UNIVERSITY OF CATANIA
Department of Mathematics and Computer Science
Bachelor of Computer Science Degree Program

Alessandro Resta

Photos Recommender For Tumblr Users

Social Media Management Project Report

Academic Year 2021 - 2022

Contents

1 Introduction **2**

2 Data **4**

3 Proposed solution **6**
3.1 Bag of Visual Words Representation 6
3.2 Kmedoids clustering 8
3.3 Content-Based Image Retrieval application 13

4 Results **14**

Conclusions **19**

Bibliografia **20**

Chapter 1

Introduction

Social media data are growing exponentially and find ways to extract value from them is one of the biggest challenges of this century.
From data it is possible to obtain a variety of information that can be used to improve customers experience.

Let's suppose that a user of the social media Tumblr wants to see new contents related to his own interests. Tumblr do has a built-in recommendation system, but it is based on a **tag** system. So if blog admins forget to add tags to their posts, or simply they don't use them, potentially interested platform users will never see recommendations of such posts and, because of the large amount of data that floods the social media every day, these people may never enjoy those untagged contents.

An interesting problem to solve could be finding a new way to recommend posts to users without using the tag system.

Due to the fact that Tumblr, like many other social media such as instagram and facebook, tends to have more photos than text, the problem will be restricted to only those posts that are actually photos.

The solution proposed in this dissertation is based on similarities between photos (image matching), in particular between the ones a user already liked and the ones a user hasn't yet stumbled on.
If a user likes certain types of photos, say landscape photos, he could be interested in other photos of the same kind, possibly posted by the same author.

To achieve this goal, the following macro steps are necessary:

1. retrieve photos liked by a user;
2. group them together by contents similarity;
3. for each group choose one representative photo and match it with other photos published by the same authors that appear in the group;
4. for each group recommend only the most similar matched photo.

Images matching is a heavy task, for this reason it involves only one photo in the cluster and not all of them. Remember that the chosen photo is part of a specific photos category, so it can represent its entire cluster, saving some computation.

Following sections will describe more accurately this process, explaining concepts and algorithms used in the implemented solution.

The adopted programming language is **Python**, which is plenty of libraries that make working with images (but data in general) smooth and easy.

Chapter 2

Data

Data used in this project are divided in: liked and not liked by a user.
Both are retrieved thanks to Tumblr OAuth2 API calls (see the official documentation [1]). My previous tutorial about how to use Tumblr API covers deeply this part.

For the sake of simplicity, a class named **OAuth2** is created inside the script **oauth2.py**, to handle the OAuth2 authorization flow and define a method **query()** that simplify data requests by passing only the end point (where data are located) as parameter.

Next another script called **tumblr_data.py** is created and defines a class named **TumblrData**. This class provides two methods:
retrieve_liked_photos() and **retrieve_candidate_photos()**.

The first one does the following:

- retrieves only the photos liked by the user;
- downloads them using a function in the same script called **download_photos()**;
- returns a **Pandas** dataframe [2] with: paths of local photos and authors as blogs name.

The second one differs only on the fact that, given a list of blogs name, it retrieves the last **n** published posts, containing photos not already liked by the user. These last are referred to as **candidates photos** and are retrieved from blogs that falls inside the same cluster (clustering will be discussed later).

n is a parameter of the method and can be changed. Note that a post of type photo can potentially have multiple photos in it, so n is referred to the number of posts (not single photos).

Chapter 3

Proposed solution

Previous chapter has shown the data retrieving part through Tumblr API. This chapter will cover the processing part of those data, illustrating algorithms implemented and giving a brief introduction to the concepts behind them [3].

3.1 Bag of Visual Words Representation

As mentioned in the introduction, a clustering of the photos has to be performed. However, they need first a fixed-length representation, so clustering algorithms can be applied.

For this project it has been used the **Bag of Visual Words** representation (BoVW) [4]. BoVW is a technique that, given a database of photos, can create a fixed-length vocabulary of visual words. From this vocabulary it is possible to get BoVW representation for new photos.

To achieve this, a script named **bovw.py** is created. This scrip contains a class called **BoVW** with the methods:

- **__init__**();
- **_extract_and_describe**();
- **_load_and_describe**();
- **get_liked_representation**();
- **get_candidate_representation**().

__init__() is the constructor of the class and, once called, it creates and initializes a BoVW object. The main parameter to pass is a database of training photos, that in this case is the one containing user's liked photos. The initialization part takes care of:

1. extracting, describing (using SIFT descriptor) and concatenating all the patches from the input database through the support method **_extract_and_describe()**;
2. applying kmeans clustering algorithm to the previously created dataset of patch descriptors, to get centroids as visual words and build the vocabulary;
3. saving the fitted kmeans model for later use.

get_liked_representation() is a method that, from a dataset of liked photos local path, return the same dataset with added BoVW representation. In particular, for each photo:

1. applies the support method **_load_and_describe()** to extract patch descriptors and gets, for each of these, the nearest cluster through **kmeans.predict()**;
2. computes a histograms of patch descriptors occurrences in each cluster through **np.histogram()** from **NumPy** [5] and gets only the normalized counts list;
3. adds the previous list as separate columns to the corresponding photo row;
4. save the new dataset to disk.

get_candidate_representation() is a method that, like the previous one, returns a BoVW representation, but this time only for candidates photos.

With the above steps, for all photos, a BoVW representation is computed. Now each one has a number of added features equal to the number of clusters, which corresponds to the length of the visual words vocabulary.

3.2 Kmedoids clustering

Once obtained a fixed-length representation for each photo thanks to the BoVW model, Kmedoids clustering can finally be implemented.

It has been decided to use **Kmedoids** algorithm because of its simplicity and better performance on this particular dataset, but others clustering algorithms can also be used. Kmeans is a valid alternative but the major problem is that clusters centers are computed by averaging the points in the cluster. In the case of photos, this leads to clusters centers that are actually not real photo because of averaging, and can deteriorate the image matching process.

Kmedoids functioning is very simple, it takes as hyperparameter a number **k** of cluster to create and computes a partition of the training set of size k. Each element of that partition represents a cluster, as compact as possible. The center of each cluster is called **medoid**, and is a point belonging to that cluster.

Like the previous section, it has been created a dedicated script called **kmedoids.py**. The main function of this script is **kmedoids_clustering()** which takes a dataset as input. The goal is to train the Kmedoids model with BoVW representations of liked photos and then associate, for each of those, a cluster label, indicating which cluster they belong to. Labels are added as new columns to the input dataset.
Kmedoids is implemented by the function **KMedoids()** [6] from **sklearn_extra.cluster**, that takes as input the number of cluster to generate.

Since the algorithm requires the hyperparameter k to work, a tuning process has been implemented, using the **elbow method** [7].

The elbow method calculates, for a predetermined sequence of k values, the **Sum of the Squared Error** (SSE). This value is given by sum of the squared Euclidean distances of each point to its closest medoid and decreases with increasing k. The best value of k to choose, according to this method, is the one which the SSE curve starts to bend at. This point is called **elbow point** and it is showed in figure 3.1.

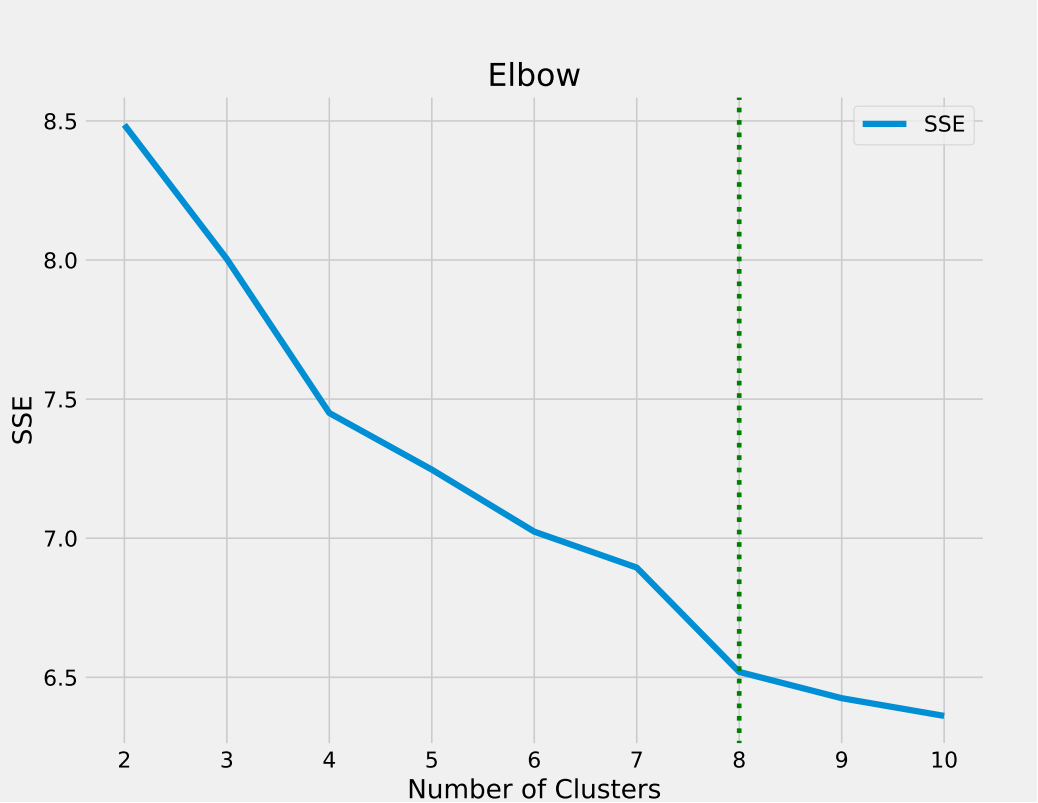

Figure 3.1: SSE curve and elbow point

The dashed vertical line **x=8** tells the elbow point, found with a python library called **KneeLocator()** [8], that takes a range of tuning values for k and a list of the corresponding SSE values. SSE values can be retrieved from the KMedoids object attribute **inertia_**.

The conclusion of this tuning part is that **k=8** is the best number of clusters for this specific dataset, so Kmedoids is fitted according to it.
Note that KMedoids tuning can produce a different better value for k, due to the fact that KMedoids choose randomly the firsts k medoids to start with, so this can lead to several cluster configurations that influence the tuning output.

In figure 3.2, are shown two sampled photos that fitted KMedoids put on the same cluster, along with their histogram of visual words occurrences.

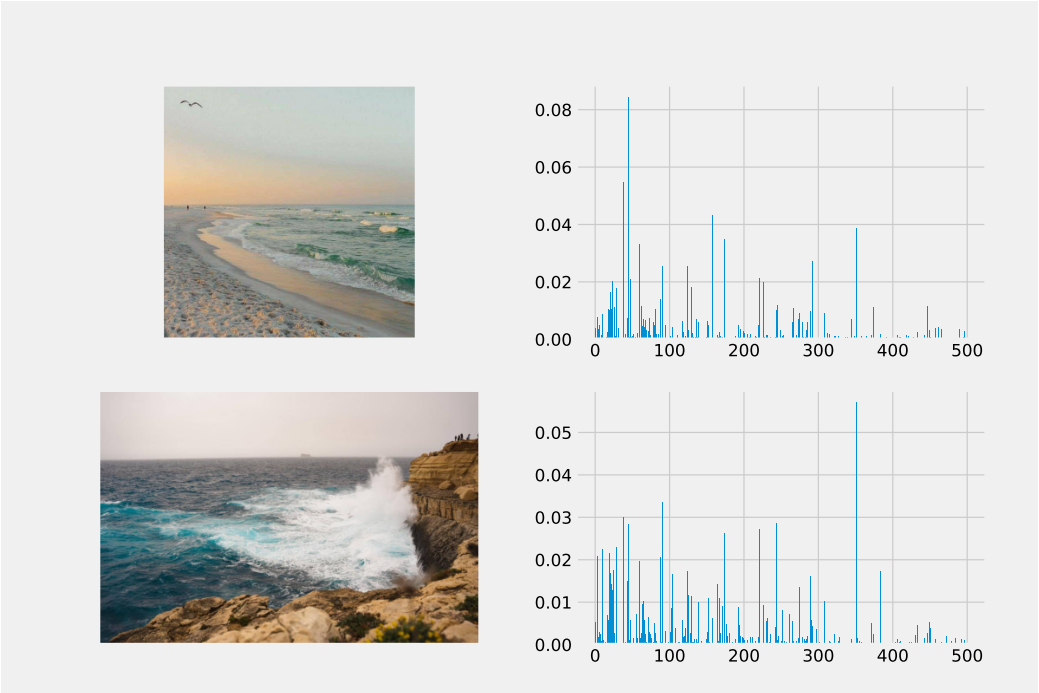

Figure 3.2: Sampled photos from cluster 5

Histograms are clearly similar and so are the photos, that show the sea. Another couple of photos clustered together is represented in figure 3.3.

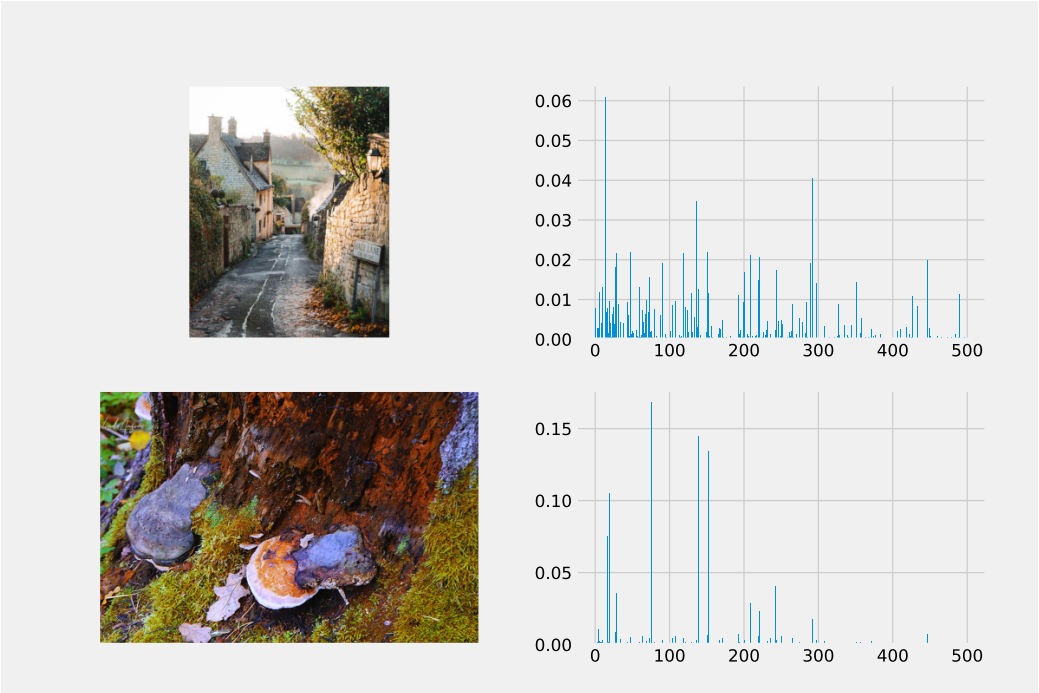

Figure 3.3: Sampled photos from cluster 4

This time both histograms and photos are not that similar, means that the clustering algorithm performed worse then before. Another case is shown in figure 3.4.

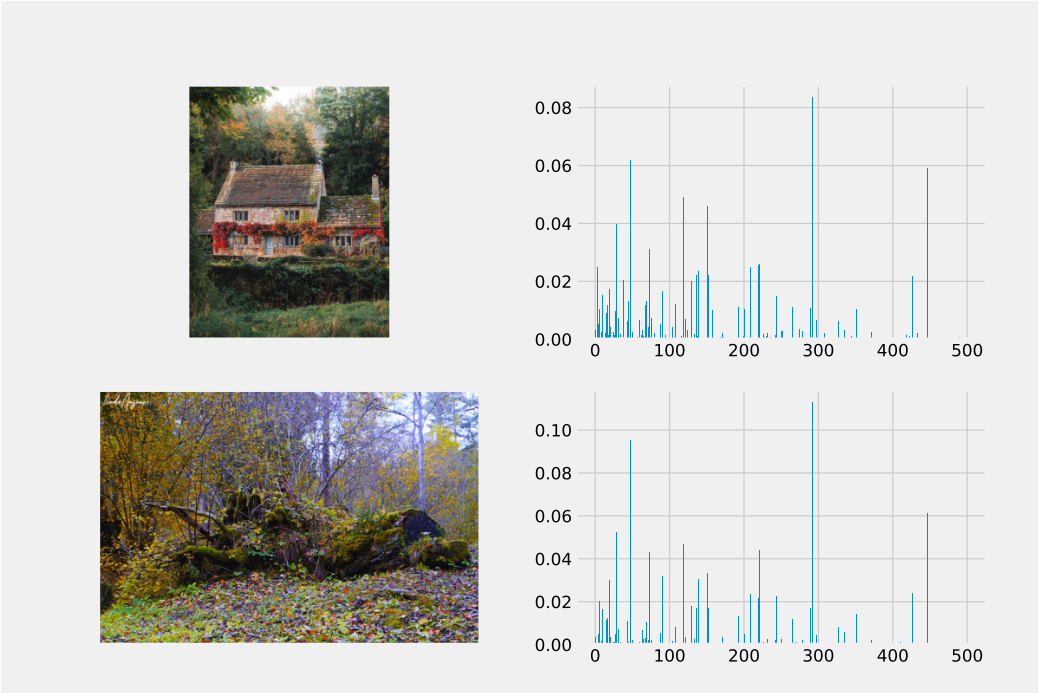

Figure 3.4: Sampled photos from cluster 3

As can be seen, the two photos are about a house and a woods, so represent dissimilar contents, but histograms are very close. This is the main problem with SIFT descriptors, photos with different contents can have very comparable histograms.

This concludes the clustering part, that introduced a new attribute, the cluster label, in the original dataframe. Note that this achievement is the fundamental part of this task solution, because somewhat similar photos have been grouped together without the help of tags.

3.3 Content-Based Image Retrieval application

Last section of this chapter is about the application of **Content-Based Image Retrieval** (CBIR) [9]. CBIR is a technique used to match a **query image** with a database of images. Intuitively, the best match is the most similar image in the database to the query.

Similarity is a concept already discussed, in fact two images are said to be similar if both have a similar BoVW representation. CBIR can hence be applied.

A script named **cbir.py** defines the following main functions:

- **prepare_candidates()**;
- **prepare_queries()**;
- **get_best_worst_recommendation()**.

prepare_candidates() retrieves candidate photos and represents them with the BoVW model. In particular, it takes a dataframe of clustered photos as input and, for each cluster, aggregates the blogs involved. After this step a blogs name list is now associated to each cluster. From this list can be called the method retrieve_candidate_photos() from the class tumblr_data to get photos from the last n blogs posts.
A BoVW representation is then applied and the results are saved and returned.

prepare_queries() retrieves, given a dataframe of clustered photos as input, the medoid for each cluster. It will be used next as query.

get_best_worst_recommendation() creates a **KDTree** [10] object from the library **sklearn.neighbors** that memorizes and indexes all candidates representations obtained from prepare_candidates(). Next the method **query()** is called from this object that, given a query representation, returns the distances and indexes of the two candidates that match the query better and worse. Queries are obtained from prepare_queries(). The return value is a dataframe that contains the local paths of all the triples: (query, best match, worst match) plus the query cluster label.

Next section will show the results of CBIR application, with plots of recommendations for each query.

Chapter 4

Results

Now that the recommendation system is completed, recommendations can be visualized by the figures below.

As one can see not all best recommendations are actually interesting, but most of them are if categories spotted by the clustering algorithm Kmedoids are not expected to be too specific. For example the best match and the query for cluster number 3 (figure 4.4) are both about bushes and trees, so a similar recommendation can be legit. All other recommendations are pretty accurate with this mind, except for the one in cluster 6 (figure 4.7), which should have had inverted suggestions, as query and worst match depict both a building.

The ones that performed better are from clusters 0, 1, 2, 3, 5 and 7. It Can be concluded that **6 out of 8** (75% of the cases) recommendations match really well the user's interests.

Of course performance may change for others users. It depends on how different the BoVW representation of liked photos is when these last are not similar to each other. To improve accuracy, BoVW representations of contents dissimilar photos must have a clear difference, so Kmedoids can do a better grouping and CBIR a better matching.

Figure 4.1: Query, best and worst match for cluster 0

Figure 4.2: Query, best and worst match for cluster 1

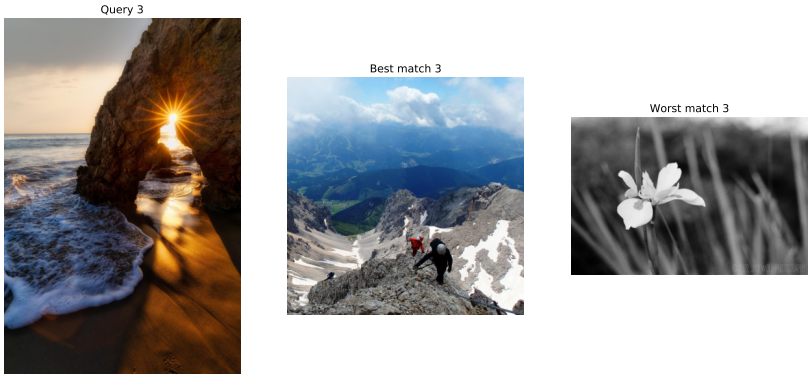

Figure 4.3: Query, best and worst match for cluster 2

Figure 4.4: Query, best and worst match for cluster 3

Figure 4.5: Query, best and worst match for cluster 4

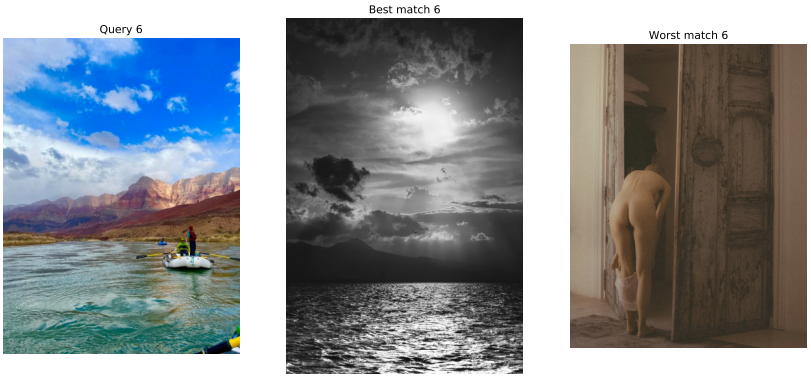

Figure 4.6: Query, best and worst match for cluster 5

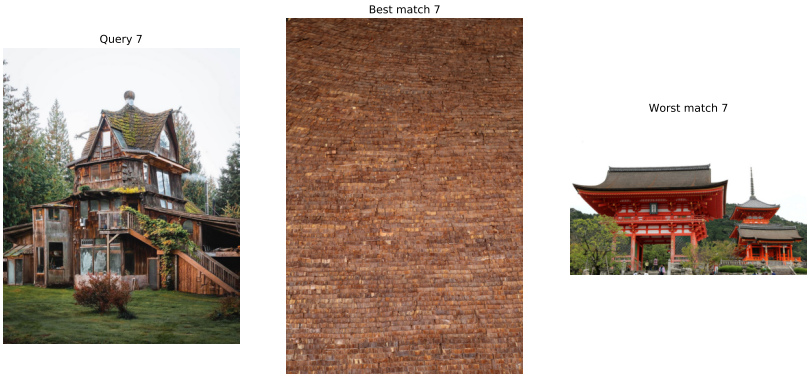

Figure 4.7: Query, best and worst match for cluster 6

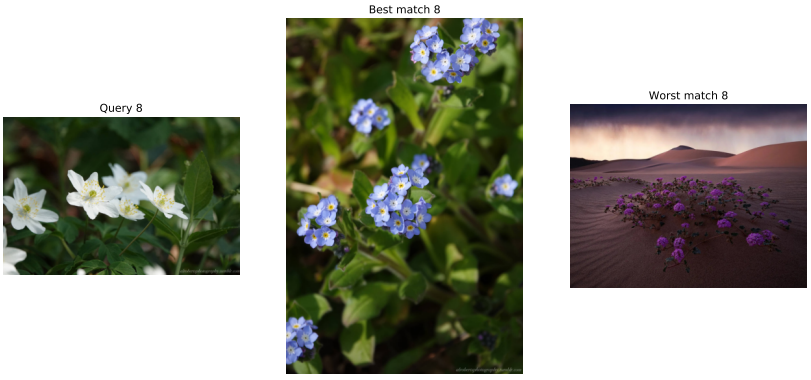

Figure 4.8: Query, best and worst match for cluster 7

Conclusions

This dissertation explained an alternative way of getting recommendations for Tumblr posts of type photo through images matching. The implemented solution worked quite good using simple tool like SIFT for patches description, BoVW model for photos representation, Kmedoids for clustering and CBIR for matching between two given representations.

All the above tool are quite heavy on large dataset of photos, especially the patch extraction and description. It has been used a library called **Pickle** [11] to save fitted models and datasets once computed. Next time scripts are executed all the models and datasets are not recomputed, but simply read from disk.

To get more accurate predictions of what a user may likes, more elaborated algorithms have to be used, but this was only a small project to show the potential of real use case of CBIR.

In conclusion, tags can't be avoided for now if one want a great balance between quality of the results and and relatively light computation, because, as said before, working with photos can be quite heavy, even for a modern computer.

Bibliography

[1] Tumblr api documentation. https://www.tumblr.com/docs/en/api/v2.

[2] Pandas official documentation. https://pandas.pydata.org/docs/l.

[3] Slides from social media management module.

[4] Bag of visual words model. http://www.robots.ox.ac.uk/~az/icvss08_az_bow.pdf.

[5] Numpy official documentation. https://numpy.org/devdocs/index.html.

[6] Kmedoids official documentation. https://scikit-learn-extra.readthedocs.io/en/stable/generated/sklearn_extra.cluster.KMedoids.html#sklearn_extra.cluster.KMedoids.

[7] Tuning kmedoids hyperparameter. https://realpython.com/k-means-clustering-python/#choosing-the-appropriate-number-of-clusters.

[8] Knee locator. https://github.com/arvkevi/kneed.

[9] Cbir. https://it.wikipedia.org/wiki/CBIR.

[10] Kdtree. https://en.wikipedia.org/wiki/K-d_tree.

[11] Pickle. https://wiki.python.org/moin/UsingPickle.