Clustering Algorithms
2018-2019

Author: Nemanja Nedic
A.M. 1115201400124

Contents

Intro

Compile & Run

Makefile is provided for compilation.

Run:
./cluster -i input_file_path -o output_file_path -c configuration_file_path -d Metric

Metric must either be "euclidean" or "cosine".

In case bad parameters are given the program will terminate abruptly (exit). In that case the appropriate ErroCode will be set which you can check with "echo $?".

No memory leaks were found. The program was tested using valgrind.

NOTE: outfile_path can be given during the runtime to the program (it will request it on its own in case it not provided. This will create a outfile for you if you don't have one ready.

Organization of directories and files

Version Control: Git/Github

Unit testing library: Catch

Imported from project1 (Locality-Sensitive-Hashing):
ErrorCodes.hpp: error codes used to handle irregular termination
myvector: a std::vector with a std::string id
CosineSimilarity: Class CosineSimilarity
Euclidean: Class Euclidean
Metric: Abstract Class Metric
HashTable: Class HashTable
ReadInput: Functions to parse arguments, config and read data sets
utility: Various functions and templates (ie Exhaustive Search, Hamming Neighbors, Comparing vectors, Modulo operation, Vector Norms, progress

bar, ...)

New files:

main.cpp: general outline of the program

Cluster: Class Cluster

ClusterSpace: A set of clusters in the same vector space & implementations of clustering algorithms.

Design details

The algorithms and related functions are implemented as described in the slides by Prof.Emiris and Dr.Chamodrakas. For specific info about each algorithm or a class take a look in the code. A description of each function/class is provided in .hpp file and additional step-by-step comments in the .cpp file.

• Program parameters and stop criterias:

rangesearch_iterations: Since range search looks only inside the corresponding bucket, increasing the radius indefinitely would only result in the whole bucket being assigned to the cluster. To avoid that a max number of range search iterations is set in .conf file. This limits the radius to being at most $MinDistanceBetweenCenters * 2^{rangesearch\_iterations}$. Alternatevely if you have good knowledge of you dataset you could set this value by hand.

center_convergence_tolerance: When updating we must check to see if you centers converge to a value in order to avoid pointless iterations. Besides checking to see if our centers have changed since the last update (PAM) we can define a tolerance for our center accuracy. When a satisfactory tolerance is met, we can stop updates. This helps when dealing with K-means update for covering centers.

hypercube_probes: This parameter speifies the max hamming distance from each bucket in Hypercube Hashtable we can look into. i.e. hypercube_probes=1 means we'll examine the members of the bucket that the center of the hashtable hashes to, plus all its hamming_distance=1 neighbors.

Other/trivial stop criteria: All vectors are assigned & Max number of

iterations(set in .conf)

NOTE: When hashing for euclidean we use the inner product of two vectors which we divide by a integer and use the truncation of that division to hash. When distances is our dataset are too small (i.e twitter_dataset most distances are ¡ 0) we must amplify the inner_product and then divide by a w. For this purpose IP_COEFFICIENT is used to multiply the inner product before division.

Behavioral analysis by results

Naturally K-means update is much faster than PAM. i.e when K-means terminates in 2.5sec PAM will do 12-20sec for 1000vectors. This difference only increases exponentially (i.e. 17.18 vs 412sec, 12 vs 225,...) .

When examining the behavior of each update algorithm we conclude that the end result is more or less the same. The only thing that changes is the fact that we cannot use K-means as it is for points that can't be represented as vectors.When possible you should definitely prefer K-means even if it means projecting your data to a vector space beforehand.
Continuing with our comparison we will focus on K-means update in order to observe init and update results and behavior.

Im most cases the top 3 combinations all include Random Initialization. For Initialization algorithm comparison there doesn't seem to be a direct answer. Sometimes K-means++ Init works better than Random Init, but there is not enough info to support this claim. This could be attributed to the fact that both are very dependent on the pseudo-random generators and therefore they are not always consistent.

As for the Assignment algorithm there seems to be a clear benefit when using RangeSearch LSH independent of the Init Algorithm. Hypercube Range Search also offers decent results but in my opinion it seems to be very sensitive about the hamming_distance that is probed. With the corrent number of probes the Hypercube Range Search is potent.

Below are some tables with times for each combination. The parameters are the same as in the provided .conf file. For every test the top3 times are marked with green colour.

algorithms\num_vectors	1000	1000	5000	5000
RandomInit—Lloyds—K-means	1.33	1.18	6.23	17.18
RandomInit—Lloyds—PAM	17.53			412.50
RandomInit—RangeLSH—K-means	1.23	1.62	6.95	12.88
RandomInit——RangeLSH—PAM	16.40			225.36
RandomInit—RangeHypercube—K-means	2.47	2.04	7.46	7.86
RandomInit—RangeHypercube—PAM	11.73			418.45
K-means++—Lloyds—K-means	1.93	2.49	8.16	20.7
K-means++—Lloyds—PAM	10.04			402.4
K-means++—RangeLSH—K-means	1.61	1.83	15.16	7.62
K-means++—RangeLSH—PAM	16.31			315.53
K-means++—RangeHypercube—K-means	1.59	1.89	10.93	14.69
K-means++—RangeHypercube—PAM	7.60			398.98

Values are in seconds.