20-05-2012

Folksonomizer

Lightweight folksonomy implementation algorithm

Adam Sobaniec (sobanieca@gmail.com)

Table of contents

1. Introduction

Folksonomy is all about, so called social tagging, where users assign keywords to the content that they own. Having tags assigned to the content allows to perform fully automatic classification. Elements are assumed to be similar if they contain similar tags. Folksonomy reduces the amount of work that moderators or administrators of the web site need to do in order to perform a classification.

This whole project is supposed to be a demonstration on how such system can be implemented. Right now, when searching the web for the information about implementation of folksonomy one can find a solutions that perform complex SQL queries to the database to find similar content. This project presents the optimization of this process so it can be scaled and used with a very large data sets. It is based on my previous research regarding the folksonomy, where I have used the ART-1 neural network in order to perform the classification. The results were more than satisfactory – the solution was fast, and the quality of a classification was very accurate. However, it required too much memory and couldn't be implemented in the production system.

In this following application I've used a similar approach, but, instead of creating an ART-1 neural network, I've created an object that I called Folksonomizer that is similar in the construction to the ART-1 neural network but it doesn't contain any decimal weights of the connections between neurons. By this simple modification, there is no reduce in the classification quality, but there is a huge optimization in the memory usage.

2. Input data

In order to perform the classification there were taken 10000 photos, downloaded from Flickr.com. The photos had on average 6 keywords assigned. Unfortunately due to the lack of tags validation, there is a small mess in the tags database. In the downloaded sample, there was no limit on the length of the keyword, or on the maximum number of keywords assigned. For this reason, some photos have even about 40 tags assigned, while the others contain keywords with white space characters.

3. Database structure

All photos are stored in the database with the following tables (diagram has been simplified to reflect only the most crucial parts):

Photos table stores all information that are related to the particular photo. The image itself is being stored inside the table as a varbinary(MAX) field. Cluster column defines the cluster to which photo has been assigned in the classification process. ***Tags*** table stores information about all tags provided by the users in the application. Column *Counter* holds an information about the number of photos to which given tag has been assigned. This information is being used later by the folksonomizer algorithm. Table ***PhotosTags*** is a standard many-to-many relation mapping table.

Such database structure is supposed to be the best choice for the keywords assigning system. You can search the internet for more information about database schemas that are used to implement folksonomy. The one used in the sample application is supposed to be fast enough and to allow the best flexibility.

4. Folksonomizer algorithm

4.1. Obtaining the training set.

In order to perform the classification there must be created a set of binary vectors that will represent the users content in an application. In this sample project each vector corresponds to a single photo with all tags assigned to it. The vectors are created basing on the information about all tags assigned to more than one photo and stored in the ***Tags*** table - so called dictionary of tags. Binary vector is being created basing on the following algorithm:

Given:
Dictionary of tags $D=\{d_1, d_2, \ldots, d_n\}$

Tags assigned to the given content $T=\{t_1, t_2, \ldots, t_m\}$

Define resulting vector $V=[v_1, v_2, \ldots, v_n]$

For each tag d_i **in** D

If d_i **is contained in** T

Set $v_i=1$

else

Set $v_i=0$

Let's consider the following example:

There is the following content of the ***Tags*** table available - ['Cat', 'Dog', 'Car', 'Ferrari', 'Red', 'Blue']. One has to build so called dictionary of tags. It's content will be all tags that are assigned to more than one photo. Let's assume that in this case all tags in the table meets this requirement.

Now let's take a photo of a Ferrari from the ***Photos*** table:

with the following tags assigned: ['Car', 'Ferrari', 'Red']

Basing on the given dictionary construction of the binary vector v is being obtained in the following way:

'Cat' is not assigned - set $v_1 = 0$

'Dog' is not assigned - set $v_2 = 0$

'Car' is assigned - set $v_3 = 1$

'Ferrari' is assigned - set $v_4 = 1$

'Red' is assigned - set $v_5 = 1$

'Blue' is not assigned - set $v_6 = 0$

Finally, obtained vector looks as follows: [0, 0, 1, 1, 1, 0].

For each photo in the database there is being created a corresponding vector. After the set of such vectors is available the Folksonomizer can be trained.

4.2. Architecture

Knowing how the training set looks like and how it is being built, we can proceed to learn the structure of Folksonomizer. The basic idea is to create a clusters that will describe the subset of vectors with common features. Each cluster contains following properties:

Cluster
Id
Prototype

Id is supposed to be a unique identifier of a particular cluster, while the prototype is a binary vector that defines it. One may say that it is an “average” vector that represents all of the vectors that

belong to the given cluster. If two vectors (in our case photos) are similar, then they will fall into the same cluster.

4.3. Classification

The main idea of a Folksonomizer algorithm is very simple. For each vector from the input set Folksonomizer tries to find the cluster for which the prototype and given input vector is as similar as possible. If such cluster has been found then the given input vector is being assigned to it and it's prototype is being updated. If no such cluster has been found, then a new cluster is being created and given vector is being assigned to it.

The whole algorithm can be described with the following pseudo code:

Given:

Set of N **input vectors** $V=\{v_1, v_2, \ldots, v_n\}$

Set of M **clusters** $C=\{c_1, c_2, \ldots, c_M\}$ //At the first execution it will be empty

For each vector v_i **in** V

  For each cluster c_i **in** C

    Take p **- prototype of** c_i

    If $dist(v_i, p)<\rho$

      Assign v_i **to** c_i //Classification performed

      Update prototype $p=p \wedge v_i$

  If NO matching cluster has been found

    Create a new cluster c_j

    Set it's prototype $p=v_i$

    Add c_j **to the set** C

    Assign v_i **to** c_j //Classification performed

The function $dist(v_i, v_j)$ describes the distance between two binary vectors. For a Folksonomizer such function is defined as the number of common ones, since content is similar if it contains as much common tags as possible. The value ρ defines threshold that is supposed to describe how many common features vectors must have to be marked as similar. In folksonomy implementation it should be set to “2” or “3”, because we want to mark two photos as similar only if they contain at least two or three common keywords assigned.

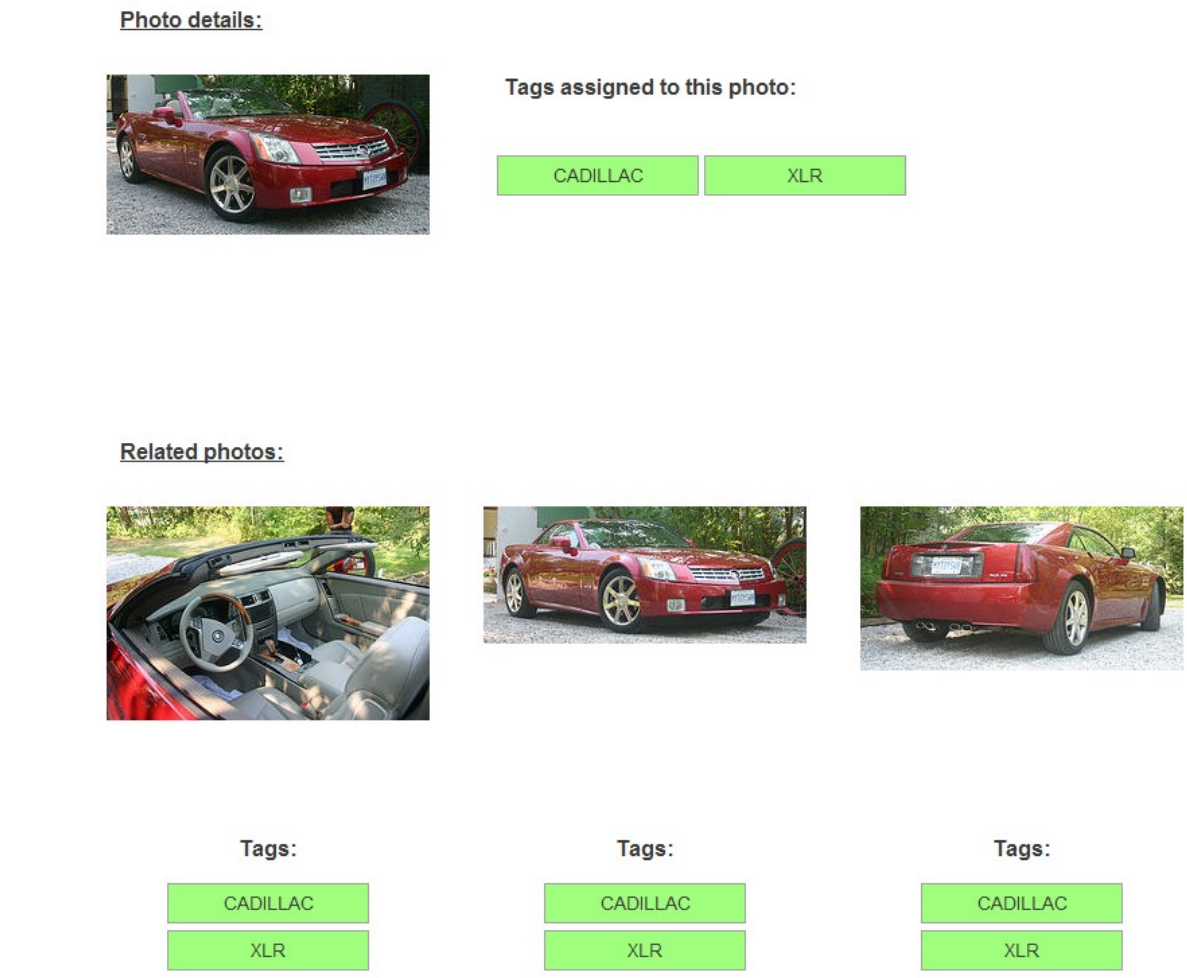
Update of the cluster's prototype is being done as an AND operation with a vector v_i . So for example when $v_i=[0,1,0,1,1]$ and $p=[1,0,1,1,1]$ then, after prototype update it will be set to $p'=v_i \wedge p=[0,0,0,1,1]$.

Id:	1
Prototype:	[1,1,0,0,1]
Assigned vectors:	[1,1,0,0,0] [1,0,0,0,1] [1,1,1,0,1] [1,1,0,1,1]

Id:	2
Prototype:	[0,0,1,1,0]
Assigned vectors:	[0,0,1,1,0] [0,0,1,1,1]

Id:	3
Prototype:	[0,0,0,1,1]
Assigned vectors:	[0,0,0,1,1]

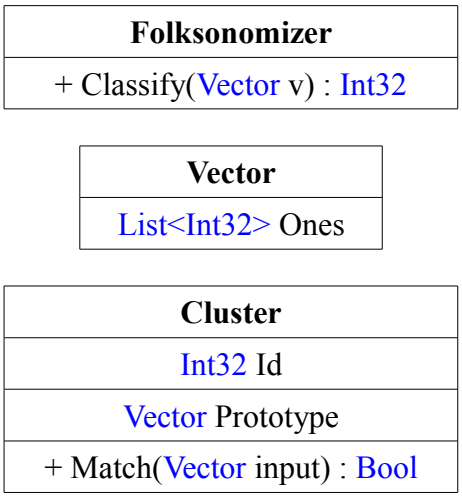
The sample result of such classification is presented in the picture below:



Knowing that each vector corresponds to a particular photo, one can store the id of a cluster to which it has been assigned and use this information later to retrieve a similar content. This way allows to perform much quicker database queries, which is great advantage when dealing with a heavily exploited web applications.

5. Implementation

In order to implement the presented algorithm the following main objects have been introduced:



Folksonomizer object contains only one public method – Classify(). This method takes an input vector and performs a classification algorithm presented earlier. As a result there is being returned the id of the cluster to which vector has been assigned. Basing on this information the ***Photos*** table is being updated, so it's *Cluster* column can be set to a proper value. Now searching for a similar content in the database can be achieved by a simple (and fast!) SQL query like:

```
select top @limit Id from Photos where Cluster = @clusterId
```

Since the input vectors are supposed to be very sparse, in order to optimize the memory usage, only the information about "ones" in the vector is being stored. Namely, if vector v has length 2500 and it contains "1" at the index 254 and 278, then the property "Ones" will be equal to $\{254, 278\}$. In most extreme cases, there might be used a plain one dimension array – it will be faster than the dynamic list. Such solution may be introduced if there is an upper limit for the number of tags that user can assign to the content that he is uploading. There may be also a need to use 64 bit integers as an index. However this consideration can be applied only for a very large web services that contain tons of data – there is no need to focus on it in this sample solution. Storing only indexes of "1"s in vector has also one very big advantage – there is no need to create a dictionary of tags and introduce the upper limit for the dictionary size.

Each cluster contains mentioned earlier unique id and a prototype that represents all of the vectors that were assigned to it. Apart from that, there is present a public method Match() that takes an input vector and verifies if it is close enough to the prototype. If it succeeds, then it updates the prototype, so it adapts to represent the new features of an input vector and features of a vectors that are already inside the given cluster.

In order to find out the details of an implementation please follow the source code of Folksonomizer attached to Folksonomizer.zip package.

Folksonomizer has been implemented in three different versions. First implementation is the straight forward implementation of the presented algorithm on the CPU. Second implementation has been done in CUDA C/C++ in order to perform a parallel computations on the GPU. Third implementation introduces a significant enhancement that uses a bit more memory but allows to perform all computations much faster.

5.1. CPU implementation

At the beginning Folksonomizer has been implemented using C# 4.0. It has been designed to reflect the presented earlier algorithm. No additional modifications have been done. As a result the classification of 10k photos downloaded from Flickr has been done with following statistics:

Number of photos classified:	10000
Classification duration:	5995 ms
Update on the database duration:	6010 ms
Memory occupied by Folksonomizer:	485 kB
Clusters generated:	3184
Clusters with more than 1 photo:	1719
Clusters with exactly one photo:	1465

All computations have been performed on a Core 2 Duo 2,53 GHz P8700 CPU.

Number of photos classified is the total number of photos that have been converted into vectors and provided as an input to the Folksonomizer.

Classification duration is the time that was necessary to perform the classification of all vectors.

Update on the database duration is the time that was necessary to perform SQL Server database update.

Memory occupied by Folksonomizer is in this implementation case very low. Only 0,5 MB.

There were totally 3184 clusters generated, where 1719 of them contained more than one photo.

5.2. GPU Implementation

Presented above algorithm can be performed faster when it is being done in parallel. For this reason I have decided to try to implement it on the Nvidia CUDA enabled GPU device. My laptop is equipped with GeForce GT240M graphics card and since it has a capabilities of running many threads simultaneously I have implemented a sample CUDA C/C++ application that was supposed to perform Folksonomizer algorithm in parallel.

Parallelized algorithm can be described by the following pseudo code:

Given:

Set of N **input vectors** $V=\{v_1, v_2, \dots, v_n\}$

Set of M **clusters** $C=\{c_1, c_2, \dots, c_M\}$ //At the first execution it will be empty

For each vector v_i **in** V

For each cluster c_i **in** C **in PARALLEL**

Take p **- prototype of** c_i

If $dist(v_i, p) < \rho$

Mark cluster c_i **as matching cluster**

If matching cluster c_i **has been found**

Assign v_i **to** c_i //Classification performed

Update prototype $p = p \wedge v_i$

else

Create a new cluster c_j

Set it's prototype $p = v_i$

Add c_j **to the set** C

Assign v_i **to** c_j //Classification performed

Since the implementation required quite low level programming I have decided to use one dimensional array to represent the photos vectors and clusters. The drawback of this solution is that I've been forced to introduce the maximum number of tags that can be assigned to a single photo. For this reason the sample classification differs a bit from the classification done by .NET version of Folksonomizer, because on Flickr some photos could have even 40 tags assigned. In this solution there has been significantly less memory required to perform the whole operation. To connect to the database I've used the ODBC drivers.

The biggest challenge when implementing such algorithm are the memory operations. Copying memory from the host to the GPU device is very slow and it requires quite a bit of work to organize the whole process in such way that it will minimize all bottleneck operations.

I have tried many combinations, but ended up with a solution which mixes the GPU and CPU operations. First of all before the beginning of the main loop where each photo is being classified there is being allocated the “worst-case” amount of memory on the GPU. Later on there is being performed the clusters matching. This operation is being done totally in parallel. Since my GPU unit was capable of creating big amount of threads, I have assigned a single thread to a single cluster. After the matching has been done, the CPU unit scans all clusters for a matching cluster. If it has been found – it's prototype is being updated and the photo is being assigned to it. Otherwise, there is being created a new cluster.

As a result the sample 10k photos downloaded from Flickr have been classified with the following statistics:

Number of photos classified:	10000
Classification duration:	2481 ms
Update on the database duration:	921 ms
Memory occupied by Folksonomizer:	~ 100 kB
Clusters generated:	3226
Clusters with more than 1 photo:	1736
Clusters with exactly one photo:	1490

The calculations have been performed two times faster than within the standard CPU implementation. The memory occupied by Folksonomizer is significantly less (5x) then the .NET corresponding solution. Number of clusters generated differs from the .NET solution. It is related to the fact that only first 7 tags are taken into account when performing classification. However the subjective quality of clustering was very satisfactory.

In order to find the details regarding this implementation please follow the source code available under project Folksonomizer.Worker.CUDA project in the Folksonomizer.zip package.

5.3. Clusters mapping enhancement

Since the Folksonomizer requires relatively small amount of memory in order to perform the computations, it is possible to construct it in such way that it will occupy more memory that will speed up the whole classification process. For this reason there has been introduced a cluster mapping that is supposed to ensure that only clusters that might be a match for particular photo are going to be checked. With this simple modification there are being omitted many checks that are unnecessary to be performed. The basic rule is very simple – instead of searching all clusters available, there is being collected the set of clusters that might be a match for a given photo. Only clusters that contain at least one common tag in a prototype with a given photo will be taken into account. The whole algorithm can be described with the following pseudo code:

Given:

Set of N **input vectors** $V=\{v_1, v_2, \ldots, v_n\}$

Set of M **clusters** $C=\{c_1, c_2, \ldots, c_M\}$ //At the first execution it will be empty

Set of T **clusters mapping** $M=\{t_1:\{c_1, c_2, \ldots\}, t_2:\{c_1, c_2, \ldots\}, \ldots\}$

For each vector v_i **in** V

Obtain matching clusters list C_M **from** M **for** v_i

For each cluster c_i **in** C_M

Take p **- prototype of** c_i

If $dist(v_i, p)<\rho$

Assign v_i **to** c_i //Classification performed

Update prototype $p = p \wedge v_i$

Update clusters mapping M

If NO matching cluster has been found

Create a new cluster c_j

Set it's prototype $p = v_i$

Update clusters mapping M

Add c_j **to the set** C

Assign v_i **to** c_j **//Classification performed**

Presented code above provides a general idea of an algorithm. It has been however implemented with few enhancements – please follow the source code to find out more details. After running this solution there has been noticed significant reduction of the calculation time, while the memory usage was still at very acceptable level. There have been obtained the following results for the 10k photos downloaded from Flickr:

Number of photos classified:	10000
Classification duration:	168 ms
Update on the database duration:	1587 ms
Memory occupied by Folksonomizer:	761 kB
Clusters generated:	3185
Clusters with more than 1 photo:	1724
Clusters with exactly one photo:	1461

The computation time for a given set have been almost 10 times faster than the solution without clusters mapping, while the memory usage have grown only two times. After obtaining such result there have been performed an additional test for a bigger amount of randomly generated data – 250k of photos.

Each photo in the sample data set have been generated by assigning random number of tags from range $\langle 2,7 \rangle$ to it. Each tag have been taken from the set of 2000 tags.

After all computations following results of the classification have been obtained:

Number of photos classified:	250000
Classification duration:	252081 ms
Update on the database duration:	40248 ms
Memory occupied by Folksonomizer:	14397 kB
Clusters generated:	145230
Clusters with more than 1 photo:	81892
Clusters with exactly one photo:	63338

One can see that results are not satisfactory. The data set has been 25 times bigger while the time needed to perform all operations was 1500 times bigger. However it is worth noticing that the memory occupancy is still on a very low level, so there may be introduced much more improvements to speed up the whole computation process. Also, the implementation on the GPU should perform the job much quicker, however it require more research. Right now I'm leaving it "as is", until I will find more time to implement it in the more effective way. You can find the whole source code in the Folksonomizer.zip package.

6. Usage

Folksonomizer can be used in three modes for each application:

- Background worker task
- Real time classification
- Mixed mode

As a „Background worker task" it is being run periodically as a separate task and performs a classification of the already existing content. All clusters are being generated from scratch. Such solution is very easy to implement however it doesn't allow to perform real time classification. User will have to wait until the Folksonomizer finish it's job to find similar content to the one he uploaded. This solution however is the most stable one and allows to reflect all changes made to the ***Tags*** table.

In a "Real time classification" mode Folksonomizer is supposed to be stored in the applications cache for the whole application life time. It is being kept in the memory from the beginning of the users content creation. In such case there is a possibility to classify a new content "on the fly" so the results are visible immediately. However there is a risk of loosing the Folksonomizer data if it is not saved correctly and an application crashes. Also there is no possibility to adapt to the changes of the ***Tags*** table content. In case when some key words has been deleted the classification process will be disturbed.

"Mixed mode" represents the mix of both presented solutions. Folksonomizer is being stored in the application cache and is available for an immediate use, but it is also regenerated periodically to reflect all changes in the ***Tags*** table. In case when Folksonomizer data has been lost such process allows to restore all clusters data. This solution is definitely the best one (although the hardest one to implement) and has been presented in the sample Folksonomizer application.

7. Summary

Presented Folksonomizer algorithm is much faster and more effective solution than the ART-1 neural network implementation, while the quality of classification remains on the same or even better level. However, there is still problem when performing classification on a very large data sets. Since the algorithm has high time complexity and a low memory complexity, it can be redesigned in such way that it will occupy more memory while performing calculations faster. It is still a subject of research. Presented GPU implementation also proved to be a proper direction of implementation, since most operations can be successfully parallelized. The bottleneck of the whole implementation will be always update performed on the database. ODBC drivers and native ADO.NET drivers behave relatively slow.