Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Smart Photo Viewer

Semester Thesis

Adrian Gämperli

gaadrian@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Michael König
Prof. Dr. Roger Wattenhofer

July 24, 2013

Acknowledgements

I would like to thank Prof. Dr. Roger Wattenhofer for the opportunity to write this thesis at the Distributed Computing Group at ETH Zürich. Furthermore I much appreciate the discussions and valuable inputs of Michael König. I also want to thank all the individuals, who tested the application and gave feedback.

Abstract

We have developed a native Android App which displays photos based on implicit user feedback. After selecting Flickr as our photo source we then evaluated the available photo attributes. Then we decided that we should use tags for the content description. A requirement in this thesis was to use only implicit feedback. The metric used is the viewing time of the photo. An algorithm was developed which recommends photos based on the previously rated photos. However, we found that our assumption that tags describe the content of the photo is not always true.

Contents

Chapter 1

Introduction

Recommender systems are very common these days. Many different cases of usage exist. They are used for product, music stream or movie recommendations and the like.

Due to digital photography everyone can take pictures without expensive costs. Some of these photos are shared on public websites others never leave their local storage. An existing problem is how it is possible to only show photos of these huge mass of photos which fit user-tailored interests.

Our task has been to develop an Android app which recommends user-tailored photos based on implicit feedback.

In chapter 2 we describe our recommender algorithm. The subsequent chapter 3 discusses its implementation. We then present the results of the thesis in chapter 4 and the conclusions in chapter 5. Finally in chapter 6 we give some ideas of how this thesis could be continued.

Chapter 2

Recommender Algorithm

In this chapter we first explain the photo rating algorithm. We then discuss the photo attributes used, and finally we introduce the photo selection process.

A sequence of displayed photos is called *stream*.The *number* of a photo is its position in the stream.

2.1 Rating algorithm

We were looking for implicit rating. We have chosen the viewing time as the metric. The algorithm maps the feedback of the user to a scale from 1 to 5, where 1 is the worst and 5 the best.

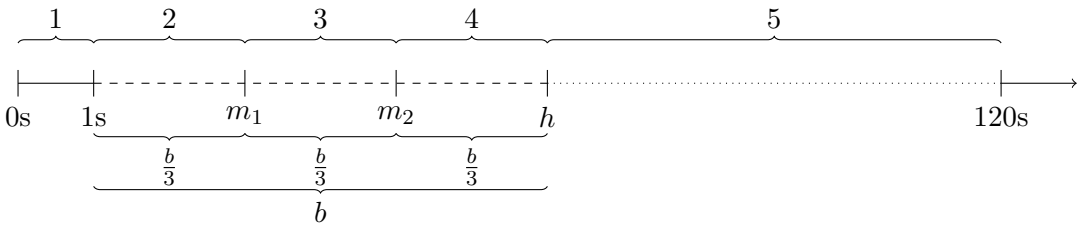

Figure 2.1: Rating algorithm

In Figure 2.1 we show how the rating algorithm works. We define a viewing time shorter than 1 second as rating 1. An important assumption is that a user viewing time for photos within the same rating will not change over time. To calibrate our rating algorithm we only use the random photos which are not based on user ratings (see initial seeds 2.3.1). We only use such photos as the goal of this thesis is that photos proposed by our application will become better. Value h is chosen as the average of the first half of the highest viewing times of these photos (see Algorithm 1). The values m_1 and m_2 are evenly distributed between 1 second and h.

Algorithm 1 Calculate h

```
1: function calculateH
2:     ratedPhotos ← getRatedPhotos(source = initialSeed)
3:     ratedPhotos.sortByViewingTimeDESC()
4:     return avg(firstHalf(ratedPhotos))
5: end function
```

Photos with a viewing time longer than 120 seconds and those without a viewing time are excluded from the algorithms described below.

2.2 Photo attributes

We are using photos from Flickr[1] as they offer a very comprehensive API. Furthermore, they have millions of users and billions of photos. Flickr provides different photo attributes. The most important ones are: title, tags, description, dates, uploader, location and user feedback (comments, favourites). In our opinion only the first 3 possibly describe the content of the picture. As both title and description are usually running text it is difficult to analyse their content. We therefore have chosen the attribute *tags* to be used in this thesis to model the user's interest. It is not possible to use user feedback as it cannot be searched and there is too much data to crawl.

2.3 Photo selection algorithm

In this section we discuss the algorithm which recommends the photos. In Figure 2.2 is an overview of the algorithms used. As the Reserve Pool algorithm (see section 2.3.2) needs rated photos, the first 10 photos of a stream are all recommended by the Seed Pool algorithm (see section 2.3.1). Afterwards only 10% of the photos are recommended by the Seed Pool algorithm and the other 90% are suggested by the Reserve Pool algorithm. In case an algorithm has no photo left to recommend, it delegates the recommendation to another algorithm.

2.3.1 Seed Pool

One important problem we faced is how to start the stream. The start is crucial as decisions for following photos are based on these starter photos. We tried using the only thing we knew about the app user (location) but were not successful. The problem was that we only knew one thing about the user, so there has not been a diversity of photos. We also tried to use the photos from Flickr Explore

[1] http://www.flickr.com/

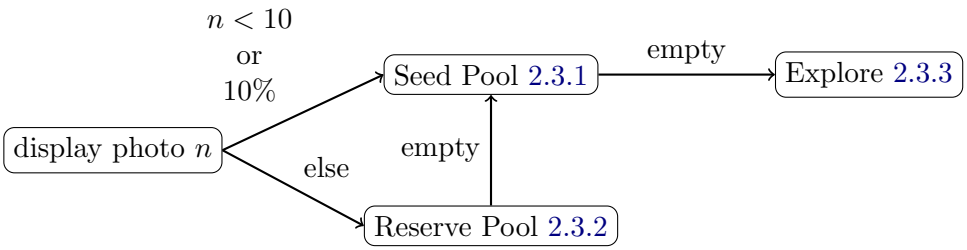

Figure 2.2: Algorithm overview; display photo with number n

(see 2.3.3). But we then still faced the problem of an unpredictable variety of different photos.

Our solution is that we create a pool of tags for every stream. We then pick a tag at random and search clusters of that tag using the Flickr API. Clusters are related tags. We then search photos in this cluster. Each tag has assigned a credit. Picking a tag from the pool reduces the credit of the tag by 1. As this worked for initialisation we extended it and we also add excellent tags to this pool. Each tag can either be initial or excellent. This may introduce some error in case an initial tag does not reach credit 0.

As already mentioned there are two different types of tags in our pool: Initial and excellent.

Initial

The initial tags (Table 2.1) are statically set in the code. The purpose of the tags is to present the user with a wide diversity of photos, as it is the basis of the Reserve Pool algorithm. The tags on that list are compiled from two sources ([1] and [2]). Initial tags have a credit of 2. This increases the chance, that if the first photo of a certain category is bad, the category is not completely lost.

abstract	car	health	people	travel
animal	color	illustrations	school	winter
architecture	family	industry	science	
art	fashion	love	sport	
autumn	food	nature	spring	
business	fun	party	summer	

Table 2.1: initial tags (compiled from [1] and [2])

Excellent

The goal of adding tags of excellent photos to the seed pool is to get a broader variety of excellent photos.

Excellent tags are added whenever a 5 rating of photo is reported by the application. But it only adds a maximum of 5 of the photo's tags to the seed pool, which is a protection against too many tags in the pool. They are chosen randomly. To be added to the pool there have to be more than 10000 photos using that tag. This helps preventing seldomly used tags from being added to the pool, which are possibly user crafted names. Each of these excellent tags get a credit of 1.

2.3.2 Reserve Pool

This algorithm recommends a photo based on previously rated photos. It can be split into 3 parts (see Figure 2.3). The first part of the algorithm suggests search parameters based on the rated photos of the current stream. Then in the second part it tries to find the best three photos based on the search parameters. It adds them to a pool, which is preserved per stream. Finally an algorithm selects a photo from this pool.

We have added this pool as it otherwise might take too long to generate the recommendation before the user switches to the next photo. Furthermore the app sends its feedback data asynchronously. This algorithm is run after new data is available to the algorithm (i.e. new feedback has been sent to the backend). So every run of the algorithm can be associated with a certain most recently rated photo. We save the number of that photo as it represents the freshness of the run.

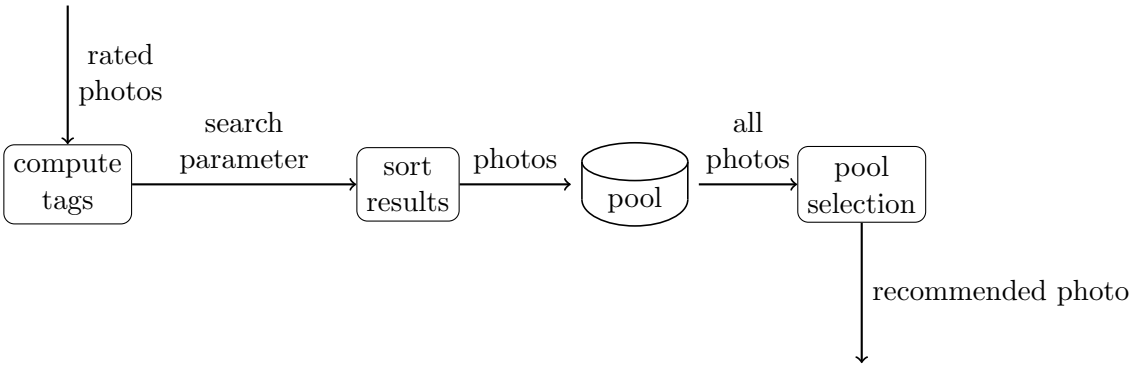

Figure 2.3: Overview Reserve Pool algorithm

Compute tags

Algorithm 2 returns a search parameter which represents the users interest.

Algorithm 2 Compute tags

```
1: function computeTags(ratedPhotos)
2:     groups ← generateGroups(ratedPhotos)
3:     groups.sortByGroupRatingDESC()                ▷ see Algorithm 3
4:     group ← groups.pickRandomWithExpDistribution(λ = log(2)/3)
5:     if std(group.memberRatings()) ≥ 1 then
6:         group.removeHalfOfTags()
7:     end if
8:     return group.tags
9: end function
```

First it tries to form groups. This works by iteratively comparing photos and saving tags which they have in common. The other tags are discarded. This group building does not generate the same groups on every run (because of built-in randomness). This is to ensure that the algorithm does not narrow down the variety too much. Photos which include all tags of such a group are called 'member of the group'. These groups are then sorted by Algorithm 3. Finally a random group is picked with an exponential probability distribution. This assures that well-ranked groups are mostly chosen but from time to time also new groups are tried to expand variety.

The main part of Algorithm 3 is on line 3. As the average is the most important property of a group it is added with factor 1. A part of the standard deviation is added to try to push groups with high standard deviation. Assuming a user has a static opinion on each tag a high standard deviation of a group means that some tags have high and some low ratings. This is why we want try to split the group to figure out which tags have low and high ratings. As we have seen in Algorithm 2 on line 5 to 7 half of all tags are removed from groups with a standard deviation greater or equal than 1. In case it is smaller than 1 an additional photo may either increase (so it will be halved) or decrease (so there might have been an outlier) the standard deviation. The summand of the number of members is to give bigger groups a small bounty as they are more stable and established.

On line 4 to 6 we try to block tags which do not have an impact on the photo. Examples may be the tags 'nikon' or 'canon'. While trying to isolate these tags we noticed that it is not always successful. This is why we block these two tags statically too.

In the last if-block we try to push groups with only one member to get more information about that tag and they do not have a standard deviation yet.

Algorithm 3 Group rating

```
1: function groupRating(group)
2:     memberRatings ← group.memberRatings()
3:     rating ← avg(memberRatings) + 0.2 ∗ std(memberRatings) + 0.01 ∗
   group.numberOfMembers()
4:     if group.numberOfTags() = 1 and std(memberRatings) > 1.5 and
   group.numberOfMembers() > 10 and avg(memberRatings) < 3 then
5:         rating ← rating − std(memberRatings)
6:     end if
7:     if group.numberOfMembers() = 1 then
8:         rating ← rating + 0.5
9:     end if
10:    return rating
11: end function
```

Sort results

Algorithm 4 is used to select photos which not only have the requested tags but are also liked by Flickr users. Using this it is possible to discard photos which might contain the right content but, for example, lack sharpness. The best 3 photos are added to the pool. The position of each photo on this list is saved as its rank. In case no photo is found the tags are sampled and a new search is tried.

Algorithm 4 Sort results

```
1: function sortResults(tags)
2:     allPhotos ← searchPhotos(tags)
3:     allPhotos.sortByViews()
4:     while photos ← allPhotos.load10Photos() do
5:         possiblePhotos ← {}
6:         for all photo ← photos do
7:             if photo.views() > 100 and photo.favourites() > 10 then
8:                 possiblePhotos ← possiblePhotos ∪ photo
9:             end if
10:        end for
11:        if number(possiblePhotos) > 0 then
12:            possiblePhotos.sortByFavouritesPerView()      ▷ #fav/#views
13:            return first3(possiblePhotos)
14:        end if
15:    end while
16:    tags.remove40Percent()
17:    return sortResults(tags)
18: end function
```

As the number of favourites is an extra API call, only 10 photos are loaded at once. To increase the chances to get a good photo the photos are first sorted by the number of views.

Pool selection algorithm

Algorithm 5 can be split in two parts. The first part assumes that the algorithm has been fast enough and enough data has been provided. Therefore it returns the first ranked result of the last insertion of pool photos. The second part is thought to be a fallback. It rates all photos in the pool belonging to the current stream. This rating algorithm tries to reflect that newer runs (based on the parent number) have had more data available, we want to show good photos (thus influence of rating) and that we try not to have two sucessive photos of the same run.

Algorithm 5 Pool selection

```
1: function selectFromPool(poolPhotos)
2:     poolPhotos.sortByRankASCParentNumberDESC()
3:     if poolPhotos.first().rank = 1 then
4:         return photos.first()
5:     end if
6:     poolPhotos.sortByPoolPhotoRating()          ▷ see Algorithm 6
7:     for all photo ← poolPhotos do
8:         if photoNotInStreams(photo) then
9:             return photo
10:        end if
11:    end for
12:    return getSeedPhoto()
13: end function
```

Algorithm 6 Pool photo rating

```
1: function poolPhotoRating(poolPhoto)
2:     return (currentNumber − parentNumber) ∗ 0.49 + rank ∗ 0.5 + (5 −
   pool.rating())
3: end function
```

2.3.3 Explore

This fallback algorithm fetches the photos which are displayed on the *Flickr Explore*[2] website. Basically it returns the first photo from that web page which is not already in the streams of the user.

[2] http://www.flickr.com/explore

Algorithm 7 Get photo from Flickr Explore

```
1: function getExplorePhoto
2:     repeat
3:         photo ← fetchNextPhotoFromFlickrExplore()
4:     until photoNotInStreams(photo)
5:     return photo
6: end function
```

Chapter 3

Implementation

3.1 Architecture

We have used a server-client model. We have chosen this model over an autonomous app as it has several advantages. Firstly our server has much better performance than a smartphone: bandwidth, power (battery), processing power are available at a multiple. This allows us to make more requests to the Flickr API to deliver better results. Furthermore this model is much easier to debug and update. As all data is stored centrally it is also possible to analyse recommended photos. This leaves more space for new ideas such as comparing to other users or analysis of the photo content. Such a system also allows the app to be migrated to another platform more easily. On the other hand it is a single point of failure and someone has to take care of a server system. However in our opinion the advantages outweigh its disadvantages.

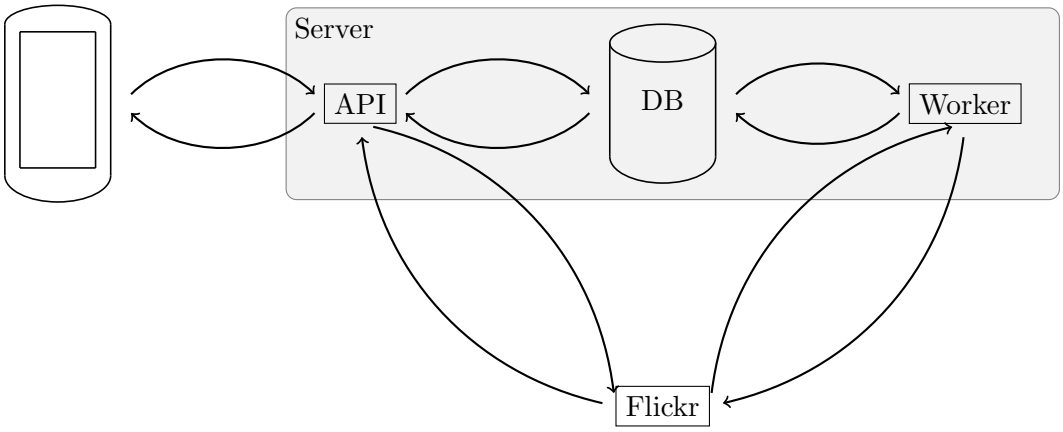

Figure 3.1: Architecture of the project

3.2 Frontend

3.2.1 Android App

To develop the Android app, the official Android SDK of Google has been used. Therefore the app is written in Java. The app basically displays a photo and asks the backend which photo it has to display next. Furthermore it sends the viewing time of each photo to the backend server for further processing. A new stream is created on every app start-up. To switch the photo the user has to swipe the picture to a side. It is possible to zoom in. This feature has been implemented using an open source widget [3]. The icon of the app is copied from the Gnome project [4].

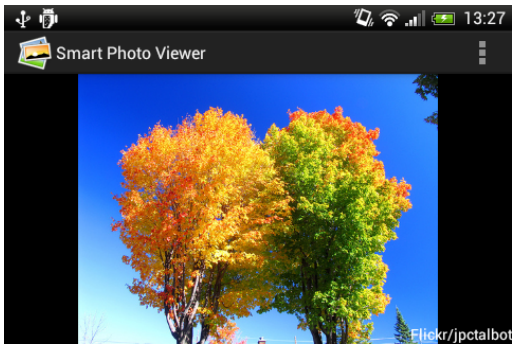

Figure 3.2: Screenshot of the Android App

3.2.2 Website

Additionally a website has been developed, which can also be used to view the photo stream. Its main purpose is to simplify debugging and also to show debugging information of the photo selection process. It also captures the viewing time and sends it to the backend.

3.3 Backend

The backend has been developed in Python. Data is stored in a mySQL[1] database. To simplify the access SQLalchemy[2] has been used.

[1] http://www.mysql.com/
[2] http://www.sqlalchemy.org/

API The API is implemented using the Flask Framework[3]. Its primary task is to respond to the clients' requests and to deliver photos to them. Furthermore it delegates long-running tasks to the workers. Thus it can send a response fast.

Worker The workers have the job to execute long-running tasks. These long-running tasks include calculating the next photo using the algorithm and adding the found photos to the Reserve pool. Furthermore, they update the Seed pool. The workers are implemented using Celery[4].

[3] http://flask.pocoo.org/
[4] http://www.celeryproject.org/

Chapter 4

Results

In this chapter, the choice of attributes and the algorithm are discussed.

4.1 Choice of attributes

The important assumption that tags always describe the content of a photo has only proven to be partially true.

Unprecise tags There are photos whose tags are unprecise. As an example we saw photos which showed a storm through a car window or a woman at a car race. Both photos had the tag 'car'. However, not everyone would probably associate these photos with the topic car.

Camera brand It seems to be common that the camera brand (and name) is added to photo tags. Most prominent tags of this type are 'nikon' and 'canon'. As we were unable to find a reliable and not conflicting solution to filter these tags, these are filtered manually.

An example of this problem is when searching for clusters for the tag 'sport'. One cluster name returned is 'canon, man, bw'. So even Flickr does not have a full solution for this problem.

Metadata Some photos include tags that describe metadata of the photo such as whether it is in colour or black-white. While this might influence the rating of the photo it does not fulfil our assumption that tags describe the content of the photo. The same example as in the previous paragraph can be used. 'bw' is an abbreviation for a black-white image.

Seldom used tags There are tags which do not describe the content. An example is 'joinplayingwithbrushesgroupifyouusethese'. We reduce the influence

of such tags in the Seed algorithm by blocking all tags with less than or equal to 10000 photos on Flickr.

4.2 Recommender Algorithm

Testing When comparing the average rating of photos based on initial tags and those photos recommended by the Reserve Pool algorithm it is not obvious whether our algorithm is better than random photos. However, most individuals who tested the application had the opinion that it is. An analysis is difficult because of the limited number of test results.

The app has been tested by friends and various people of the Distributed Computing group. The feedback could be used to improve the application.

Performance Our implementation had performance issues, which have been solved by using background workers.

Chapter 5

Conclusions

We can conclude that it is possible to use implicit feedback to build a photo recommender system. Furthermore the viewing time of a photo has been proven to be useful.

We have seen that it is very important to have precise tags on photos to be able to make good recommendations.

We learned that when only using positive information not all tags can be precisely understood. Therefore no data should be discarded to get a comprehensive result.

Chapter 6

Outlook

Collaborative filtering Currently the application has a very small user base. As the user base grows it can also be tried to build a recommender system based on collaborative filtering such as on friendship or similarities in previously viewed photos.

Multiple sources At the moment only photos from Flickr are included. It could be researched whether multiple photo sharing websites can be combined to display even better photos.

Using photo content It could be researched into whether a photo can be analysed to characterise more precisely the user's interests. As an example it could be tried whether dominant colors have an impact on the result or whether it is possible to use object and face recognition to automatically add tags or make them more precise. In case it is possible it could be used for local photo collections.

Negative searches Currently we only try to track what a user likes. However we did not research into whether we could use gathered information to resemble tags which the user dislikes. Using this it could be possible to exclude photos which are not to recommend because the user dislikes a part of the photo.

Matching to groups A different approach would be to try to understand all tags and match them to categories. This is basically the reverse of the clustering function used in the Seed algorithm. Using this approach it could be possible to better understand the precise meaning of wide tags.

Non-content tags As described some tags such as 'nikon' or 'canon' do not describe the content. It could be tried to filter these tags automatically and not manually anymore.

Dynamic photo rating calibration In this thesis the photo rating calibration is based on the initial seeds and the scale of the ratings is the same for all photos of a stream. However assuming that our photo recommendations are becoming increasingly better it could be tried to adapt the scale so that finer predictions can be made.

Bibliography

[1] PhotoSpin, Inc: Browse the most popular categories. https://www.photospin.com/browse_photos.asp Accessed: 18.07.2013.

[2] Photobucket: Top categories. http://photobucket.com/browse Accessed: 18.07.2013.

[3] Kenzo, I.: scale-imageview-android. https://github.com/matabii/scale-imageview-android Accessed: 18.07.2013.

[4] Gnome Project: Gnome web icons. http://www.iconfinder.com/icondetails/55605/64/gallery_images_photo_photos_icon Accessed: 18.07.2013.