Near-Duplicate Detection

Marco Del Treppo

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Task

The task is to implement a detector of pairs of similar items, analyzing the "325 Bird Species" dataset published on Kaggle. The detector must consider the images in the dataset and output the pairs inferred as similar.

Dataset

Data set of 325 bird species. 47332 training images, 1625 test images and 1625 validation images. All images are 224 X 224 X 3 color images in jpg format. Data set includes a train set, test set and validation set. Each set contains 325 sub directories, one for each bird species.

As reported by the creator: "Once the image files for a species was downloaded they were checked for duplicate images using a python duplicate image detector program I developed. All duplicates detected were deleted in order to prevent their being images common between the training, test and validation sets."

For the purpose of this project all the images from train, test and validation set were used. The total number of picture is then 50582 that impose for a "brute force" approach to look at circa 2.558.538.724 different possible pairs.

Methodology

The major goal of finding candidate pairs was accomplished in a for loop using the list of all paths as reference. The subsequent steps were done for all the 50582 images.

Convert the image to gray scale and resize to 17 X 16. Compute the "signature" of the image using dHash. For each band of 51 binary values of the signature compute the hash using .tobytes() and map to the respective bucket.

From all the buckets with more than one item is possible to create the candidate set of pairs. For each pair of the candidate set is than computed the xor similarity to check if the candidate are similar or just false positive.

dHash

As reported on this "Hacker Factor" blog entry the dHash algorithm works on the difference between adjacent pixels. This identifies the relative gradient direction. In this case, the 17 pixels per row yields 16 differences between adjacent pixels. Sixteen rows of sixteen differences becomes 256 bits.

As for MinHash dHash may reduce the dimensionality of the data.

.tobytes()

Because of how the .tobytes() function was used can be seen as a hash function mapping a band of a signature to a bucket indeed is actually what it's happening.

Differently form what we have seen during the course here collision is minimized. Only equal band are mapped to the same bucket.

Parameters

hash_size

The **hash_size** parameter control the length of the signature. The number of possible signature depends on the **hash_size** with relation 2^{hash_size}. When **hash_size** value is low the number of collision is higher, the opposite is also true.

bands

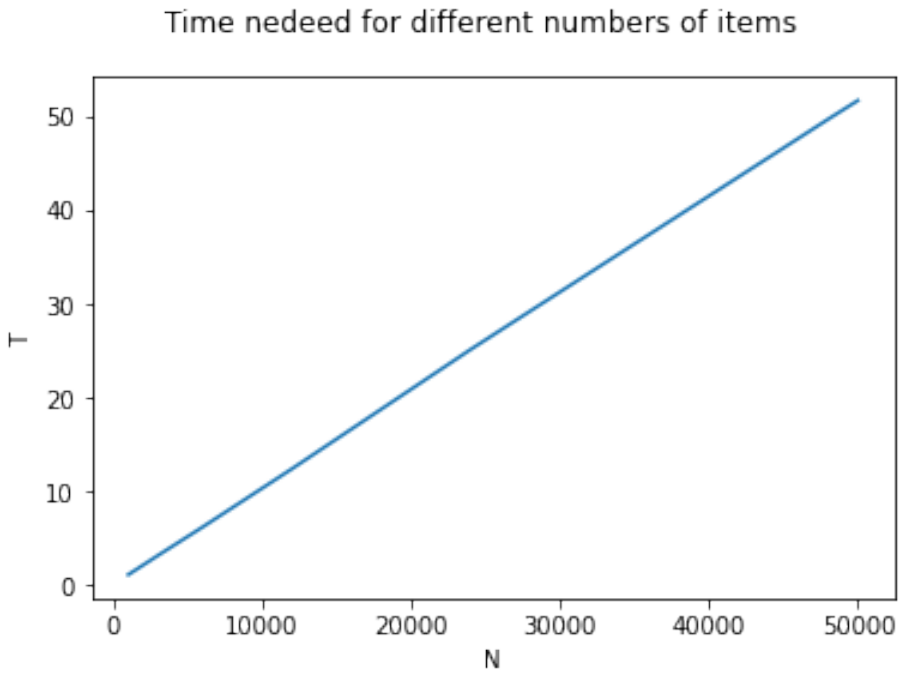
The **bands** parameter control the number of bands in which the signature will be split. An higher value in **bands** parameter will result in more candidate pairs. Assuming, correctly, that the probability for two images of sharing a signature band is higher when the signature band length is lower, higher value of **bands** will result in more candidate pairs after fixing the **hash_size**.

Result

Since the results depend on the signature size and the number of bands, they will not be reported in extended form in this section. The code to replicate the experiment is available on Github.

With a signature size of 256 and 5 bands it is evident that there are duplicates in the dataset. In particular the results show how there are duplicates between the different sets (test, train, validation). This was probably caused by an inefficient pre-processing before publishing the dataset on Kaggle.

To demonstrate the linearity in the complexity of this approach we show the graph of the time needed to complete the check and the number of images taken into account.



From the graph seems consistent to assume that the complexity of the problem is now $O(N)$.

Conclusion

In conclusion, the proposed approach seems efficient and scalable. The time to find duplicates among the over 50000 images is about 1 minute and a half and from the results, being present duplicates in the sets (train, test, validation) seems more efficient than the one used in the creation of the dataset.

Moreover most of the time in this specific experiment is spent in the resizing of the image and conversion to gray scale and is about one minute.

After having created a bucket list for each band it is easy to test the similarity of a new image against all those already analyzed.

A critical point is perhaps that of the choice of the function .tobytes() as Locality Sensitive Hashing. In the extreme case where every possible realization of a band consisting of 51 binary values was possible there would be 2.251.799.813.685.248 different possible buckets. Introducing a hash function with collision possibility could solve this problem at the expense of accuracy.