1η Εργασία 2021 - 2022

Συνεργάτες:

- **Βησσαρίων Μουτάφης , A.M:1115201800119, mail: sdi1800119@di.uoa.gr**
- **Αρίστη Παπασταύρου , A.M:1115201800154, mail: sdi1800154@di.uoa.gr**

1. Example Section

NOTE: Είναι απαραίτητο πριν από κάθε make, να κάνετε πρώτα make clean και μετά οποιοδήποτε make, μιας και από αλγόριθμο σε αλγόριθμο έχουμε κάποια διαφορετικά directives τα οποία μπορεί να μην κάνουν compile μέρος του κώδικα.

Για το compilation και run της εφαρμογής παρέχουμε ένα Makefile στο root directory. Οι πιθανές εντολές είναι:

```
~$ make lsh # make the ht test case
~$ make hypercube # make the hypercube test case
~$ In order to print the accuracy of each of the above algorithms in the
    output file you have to type the above commands like this: make hypercube
    verbose=1
~$ make clustering # make the clustering test case
~$ make clean # delete objective files
~$ bash ./run-cluster.sh [input file dir] [statistics output folder name] [
    output folder name] #script created to run all different methods of
    clustering and
```

2. Κατάλογος αρχείων πηγαίου κώδικα

Το project μας έχει οργανωθεί σε src και include folders. Ακόμη θα βρείτε 2 bash scripts και ένα Evaluation folder που έχουμε συμπεριλάβει outputs από runs που κάναμε με το small input.
Στο include folder θα βρείτε τις δηλήσεις όλων των κλάσεων του προγράμματος μας καθώς και τις δηλώσεις των συναρτήσεων που χρησιμοποιούνται μέσα στο πρόγραμμα. Τα αρχεία του include folder είναι τα εξής:

- ArgumentParser.hpp
- global_variables_namespace.hpp
- NearestNeighboursSolver.hpp
- ClusterSolver.hpp

- Hashing.hpp
- Point.hpp
- Evaluator.hpp
- hyper_cube.hpp
- Profiler.hpp
- FileHandler.hpp
- LSHNearestNeighbours.hpp
- Utilities.hpp

Στο src folder θα βρείτε την υλοποίηση όλων των συναρτήσεων του προγράμματος μας καθώς και την υλοποίηση των διάφορων constructors,destructors.Τα αρχεία του src folder είναι τα εξής:

- Το Cluster folder στο οποίο θα βρείτε τα cpp αρχεία που αφορούν την υλοποίηση του 2ου μέρους της εργασίας (clustering).Τα αρχεία αυτα είναι:
  1. AssignmentStepRoutines.cpp
  2. clustering.mk - make file helper file for clustering
  3. clustering_main.cpp - main function of clustering
  4. KMeans_pp_Solver.cpp - includes KMeans init/Lloyd/Reverse Assignment/Silhouette routines
- Το LSH folder στο οποίο θα βρείτε τα cpp αρχεία που αφορούν την υλοποίηση του LSH αλγορίθμου. Τα αρχεία αυτα είναι:
  1. lsh_main.cpp
  2. lsh.mk
  3. LSHNearestNeighbours.cpp
- Το Hypercube folder στο οποίο θα βρείτε τα cpp αρχεία που αφορούν την υλοποίηση του hypercube αλγορίθμου. Τα αρχεία αυτα είναι:
  1. hypercube.cpp
  2. hypercube_main.cpp
  3. hypercube.mk
- Το utils folder το οποίο περιέχει files cpp files με utility/general functions που χρησιμοποιούνται στα παραπάνω folders. Τα αρχεία αυτα είναι:
  1. ArgumentParser.cpp
  2. FileHandler.cpp

3. Profiler.cpp
4. Evaluator.cpp
5. Hashing.cpp
6. Utilities.cpp

Τέλος μέσα στο src folder θα βρείτε και τα παρακάτω αρχεία:

1. NearestNeighboursSolver.cpp
2. rules.inc
3. Point.cpp
4. src.inc

Το script, run.sh, εκτελεί διαφορετικά τρεξίματα για τους αλγορίθμους του 1ου σκέλους της εργασίας συνδυάζοντας κάθε φορά διαφορετικά hyper parameters.Αντίστοιχα το run_cluster.sh εκτελεί διαφορετικά τρεξίματα για τον K-Means με Lloyd, Reverse Assignment (LSH και HyperCube) με διαφορετκους συνδυασμούς από Hyperparameters.

Ενδεικτικό τρέξιμο:

```
~$ bash ./run.sh ./ ./ stats outs
~$ bash ./run-cluster.sh ./ stats outs
```

3. Point Class

Μία από τις πιο βασικές κλάσεις του προγράμματος μας είναι το Point όπου ουσιαστικά αναπαριστά το σημείο του οποίου κάθε φορά παίρνουμε τα coordinates από το input και query file.Το point περιέχει ένα vector με coordinates, id του point,τα dimensions του, ένα boolean-type-variable marked για το reverse assignment algorithm, και 2 int για το first και second closest centroid.Στην κλάση αυτή επίσης περνάμε ένα DistanceMetric dist, όπου ερμηνεύει την generic λειτουργικότητα αλλαγής μετρικών. Αυτή την στιγμή η default metric είναι η L2(euclid metric), και έχουμε υλοποιήσει και την L1. Ο χρήστης απλά αρκεί στην αντίστοιχη main, στον ορισμό του filehandler να γράψει το εξής: **FileHandler filehandler(L1_norm)**, για να αλλάξει μετρική. Παρόλα αυτά, αν ο χρήστης θελήσει να περάσει κάποια άλλη μετρική, μπορεί κάλλιστα να υλοποιήσει την αντίστοιχη συνάρτηση και να την περάσει σαν όρισμα στον FileHandler constructor της main, όπως δείξαμε και παραπάνω. Το μόνο προαπαιτούμενο είναι η συνάρτηση της νέας μετρικής να είναι ορισμένη με βάση τον τύπο:
typedef double (*DistanceMetric)(const vector<double>&, const vector<double>&);

4. Nearest Neighbours Implementations

4.1 Nearest Neighbours Solver

Αποτελεί γενικό interface το οποίο θα χρησιμοποιήθει στον Evaluator (δες παρακάτω), ώστε να αξιολογηθεί ο αλγόριθμος βάσει κάποιων μετρικών. Το output που ζητήθηκε ειναι η default μετρική. Για τις έξτρα μετρικές στο τέλος του output φακέλου θα πρέπει να κάνετε compile στο make με το argument $verbose = 1$ (δες άρθρο για Utilities/Evaluator)

4.2 Locality Sensitive Hashing Implementation

Ο αλγόριθμος LSH, βασισμένος στις k-hash functions με τύπο

$$h_i(\boldsymbol{p}) = \frac{\boldsymbol{p} \cdot \boldsymbol{v_i} + t_i}{w}$$

υλοποιήθηκε από τις classes *Hashing* και *LSHHasing*. Η κλάση Hashing φροντίζει για την παραγωγή και αποθήκευση των $\boldsymbol{v_i}$ και t_i καθώς και την απόδοση των τιμών $h_i(\cdot)$.

Η κλάση *LSHHashing*, κάνει την δουλειά των $g(\cdot)$. Είναι υπεύθυνη για τον υπολογισμό της τιμής

$$\sum_{i=1}^{k} r_i \cdot h(p),$$

όπου οι τιμές r_i είναι ξεχωριστές για κάθε amplified function g.

Η κλάση που υλοποιεί το LSH Nearest Neighbours, (*LSHNearestNeighbours*), αρχικοποιεί L συναρτήσεις και αντίστοιχα Hashtables και παρέχει συναρτήσεις για εισαγωγή του σημείου στα hashtable, για αναζήτηση nearest neighbour, k-nearest-neighbours και range-nearest-neighbours με βάση ένα σημείο query.

Οι δομες που χρησιμοποίηθηκαν ήταν

1. Ενα unordered_map με $\frac{dataset.size()}{8}$ Buckets με κλειδιά το bucket_id
2. Buckets: unordered maps με key το $ID(\boldsymbol{p}) = g(\boldsymbol{p})$ για να μπορεί να γίνει αμέσως ο έλεγχος $ID(query) = ID(p)$ στην αναζήτηση
3. Κάθε Bucket[ID] κελί περιέχει μία λίστα με pointers σε Point class, όπου περιέχονταν στην αρχή σε ενα list-dataset, το οποίο φτιάχνουμε 1 φορά σε όλο το αρχείο (δες FileHandler class)
4. Ένα set για να κρατάω τους nearest neighbours από το search space ώστε να εισάγονται ταξινομημένοι και να επιστρέφω τους πρώτους k
5. Μια λίστα rest όπου εισάγει όλα τα lists του bucket με στοιχεία που είχαν διαφορετικό $ID(p)$ από το query.

Implementation Notes

Η εισαγωγή είναι ελαφρά πιο αργή εφόσον θέλουμε να χρησιμοποιήσουμε και hashing ως προς το $ID(p)$ για να κάνουμε την αναζήτηση για σημεία με ίδιο ID, στο Bucket πιο αποδοτική. Ακόμα, αν δεν έχουμε βρει K γείτονες ή είμαστε στην περίπτωση range based search, ψάχνουμε όλο το bucket άρα και όλα τα στοιχεία που βρίσκονται μέσα στις λίστες της rest.

Για να ορίσουμε το παράθυρο w θα πάρουμε ενα sample 1000 αποστάσεων και θα ορίσουμε ως παράθυρο το **average distance**.

Η χρήση της κάθε amplified function γίνεται με απλή κλήση, λόγω του operator() overload. Οι τιμές h_i υπολογίζονται επιτόπου καθώς δεν υπήρχε λόγος για αποθήκευση. Οι πράξεις με mod γίνονται βάση αυτοσχέδιας συνάρτησης ώστε να παρουμε θετικό υπόλοιπο και να κάνουμε σωστό hashing. Όλες οι τιμές περιορίζονται σε 32-bit unsigned int μεταβλητές και τα κατάλληλα assert πραγματοποιούνται πριν επιστραφούν οι τιμές (ισως καθυστερήσει λίγο το performance αλλά είναι αναγκαίο sanity check).

Experimental Notes

Καθώς τρέξαμε τον αλγόριθμο με το μικρο input, παρατηρούμε πολύ μεγάλο speedup σε σχέση με το bruteforce κομμάτι, το οποίο μειώνεται όπως θα ήταν λογικό, καθώς το L αυξάνεται. Το ενδιαφέρον να παρατηρήσουμε εδώ ειναι το ηψηλό accuracy του αλγορίθμου καθώς και την μειωσή του καθώς το k αυξάνεται (ενώ το speedup αυξάνεται). Παραυτά η ταυτόχρονη αύξηση των k και L με ένα σταθερό ρυθμό και μεχρι ένα όριο, το οποίο παρατηρεί κανείς στα αρχεία που ανεβάσαμε, δείχνει την διατήρηση υψηλού speed up και accuracy, καθώς και την απότομη μείωση του average approximate error. Τέλος παρατηρούμε πως για μικρά k (πχ $k \in \{4, 5\}$) και μεγάλα L (άνω του 20) ο LSH είναι *πιό αργός* κατά μέσο ορο στο small dataset.

Ανεβάζοντας το N στο 10 και το 100, παρατηρούμε ξανά την επίδραση του k στην συσσώρευση των σημείων στα hash table και καταλαβαίνουμε πως ακριβώς επιδρά και ο συνδυασμός με τα r_i στην αποτελεσματικότητα του αλγορίθμου. Ειδικά στην περίπτωση των 100 γειτόνων παρατηρούμε πως το L και το k πρέπει να ξεπερνούν αρκετά τις default values για να πάρουμε *per point accuracy* της τάξης >90%.

4.3 HyperCube Implementation

Η τυχαιοποιημένη προβολή σε Hypercube είναι μια παρόμοια αλγοριθμική τεχνική με τον LSH. Ωστόσο, αντί για L hashtables με τη δική τους AmplifiedHashFunction, το σύνολο δεδομένων μας αποθηκεύεται στις κορυφές ενός Hypercube.
Αρχικά για τις ανάγκες υλοποίησης του Hypercube δημιουργήσαμε την κλάση HyperCube-Hashing. Αυτή η κλάση όπως και στο LSH,είναι derived της Hashing και κάθε φορά που ένα στοιχείο έρχεται ως είσοδος, μετατοπίζεται και τότε ο τύπος "h" υπολογίζεται σε αυτό.
Στη συνέχεια οι f functions,το μόνο που χρειάζεται να κάνουν είναι να αντιστοιχίσουν το αποτέλεσμα της h στο 0,1 (αναλόγως με το αν υπάρχει ήδη στις f ή όχι).
Τέλος δημιουργείται ένας υπερκύβος διάστασης dd παράλληλα με τα d' HashFunctions και f functions. Κάθε φορά που εισάγεται ένα στοιχείο, χασάρεται πρώτα από κάθε HyperCube-Hash function και στη συνέχεια αντιστοιχίστηκε στο 0,1 από τις f. Το αποτέλεσμα είναι ένας δυαδικός αριθμός μήκους dd (που είναι ισοδύναμος με έναν δεκαδικό αριθμό, δηλαδή τον δείκτη της κορυφής στην οποία το στοιχείο θα εισαχθεί).
Στη συνέχεια υπολογίζουμε ένα σύνολο κορυφών με απόσταση $hamming < hd$ και πραγματοποιούμε loop πάνω από τα σημεία που κατακερματίστηκαν στην ίδια κορυφή ή σε μια κορυφή με μια συγκεκριμένη απόσταση hamming.(Για να βρούμε τις κοντινές κορυφές κάθε φορά χρησιμοποιούμε την αναδρομική συνάρτηση **nearVertices**, με optimization όπου χρησιμοποιεί visited set και ουσιαστικά έχει παρόμοια λειτοθργικότητα με την dfs).

Οι δομες που χρησιμοποίηθηκαν ήταν :

1. Ενα unordered_map από Buckets με κλειδιά το bucket_id = $[f_1(h_1(p))...f_{d'}(h_{d'}(p))]$.
2. Buckets: list of Points classes.
3. Καθε $f_i(\cdot)$ σώζει τις τιμές της για τα διάφορα $h_i(\cdot)$ σε ένα unordered map.

Experimental Notes

Παρατηρούμε ότι ο Hypercube για τα default ορίσματα (k = 14, M=10, probes = 2) δεν αποδίδει καλά.Συγκεκριμένα τα στατιστικά είναι:

```
Total Stats:
Average per point accuracy: 0.03
Average brute_KNN/approx_KNN time ratio: 365.453
Average approx_KNN dist / true_KNN dist error: 1.04919
Max approx_KNN dist / true_KNN dist: 8.21462
```

Τα αποτελέσματα αυτά είναι αναμενόμενα. Αν σκεφτούμε πως με $K = 14$ κάνουμε decrease από 128 dimensions σε $dd = 14$, πολλά σπό τα points του dataset μας θα γίνουν hash στα ίδια buckets. Άρα με M = 10 και probes = 2 θα ψάξει μόνο 2 κορυφές και λίγα σημεία. Οπότε αν σε αυτό το συμπέρασμα, συμπεριλάβουμε και το γεγονός ότι το πρόγραμμα μας έχει και μια τυχαιότητα (κοντινά points μπορεί να κάνουν project σε 0 ή 1 (άρα πιθανότητα 50 %)) είναι λογικό να βγάζουμε χαμηλό accuracy. Έτσι λοιπόν αν αυξήσουμε το M και τα probes και δώσουμε input με ορίσματα:

$$k = 12, probes = 12, M = 3000$$

```
Total Stats:
Average per point accuracy: 0.69
Average brute_KNN/approx_KNN time ratio: 2.68306
Average approx_KNN dist / true_KNN dist error: 0.0622795
Max approx_KNN dist / true_KNN dist: 2.58739
```

$$k = 12, probes = 28, M = 5000$$

```
Total Stats:
Average per point accuracy: 0.81
Average brute_KNN/approx_KNN time ratio: 1.28808
Average approx_KNN dist / true_KNN dist error: 0.0398563
Max approx_KNN dist / true_KNN dist: 2.38207
```

5. Clustering

Το clustering υλοποιήθηκε σύμφωνα με το προηγούμενο evaluation paradigm:

- 1 File Reader για να διαβάσει το input από το input file
- 1 Kmeans++ Solver class όπου υλοποιεί όλη την λογική του clustering και διαχειρίζεται την δικιά του μνήμη. Ο Solver δέχεται μια συνάρτηση για το assignment step, το dataset και το πλήθος των clusters.
- 1 Evaluator class το οποίο φροντίζει να τρέξει το clustering να πάρει να αποτελέσματα, να κάνει το time-profiling του εκάστοτε αλγορίθμου και να εκτελέσει την silhouette για την αξιολόγηση. Στο τέλος τα εκτυπώνει όλα αυτά σε ένα out-file του οποίου το όνομα το ορίζει ο χρήστης.

Στην συνέχεια θα παρουσιάσουμε τα κομμάτια που χωρίστηκε ο αλγόριθμος KMeans++ και τα assignment step functions.

5.1 KMeans Clustering Solver

Η class Που χρησιμοποιείται για το clustering βασίζεται στην γενική μορφή του Kmeans αλγορίθμου και υλοποιεί το cluster init (Kmeans ++), το centroid update step και την κλήση του assignment step, καθώς και την loop που υλοποιεί την σύγκλιση του αλγορίθμου.

Kmeans++ Init

Ο αλγόριθμος για το cluster init βασίζεται στο Kmeans++ init. Ως μια βελτίωση του αλγορίθμου των διαφανειών, δημιουργούμε ενα hashtable με κλειδιά τα Points του dataset που δεν είναι centroids και τους αναθέτουμε ως values τα $D(i)$ για το εκάστοτε t. Κάθε φορά που το t αυξάνεται, αντί να ψάχνουμε τα t centroids για να βρούμε το min distance, απλά ανανεώνουμε το $D(i)$ με βάση την απόσταση από το πιο πρόσφατα inserted centroid και το $D(i)$ από τον πρηγούμενο γύρο. Με αυτό τον τρόπο μειώνουμε τους ελέγχους και τις αποστάσεις που πρέπει να υπολογιστούν. Επίσης για την γρήγορη αναζήτηση του κατάλληλου $P(r)$ σώζουμε όλες τις τιμές σε ενα set και ψάχνουμε εκεί βάσει uniform probability x.

Main Algorithm

Ο βασικός αλγόριθμος υλοποιεί την συνεχή ανάθεση σημείων και ανανέωση των centroids, λαμβάνοντας υπ'όψιν τα max_iter και τα max_points_changes ως threshold για να επιτευχθεί γρηγορότερη σύγκλιση και να μην γίνονται ανώφελες αναθέσεις (δες ClusterSolver.hpp). Τα points changes πρέπει να επιστρέφονται από τους assignment step αλγορίθμους. Για κοινή χρήση μεταξυ των assignment και Main loop, χρησιμοποιούμε και ένα table που ονομάζεται unassigned points και στο τέλος της main loop ελέγχει για points χωρίς cluster και τα εισάγει στο κοντινότερο cluster με brute force τρόπο (Χρήσιμο στο reverse assignment).

5.2 Reverse Assignment

Ο αλγόριθμος του reverse assignment υλοποιείται σύμφωνα με τις διαφάνειες και χρησιμοποιώντας εσωτερικές static μεταβλητές ώστε να αρχικοποιήσει 1 φορά την ακτίνα αναζήτησης και στο τέλος της κλήσης του να την διπλασσιάζει. Για να αποφευχθεί η απόκλιση θέτουμε ένα έλεγχο με βάση την ακτίνα αναζήτησης (δες R max στο αρχείο ClusterSolver.hpp) ώστε μετά από μία μέγιστη ακτίνα αναζήτησης, τα στοιχεία που μένουν στο unassigned table να αναθέτονται με brute force τρόπο. Για να παρατηρήσουμε τις συγκλίσεις των αλγορίθμων μπορούμε να παίξουμε με αυτές τις hyperparameters της σύγκλισης και να παρατηρήσουμε τις αναθέσεις που θα μας πετάξει στο stdout όταν κάνουμε compile με $verbose = 1$ στο make.

Οι 2 αλγόριθμοι που υλοποιούν το range search (LSH και Hypercube) δίνονται μέσω ξεχωριστών interface functions (δες AssignmentStep.cpp) έτσι ώστε να περνάμε τους Nearest neighbours solvers (οι οποίοι ορίζονται ως static objects για μοναδική αρχικοποίηση) στο γενικό interface του reverse assignment. Με αυτό τον τρόπο θα χρειαστεί να περνάμε 1 συγκεκριμένο τύπο assignment function στον constructor του cluster solver χωρίς να πρέπει να γνωρίζουμε την υλοποίηση του, ώστε να βασιζόμαστε στο 1ο ερώτημα χωρίς να χρειαστεί να συνδέσουμε με οποιον άλλο τρόπο τα εσωτερικά στοιχειά του ενός solver με τους άλλους.

Κατά την ανάθεση θέτουμε και το cluster id στο αντίστοιχο point έτσι ώστε να διευκολύνουμε την εκτέλεση της silhouette.

5.3 Lloyd

Στην συνάρτηση Lloyd, έχει υλοποιηθεί μόνο το βήμα ανάθεσης του αλγορίθμου μιας και το update part γίνεται από άλλη συνάρτηση στον K-Means, βάση της υλοποίησης μας. Η υλοποίηση του αλγορίθμου του Lloyd είναι πιστή εφαρμογή των διαφανειών. Συγκεκριμένα αυτό που κάνουμε είναι να:

- Parse όλα τα points του dataset μας
- Για κάθε point βρίσκουμε το closest centroid του
- Εάν βρισκόμαστε στο 1ο step του Lloyd τότε απλά πρσοσθέτουμε το point στο unordered map του κάθε cluster
- Αλλιώς, συγκρίνουμε το cluster_id του point (που ήδη έχει), με το closest_centroid id που βρήκαμε σε αυτό το loop.
- Εάν είναι ίδια, τότε απλά κάνουμε update την απόσταση του point από το centroid του cluster στο οποίο ήδη βρισκόταν.
- Αλλιώς, αναθέτουμε το point στο νέο του cluster, διαγράφουμε το point από το unordered map του παλιού cluster και αυξάνουμε τα points changed

Ένα σημαντικό *optimization* που κάναμε στην υλοποίηση αυτή, είναι στην κλάση του Point να κρατάμε το current cluster id στο οποίο έχει ανατεθεί το Point μας.Με τον τρόπο αυτό εξοικονομούμε χρόνο από το να κάνουμε search στα unoredered maps του κάθε cluster, για να βρούμε το point του dataset για το οποίο κάνουμε loop εκείνη την στιγμή.

5.4 Silhouette

Ο αλγόριθμος της Silhouette με complexity $O(k*n*m)+O(n^2)$, είναι αρκετά χρονοβόρος/κοστοβόρος μια και πρέπει για κάθε point που υπάρχει μέσα στο dataset μας, να βρούμε το second_closest_cluster σε αυτό (το οποίο με την ίδια λογική που ακολουθήσαμε στον Lloyd, το αποθηκεύουμε στο Point.). Οπότε ενώ κάνουμε parse το dataset, σώζουμε σε μεταβλητές τις τιμές των a και b για κάθε point (αντί να έχουμε vectors που να δεσμεύουν μνήμη προκειμένου να αποθηκεύουν συγκεντρωτικά όλες τις τιμες των a και b, και να έχουν και μεγάλο χρόνο search μετά). Τέλος επιστρέφουμε ένα vector centroid-θέσεων όπου περιέχουν την average silhouette value για κάθε cluster. (Για τον υπολογισμό του total average silhouette value απλά αθροίζουμε το προηγούμενο vector και το διαιρούμε με το πλήθος των clusters)

5.4.1 Experimental Notes

Όπως περιμέναμε στα μεγάλα input cases οι approximate algorithms συγκλίνουν γρηγορότερα με μεγάλυτερο βέβαια σφαλμα (μικρότερη τιμή της silhouette). Παίζοντας λίγο με τις hyperparameters θα παρατηρήσουμε πως όσο αυξάνεται το k και τα L/probes/M η σύγκλιση επιτυγχάνεται πιο γρήγορα, αλλά το κάθε assignment step παίρνει παραπάνω χρόνο.

6. Utilities

6.1 File Handler

Προκειμένου να έχουμε όσο λιγότερη επανάληψη κώδικα είναι δυνατή αλλά και περισσότερη απόκρυψη πληροφορίας, αντί να έχουμε στην main κώδικα για parse αρχείων και open/close files, δημιουργήσαμε μία FileHandler κλάση.Στο class αυτό, έχουμε private member ένα file descriptor, με τον οποίο ανοίγουμε/κλείνουμε αντίστοιχα κάθε φορά το αρχείο του οποίου το όνομα τους σώζεται σε ένα private member string. Ακόμα στην κλάση αυτή έχουμε συναρτήση για parse του input και query file, όπου αποθηκεύει τα points του πρώτου σε ένα std::list (το dataset μας).Τέλος έχουμε 2 συναρτήσεις **print_to_file** οι οποίες έχουν ρόλο να παράγουν τα αντίστοιχα output files που ζητούνται από την εργασία.

6.2 Evaluator

Ο evaluator αποτελεί βασικό εργαλείο στην εκτέλεση και εκτίμηση των αλγορίθμων. Αρχικά έχει συναρτήσεις που κάνουν evaluate και profile τους αλγορίθμους για το nearest neighbour. Οι συναρτήσεις αυτές δέχονται έναν solver ενα dataset-list και ορίσματα που αφορούν τα queries όπως το N, το R και το query file name. Μετά βαίνουν στην εκτέλεση των αλγορίθμων και ύστερα στην εκτέλεση των αξιολογήσεων.

Όσων αφορά τους Nearest Neighbours αλγορίθμους υπολογίζουν τους nearest neighbours με bruteforce τεχνική και (αν δώσετε το verbose=1 στο compilation) εκτυπώνουν τα στατιστικά που ζητάει η εκφώνηση (κάθως και αλλες μετρικές που θα περιγράψουμε παρακάτω).

Για το clustering, ακολουθούμε ίδια λογική με τα αντίστοιχα ορίσματα και το evaluation βάσει της shilouette.

Οι εκτυπώσεις βασίζονται όλες στον file handler καθώς αυτός υλοποιεί τις κατάλληλες ροουτινές και δίνει μια σχετική "καθαρότητα" στο module αυτό.

Οι έξτρα μετρικές στο Nearest Neighbours είναι οι εξής (έστω n queries)

1. Average per point accuracy $= \frac{1}{n}\sum_i^n \frac{appr\_neighbours\ in\ true\_neighbours}{true\_neighbours}$.
2. Average approximate algorithm time / brute force time over all queries.
3. Average proximity error $= \frac{1}{n}\sum_i^n \frac{1}{k_i}\sum_j^k {}_i\frac{d_{ij}}{d'_{ij}} - 1$, όπου d_{ij} είναι το approximate distance για τον j-th γείτονα του i-th query και αντίστοιχα το d'_{ij} είναι το true distance. Οπότε το κλάσμα πρέπει να αθροίσει σε 1 αν ο αλγόριθμος είναι 100% ακριβής και άρα το error να είναι μηδέν, ενώ θα είναι μεγαλύτερο του άσσου αν κάποια queries εχουν κάνει τοσο κακό perform ώστε να μην βρούν καν γείτονα.
4. Max proximity ratio όπου είναι το μέγιστο κλάσμα $\frac{d_{ij}}{d'_{ij}}$ βάσει της προηγούμενης μετρικής και δείχνει πόσο καλά αντιπροσωπεύει τους γείτονες η χειρότερη περίπτωση ανάμεσα σε όλα τα queries.

6.3 Argument Parser

Το module του ArgumentParser παρέχει ενα generic interface για να μπορέσουμε να σεταρουμε μεταβλητές με βάση την γραμμή εντολών που αντιπροσωπεύουν numerics, strings, flags. Ο argument parser δέχεται ως όρισμα το συνολικό αριθμό μεταβλητών που περιμένει να

παρει και **3 arrays (categorical, numerical, flags)**, μαζί με το μήκος τους. Ύστερα καλώντας την *parse_args(argc, argv)*, σετάρουμε τις τιμές με βάση την argument setter που έχουμε δώσει στον constructor του parser. Η argument setter συνάρτηση **πρέπει** να αναφέρεται σε global ή static για το αρχείο μεταβλητές, ώστε οι αλλαγές ανάμεσα στα module-calls και στο αρχείο που το χρησιμοποιεί να λάβουν μέρος. Όλοι οι τυπικοί έλεγχοι (λάθος όρισμα, λάθος αριθμός arguments) γίνεται από το argument parser.

6.4 Profiler

Το profiler module δίνει ένα πολύ απλό interface για να μπορέσουμε να χρονομετρήσουμε μια διαδικασία και να παρουμε το duration χωρίς να ασχοληθούμε με κλήσης και βιβλιοθήκες συστήματος. Στην αρχή καλούμε το init function, πριν ξεκινήσουμε την ρουτίνα που θέλουμε να χρονομετρήσουμε καλούμε την profiler_start() και αντίστοιχα την stop όταν τελειώσει η ρουτίνα. Για να πάρουμε τον χρόνο-διάρκεια, σε **milliseconds**, καλούμε απλά την profiler_get_duration() και εκτυπώνουμε τον χρόνο.

6.5 General Utilities

Ο φάκελος utils/Utilities.cpp περιέχει utility functions γενικού χαρακτήρα οι οποίες χρησίμευσαν στην γενίκευση ορισμένων ρουτινών και ενεργειών και το καλύτερο abstraction του κώδικα. Οι συναρτήσεις αυτές εκτείνονται από την παραγωγή τυχαίων νουμέρων εώς και το bitwise-estimation του hamming distance 2 32-bit αριθμών. Επίσης εκεί περιέχεται και το struct my_less που εξυπηρετεί ως overload της compare ορισμένων stl δομών που χρησιμοποιούνται σε σημεία του προγράμματος.