Project
Εργασία 1

Μέλη:
Σπυρίδων Μπριάκος (1115201700101)
Αποστολοπούλου Αλεξάνδρα (1115201700005)

-Μεταγλώττιση:
Για την μεταγλώττιση και το τρέξιμο του προγράμματος:

- make lsh && ./lsh -d data/train-images-idx3-ubyte -q data/t10k-images-idx3-ubyte -o output -k 4 -L 5 -N 1 -R 10000
- make cube && ./cube -d data/train-images-idx3-ubyte -q data/t10k-images-idx3-ubyte -o output -k 14 -R 10000 -N 1 -probes 2 -M 10
- make cluster && ./cluster -i data/train-images-idx3-ubyte -c data/cluster.conf -o output -complete no -m Hypercube

Γενικά μπορείτε να το τρέξετε και χωρίς παραμέτρους και μέσα στο πρόγραμμα δίνονται οι default τιμές που ορίζονται από την εκφώνηση.
Σημείωση: Γενικά για να δείτε τα αποτελέσματα στο αρχείο output, πρέπει να το κάνετε πριν δώσετε απάντηση (ναι ή όχι) στην ερώτηση αν θέλετε να συνεχίσετε την αναζήτηση με άλλο dataset, γιατί το έχουμε ρυθμίσει έτσι ώστε να αδείαζει κάθε φορά για να μπούνε τα καινούρια αποτελέσματα στο επόμενο τρέξιμο του προγράμματος.

-Αρχεία Κώδικα & Επικεφαλίδων:

- ***main.cpp:*** Εδώ γίνονται οι κατάλληλοι έλεγχοι εισόδου και αρχικοποιήσεις των μεταβλητών των παραμέτρων εισόδου. Ανάλογα με την μορφή εισόδου που έχει δωθεί (Lsh, Hypercube ή Cluster) γίνεται η κατάλληλη κλήση αρχικοποίησης των κλάσεων, προκειμένου να μπούνε τα δεδομένα στις διάφορες κλάσεις και να αρχικοποιηθούν όλες οι δομές. Μετά το πέρας κάθε ερωτήματος του χρήστη, τον ρωτάει αν θέλει να συνεχίσει την αναζήτηση για διαφορετικό αρχείο αναζήτησης (με απάντηση y/n).
- ***lsh.cpp:*** Αρχικά, υπάρχει συνάρτηση που διαβάζει το MNIST dataset και ενημερώνει τις δομές και τα μέλη της κλάσης. Ορίζει το HashtableSize να είναι ίσο με τον Αριθμό_των_εικόνων/8, το m = 423255 (καθώς με αυτό μετά από πολλές δοκιμές παρατηρήσαμε ότι με αυτή την τιμή γίνεται ο καλύτερος κατακερματισμός στα buckets), το M έτσι όπως ορίζεται στις διαφάνειες και το W=4000. Μετά κάνουμε exhausting αναζήτηση για να βρούμε τους κοντινότερος πραγματικούς γείτονες και τους αποθηκέυουμε σε έναν πίνακα και αρχικοποιούμε τα s_i. Ύστερα, βάζει τις εικόνες στα buckets.
  Μετά από όλα αυτά καλείται η συνάρτηση που ευθύνεται για τον Approximate Lsh, που στην ουσία για κάθε query ψάχνει σε ποιό bucket πέφτει κι αν δεν είναι άδειο τότε για κάθε εικόνα αποθηκεύει την Manhattan απόσταση τους σε ένα priority queue. Έτσι όταν τελειώσει την αναζήτηση στο τέλος να πάρει τους τους τοπ N κοντινότερους γείτονες μαζί με τις αποστάσεις τους.
  Τέλος, καλείται μία συνάρτηση που υπολογίζει το Range Search για να βρει τους κοντινότερους γείτονες που υπάρχουν μέσα σε μία ορισμένη ακτίνα R, η οποία έχει παρόμοια λειτουργικότητα με την approximate αλλά με έναν παραπάνω περιορισμό. Να σημειωθεί ότι παράλληλα με την εισαγωγή στην ουρά προτεραιότητας εισάγουμε και τα id των γειτόνων και σε ένα unorder set, έτσι ώστε να μπορέσουν να αποφευχθούν τα διπλότυπα.
  - ***lsh.h:*** Εδώ υπάρχει ο ορισμός της κλάσης LSH καθώς και τα πρότυπα των συναρτήσεων της.

- ***hypercube.cpp:*** Αρχικά, υπάρχει συνάρτηση που διαβάζει το MNIST dataset και ενημερώνει τις δομές και τα μέλη της κλάσης. Ορίζει το HashtableSize να είναι ίσο με 2^k, το m = 423255 (καθώς με αυτό μετά από πολλές δοκιμές παρατηρήσαμε ότι με αυτή την τιμή γίνεται ο καλύτερος κατακερματισμός στα buckets), το M έτσι όπως ορίζεται στις διαφάνειες και το W=4000. Μετά κάνουμε exhausting αναζήτηση για να βρούμε τους κοντινότερους πραγματικούς γείτονες, τους αποθηκέυουμε σε έναν πίνακα και αρχικοποιούμε τα s_i. Ύστερα, βάζει τις εικόνες στα buckets.
  Μετά από όλα αυτά καλείται η συνάρτηση που ευθύνεται για τον Approximate Hypercube, που στην ουσία για κάθε query ψάχνει σε ποιό bucket πέφτει κι αν δεν είναι άδειο τότε για κάθε εικόνα αποθηκεύει την Manhattan απόσταση τους σε ένα priority queue. Παράλληλα, αυξάνουμε έναν μετρητή για να φροντίσουμε να μην ελέγξουμε παραπάνω από M εικόνες. Σε περίπτωση που τελειώσουν οι εικόνες στα buckets και μένουν ακόμα εικόνες να εξετάσουμε ή το bucket τυχαίνει να είναι άδειο, τότε ψάχνουμε με απλή Hamming distance να βρούμε τα buckets που έχουν απόσταση αρχικά 1. Τότε ψάχνουμε κι αυτές τις εικόνες αυξάνοντας παράλληλα και τον μετρητή του probes για να μην ξεπεράσουμε τον max_probes που έδωσε ο χρήστης. Έτσι καταλαβαίνουμε πως άμα τελειώσουν όλα τα buckets με Hamming distance 1 και δεν δεν έχουμε φτάσει ακόμα τον μέγιστο αριθμό εικόνων αλλά ούτε τον μέγιστο αριθμό κορυφων, θα ψάξουμε τα buckets με Hamming distance 2 κοκ. Σε κάθε περίπτωση, ο αλγόριθμος διακόπτεται έαν φτάσουμε των μέγιστο αριθμό εικόνων ή κορυφων. Γι’ αυτόν τον λόγο είναι λογικό, για μικρό αριθμό εικόνων και κορυφών ο hypercube να πηγαίνει σχετικά χειρότερα από τον lsh.
  Τέλος, καλείται μία συνάρτηση που υπολογίζει το Range Search για να βρει τους κοντινότερους γείτονες που υπάρχουν μέσα σε μία ορισμένη ακτίνα R, η οποία έχει παρόμοια λειτουργικότητα με την approximate αλλά με έναν παραπάνω περιορισμό.
  - ***hypercube.h:*** Εδώ υπάρχει ο ορισμός της κλάσης HyperCube καθώς και τα πρότυπα των συναρτήσεων της.
- ***bucket.cpp:*** Εδώ υπάρχουν συναρτήσεις που σχετίζονται με την επεξεργασία/εισαγωγή στα buckets μας. Συγκεκριμένα, υπάρχουν συναρτήσεις που εισάγουν τις εικόνες των train δεδομένων μας στα buckets υπολογίζοντας τις g_i values (για lsh ή reverse assignment lsh) και τις f_i values αντίστοιχα (για hypercube ή reverse assignment hypercube). Ο υπολογισμός αυτών τον values γίνεται με την κατάλληλη εφαρμογή των τύπως για τις α_ι τιμές, μετά τις h_p τιμές με τον τύπο του exponatiation και τέλος την συνένωσή τους και την διαίρεσή τους με HashtableSize.
  - ***bucket.h:*** περιέχει τον ορισμό της κλάσης bucket καθώς και και τα πρότυπα συναρτήσεων της. Στην ουσία ως ένα bucket ορίζουμε ένα vector από ζεύγη εικόνων και των αρχικών g_i value τους. Η αποθήκευση των originall g_i values για κάθε εικόνα έγινε για να εξετάζουμε καθε φορά στον lsh τις εικόνες μόνο με ίδια αρχικά g_i values, κι όχι αυτά που έτυχαν να πέσουν στο ίδιο bucket μετά το %M . Βέβαια σε αυτό το σημείο καλό θα ήταν να σημειώσουμε πως αυτή η παραπάνω προσθήκη δεν βελτίωσε ούτε σε απόδοση ούτε σε χρόνο τον κώδικα.
- ***exhausting.cpp:*** Εδώ υπάρχουν οι δύο συναρτήσεις που υπολογίζουν την ακριβή απόσταση των εικόνων του query dataset με το train dataset και παίρνουμε τους N καλύτερους. Επίσης, υπάρχει και η συνάρτηση που υπολογίζει την Manhattan distance δύο διανυσμάτων εικόνων.
  - ***exhausting.h:*** Εδώ υπάρχουν τα πρότυπα των 3 συναρτήσεων που περιγράφθηκαν παραπάνω.
- ***kmeans.cpp:*** Εδώ υπάρχουν συναρτήσεις της κλάσης Kmeans. Μία εξ' αυτών είναι υπεύθυνη για την αρχικοποίηση των παραμέτρων της κλάσης που προέρχονται από το αρχείο cluster.conf. Επίσης, σε μία συνάρτηση γίνεται η αρχικοποίηση των κεντροειδών σύμφωνα με τον αλγόριθμο Kmeans++ των διαφανειών. Τέλος υπάρχει και η υλοποίηση της συνάτησης της σιλουέτας, η οποία ουσιαστικά υπολογίζει τις τιμές Silhouette του κάθε cluster αλλά και του συνολικού αλγόριθμου (μέσος όρος).

    - ***kmeans.h:*** Εδώ εμπεριέχεται ο ορισμός της κλάσης kmeans συνοδευόμενος από τα πρότυπα των συναρτήσεων που αναλύθηκαν παραπάνω. Αποδείχθηκε εξαιρετικά χρήσιμη και στις 3 μεθόδους clustering καθώς είναι αυτή που διαβάζει το mnist dataset. Επίσης, υπάρει ο ορισμός της κλάσης Nearest_Centroids, η οποία μας βοήθησε στην αποθήκευση 4 σημαντικών μεταβλητών για κάθε εικόνα του dataset μας. (1ο κοντινότερο κεντροειδές, 2ο κοντινότερο κεντροειδες καθώς και οι αντίστοιχες αποστάσεις)
- ***cluster.cpp:*** Εδώ υλοποιείται ο αλγόριθμος του clustering με την εξής λογική: εφόσον έχουν αρχικοιηθεί τα κεντροειδή με την kmeans++, τρέχει μία λούπα στην οποία αρχικά γίνεται η ανάθεση των images του dataset (είτε με lloyd method είτε με reverse assignemnt lsh είτε με reverse assignment hypercube). Έπειτα, υπολογίζουμε την τιμή της objective function κι αν ο ρυθμός μείωσης της objective function είναι μικρότερος από ένα κατώφλι ε ή ο αριθμός των επαναλήψεων έχει φτάσει το ανώτατα όριο, τερματίζουμε τον αλγόριθμο. Αν δεν τερματιστεί ο αλγόριθμος, τότε ανανεώνουμε τα κεντροειδή βάσει της τεχνικής k-medians. Να σημειωθεί πως υπάρχουν δύο συναρτήσεις update οι οποίες έχουν ακριβώς τον ίδιο σκοπό, απλώς η μία είναι πιο αποδοτική. Άπαξ και τελειώσει ο αλγόριθμος, υπολογίζεται η σιλουέτα και γίνονται οι κατάλληλες εκτυπώσεις.
    - **cluster.h:** Εδώ υπάρχει ο ορισμός της κλάσης cluster μαζί με τα πρότυπα των συναρτήσεων, η οποία ουσιαστικά δουλεύει κάθε φορά με μία από της μεθόδους cluster.
- ***ra_lsh.cpp:*** Εδώ υπάρχει συνάρτηση η οποία κάνει αρχικοποίηση με βάσει τον lsh αλγόριθμο που υλοποιήθηκε στο πρώτο ερώτημα. Επίσης, υπάρχει η lsh_assign() η οποία κάνει ανάθεση κεντροειών στα σημεία βάσει του αλγορίθμου lsh. Σε όσα σημεία δεν έχουν ανατεθεί κεντροειδή κάνουμε exhausing search(lloyd).
    - ***ra_lsh.h*****:** Εδώ υπάρχει ο ορισμός της κλάσης reverse assignment lsh μαζί με τα πρότυπα των συναρτήσεων.
- ***ra_hypercube.cpp:*** Εδώ υπάρχει συνάρτηση η οποία κάνει αρχικοποίηση με βάσει τον hypercube αλγόριθμο που υλοποιήθηκε στο πρώτο ερώτημα. Επίσης, υπάρχει η hypercube_assign() η οποία κάνει ανάθεση κεντροειών στα σημεία βάσει του αλγορίθμου hypercube. Σε όσα σημεία δεν έχουν ανατεθεί κεντροειδή κάνουμε exhausing search(lloyd).
    - **ra_hypercube.h:** Εδώ υπάρχει ο ορισμός της κλάσης reverse assignment hypercube μαζί με τα πρότυπα των συναρτήσεων.

<u>-Παρατηρήσεις:</u>

- ***Παράμετρος W:*** Κατά τον πειραματισμό μας με την μεταβλητή W, αντιληφθήκαμε πως (τόσο για τον lsh όσο και για τον hypercube, αφού η h values υπολογίζονται με τον ίδιο ακριβώς τρόπο) όσο το W έπαιρνε μεγάλες τιμές (πχ 40.000), τόσο ο κατακερματισμός στα buckets ήτανε μικρότερος. Αυτό οφείλεται στο γεγονός πως αυξάνονται οι πιθανότητες να έχουνε πολλές εικόνες ίδια g_i (lsh) ή f_i values (hypercube), διότι πάρα πολλά a_i είναι ίσον με -1. Αντιθέτως, εάν έχουμε πολύ μικρό W (πχ. 100-400) τότε έχουμε πολύ καλό κατακερματισμό στα buckets, κι άρα ο lsh αλγόριθμος θα εξετάσει λιγότερες εικόνες. Συμπερασματικά, για μεγάλες τιμές του W αντιλαμβανόμαστε πως θα έχουμε πολύ καλή ακρίβεια, αλλά θα κάνουμε "εκπτωση" στον χρόνο. Αντιθέτως, για μικρές τιμές W θα έχουμε εξαιρετικά καλή χρονική απόδοση, αλλά όχι μεγάλη ακρίβεια. Εμείς διαλέξαμε βάσει αυτών ως W=4000 θεωρώντας ότι ισσοροπήσαμε καλά μεταξύ ακρίβειας και χρόνου.
- ***Παράμετρος m:*** Μετά από εκτενή πειραματισμό, συμπεράναμε πως μας βοηθάει το m να είναι περιττός αριθμός και συνειδητοποησαμε πως ένας μεγάλος αριθμός (423255) μας έδωσε την καλύτερη αποτελεσματικότητα του αλγορίθμου.
- ***LSH vs HyperCube:*** Ο lsh στις περισσότερες περιπτώσεις μας δίνει καλύτερα αποτελέσματα σε σχέση με τον hypercube, με την προυπόθεση πως χρησιμοποιούμε τις default τιμές των παραμέτρων της εκφώνησης. Αν όμως αυξήσουμε τις παραμέτρους M, probes του hypercube, τότε παρατηρούμε εμφανή βελτίωση στην απόδοση το υπερκύβου.

- ***Clustering:*** Όσον αφορά το clustering, παρατηρήθηκε πως οι χρόνοι είναι παρεμφερείς και με τις τρείς μεθόδους, όπως και οι σιλουετ τιμές με πολύ μικρές διαφορές. Χαρακτηριστικά με τις τρεις διαφορετικές μεθόδους το clustering δίνει σιλουέτα από 0.05 εως 0.14. Μια διαφορά που παρατηρήσαμε είναι πως στην περίπτωση του reverse assignment γίνονται περίσσοτερες επαναλήψεις ουτώς ώστε να συγκλίνει ο αλγόριθμος έναντι του Lloyd's Clustering.

-Σημειώσεις:

- Αξίζει να σημειωθεί, ότι τα αποτελέσματα του αλγορίθμου και του κάθε αλγορίθμου φαίνονται στο αρχείο output και για όλες τις κλάσεις υπάρχουν destructor για την κατάλληλη αποδεύσμεση της μνήμης (κανένα memory leak, μπορεί να επιβεβαιωθεί και με τον valgrind).
- Υπάρχουν 3 script που μπορείτε να τρέξετε που κάνουν την μεταγλώττιση κι ύστερα το τρέξιμο με τις εξής εντολές:
  - ./scripts/lsh.sh
  - ./scripts/cube.sh
  - ./scripts/cluster.sh