How TuBe Popular

Data Management and Visualization project report

Studenti:
Leonardo Alchieri - 860624
Davide Badalotti - 861354
Lucia Ravazzi - 852646
Pietro Bonardi - 859505
Ilenia Lo Monaco - 769348

GitHub Repository:
https://github.com/LeonardoAlchieri/Top-Of-Youtube

February 10, 2021

Contents

List of Figures

Introduction

Over the course of the last five to ten years, YouTube has been gradually growing its relevance inside the internet landscape. From a platform where semi-anonymous users used to share their experiences and interests with a limited range of people, it has now become a place where all kinda of individuals and organizationa can profit from, by reaching millions of people of all ages and backgrounds.

Nowadays, content creators are more and more willing to put a lot of effort and resources in their product to assure themselves maximum profits. Thanks to this process, it is clear that the content quality of videos made by famous creators is rapidly increasing. But this process goes beyond just that.

Beside increasing the efforts put in every creation, popular channels have developed numerous techniques to make their videos more *appealing* from the outside, in order to tempt users to click on them, even though not totally interested. Techniques as such include, for example, a particular use of capital letters inside titles, the use of certain words, the editing and choice of the thumbnail, the format of the description and more.

In this project, our aim is to answer the following question: **What are the techniques that can make a video snippet look more appealing?**

We tried to answer this questions using the information given us by big data analyisis. For this project, we collected a high volume of data from multiple sources and in different formats. To store the data, we have implemented an architecture that allowed us to distribute files across multiple supports, thus making querying and analysis much more efficient. As we will show, this architecture allows efficient scaling, allowing for storage of way higher volume than the one we used.

After storing all the data collected, we have implemented a procedure to integrate all the different sources. This procedure, together with the one described early, supports high volumes of data and it is able to scale efficiently, as we will show in this report.

To further increase the efficiency of the loading and integrating processes, we have built our code in order to support parallelization, both locally and across s cluster of computers.

After analyzing the data, we have summarized our conclusions and tried to prove our points with the use of data visualizations, designed to be the most effective, informative and elegant as possible.

Chapter 1

Data Management

1.1 Data sources description

For this project, we decided to get data from three different sources, described in this section.

1.1.1 Kaggle dataset

The first dataset used is a collection of `.csv` files retrieved from the website https://www.kaggle.com. Each `.csv` file contains useful information about the videos that have been trending on Youtube on each day, starting from 2017-01-12 until 2018-05-31.

Every `.csv` file refers to the trending videos in a single country. For this type of analysis, we decided to use the files referring to the USA, Canada and Great Britain.

Why did we choose to use this dataset?

In our research , the main focus is to determine the features that make a video look more appealing.

We can assume that these kind of techniques are more frequently used in popular videos than in the more common ones. The basis of this assumption is the fact that each technique requires a certain kind of effort to be implemented, and so it is most likely to be found in videos made for large audiences, thus our choice to analyze this type of data.

We are aware of the fact that not all trending videos have this kind of characteristic, and we will thus confront this problem in later stages of the analysis.

Contents

The information contained in each `.csv` are the following:

- `video_id`: a string which uniquely identifies the video inside youtube servers

- `trending_date` : the day in which the video has been trending
- `title` : the title of the video
- `channel_title` : the name of the channel that uploaded the video
- `category_id` : a number which idenifies the category of the video
- `publish_time` : the day in which the video has been published
- `tags` : the tags associated with the video. They are chosen by the publisher and used to determine the related videos
- `views` : the number of views that the video has on the `trending_date`
- `likes` : the number of likes that the video has on the "trending_date"
- `dislikes` : the number of dislikes that the video has on the `trending_date`
- `comment_count` : the number of comments that the video has on the `trending_date`
- `thumbnail_link` : the url string of the thumbnail of the video
- `comments_disabled` : a boolean value which indicates if, on the `trending_date`, the comments of the video were disabled
- `rating_disabled` : a boolean value which indicates if, on the `trending_date` the likes and dislikes of the video were disabled
- `video_error_or_removed` : a boolean value which indicates if on the `trending_date` the video was available

It is important to point out that the information reported refers to each single day in which a video has been trending on Youtube. This gives us the possibility to determine if, during that period, some elements of the video are modified, for example the title or the description. Also, we can analyze how the statistics vary over time.

Notes on the use of the Kaggle dataset

The datasets from Kaggle are available only via manual download only through a browser. This fact is in contrast with the requests of this assignment, that requires to implement an architecture able to handle big amounts of data.

To resolve this problem, we decided to use these datasets in an alternative way. Basically, we created a SQL server on a Microsoft's Azure remote machine , called *tars1* (also described later) and we loaded the `.csv` files on it, after dowloading it from Kaggle.

During all the procedures, we treated this database *as* an external source that contained large quantities of data, to which we had access. We also proceeded without making assumptions on the time interval to which the data refers, in order to make the procedure suitable for future uses.

We are aware of the fact that relational databases are not built to handle large amounts of data; however, in this case, we think of the database as an object whose architecture we cannot control, to which we will access remotely.

The code used to load the data on the remote SQL server is contained in the `dataPreloading/preloadKaggle.py` file. The code simply uploads the `.csv` stored locally on the remote server without making any changes to the structure. From three separate `.csv` files (one for each country), we created three different tables containing all the data.

1.1.2 Youtube API dataset

Why did we choose to use this dataset?

The Kaggle dataset contains information about videos that have been trending over a certain time. The scope of our research is not limited to trending videos only, se we decided to integrate the source described early with other videos retrieved with the official Youtube API.

After requesting a *developer key*, the website allow us to search for Youtube videos matching certain criteria. In Section 1.3.1 we will go into detail about these criteria and what we have applied.

Contents

For this analysis, we only requested the `video_id` of random videos in each category. Ideally , the API gives access to all the video elements; however, the API sets a maximum number of daily requests for each user. This quota is, in our case, very low, allowing us to retrieve only 300 full videos per day per account. In order to get the largest amount of data, we decided to request as few fields as possible for each video. The `video_id` is enough for our purposes.

With this kind of request, we are able to retrieve as much as 10 thousand video a day, using around three keys. The Youtube API request returns list of `.json` files, each containing one single `video_id`.

1.1.3 Scraper

Why did we choose to use this dataset?

Due to the quota problem described earlier, we were unable to retrieve information about the videos obtained via the Youtub API. For this reason, we decided to implement a procedure that

allows us to get the data we need directly from the website, thanks to a web scraping method. This way, we have access to all the information we need.

The purpose of our Youtube Scraper, though, is not only to provide information on the videos from the API dataset, but also to enrich all the other data from the Kaggle dataset.

The reason is simple: the scraper is able to also get the comments of the videos. This information is not available in the Kaggle dataset, and we think it could be very useful for our purposes, since it allows us, through the method of sentiment analysis, to have an insight on users opinion about the video.

To summarize, the question we would like to answer by looking at the comments is: *If a video makes use of certain techniques to appear more interesting, will it also be more appreciated?*

Contents

The data retrieved with the Youtube Scraper are in .json format and contain these fields:

- `id`: the same string used in to identify the videos described in the previous sections
- `title`: the title of the video
- `ref_date`: the date in which the video info has been scraped. This information has the same function of the field `trending_date` in the Kaggle dataset. Namely, indicating the day all the sother field (views, comments, like, dislikes, etc..) refer to. This way, we are able to analyze the same video during an arbitrary period of time.
- `category`: the category of the video reported inside the page. This category is different from the categories to which the field `category_id` (in the previous datasets) refers to.
- `channel_name` : the name of the video publiser
- `channel_subscribers`: the number of subscribers to the channel
- `comments_count`: the number of comments to the video
- `comments`: a list of documents containing information about every comment to the video. Each document contains:
  - `text`: the text of the comment
  - `user`: the author of the comment
  - `replies_count`: the number of replies to the single comment
  - `likes`: the number of likes to the comment
- `date`: the date of publishing

- `description`: the description of the video
- `dislikes`
- `likes`
- `url`: the url of the video
- `views`: the number of views
- `thumbnail_link`: the url referring to the thumbnail of the video

1.2 Data Storage Architecture

To efficiently store the data we collect, we chose to use the **MongoDB database management system**.

The features that brought us to this choice are:

- Schema-free paragidm: MongoDB is a document database, able to store semi-structured data. This kind of data is not constrained by a schema, and it's exactly what we need in our particular analysis.

  MongoDB, in particular, stores data in flexible, JSON-like documents, giving the possibility for structure changes over time. The documents we are going to store are characterized by the presence of multiple sub-documents, as we will show towards the final steps of our procedure.

  MongoDB gives us the possibility to embed documents, and thus to store and query them efficiently. Moreover, the structure complexity of our data makes it unsuitable for other types of databases, like a key value store or a column one. We would be, for example, very challenged to store the video comments in column-type DBMS.

- Scalability: MongoDB is a distribuited database at its core. For this reason, horizontal scaling and geographic distribution are built in and relatively easy to set-up and use, as we will show in Section 1.2.1.

- Bulk-operations: as we will show, our codes make heavy use of bulk-write and update operations. This kind of commands allow us to pre-define a large amount of requests to the server. Rather than sending the request one-by-one and collecting the results afterwards, the `mongos` server is able to handle all the requests and collect the results at once, without the need of multiple round trip to the server.

  In our case, where servers are located in very distant places, this could make the code much more efficient.

1.2.1 Architecture description

MongoDB supports horizontal scaling through **sharding**: namely, a database architecture that distributes data by key ranges and stores data among various database instances, e.g. in different computers.

A MongoDB sharded cluster has three main components:

- Shard: each instance of the database containing a subset of the sharded data. Each shard can also be deployed in a replica set.

- Config Servers: they store metadata and configuration settings for the cluster. They need to be deployed in replica set.
- Router: called `mongos` serves as a query router, providing an interface between client and database.

For this project, we made use of 3 different machines, provided by *Microsft's Azure* platform. Two of these are located in Western Europe and one is located in Northern Europe. We depict a schema for our architecture in Figure 1.1.

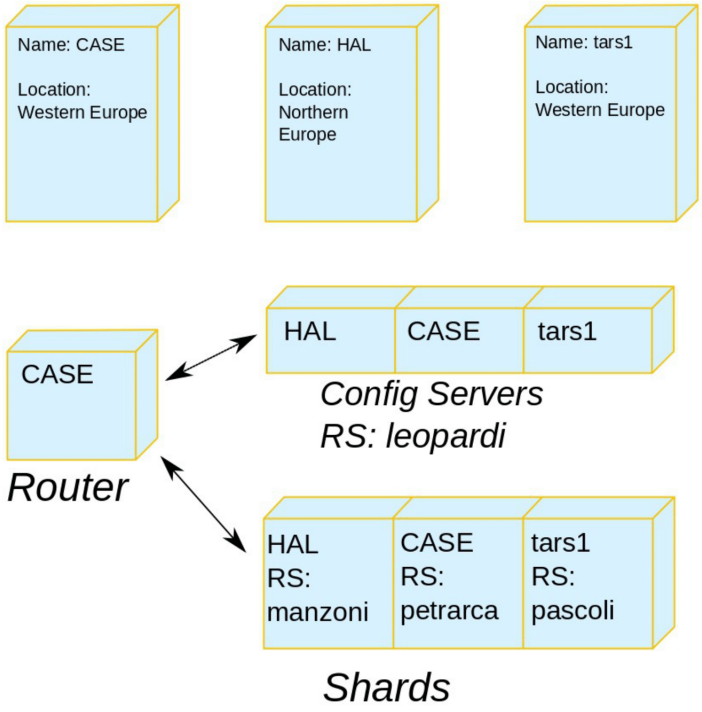


Figure 1.1: Cluster structure. The first part of the image represents the different machines, while the second part shows their role inside the cluster. (RS stands for Replica Set)

As we can see, each machine hosts at least 2 MongoDB instances, in particular:

- `tars1`, which is also the host of the mysql server, serves as a config server (replica set named `leopardi`) and as a shard (replica set named `pascoli`)
- `CASE` serves as a config server (replica set named `leopardi`), as a shard (replica set named `petrarca`), and as a mongos (named `dante`)
- `HAL` serves as config server (replica set named `leopardi`) and as a shard (replica set named `manzoni`)

In total, we have one config server replica set, made of 3 replicas, 3 shard, each part of a different replica set and one mongos. This particular architecture allows us to scale horizontally very easily.

To add a new mongod instance to a current existing replica set, it is only required to connect to the replica set's primary server and add the new server with the following command:

```
rs.add( { host: "<public ip address>:<listening port>", priority: 0,
    votes: 0 } )
```

Where `rs` stand for the name of the replica set. The mongos instance will automatically recongnize the new server. On the other hand, to add a new shard, it is only required to connect to the mongos instance and add the corresponding mongod instance with the command:

```
sh.addShard( "<public ip address>:<listening port>" )
```

It is fair to say that every mongod and mongos instance require little configuration, involving only a few settings inside a config file for each instance.
Anyways, the shard map of our architecture is easily obtainable by connecting to the mongos instance with the following command:

```
mongo --host case.bounceme.net --port 80
```

After connecting to the mongos instance, one can see the shard map by typing:

```
use <database_name>
sh.runCommand("getShardMap")
```

It is also important to specify that MongoDB allows efficient querying when the operation targets a single cluster, through the use of the *shard key*. If a shard key is included in the query fields, MongoDB is able to use metadata from the config server to route the queries to shards.

If a shard key is not present, the query is called *scatter-gather query*, because the mongos must direct the query to all shards, resulting in a very inefficient task.

One can see then how it is important to choose the right *shard key*, by taking into account its cardinality (the number of possible value), its frequency (how many times any values appears in the data) and its rate of change (the distribution of the values inside the data).

It is also possible to define a *hashed shard key*, mainly for object with high cardinality and monotonically increasing. A hashed shard key ensures more even data distribution across all shards.

In our analysis, we have chosen to use a range based, compound shard key for the first steps.

This key takes two fields: `video_id` and `ref_date`. These fields, beside being the center of interest of our operations, constitute a good example of values that have high enough cardinality and change monotonically. It is now, for us, very convenient to query the database on the field `video_id` only or on both fields. One should note that a query only on the second key is interpreted as a normal one, i.e. the first key bust always be present for efficient jobs.

1.3 Data Loading

Once the architecture had been established, we implemented some loading and preparation scripts, in order to have our database up and running for analysis. As stated in Section 1.1, we decided to integrate and enrich three different data sources; as expectable, the process of loading such data unto our MongoDB has been divided in three separate steps:

- Getting and uploading directly *id*s from the **Youtube API**.
- Moving Kaggle data from the MariaDB reletional database (which we have taken as source) and uploading it as documents.
- Scraping data, using as reference *id*s from both Kaggle and API databases, and uploading it directly as a Mongo collection.

As stated previously, in order to allow our system to scale out as much as possible, we implemented a structure that would allow for parallelization and not usage of a local filesystem storage.

In the next Sections we will describe in detail the methodologies applied in each case, with particular attention to the scripts utilized.

1.3.1 API

Tools

The **Youtube API** is a very expansive and accessible tool that allows, through `html` queries, to retrieve data regarding each video on the platform in a document format. As stated previously, we only had access to around a few thousand indexes, i.e. one field of information, per day per account. To overcome this limit, we both implemented queries throughout multiple days and with the use of multiple accounts.

To implement the usage of the API with our database Mongo, we thought that **python** would have been the fittest choice. Indeed, both platforms have easily downloadable libraries for this programming language.

The first one is simply called `apiclient`[1] and can be installed via `PyPI`. It is designed to run queries across multiple APIs, one of which is, obviously, Youtube.

The second one is `pymongo`, a delightful Mongo interface for Python.

Query API

Due to the limitations applied by Youtube in the number of requests that can be run daily, we decided to query for as little information as possible for each video, as already mentioned. Indeed,

[1] https://pypi.org/project/apiclient/

from each one we only took two indexes, `videoID`, the unique identifier for the video and necessary for the scraping, and `nextPageToken`, which allowed to move along result pages. We show in the Snippet below part of the code used to implement the procedure desired.

```
res = youtube.search().list(part = 'snippet',
                type = 'video',
                maxResults = MAXIMUM,
                publishedAfter = first_date,
                publishedBefore = second_date,
                pageToken = next_page_token,
                regionCode = region,
                fields = 'items/id/videoId,nextPageToken',
                videoCategoryId = categoryId
            ).execute()
upserts = [ UpdateOne(
        {'id':x["id"]["videoId"]},
        {
            '$setOnInsert':{"id":x["id"]["videoId"]},
            '$addToSet': {"regionCode": str(region)}
        }, upsert=Trued) for x in res["items"]]

collectionMongo.bulk_write(upserts)

next_page_token = res.get('nextPageToken')
```

As one can see, the query we run does have many fields. In order to render the code more accessible as possible, we implemented the use of a **configuration file** for specifying both API query items and the connection to the MongoDB instance. Between the two configurations, the fields that could be specified for this loading script are the following:

- `first_date` and `second_date`: they specify the timeframe onto which the video was published. In our case, we decided to use the first and last date reported on the Kaggle dataset.
- `key`: unique identifier for the API account used.
- `videosPerPage`: if left empty, the program would search for all possible videos in a result page (50). In the Snippet above, it called `MAXIMUM`.
- `numPages`: the number of pages to take into consideration.
- `regionList`: to be written as the abbreviation of the country name, e.g. CA for Canada; it specifies the region to use a filter for the research. We decided to implement searches only for US, i.e. United States.
- `categoryIds`: to be written as list, it specifies onto which categories to filter the search. Identified by a number as key, we used only those that appeared in the Kaggle dataset.

- `host`: to specify the host through which to access the MongoDB. Specifically, for a sharded database as the one we have deployed, it should be the access to the *router*. In our case, as we housed it in our **CASE** Azure server, the access is `case.bounceme.net:80`.[2]
- `database`: used to specify which database to create the API collection on.

In order to minimize the communications between the computer where the scripts would be executed (be it on the server or a remotely connected device), we used the **bulk writing** paradigm. This allows for the execution of different operation on the database all at once.

As shown the previous Snippet, the operations executed were a series of single updates with `upsert` option: this means that, for each query, identified by the `videoId` as obtained through the API, a new document composed by only the `id` field would be written. If the document existed already, it would only add to the field `regionCode` the one used for the API search (videos can be viewable in multiple regions).

As `shardKey`, i.e. the document field onto which to shard the collection, we decided to implement only the video id, called `id` in the document, taken hashed.

Due to the simplicity and lightness of the data obtained by this script, we decided not to implement parallelization of the code. We would like to point out, however, that, if need be, some form of parallilazion, either in the API searches or even with multipe API keys, could be easily achieved.

Collection loaded

After the process was run successfully (using around 3 key API accesses), we managed to garner more then 14'000 new video ids. We report hereafter the answer given by the router when run with the command `db.API.getShardDistribution()`, which gives information regarding the number of documents, the size and how they are stored between the different shards. As one can see, each shard houses roughly one third of the collection.

```
Shard manzoni at manzoni/168.61.100.92:27020
 data : 292KiB docs : 4346 chunks : 2
 estimated data per chunk : 146KiB
 estimated docs per chunk : 2173

Shard petrarca at petrarca/94.245.111.63:27020
 data : 300KiB docs : 4454 chunks : 2
 estimated data per chunk : 150KiB
 estimated docs per chunk : 2227

Shard pascoli at pascoli/52.136.247.24:27020
 data : 292KiB docs : 4341 chunks : 2
 estimated data per chunk : 146KiB
 estimated docs per chunk : 2170
```

[2] We had to deploy the 80 port in order to allow connection from places where other ports are blocked.

```
Totals
 data : 885KiB docs : 13141 chunks : 6
 Shard manzoni contains 33.07% data, 33.07% docs in cluster, avg obj size on shard : 69B
 Shard petrarca contains 33.89% data, 33.89% docs in cluster, avg obj size on shard : 69B
 Shard pascoli contains 33.03% data, 33.03% docs in cluster, avg obj size on shard : 69B
```

Testing

As required, we tested the architecture every time it was deployed. As for the API dataset, even though its quite small size (not even 1MB) probably renders any testing useless, we nevertheless run one. In particular, we uploaded for multiple times at different steps the dataset. In Figure 1.2 we show the average loading time and its standard deviation for each percentage of loading. As one might expect for such light dataset, there isn't really a noticeable difference; it should also be noted that the internet connection might have been an effect big enough to cover anything database related.

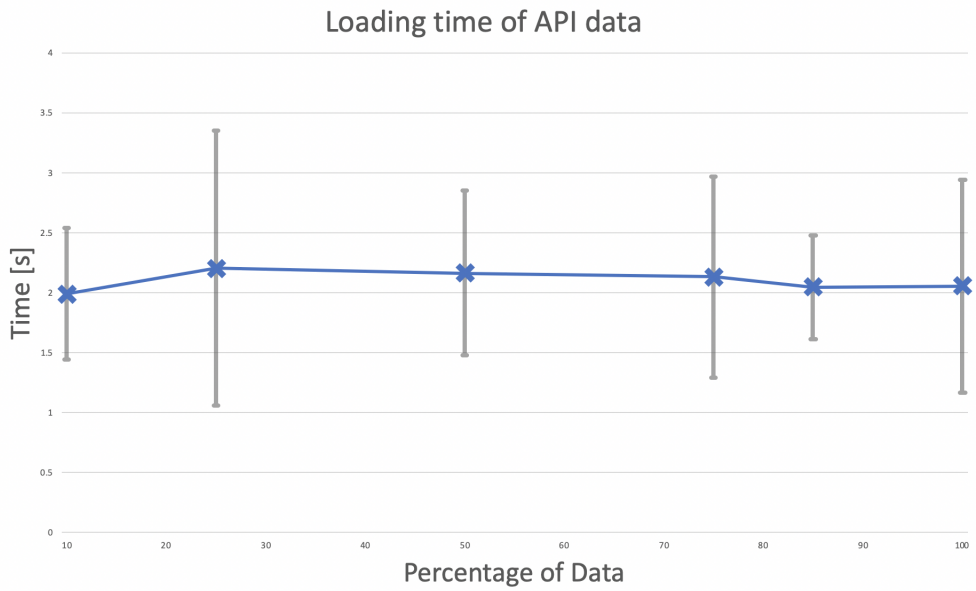


Figure 1.2: Loading test for the API data.

1.3.2 Kaggle

Tools

As stated in Section 1.1, we decided to load the downloaded Kaggle dataset onto a MariaDB relational database. We than treated this latter as the "source" of our data. Because of this, when we implemented a script to load the aforementioned data on Mongo, we had to connect to both the MySQL instance (used by MariaDB) and the MongoDB router.

In order to connect to the relational database that housed the initial data, we utilized the `sqlalchemy` library, which implements methods to run SQL queries directly in Python. In order to allow the script to run on the most diverse type of computers, we limited each query SQL to a batch of the total dataset (we used 2000, but it can be taken to less than this).

The `pymongo` library was used to connect to the MongoDB router, as described previously.

In this case, due to the larger size of the dataset compared to the one for the API, we decided to implement a very simple form of parallelization. While not really the most effective, we prepared the code to be run concurrently on the 3 different tables housed on MariaDB, i.e. `USvideos`, `CAvideos` and `GBvideos`. To do this, the library of our choise has been `mpi4py`, a Python implementation of the C++ famous library `MPI`. While it does parallelize all of the code, even with some known limitations, we preferred it to other solutions, e.g. `multiprocess`, for the possibility of running threads on different computers, hence having more horizontal scalability.

SQL query

Again, for the purpose of limiting the number of connections to the document database we used, we implemented bulk operations. In the snippet below is reported a part of the script. In particular, we would like to point to the SQL query used, which selects only some of the fields present in the SQL table, and the operations executed following. Firstly, we changed some of the column names in order to match those as produced by the scraping script.

```
chunksSQL = pd.read_sql("SELECT video_id, trending_date, title,
                        channel_title, publish_time, tags,
                        views, likes, dislikes, comment_count,
                        thumbnail_link,comments_disabled, ratings_disabled,
                        video_error_or_removed, description  FROM "+str(
                            country_list[ID]),
                        con=connectorSQL, chunksize=2000)

logging.info("["+str(ID)+"] Data loaded from SQL.")
count = 0
for chunk in chunksSQL:
    logging.info("["+str(ID)+"] Step "+str(count))
    # Change column names to fit our naming schema
    chunk = chunk.rename(columns={"trending_date": "ref_date",
                                  "video_id": "id",
                                "channel_title": "channel_name",
                                "publish_time": "date",
                                "comment_count": "comments count"})
    # Change date
    chunk["date"] = chunk["date"].apply(lambda x: x[:10])
    # Change ref_date
    chunk["ref_date"] = chunk["ref_date"].apply(clean_ref_date)
    chunk = chunk.to_dict("records")
```

```
23      # Add regionCode & Kaggle
24      upserts = [ UpdateOne(
25          {'id':x['id'], 'ref_date':x['ref_date']},
26          {
27              '$setOnInsert':x,
28              '$addToSet': {"regionCode": str(country_list[ID][:2])}
29          }, upsert=True) for x in chunk]
30
31      collectionMongo.bulk_write(upserts)
32      count += 1
```

We then performed an updating operation, similar to the one used for the API loading, with a small difference in the query: due to the indexation (not reported in the Snippet) of the sharded collection by `id` and `ref_date`, we had to run the query along these values.

Again, the region code was inserted as a list (with the function `addToSet`). For this instance, we used the name of the country as reported on the SQL table, e.g. `US` for *USvideos*.

In this case, we decided to opt for some form of **schema transformation**: we discarded a few non useful to use fields and changed the names of a few others, as is depicted in the code from line `13`. The process was done to have consistent naming with the scraping data, in order to allow an easier integration a posteriori.

Collection loaded

After the process was run successfully on three threads, as designed, we observed the collection to be sharded as expected. As done previously for the API, we show hereafter the details of how the data is divided into the different shards. As can be easily seen, each shard houses about one third of the whole collection.

```
Shard manzoni at manzoni/168.61.100.92:27020
 data : 64.54MiB docs : 39025 chunks : 3
 estimated data per chunk : 21.51MiB
 estimated docs per chunk : 13008

Shard petrarca at petrarca/94.245.111.63:27020
 data : 48.08MiB docs : 29019 chunks : 3
 estimated data per chunk : 16.02MiB
 estimated docs per chunk : 9673

Shard pascoli at pascoli/52.136.247.24:27020
 data : 67.24MiB docs : 40457 chunks : 3
 estimated data per chunk : 22.41MiB
 estimated docs per chunk : 13485

Totals
 data : 179.87MiB docs : 108501 chunks : 9
 Shard manzoni contains 35.88% data, 35.96% docs in cluster, avg obj size on shard : 1KiB
 Shard petrarca contains 26.73% data, 26.74% docs in cluster, avg obj size on shard : 1KiB
```

```
Shard pascoli contains 37.38% data, 37.28% docs in cluster, avg obj size on shard : 1KiB
```

Testing

We show in Figure 1.3 how the loading script behaves for different percentages of the dataset. Even though affacted by a high degree of error, due to instability related to internet connection, one can distinguish some sublinear trend. While a few hundred megabytes is not the heaviest test for such a distributed DBMS as the one we have deployed, we believe it is nevertheless an important result for the effectiveness of our architecture.

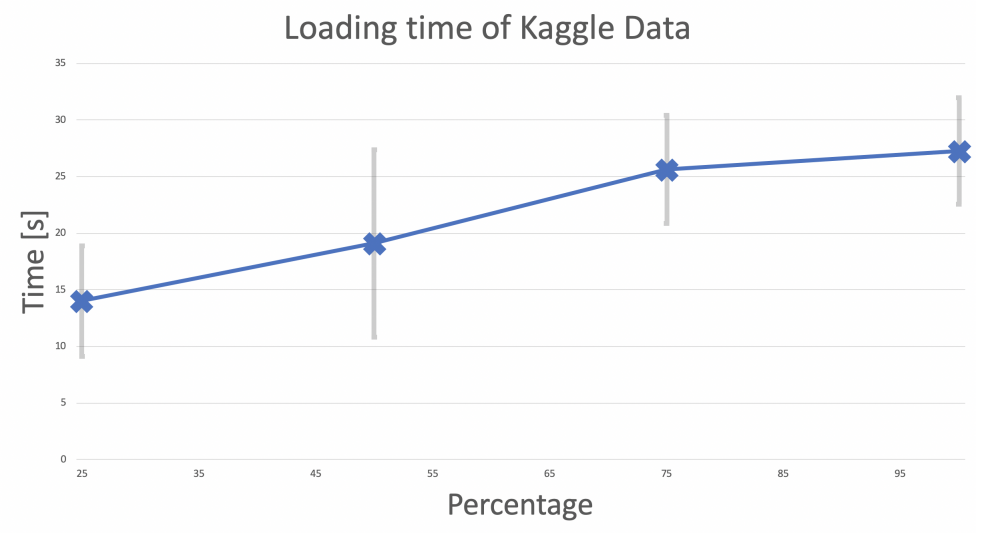


Figure 1.3: Load time for different percentages of the dataset.

1.3.3 Scraper

Tools

The most important data we used for our project is centered around the procedure of **scraping**. This name refers to the technique of using some automated software to extrapolate data directly from a webpage, usually via looking at the `html` code.

In our case, as reported in Section 1.1.3, we were interested in both confronting the information provided by the Kaggle dataset and obtaining comments related data.

In order to achieve this, we decided to create our own specialized library, with the aim of sharing our work as much as possible. Indeed, our scraper is now present as a standalone project on **PyPI** and can be downloaded using the command `pip install YoutubeScraper`.[3] To build our library, we mostly relied on the `selenium` library, which allows for both information extraction

[3] https://pypi.org/project/YoutubeScraper/

and the automation of interactions. We decided to utilize this building tool, as opposed to others such as `BeautifulSoup`, for the way the webpage is built: when one loads a Youtube video, in order to get to the comments there needs to be some scroll down action. On the downside, we had to rely on the use of a browser, more specifically **Google Chrome**, and its rendering of each single page.

We will not go into detail on the way our `YoutubeScraper` library works and we shall treat it as another tool en par with the others we have deployed.

Doing scraping

In the implementation of the scraping, once our library had been imported, little had to be done. It should be noted that the scraping is executed using only the video `id`, which we took either from the Kaggle or API dataset. In order not to split the process with multiple scripts, we implemented a simple trick: one should run the code with `1` or `2` depending on the choice of the database, respectively Kaggle or API.

This script as well used a configuration file, where one can only change hostname and port for the Mongo.

In the Snippet below we show part of the script. Again, not depicted, we opted for a sharded collection and used the `id` and `ref_date` as keys onto which organize it.

```
1  BATCH_SIZE = round(sourceCollection.count_documents({})/comm.Get_size() +
       0.5)
2  logging.info("["+str(ID)+"] Batch size calculated.")
3  cursor = sourceCollection.find().skip(BATCH_SIZE*ID).limit(BATCH_SIZE)
4  logging.info("["+str(ID)+"] Scraping data.")
5  upserts = [ UpdateOne(
6          {'id':doc['id'], 'ref_date':str(date.today())},
7          {# If there a duplicate, the scrapign will not be done.
8              '$setOnInsert':Scraper(id = doc['id'], driver = driver, scrape=
                   True, close_on_complete=False,
9              speak=False).as_dict()
10             }, upsert=True) for doc in cursor]
11 logging.info("["+str(ID)+"] Data succesfully scraped.")
12 logging.info("["+str(ID)+"] Bulk writing data on Mongo.")
13 resultCollection.bulk_write(upserts)
14 logging.info("["+str(ID)+"] Data saved succesfully.")
```

It should be noted that the process is not the fastest possible, due to a wait time of about 4 seconds per video, necessary for both preventing IP blocking by the company and to render correctly all of the comments. A parallelized implementation of the code, made possible with the use of the `mpi4py` library as before, allowed us to run multiple rendering instances.[4]

[4]Unfortunately, Google Chrome rendering engine does not support hyperthreading.

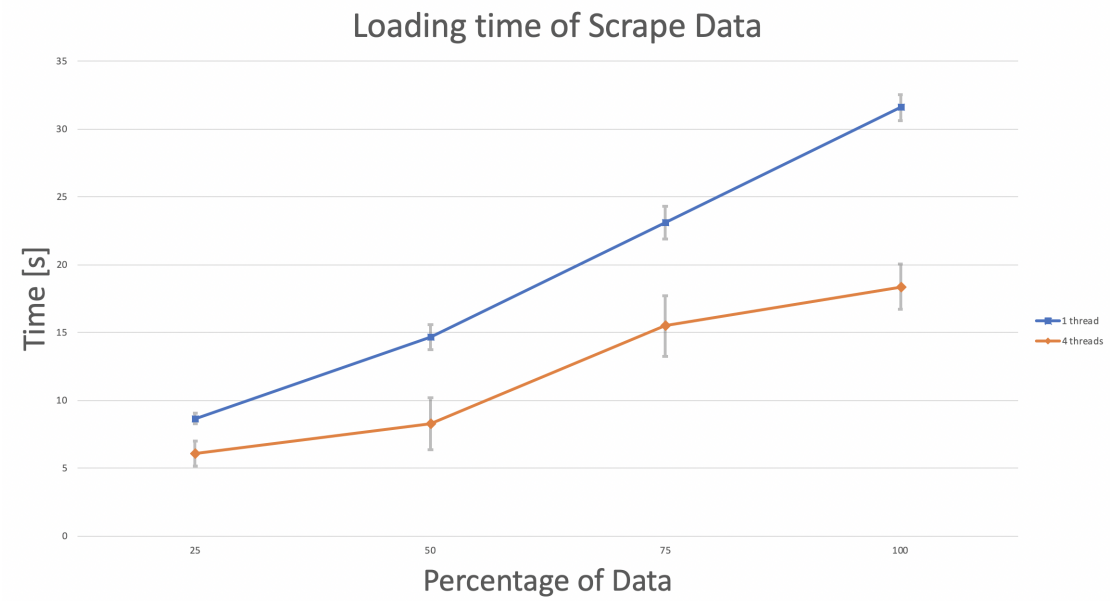


Figure 1.4: Load time for different percentages of the dataset. The process is reported when run on 1 thread or 4 thredas, to show the handling of parallel computazation.

Testing

Unfortunately, doing testing with the above code proved to be too big of a burden for us. Indeed, the whole scraping procedure ended up taking more than two days of full work, with the use of 10 total threads. But, due to the systematic nature of the scraping time, we thought that a simple loading test should still be representative of the handling of these data by our architecture.

Figure 1.4 shows the trends for the loading of the dataset obtained by loading the scraping data onto its collection. We decided to depict the process when run on one thread or 4 threads. As one can see, where the non-parallelized one shows a somewhat linear trend, when the code is parallelized there is a noticeable improvement in both total loading time and trend.

1.4 Data quality assessment

After loading all of the data from the various sources, we were interested in assessing the quality of the data just collected. We were particularly interested in completeness and accuracy.

We show in the following Section a main problem related to calculating accuracy in our data.

1.4.1 Data accuracy: How to measure?

To measure the **syntactic accuracy** of our data, we need to find a way to decide whether a certain field has been reported correctly. In other words, it must have the same value as on the website.

For many fields, that do not change over time, this task is fairly easy, as we are able to retrieve our data from two different sources. If both sources report the same value, then we can assume that the value is correct.

What if they don't? As we read on the Kaggle page of our dataset (https://www.kaggle.com/datasnaek/youtube-new), the data reported in the .csv files comes directly from the official Youtube API. This provides very accurate and complete data. Since the Kaggle dataset comes directly from official sources, we can assume it has maximum accuracy, and, for this reason, it will be considered as the most reliable source.

1.4.2 Youtube videos are dynamical entities

What stated in the previous chapter is true for those field that do not change over time, for example the upload date or the thumbnail link.

Any other field, though, including the video title, description, channel name and obviously views, comments, etc..., can change over time.

We were not able to establish a method to determine the accuracy of these values, since they could vary at any given time. Given this analysis, we still considered data whose origin is in the Kaggle dataset to be as accurate with the Youtube website as possible, at the time of extraction.

1.4.3 Data Completeness

On the other hand, we could easily assess the completeness of our data with two simple scripts, namely `kaggleCompleteness.py`, `scraperCompleteness.py`. These are made to test the completeness of, respectively, the kaggle collection and the scraper collection.

To run, one must first set up the relative config file, named `configMongoQuality.yml`. In this file, the field host corresponds to the IP address of the machine hosting the `mongos` instance, named CASE.

The `database` field corresponds to the name of the database we are referring to. Then we have two field that allow us to tune our procedure properly, in order to get adeguate results.

- The field `sample_size` determines the number of data to be sampled for the completeness measure.
- The field `field_names` determines which field the completeness will be measured on

Once the config file had been set up, we run the scripts.

The structure is fairly simple: each scripts establishes a connection to the mongodb server, then takes a sample of the data from each collection and counts the percentage of `null` or “Unknown" values in the fields reported inside the configuration file.

All the procedure involves only one round trip to server for querying, as we show in the Snippet below:

```
1 sample_dict = kaggleCollection.aggregate(
2         [{
3             "$sample" : { "size" :  SAMPLE_SIZE }
4         }]
5     )
6 result = field_completeness(sample_dict, SAMPLE_SIZE,FIELD_NAMES)
```

As shown, the result of the query is passed to a function called `field_completeness` that returns a dictionary with completeness values and the errors associated with the sample. The function is implemented as follow:

```
1  def field_completeness(doc_list, doc_list_length, field_names):
2      mean_std_dict = { key : 0 for key in field_names}
3      # iterate on all the documents
4      for doc in doc_list:
5          for key in mean_std_dict.keys():
6              if doc[key] == None or doc[key] == "Unknown":
7                  mean_std_dict[key] += 1
8
9      for (key, value) in mean_std_dict.items():
10         p = value/doc_list_length
11         mean_std_dict[key] = { 'mean' : "{:.4f}".format(p),
12                                'std' : "{:.4f}".format(numpy.sqrt(p*(1-p)/
                                      doc_list_length))}
13     return mean_std_dict;
```

The arguments of the funcion are the cursor returned by the query, the sample size and the name of the fields considered. From line 2 to line 7, the function loops through the documents to count `null` or “Unknown" values. From line 9 to line 12, the percentage on the total and relative standard error are calculated.

Data completeness of the Kaggle dataset

The results obtained for the main fields of our records are reported below.

Table 1.1: Completeness results on the Kaggle dataset on a sample of 10000 records

Field_name	Percentage of null values	Error
Title	0.00%	0.00%
Likes	0.00%	0.00%
Dislikes	0.00%	0.00%
Views	0.00%	0.00%
Date	0.00%	0.00%
Thumbnail Link	0.00%	0.00%

Data completeness of the scraping dataset

Table 1.2: Completeness results on the scraping dataset on a sample of 10000 records

Field_name	Percentage of null values	Error
Title	4.7%	0.7%
Likes	5.6%	0.7%
Dislikes	7.8%	0.9%
Views	5.5%	0.7%
Date	0.3%	0.2%

As shown in Tables 1.1 and 1.2, the data coming from the Kaggle dataset hasn't shown the presence of any null value.

On the other side, the scraper isn't always able to retrieve data from all the fields. The percentage of null values, however, for all fields analysed, less than 10%. The most incomplete field is shown to be `dislikes`.

How to improve

It is theoretically possible to use the data coming from the Kaggle dataset to replace null values, however, we decided to keep the null values for all the fields that can vary over time, in order to preserve data **temporal internal consistency**, since the percentage of null values is not high enough to forbid analysis.

1.4.4 Videos characteristics change over time

Not only views, likes, dislikes and comments change with the passing of time. In Youtube landscape, it is now common practice to edit video titles and description when needed, so that all aspects of a Youtube video can vary over time. Reasons for this could be multiple. For example, a video title could be changed for advertising reasons. For example, in 2019, the famous rapper "Tyler, the Creator" uploaded a video on his Youtube Channel in order to promote his upcoming album, which contained a preview of a song and was titled "—-'- ——". Given the rapper's fame and the enigmatic title, the video quickly reached 2 million views in just 24 hours. After a week, the video title has been changed to the actual song title, namely "IGOR'S THEME" (https://www.youtube.com/watch?v=CEVXcP3VC3Y).

This is only one of many examples that shows how a video title can completely change over the course of small periods of time, but there could be many others in any video category, from news to gaming.

In order to efficiently conduct our analysis, we wanted to have an idea of the magnitude of this phenomenon with a script (`dataConsistency.py`). This script takes sample values from both the Kaggle and the scraper dataset and checks, for each videos, if any field has changed over time. First of all, we took a sample of both databases, grouped by `id` and, for each field, save all the different values encountered.

```
sample_dict = kaggleCollection.aggregate(
      [{
        "$sample" : { "size" :  SAMPLE_SIZE }
      },{
        "$group" : {
                    "_id": "$id",
                    "ref_dates" : { "$addToSet" : "$ref_date" },
                    "titles" : { "$addToSet" : "$title" },
                    "channel_names" : { "$addToSet" : "$channel_name" },
                    "descriptions" : { "$addToSet" : "$description" },
                    "dates" : { "$addToSet" : "$date" },
                    }
      }]
    )
```

Then we simply checked whether or not there was a variation over time, by retrieving the length of each set after having removed `null` or "Unknown" values.

The results are reported below, in Table 1.3. As shown clearly, both titles and description are commmonly changed over the course of time. Channel names, instead, seldomly change. Finally, of course, once the publish date is set, it cannot change.

It is obviously impossible for us to decide whether changes in titles and description are caused by errors inside the data or by actual changes to the video. In this case, data accuracy cannot be

Table 1.3: Variation of values in fields for each video

Field_name	Percentage of values that have varied over time	Error
Title	12.7%	0.8%
Channel Name	1.2%	0.4%
Description	30.1%	1.9%
Publish Date	0.0%	0.0%

measured effectively for the first 3 fields.

On the other hand, we can see that the publish date of videos never changes over time. We are sure of the fact that, once set, this value cannot be changed, so, if there was a change in values, it had to be caused by an error inside the data.

For the sample taken, no dates are reported differently, and we can conclude that, in virtually all records, this field is accurate.

1.5 Data enrichment & Integration

Once the data was loaded onto the document DBMS and some quality controls were performed, we deployed some form of enrichment and integration in order to obtain a final collection, onto which to build our analysis.

All of this was performed directly on the document collections present in our distributed Mongo architecture, with the use of Python scripts to allow replicability.

1.5.1 Initial enrichment

As a first step, we decided to focus on enriching separate collections reciprocally. Our process can be divided into three steps, which can be performed independently of one another.

- Enrich the Kaggle collection with categories from the scraping one.
- Enrich the scraping collection with the region code from the API one.
- Enrich the scraping collection with the publish `date` from the Kaggle collection.

We explain hereafter in more detail these steps.

Enrich Kaggle with scraping

During the scraping phase, as described in Section 1.1, we obtained information regarding the `category` of a video, e.g. *People & Vlogs* or *Music*. On the other hand, the Kaggle dataset, as present on the SQL database, has information regarding `category_id`. While these two may seem to represent ther same concept, we found that they unfortunately depict slightly different information. In order to avoid any conflicts, as a course of action we decided best to choose between one of the two fields: we opted for the `category` from the scraping.

Even though this information could at times be unreachable, for mere limitations of the scraper, it was consistent along all of the data scraped. Indeed, if we would have chosen to keep only the `category_id` from the Kaggle dataset, we would have encountered problems when dealing with data scraped using the API collection.

We do not report the code used in this instance, for it consists of a simple insertion of a field. Again, even in this case, we deployed parallelization of the scripts and the use of configuration files.

For this one, we decided not to run any tests: they would have been equal to the ones of other simple updates, which will be shown in the coming sections.

Enrich scraping with API

For this instance, the process performed was quite simple and fast. The only other field beside `id` present in the API collection is `regionCode`, information abstent in the scraping collection. The process consisted then only of a simple query over the scraping collection and, when there was a match between the `id` in the API and the one the scraping, a field would be inserted.

Again, we did not perform a test for this instance, for the same reason as the one explained before.

Enrich scraping with Kaggle

In this instance, we deployed a conflict resolution techinque, with the aim of making the collections as consistent with one another as possible.

Both the Kaggle and scraping collections presented the same field, named `date`, which indicates the day of publish of a video. Due to the immutable nature of this information, we believe the two dates should coincide. After performing quality test, as described in Section 1.4, we believed best to resolve conflicting information between the two collections.

We believe that the `date` reported on the Kaggle collection, which comes in some form from the Youtube API, is probably more accurate than the one present on the scraping, which depends on our library's precision.

We then run a simple script that would insert in the scraping collection the `date` from the Kaggle one, regardless of its initial presence.

In this case we decided to perform some tests, which we think might be representative for whole enrichment process. We report in Figure 1.5 the results obtained. Even though this one was performed using only one thread, for sake of somplicity, the results are quite remarkable. It should be noted that statistical differences are due both to the internet connection with which the test was performed and how each collection (we had to use different collections for each percentage step) was sharded.

1.5.2 Integration into a single collection

After all of the enrichment had been performed, our next step was to develop some scripts that would put the data together into a single collection, where the information would be finally available. This procedure consisted of a step where we decided which schema to use for the final data and, afterwards, how to implement it.

Integrated schema

We weighted 3 main choises for a final integrated schema:

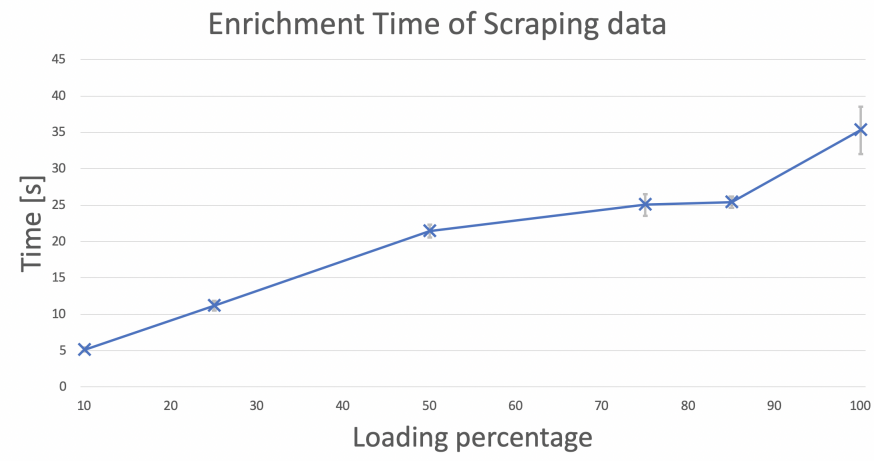


Figure 1.5: Loading times for different percentages of both datasets.

- Fully embedded documents. In this case, each document would have a field `ref_date` that would indicate all of the times its information was registered and, for each field with mutable information, all of the values, e.g. an array of values for the views.
- Fully referenced documents. In this case, we would join the two collections without operating any field changes. The documents would than be referenced using the `id` key.
- A hybrid approach.

We decided that, for the way the data presented to us, the fittest method was a hybrid approach. In particular, we used as keys to organize the collection both the `id` and the `ref_date`. This way we managed to retain historical data in separated documents.

But we also decided to insert some fields, more specifically `comments_disabled`, `video_error_or_removed` and `ratings_disabled`, into those documents that lacked them.

Some other embedding, obtained through calculated fields, will be explained later.

Code

In the Snippet below we show part of the script used to achieve the integration.

```
upserts = [ UpdateMany(
{'id':kaggleDoc["id"], 'ref_date': kaggleDoc['ref_date']},
    {
        '$setOnInsert': kaggleDoc,
    }, upsert=True) for kaggleDoc in cursorKaggle]

upserts2 = [ UpdateMany(
    {'id':kaggleDoc["id"], 'ref_date': kaggleDoc['ref_date']},
    {
        '$set': {"comments_disabled": kaggleDoc["comments_disabled"],
```

```
        "video_error_or_removed": kaggleDoc["video_error_or_removed"],
            "ratings_disabled": kaggleDoc["ratings_disabled"]}
    },) for kaggleDoc in cursorKaggle]
upserts.extend(upserts2)
logging.info("Updating documents.")
try:
    scraperCollection.bulk_write(upserts)
except pymongo.errors.BulkWriteError as bwe:
    print(bwe.details)
    raise
```

In this part, we show the update operations that were performed. Even though we are sure our datasets didn't contain duplicates, we opted for `UpdateMany` operations to render our code the most usable as possible by others. Unfortunately, we had to separate the two queries, due to limitations in the Mongo language. The operations were nevertheless executed with a bulk writing, which, as previously said, minimizes the number of connections with the server.

As with most of the scripts produced for this project, we enabled parallelization of this integration in as many threads as one needs, even on cluster computing.

We decided to update the scrape collection as opposed to creating new one for sake of optimization: even though we didn't lack any storage space, someone who did might find our solution desirable.

Testing

We have tested this procedure as required for multiple percentages of our database. In Figure 1.6 we show the trend for the loading times. One might notice that there is a drop after 75%: we believe this is mostly due to internet connection uncertainty. Indeed, the bigger the data to load, the most uncertainty to connection we encountered.

Nevetheless, we believe the trend is desirable and shows the effectiveness of our architecture.

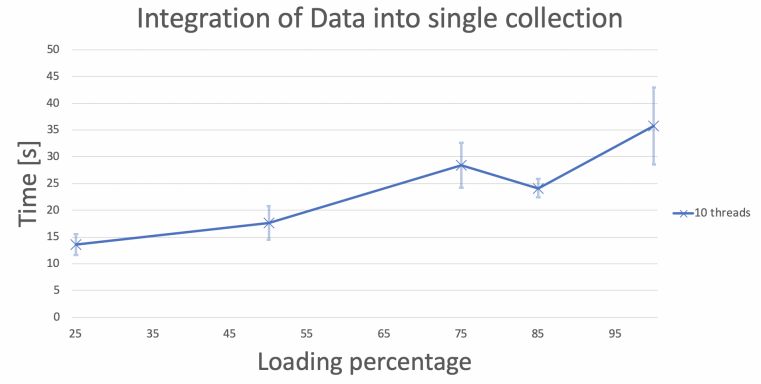


Figure 1.6: Loading time for different for different percentages of both datasets.

1.5.3 Final enrichment

Once we managed to have all of the data integrated into a single collections, we decided to apply some final enrichment. In particular we applied some calculations to add useful information for future analysis. In particular, we created a few embedded fields from calculations. These include:

- `thumbnail_as_bytes`: we added the information regarding the video thumbnail. We decided to save all of the data from it, as *UTF-8* codified string of base64 bytes.
- `title`: instead of having only the title, we created a subdocument containing:
  - `text`: the actual text of the title.
  - `length`: its length.
  - `first_letter_is_capital`: a `True`/`False` field for the capitalization of the first letter.
  - `perc_capital_letter`: percentage of capital letters over the length of the title.
  - `slash`: number of slashes, `/`.
  - `pipe`: number of pipes, `|`.
  - `dash`: number of dashes, `-`.
  - `exclamation_mark`: number of exlamation marks, `!`.
  - `asterisk`: number of asterisks, `*`.
  - `question_mark`: number of question marks, `?`.
  - `e_comm`: number of commercial Es, `&`.
  - `hashtag`: number of hashtags, .
  - `featuring`: number of times the word *featuring*, or some forms close to it, appeared.
- `description`: again, instead of having a single value, we embedded a document with:
  - `text`: actual text of the description.
  - `hashtags`: number of hashtags in the description.
- `frac_likes`: number of likes over the views.
- `frac_dislikes`: number of dislikes over the views.
- `domimantColor`: the dominant color in the thumbnail. To get this value, we built a function using the library `scipy`.
- `comments`: even though each comment was already by itself a document, we report the fields that were added to each one:

  - `polarity`: the polarity of each comment (a value between -1, very negative, and $+1$, very positive), calculated using the library `textblob`.
  - `subjectivity`: similar to the positivity, between 0 and 1.

- `mean_polarity_comments`: the average of all polarities.

After all of these fields were calculated, the data were ready to be analysed.

Scripts

Due to the loading time for the thumbanail enrichment, we decided it was best to separate this final enrichment in two different scripts. We would like to point out how this was done for purely operative reasons, and the two can be joined together.

We report in the Snippet below not the update task done to add the `thumbnail_as_bytes` to the integrated collection, but the function that takes the url for the thumbnail and returns the *UTF-8* encode string.

```
url = "https://i.ytimg.com/vi/"+diz["id"]+"/hqdefault.jpg"
response = requests.get(url)
base64_bytes = base64.b64encode(response.content)
base64_string = base64_bytes.decode("utf-8")
return base64_string
```

This script utilizes the library `base64` in order to convert from bytes, which are not json seriazable, to strings.

Below is the update bit of the second script, used to add all of the other fields. As one may notice, two different queries were run. Indeed, the first one adds fields for every element of the collection, while the second can only calculate comment information for those who have the `comments` section.

```
upserts = [ UpdateOne(
    {'id':scraperDoc["id"], "ref_date": scraperDoc["ref_date"]},
    {
        '$set': {
                "title": title_analizer(scraperDoc["title"]),
                "description": hashtags_analizer(scraperDoc["description"]),
                "frac_likes": frac_likes_analizer(scraperDoc),
                "frac_dislikes": frac_dislikes_analizer(scraperDoc),
                "dominantColor": domimantColor_analizer(scraperDoc)
                }
    }) for scraperDoc in cursorScraper]
upserts2 = [ UpdateOne(
    {'id':scraperDoc["id"], "ref_date": scraperDoc["ref_date"], "comments": {
        '$exists': True}},
    {
```

```
15          '$set': {
16                      "comments": [{
17                                      "text": comment["text"],
18                                      "likes": comment["likes"],
19                                      "replies count": replies_count_analizer(
                                            comment),
20                                      "user": comment["user"],
21                                      "polarity": polarity_analizer(comment),
22                                      "subjectivity": subjectivity_analizer(
                                            comment)} for comment in scraperDoc["
                                            comments"]],
23                      "mean_polarity_comments": mean_polarity_analizer(
                            scraperDoc),
24                  }
25      })for scraperDoc in cursorScraper]
26  upserts.extend(upserts2)
27  logging.info("Updating documents.")
28  scraperCollection.bulk_write(upserts, ordered=False)
29  cursorScraper.close()
```

In order to make the process the most efficient as possible, we gave the bulk operations unordered, which, according to the offical documentation, optimizes the operations in a sharded collection.

Again, this code can be run with multiple threads and on different computers simultaneously.

For this operations we decided not to run any testing, for two main reasons. Firstly, the process can be quite long, due to calculations and uploading of the thumbnails. Secondly, we didn't find any procedural difference with other updates and queries run previosuly, hence it would have been a longer repetition of previous tests.

Final result

After all processes described in this Section were completed, we obtained one final collection, onto which operate our analysis. We report the structure, as given by the mongo query `getShardDistribution`, for the final collection, identified as `scrape` for reasons already mentioned.

```
Shard manzoni at manzoni/168.61.100.92:27020
 data : 1.62GiB docs : 58226 chunks : 40
 estimated data per chunk : 41.5MiB
 estimated docs per chunk : 1455

Shard pascoli at pascoli/52.136.247.24:27020
 data : 1.37GiB docs : 43195 chunks : 39
 estimated data per chunk : 36.14MiB
 estimated docs per chunk : 1107

Shard petrarca at petrarca/94.245.111.63:27020
 data : 1.55GiB docs : 49969 chunks : 39
 estimated data per chunk : 40.73MiB
```

```
 estimated docs per chunk : 1281

Totals
 data : 4.54GiB docs : 151390 chunks : 118
 Shard manzoni contains 35.63% data, 38.46% docs in cluster, avg obj size on shard : 29KiB
 Shard pascoli contains 30.25% data, 28.53% docs in cluster, avg obj size on shard : 33KiB
 Shard petrarca contains 34.1% data, 33% docs in cluster, avg obj size on shard : 32KiB
```

As shown above, the total collection exceeds the `2GB` threshold for the project and the collection is in itself successfully distributed.

1.6 Connection to Tableau

After all of the proceedings described in the previous Sections, hence with the final collection enriched and integrated, we encountered the problem of using tools to analyse such data. Due to the requirments for the project, i.e. the use of the **Tableau** platform, we decided to connect the latter directly to our DDBMS.

Fortunately for us, both platforms have thought at the same question previously, thus having a quite simple answer.

As mentioned in the online guide, our process was divided into 2 main steps:

- Create a SQL interface for the collection. This was achieved with the use of **Mongo BI connector** and `mongosqld`, a connector that builds a SQL schema on top of a document collection.

  Unfortunately, the Linux distribution used in our server (**Ubuntu server**) did not support one of the libraries, `OpenCSS`, necessary to run the BI connector. Thus, we were forced to install this instance on our local computers.

  Once that was achieved, we conncted the `mongosqld` instance to our remote server.

- `MongoDB BI Connector`, a different software than the one above, as it is an add-on software for Tableau. This allowed for communication with the `mongosqld` instance, running locally.

When the above steps were executed, the collection could be queried via SQL inside Tableau to obtain tables for analysis. It should be noted that a particular either schema can be defined or the MongoDB BI connector will try to automatically fit one: we found that the latter worked flawlessly with our collection, and thus kept it.

Chapter 2

Data Visualization

We briefly explain in this chapter the procedure related to data exploration, i.e. the visualizations used to garner insight about the data at hand, and user tests.

2.1 Data Exploration

2.1.1 Defining popular

For the exploration, we connected directly to the final mongo collection, as detailed in Section 1.6.

While the term **popular** is quite wide and can comprehend multiple factors, we decided to design a separation based on the number of views. For this purpose, we show in Figure 2.1 a boxplot of the views in one category, as an example.

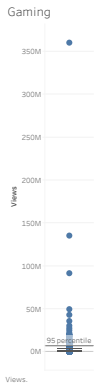


Figure 2.1: Box plot for the views of the **gaming** category.

Just by looking at this representation, not too much information can be extracted: as one can see,

the presence of a few outliers with a large number of views makes difficult to distinguish much. Consequently, we portrayed the same information in a lineplot with base-10 logarithmic scale along the vertical axis. In Figure 2.2 we depict such graph for only the **Gaming** category, for similar trends are present in the others.

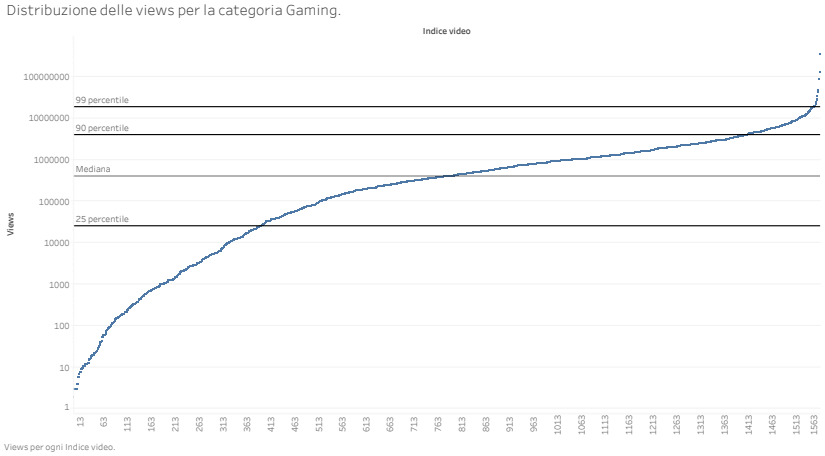


Figure 2.2: Views of gaming category in log scale. It could be noted horizontal axis that represent percentiles of order 0.25, 0.5, 0.90, 0.99.

In the light of these considerations, we decided to classify a video as popular when its views were greater than the percentile of 0.90 of the distribution of its category. On the other hand, we believe a good sample space for less popular videos could be the 0.25 to 0.5 percentile range. We decided not to take videos in lower percentiles due to the impact of very unpopular ones, e.g. videos with around a few hundred views have very little comment information and usually have automatically generated thumbnails.

With a sample of videos for both classes of interest, we then tried to outline differences in the categories using *title*, *thumbnail* and *comment* information.

2.1.2 Title analysis

As described in Section 1.5.3, our dataset had information regarding the number of capital letters and “wildcards" in the title. We tried to visualize possibile differences in these fields between the popular and unpopular sets, via computing mean values and standard deviation for three characteristics: length of the title, number of capitals and total number of “wildcards" in it. In Figure 2.3 we show what we have evaluated for the *Gaming* category in the popular set, as reference for all.

All three distributions are, as expected, asymmetric and, in general, we find that the longer the video, the higher the number of “wildcards" and capital letters.

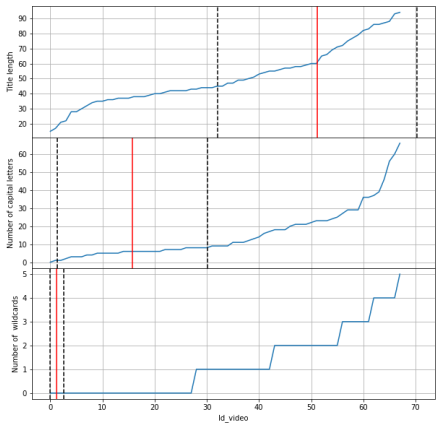


Figure 2.3: For each popular video in gaming category, we plot title length, number of capital letters and number of wildcards. The red continuous vertical lines represent the average values, while the dotted ones are average $\pm$ standard deviation.

2.1.3 Thumbnail analysis

The second element of interest was the thumbnail, i.e. the placeholder image for the video when shown on the main/search page. Our dataset, as detailed in Section 1.5.3, had an already calculated field for the dominant colour in it. We would like to give emphases to the difference between dominant colour, i.e. the most common one, and average colour: the latter would have been meaningless to us.

In particular, we decided to cluster the colours within a finite domain composed of: red, orange, yellow, cyan, green, purple, black and white. What we discovered is depicted in Figure 2.4: those are the most present colours in all of the thumbnails analysed, be them popular or unpopular.

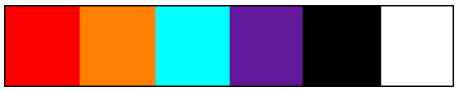


Figure 2.4: Dominant colors in our dataset of popular/unpopular videos.

2.1.4 Comments analysis

The third element that we wanted to explore were comments, in particular their polarity, as calculated in Section 1.5.3. The higher is the polarity, the higher is the number of positive words in a comment (such as great, beautiful etc..), while the lower is the polarity, the higher is the number of negative words (such as bad, worst, awful etc..).

Unfortunately, due both to limitations in the procedure, as sometimes the text was not labelled

properly, and in the distribution of these, we decided not to proceed. In our explorations, we found little to no significant difference between the various categories and sample spaces. We opted for not depicting these exploratory visualizations for they were too complicated to be useful on paper.

2.2 Infographics

We report the two infographics produced in the project. We think they achieve their goal of showing some information regarding non-video data on Youtube. We explain in the following Sections their meaning and tasks.

2.2.1 Questions to answer and Target

With our visualizations we would like to emphasize differences between popular and unpopular videos regarding context elements of Youtube videos, specifically title and thumbnail ones.

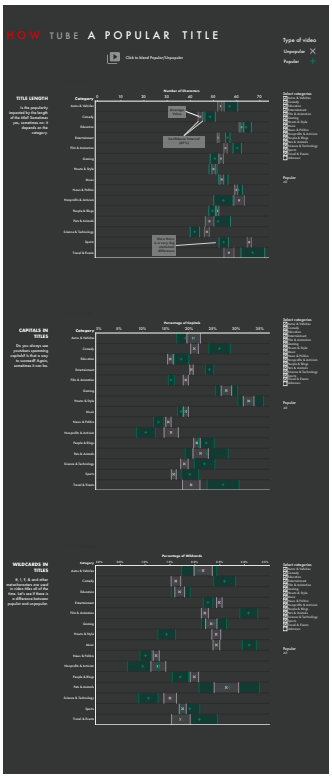
At the same time, in some cases, we thought interesting highlighting differences between the categories at hand. Nevertheless, this has been a secondary focus and thus not our main goal.

Given the scope of our analysis and the information obtained, we had to focus our attention on both content creators and end-users who posses some base statistics understanding. In order to study our target, we defined a **proto-person**.

PROTO-PERSON	Luca, 24 years old, University student from a scientific faculty.
Hobby	Youtube gamer
Needs	He wants to gain more popularity in his Youtube videos
Habits	He uploads videos of himself playing Minecraft daily
Hopes	He wants to become a top-performing content creator, and thus wants a way to make his videos more popular
Worries	Details matter and he doesn't want to waiste good content that cannot reach a large audience due to contextuals details, e.g. title and thumbnail.

Just like our pro-person, many others could be interested in advancing their Youtube popularity through contextual means: we hope our infographics are capable of delivering such information.

2.2.2 How Tube a Great Title



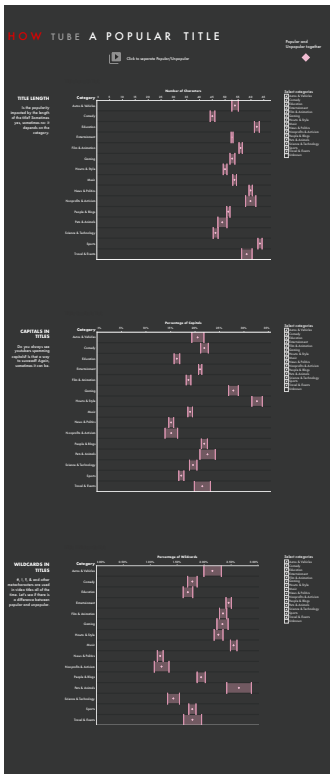


Figure 2.5: First Visualization

Description

This data visualization wants to show the characteristics of the title, in order to understand if there could a difference between popular and unpopular ones. Especially, we will focus on:

- average length
- average number of capital letters
- average number of wildcards

It consists of 3 plots, duplicated in two visualizations. On the left is described the difference between popular and unpopular, represented through the use of different marks and colors, while on the right is depicted the same information, without distinction in the two sample spaces. In particular, each plot represents characteristic: *title length*, *percentage of capital letters per character in the title* and *percentage of wildcards per character in the title*. In particular, each one of these plots portrays in the *y-axis* the category and on the *x-axis* the value measured.

Each mark represents the average value: on the left distinct between popular and unpopular videos; on the right with all videos together. The bars around each mark are **confidence intervals** at 67%.

Heuristic evaluation

Some users (3 people) have highlighted the following problems, after which we have modified our infographic. We report briefly their observations.

1. Categories on the *x-axis* are not intuitive. We decided to flip the axis.
2. The color red for popular is misleading, as mostly associated with something bad.
3. It is not immediate to spot the *x-axis* if below. We then moved it on top of the plot.

Psychometric test

We built a psychometric scale that probed the following characteristics in our visualization: utility, informativity, intuitiveness, clarity, originality and completeness. We decided to take such fields from the **Cabitza-Locoro scale**, with the use of personal ones as well. [1]

Each characteristic has a score from 1 to 6. For each one we made 3 stacked bar chart: the first one represents the % of answers between $\{1,2\}$,the second one the % of answers between $\{3,4\}$ and the third one the % of answers between $\{5,6\}$. The results of our test are shown in Figure 2.6 and 2.7

[1] http://progettoyoutube.altervista.org/QUESTIONARIO.html

We briefly explain the procedure we carried out. Each user was required to answer all of the questions; once completed, we obtained and saved data as HTML tables, which we used to execute the analysis.

Each table contained both the information regarding how the user answered the questions and personal once, e.g. age group, education and work/study field. Indeed, most of the people we submitted were in the 20-30 age bracket, with computer-science/scientific knowledge.

We would like to point out that the black dots in Figure 2.6 represent the ration between the answers 3 and 4 to their sum, with its size being the confidence interval (confidence level: 67%) to such calculation,

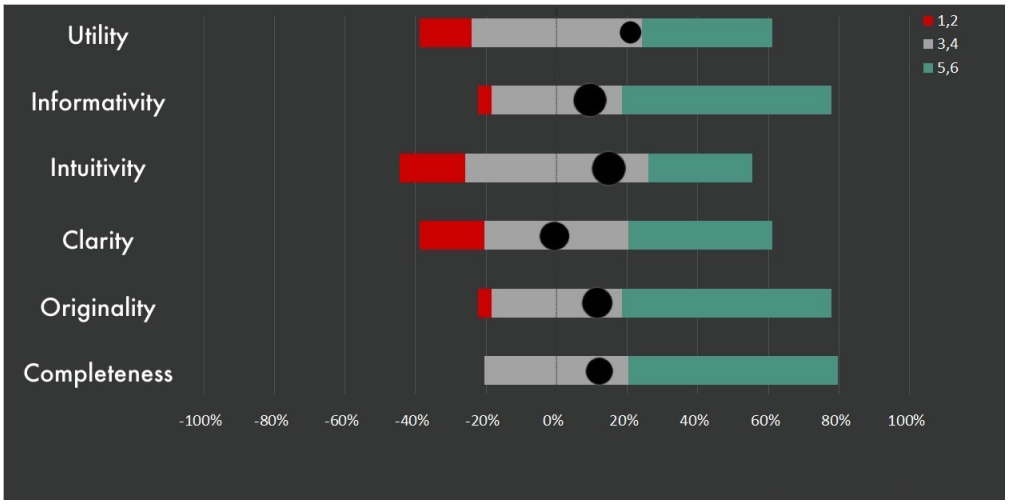


Figure 2.6: Stacked bar charts for each characteristic.

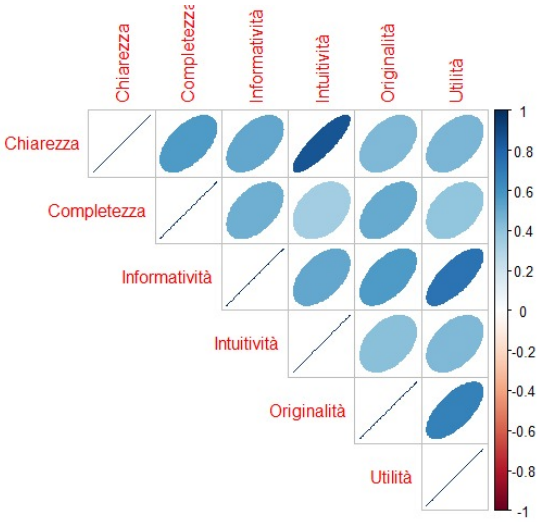


Figure 2.7: Correlogram for the parameters evaluated above.

User tests

We probed the efficacy and communicative power of our data visualization via questions, whose intent was to push 8 users to explore and understand more in depth.

The questions asked are the following ones:

Task In the title length plot, deselect all categories and select only *comedy*.

Task 2.1 What is the average length of the title for the category *sport*?

Task 2.2 What is the size of the confidence interval of the title for the category *sport*?

Task 2.3 What is the size of the confidence interval of the title for the category *music*?

Task 3 What is the average value of capital letters in the popular video for the category *news & politics*?

Task 4 What is the difference in the average percentage of wildcards between popular and unpopular videos for the category *music*?

We then plotted the results in a stacked bar chart, for the answers, and a violin plot, for the time that users took to respond. We show these graphs in Figure 2.8.

The first one shown depicts correct to incorrect answers and their relative error, shown as a overlapping of the bars.

The second one is a violin plot with the times of completion of each task, as described above. The black lines represent the average times taken by us, while the dashed ones represent the standard deviation. We decided to depict this information for it is the *optimal* band if answers.

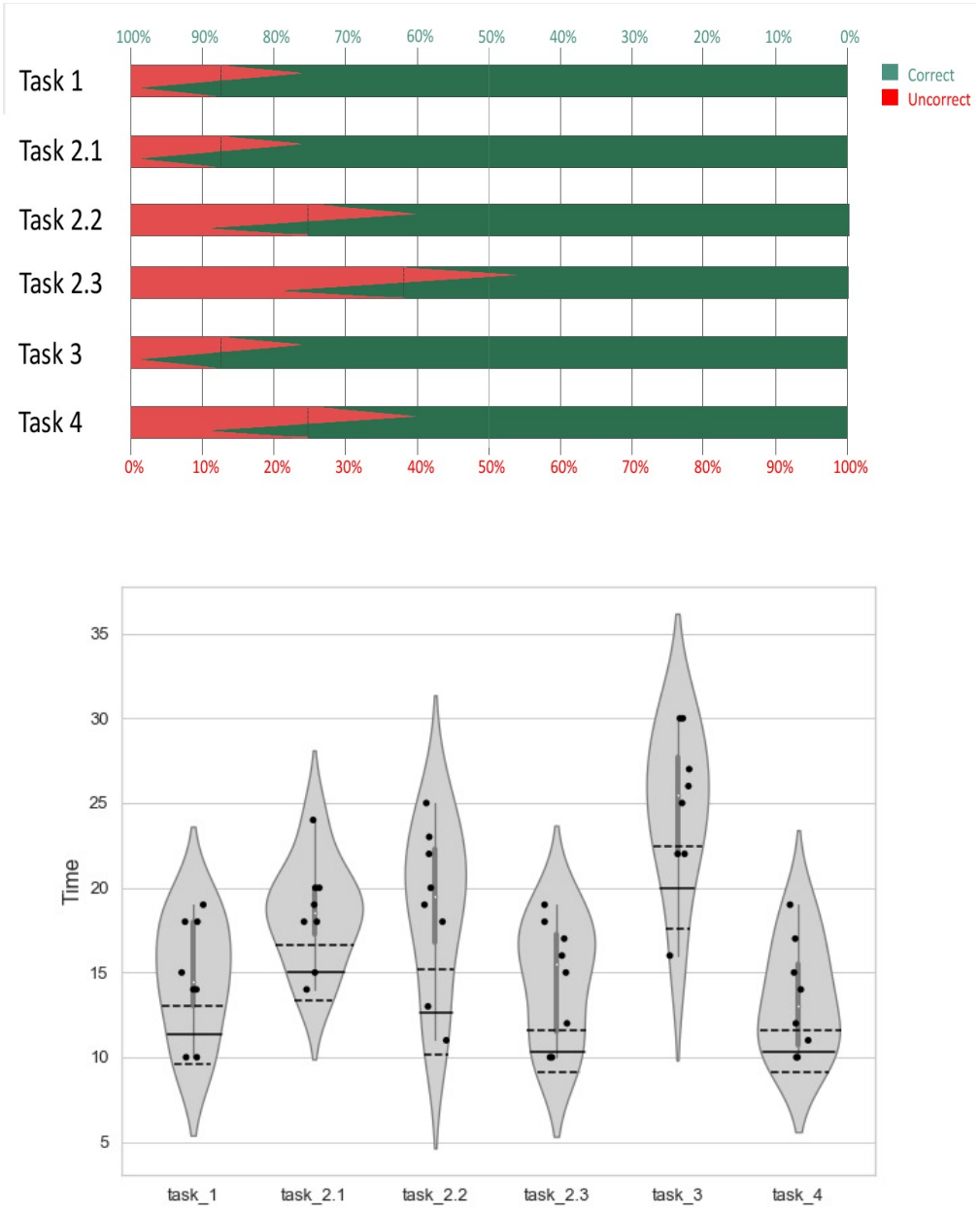


Figure 2.8: First, stacked bar chart of % correct answers with its confidence interval (confidence level: 67%). Then, the violin plot.

2.2.3 How Tube a Great thumbnail

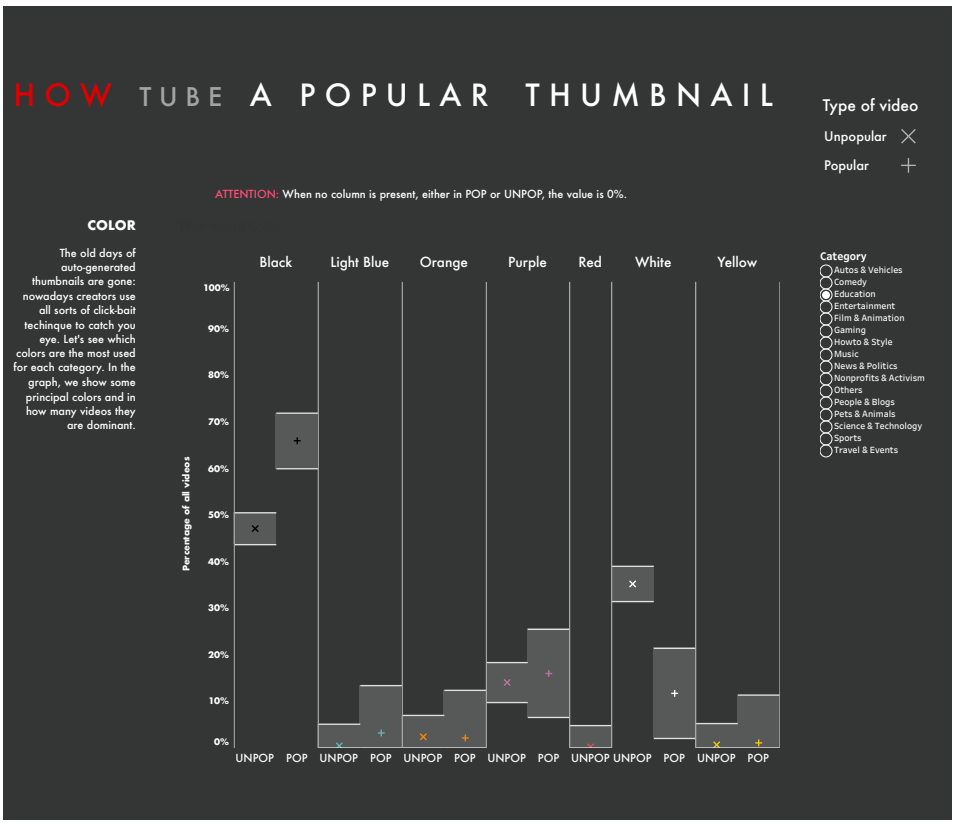


Figure 2.9: Second Visualization

Description

This data visualization wants to show the most frequent dominant color of thumbnails in popular/unpopular videos. Due to the interactive nature of the visualization, we decided to show only on category (*education*). Indeed, if one interacts with it, he or she can change category and see how the different colors are domimant in each one of them.

The infographics is divided by color, whose relative frequency is obtained as described in Section 2.1.3. Each box is divided into two classes: popular and unpopular videos.

Each mark has Y-value that represent the percentage of videos whose thumbnail has that color in its category.

Heuristic evaluation

During the assessment, our strict users, again 3 people, have depicted some problems. We describe here some of them.

1. A color legend, which we depicted, was redundant and took away attention to more important information. We obviously decided to drop it.

2. The words *popular* and *unpopular* on the *x-axis* label were to long and difficult to read. We decided to shorten them.

3. When a column was not present, it was immediately clear why. We added a small warning to clarify.

Psychometric test

We have obtained this test the same manner as the other visualization. We just report in the Figure 2.10 below our results.

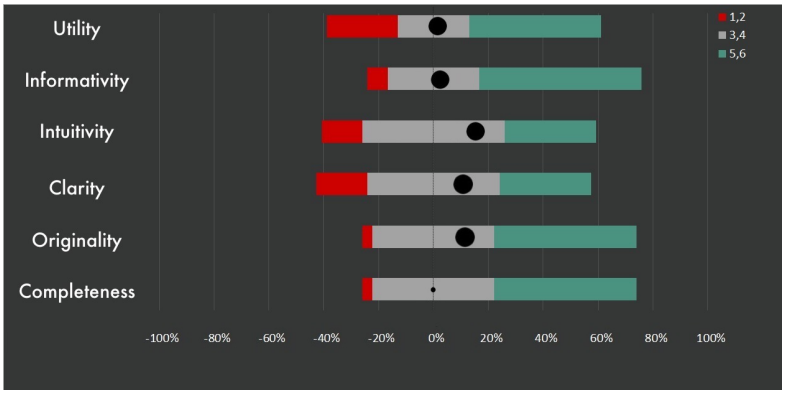


Figure 2.10: Stacked bar charts for each characteristic.

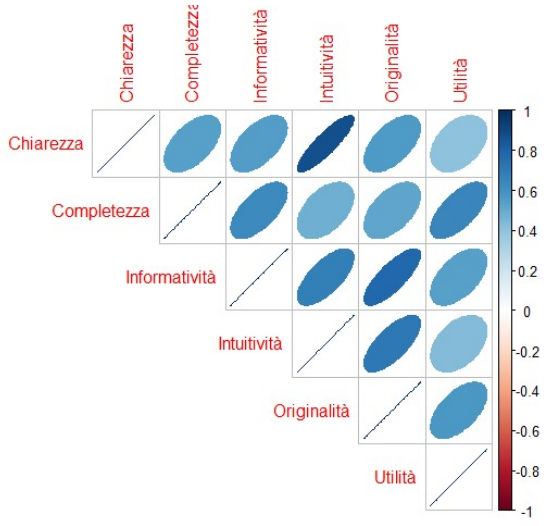


Figure 2.11: Correlogram for the parameters evaluated above.

User tests

Again, we operated in the same manner as the first visualization.

The questions asked are the following ones:

Task 1 In the *gaming* category, how many popular videos (in percentage) have black as dominant color?

Task 2 In the popular *music* videos, is there more with black or orange as dominant?

We then plotted the results as described for the first visualisation. We show in Figure 2.12 the violin and bar stacked plots respectively for this second viz.

As for the meaning of the graphs, they have been produced as previously, in Section 2.2.2

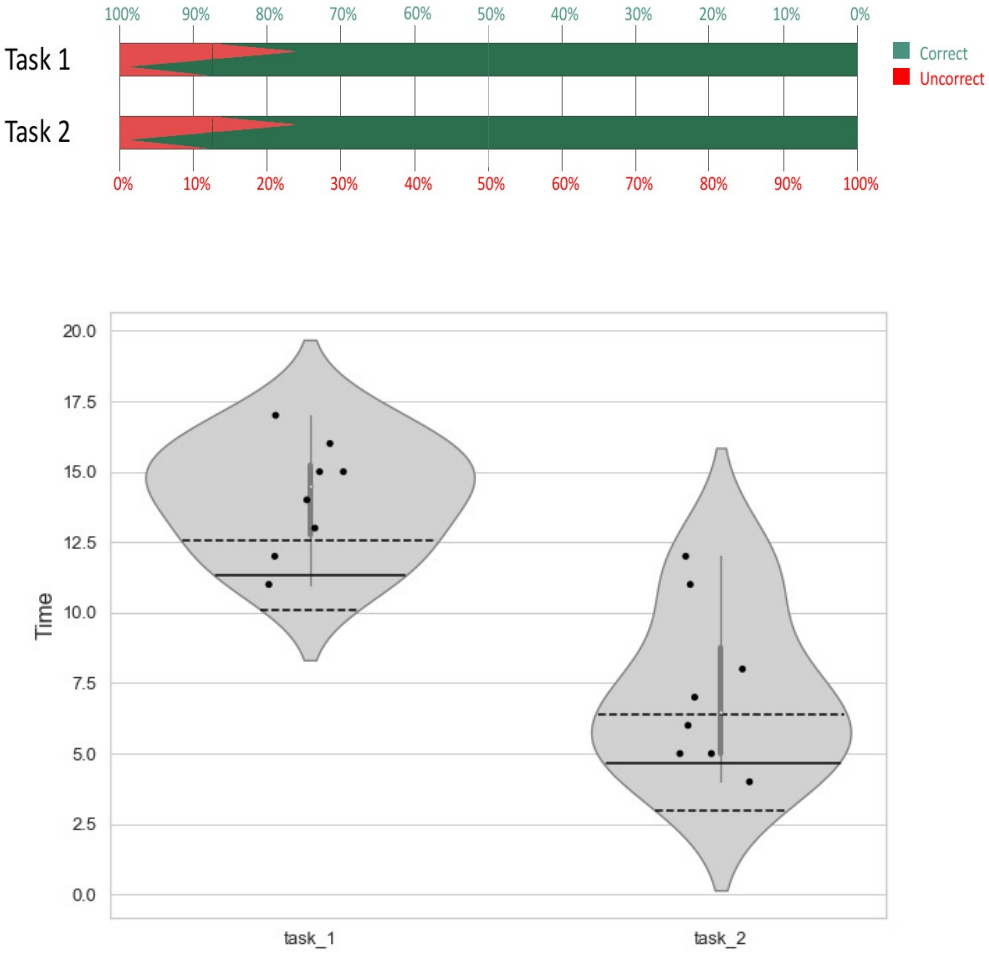


Figure 2.12: First, stacked bar chart of % correct answers with its confidence interval. Then, the violin plot.

Chapter 3

Conclusions

In conclusion, we believe our work has been successful in representing Youtube data, and especially those non-video elements can have an impact in how much videos are clicked and watched. Furthermore, in order to achieve this analytical results, we had to built a very efficient and horizontally scalable architecture, capable of housing even more data than the one needed for our project.

We hope our work can be useful to other people for both the data management and visualization parts. In the former, for reusing part of our code or the architecture basis, and in the latter to give insight for Youtube users and creators.

A

Chapter 4

Roles

We report, approximately, which member of the group did what.

- Leonardo Alchieri. `YoutubeScraper` library for scraping; data loading scripts; data enrichment & integration scripts; connection to Tableau; data management report writing.

  Help setup the architecture.

- Davide Badalotti. Setup DDBMS; data loading scripts; data quality assessment; data management report writing.

  Help with data enrichment & integration scripts.

- Lucia Ravazzi. API scripts; data enrichment scripts; data analysis & visualization; data visualization report writing.

- Pietro Bonardi. Setup DDBMS; data analysis & visualization.

  Help with data enrichment scripts and user test.

- Ilenia Lo Monaco. User test.

  Help data visualization.