabled
- ratings_disabled
- video_error_or_removed
- description

An example:

```
1ZAPwfrtAFY,17.14.11,"The Trump Presidency: Last Week
   Tonight with John Oliver (HBO)","LastWeekTonight
   ",24,2017-11-13T07:30:00.000Z,"last week tonight
   trump presidency"|"last week tonight donald trump
   "|"john oliver trump"|"donald trump
   ",2418783,97185,6146,12703,https://i.ytimg.com/vi/1
   ZAPwfrtAFY/default.jpg,False,False,False,"One year
   after the presidential election, John Oliver
   discusses what we've learned so far and enlists our
    catheter cowboy to teach Donald Trump what he hasn
   't.Connect with Last Week Tonight online...
   Subscribe to the Last Week Tonight YouTube channel
   for more almost news as it almost happens: www.
   youtube.com/user/LastWeekTonightFind Last Week
   Tonight on Facebook like your mom would: http://
   Facebook.com/LastWeekTonightFollow us on Twitter
   for news about jokes and jokes about news: http://
   Twitter.com/LastWeekTonightVisit our official site
   for all that other stuff at once: http://www.hbo.
   com/lastweektonight"
```

The category name **json** file fields are the following (as above underlined are used):

- kind
- etag
- items [
  - kind
  - etag
  - id
  - snippet
    - channelId
    - title

        * assignable

  -]

An example:


```
{
 "kind": "youtube#videoCategoryListResponse",
 "etag": "\"m2yskBQFythfE4irbTIeOgYYfBU/S730Ilt-Fi-
    emsQJvJAAShlR6hM\"",
 "items": [
  {
   "kind": "youtube#videoCategory",
   "etag": "\"m2yskBQFythfE4irbTIeOgYYfBU/
      Xy1mB4_yLrHy_BmKmPBggty2mZQ\"",
   "id": "1",
   "snippet": {
    "channelId": "UCBR8-60-B28hp2BmDPdntcQ",
    "title": "Film & Animation",
    "assignable": true
   }
  },
  ...
```


2 Data preparation

- **user:** mgondolini
- **address:** isi-vclust4.csr.unibo.it
- **file paths:**

  Dataset:

  - `user/mgondolini/exam/dataset/US_category_id_flat.scala`
  - `user/mgondolini/exam/dataset/cleaned/GBvideos.csv`
  - `user/mgondolini/exam/dataset/cleaned/CAvideos.csv`
  - `user/mgondolini/exam/dataset/cleaned/USvideos.csv`

  Jars:

  - `user/mgondolini/exam/MapReduce/MapReduce-1.0.jar`
  - `user/mgondolini/exam/SparkSQL/SparkSQL-1.0.jar`

  Scala:

  - `user/mgondolini/exam/SparkSQL/TopTagsInVideosCategories.scala`
  - `user/mgondolini/exam/SparkSQL/AvgCommentsClassification.scala`

2.1 Pre-processing

During the first tests we found difficulty to read the csv dataset because of the newlines inside the description column. We identified two possibilities: remove description column or clean it. To keep the dataset large enough to justify big data technologies we chose the second solution. For this reason we used a short script to clean up problematic newlines. The script sets as separator "," then loop from 16th field to the end of the file, for each field remove *"\n"*, $\backslash n$ and $\backslash r$.

```
awk 'BEGIN{FS=OFS=","} {for(i=16;i<=NF;i++) {gsub(/\\n
    /,"",$i);gsub(/\r/,"",$i);gsub(/\n/,"",$i)} }1'
    CAvideosOriginal.csv > output.csv
```

Similar to the previous case, we had the need to flatten the json file and filter it only with significant fields. We created a **jq** script to clean this json file.

```
jq -c '(.items[] | {id, category: .snippet.title})'
    US_category_id.json > US_category_id_flat.json
```

3 Jobs

3.1 Job #1: Average comments number for each video classification in good, bad and neutral

This job aims to evaluate a correlation between liked videos and not with the number of comments. First, every video is classified as **good**, **bad** and **neutral**. After that, the videos are grouped according to the classification and the average number of comments is calculated for each classification. The classification of the video is done with the evaluation of the proportion given by the division between likes and dislikes.

3.1.1 MapReduce

- Command to run this job:

hadoop jar exam/MapReduce/MapReduce-1.0.jar job1.AvgCommentsClassification exam/dataset/cleaned/GBvideos.csv exam/dataset/cleaned/CAvideos.csv exam-/dataset/cleaned/USvideos.csv exam/MapReduce/output

- About parameters:
  - first 3 input are trending videos dataset
  - last input is the directory output where to put result
- YARN History's url
  1. `http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1560510165054_0910`

- input files/table:
  - CAvideos.csv
  - USvideos.csv
  - GBvideos.csv
- output files/table: `user/mgondolini/exam/MapReduce/output`
- Implementation (KEY → VALUE):
  1. **input**
     - VIDEO
  2. **mapper**:
     - splits record until ratings_disabled field
     - filters if ratings or comments disabled or header
     - calculate proportion between likes and dislikes
     - set classification based on proportion
     - transformation:
       - **in:** VIDEO
       - **out:** CLASSIFICATION → COMMENT_COUNT
  3. **shuffling**
  4. **reducer**
     - keeps elements count and calculates total
     - calculate average comment number
     - transformation:
       - **in:** CLASSIFICATION → [COMMENT_COUNT, ...]
       - **out:** CLASSIFICATION → AVG_COMMENTS
- Performance
  - the number of mappers is the default, chosen by the framework based on the splits
  - the most convenient number of reducers is one, because the dataset is not large enough to justify the overhead of reducer parallelization; at a theoretical level the number of reducers must be calculated based on the available cluster nodes
- Output:

```
neutral videos:4959.0 - average comments: 8284.004033071184
good videos:106146.0 - average comments: 8635.72786539295
bad videos:7742.0 - average comments: 13390.340738827177
```

  Analysing the output, we can see that the most commented videos are the most disliked ones, while neutral and good videos maintain almost the same number of comments.

3.1.2 SparkSQL

- Command to run this job:
  `spark2-submit --class AvgCommentsClassification exam/SparkSQL/SparkSQL-1.0.jar`
  or get the code from `exam/SparkSQL/AvgCommentsClassification.scala`
- About parameters: same as MapReduce Job but inside source
- YARN History's url:
  1. `http://isi-vclust0.csr.unibo.it:18089/history/application_1560510165054_0921/jobs/`
- input files / tables
  - CAvideos.csv
  - USvideos.csv
  - GBvideos.csv
- output files / tables:
  - `exam/SparkSQL/output/AvgCommentsClassification.csv`
  - Hive Table: *commentsclassificationtable* stored in *exam_mgondolini* database
    `http://isi-vclust0.csr.unibo.it:8889/hue/metastore/table/exam_mgondolini/commentsclassificationtable`
- Implementation
  - **reading input**
    - read csv file with "," as delimiter while dropping malformed
  - **input union**
    - union of three DF filtered by comments and ratings enabled
  - **video classification and counter**
    - creation of 3 DF: good, bad and neutral; using where condition over union DF with specific condition for classification (empirically evaluation of proportion)
    - count elements in filtered DF
  - **classification aggregation with average**
    - aggregate the entire classified DF with comments total
    - divide comments total by elements counted
- Performance
  - Cache is used to use memory efficiently in tables where we expect to run multiple tasks or queries

- Output: `classificationDF`

```
+--------------+-------------+------------------+
|Classification|Videos Number|  Average Comments|
+--------------+-------------+------------------+
|       neutral|         4959| 8284.004033071184|
|          good|       106146|  8635.72786539295|
|           bad|         7415|13930.734996628456|
+--------------+-------------+------------------+
```

  Videos number and average comments for bad classification videos are slightly less compared to the MapReduce result. This is probably due to a input reading difference between the two frameworks and the programming languages.

- Job's result is saved also in Hive in the course's cluster (see output files/tables)..

3.2 Job #2: Top 10 of the most used tags for each video category

This job calculates the occurrences of a tag in a specific category and then filters the most used ones within it.

3.2.1 MapReduce

- Command to run this job:

hadoop jar exam/MapReduce/MapReduce-1.0.jar job2.TopTagsInVideosCategories exam/dataset/cleaned/GBvideos.csv exam/dataset/cleaned/CAvideos.csv exam/dataset/cleaned/USvideos.csv exam/MapReduce/output1 exam/MapReduce/output2 exam/MapReduce/output3 exam/dataset/US_category_id_flat.json

- About parameters:
  - first 3 input are trending videos dataset
  - then 3 output, outputN where N is the job order
  - last is category name mapping in json
- YARN History's url
  1. `http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1560510165054_0911`
  2. `http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1560510165054_0912`
  3. `http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1560510165054_0913`

- input files/table
  - CAvideos.csv
  - USvideos.csv
  - GBvideos.csv
  - US_category_id_flat.json
- output files/table
  - `user/mgondolini/exam/MapReduce/output1`
  - `user/mgondolini/exam/MapReduce/output2`
  - `user/mgondolini/exam/MapReduce/output3`
- Implementation (KEY → VALUE):
  1. **input**
     - VIDEO
  2. **mapper A**
     - filter if header
     - splits record until tags column
     - splits tags field
     - for each tag write new line with category, tag (lower case) and one occurrence
     - transformation:
       - **in:** RECORD
       - **out:** ... CATEGORY#tag → 1 ...
  3. **combiner A**
     - use reducer A
     - sum partially the combination of category and tag in same node
  4. **shuffling**
  5. **reducer A**
     - sum all occurrences of a specific combination of category and tag
     - transformation:
       - **in:** CATEGORY#TAG → N
       - **out:** CATEGORY#TAG → TOTAL
  6. **mapper B**
     - splits category, tag and total
     - puts total also in key
     - transformation:
       - **in:** CATEGORY#TAG → TOTAL
       - **out:** CATEGORY#TOTAL → TAG#TOTAL

7. **sorting**
   - defines how mapper output will be sorted through a comparator
   - splits key elements (category and total)
   - if category is the same compare total and evaluate descendant
   - else compare category
8. **partitioner**
   - defines how items will be sorted in partitions
   - using hash code of category instead of completed key
9. **grouping**
   - defines how items will be grouped in single reduce call
   - using category instead of all key
10. **reducer B**
    - split key
    - insert first N (case study: N=10) elements of values with category as key
    - transformation:
      * **in:** CATEGORY#TOTAL → [TAG#TOTAL, . . .]
      * **out:** CATEGORY → [TAG#TOTAL, . . . , N]
11. **mapper C1**
    - parse JSON line to take category id and name value
    - add flag in value, useful for join
    - transformation:
      * **in:** RECORD
      * **out:** CATEGORY_ID → "join-category"#CATEGORY_NAME
12. **mapper C2**
    - dummy mapper that reads reducer B output
13. **reducer C**
    - iterate over values (from mapper C1 and C2)
    - if value comes from C1 set it as key
    - if value comes from C2 set it as value
    - transformation:
      * **in:** CATEGORY → [TAG#TOTAL, . . . , N] **or** "join-category"#CATEGORY_NAME
      * **out:** CATEGORY_NAME → [TAG#TOTAL, . . . , N]

- Performance
  - the number of mappers is the default, chosen by the framework based on the splits

  - the most convenient number of reducers is one, because the dataset is not large enough to justify the overhead of reducer parallelization; at a theoretical level the number of reducers must be calculated based on the available cluster nodes
  - in first job a combiner is used to sum elements because the operation is associative and commutative; this reduces intermediate data and network traffic
  - a custom partitioner is used to maintain the order but this could lead to unbalancing the load on the reducers if some categories are much more present than others
  - combination of sorting comparator, grouping comparator and partitioner makes it possible to reach the goal of the job without having to use in memory data structures inside the reduce call.
  - the join job was held at the end to apply it to as few records as possible (transform the key from number to string)

- Output:

```
Film & Animation ["trailer"#1743, "movie"#1119, "
    animation"#895, "film"#832, "official"#650, "
    comedy"#613, "disney"#481, "2018"#456, "movies
    "#429, "drama"#418]
Autos & Vehicles ["doug demuro"#194, "demuro"#163,
     "car"#98, "tesla"#97, "turbo"#75, "crash"#72,
    "cars"#68, "diy"#68, "review"#58, "race"#55]
People & Blogs  ["funny"#1015, "buzzfeed"#966, "
    comedy"#893, "vlog"#715, "fashion"#636, "food
    "#424, "family"#383, "buzzfeedvideo"#366, "
    christmas"#351, "comedian"#326]
Comedy  ["funny"#3551, "comedy"#3035, "humor
    "#1790, "talk show"#1438, "nbc"#1088, "
    television"#1074, "nbc tv"#1067, "parody"#833,
    "clip"#803, "video"#782]
[...]
```

3.2.2 SparkSQL

- Command to run this job: `spark2-submit --class TopTagsInVideosCategories exam/SparkSQL/SparkSQL-1.0.jar`
  or get the code from `exam/SparkSQL/TopTagsInVideosCategories.scala`
- About parameters: same as MapReduce Job but inside source
- YARN History's url:
  1. `http://isi-vclust0.csr.unibo.it:18089/history/application_1560510165054_0922/jobs/`

- input files / tables:
  - CAvideos.csv
  - USvideos.csv
  - GBvideos.csv
  - US_category_id_flat.json
- output files / tables:
  - `user/mgondolini/exam/SparkSQL/output/TopTagsInVideosCategories.csv`
  - Hive Table: *tagsgroupedbycategorytable* stored in *exam_mgondolini* database `http://isi-vclust0.csr.unibo.it:8889/hue/metastore/table/exam_mgondolini/tagsgroupedbycategorytable`
- Implementation
  - **reading input**
    - read csv file with "," as delimiter while dropping malformed
    - read json category file
  - **input union**
    - union of three DF
  - **broadcast join**
    - broadcast join between union DF and category names (really small DF)
  - **DF selection with only necessary columns** (category and tags)
  - **exploding tag list**
    - new lines for each tag inside a specific record
    - filtering of [none] tag
    - transformation of every tag to lower case
  - **count occurrences of same tag for each category and store them in a new column**
    - select and count all record grouped by category and tag
    - create column with counter
  - **select top 10 rows for each category, descending by counter**
    - first select row_number() over partition by category and ordered descendant by counter
    - starting from previous table select category, tag and counter of first 10 elements ordered by category and counter
  - **group by category, creating a list of tags#count**
    - group by category and format output

- Performance
  - the broadcast join is used to boost performance because category names table is really smaller than videos table
  - cacheTable is used to use memory efficiently in tables where we expect to run multiple tasks or queries
  - temporary tables is used to create a lazily evaluated "view" that we can use in sql query directly
- Output: `groupedByCategoryString.collect()`

```
[Film & Animation,"trailer"#1743,"movie"#1119,"
   animation"#895,"film"#832,"official"#650,"
   comedy"#613,"disney"#481,"2018"#456,"movies
   "#429,"drama"#418]
[Autos & Vehicles,"doug demuro"#194,"demuro"#163,"
   car"#98,"tesla"#97,"turbo"#75,"crash"#72,"diy
   "#68,"cars"#68,"review"#58,"race"#55]
[People & Blogs,"funny"#1015,"buzzfeed"#966,"
   comedy"#893,"vlog"#715,"fashion"#636,"food
   "#424,"family"#383,"buzzfeedvideo"#366,"
   christmas"#351,"comedian"#326]
[Comedy,"funny"#3551,"comedy"#3035,"humor"#1790,"
   talk show"#1438,"nbc"#1088,"television"#1074,"
   nbc tv"#1067,"parody"#833,"clip"#803,"video
   "#782]
[...]
```

- Job's result is saved in Hive in the course's cluster (see output files/tables).