Κ23α - Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2023– 2024

ΕΥΑΓΓΕΛΟΥ ΓΕΩΡΓΙΑ ΑΜ: 1115202000050 sdi2000050@di.uoa.gr

ΛΟΞΟΣ ΑΝΔΡΕΟΥ ΑΛΕΞΑΝΔΡΟΣ ΑΜ: 1115202000117 sdi2000117@di.uoa.gr

Κατασκευή γράφου k-πλησιέστερων γειτόνων για γενικά μέτρα ομοιότητας – NN Descent Algorithm

Εισαγωγή

Η κατασκευή γράγου k κοντινότερων γειτόνων (KNNG-K Nearest Neighbors Graph) είναι μια δημοφιλής τεχνική στη μηχανική μάθηση και την εξόρυξη δεδομένων, που χρησιμοποιείται μεταξύ άλλων για εργασίες ταξινόμησης και παλινδρόμησης. Χρησιμεύει ως ένα ισχυρό εργαλείο για τον εντοπισμό σχέσεων και εξαρτήσεων μέσα σε σύνολα δεδομένων, επιτρέποντας εφαρμογές στην ταξινόμηση, ομαδοποίηση και ανίχνευση προβλημάτων. Ένας KNNG καταγράφει την εγγενή δομή των δεδομένων αντιπροσωπεύοντας κάθε σημείο ως κόμβο σε ένα γράφημα και συνδέοντάς το με τους k πλησιέστερους γείτονές του με βάση μια μετρική ομοιότητας (π.χ. ευκλείδεια απόσταση, manhattan απόσταση κ.α).

Μία από τις προκλήσεις στην κατασκευή του KNNG έγκειται στην υπολογιστική πολυπλοκότητα, ιδιαίτερα όταν έχουμε να κάνουμε με μεγάλα σύνολα δεδομένων. Ο ακριβής υπολογισμός των k-πλησιέστερων γειτόνων για κάθε σημείο μπορεί να είναι απαγορευτικός όσον αφορά τις απαιτήσεις χρόνου και μνήμης. Για να αντιμετωπιστεί αυτό, έχουν αναπτυχθεί κατά προσέγγιση αλγόριθμοι για την αποτελεσματική κατασκευή του KNNG όπως ο αλγόριθμος NN-descent.

Ο αλγόριθμος NN-descent είναι μια ευρετική προσέγγιση που έχει σχεδιαστεί για να προσεγγίζει τους k-πλησιέστερους γείτονες για κάθε σημείο δεδομένων μέσα σε ένα δεδομένο σύνολο. Αναπτύχθηκε ως μέθοδος για την κατασκευή του KNNG και βασίζεται στη διαισθητική αντίληψη ότι οι γείτονες των γειτόνων είναι πιθανό να είναι γείτονες οι ίδιοι. Αυτή η ευρετική χρησιμοποιείται για να βελτιώσει επαναληπτικά την εκτίμηση των k-πλησιέστερων γειτόνων για κάθε σημείο.

Ο αλγόριθμος ξεκινά αρχικοποιώντας το γράφημα με τυχαίες ακμές που συνδέουν κόμβους με ένα σύνολο αρχικών γειτόνων. Στη συνέχεια, εισέρχεται σε μια επαναληπτική διαδικασία όπου, για κάθε κόμβο, εξερευνά τους γείτονες των γειτόνων του, βελτιώνοντας τους k-πλησιέστερους γείτονες με βάση υπολογισμούς απόστασης. Αυτή η τοπική προσέγγιση επιτρέπει στον αλγόριθμο να ενημερώνει αποτελεσματικά το γράφημα, επιδιώκοντας να βελτιώσει την ακρίβεια της προσέγγισης k-πλησιέστερου γείτονα.

Κυρίως, ο αλγόριθμο NN-descent χαρακτηρίζεται από την προσαρμοστικότητά του σε διαφορετικές μετρήσεις απόστασης. Οι χρήστες μπορούν να ορίσουν προσαρμοσμένες συναρτήσεις απόστασης, φιλοξενώντας σύνολα δεδομένων διαφορετικών διαστάσεων και χαρακτηριστικών. Αυτή η ευελιξία καθιστά το NN-descent ένα ευέλικτο εργαλείο για την κατασκευή KNNG σε ένα ευρύ φάσμα εφαρμογών.

Συνοπτικά, το k-Nearest Neighbors Graph και ο αλγόριθμος NN-descent αντιπροσωπεύουν συλλογικά μια κεντρική τομή της ανάλυσης δεδομένων και της υπολογιστικής αποτελεσματικότητας. Η σημασία τους εκτείνεται σε όλους τους κλάδους, παρέχοντας μια θεμελιώδη δομή για την κατανόηση των σχέσεων μέσα σε σύνολα δεδομένων και προσφέροντας ένα αποτελεσματικό μέσο προσέγγισης σε σενάρια όπου οι ακριβείς λύσεις είναι υπολογιστικά ακριβές.

Εφαρμογή 1 - Δημιουργία γράφου και αλγόριθμος NN-descent

- ***Αναπαράσταση γράφου:***

  Η αναπαράσταση του γράφου γίνεται μέσω λιστών γειτνίασης για κάθε κόμβο. Οι κόμβοι αποθηκεύονται σε έναν πίνακα που κρατείται στην δομή του γράφου μαζί με το πλήθος τους. Κάθε κόμβος λοιπόν διατηρεί μια λίστα γειτνίασης προς τους k κοντινότερους κόμβους και μια με όλους τους αντίστροφους γειτονές του, μαζί με τις συντεταγμένες του σημείου του και τον αριθμό του (θέση του στον πίνακα). Ο γράφος αρχικοποιείται με τυχαίες ακμές που συνδέουν τους κόμβους με τους αρχικούς K γείτονές τους.

- ***NN-descent αλγόριθμος για γράφο:***

  Ο βασικός αλγόριθμος που χρησιμοποιείται σε αυτόν τον κώδικα είναι ο αλγόριθμος NN-descent, όπως περιγράψαμε παραπάνω για την κατασκευή του KNNG και την εύρεση των K πλησιέστερων γειτόνων.Ο αλγόριθμος βασίζεται στην αρχή ότι ο γείτονας ενός γείτονα είναι πιθανό να είναι γείτονας. Επαναληπτικά βελτιώνει τους K πλησιέστερους γείτονες εξερευνώντας τους γείτονες των γειτόνων. Η απόσταση μεταξύ των κόμβων υπολογίζεται χρησιμοποιώντας μια συνάρτηση απόστασης που ορίζει ο χρήστης. Αυτή η συνάρτηση μπορεί να χειριστεί δεδομένα αυθαίρετων διαστάσεων και υποστηρίζει εναλλακτικές μετρήσεις απόστασης όπως Euclidean, Manhattan, Chebyshev κ.λπ. Ο αλγόριθμος ενημερώνει επαναληπτικά τον γράφο μέχρι να μην γίνονται περαιτέρω βελτιώσεις. Λαμβάνει υπόψη κάθε κόμβο στο γράφημα και εξερευνά τους γείτονες των γειτόνων του, βελτιώνοντας τους K πλησιέστερους γείτονες με βάση τους υπολογισμούς της απόστασης. Ο κώδικας χρησιμοποιεί μια βοηθητική δομή δεδομένων για τη διαχείριση γειτόνων και αποστάσεων KDistance στην οποία αποθηκεύεται προσωρινά για κάθε κόμβο οι k κοντινότεροι γείτονες μαζί με τις αποστάσεις τους. Ο κώδικας περιλαμβάνει συνάρτηση για την ταξινόμηση των k πλησιέστερων γειτόνων με βάση τις αποστάσεις που διατηρούμε στη βοηθητική δομή. Εάν απαιτείται ενημέρωση, ο

γράφος τροποποιείται ώστε να αντικατοπτρίζει τους νέους k πλησιέστερους γείτονες. Για τη βελτιστοποίηση του υπολογισμού, ο αλγόριθμος εκτελεί τοπικούς υπολογισμούς απόστασης για κάθε κόμβο ανεξάρτητα. Αυτό εξασφαλίζει αποτελεσματικές ενημερώσεις χωρίς περιττό πλεονασμό. Ο αλγόριθμος NN-descent επαναλαμβάνεται μέσω των κόμβων του γράφου, ενημερώνοντας τους k πλησιέστερους γείτονες με βάση τις αποστάσεις από τους γείτονες των γειτόνων. Εάν πραγματοποιηθούν ενημερώσεις κατά τη διάρκεια της επανάληψης, ο γράφος ενημερώνεται κατάλληλα ώστε να αντικατοπτρίζει τους νέους k πλησιέστερους γείτονες. Η διαδικασία συνεχίζεται επαναληπτικά έως ότου δεν είναι δυνατή η περαιτέρω ενημέρωση, με αποτέλεσμα έναν KNNG.

- ***NN-descent αλγόριθμος για σημείο στο επίπεδο:***

  Ο αλγόριθμος NN-descent μπορεί να εφαρμοστεί επίσης σε ένα μόλις σημείο στο επίπεδο για την εύρεση των k πλησιέστερων σημείων του από το σύνολο των κόμβων του γράφου. Ακολουθώντας λοιπόν παρόμοια στρατηγική με την παραπάνω ξεκινώντας από δοσμένο σημείο διαπερνάμε όλους τους κόμβους του γράφου και υπολογίζουμε αποστάσεις. Χρησιμοποιείται και πάλι η βοηθητική δομή KDistance στην οποία διατηρούμε ύστερα από κάθε επανάληψη τους k κοντινότερους κόμβους του σημείου, ταξινομημένους. Η διαδικασία αυτή επαναλαμβάνεται εως ότου έχουμε επισκευτεί όλους τους κόμβους τους γράφου. Τελικά στην δομή KDistance έχουμε του k κοντινότερους κόμβους του σημείου προς αναζήτηση.

Παρατηρούμε, λοιπόν οτι η υλοποιημένη αναπαράσταση γράφου και ο αλγόριθμος NN-descent παρέχουν ένα ευέλικτο πλαίσιο για την αποτελεσματική εύρεση των k πλησιέστερων γειτόνων τόσο σε σενάρια γράφου όσο και σε σενάρια μεμονομένου σημείου στο χώρο. Ο κώδικας επιδεικνύει ευελιξία επιτρέποντας στους χρήστες να ορίζουν προσαρμοσμένες λειτουργίες απόστασης και υποστηρίζει διάφορες μετρικές απόστασης. Πρόκειται λοιπός για έναν αρκετά αποτελεσματικό και ευέληκτο αλγόριθμο, που επιδέχεται περεταίρω βελτιστοποιήσεις όπως θα δούμε στη συνέχεια.

Εφαρμογή 2 - Βελτιστοποιήσεις Local Join , Σταδιακή Αναζήτηση , Δειγματοληψία και Πρόωρος Τερματισμός

Στο δεύτερο στάδιο της εφαρμογής υλοποιήθηκαν βελτιστοποιήσεις του αλγορίθμου NN-descennt που είχε ολοκληρωθεί στο πρώτο. Η όλη διαδικασία πραγματοποιείται στην συνάρτηση local join η οποία έχει την εξής μορφή:

void local_join(Graph* graph, int k, float (distance_value)(point, point), double d, double p)

- ***Local Join :***

  Για την πρώτη βελτιστοποίηση δηλαδή το local join έχουμε χρησιμοποιήσει την δομή KDistance. Η δομή αυτή περιέχει ένα κόμβο, μία απόσταση και ένα flag. Μέσα σε κάθε Node έχουμε έναν πίνακα από KDistance το ljarray που πρακτικά κρατάει σε κάθε επανάληψη του αλγορίθμου το ποιοι κόμβοι πρέπει να είναι οι k γείτονες του συγκεκριμένου κόμβου μετά από το τρέχον iteration. Ο πίνακας με τα KDistance γίνεται sort (με την συνάρτηση void sort(KDistance** kd, int k)) ώστε να γνωρίζουμε ότι ο k γείτονας του κόμβου με την μεγαλύτερη απόσταση από αυτόν βρίσκεται πάντα στην τελευταία θέση αυτού του πίνακα. Για τον αλγόριθμο, για κάθε κόμβο του γράφο παίρνουμε τους γείτονες του στη λίστα neighbors (αντίστροφους και κανονικούς σε μία λίστα, χρησιμοποιώντας την συνάρτηση ListNode* connectlist(ListNode* a, ListNode* b) και βλέπουμε αν κάθε γείτονας πρέπει να μπει στην λίστα kneighbors του άλλου και αν πρέπει, και δεν υπάρχει ήδη στην λίστα τον αποθηκεύουμε προσωρινά στον πίνακα ljarray. Οι διπλοί υπολογισμοί αποφεύγονται υπολογίζοντας τις αποστάσεις του κάθε γείτονα στη λίστα neighborσs μόνο με τους κόμβος που είναι μετά από αυτόν στην λίστα, αφού με αυτόν τον τρόπο ο έλεγχος με τους προηγούμενος έχει πραγματοποιηθεί όταν εξετάζαμε ως currentnode τους προηγούμενος κόμβος της λίστας. Όταν τελειώσει αυτή η διαδικασία τότε βλέπουμε για ποιους κόμβους οι κόμβοι του ljarray είναι διαφορετικοί από τους κόμβους στη λίστα kneighbors και για αυτούς τους κόμβους κάνουμε update τους γείτονες.

- ***Σταδιακή αναζήτηση (Incremental Search) :***

  Για την δεύτερη βελτιστοποίηση προσθέσαμε 2 bool flags το ένα στο Listnode. Σε κάθε επανάληψη η τιμή του flag αποθηκεύεται προσωρινά όπως και οι κόμβοι με ljarray και στο τέλος μαζί με τους κόμβους ανανεώνονται όπου χρειάζεται και το flag των κόμβων. Αν υπήρχαν από πριν οι κόμβοι θα πάρουν ως flag false, ενώ αν είναι καινούργιοι στην λίστα η τιμή του flag γίνεται true.

- ***Δειγματοληψία :***

  Αρχικά η μεταβλητή p έχει οριστεί να έχει τιμή 0.5 όμως μετά τροποποιήθηκε ώστε να δίνεται από τον χρήστη. Η δειγματοληψία για όλους τους κόμβους γίνεται στην αρχή κάθε επανάληψης. Εκεί χρησιμοποιούμε τις εξής 3 συναρτήσεις για κάθε κόμβο του γράφου (Node* rednode) : ListNode* true_neighbors(ListNode* list), ListNode* false_neighbors(ListNode* list), ListNode* getpk(int pk, ListNode* list). Η συνάρτηση true_neighbors χρησιμοποιείται για να μας επιστρέψει τους γείτονες του rednode οι οποίοι έχουν true flag, οπότε την καλούμε μία φορά για την λίστα των k γειτόνων και μία φορά για την λίστα των αντίστροφων γειτόνων. Αφού έχουμε πάρει όλους τους true γείτονες του κόμβου στις λίστες new_kneighbors και new_rneighbors τότε καλούμε την συνάρτηση getpk για αυτές τις δύο λίστες ώστε να μας επιστρέψει μια λίστα με μέγεθος pk (pk = p * k) ή με μέγεθος μικρότερο του pk αν και μόνο αν τα στοιχεία της λίστας new[i] είναι λιγότερα από pk. Όταν γίνει αυτό, συνδέουμε τις 2 λίστες σε μία και την βάζουμε στον πίνακα new (χρησιμοποιούμε έναν πίνακα με λίστες, όπου στον πίνακα σε κάθε θέση i βάζουμε την λίστα που προκύπτει από την δειγματοληψία του κόμβου με id i). Επίσης για κάθε κόμβο βάζουμε την λίστα με όλους τους γείτονες του με false flag στον πίνακα old (αντίστοιχος πίνακας λιστών). Επομένως μετά το local join πραγματοποιείται υπολογίζεται μόνο για τους επιλεχθέντες κόμβους και μεταξύ των επιλεχθέντων κόμβων και παλιών κόμβων.

- ***Πρόωρος τερματισμός :***

  Για τον πρόωρο τερματισμό αρχικά είχε ορίσει το d = 0,001 όμως και αυτό αργότερα τροποποιήθηκε ώστε να δίνεται από τον χρήστη. Γενικά απλώς κρατάμε τον αριθμό των αλλαγών στην μεταβλητή updatecounts και όταν αυτός ο αριθμός σε κάποια επανάληψη παραμείνει μικρότερος του d*k*numofnodes τότε τερματίζεται η όλη διαδικασία!

Εφαρμογή 3 - Βελτιστοποίηση υπολογισμού απόστασης , Random Projection Trees και Παραλληλοποίηση

- ***Βελτιστοποίηση υπολογισμού απόστασης :***

  Στο τρίτο παραδοτέο αρχικά ζητήθηκε να υλοποιήσουμε μια βελτιστοποίηση για τον υπολογισμό της ευκλείδειας απόστασης αφού δεν χρειάζεται να υπολογίζεται και η τετραγωνική ρίζα που ορίζει η απόσταση. Αρχικά προσθέσαμε στη δομή point την νόρμα του σημείου και μετά χρησιμοποιήσαμε την συνάρτηση float dot_product(point p1, point p2) για τον υπολογισμό της απόστασης σύμφωνα με την εκφώνηση.

- ***Αρχικοποίηση KNN γράφου :***

  Η δεύτερη βελτιστοποίηση πραγματοποιήθηκε στην συνάρτηση void randomprojection(void* args) ενώ χρησιμοποιήθηκαν και κάποιες βοηθητικές συναρτήσεις (void normalvector(int p1, int p2, Node** nodes, float* vector), int* splithyperplane(float* vector, int dim, int* subset, Node** nodes, int* size)).
  Στην randomprojection επιλέγουμε τυχαία δύο σημεία και όσο το μέγεθος του subset είναι μικρότερο από τον αριθμό D τότε καλούμε την συνάρτηση normalvector η οποία υπολογίζει το κανονικό διάνυσμα μεταξύ αυτών των δύο σημείων και μετά καλεί την

splithyperplane χωρίζοντας και πάλι το subset. Όταν ολοκληρωθεί αυτό το while και έχουμε πλέον το subset στο επιθυμητό μέγεθος, καλούμε την getknodes η οποία βρίσκει και αναθέτει τις αποστάσεις μεταξύ των σημείων του subset.

- ***Παράλληλη Εκτέλεση :***

Αρχικά για την παράλληλη εκτέλεση χρειάστηκε να δημιουργηθούν δύο δομές, η δομή job που περιλαμβάνει μια συνάρτηση και τα argument της, και η δομή του jobscheduler. ¨Έγινε παραλληλοποιήση του random projection αλλά και ο υπολογισμός της νόρμας των σημείων. Για την χρήση του job scheduler εχουμε τις εξής συναρτήσεις :

i) JobS* initialize_scheduler(int execution_threads) η οποία κάνει initialize τον scheduler,το mutex και το condition variable, δεσμεύει μνήμη για τα περιεχόμενα του και κάνει set το destroy flag σε false.
ii) int submit_job(JobS* sch, Job* j) η οποία βάζει το job στο queue του scheduler και κάνει signal το condition variable.
iii) int start_execute(JobS* sch) η οποία τα threads τα οποία θα τρέχουν την execute συνάρτηση.
iv) void* execute(void* s) η οποία περιμένει συνεχώς για τα jobs στο queue και μετά τα εκτελεί .
v) int wait_all_tasks_finish(JobS* sch) η οποία περιμένει ουσιαστικά να τελειώσουν όλες οι δουλειές στο queue και καλείται πριν καλέσουμε την συνάρτηση destroy.
vi) int destroy_scheduler(JobS* sch) η οποία κάνει set το destroy flag σε true, καταστρέφει τον scheduler, κάνει broadcast σε όλα τα Threads που περιμένουν και αποδεσμεύει την μνήμη που είχε δεσμευτεί.

Η χρήση του scheduler γίνεται στην main συνάρτηση μας που κάνουν submit έναν αριθμό από randomprojections καθώς και κατά την εκτέλεση του όταν έρθει η ώρα να υπολογιστούν οι νόρμες. Αφού ολοκληρωθούν όλες οι δουλειές, τότε καταστρέφουμε τον scheduler ολοκληρώνουμε τον γράφο και εκτελούμε τον αλγόριθμο.

Στατιστικά και Μετρήσεις

Παρακάτω παρουσιάζονται μετρήσεις από κάθε στάδιο της εργασίας , όσον αφορά την αποτελεσματικότητα του κώδικα, σε σύγκριση με τα αποτελέσματα του brute force αλγορίθμου (Similarity Percentage), και τον χρόνο διάρκειας κάθε εκτέλεσης (Execution Time).

Εφαρμογή 1:

Dataset / K	20 /K 10	50 /K 10	200 /K 10	200 /K 50	300 /K 10	300 /K 100	500 /K 200	1000 /K 20
Similarity Percentage	100%	100%	100%	100%	98,23%	100%	100%	99,01%
Execution Time	0,0675s	0,13s	1,024s	9,932s	2,209s	59,806s	7m 53,015s	38,738s

Σχόλια / Παρατηρήσεις:

Από τα παραπάνω στοιχεία παρατηρούμε τα εξής:

1. Όσο μεγαλύτερο είναι το K σε σχέση με το μέγεθος των δεδομένων τόσο μεγαλύτερη είναι και η αποτελεσματικότητα του αλγορίθμου.
2. Όσο μεγαλύτερο είναι το K τόσο μεγαλύτερος και ο χρόνος εκτέλεσης.
3. Οι χρόνοι για μεγαλύτερα datasets και μεγαλύτερα K ήταν αρκετά μεγάλοι για να εφαρμοστούν πειράματα.

Εφαρμογή 2:

Dataset / K	20 /K 10	50 /K 10	200 /K 10	200 /K 50	300 /K 10	300 /K 100	500 /K 200	1000 /K 20	1000 /K 200	2000 /K 200
Similarity Percentage	100%	100%	95,9%	100%	93,3%	100%	100%	97,1%	100%	100%
Execution Time	0,043s	0,082s	0,314s	2,970s	0,527s	9,667s	1m 18,105s	5,534s	2m 38,217s	6m 15,703s

Σχόλια / Παρατηρήσεις:

Ύστερα από τις βελτιστοποιήσεις της εφαρμογής 2 (Local Join , Σταδιακή Αναζήτηση , Δειγματοληψία και Πρόωρος Τερματισμός) παρατηρούμε τις εξής αλλαγές στην εκτέλεση του κώδικα:

1. Αρχικά βλέπουμε ότι η εκτέλεση έγινε αρκετά ταχύτερη, σε σημείο που μας επιτράπηκε να εφαρμόσουμε τον κώδικα και σε μεγαλύτερο αριθμό δεδομένων.
2. Χάνουμε λίγη ακρίβεια στα αποτελέσματα του κώδικα κυρίως μέσα από τον πρόωρο τερματισμό και την δειγματοληψία.

Βλέπουμε δηλαδή ότι μετά από έναν αριθμό βελτιστοποιήσεων στον κώδικα θυσιάζοντας λίγη από την ακρίβεια των αποτελεσμάτων κερδίζουμε αρκετά σε ταχύτητα (σχεδόν υποδιπλασιασμός του χρόνου εκτέλεσης) ώστε να μπορέσουμε να πειραματιστούμε περαιτέρω και με τα μεγαλύτερα datasets.

Εφαρμογή 3:

Dataset / K	20 /K 10	50 /K 10	200 /K 10	200 /K 50	300 /K 10	300 /K 100	500 /K 200	1000 /K 20	1000 /K 200	2000 /K 200
Similarity Percentage	100%	97%	85,4%	98,3%	80,6%	99,75%	99,8%	84,9%	99,57%	99,09%
Execution Time	0,032s	0,050s	0,215s	0,657s	0,218s	3,751s	51,194s	6,386s	4m 49,520s	25m 16,181s

Σχόλια / Παρατηρήσεις:

Στο τελευταίο στάδιο της εργασίας εφαρμόστηκε βελτιστοποίηση στους υπολογισμούς αποστάσεων και στην αρχικοποίηση του γράφου με random projection trees. Επίσης η βασικότερη υλοποίηση στον τελικό κώδικα είναι η παραλληλοποίηση των εκτελέσεων των παραπάνω, το οποίο επέφερε τα εξής αποτελέσματα:

1. Παρατηρούμε αρχικά ότι για μικρό μέγεθος δεδομένων υπήρξε μια μικρή βελτίωση στον χρόνο εκτέλεσης χωρίς ιδιαίτερες αλλαγές στην αποτελεσματικότητα του αλγορίθμου.
2. Ωστόσο αυτό που προκαλεί ιδιαίτερο ενδιαφέρον είναι η αρκετά μεγάλη αύξηση του χρόνου στα μεγαλύτερα datasets, το οποίο μπορούμε να καταλάβουμε ότι προκαλείται από την διαδικασία του random projection για την αρχικοποίηση του γράφου, καθώς μπορεί να επιφέρει σχετικά απρόβλεπτη συμπεριφορά λόγω της τυχαιότητας του αλγορίθμου.

Συμπερασματικά, τα πειράματα που διεξήχθησαν στον αλγόριθμο NN-descent για τη δημιουργία ενός γράγου k-πλησιέστερων γειτόνων έδωσαν πολύτιμες πληροφορίες για την αποτελεσματικότητα και την

απόδοσή του. Οι στατιστικές μετρήσεις που παρουσιάζονται για κάθε εφαρμογή υπογραμμίζουν βασικές πτυχές της συμπεριφοράς του αλγορίθμου.

Συνολικά, τα πειράματα υπογραμμίζουν την αντιστάθμιση μεταξύ της αποτελεσματικότητας του αλγορίθμου, της ακρίβειας και του χρόνου εκτέλεσης. Τα ευρήματα υποδηλώνουν ότι η προσεκτική εξέταση των τιμών των παραμέτρων και των τεχνικών βελτιστοποίησης είναι ζωτικής σημασίας, ειδικά όταν πρόκειται για μεγαλύτερα σύνολα δεδομένων.