Table of Contents

Steps in Bundle Min Hashing

FlickrLogos-v2 folder is assumed to be in ./data/ directory of bundle_min_hashing_source folder

1. Create KeyPoints

File: create_keys.py
Run: $python create_keys.py

Installations needed:

1. Opencv - $apt-get install libopencv-dev python-opencv

Dependencies:

1. Convert jpg to pgm - use jpg2pgm.cpp - instructions to compile in the cpp file
2. Compile RootSIFT
    i) cd ./rootSift
    ii) make -j4 (if 4 cores else just make) - *ignore warnings*
    iii) For 64 bit machine executable will be made in folder **g64**

Variables to initialize:

1. BASE_PATH - Relative paths of images are added to this base path of flickr dataset
2. KEYS_FOLDER_BASE - Folder where key files will be made
3. RELPATH_FILENAME - File containing relative paths of images (file already in flickr dataset)
4. ROOTSIFT_PATH - Path to sift executable (keep space after path)
5. NUM_CORES - Number of cores to use
6. TOTAL_IMAGES - Num Images to train from the RELPATH_FILENAME

2. Create .h5 file for Clustering

We use library fastcluster by philbinj (details in next section) for fast clustering which works on .h5 files

This step creates a single .h5 file holding all keypoints together. Since num of keypoints is around 12M the size of file is around 6GB. Hence this step needs around **7GB of RAM.**

File: create_h5.py
Run: $python create_h5.py

Installations Needed:

1. Pytables - $pip install tables
2. Numpy - $pip install numpy

Dependencies:

1. Count total number of keypoints to create numpy array of that size and thus create .h5 file to be used for clustering
   Run $python count_keys.py - after setting **FILE_RELPATHS** and **KEYS_DIR** variables in count_keys.py

   You shall see “Total Keypoints = XXXXX” as final line of output

Variables to initialize:

1. MAX_POINTS_NUM - Set value to output of running dependency#1
2. KEYS_DIR - Folder containing all key files to train
3. OUTPUT_FILE - filename of output file (has to be a .h5 file)

3. Clustering using fastcluster

Using fastcluster library by philbinj - Github Link

This step clusters 12M keypoints to 1M keypoints. The output is .h5 file. The size of vocabulary file is around 420Mb.

Vocabulary File can be found on google drive. [Link] You can directly use this file for sketch creation.

Installations:

1. Git clone fastcluster to place of your choice (for installation follow its readme)
2. Git clone fastann to place of your choice (for installation follow its readme) - fastcluster needs fastann to be installed

Fastcluster readme asks to install openmpi which leads to error many a times.
Follow below steps to correctly install openmpi:

1. wget https://www.open-mpi.org/software/ompi/v3.0/downloads/openmpi-3.0.0.tar.gz - to location of your choice
2. tar zvxf openmpi-3.0.0.tar.gz
3. cd openmpi-3.0.0/
4. ./configure
5. sudo make -j2 all install
6. export LD_PRELOAD=/usr/lib/libmpi.so (the file should already be existing at that location)

Dependencies:

1. .h5 file containing keypoints as created in step2 needed

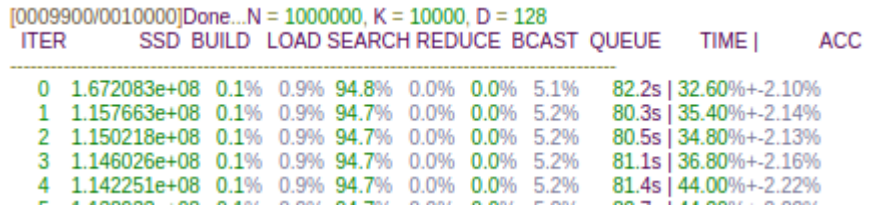
Variables to initialize:

1. K = 1000003 (Nearest prime number to 1M for ease of hashing later)
2. niters = 10 (you will get around 80% accuracy in 5-6 iters which don't change much in further iterations). It takes nearly 1hr per iteration
3. ntrees = 16
4. Specify name of input and output .h5 files

Run (remember to run it using tmux since it runs for around 10 hrs):

1. python fastcluster.py

Sample response on terminal



4. Convert .h5 dictionary to .txt

This is redundant step if you can directly read from .h5. I for sheer laziness to learn .h5 structure converted it back to simple format which can easily be read later.

File: h52txt.py
Run $python h52txt.py
Variables to initialize:

1. INPUT_H5_FILENAME - input .h5 dictionary file
2. OUTPUT_TXT_FILENAME - output txt dictionary file

5. Create train sketches

This step creates sketches for all training files and stores them in .txt file for each .key file.
We use fastann for fast assigning to VWs.
We use unique VWs in bundles to remove multiple copies of VW in a bundle which means burstiness increases chances of collision.

Three sketches are created for each bundle. The format of sketch file is:
Central_visual_word \t min_hash1 \t min_hash2 \t min_hash3 (tab separated)
Thus the 3 sketches of a bundle becomes - (central_visual_word, minhash1), (central_visual_word, minhash2), (central_visual_word, minhash3)

File: create_sketches.cpp
Compile: g++ -O2 `pkg-config --cflags opencv` -o create_sketch create_sketch.cpp `pkg-config --libs opencv` -lfastann
Run: ./create_sketch

Installations:

1. Git clone fastann to place of your choice (for installation follow its readme) - already done in step 3

Dependencies:

1. sketch.h - class to store sketch

Variables to initialize:

1. dictionary_path - path to txt file of dictionary
2. dir - path where the key files are kept
3. sketch_folder - path where you want sketches to be stored - make sure to create this folder before running code
4. BUNDLE_SIZE - min num of VWs in a bundle - we used 4 though paper states 3
5. NEIGHBORHOOD_RATIO - ratio of radius of central feature and cutoff radius for bundling - we use 2 though paper mentions to use 1
6. DICTIONARY_SIZE - 1000003 (nearest prime number to 1M - for ease in hashing)
7. 3 hashes - hasha1, hasha2 and hasha3 - no need to change

6. Load sketches to Redis

The sketch is stored as key:val pair in redis with key being the hash value of the sketch and the value being the name of the file whose sketch it is. Thus, each key in redis has multiple filenames as comma separated string.

File: load_sketches_redis_selectdb
Compile : g++ -o load_sketches_redis -O3 -pthread load_sketches_redis_selectdb.cpp -g -ggdb -lhiredis -lm
Run: ./load_sketches_redis

After running (check num of sketches in redis):

1. $redis-cli
2. 127.0.0.1:6379[1]> select 1 (to choose database)
3. 127.0.0.1:6379[1]> dbsize

This will return size of sketches.
Hence num of bundles = dbsize/3 (3 sketches for each bundle)

Installations:

1. sudo apt-get install redis-server
2. sudo apt-get install libhiredis-dev

Dependency:

1. sketch.h - self made class to store sketch

Variables to initialize:

1. sketches_train_relpaths - provided with the folder bundle_min_hashin_source
2. sketches_train_basepath - path to folder where sketches of training samples are kept
3. select_database - for redis (we can keep multiple versions of sketches in redis in different database using above variable)

Sharing the link of redis database [Link]
To directly load this db:

1. sudo service redis-server stop
2. sudo cp /path/to/dump.rdb /var/lib/redis/dump.rdb
3. sudo chown redis:redis /var/lib/redis/dump.rdb
4. sudo chmod 644 /var/lib/redis/dump.rdb
5. sudo service redis-server start

////// Check if the data is loaded ////
$redis-cli
127.0.0.1:6379[1]> dbsize
(integer) 15866335

7. Create test keypoints and sketches

Follow steps 1 and 5 to create test keypoints in folder keys_test and sketches in folder sketches_test

8. Match test sketches and note collisions

All sketches from the test file are converted to hashes and we look up for each hash in redis. Thus, we get list of comma separated filenames as collision result of test file.

File: match_sketches_redis.cpp
Compile : g++ -o match_sketches_redis -O3 -pthread match_sketches_redis.cpp -g -ggdb -lhiredis -lm
Run: ./match_sketches_redis

Output file format:
Test_file, filename_collision1, filename_collision2, filename_collision3, ….

Dependencies:

1. Redis

Variables to initialize:

1. sketches_test_relpaths - file containing relative path of sketches - given with code
2. sketches_test_basepath - folder containing test sketches
3. result_file - output file that stores collisions of each hash of test sketch file
4. select_database - 1 as default

9. Classify brand from collisions

File: eval.py
Run: python eval.py

Variables to initialize:

1. INPUT_FILE_COLLISIONS - input file containing output of step 8 which contains collisions of test file with train files
2. OUTPUT_FILE_CLASSIFIED - output file containing classification result
3. NUM_QUERIES - 960 query test images containing logo

OUTPUT_FILE_CLASSIFIED format:
Testfile, ground_truth_brand, classified_brand, judgement

Judgement:
True: classified_brand matches ground_truth_brand
NA: either no collision found or classified brand is no-logo
False: classified_brand does not match ground_truth_brand

The code prints precision and recall on terminal and creates classification output file.