ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΣΥΣΤΗΜΑΤΩΝ ΜΕΤΑΔΟΣΗΣ ΠΛΗΡΟΦΟΡΙΑΣ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΛΙΚΩΝ

Τρισδιάστατη ταξινόμηση δεδομένων γονιδιακής έκφρασης με τη χρήση μεθοδολογιών μηχανικής μάθησης: Χρωμοσωμικές ταξινομήσεις σε δύο στάδια

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μαρία Α. Σδράκα

Επιβλέπων: Δημήτριος Κουτσούρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΣΥΣΤΗΜΑΤΩΝ ΜΕΤΑΔΟΣΗΣ ΠΛΗΡΟΦΟΡΙΑΣ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΛΙΚΩΝ

Τρισδιάστατη ταξινόμηση δεδομένων γονιδιακής έκφρασης με τη χρήση μεθοδολογιών μηχανικής μάθησης: Χρωμοσωμικές ταξινομήσεις σε δύο στάδια

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μαρία Α. Σδράκα

Επιβλέπων: Δημήτριος Κουτσούρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Ιουνίου 2016.

.....	.....	.....
Δ. Κουτσούρης Καθηγητής Ε.Μ.Π.	Γ. Ματσόπουλος Αναπληρωτής Καθηγητής Ε.Μ.Π.	Γ. Λάμπρου Μέλος Ε.ΔΙ.Π. Ιατρικής Σχολής Ε.Κ.Π.Α.

Αθήνα, Ιούνιος 2016

.....

Μαρία Α. Σδράκα

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι μικροσυστοιχίες DNA αποτελούν μία από τις πιο διαδεδομένες πειραματικές μεθόδους στη γονιδιακή ανάλυση ιστολογικών δειγμάτων. Μέσα από αυτές τις διατάξεις γίνεται εφικτή η επισκόπηση της έκφρασης μεγάλου όγκου γονιδίων από πολλαπλά δείγματα ταυτόχρονα. Το γεγονός αυτό τις καθιστά ιδανικό εργαλείο για την ανάλυση και μελέτη καρκινικών ιστών, με στόχο την εξεύρεση των κανόνων που διέπουν γενικά το μηχανισμό της ογκογένεσης και την περαιτέρω κατανόηση της νόσου.

Στην παρούσα εργασία εστιάσαμε τις προσπάθειές μας στην αξιοποίηση της πληροφορίας ότι κάθε γονίδιο σε έναν οργανισμό ανήκει αποκλειστικά σε κάποιο χρωμόσωμά του. Έτσι, δημιουργήσαμε μια νέα τρισδιάστατη δομή, διαχωρίζοντας τα γονίδια ανά χρωμοσωμικό ζεύγος, και στη συνέχεια εφαρμόσαμε διάφορες τεχνικές συσταδοποίησης, προκειμένου να απομονώσουμε τις ομάδες γονιδίων που παρουσιάζουν κοινό προφίλ έκφρασης.

Πιο συγκεκριμένα, αναπτύξαμε μια παραλλαγή του δημοφιλή αλγορίθμου k-Means, με την ικανότητα να χειρίζεται και να ομαδοποιεί δισδιάστατους πίνακες, αντί διανυσμάτων. Έπειτα εφαρμόσαμε τον κλασικό k-Means στα κεντροειδή του τρισδιάστατου πίνακα και παρατηρήσαμε υψηλότερης ακρίβειας αποτελέσματα και ταχύτερους χρόνους σύγκλισης. Και στις δύο περιπτώσεις εκτελέσαμε τους αλγορίθμους ξεχωριστά στις τρεις πιθανές τομές του πίνακα: (α) κατά τον άξονα των χρωμοσωμάτων, (β) κατά τον άξονα των γονιδίων και (γ) κατά τον άξονα των δειγμάτων. Τέλος, χρησιμοποιήσαμε μια μέθοδο συσταδοποίησης υποχώρου με το όνομα δ-TRIMAX ώστε να εντοπίσουμε συστάδες και στις τρεις διαστάσεις ταυτόχρονα. Όλα τα παραπάνω υλοποιήθηκαν με χρήση της γλώσσας προγραμματισμού Python και πλήθος βιβλιοθηκών της.

Στις τελευταίες ενότητες αξιολογήσαμε τα αποτελέσματα των διαφόρων συσταδοποιήσεων που επέστρεψαν οι αλγόριθμοι με τυπικές μεθόδους και διαπιστώσαμε την ποιότητα των συστάδων σε κάθε περίπτωση.

Λέξεις κλειδιά: *μηχανική μάθηση, υπολογιστική νοημοσύνη, συσταδοποίηση, καρκίνος, γονιδιακή έκφραση, χρωμοσώματα, k-means, δ-TRIMAX, μικροσυστοιχίες DNA, τρισδιάστατος πίνακας*

Abstract

DNA microarrays are amongst the most popular experimental techniques for observing the gene expression of multiple samples simultaneously. Through the use of DNA microarrays on cancerous tissues, researchers are able to study and analyse the behaviour of the genes in various stages of the disease, with the aim to explain some of the rules that govern the process of oncogenesis.

The objective of this diploma thesis is to take advantage of the fact that every gene in a living organism belongs exclusively to one of its chromosomes. In this manner, we created a three-dimensional array by dividing the genes obtained by microarray experiments according to their respective chromosome pairs. Afterwards, we applied numerous clustering methods in order to isolate groups of genes that display a common expression profile.

More specifically, we developed a variation of the popular k-Means algorithm, which is able to handle and cluster whole arrays instead of single vectors. We also applied the classic k-Means algorithm to the centroids of the three-dimensional array and acquired higher quality results and faster convergence. In both cases, we ran the algorithms for all axes of the array: (a) the chromosome axis, (b) the gene axis and (c) the sample axis. Finally, we used a subspace clustering technique, with the name δ-TRIMAX, in order to reveal clusters in all axes simultaneously. All of the above was implemented using the Python programming language and various libraries.

In the last chapters we evaluated the obtained results with standard methods and commented on the quality of the clusters in each case.

Keywords: *machine learning, computational intelligence, clustering, cancer, gene expression, chromosomes, k-means, δ-TRIMAX, DNA microarrays, three-dimensional array.*

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή κο Δ. Κουτσούρη, διευθυντή του Εργαστηρίου Βιοϊατρικής Τεχνολογίας στο Εθνικό Μετσόβιο Πολυτεχνείο, για την ανάθεση της διπλωματικής αυτής εργασίας και την ευκαιρία που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα.

Επίσης θα ήθελα να ευχαριστήσω θερμά τον κο Γ. Λάμπρου, μέλος Ε.ΔΙ.Π. της Ιατρικής Σχολής του Ε.Κ.Π.Α. και ερευνητή του Χωρέμειου Ερευνητικού Εργαστηρίου Νοσοκομείου Παίδων «Αγία Σοφία», για τις ενδιαφέρουσες ιδέες του, αλλά και για την καθοδήγηση και την υποστήριξη που μου έδειξε σε όλη αυτή την προσπάθεια. Τον ευχαριστώ ακόμη για την εμπιστοσύνη του και για όλες τις ενδιαφέρουσες συζητήσεις που είχαμε.

Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου για την αμέριστη στήριξή τους όλα τα χρόνια των σπουδών μου και για την εμπιστοσύνη τους στις επιλογές μου.

Περιεχόμενα

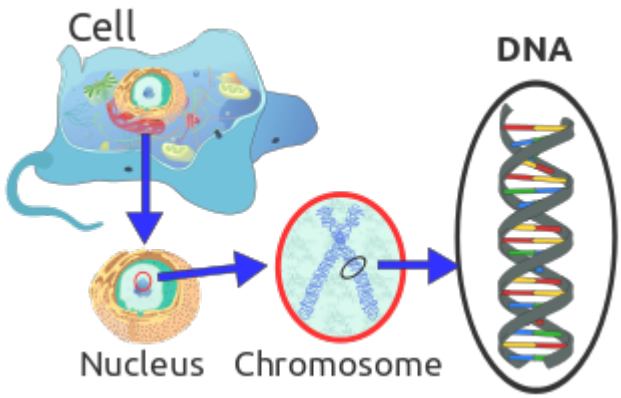
Κατάλογος Σχημάτων

Σημείωση: Σε όλη την έκταση του παρόντος κειμένου, τα διανυσματικά μεγέθη σημειώνονται με έντονη γραμματοσειρά.

Α. ΕΙΣΑΓΩΓΗ

Α.1) Βιολογικο Υποβαθρο

Το *κύτταρο* αποτελεί το μικρότερο δομικό συστατικό της έμβιας ύλης και περιλαμβάνει μια συστηματικά οργανωμένη ομάδα μορίων που βρίσκονται σε δυναμική αλληλεπίδραση μεταξύ τους. Διαθέτει μορφολογική, φυσική και χημική οργάνωση και την ικανότητα της αφομοίωσης, της ανάπτυξης και της αναπαραγωγής. Υπάρχουν οργανισμοί, όπως κάποια μικρόβια, που αποτελούνται από ένα μόνο κύτταρο (μονοκύτταροι), ενώ άλλοι οργανισμοί, όπως ο άνθρωπος, αποτελούνται από τρισεκατομμύρια κύτταρα (πολυκύτταροι). Μεγάλες ομάδες ομοειδών κυττάρων, κατά σύσταση και ορισμένη φυσιολογική λειτουργία, χαρακτηρίζονται ως ιστοί (π.χ. μυϊκός ιστός).

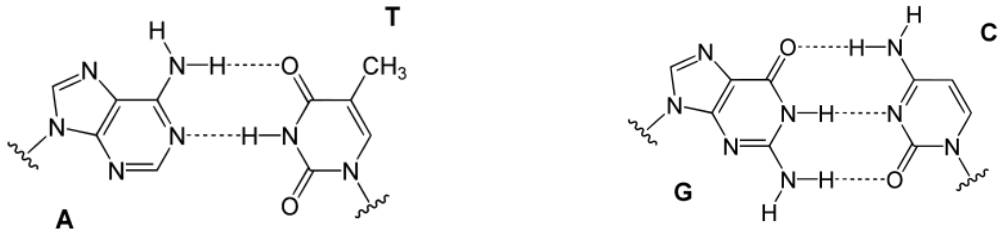


Σχήμα 1: Η τοποθεσία του DNA στο ευκαρυωτικό κύτταρο.

Στον πυρήνα των *ευκαρυωτικών* κυττάρων (δηλ. κυττάρων με σαφώς σχηματισμένο πυρήνα) υπάρχει η *χρωματίνη*, η οποία δομείται σε *χρωμοσώματα* κι εμπεριέχει τα *νουκλεϊκά οξέα* (πολυμερή). Το μονομερές του νουκλεϊκού οξέος είναι το *νουκλεοτίδιο*, το οποίο αποτελείται από: ένα μόριο σακχάρου, ένα ανιόν (τη φωσφάτη) και μια νουκλεοτιδική βάση. Κατά το σχηματισμό του νουκλεϊκού οξέος, το σάκχαρο ενός νουκλεοτιδίου ενώνεται με τη φωσφάτη του επόμενου νουκλεοτιδίου στην αλυσίδα. Επίσης τα νουκλεοτίδια του ίδιου είδους νουκλεϊκού οξέος έχουν ίδιο σάκχαρο και ίδια φωσφάτη, και διαφέρουν μόνο στη βάση.

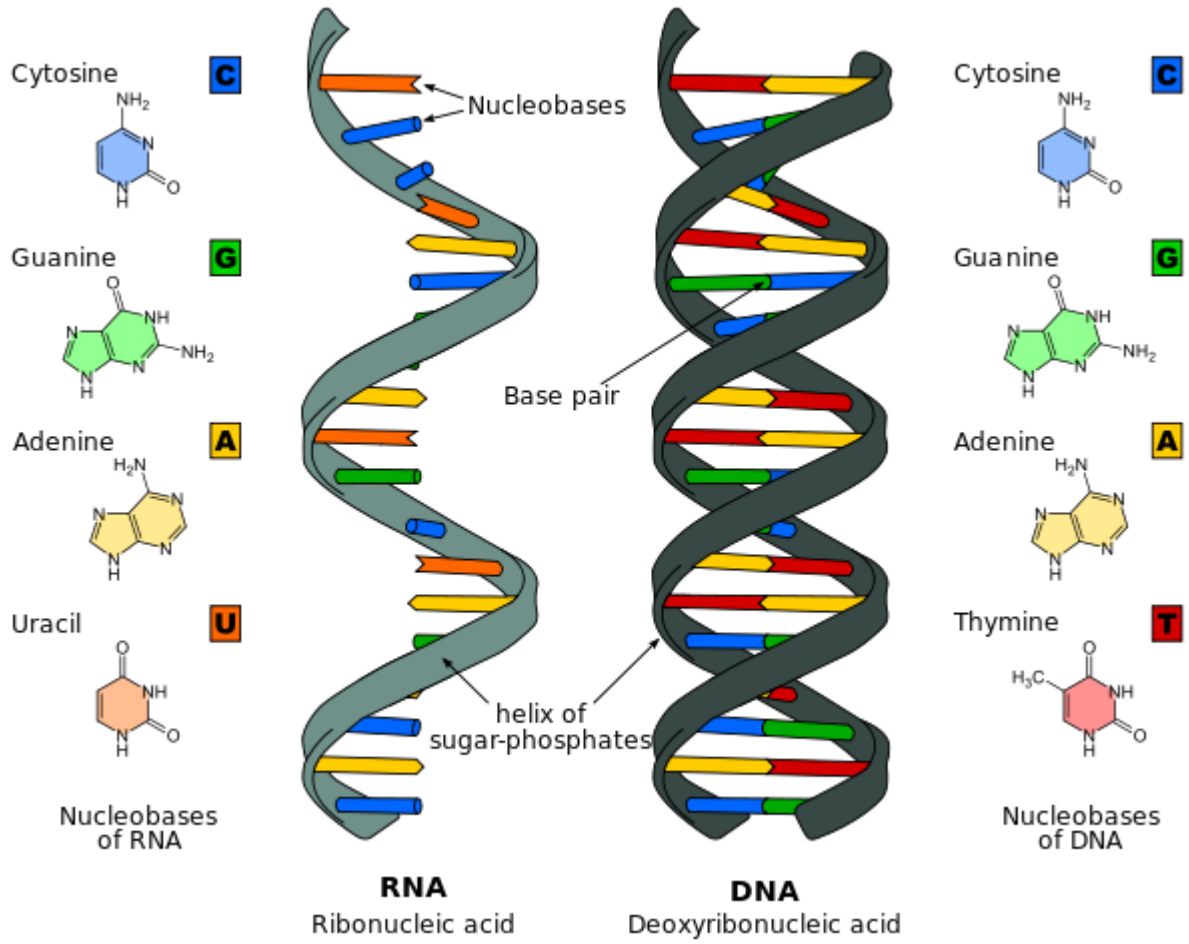
Το *δε(σ)οξυριβο(ζο)νουκλεϊ(νι)κό οξύ* (*Deoxyribonucleic acid - DNA*) είναι νουκλεϊκό οξύ που περιέχει τις γενετικές πληροφορίες που καθορίζουν τη βιολογική ανάπτυξη όλων των κυτταρικών μορφών ζωής και των περισσοτέρων ιών. Στο DNA το σάκχαρο είναι η δεοξυριβόζη και οι βάσεις είναι: θυμίνη (T), αδενίνη (A), κυτοσίνη (C) και γουανίνη (G). Σύμφωνα με το μοντέλο των J. Watson και F. Crick (1953) [1], ένα μόριο DNA αποτελείται από δύο

αλυσίδες οι οποίες είναι αντιπαράλληλες και περιελίσσονται μεταξύ τους σχηματίζοντας μια δεξιόστροφη διπλή έλικα. Η ένωση μεταξύ των δύο αλυσίδων υλοποιείται με δεσμούς υδρογόνου και τα ζεύγη των βάσεων που ενώνονται είναι καθορισμένα: η αδενίνη με τη θυμίνη και η γουανίνη με την κυτοσίνη (Σχήμα 2). Έτσι, οι αλυσίδες που προκύπτουν είναι συμπληρωματικές.



(α) *(β)*

Σχήμα 2: (α) Ζεύξη αδενίνης - θυμίνης (β) Ζεύξη γουανίνης - κυτοσίνης.

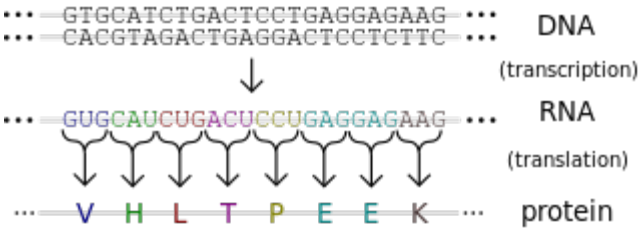


Σχήμα 3: Τμήματα αλυσίδων DNA και RNA.

Στη χρωματίνη υπάρχει επίσης κι ένα άλλο νουκλεϊκό οξύ, το *ριβο(ζο)νουκλεϊκό οξύ* (*Ribonucleic acid - RNA*), του οποίου το σάκχαρο είναι η ριβόζη και βάσεις: αδενίνη (A), ουρακίλη (U), κυτοσίνη (C) και γουανίνη (G). Κύρια διαφορά του RNA από το DNA είναι ότι το μόριό του είναι μονόκλωνο έναντι του δίκλωνου του DNA, αποτελείται δηλαδή από μια μόνο αλυσίδα. Τέλος, τα δύο αυτά νουκλεϊκά οξέα σε συνδυασμό αποτελούν το γενετικό υλικό των οργανισμών.

Γονίδια

Το DNA αποτελείται από συγκεκριμένες αλληλουχίες βάσεων, τα *γονίδια*, οι οποίες περιέχουν τις απαραίτητες πληροφορίες για τη σύνθεση πρωτεϊνών ή μορίων RNA (π.χ. tRNA, sRNA). Με ελάχιστες εξαιρέσεις, κάθε κύτταρο του ανθρώπινου οργανισμού περιέχει αντίγραφα όλων των γονιδίων του (~ 20,000). Κάθε ένα από αυτά τα γονίδια είναι ενεργό ή ανενεργό, ανάλογα με το είδος του κυττάρου στο οποίο ανήκει. Για παράδειγμα, σε ένα μυϊκό κύτταρο τα γονίδια που παράγουν τις μυϊκές πρωτεΐνες (π.χ. ακτίνη, μυοσίνη) είναι ενεργά, ενώ εκείνα που συμβάλλουν στην παραγωγή ινσουλίνης είναι ανενεργά. Η διαδικασία εκείνη που προκαλεί τη μεταφορά των κωδικοποιημένων πληροφοριών του γονιδίου στο λειτουργικό του προϊόν ονομάζεται *γονιδιακή έκφραση* και περιγράφεται από το Κεντρικό Δόγμα της Μοριακής Βιολογίας [2].

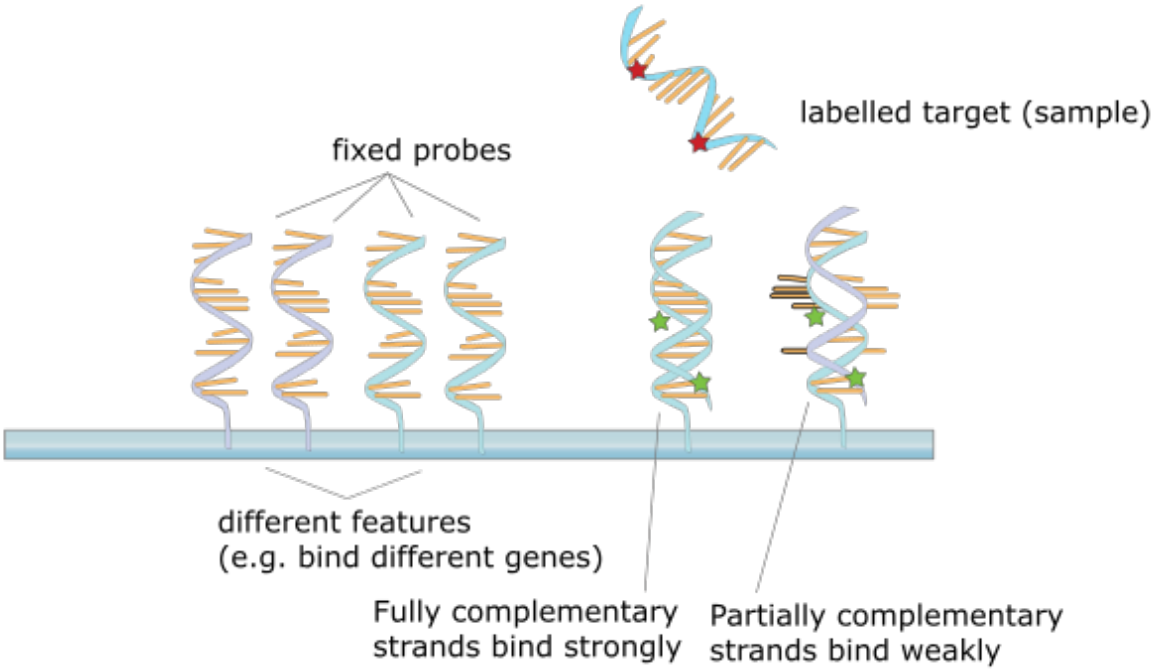


Σχήμα 4: Η διαδικασία της γονιδιακής έκφρασης.

Το πρώτο στάδιο της γονιδιακής έκφρασης είναι η *μεταγραφή* (*transcription*), κατά την οποία η διπλή έλικα του DNA ανοίγει προσωρινά στο σημείο του γονιδίου κι έπειτα με τη βοήθεια ενζύμων δημιουργείται μια συμπληρωματική αλυσίδα RNA από την γενετική πληροφορία που βρίσκεται εκεί. Στην περίπτωση που το RNA που δημιουργείται είναι αγγελιοφόρο RNA (mRNA), τότε μόλις ολοκληρωθεί η μεταγραφή αυτό ταξιδεύει στα ριβοσώματα (κυτταρικά σωματίδια) για το δεύτερο στάδιο, τη *μετάφραση* (*translation*), κατά το οποίο κωδικοποιούνται οι πρωτεΐνες.

Μικροσυστοιχίες DNA

Η *μικροσυστοιχία DNA* (*DNA microarray* ή *DNA chip*) είναι μία διάταξη από ακινητοποιημένα νουκλεϊκά οξέα (probes) που αντιπροσωπεύουν μοναδικά γονίδια πάνω σε μία στερεή επιφάνεια από γυαλί ή πυρίτιο. Η κύρια λειτουργία της βασίζεται στην εκμετάλλευση της αρχής της συμπληρωματικότητας ανάμεσα στα νουκλεϊκά οξέα, με στόχο τη μέτρηση και ανίχνευση της ποσότητας mRNA στο εκάστοτε βιολογικό δείγμα, και άρα την επισκόπηση της γονιδιακής έκφρασης. Πρόκειται για μια τεχνολογία που εξελίσσεται ραγδαία τα τελευταία χρόνια και παρέχει τη δυνατότητα στους ερευνητές να εξετάζουν ταυτόχρονα την έκφραση χιλιάδων γονιδίων σε μια δεδομένη χρονική στιγμή.



Σχήμα 5: Η διαδικασία της υβριδοποίησης.

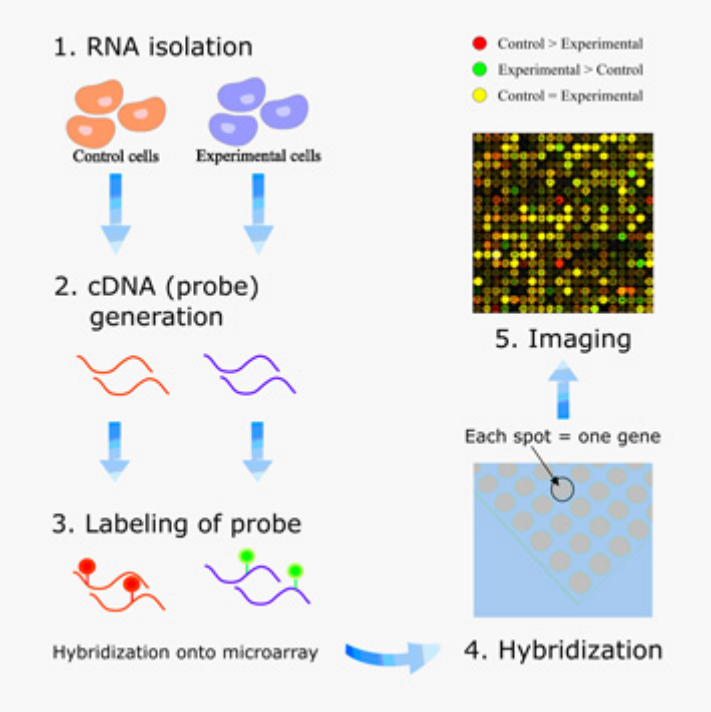
Αν και πλέον χρησιμοποιούνται ποικίλες μέθοδοι για την κατασκευή των μικροσυστοιχιών DNA, η φιλοσοφία που διέπει αυτό το είδος πειραμάτων είναι θεμελιωδώς η ίδια. Τα βήματα της διαδικασίας περιγράφονται παρακάτω [3]:

- Αρχικά, συλλέγεται ένας αριθμός δειγμάτων ελέγχου (control samples) τα οποία αποτελούν το σημείο αναφοράς για το συγκεκριμένο πείραμα.
- Έπειτα συλλέγονται τα δείγματα προς μελέτη (samples).
- Όλα τα παραπάνω δείγματα αναμειγνύονται με κατάλληλα διαλυτικά μέσα ώστε να

γίνει διαχωρισμός του RNA από τα υπόλοιπα συστατικά των κυττάρων (DNA, πρωτεΐνες, κλπ).

- Τα μείγματα εισάγονται στον αναδεύτη (Vortex mixer) για να διαλυθεί ο ιστός, κι έπειτα στον φυγοκεντρητή (micro-centrifuge) ώστε να διαχωριστεί και να απομονωθεί το RNA.
- Στη συνέχεια γίνονται διαδοχικές πλύσεις των δειγμάτων με ειδικές ουσίες ώστε να διατηρηθεί μόνο το mRNA και να απομακρυνθούν τα υπόλοιπα είδη RNA (tRNA, rRNA).
- Έπειτα εισάγονται ειδικές φθορίζουσες ουσίες στα δείγματα (συνήθως πράσινο χρώμα για τα controls και κόκκινο για τα samples), διαδικασία η οποία έχει ως αποτέλεσμα την παραγωγή χρωματισμένου συμπληρωματικού DNA (cDNA) μέσω της αντίστροφης μεταγραφής.
- Τα δείγματά μας εισάγονται στη μικροσυστοιχία DNA, η οποία σε κάθε κελί περιέχει ένα πλήθος πανομοιότυπων μονόκλωνων μορίων DNA, το καθένα από τα οποία αντιπροσωπεύει ένα συγκεκριμένο γονίδιο. Έτσι λαμβάνει χώρα η υβριδοποίηση του cDNA (Σχήμα 5), διαδικασία κατά την οποία τμήματα του cDNA των δειγμάτων μας προσκολλώνται στα συμπληρωματικά μόρια DNA που βρίσκονται στη μικροσυ-στοιχία.
- Γίνεται πλύση της μικροσυστοιχίας για την απομάκρυνση των μορίων cDNA που δεν υβριδοποιήθηκαν.
- Στο τελικό βήμα η μικροσυστοιχία εισάγεται σε ειδικό σαρωτή (microarray scanner), όπου καθίσταται ορατή η διαφορά έκφρασης ανάμεσα στα δείγματα και γίνεται επεξεργασία της εικόνας για περαιτέρω υπολογιστική ανάλυση.

Η τελική εικόνα που λαμβάνουμε από το πείραμα παρέχει πολλαπλές πληροφορίες. Τα κελιά εκείνα που έχουν πιο έντονη πράσινη απόχρωση μαρτυρούν ότι τα γονίδια που περιέχουν εκφράζονται κυρίως στα δείγματα ελέγχου (*up-regulation*). Αντίστοιχα, τα κελιά με πιο έντονη κόκκινη απόχρωση δείχνουν ότι τα γονίδια που περιέχουν εκφράζονται κυρίως στα προς μελέτη δείγματα (*down-regulation*). Τέλος, τα κελιά με κίτρινη απόχρωση δείχνουν ότι τα συγκεκριμένα γονίδια εκφράζονται εξίσου σε όλα τα δείγματα, ενώ τα μη χρωματισμένα κελιά αποδεικνύουν την απουσία έκφρασης αυτών των γονιδίων σε όλα τα δείγματα.



Σχήμα 6: Η πειραματική διαδικασία των μικροσυστοιχιών DNA.

Είναι σύνηθες κατά την προεπεξεργασία των δεδομένων για χρήση σε υπολογιστικές μεθόδους να εφαρμόζεται ο τύπος:

$$c_{i,j} = \log_2 \frac{R_{i,j}}{G_{i,j}} \qquad \textbf{(1)}$$

όπου $c_{i,j}$ είναι η τελική τιμή της γονιδιακής έκφρασης στο κελί (i, j) της μικροσυστοιχίας

$R_{i,j}$ είναι η τιμή φθορισμού του κόκκινου χρώματος (sample)

$G_{i,j}$ είναι η τιμή φθορισμού του πράσινου χρώματος (control)

Με τον τρόπο αυτό γίνεται κανονικοποίηση (normalization) των δεδομένων ώστε να διευκολυνθεί η μετέπειτα επεξεργασία και ανάλυσή τους.

Τα βασικά πλεονεκτήματα αυτής της πειραματικής μεθόδου είναι το χαμηλό σχετικά κόστος και η δυνατότητα ταυτόχρονης μελέτης χιλιάδων γονιδίων σε πολλά δείγματα. Από την άλλη, όμως, πλευρά, τα πειράματα που βασίζονται στις DNA μικροσυστοιχίες συχνά εμφανίζουν δυσκολία στην αναπαραγωγή ή/και ανακρίβεια αποτελεσμάτων εξαιτίας προβλημάτων κατά την υβριδοποίηση (για παράδειγμα είναι σύνηθες το φαινόμενο της υβριδοποίησης τμημάτων του cDNA σε probes με τα οποία μοιράζονται πολλά συμπληρωματικά νουκλεϊκά οξέα αλλά όχι όλα, κάτι που στη βιβλιογραφία αναφέρεται ως cross-hybridisation).

Ακόμη, ένας περιορισμός που θέτει η συγκεκριμένη μέθοδος είναι το γεγονός ότι ενώ διαπιστώνουμε τα επίπεδα έκφρασης των εκάστοτε γονιδίων, δεν αντλούμε καμία πληροφορία σχετικά με την παραγωγή ή μη πρωτεϊνών. Για παράδειγμα, υπάρχει πιθανότητα κάποιο γονίδιο να εκφράζεται τόσο στο control όσο και στο προς μελέτη δείγμα, αλλά τελικά λόγω κάποιας μετάλλαξης να παράγεται πρωτεΐνη μόνο στο control και όχι στο δείγμα. Τέλος, συχνά συναντώνται δυσκολίες στην ανταλλαγή και το διαμοιρασμό των δεδομένων εξαιτίας της απουσίας προτυποποίησης (standardisation). Τα τελευταία χρόνια γίνονται προσπάθειες να αντιμετωπιστεί το πρόβλημα αυτό με τη θεσμοθέτηση προτύπων και κανόνων, όπως το MIAME [4].

A.2) Μηχανικη Μαθηση

Η *Μηχανική Μάθηση* (*Machine Learning*) είναι ένας τομέας της Τεχνητής Νοημοσύνης που αφορά αλγορίθμους και μεθόδους που επιτρέπουν στους υπολογιστές να “μαθαίνουν” και να προσαρμόζουν κατάλληλα τη λειτουργία τους χωρίς να έχουν προγραμματιστεί ρητώς για το σκοπό αυτό. Αν και ο όρος χρησιμοποιούταν ήδη από τη δεκαετία του '60, ο πρώτος επίσημος ορισμός δόθηκε από τον Tom M. Mitchell το 1997 [5]:

> *«Ένα πρόγραμμα υπολογιστή λέγεται ότι μαθαίνει από εμπειρία E ως προς μια κλάση εργασιών T και ένα μέτρο επίδοσης P, αν η επίδοσή του σε εργασίες της κλάσης T, όπως αποτιμάται από το μέτρο P, βελτιώνεται με την εμπειρία E.»*

Και το πρωτότυπο:

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

Με άλλα λόγια, ο τομέας της Μηχανικής Μάθησης μελετά την ανάπτυξη προγραμμάτων που έχουν τη δυνατότητα να εντοπίζουν μη εμφανή μοτίβα σε ένα σύνολο δεδομένων, να εξάγουν συμπεράσματα από αυτά και να πραγματοποιούν κατάλληλες προβλέψεις.

Σε γενικές γραμμές, οι αλγόριθμοι Μηχανικής Μάθησης διακρίνονται σε τέσσερις βασικές

κατηγορίες:

1. *Επιβλεπόμενη ή επιτηρούμενη μάθηση* (*Supervised learning*): Ο αλγόριθμος προσπαθεί να βρει κατάλληλη συνάρτηση για να απεικονίσει δεδομένες εισόδους σε γνωστές, επιθυμητές εξόδους, με απώτερο στόχο τη γενίκευση της συνάρτησης αυτής και για εισόδους με άγνωστη έξοδο.
2. *Μη επιβλεπόμενη ή μη επιτηρούμενη μάθηση* (*Unsupervised learning*): Ο αλγόριθμος επιχειρεί να εντοπίσει μοτίβα και συσχετισμούς σε δεδομένες εισόδους χωρίς να γνωρίζει a priori την επιθυμητή έξοδο.
3. *Ενισχυτική μάθηση* (*Reinforcement learning*): Ο αλγόριθμος αλληλεπιδρά με ένα δυναμικό περιβάλλον για την επίτευξη κάποιου στόχου (π.χ. την οδήγηση οχήματος ή τη νίκη σε επιτραπέζιο παιχνίδι) μαθαίνοντας έτσι μια στρατηγική ενεργειών για μια δεδομένη κατάσταση.
4. *Ημι-επιβλεπόμενη ή ημι-επιτηρούμενη μάθηση* (*Semi-supervised learning*): Ο αλγόριθμος δέχεται ένα σύνολο εισόδων, στο οποίο άλλες έχουν γνωστή κι άλλες άγνωστη έξοδο, και τις χρησιμοποιεί συνδυαστικά για την επίτευξη του στόχου.

Αλγόριθμοι συσταδοποίησης (Clustering algorithms)

Στη συγκεκριμένη εργασία θα ασχοληθούμε αποκλειστικά με *αλγορίθμους συσταδοποίησης* ή *ομαδοποίησης* (*clustering algorithms*), οι οποίοι ανήκουν στην κατηγορία της μη επιβλεπόμενης μάθησης και στοχεύουν στο διαχωρισμό των δεδομένων σε λογικές ομάδες βάσει κάποιου ορισμένου κριτηρίου. Σύμφωνα με τον ορισμό που δίνεται στο [6]:

> *«Η συσταδοποίηση είναι μια μορφή μη επιβλεπόμενης μάθησης, δια της οποίας ένα σύνολο παρατηρήσεων (δηλ. σημείων δεδομένων) διαμερίζεται σε φυσικές ομαδοποιήσεις ή συστάδες προτύπων με τέτοιο τρόπο ώστε το μέτρο ομοιότητας μεταξύ οποιουδήποτε ζεύγους παρατηρήσεων αντιστοιχίζεται σε κάθε συστάδα να ελαχιστοποιεί μια καθορισμένη συνάρτηση κόστους.»*

Οι *συστάδες* ή *ομάδες* (*clusters*) μπορούν γενικά να οριστούν με δύο τρόπους. Έστω ότι το σύνολο των δεδομένων μας αποτελείται από N παρατηρήσεις: $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N\}$.

- *Αυστηρή συσταδοποίηση* (H*ard ή crisp clustering*), όπου θεωρούμε ότι κάθε διάνυσμα

εισόδου $\boldsymbol{x_i}$ ανήκει αποκλειστικά σε μία ομάδα. Ορίζουμε ως *m*-συσταδοποίηση *R* τη διαμέριση του *X* σε *m* ομάδες $C_1, C_2, ..., C_m$ τέτοιες ώστε να πληρούν τις παρακάτω προϋποθέσεις:

i. $C_i \neq \emptyset, \ για \ \ i=1,...,m$

ii. $\cup_{i=1}^{m} C_i = X$

iii. $C_i \cap C_j = \emptyset, \ για \ \ i \neq j \ \ και \ \ i, j=1,...,m$

- *Ασαφής συσταδοποίηση* (*Fuzzy clustering*), όπου κάθε διάνυσμα εισόδου $\boldsymbol{x_i}$ μπορεί να ανήκει ταυτόχρονα σε διαφορετικές ομάδες με διαφορετικό βαθμό βεβαιότητας. Σε αυτή την περίπτωση, η *m*-συσταδοποίηση του *X* χαρακτηρίζεται από *m* συναρτήσεις u_j που ορίζονται ως:

  $u_j: X \rightarrow [0, 1]$, για $j = 1, ..., m$

  και ισχύουν:

  $$\sum_{j=1}^{m} u_j(\boldsymbol{x_i}) = 1, \ για \ \ i=1,2,...,N \qquad \textbf{(2)}$$

  και

  $$0 < \sum_{i=1}^{N} u_j(\boldsymbol{x_i}) < N, \ για \ \ j=1,2,...,m \qquad \textbf{(3)}$$

  Οι συναρτήσεις αυτές ονομάζονται *συναρτήσεις συμμετοχής* και ποσοτικοποιούν τη βεβαιότητα που έχουμε για το αν κάποιο διάνυσμα εισόδου *i* ανήκει στην ομάδα *j*. Τιμές των συναρτήσεων κοντά στη μονάδα (1) υποδηλώνουν μεγάλο βαθμό συμμετοχής του διανύσματος στην αντίστοιχη ομάδα, ενώ τιμές κοντά στο μηδέν (0) υποδηλώνουν απουσία συμμετοχής.

Τα βασικά βήματα για την εξαγωγή συσταδοποίησης από ένα σύνολο δεδομένων περιγράφονται παρακάτω:

- *Επιλογή χαρακτηριστικών γνωρισμάτων (features)*: Κάθε σημείο εισόδου $\boldsymbol{x_i}$ αποτελεί ένα *διάνυσμα γνωρισμάτων* (*feature vector*). Τα γνωρίσματα αυτά είναι μετρήσιμες εγγενείς ιδιότητες της εισόδου, βάσει των οποίων γίνεται ο διαχωρισμός σε ομάδες. Για παράδειγμα, αν θέλαμε να κατηγοριοποιήσουμε το σύνολο των ζώων, θα μπορούσαμε να επιλέξουμε ως γνώρισμα το φυσικό τους περιβάλλον κι έτσι θα

προέκυπταν οι ομάδες: ζώα ξηράς, υδρόβια, κλπ. Επιθυμούμε λοιπόν να επιλέξουμε το μικρότερο δυνατό σύνολο γνωρισμάτων ώστε να κωδικοποιείται όσο το δυνατόν περισσότερη πληροφορία και να επιτυγχάνεται η μεγαλύτερη δυνατή ομοιογένεια σε κάθε ομάδα.

- *Επιλογή μέτρου εγγύτητας ή γειτνίασης (proximity measure)*: Το μέτρο εγγύτητας εκφράζει το βαθμό "ομοιότητας" ή "εγγύτητας" μεταξύ δύο αντικειμένων. Στην περίπτωση των αλγορίθμων συσταδοποίησης επιθυμούμε να μετρήσουμε την ομοιότητα ανάμεσα σε δύο διανύσματα γνωρισμάτων ώστε να αποφασίσουμε την ένταξή τους ή μη σε κάποια ομάδα.

- *Επιλογή κριτηρίου συσταδοποίησης (clustering criterion)*: Το κριτήριο αυτό εξαρτάται από τον τύπο των συστάδων που αναμένουμε να βρει ο αλγόριθμος (Σχήμα 7) και εκφράζεται με τον ορισμό κατάλληλης συνάρτησης κόστους ή κάποιου άλλου τύπου κανόνων.

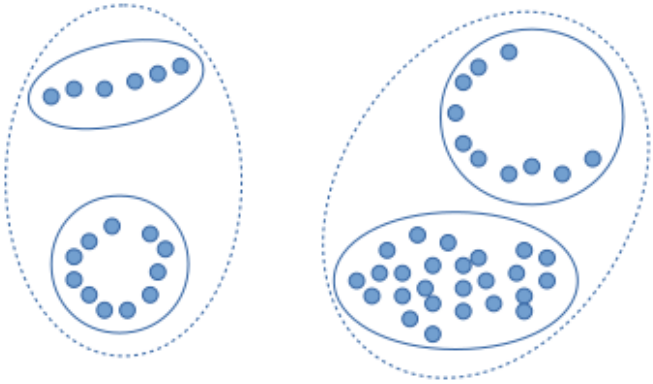
(α) *(β)* *(γ)*

Σχήμα 7 : (α) συμπαγείς συστάδες. (β) επιμήκεις συστάδες. (γ) σφαιρικές και ελλειψοειδείς συστάδες.

- *Επιλογή αλγορίθμου συσταδοποίησης (clustering algorithm)*: Σε αυτό το στάδιο γίνεται η επιλογή του κατάλληλου αλγορίθμου που θα οδηγήσει σε μια καλή συσταδοποίηση για ένα σύνολο δεδομένων.

- *Επικύρωση αποτελεσμάτων (validation)*: Αξιολογούνται τα αποτελέσματα του αλγορίθμου συσταδοποίησης σύμφωνα με κατάλληλα κριτήρια ορθότητας.

- *Ερμηνεία αποτελεσμάτων*: Αποτελεί το τελευταίο στάδιο της διαδικασίας συσταδοποίησης, όπου εμπειρογνώμονες του πεδίου εφαρμογής καλούνται να εξάγουν συμπεράσματα από τις παραχθείσες συστάδες, συνδυάζοντας κι άλλα στοιχεία και πειραματικές αναλύσεις.



Σχήμα 8: Υποκειμενικότητα συσταδοποίησης. Ένας αλγόριθμος ανακαλύπτει 2 συστάδες (διακεκομμένη γραμμή), ενώ ένας άλλος ανακαλύπτει 4 (συνεχής γραμμή).

Οι αλγόριθμοι συσταδοποίησης μπορούν με τη σειρά τους να διαχωριστούν σε μικρότερες κατηγορίες, ανάλογα με τον τρόπο λειτουργίας τους και τα κριτήρια που χρησιμοποιούν [7]:

- ***Ακολουθιακοί αλγόριθμοι* (*Sequential algorithms*)**: Εξετάζουν διαδοχικά τα δεδομένα και παράγουν μια μοναδική συσταδοποίηση. Συνήθως η σειρά με την οποία παρουσιάζονται τα δεδομένα στον αλγόριθμο επηρεάζουν την ποιότητα του τελικού αποτελέσματος.

- ***Ιεραρχικοί αλγόριθμοι* (*Hierarchical algorithms*)**: Παράγουν μια ιεραρχία συστάδων, όπου οι συστάδες κάθε επιπέδου προκύπτουν από το προηγούμενο είτε με διάσπασή τους (*διαιρετικοί ιεραρχικοί αλγόριθμοι, divisive hierarchical algorithms*) είτε με συνένωσή τους (*συσσωρευτικοί ιεραρχικοί αλγόριθμοι, agglomerative hierarchical algorithms*).

- ***Αλγόριθμοι βασιζόμενοι στη βελτιστοποίηση συνάρτησης κόστους (Algorithms based on cost function optimization)***: Οι αλγόριθμοι αυτοί ποσοτικοποιούν την ποιότητα της συσταδοποίησης μέσω μιας συνάρτησης κόστους J και τερματίζουν όταν επιτευχθεί κάποιο τοπικό της ακρότατο. Συνήθως ο αριθμός των συστάδων m ορίζεται στην είσοδο του αλγορίθμου και παραμένει σταθερός καθ' όλη τη διάρκεια εκτέλεσής του.

  Η κατηγορία αυτή μπορεί να διαιρεθεί σε μικρότερες, ανάλογα με το αν κάθε στοιχείο

εισόδου $\boldsymbol{x_i}$ ανήκει αποκλειστικά σε μια συστάδα ή σε περισσότερες ταυτόχρονα.

- ***Αλγόριθμοι επέκτασης και οριοθέτησης (Branch and bound algorithms)***: Έχουν τη δυνατότητα να ανακαλύπτουν την ολική βέλτιστη συσταδοποίηση με τη βοήθεια κατάλληλων παραμέτρων, χωρίς να εξετάζουν όλες τις πιθανές συσταδοποιήσεις. Το μειονέκτημά τους είναι η τεράστια υπολογιστική πολυπλοκότητα.

- ***Γενετικοί αλγόριθμοι (Genetic algorithms)***: Ξεκινούν από έναν αρχικό πληθυσμό συστάδων και σε κάθε βήμα παράγουν νέες, καλύτερες συστάδες από τις προηγούμενες προσομοιώνοντας φυσικές γενετικές διαδικασίες.

- ***Αλγόριθμοι αναζήτησης κοιλάδας (Valley-seeking algorithms)***: Αυτοί οι αλγόριθμοι μεταχειρίζονται τα διανύσματα γνωρισμάτων ως στιγμιότυπα μιας (πολυδιάστατης) τυχαίας μεταβλητής *X*. Υποθέτουν ότι οι περιοχές του *X* όπου συσσωρεύεται μεγάλος αριθμός διανυσμάτων αντιπροσωπεύουν περιοχές υψηλών τιμών στην κατανομή πυκνότητας πιθανότητας του *X* κι επομένως αποτελούν υποψήφιες συστάδες.

- ***Αλγόριθμοι ανταγωνιστικής μάθησης (Competitive learning algorithms)***: Οι αλγόριθμοι που ανήκουν σε αυτή την κατηγορία δεν κάνουν χρήση συναρτήσεων κόστους, αλλά παράγουν πολλές διαφορετικές συσταδοποιήσεις και εν τέλει συγκλίνουν στη βέλτιστη βάσει μιας μετρικής αποστάσεων.

- ***Αλγόριθμοι βασιζόμενοι σε τεχνικές μορφολογικού μετασχηματισμού (Algorithms based on morphological transformation techniques)***: Κάνουν χρήση μορφολογικών μετασχηματισμών ώστε να επιτύχουν καλύτερο διαχωρισμό των συστάδων.

- ***Αλγόριθμοι βασιζόμενοι στην πυκνότητα (Density-based algorithms)***: Θεωρούν τις συστάδες ως περιοχές του χώρου δεδομένων με ιδιαίτερα μεγάλη πυκνότητα. Υπάρχουν διάφορες μέθοδοι ορισμού της πυκνότητας κι έτσι προκύπτει πληθώρα διαφορετικών αλγορίθμων σε αυτή την κατηγορία. Βασικό τους πλεονέκτημα είναι ότι δε χρειάζεται να εξετάσουν τα δεδομένα πολλές φορές, κι έτσι ενδείκνυνται για τη συσταδοποίηση μεγάλου όγκου δεδομένων.

- ***Αλγόριθμοι συσταδοποίησης υποχώρου (Subspace clustering algorithms)***: Αντιμετωπίζουν τα προβλήματα που προκύπτουν από τη μεγάλη διαστατικότητα των δεδομένων, ειδικά σε περιπτώσεις που ο χώρος των γνωρισμάτων μπορεί να είναι μερικών εκατοντάδων ή χιλιάδων διαστάσεων.

- ***Αλγόριθμοι πυρήνα (Kernel-based algorithms)***: Κάνουν χρήση συναρτήσεων πυρήνα ώστε να απεικονίσουν τον αρχικό χώρο γνωρισμάτων σε έναν χώρο μεγαλύτερων διαστάσεων και να υπολογίσουν τις όποιες αποστάσεις μέσω εσωτερικών γινομένων, κι όχι άμεσα, αποφεύγοντας έτσι επίπονους υπολογισμούς (“kernel trick”).

Στην παρούσα εργασία συγκεκριμένα θα μας απασχολήσουν δύο αλγόριθμοι συσταδοποίησης: ο k-Means που βασίζεται σε βελτιστοποίηση συνάρτησης κόστους, και ο αλγόριθμος συσταδοποίησης υποχώρου δ-Biclustering. Στις επόμενες ενότητες θα παρουσιάσουμε τη βασική λειτουργία αυτών των μεθόδων, ενώ στα επόμενα κεφάλαια θα αναλύσουμε διεξοδικότερα τις τροποποιήσεις που προτείνουμε για περαιτέρω επέκτασή τους σε δεδομένα τριών διαστάσεων.

Ο αλγόριθμος k-Means (ή c-Means ή ISODATA)

Ο αλγόριθμος k-Means (ή αλλιώς c-Means ή ISODATA), προτάθηκε αρχικά από τον Stuart Lloyd το 1957 [8] ως μια τεχνική για παλμοκωδική διαμόρφωση σήματος (γι' αυτό και συχνά αποκαλείται επίσης «αλγόριθμος του Lloyd»). Μια δεκαετία αργότερα, ο ίδιος αλγόριθμος παρουσιάστηκε παράλληλα σε δύο διαφορετικές δημοσιεύσεις: των G. H. Ball και D. J. Hall που τον ονόμασαν ISODATA [9] και του James MacQueen που του έδωσε το όνομα k-Means [10].

Αποτελεί έναν από τους πιο δημοφιλείς αλγορίθμους συσταδοποίησης, αφού χάρη στην απλότητα και την αποτελεσματικότητά του ενδείκνυται για το χειρισμό μεγάλου όγκου δεδομένων, ενώ παράλληλα βρίσκει εφαρμογή σε πάρα πολλά πεδία. Ήδη στη διεθνή βιβλιογραφία υπάρχουν ποικίλες δημοσιεύσεις και προτάσεις για επέκταση των δυνατοτήτων του με χρήση διαφόρων προχωρημένων τεχνικών.

Ο αλγόριθμος k-Means ανήκει στην κατηγορία των μεθόδων που βασίζονται στη βελτιστοποίηση μιας καλά ορισμένης συνάρτησης κόστους και ο απώτερος στόχος του είναι η εύρεση συμπαγών συστάδων μέσα στο σύνολο δεδομένων. Κάνει χρήση αυστηρής συ-

σταδοποίησης, που σημαίνει ότι για κάθε χρονική στιγμή κάθε σημείο της εισόδου μπορεί να ανήκει το πολύ σε μία συστάδα. Πιο αναλυτικά, έστω ότι έχουμε επιλέξει να μελετήσουμε M γνωρίσματα (features) για ένα συγκεκριμένο πρόβλημα, κι έτσι έχει προκύψει το σύνολο δεδομένων $X = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_N}\}$, όπου κάθε $\boldsymbol{x_i}$ αποτελεί ένα διάνυσμα γνωρισμάτων M διαστάσεων. Τροφοδοτούμε τα δεδομένα στον αλγόριθμο, μαζί με ένα πλήθος συστάδων k που έχουμε επιλέξει. Έπειτα, ο k-Means επιλέγει τυχαία τα κέντρα $\boldsymbol{\theta_j}$, $j = 1, 2, ..., k$, των k συστάδων, και σε κάθε επανάληψη: α) ορίζει κάθε πρότυπο $\boldsymbol{x_i}$ στη συστάδα με το κοντινότερο σε αυτό κέντρο και β) επαναϋπολογίζει κάθε κέντρο $\boldsymbol{\theta_j}$ ως το μέσο όρο των διανυσμάτων $\boldsymbol{x_i}$ που ανήκουν στη j-στή συστάδα. Ο αλγόριθμος τερματίζει όταν δεν υπάρχει πια καμία αλλαγή στα κέντρα $\boldsymbol{\theta_j}$.

Παρατηρούμε λοιπόν ότι ο k-Means χρησιμοποιεί το μέσο διάνυσμα $\boldsymbol{\theta_j}$ ως αντιπρόσωπο κάθε συστάδας, μέσω της σχέσης:

$$\boldsymbol{\theta_j} = \frac{1}{|C_j|}\sum_{i \in C_j} \boldsymbol{x_i} \qquad \textbf{(4)}$$

όπου $|C_j|$ το πλήθος των στοιχείων της j-στής συστάδας

Παράλληλα, η συνάρτηση κόστους J που επιχειρεί να βελτιστοποιήσει ο αλγόριθμος είναι μια μετρική απόστασης (τυπικά η Ευκλείδεια απόσταση) ανάμεσα στο εκάστοτε διάνυσμα $\boldsymbol{x_i}$ και τον αντιπρόσωπο $\boldsymbol{\theta_j}$ της συστάδας στην οποία τελικά θα τοποθετηθεί, δηλαδή:

$$J(\theta) = \sum_{j=1}^{k}\sum_{i \in C_j} \|\boldsymbol{x_i} - \boldsymbol{\theta_j}\|^2 \qquad \textbf{(5)}$$

και η ανάθεση του διανύσματος x_i στη συστάδα C_j γίνεται μέσω του κανόνα:

$$C_j = \{\boldsymbol{x_i}: \|\boldsymbol{x_i} - \boldsymbol{\theta_j}\|^2 \leqslant \|\boldsymbol{x_i} - \boldsymbol{\theta_p}\|^2,\ \forall j, p,\ j \neq p,\ 1 \leqslant p \leqslant k,\ 1 \leqslant j \leqslant k\} \qquad \textbf{(6)}$$

Η ακριβής λειτουργία του αλγορίθμου περιγράφεται στον παρακάτω ψευδοκώδικα:

Αλγόριθμος k-Means

Είσοδος:
    Δεδομένα εισόδου $x_1, x_2, ..., x_N$
    Πλήθος συστάδων k
Έξοδος:
    Συστάδες $C_1, C_2, ..., C_k$

```
Μέθοδος:
C₁ = C₂ = ... = Cₖ = Ø
Αρχικοποίησε τα κέντρα θ₁, θ₂, ..., θₖ σε τυχαίες τιμές
Επανάλαβε {
    Για i = 1 έως N {
        Υπολόγισε τον κοντινότερο αντιπρόσωπο θⱼ του xᵢ
        Cⱼ = Cⱼ U xᵢ
    }
    Για j = 1 έως k {
        Όρισε το θⱼ ως το μέσο όρο των διανυσμάτων xᵢ που
        ανήκουν στη συστάδα j
    }
} Μέχρι να μην υπάρξει καμία αλλαγή στα κέντρα θⱼ
```

Πολυπλοκότητα: O(Nkq), όπου N το πλήθος των στοιχείων εισόδου, k το πλήθος των συστάδων και q το πλήθος των επαναλήψεων που απαιτείται για σύγκλιση του αλγορίθμου.

Ένα από τα βασικά χαρακτηριστικά του k-Means είναι ότι δεν συγκλίνει απαραίτητα στο ολικό ελάχιστο της συνάρτησης κόστους, αλλά τις περισσότερες φορές εγκλωβίζεται σε τοπικά ελάχιστα. Το γεγονός αυτό μπορεί εν μέρει να αντιμετωπιστεί με διαδοχικές εκτελέσεις του αλγορίθμου και προβολή τελικά του καλύτερου αποτελέσματος. Στην πράξη η συσταδοποίηση που επιτυγχάνει ο k-Means είναι αρκετά καλή και προσεγγίζει ικανοποιητικά τη βέλτιστη λύση. Αξίζει να σημειωθεί ότι το πρόβλημα συσταδοποίησης που καλείται να επιλύσει ο k-Means ανήκει στην κλάση NP-hard [11], επομένως επιδέχεται επίλυση μόνο μέσω ευριστικών μεθόδων.

Είναι επίσης προφανές ότι η ποιότητα των συστάδων που εντοπίζει ο αλγόριθμος εξαρτάται άμεσα και από τις αρχικές τιμές που αναθέτει στα κέντρα $\boldsymbol{\theta_j}$, αφού με διαφορετική αρχικοποίησή τους τα αποτελέσματα μπορεί να ποικίλλουν σημαντικά. Για το σκοπό αυτό έχει προταθεί μια παραλλαγή του k-Means με την ονομασία **k-Means++** [12], ο οποίος διαθέτει μια βελτιστοποιημένη τεχνική αρχικοποίησης των κέντρων και παρουσιάζεται συνοπτικά παρακάτω:

1. Επιλέγεται τυχαία ένα κέντρο $\boldsymbol{\theta_1}$ από τα δεδομένα, θεωρώντας ομοιόμορφη κατανομή.
2. Για κάθε διάνυσμα $\boldsymbol{x_i}$ υπολογίζεται η απόστασή του $D(\boldsymbol{x_i})$ από το κοντινότερο κέντρο.

3. Επιλέγεται ένα κέντρο $\boldsymbol{\theta_j}$ από τα δεδομένα, με πιθανότητα επιλογής του κάθε $\boldsymbol{x_i}$ ανάλογη της $D(\boldsymbol{x_i})^2$.
4. Τα βήματα 2-3 επαναλαμβάνονται μέχρι να επιλεγούν *k* κέντρα.
5. Στη συνέχεια εφαρμόζεται ο κλασικός k-Means αλγόριθμος.

Ένα ακόμη ζήτημα που πρέπει να ληφθεί υπ' όψη πριν τη χρήση του k-Means είναι η εύρεση μιας κατάλληλης τιμής του *k* για να δοθεί ως είσοδος στον αλγόριθμο. Έχουν κατά καιρούς προταθεί διάφορες μέθοδοι για την απομόνωση "καλών" τέτοιων τιμών που θα αποφέρουν ποιοτικές συστάδες και στην παρούσα εργασία θα γίνει χρήση της μετρικής *f*(*K*) όπως περιγράφεται στην αντίστοιχη δημοσίευση των D. T. Pham, S. S. Dimov και C. D. Nguyen (2005) [13]. Η επιλογή αυτή έγινε λόγω της σχετικά μικρής πολυπλοκότητας της μεθόδου και της αποτελεσματικότητάς της ακόμη και σε μεγάλο όγκο δεδομένων. Πιο συγκεκριμένα, ορίζεται η *παραμόρφωση* (*distortion*) ως ένα μέτρο της απόστασης ανάμεσα στα στοιχεία της συστάδας και το κέντρο της:

$$I_j = \sum_{x_i \in C_j} \|\boldsymbol{x_i} - \boldsymbol{\theta_j}\|^2 \qquad \textbf{(7)}$$

Ο συνολικός αντίκτυπος των παραμορφώσεων όλων των συστάδων για δεδομένη τιμή του *k* δίνεται από τη σχέση:

$$S_k = \sum_{j=1}^{k} I_j \qquad \textbf{(8)}$$

Έπειτα οι συγγραφείς καταλήγουν στον εξής ορισμό:

$$f(K) = \begin{cases} 1, & \alpha\nu \quad K=1 \\ \dfrac{S_K}{\alpha_K S_{K-1}}, & \alpha\nu \quad S_{K-1} \neq 0, \ \forall K>1 \\ 1, & \alpha\nu \, S_{K-1}=0, \ \forall K>1 \end{cases} \qquad \textbf{(9)}$$

$$\alpha_K = \begin{cases} 1-\dfrac{3}{4N_d}, & \alpha\nu \quad K=2 \ \ \kappa\alpha\iota \ \ N_d>1 \\ \alpha_{K-1}+\dfrac{1-\alpha_{K-1}}{6}, & \alpha\nu \quad K>2 \ \ \kappa\alpha\iota \ \ N_d>1 \end{cases} \qquad \textbf{(10)}$$

όπου N_d: οι διαστάσεις του χώρου δεδομένων και

α_K: συντελεστής βάρους

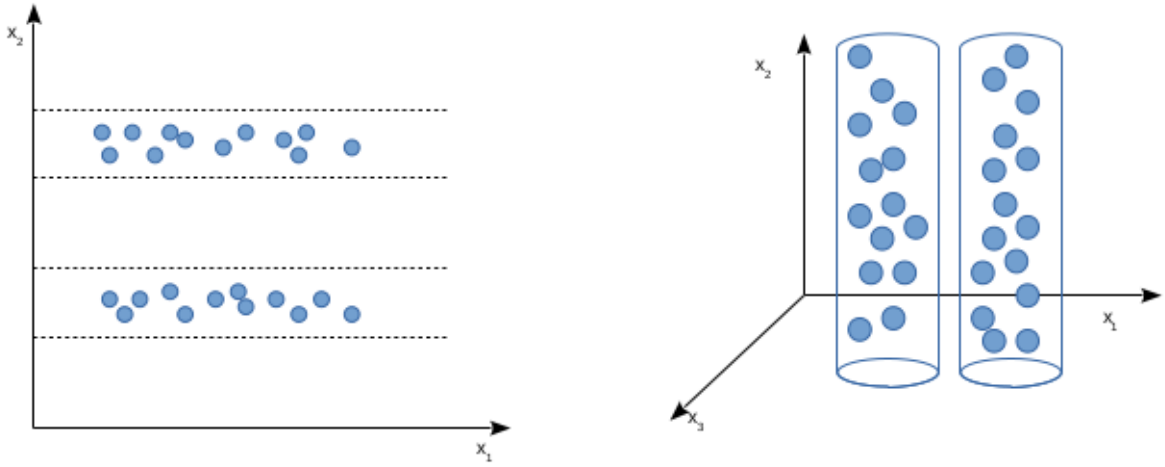
Σύμφωνα με τα παραπάνω, η συνάρτηση $f(K)$ που ορίσαμε αποτελεί το λόγο της πραγματικής προς την εκτιμώμενη παραμόρφωση, κι επομένως η τιμή της ελαττώνεται όταν υπάρχουν περιοχές στο χώρο δεδομένων μας με υψηλή πυκνότητα. Έτσι λοιπόν μπορούμε να υπολογίσουμε τη μετρική αυτή για διάφορες τιμές του k, να επιλέξουμε εκείνες για τις οποίες η $f(K)$ αποκτά τη μικρότερη τιμή κι εν τέλει να αποφασίσουμε για την καταλληλότερη από αυτές με εποπτεία της συσταδοποίησης που προκύπτει.

Τέλος, είναι σημαντικό να αναφερθεί ότι ο k-Means είναι ευαίσθητος στο θόρυβο και την ύπαρξη ακραίων τιμών στα δεδομένα (outliers). Αν κατά την εκτέλεσή του βρεθεί αντιμέτωπος με κάποια ακραία τιμή, τότε αναπόφευκτα θα την αναθέσει σε κάποια συστάδα, επηρεάζοντας άμεσα την τιμή του αντίστοιχου κέντρου κι επομένως την ποιότητα της τελικής συσταδοποίησης. Για το λόγο αυτό ενδείκνυται να γίνεται κατάλληλη κανονικοποίηση των δεδομένων πριν από τη χρήση τους ή/και απομάκρυνση του θορύβου όπου είναι εφικτό.

δ-Biclustering

Όπως έχει αναφερθεί στις προηγούμενες παραγράφους, όταν εξετάζουμε M αριθμό γνωρισμάτων, τότε κάθε σημείο των δεδομένων μας (datapoint) είναι ένα διάνυσμα M-διαστάσεων. Στην περίπτωση όμως που το M είναι ένας αρκετά μεγάλος αριθμός (συνήθως > 10 - 15), τα διανύσματά μας αποτελούν σημεία ενός υπερχώρου (hyperspace) $\Re^M$ με πολύ διαφορετικές ιδιότητες. Πιο συγκεκριμένα, όταν τα δεδομένα ακολουθούν την ίδια κατανομή παρατηρείται το φαινόμενο η απόσταση των δύο κοντινότερων σημείων να γίνεται ίση με την απόσταση των δύο πιο απομακρυσμένων σημείων του χώρου, αναιρώντας έτσι τη χρησιμότητα των μετρικών απόστασης. Το πρόβλημα αυτό αναφέρεται στη βιβλιογραφία ως «*η κατάρα της διαστατικότητας*» (“*the curse of dimensionality*”) [14].

Ένα ακόμη πρόβλημα της υψηλής διαστατικότητας των δεδομένων αποτελεί το γεγονός ότι πολύ συχνά μόνο ένας μικρός αριθμός γνωρισμάτων συνεισφέρει στη διαμόρφωση συστάδων, οι οποίες τελικά βρίσκονται σε έναν μικρό υποχώρο του αρχικού υπερχώρου (Σχήμα 9). Έτσι, αλγόριθμοι οι οποίοι λαμβάνουν υπ' όψη το σύνολο των γνωρισμάτων αδυνατούν να εντοπίσουν τέτοιου είδους συγκεντρώσεις των δεδομένων, δίνοντας τελικά ανακριβείς συσταδοποιήσεις.



(α) *(β)*

Σχήμα 9: (α) Οι δύο συστάδες προβάλλονται πάνω στον άξονα x_2 (β) Οι δύο συστάδες προβάλλονται στον υποχώρο των (x_1, x_3)

Για την αντιμετώπιση των παραπάνω προβλημάτων έχουν προταθεί ποικίλοι αλγόριθμοι οι οποίοι κατορθώνουν να εντοπίζουν συστάδες σε διάφορους υποχώρους του αρχικού χώρου δεδομένων. Μια κατηγορία τέτοιων μεθόδων αποτελεί το Biclustering ή Co-clustering. Από αυτές θα μας απασχολήσει κυρίως ο αλγόριθμος δ-Biclustering που προτάθηκε το 2000 από τους Y. Cheng και G. M. Church [15], ο οποίος δέχεται ως είσοδο έναν πίνακα N x M, με γραμμές τα διανύσματα εισόδου $\boldsymbol{x}_i$, και αναζητά συμπαγείς συστάδες (που εδώ καλούνται δ-biclusters) ταυτόχρονα σε γραμμές και στήλες.

Θα περιγράψουμε συνοπτικά τη λειτουργία του αλγορίθμου. Έστω N το σύνολο των παρατηρήσεων, M το σύνολο των γνωρισμάτων και x_{ij} το στοιχείο του πίνακα που αντιστοιχεί στην i-στή παρατήρηση για το j-στό γνώρισμα. Έστω ακόμη $I \subset X$ και $J \subset Y$ υποσύνολα των παρατηρήσεων και των γνωρισμάτων. Στο ζεύγος (I, J) αντιστοιχεί μια τιμή που ονομάζεται Μέσο Τετραγωνικό Υπόλοιπο (Mean Squared Residue, MSR) και ορίζεται με τον ακόλουθο τύπο:

$$H(I,J) = \frac{1}{|I||J|}\sum_{i\in I,\, j\in J}\left(x_{ij} - x_{iJ} - x_{Ij} + x_{IJ}\right)^2 \qquad \textbf{(11)}$$

όπου $$x_{iJ} = \frac{1}{|J|}\sum_{j\in J} x_{ij} \qquad \textbf{(12)}$$

$$x_{Ij} = \frac{1}{|I|}\sum_{i\in I} x_{ij} \qquad \textbf{(13)}$$

$$x_{IJ} = \frac{1}{|I||J|}\sum_{i\in I,\, j\in J} x_{ij} \qquad \textbf{(14)}$$

Μια συστάδα αποτελεί δ-bicluster όταν ισχύει $H(I, J) \leq \delta$, για κάποια θετική σταθερά δ.

Συμπεραίνουμε λοιπόν ότι για ένα ιδανικό δ-bicluster είναι $H(I, J) = 0$, αφού για τέτοια τιμή του MSR τα στοιχεία της συστάδας εμφανίζουν ομοιόμορφη διακύμανση. Επομένως, χρησιμοποιώντας αυτή τη μετρική συνάρτηση, ο αλγόριθμος προσπαθεί να ανακαλύψει συστάδες με το ελάχιστο δυνατό MSR. Σύμφωνα με τους εμπνευστές του δ-Bicluster το πρόβλημα αυτό ανάγεται σε NP-hard, επομένως η προτεινόμενη τεχνική δίνει προσεγγιστικές συσταδοποιήσεις.

Η λειτουργία του αλγορίθμου περιγράφεται συνοπτικά στον παρακάτω ψευδοκώδικα:

Αλγόριθμος δ-Biclustering

```
    Είσοδος:
        Ο N x M πίνακας δεδομένων, A
        Η παράμετρος πολλαπλών διαγραφών, α ≥ 1
        Το μέγιστο αποδεκτό MSR, δ ≥ 0
        Το πλήθος των δ-biclusters, n
    Έξοδος:
        Τα n δ-biclusters του A

    Μέθοδος:
        δ-biclusters = Ø
        Για i = 1 έως n {
            B = Πολλαπλή_Διαγραφή_Κόμβων(A, δ, α)
            C = Απλή_Διαγραφή_Κόμβων(B, δ)
            D = Προσθήκη_Κόμβων(C)
            δ-biclusters = δ-biclusters U D
            Αντικατάσταση των στοιχείων του A που βρίσκονται
            και στον D με τυχαίους αριθμούς
         }
```

Πολλαπλή_Διαγραφή_Κόμβων

Είσοδος:

Ο I x J πίνακας δεδομένων, M

Η παράμετρος πολλαπλών διαγραφών, α ≥ 1

Το μέγιστο αποδεκτό MSR, δ ≥ 0

Έξοδος:

Ένα δ-bicluster M' που είναι υπο-πίνακας του M και περιέχει I' γραμμές και J' στήλες, με MSR όχι μεγαλύτερο του δ

Μέθοδος:

```
M' = M, I' = I, J' = J
Επανάλαβε {
    Υπολόγισε το H(I', J')
    Αν H(I', J') ≤ δ{
        Επίστρεψε το M'
    }
    Αν #γραμμών > 100 {
       Διάγραψε τις γραμμές i∈I' για τις οποίες
       ισχύει:
```

$$\frac{1}{|J'|}\sum_{j\in J'}\left(x_{ij}-x_{iJ}-x_{Ij}+x_{IJ}\right)^2 > \alpha H(I',J')$$

```
       Υπολόγισε το H(I', J')
    }
    Αν #στηλών > 100 {
       Διάγραψε τις στήλες j∈J' για τις οποίες
       ισχύει:
```

$$\frac{1}{|I'|}\sum_{i\in I'}\left(x_{ij}-x_{iJ}-x_{Ij}+x_{IJ}\right)^2 > \alpha H(I',J')$$

```
       Υπολόγισε το H(I', J')
    }
} Μέχρι να μη γίνει καμία νέα διαγραφή
Επίστρεψε το M'
```

Απλή_Διαγραφή_Κόμβων

Είσοδος:

Ο I x J πίνακας δεδομένων, M

Το μέγιστο αποδεκτό MSR, δ ≥ 0

Έξοδος:

Ένα δ-bicluster M' που είναι υπο-πίνακας του M και περιέχει I' γραμμές και J' στήλες, με MSR όχι μεγαλύτερο του δ

Μέθοδος:

M' = M, I' = I, J' = J

Υπολόγισε το H(I', J')

Όσο H(I', J') > δ επανάλαβε {

Βρες τη γραμμή i∈I' με το μέγιστο:

$$d(i) = \frac{1}{|J'|}\sum_{j\in J'}(x_{ij}-x_{iJ}-x_{Ij}+x_{IJ})^2$$

Βρες τη γραμμή j∈J' με το μέγιστο:

$$d(j) = \frac{1}{|I'|}\sum_{i\in I'}(x_{ij}-x_{iJ}-x_{Ij}+x_{IJ})^2$$

Διάγραψε τη γραμμή ή στήλη με το μεγαλύτερο d

Υπολόγισε το H(I', J')

}

Επίστρεψε το M'

Προσθήκη_Κόμβων

Είσοδος:

Ο I x J πίνακας δεδομένων, M

Έξοδος:

Ένα δ-bicluster M' που περιέχει I' γραμμές και J' στήλες, τέτοια ώστε I ⊂ I', J ⊂ J', και το MSR του M' ≤ το MSR του M

Μέθοδος:

```
Επανάλαβε {
    I = I' και J = J'
    Υπολόγισε το H(I, J)
    Πρόσθεσε τις στήλες j∉J για τις οποίες ισχύει:
        (1/|I|) Σ_{i∈I} (x_ij − x_iJ − x_Ij + x_IJ)^2 ⩽ H(I, J)
    ώστε να προκύψει το J'
    Υπολόγισε το H(I, J')
    Πρόσθεσε τις γραμμές i∉I για τις οποίες ισχύει:
        (1/|J|) Σ_{j∈J} (x_ij − x_iJ − x_Ij + x_IJ)^2 ⩽ H(I, J)
    ώστε να προκύψει το I'
    Υπολόγισε το H(I', J')
} Μέχρι να μη γίνει καμία νέα προσθήκη
Επίστρεψε το M'
```

Πολυπλοκότητα: O(NM), όπου N το πλήθος των γραμμών και M το πλήθος των στηλών του πίνακα δεδομένων.

Με κατάλληλη επιλογή των παραμέτρων *α* και *δ* του αλγορίθμου, καθώς και κατάλληλο προσδιορισμό του κατωφλίου για την πολλαπλή διαγραφή κόμβων (παραπάνω έχει οριστεί αυθαίρετα ως 100) ο δ-Biclustering επιτυγχάνει αξιοσημείωτα αποτελέσματα με σχετικά μικρό χρόνο εκτέλεσης. Αξίζει τέλος να σημειώσουμε ότι όσο μικρότερη είναι η τιμή της παραμέτρου *δ*, τόσο υψηλότερη θα είναι και η ποιότητα των τελικών συστάδων, ενώ όσο μεγαλύτερη η τιμή της *α*, τόσο γρηγορότερα θα γίνεται η διαγραφή των κόμβων και άρα η σύγκλιση του αλγορίθμου.

Αξιολόγηση των τελικών συστάδων (Cluster Evaluation)

Έπειτα από την εφαρμογή κάποιας μεθόδου συσταδοποίησης είναι ιδιαίτερα σημαντικό να αποφασίσουμε για την ποιότητα του τελικού αποτελέσματος ή/και την επιλογή των κατάλληλων τιμών για τις παραμέτρους εισόδου των αλγορίθμων. Λόγω της φύσης της μη-επιβλεπόμενης μάθησης η διαδικασία αυτή είναι αρκετά περίπλοκη, αφού δεν διαθέτουμε εκ των προτέρων καμία πληροφορία για τη δομή ή ακόμη και την ύπαρξη συστάδων στα

δεδομένα μας [16]. Παρ' όλα αυτά, έχουν προταθεί ποικίλες μέθοδοι για την εκτίμηση της ποιότητας και της ακρίβειας των συσταδοποιήσεων, οι οποίες χωρίζονται σε δυο γενικές κατηγορίες. Έστω *X* το σύνολο των δεδομένων μας και *C* η συσταδοποίηση που αποκαλύπτει ο αλγόριθμός μας.

1. **Εξωτερικά κριτήρια**

   Στην περίπτωση αυτή γνωρίζουμε εκ των προτέρων ή επιβάλλουμε οι ίδιοι μια δομή στα δεδομένα μας και εφαρμόζουμε τον αλγόριθμο. Με τον τρόπο αυτό μπορούμε άμεσα να αποφανθούμε για την ποιότητα της συσταδοποίησης *C*.

   Μπορούμε επίσης να μελετήσουμε την ομοιότητα της *C* με μια τυχαία διαμέριση του *X*. Η πιο διαδεδομένη μέθοδος είναι η επαναληπτική διαμέριση του *X* σε ανεξάρτητα υποσύνολα με ομοιόμορφη τυχαία δειγματοληψία και η σύγκρισή τους με τη συσταδοποίηση *C* μέσω της χρήσης κατάλληλων μετρικών και στατιστικών τεστ. Για μεγαλύτερη ευκολία συνήθως γίνεται χρήση τεχνικών Monte Carlo.

2. **Εσωτερικά κριτήρια**

   Με τη χρήση εσωτερικών κριτηρίων λαμβάνουμε υπ' όψη αποκλειστικά και μόνο τη συσταδοποίηση *C*, αγνοώντας οποιαδήποτε άλλη εξωτερική πληροφορία, σε μια προσπάθεια να εκτιμήσουμε το βαθμό ομοιότητας των στοιχείων στο εσωτερικό της κάθε συστάδας και το βαθμό ανομοιότητας ανάμεσα σε διαφορετικές συστάδες.

Στη συγκεκριμένη εργασία θα κάνουμε χρήση αποκλειστικά εσωτερικών κριτηρίων. Παρουσιάσαμε ήδη σε προηγούμενη παράγραφο τη *μέθοδο f(K)* για την επιλογή κατάλληλης τιμής της παραμέτρου *k* στον αλγόριθμο k-Means. Θα χρησιμοποιήσουμε ακόμη δύο σημαντικούς δείκτες για την εκτίμηση της ποιότητας των τελικών συστάδων τους οποίους και θα αναλύσουμε στη συνέχεια. Ο πρώτος από αυτούς είναι ο *τροποποιημένος δείκτης Γ του Hubert* (*modified Hubert Gamma statistic*) [17], που ορίζεται με τον τύπο:

$$\Gamma = \frac{1}{L}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} P(i,j)Q(i,j) \qquad \textbf{(15)}$$

όπου *L*: το πλήθος όλων των πιθανών ζευγών στο σύνολο δεδομένων *X*

N: το πλήθος των δεδομένων στο σύνολο *X*

P(*i*,*j*): ο πίνακας ομοιότητας των στοιχείων του συνόλου *X*

Q(*i*,*j*): ο πίνακας ομοιότητας μεταξύ των $\boldsymbol{\theta_{Ci}}$, $\boldsymbol{\theta_{Cj}}$, δηλαδή των κέντρων των συστάδων στις οποίες ανήκουν τα στοιχεία $\boldsymbol{x_i}$, $\boldsymbol{x_j}$.

Παρατηρούμε ότι στην περίπτωση συμπαγών συστάδων η απόσταση ανάμεσα στα $\boldsymbol{\theta_{Ci}}$ και $\boldsymbol{\theta_{Cj}}$ είναι σχεδόν ίδια με την απόσταση ανάμεσα στα $\boldsymbol{x_i}$ και $\boldsymbol{x_j}$, επομένως η τιμή του δείκτη Γ αυξάνεται. Έτσι λοιπόν αρκετά υψηλές τιμές του Γ υποδηλώνουν καλή ποιότητα συσταδοποίησης. Τέλος, πρέπει να σημειωθεί ότι είναι απαραίτητη η χρήση της ίδιας μετρικής απόστασης για τους πίνακες *P* και *Q*.

Ακόμη θα χρησιμοποιήσουμε τον *δείκτη των Davies-Bouldin* (*Davies-Bouldin index* ή *DB index*) [18], ο οποίος παρέχει αρκετά αξιόπιστα αποτελέσματα στην πλειοψηφία των περιπτώσεων [19] και ορίζεται ως:

$$DB = \frac{1}{k}\sum_{i=1}^{k} \max_{i \neq j}\left(\frac{S_i + S_j}{d_{ij}}\right) \qquad \textbf{(16)}$$

όπου k: ο αριθμός των συστάδων

d_{ij}: η απόσταση ανάμεσα στις συστάδες C_i και C_j

και S_i το μέτρο της διασποράς στο εσωτερικό της συστάδας C_i που ορίζεται με τον τύπο:

$$S_i = \left(\frac{1}{|C_i|}\sum_{\boldsymbol{x} \in C_i} \|\boldsymbol{x} - \boldsymbol{\theta_i}\|^r\right)^{\frac{1}{r}} \qquad \textbf{(17)}$$

Μικρές τιμές του δείκτη DB υποδηλώνουν συμπαγείς, καλά διαχωρισμένες συστάδες, αφού σε αυτή την περίπτωση η απόσταση d_{ij} είναι αρκετά μεγαλύτερη από το άθροισμα $S_i + S_j$.

Α.3) Το Πρόβλημα

Η επιστήμη της Βιολογίας ασχολείται με ιδιαίτερα περίπλοκα φαινόμενα, όπως είναι η εμβρυογένεση, η νευροφυσιολογία, η ογκογένεση, και άλλα. Ιδιαίτερο ενδιαφέρον παρουσιάζει το φαινόμενο της ογκογένεσης, τόσο από βιολογικής όσο και από κοινωνιολογικής πλευράς, αφού ο καρκίνος αποτελεί μια θανατηφόρα ασθένεια στην πλειοψηφία των περιπτώσεων και επηρεάζει άμεσα την ποιότητα ζωής των ασθενών. Παρά τις εκτεταμένες μελέτες που έχουν πραγματοποιηθεί κι εξακολουθούν να πραγματοποιούνται σχετικά με τη διαδικασία της ογκογένεσης, γνωρίζουμε ακόμη πολύ λίγα για το μηχανισμό που την υποκινεί [20].

Το κόστος των θεραπειών που εφαρμόζονται σήμερα για τα διάφορα είδη καρκίνου είναι συχνά απαγορευτικό. Αυτό οφείλεται στο γεγονός ότι οι έρευνες για την ανακάλυψη νέων φαρμάκων απαιτούν υψηλή χρηματοδότηση και είναι αρκετά χρονοβόρες, αφού βασίζονται κυρίως στο δόγμα του “trial and error”, δηλαδή πραγματοποιούνται πολλαπλές δοκιμές και επιβάλλονται συνεχώς διορθώσεις βάσει των παρατηρήσεων, μιας και οι γνώσεις μας σχετικά με τη διαδικασία της καρκινογένεσης είναι ιδιαίτερα ελλειπείς. Έτσι, λοιπόν, είναι εμφανές ότι αποδομώντας το μηχανισμό της ογκογένεσης και κατανοώντας τους κανόνες που τη διέπουν θα είμαστε σε θέση να αντιμετωπίζουμε πιο γρήγορα και αποδοτικά την ασθένεια αυτή, παρέχοντας φάρμακα και θεραπείες με πιθανώς λιγότερες παρενέργειες και οικονομικά πιο προσιτές.

Σε γενικές γραμμές, υπάρχουν δύο διαφορετικές προσεγγίσεις του προβλήματος εύρεσης θεραπειών για την ασθένεια του καρκίνου:

1. Μελέτη συγκεκριμένου υποτύπου καρκίνου ή/και ασθενή και ανάπτυξη εξατομικευμένων θεραπειών (personalized/individualized therapies): Βάσει αυτής της προσέγγισης γίνεται χρήση των γενετικών δεδομένων ενός ασθενούς και μελέτη του ατομικού του γονιδιώματος για την εξεύρεση της κατάλληλης θεραπευτικής αγωγής και τη βέλτιστη προσαρμογή της. Με τον τρόπο αυτό ελαχιστοποιούνται οι παρενέργειες και μεγιστοποιείται η πιθανότητα επιτυχίας της θεραπείας για τη συγκεκριμένη περίπτωση.
2. Μελέτη του γενικού μηχανισμού που διέπει την ογκογένεση και ανάπτυξη καθολικής θεραπείας: Κατά την προσέγγιση αυτή θεωρείται ιδιαίτερα σημαντική η κατανόηση των γενικών κανόνων που περιγράφουν τη διαδικασία της καρκινογένεσης, διότι με τον τρόπο αυτό θα καταστεί πιο εύκολη η εξεύρεση αποτελεσματικών θεραπειών ή ακόμη και μιας καθολικής θεραπείας ικανής να εφαρμοστεί σε οποιονδήποτε τύπο καρκίνου.

Ακολουθώντας τη λογική της δεύτερης προσέγγισης, καθίσταται ιδιαίτερα σημαντική η ανακάλυψη κοινών χαρακτηριστικών ανάμεσα στα διαφορετικά είδη όγκων. Γνωρίζουμε ότι διαφορετικές νεοπλασίες κατέχουν διαφορετικά χαρακτηριστικά και διαφορετικό φαινότυπο, επομένως η ανακάλυψη κοινών στοιχείων θα αποτελέσει τη βάση για περαιτέρω έρευνα πάνω σε κάποιον κοινό μηχανισμό. Στην παρούσα εργασία θα κινηθούμε προς αυτή την κατεύθυνση και θα επιχειρήσουμε να εντοπίσουμε ομάδες γονιδίων με κοινή έκφραση ανάμεσα σε διαφορετικούς τύπους νεοπλασιών. Για το σκοπό αυτό θα κάνουμε χρήση τεχνικών μηχανικής μάθησης πάνω σε δεδομένα από πειράματα μικροσυστοιχιών DNA. Θα

εφαρμόσουμε αλγορίθμους συσταδοποίησης και θα ελέγξουμε τα αποτελέσματα με τυπικές μεθόδους.

Ήδη υπάρχουσες προσεγγίσεις

Έχουν δημοσιευτεί ποικίλες μελέτες [21]–[27] σχετικά με την εφαρμογή τεχνικών συσταδοποίησης σε δεδομένα γονιδιακής έκφρασης. Η συντριπτική πλειοψηφία των μελετών αυτών ακολουθούν την εξής διαδικασία: σχηματίζουν μια δομή *G* x *S* με τα δεδομένα των μικροσυστοιχιών (όπου *G* το πλήθος των γονιδίων και *S* το πλήθος των δειγμάτων) και στη συνέχεια εκτελούν κάποιον αλγόριθμο συσταδοποίησης πάνω σε αυτά (συνήθως τον k-Means λόγω της απλότητάς του).

Σε κάποιες περιπτώσεις [28]–[37] γίνεται επίσης μια προσπάθεια μελέτης της εξέλιξης της γονιδιακής έκφρασης στο χρόνο με την εκτέλεση διαδοχικών πειραμάτων και τη δημιουργία έπειτα μιας τρισδιάστατης δομής *G* x *S* x *T* (όπου *T* το πλήθος των χρονικών στιγμών που έγιναν τα πειράματα). Στη συνέχεια εφαρμόζεται και πάλι κάποιος αλγόριθμος συσταδοποίησης (εδώ συνήθως προτιμώνται τεχνικές συσταδοποίησης υποχώρου) ώστε να βρεθούν τα γονίδια εκείνα που παρουσιάζουν κοινή έκφραση σε συγκεκριμένες χρονικές στιγμές/περιόδους.

Η δική μας προσέγγιση

Υπάρχει γενικά πληθώρα άρθρων στη βιβλιογραφία σχετικά με τη διαχείριση και την ανάλυση των δεδομένων από μικροσυστοιχίες DNA. Σε καμία όμως περίπτωση δεν έχει γίνει προσπάθεια μελέτης των χρωμοσωμάτων κατά τη συσταδοποίηση. Γνωρίζοντας ότι τα γονίδια χωρίζονται εκ φύσεως σε ομάδες βάσει των χρωμοσωμάτων (συγκεκριμένα ο ανθρώπινος οργανισμός διαθέτει 23 ζεύγη χρωμοσωμάτων), επιχειρήσαμε να λάβουμε υπ' όψη την κατηγοριοποίηση αυτή στους υπολογισμούς μας με την προοπτική της αποκάλυψης συσχετίσεων ή αλληλεπιδράσεων που δεν ήταν εμφανείς πρωτύτερα.

Πιο συγκεκριμένα, δημιουργήσαμε μια τρισδιάστατη δομή *G* x *S* x *C* (όπου *C* το πλήθος των χρωμοσωμάτων) και τροποποιήσαμε τον αλγόριθμο k-Means ώστε να εφαρμοστεί στα νέα αυτά τρισδιάστατα δεδομένα. Επίσης, κάναμε χρήση μιας παραλλαγής του δ-Biclustering

με την ονομασία δ-TRIMAX που παρέχει τη δυνατότητα εντοπισμού triclusters με ταυτόχρονη διερεύνηση όλων των διαστάσεων του πίνακα. Τέλος, ελέγξαμε την ποιότητα των αποτελεσμάτων και συγκρίναμε τις συσταδοποιήσεις που λάβαμε από τις παραπάνω μεθόδους.

Α.4) ΥΛΟΠΟΙΗΣΗ

Γλώσσα προγραμματισμού

Επιλέξαμε να χρησιμοποιήσουμε τη γλώσσα προγραμματισμού Python (έκδοση 3.5.1), επειδή αποτελεί μια γλώσσα υψηλού επιπέδου, αρκετά εύχρηστη και ευέλικτη, ενώ παράλληλα είναι ανοικτού κώδικα (υπό την άδεια Python Software Foundation License, PSFL [38]) και cross-platform. Επίσης επιτρέπει τη χρήση ποικίλων στυλ προγραμματισμού (αντικειμενοστρεφή, διαδικαστικό, συναρτησιακό, κλπ) και συνοδεύεται από πλήθος βιβλιοθηκών για κάθε πιθανή ανάγκη.

Επιλογή βιβλιοθηκών

Έγινε χρήση της στοίβας SciPy, η οποία αποτελεί μια ολοκληρωμένη συλλογή πακέτων και βιβλιοθηκών της γλώσσας Python για επιστημονική χρήση. Πιο συγκεκριμένα, περιλαμβάνει:

- SciPy [39]: Περιέχει εύχρηστες και αποδοτικές ρουτίνες για αριθμητικούς υπολογισμούς (π.χ. μαθηματική ολοκλήρωση) και βελτιστοποίηση υπολογισμών.
- NumPy [40]: Παρέχει αποδοτικές *N*-διάστατες δομές δεδομένων για γενική χρήση και ποικίλες ρουτίνες για το χειρισμό και την επεξεργασία τους. Περιέχει ακόμη ρουτίνες γραμμικής άλγεβρας, μετασχηματισμών Fourier και γεννήτριες ψευδοτυχαίων αριθμών.
- Pandas [41]: Παρέχει αποδοτικές και ευέλικτες δομές για δεδομένα με ετικέτες, και ποικίλες ρουτίνες για το χειρισμό τους (π.χ. ιεραρχική ονομασία στηλών/γραμμών, ρουτίνες εισόδου/εξόδου δεδομένων με υποστήριξη πλήθους τύπων αρχείων, κλπ).
- iPython [42]: Προσφέρει ένα δυναμικό κέλυφος για άμεση και αποδοτική αλληλεπίδραση με τη γλώσσα Python, με υποστήριξη οπτικοποίησης δεδομένων και

μαθηματικών τύπων, καταγραφής ιστορικού, παράλληλης επεξεργασίας και πλήθος άλλων εργαλείων.

- Matlpotlib [43]: Επιτρέπει τη δημιουργία διαφόρων ειδών γραφικών παραστάσεων και γραφημάτων, με δυνατότητα αλληλεπίδρασης και πλήρους μορφοποίησης.

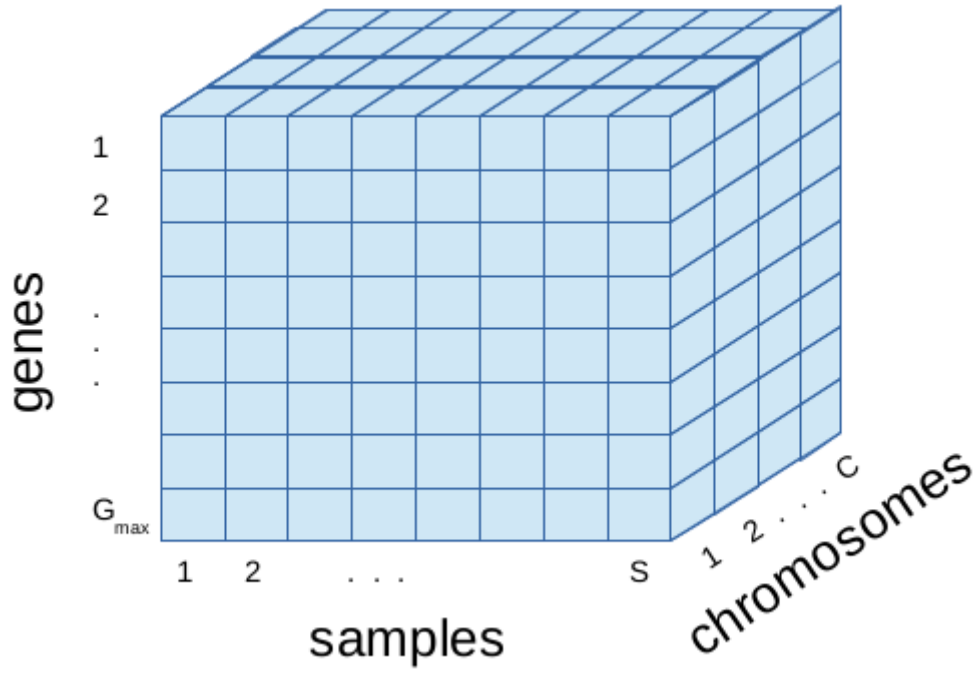
Επιπρόσθετες βιβλιοθήκες που χρησιμοποιήσαμε:

- Seaborn [44]: Παρέχει καλύτερη οπτικοποίηση των γραφημάτων που παράγονται από το Matplotlib.
- Scikit-learn [45]: Μια εξαιρετικά δυνατή βιβλιοθήκη με πλήθος συναρτήσεων και μεθόδων για ταξινόμηση δεδομένων, συσταδοποίηση, γραμμική παλινδρόμηση, μείωση διαστατικότητας, κανονικοποίηση, αξιολόγηση συστάδων, κ.ά.

Τέλος, έγινε χρήση του Ολοκληρωμένου Περιβάλλοντος Ανάπτυξης (Integrated Development Environment, IDE) Spyder [46], το οποίο παρέχει δυναμικό επεξεργαστή κειμένου, διαδραστικές κονσόλες Python και iPython, δυναμική προβολή τεκμηρίωσης κώδικα, γραφική περιήγηση στις μεταβλητές και τα αρχεία, ιστορικό εντολών, και πολλά άλλα.

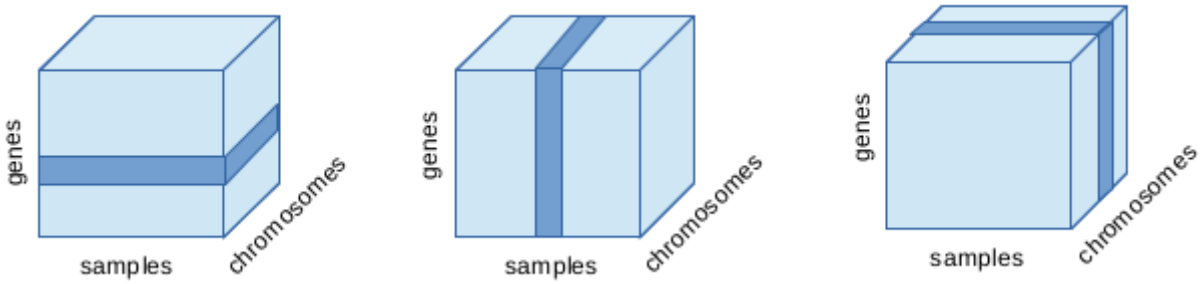
Β. ΚΑΤΑΣΚΕΥΗ ΤΡΙΣΔΙΑΣΤΑΤΟΥ ΠΙΝΑΚΑ

Στην προσπάθειά μας να εισάγουμε την πληροφορία των χρωμοσωμάτων στη διαδικασία συσταδοποίησης, κατασκευάσαμε μια ιδιάζουσα τρισδιάστατη δομή με τα αρχικά μας δεδομένα. Έστω X ο αρχικός πίνακας με τα αποτελέσματα από τα πειράματα των μικροσυστοιχιών, διαστάσεων G x S (όπου G ο αριθμός των γονιδίων και S ο αριθμός των δειγμάτων). Το πρώτο βήμα ήταν να διαχωρίσουμε τα γονίδια ανά χρωμόσωμα, απ' όπου προέκυψαν 23 ομάδες (για τα 23 ζεύγη των ανθρώπινων χρωμοσωμάτων). Έπειτα συνενώσαμε τις ομάδες αυτές κατά αύξοντα αριθμό χρωμοσώματος σε μια νέα διάσταση, καταλήγοντας τελικά σε μια δομή G_{max} x S x C (όπου G_{max} το πλήθος των γονιδίων στο ζεύγος χρωμοσωμάτων με τα περισσότερα γονίδια, S το πλήθος των δειγμάτων και C το πλήθος των χρωμοσωμικών ζευγών). Η μορφή του τελικού τρισδιάστατου πίνακα φαίνεται στο Σχήμα 10.



Σχήμα 10: Ο τρισδιάστατος πίνακας.

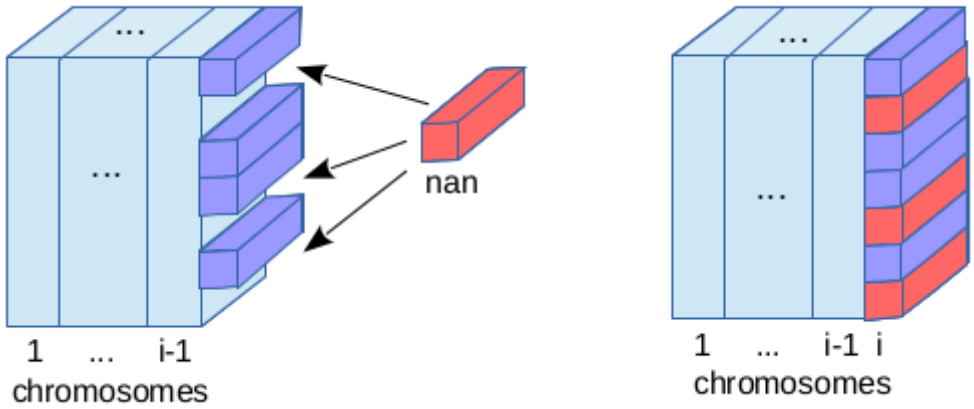
Στη νέα αυτή δομή μπορούν να δημιουργηθούν τριών ειδών τομές, όπως φαίνεται στο Σχήμα 11. Μια τομή στον άξονα των γονιδίων, μια στον άξονα των δειγμάτων και μία στον άξονα των χρωμοσωμάτων.



(α) *(β)* *(γ)*

Σχήμα 11: Οι διαφορετικές τομές του τρισδιάστατου πίνακα: (α) στον άξονα των γονιδίων (β) στον άξονα των δειγμάτων (γ) στον άξονα των χρωμοσωμάτων.

Κατά τη δημιουργία της τρισδιάστατης αυτής δομής προέκυψαν τα εξής προβλήματα: (α) Με ποια σειρά θα τοποθετηθούν τα γονίδια σε κάθε ζεύγος χρωμοσωμάτων, και (β) πώς θα αντιμετωπιστούν οι "κενές" θέσεις, δεδομένου ότι το πλήθος των γονιδίων μειώνεται σταθερά από το πρώτο ως το τελευταίο ζεύγος χρωμοσωμάτων (δηλ. $G_i \geq G_j$, όπου $1 \leq i < j \leq 23$). Έπειτα από διαδοχικούς πειραματισμούς, αποφασίσαμε ότι δεν έχει τόση σημασία η σειρά των γονιδίων μέσα στο ίδιο το ζεύγος χρωμοσωμάτων, αλλά η σχέση τους με τα γονίδια άλλων ζευγών χρωμοσωμάτων που γειτονεύουν με αυτά στον τρισδιάστατο πίνακα. Έτσι, λοιπόν, καταλήξαμε ότι η βέλτιστη πρακτική είναι η τοποθέτηση γονιδίων με παρόμοιο γονιδιακό προφίλ κατά μήκος του άξονα των χρωμοσωμάτων.



Σχήμα 12: Η προσθήκη nan τιμών για τη συμπλήρωση του i-στού ζεύγους χρωμοσωμάτων, μετά την τοποθέτηση όλων των γονιδίων του.

Όσον αφορά τις κενές θέσεις, εκμεταλλευτήκαμε τη δυνατότητα της βιβλιοθήκης NumPy της

Python για χρήση του ειδικού τύπου μεταβλητής nan (Σχήμα 12), ο οποίος αντιπροσωπεύει την απουσία τιμής (nan = not a number). Η βιβλιοθήκη αυτή περιλαμβάνει και πλήθος εργαλείων για τη διεξαγωγή αριθμητικών και στατιστικών υπολογισμών σε έναν πίνακα αγνοώντας τις τιμές nan, τα οποία και χρησιμοποιήθηκαν ευρύτατα στη μετέπειτα ανάλυση.

Πιο συγκεκριμένα, ο αλγόριθμος κατασκευής του τρισδιάστατου πίνακα φαίνεται παρακάτω:

Αλγόριθμος Κατασκευής Τρισδιάστατου Πίνακα

Είσοδος:
Τα υποσύνολα {C_1, C_2, ..., C_{23}}, όπου κάθε C_i περιέχει τα γονίδια του αρχικού πίνακα X που ανήκουν στο ζεύγος χρωμοσωμάτων i

Έξοδος:
Ο τρισδιάστατος πίνακας X_{3D}

Μέθοδος (με τα τελευταία γονίδια):
X_{3D} = C_1
Για i = 2 έως 23 {
Υπολόγισε τον πίνακα distances, όπου το κελί distances[j,j'] περιέχει την Ευκλείδεια απόσταση:
$$d(\boldsymbol{j}, \boldsymbol{j}') = \|\boldsymbol{j} - \boldsymbol{j}'\|$$
ανάμεσα στα **j** $\in$ C_i και **j'** $\in$ C_{i-1}
Επανάλαβε {
Βρες τη μικρότερη τιμή distances[j,j'] στον πίνακα distances
Τοποθέτησε το γονίδιο **j** πίσω από το **j'** στον X_{3D}
Διάγραψε τη j γραμμή και j' στήλη του distances
} Μέχρις ότου ο πίνακας distances να αδειάσει
Γέμισε τις κενές γραμμές του i-στού ζεύγους χρωμοσωμάτων στον X_{3D} με nan
}

Μέθοδος (με κεντροειδή):

$X_{3D} = C_1$

centroids = C_1

Για i = 2 έως 23 {

Υπολόγισε τον πίνακα distances, όπου το κελί distances[j,c] περιέχει την Ευκλείδεια απόσταση

$$d(\boldsymbol{j},\boldsymbol{c}) = \|\boldsymbol{j}-\boldsymbol{c}\|$$

ανάμεσα στα **j** ∈ C_i και **c** ∈ centroids

Επανάλαβε {

Βρες τη μικρότερη τιμή distances[j,c] στον πίνακα distances

Τοποθέτησε το γονίδιο **j** πίσω από το γονίδιο που αντιστοιχεί στο κεντροειδές **c** στον X_{3D}

Διάγραψε τη j γραμμή και c στήλη του distances

} Μέχρις ότου ο πίνακας distances να αδειάσει

Γέμισε τις κενές γραμμές του i-στού ζεύγους χρωμοσωμάτων στον X_{3D} με nan

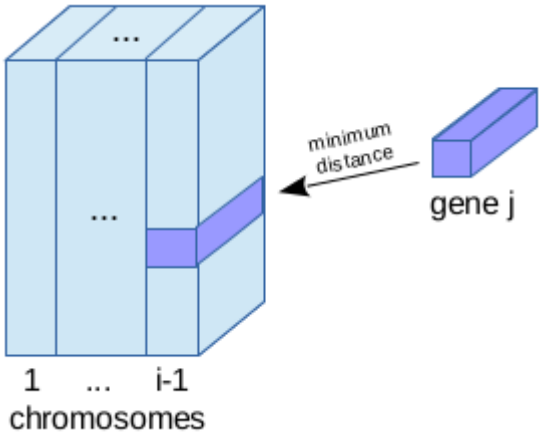
Υπολόγισε εκ νέου τα κεντροειδή του X_{3D}

}

Πολυπλοκότητα: O(CG), όπου C το πλήθος των ζευγών χρωμοσωμάτων και G το μέγιστο πλήθος γονιδίων σε ένα ζεύγος χρωμοσωμάτων. Υπό πραγματικές συνθήκες είναι C << G, επομένως τελικά έχουμε O(G).

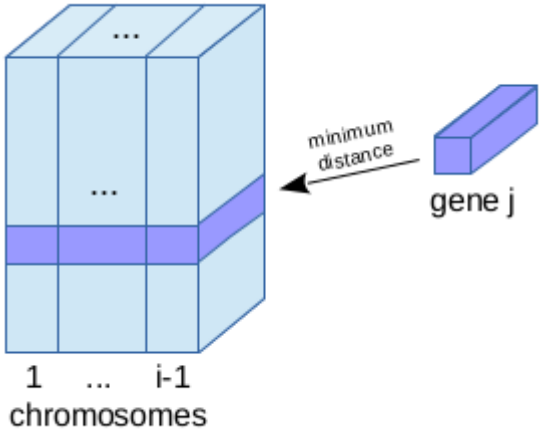
Όπως φαίνεται στον παραπάνω ψευδοκώδικα, η μέθοδος είναι επαναληπτική, δηλαδή ελέγχουμε ένα προς ένα τα ζεύγη χρωμοσωμάτων και τα τοποθετούμε διαδοχικά στον τελικό πίνακα. Επίσης, έχουμε υλοποιήσει δύο διαφορετικές μεθόδους για την τελική τοποθέτηση των γονιδίων:

- Μέθοδος με τα τελευταία γονίδια: Σύμφωνα με αυτήν, κάθε φορά που εξετάζουμε τα γονίδια ενός ζεύγους χρωμοσωμάτων, λαμβάνουμε υπ' όψη αποκλειστικά και μόνο τα γονίδια του προηγούμενου ζεύγους.



Σχήμα 13: Η μέθοδος με τα τελευταία γονίδια.

- Μέθοδος με κεντροειδή: Στην προκειμένη περίπτωση κάθε φορά που εξετάζουμε τα γονίδια ενός ζεύγους χρωμοσωμάτων, λαμβάνουμε υπ' όψη τα κεντροειδή της κάθε οριζόντιας τομής του τρισδιάστατου πίνακα. Με τον όρο κεντροειδές (centroid) εννοούμε ένα διάνυσμα *S* διαστάσεων που αποτελεί τον αριθμητικό μέσο όρο των προηγούμενων *i*-1 διανυσμάτων *S* διαστάσεων στην ίδια οριζόντια τομή του πίνακα.



Σχήμα 14: Η μέθοδος με τα κεντροειδή.

Έπειτα από μια σειρά δοκιμών καταλήξαμε ότι με χρήση κεντροειδών έχουμε αρκετά καλύτερα αποτελέσματα.

Γ. ΣΥΣΤΑΔΟΠΟΙΗΣΗ

Προκειμένου να προσαρμόσουμε τη διαδικασία της συσταδοποίησης στη νέα τρισδιάστατη δομή μας, προτείνουμε κάποιες τροποποιήσεις στον αλγόριθμο k-Means, ή εναλλακτικά τη χρήση μιας επέκτασης του δ-Biclustering για τρισδιάστατο πίνακα. Στη συνέχεια περιγράφεται λεπτομερώς ο κάθε αλγόριθμος.

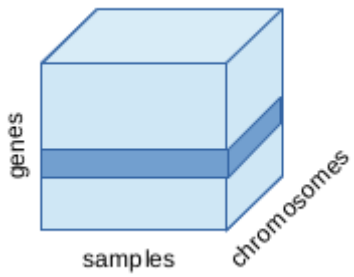
Γ.1) K-MEANS

Επέκταση αλγορίθμου σε τρεις διαστάσεις με νόρμες πινάκων

Αρχικά μετατρέψαμε τον κλασικό αλγόριθμο k-Means ώστε να έχει τη δυνατότητα να διαχειρίζεται απευθείας δεδομένα στη μορφή τρισδιάστατου πίνακα. Για να επιτευχθεί αυτό, θεωρούμε πλέον ότι το σύνολο δεδομένων μας αποτελείται από δισδιάστατους πίνακες αντί για διανύσματα. Έτσι, κατά την αρχικοποίηση του αλγορίθμου επιλέγουμε τη διάσταση ως προς την οποία θα γίνει η συσταδοποίηση (δηλ. ποιο είδος τομής θα θεωρήσουμε ως δεδομένα) και στη συνέχεια εκτελείται ο k-Means συνενώνοντας τομές που παρουσιάζουν κοινό προφίλ έκφρασης. Σε κάθε περίπτωση, τα κέντρα των συστάδων αποτελούν επίσης δισδιάστατους πίνακες, ενώ η μετρική απόστασης που χρησιμοποιείται είναι πλέον κάποια νόρμα πινάκων. Παρακάτω δείχνουμε τη λειτουργία του τροποποιημένου αυτού αλγορίθμου σε κάθε είδος τομής:

- <u>Τομή στον άξονα των γονιδίων</u>

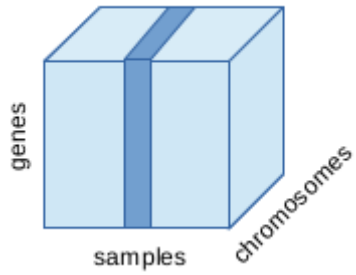
  Εδώ κάθε υποπίνακας δεδομένων αποτελείται από την έκφραση ενός γονιδίου από κάθε ζεύγος χρωμοσωμάτων σε όλα τα δείγματα, κι έχει διαστάσεις *S* x *C*.



Σχήμα 15: Τομή στον άξονα των γονιδίων.

- Τομή στον άξονα των δειγμάτων

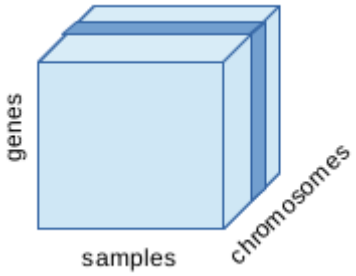
  Σε αυτή την περίπτωση ο κάθε υποπίνακας που εξετάζουμε αποτελείται από την έκφραση όλων των γονιδίων σε όλα τα χρωμοσώματα για ένα συγκεκριμένο δείγμα, κι έχει διαστάσεις *G* x *C*.



Σχήμα 16: Τομή στον άξονα των δειγμάτων.

- Τομή στον άξονα των χρωμοσωμάτων

  Τώρα κάθε υποπίνακας περιέχει την έκφραση όλων των γονιδίων ενός συγκεκριμένου ζεύγους χρωμοσωμάτων σε όλα τα δείγματα, κι έχει διαστάσεις *G* x *S*.



Σχήμα 17: Τομή στον άξονα των χρωμοσωμάτων.

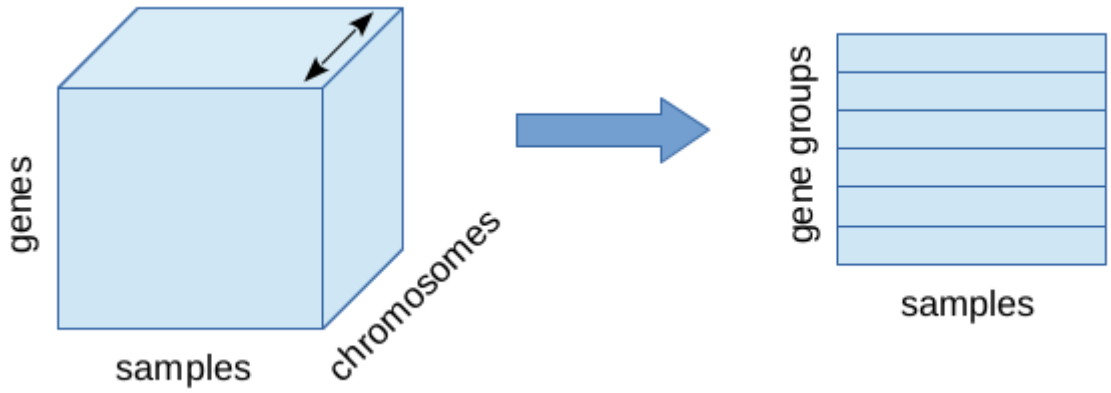
Η αξιολόγηση των συστάδων που παράγει ο παραπάνω αλγόριθμος έγινε κανονικά με χρήση των δεικτών που αναφέρθηκαν στην ενότητα Α.2.

Χρήση κοινού δισδιάστατου k-means στα κεντροειδή του πίνακα

Αν και αρκετά ενδιαφέρουσα, η επέκταση του k-Means που προτάθηκε παραπάνω πάσχει αρκετά από άποψη ταχύτητας, αλλά και αποδοτικότητας. Παρατηρήσαμε ότι ο αλλεπάλληλος υπολογισμός των αποστάσεων μεταξύ πινάκων είναι ιδιαίτερα χρονοβόρος και απαιτεί

πολλούς υπολογιστικούς πόρους. Ακόμη διαπιστώσαμε ότι στην περίπτωση της συσταδοποίησης τομών στον άξονα των γονιδίων για τιμές του *k* μεγαλύτερες από κάποια συγκριτικά χαμηλή τιμή (~ 1/120 του πλήθους των γονιδίων) ο αλγόριθμος δε συγκλίνει ποτέ. Κατά πάσα πιθανότητα αυτό συμβαίνει διότι με τις νόρμες πινάκων λαμβάνουμε υπ' όψη πολύ περισσότερες τιμές απ' ό,τι με τις αντίστοιχες νόρμες διανυσμάτων, κι έτσι αυξάνεται ραγδαία η διαστατικότητα του χώρου, καθιστώντας πολύ δύσκολη την ορθή σύγκριση μεταξύ των δεδομένων.

Για όλους τους παραπάνω λόγους, δοκιμάσαμε να φτιάξουμε έναν πίνακα με τα κεντροειδή της τρισδιάστατης δομής και να χρησιμοποιήσουμε σε αυτόν τον κλασικό k-Means αλγόριθμο. Η συγκεκριμένη μέθοδος αποδείχτηκε αρκετά πιο γρήγορη και ικανή να παράγει υψηλής ποιότητας αποτελέσματα.



Σχήμα 18: Ο μετασχηματισμός του τρισδιάστατου πίνακα σε δισδιάστατο, με υπολογισμό του κεντροειδούς ως προς τον άξονα των χρωμοσωμάτων. Με παρόμοιο τρόπο δημιουργούνται οι αντίστοιχοι δισδιάστατοι πίνακες ως προς τους υπόλοιπους άξονες.

Γ.2) SUBSPACE CLUSTERING

δ-TRIMAX

Σε μια προσπάθεια ανάλυσης δεδομένων που προέκυψαν με διαδοχικά πειράματα μικροσυστοιχιών DNA σε διαφορετικές χρονικές στιγμές, μια ομάδα ερευνητών δημοσίευσε το 2013 ένα νέο αλγόριθμο συσταδοποίησης υποχώρου με το όνομα δ-TRIMAX, ικανό να διαχειρίζεται τρισδιάστατες δομές δεδομένων [30]. Ο αλγόριθμος αυτός βασίζεται στον δ-Biclustering αλγόριθμο των Y. Cheng και G. M. Church και αποτελεί ουσιαστικά μια

επέκτασή του για την ανάλυση δεδομένων της μορφής *G* x *S* x *T* και τον εντοπισμό ομάδων γονιδίων που εμφανίζουν κοινή έκφραση σε κάποιο υποσύνολο των δειγμάτων, σε ένα υποσύνολο χρονικών στιγμών.

Βασιζόμενοι λοιπόν στην εργασία αυτή, ορίζουμε τον πίνακα δεδομένων *X* της μορφής *G* x *S* x *C*, όπου κάθε στοιχείο του x_{gsc} περιέχει την τιμή της μετρούμενης γονιδιακής έκφρασης για το γονίδιο *g* του ζεύγους χρωμοσωμάτων *c*, στο δείγμα *s*. Ορίζουμε επίσης ως tricluster τον υποπίνακα $M(G', S', C') = [m_{gsc}]$, που περιέχει ένα υποσύνολο γονιδίων (G') από ένα υποσύνολο χρωμοσωμάτων (C') σε ένα υποσύνολο δειγμάτων (S'). Τέλος, στόχος του αλγορίθμου είναι η εξεύρεση των tricluster που ικανοποιούν τη συνθήκη:

$$m_{gsc} = \Gamma + \alpha_g + \beta_s + \eta_c$$

όπου Γ: μια σταθερή τιμή

α_g: συντελεστής μετατόπισης για το *g*-στό γονίδιο

β_s: συντελεστής μετατόπισης για το *s*-στό δείγμα

η_c: συντελεστής μετατόπισης για το *c*-στό ζεύγος χρωμοσωμάτων

Κατ' επέκταση μπορούμε να ορίσουμε τους παραπάνω συντελεστές μετατόπισης ως:

$$\alpha_g = m_{gSC} - m_{GSC} \qquad \textbf{(18)}$$

$$\beta_s = m_{GsC} - m_{GSC} \qquad \textbf{(19)}$$

$$\eta_c = m_{GSc} - m_{GSC} \qquad \textbf{(20)}$$

όπου:

$m_{gSC} = \frac{1}{|S'||C'|} \sum_{s \in S', c \in C'} m_{gsc}$ ο μέσος όρος του *g*-στού γονιδίου

$m_{GsC} = \frac{1}{|G'||C'|} \sum_{g \in G', c \in C'} m_{gsc}$ ο μέσος όρος του *s*-στού δείγματος

$m_{GSc} = \frac{1}{|G'||S'|} \sum_{g \in G', s \in S'} m_{gsc}$ ο μέσος όρος του *c*-στού ζεύγους χρωμ/των

$m_{GSC} = \frac{1}{|G'||S'||C'|} \sum_{g \in G', s \in S', c \in C'} m_{gsc}$ ο μέσος όρος του tricluster

Με βάση τα παραπάνω και θεωρώντας την τιμή της σταθεράς Γ ίση με το μέσο όρο του tricluster: $\Gamma = m_{GSC}$, η συνθήκη για την ένταξη ενός στοιχείου m_{gsc} σε κάποιο tricluster γίνεται:

$$\begin{aligned} m_{gsc} &= \Gamma + \alpha_g + \beta_s + \eta_c = \\ &= m_{gSC} + m_{GsC} + m_{GSc} - 2m_{GSC} \end{aligned} \qquad \textbf{(21)}$$

Το υπόλοιπο (residue) του κάθε στοιχείου, r_{gsc}, προκύπτει αν αφαιρέσουμε από την πραγματική τιμή του m_{gsc} την προσδοκώμενη από τον τύπο **(21)**. Έτσι, το Μέσο Τετραγωνικό Υπόλοιπο (Mean Squared Residue, MSR) που όρισαν αρχικά οι Y. Cheng και G. M. Church για το δ-Biclustering γίνεται τώρα:

$$\begin{aligned} S &= \frac{1}{|G'||S'||C'|} \sum_{g\in G', s\in S', c\in C'} r_{gsc}^2 = \\ &= \frac{1}{|G'||S'||C'|} \sum_{g\in G', s\in S', c\in C'} \left(m_{gsc} - m_{gSC} - m_{GsC} - m_{GSc} + 2m_{GSC}\right)^2 \end{aligned} \quad \textbf{(22)}$$

Ο αλγόριθμος που περιγράφει τη διαδικασία εξεύρεσης των triclusters δίνεται με τον ακόλουθο ψευδοκώδικα:

Αλγόριθμος δ-TRIMAX

```
Είσοδος:
    Ο G x S x C πίνακας δεδομένων, A
    Η παράμετρος πολλαπλών διαγραφών, λ > 1
    Το μέγιστο αποδεκτό MSR, δ ≥ 0

Έξοδος:
    Όλα τα δ-triclusters του A

Μέθοδος:
    δ-triclusters = Ø
    Επανάλαβε {
        B = Πολλαπλή_Διαγραφή_Κόμβων(A, δ, λ)
        C = Απλή_Διαγραφή_Κόμβων(B, δ)
        D = Προσθήκη_Κόμβων(C)
        δ-triclusters = δ-triclusters U D
        Αντικατάσταση των στοιχείων του A που βρίσκονται
        και στον D με τυχαίους αριθμούς
     } Μέχρις ότου να μην προστίθεται άλλο γονίδιο σε
       δ-tricluster
```

Πολλαπλή_Διαγραφή_Κόμβων

Είσοδος:

Ο G x S x C πίνακας δεδομένων, M

Η παράμετρος πολλαπλών διαγραφών, λ > 1

Το μέγιστο αποδεκτό MSR, δ ≥ 0

Έξοδος:

Ένα δ-tricluster M' που περιέχει υποσύνολο γονιδίων G', υποσύνολο δειγμάτων S' και υποσύνολο ζευγών χρωμ/των C' με MSR όχι μεγαλύτερο του δ

Μέθοδος:

M' = M, G' = G, S' = S, C' = C

Επανάλαβε {

Υπολόγισε το S

Αν S ≤ δ{

Επίστρεψε το M'

}

Αν #γονιδίων > 100 {

Διάγραψε τα γονίδια g∈G' για τα οποία ισχύει:

$$\frac{1}{|S'||C'|}\sum_{s\in S',c\in C'}\left(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC}\right)^2 > \lambda S$$

Υπολόγισε το S

}

Αν #δειγμάτων > 100 {

Διάγραψε τα δείγματα s∈S' για τα οποία ισχύει:

$$\frac{1}{|G'||C'|}\sum_{g\in G',c\in C'}\left(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC}\right)^2 > \lambda S$$

Υπολόγισε το S

}

Αν #ζευγών χρωμ/των > 100 {

Διάγραψε τα ζεύγη χρωμ/των c∈C' για τα οποία ισχύει:

$$\frac{1}{|G'||S'|}\sum_{g\in G',s\in S'}\left(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC}\right)^2 > \lambda S$$

Υπολόγισε το S

```
        }
    } Μέχρι να μη γίνει καμία νέα διαγραφή
    Επίστρεψε το M'
```

Απλή_Διαγραφή_Κόμβων

Είσοδος:

Ο G x S x C πίνακας δεδομένων, M

Το μέγιστο αποδεκτό MSR, δ ≥ 0

Έξοδος:

Ένα δ-tricluster M' που περιέχει υποσύνολο γονιδίων G', υποσύνολο δειγμάτων S' και υποσύνολο ζευγών χρωμ/των C' με MSR όχι μεγαλύτερο του δ

Μέθοδος:

M' = M, G' = G, S' = S, C' = C

Υπολόγισε το S

Όσο S > δ επανάλαβε {

Βρες το γονίδιο g∈G' με το μέγιστο:

$$\mu(g) = \frac{1}{|S'||C'|}\sum_{s\in S', c\in C'}(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC})^2$$

Βρες το δείγμα s∈S' με το μέγιστο:

$$\mu(s) = \frac{1}{|G'||C'|}\sum_{g\in G', c\in C'}(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC})^2$$

Βρες το ζεύγος χρωμ/των c∈C' με το μέγιστο:

$$\mu(c) = \frac{1}{|G'||S'|}\sum_{g\in G', s\in S'}(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC})^2$$

Διάγραψε το γονίδιο, το δείγμα ή το ζεύγος χρωμ/των με το μεγαλύτερο μ.

Υπολόγισε το S

}

Επίστρεψε το M'

Προσθήκη_Κόμβων

Είσοδος:

Ο G x S x C πίνακας δεδομένων, M

Έξοδος:

Ένα δ-tricluster M' που περιέχει υποσύνολο γονιδίων G', υποσύνολο δειγμάτων S' και υποσύνολο ζευγών χρωμ/των C', τέτοια ώστε G ⊂ G', S ⊂ S', C ⊂ C' και το MSR του M' ≤ το MSR του M

Μέθοδος:

M' = M

Επανάλαβε {

G = G', S = S', C = C'

Υπολόγισε το S

Πρόσθεσε τα γονίδια g∉G για τα οποία ισχύει:

$$\frac{1}{|S||C|}\sum_{s\in S,c\in C}\left(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC}\right)^2 < S$$

ώστε να προκύψει το G'

Υπολόγισε το S

Πρόσθεσε τα δείγματα s∉S για τα οποία ισχύει:

$$\frac{1}{|G||C|}\sum_{g\in G,c\in C}\left(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC}\right)^2 < S$$

ώστε να προκύψει το S'

Υπολόγισε το S

Πρόσθεσε τα ζεύγη χρωμ/των c∉C για τα οποία ισχύει:

$$\frac{1}{|G||S|}\sum_{g\in G,s\in S}\left(m_{gsc}-m_{gSC}-m_{GsC}-m_{GSc}+2m_{GSC}\right)^2 < S$$

ώστε να προκύψει το C'

Υπολόγισε το S

} Μέχρι να μη γίνει καμία νέα προσθήκη

Επίστρεψε το M'

Πολυπλοκότητα: O(GSC), όπου G ο αριθμός των γονιδίων, S ο αριθμός των δειγμάτων και C ο αριθμός των ζευγών χρωμ/των.

Μία ακόμη τροποποίηση που προτείνουμε στον παραπάνω αλγόριθμο είναι η προσθήκη nan αντί τυχαίων τιμών σε κάθε νέα αναζήτηση συστάδων. Με τον τρόπο αυτό αποφεύγουμε την πιθανή επαναχρησιμοποίηση των κελιών αυτών σε επόμενα δ-triclusters.

Επίσης, για τις παραμέτρους *δ* και *λ* καθώς και για τα κατώφλια πολλαπλής διαγραφής κόμβων ισχύουν τα ίδια με τον αλγόριθμο δ-Biclustering (εδώ η παράμετρος *λ* αντιστοιχεί στην παράμετρο *α* του δ-Biclustering).

Δ. ΕΦΑΡΜΟΓΗ

Δ.1) Δεδομενα που Χρησιμοποιησαμε

Πλατφόρμες

Για τους σκοπούς της παρούσας μελέτης χρησιμοποιήθηκαν δύο διακριτά σύνολα δεδομένων με ασθενείς που πάσχουν από καρκίνο της ουροδόχου κύστης σε διάφορα στάδια της νόσου:

- Με χρήση της πλατφόρμας CodeLink αναλύθηκαν 10 δείγματα από παθολογικούς και 5 δείγματα ελέγχου από υγιείς ιστούς, τα οποία αποκτήθηκαν μέσω χειρουργικής επέμβασης σε ασθενείς του Ασκληπιείου Νοσοκομείου Αθηνών [47].

- Έγινε ακόμη χρήση του συνόλου δεδομένων GSE7476 [48], το οποίο διατίθεται δημόσια στη διαδικτυακή βάση του National Center for Biotechnology Information (NCBI), και περιέχει 9 δείγματα από παθολογικούς και 3 δείγματα ελέγχου από υγιείς ιστούς, η μελέτη των οποίων έγινε μέσω της πλατφόρμας Affymetrix.

Διευκρινίζεται ότι σε όλα τα σχήματα που ακολουθούν, για λόγους εξοικονόμησης χώρου και καλύτερης οπτικοποίησης, παραλείπουμε τα ακριβή ονόματα των δειγμάτων και αντ' αυτών τα αριθμούμε από το 0 έως το 26. Οι αριθμοί αυτοί αντιστοιχούν στα δείγματα: 8B_control, 11B_control, 25B_control, 26B_control, 29B_control, 2A_T1-gr3, 3A_T2-gr3, 4A_T1-gr2, 10A_T1_gr3, 16A_T1-gr2, 17A_T1-gr2, 22A_T2-gr3, 26A_T2-gr3, 27A_T1-gr3 και 29A_T2-gr3 του CodeLink και στα: GSM180991_Normal, GSM180992_Normal, GSM180993_Normal, GSM180994_Ta low grade, GSM180995_Ta low grade, GSM180996_Ta low grade, GSM180997_T1 high grade, GSM180998_T1 high grade, GSM180999_T1 high grade, GSM181000_T2, T3 or T4 (high grade), GSM181001_T2, T3 or T4 (high grade) και GSM181002_T2, T3 or T4 (high grade) του GSE7476.

Προετοιμασία δεδομένων

Εξαιτίας διάφορων ατελειών στην κατασκευή των μικροσυστοιχιών DNA, αλλά και παρεμβολών κατά την εκτέλεση της πειραματικής διαδικασίας, καθίσταται απαραίτητη η πραγματοποίηση κατάλληλης προεπεξεργασίας των δεδομένων ώστε να απομακρυνθεί κάθε είδος θορύβου. Ενδεικτικά αναφέρουμε τα βήματα που ακολουθήθηκαν από το ερευνητικό προσωπικό του Χωρέμειου Ερευνητικού Εργαστηρίου Νοσοκομείου Παίδων «Αγία Σοφία» κατά την προετοιμασία των CodeLink δεδομένων. Αντίστοιχες ενέργειες έγιναν και για το σύνολο GSE7476 από τους δημιουργούς του.

Με χρήση, λοιπόν, του επίσημου πακέτου R της CodeLink πραγματοποιήθηκαν τα εξής:

- **Φιλτράρισμα και επιδιόρθωση φόντου**

  Για το φιλτράρισμα των δεδομένων έγινε χρήση του κατωφλίου:

  $$S < B_L + 1.5\sigma_{B_L}$$

  όπου S: η μετρούμενη ένταση σήματος

  B_L: η ένταση του φόντου τοπικά

  σ_{B_L}: η τυπική απόκλιση της έντασης του φόντου τοπικά

  Τα spots που έδιναν ένταση μικρότερη αυτού του κατωφλίου σημαδεύτηκαν με ειδική επισήμανση (X), ώστε να απορριφθούν κατά την ανάλυση.

  Η επιδιόρθωση του φόντου έγινε με αφαίρεση του καθολικού μέσου της έντασης φόντου από τον τοπικό μέσο. Δηλαδή για κάθε spot έχουμε:

  $$B_L = B_L - B_G$$

  όπου B_G: ο μέσος της ένταση φόντου σε όλη τη μικροσυστοιχία

- **Κανονικοποίηση των δεδομένων**

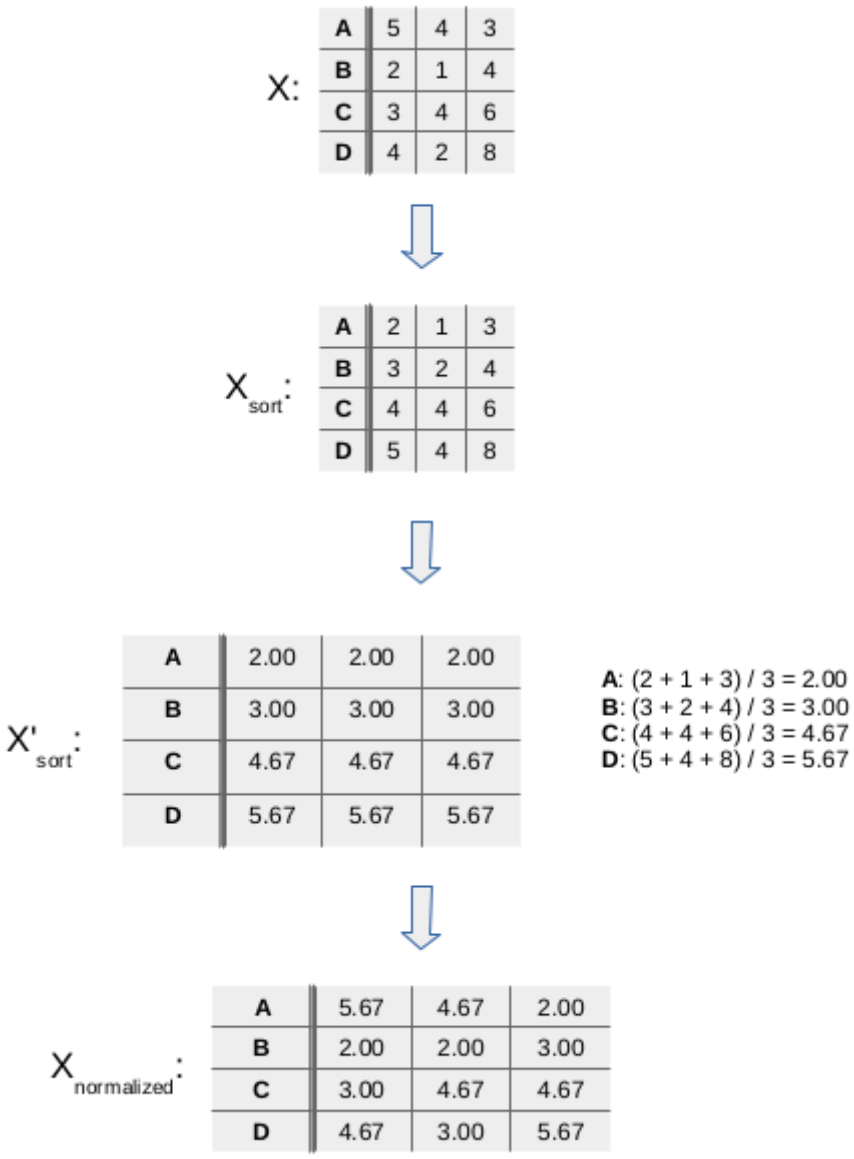
  Έπειτα έγινε κανονικοποίηση των δεδομένων με χρήση της προεπιλεγμένης μεθόδου στο πακέτο CodeLink [49], δηλαδή της *κανονικοποίησης ολικού μέσου* (*global median normalization*). Βάσει αυτής της μεθόδου η ένταση του σήματος σε κάθε spot διαιρείται με το μέσο (median) της έντασης σε όλη τη μικροσυστοιχία.

Στη συνέχεια πραγματοποιήθηκε ταυτόχρονη κανονικοποίηση των δεδομένων από όλες τις πλατφόρμες προκειμένου να απαλειφθούν οι διαφοροποιήσεις που προέκυψαν εξαιτίας της

πειραματικής διαδικασίας, των διαφορετικών μικροσυστοιχιών και της ποικιλομορφίας των δειγμάτων. Πιο συγκεκριμένα, προτιμήθηκε η εφαρμογή της *κανονικοποίησης ποσοστημορίων* (q*uantile normalization*) [50]–[52], η οποία σύμφωνα με διάφορες μελέτες [53], [54] παράγει παρόμοια ή ακόμη και καλύτερα αποτελέσματα συγκριτικά με άλλες αντίστοιχες μεθόδους όταν εφαρμόζεται σε δεδομένα γονιδιακής έκφρασης. Η μέθοδος αυτή προτάθηκε το 2001 από τον B. Bolstad και στόχος της είναι ο μετασχηματισμός των δεδομένων ώστε να χαρακτηρίζονται από τον ίδιο μέσο όρο και την ίδια διασπορά. Ο αλγόριθμος που ακολουθείται είναι ο εξής:

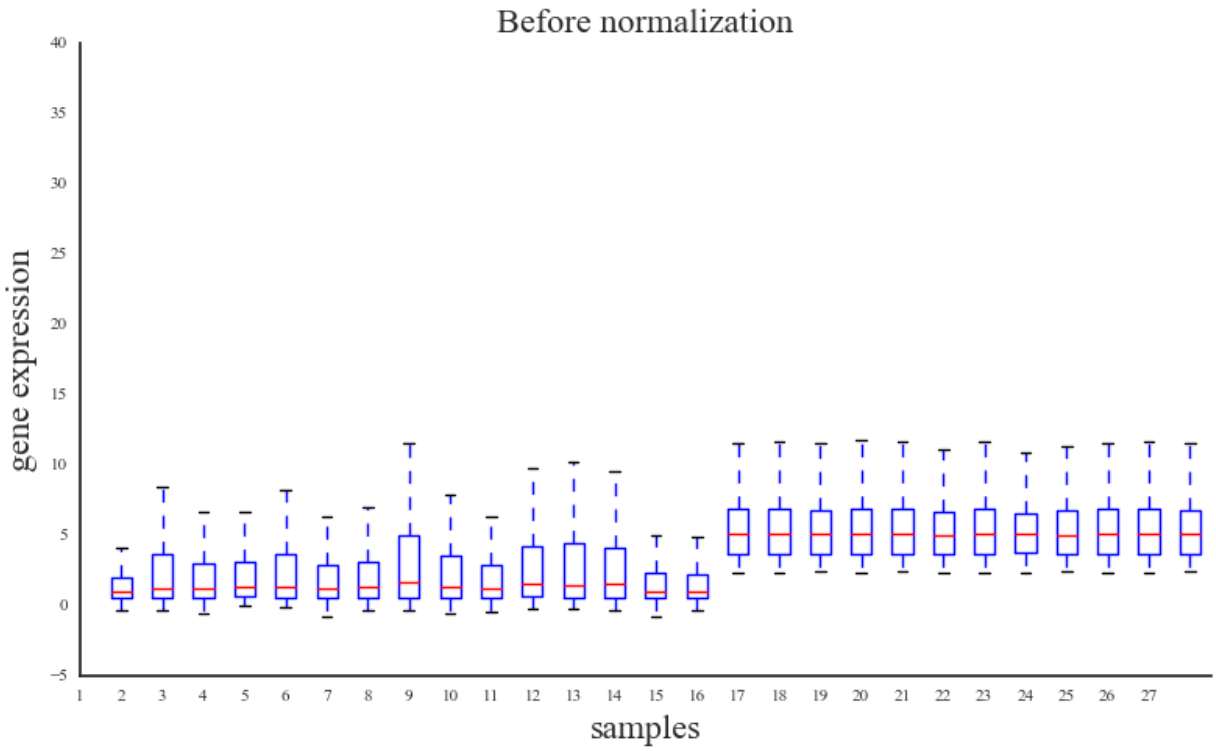
1. Έστω ότι έχουμε G διανύσματα μήκους S. Σχηματίζουμε το σύνολο δεδομένων X διαστάσεων G x S, όπου κάθε διάνυσμα αποτελεί μια στήλη.
2. Ταξινομούμε κάθε στήλη του πίνακα X κατά αύξουσα σειρά ώστε να προκύψει ο πίνακας X_{sort}.
3. Υπολογίζουμε το μέσο όρο κάθε γραμμής του X_{sort} και αντιστοιχίζουμε αυτή την τιμή σε κάθε τιμή της γραμμής ώστε να προκύψει ο πίνακας X'_{sort}.
4. Δημιουργούμε τον κανονικοποιημένο πίνακα $X_{normalized}$ μεταβάλλοντας τη σειρά των στοιχείων κάθε στήλης του X'_{sort} σύμφωνα με τη σειρά των στοιχείων του X.

Στο Σχήμα 19 φαίνεται ένα παράδειγμα της κανονικοποίησης ποσοστημορίων σε έναν πίνακα διαστάσεων 4 x 3 με ακέραιες τιμές.

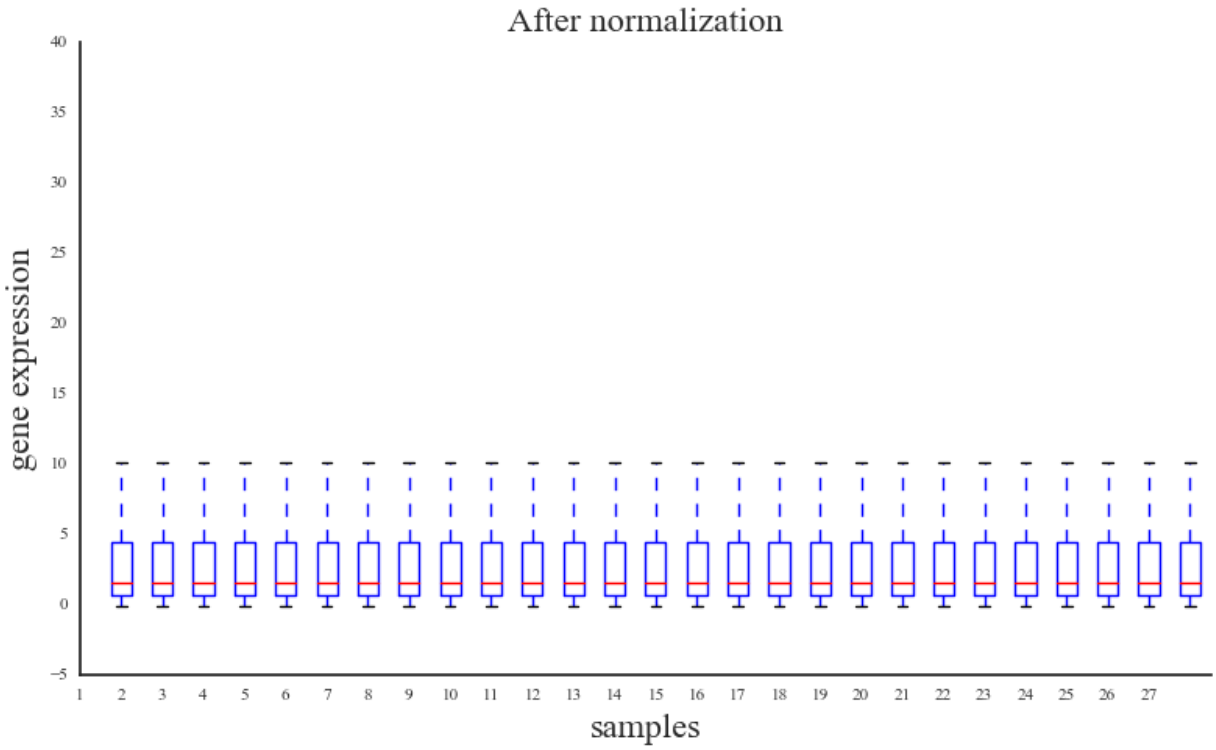


Σχήμα 19: Παράδειγμα κανονικοποίησης ποσοστημορίων

Στα παρακάτω σχήματα φαίνεται η επίδραση της κανονικοποίησης ποσοστημορίων στα δεδομένα μας. Στους οριζόντιους άξονες βρίσκονται τα δείγματα και στους κατακόρυφους η γονιδιακή έκφραση.



Σχήμα 20: Box plot των δεδομένων πριν από την κανονικοποίηση ποσοστημορίων.



Σχήμα 21: Box plot των δεδομένων μετά την κανονικοποίηση ποσοστημορίων.

Έπειτα από την κανονικοποίηση έγινε χρήση της μεθόδου *k Nearest Neighbors (k-NN) imputation* [55] ώστε να αντιμετωπιστεί η απουσία τιμών στο σύνολο δεδομένων. Είναι σύνηθες κατά την εκτέλεση πειραμάτων με DNA μικροσυστοιχίες να απουσιάζουν κάποιες τιμές, είτε λόγω χαμηλής ανάλυσης της εικόνας, είτε λόγω αστοχίας των υλικών της πειραματικής διάταξης (σκόνη, γραντζουνιές, κλπ). Το γεγονός αυτό επηρεάζει άμεσα τη στατιστική ανάλυση και πολλές φορές καθίσταται αδύνατη η χρήση αλγορίθμων όπως ο k-Means, οι οποίοι απαιτούν ένα πλήρες σύνολο δεδομένων χωρίς απουσία τιμών. Σύμφωνα με ποικίλες μελέτες [56], [57], στην περίπτωση των DNA μικροσυστοιχιών συνίσταται η τεχνική *k-NN imputation*, διότι κατορθώνει να διατηρήσει μια σταθερή απόδοση όσο μεγάλο κι αν είναι το ποσοστό των απόντων τιμών, είναι λιγότερο ευαίσθητη στο θόρυβο και στην τιμή της παραμέτρου *k*, ενώ παράλληλα επιτυγχάνει ακριβέστερη εκτίμηση σε δεδομένα γονιδιακής έκφρασης.

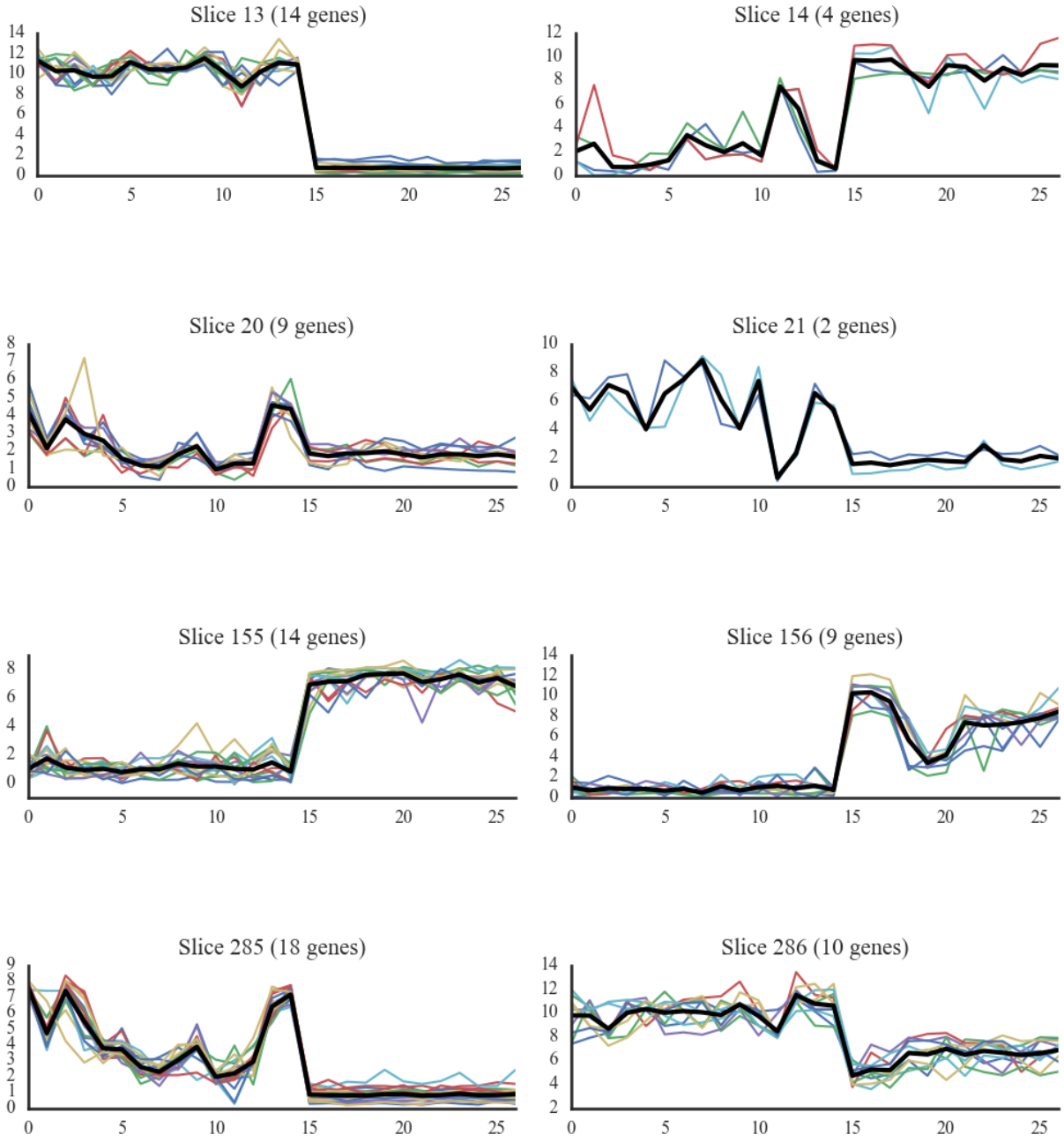
Ο αλγόριθμος που περιγράφει την παραπάνω μέθοδο είναι ο εξής:

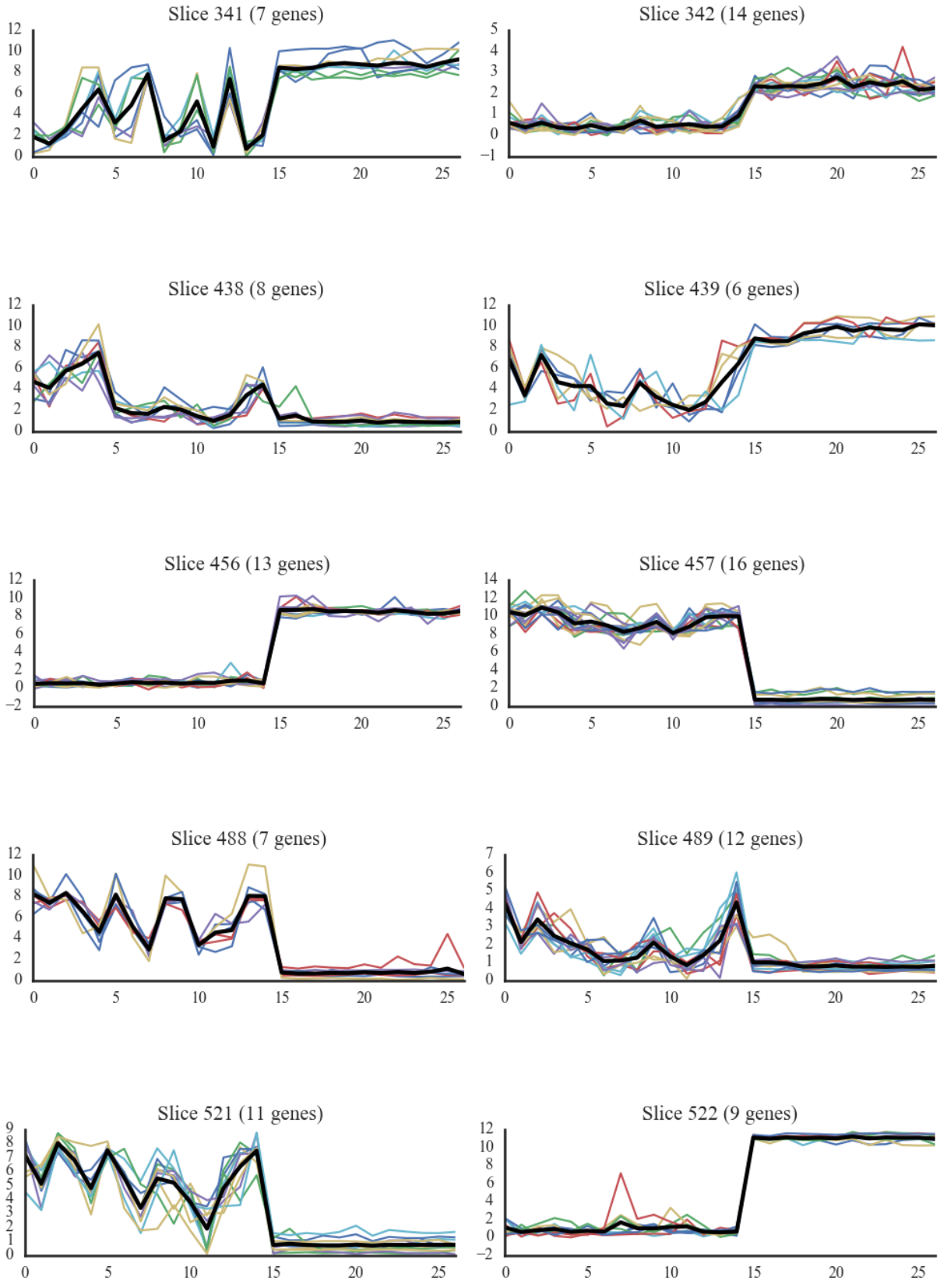
> Έστω ότι έχουμε το σύνολο δεδομένων *X*, διαστάσεων *G* x *S*. Έστω ακόμη ότι το γονίδιο *g* δεν έχει καμία τιμή στο δείγμα *s* (δηλ. το κελί x_{gs} είναι κενό).
>
> 1. Ορίζουμε το σύνολο *X** το οποίο περιέχει μόνο τα γονίδια εκείνα που έχουν πλήρως συμπληρωμένες όλες τις στήλες τους.
> 2. Με χρήση κάποιας μετρικής (προτείνεται η Ευκλείδεια απόσταση) βρίσκουμε τα *k* γονίδια του *X** (δηλ. τους *k* γείτονες) που μοιάζουν περισσότερο στο γονίδιο *g* στα υπόλοιπα *S*-1 δείγματα.
> 3. Υπολογίζουμε τη μέση τιμή των *k* γειτόνων στο δείγμα *s* και χρησιμοποιούμε αυτή την τιμή για να συμπληρώσουμε το κελί x_{gs}.
>
> Επαναλαμβάνουμε τα παραπάνω βήματα για κάθε απούσα τιμή στο *X*.

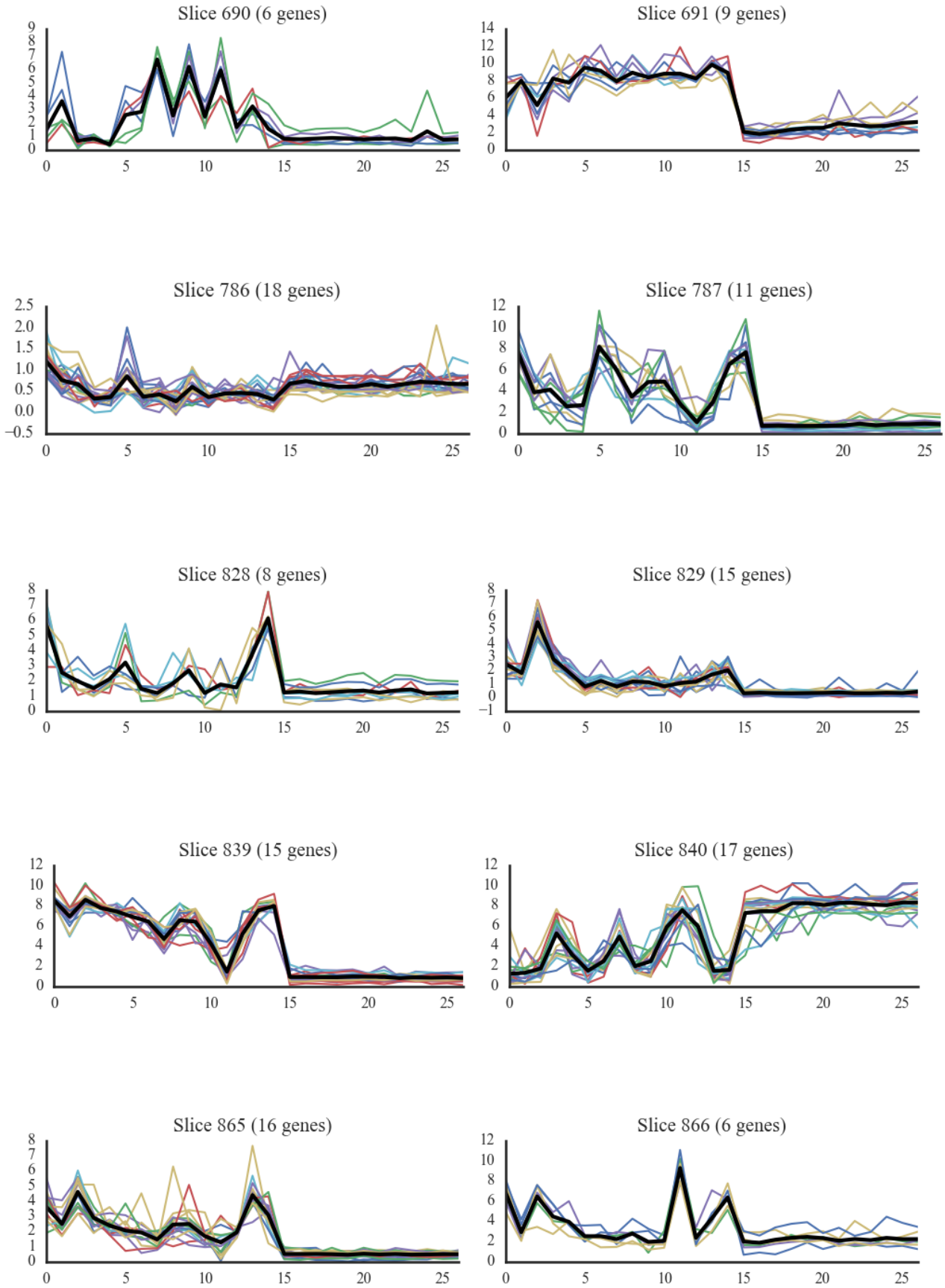
Πολλοί ερευνητές προειδοποιούν ότι σε δεδομένα DNA μικροσυστοιχιών η μέθοδος δεν παρουσιάζει μεγάλη ευαισθησία στην τιμή του *k*, οπότε και προτείνονται χαμηλές τιμές όπως $k = 1$ ή 2. Αξίζει να σημειωθεί ότι καθώς το *k* προσεγγίζει το συνολικό πλήθος των γονιδίων, *G*, ο αλγόριθμος ταυτίζεται με την τεχνική mean imputation, κατά την οποία στο κελί x_{gs} εκχωρείται η μέση τιμή όλων των γονιδίων στο συγκεκριμένο δείγμα, με αποτέλεσμα να συνεισφέρουν και γονίδια τα οποία παρουσιάζουν πολύ διαφορετικό προφίλ έκφρασης. Στην παρούσα εργασία θεωρήσαμε ότι θέτοντας $k = 1$ ο αλγόριθμος επιτυγχάνει αρκετά καλά αποτελέσματα.

Δ.2) Στάδιο Πρώτο: κατασκευη τρισδιαστατου πινακα

Στα επόμενα σχήματα δίνονται κάποιες τομές του τρισδιάστατου πίνακα κατά μήκος του άξονα των γονιδίων (με έντονη γραμμή είναι το κεντροειδές της τομής).




Slice 341 (7 genes)
Slice 342 (14 genes)
Slice 438 (8 genes)
Slice 439 (6 genes)
Slice 456 (13 genes)
Slice 457 (16 genes)
Slice 488 (7 genes)
Slice 489 (12 genes)
Slice 521 (11 genes)
Slice 522 (9 genes)



Σχήμα 22: Κάποιες τυχαίες τομές του τρισδιάστατου πίνακα. Στις παρενθέσεις αναγράφεται ο

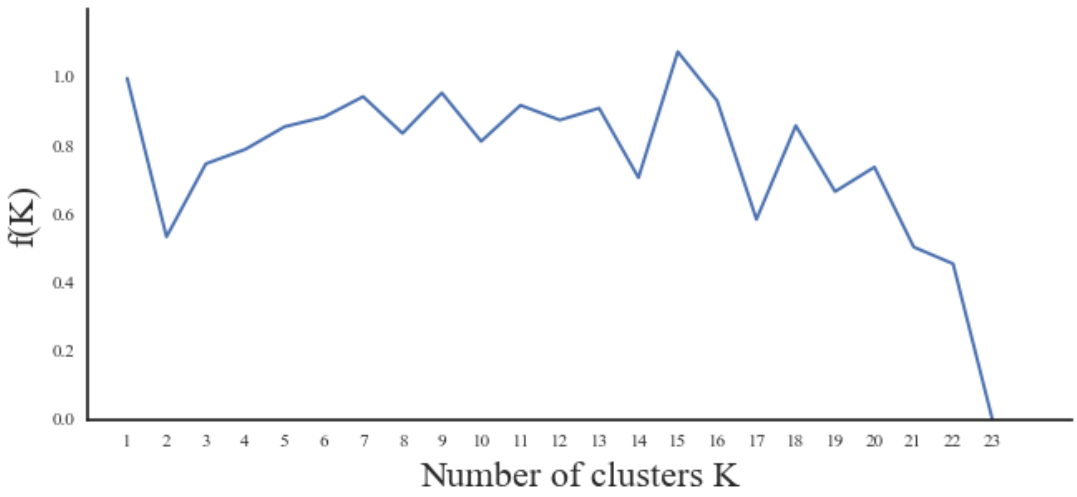
αριθμός των γονιδίων που ανήκουν στη συγκεκριμένη τομή. Στον κατακόρυφο άξονα απεικονίζεται η γονιδιακή έκφραση και στον οριζόντιο τα δείγματα.

Δ.3) Σταδιο Δευτερο: Συσταδοποιηση

Στην ενότητα αυτή παρουσιάζουμε τα αποτελέσματα από τις διάφορες μεθόδους συσταδοποίησης που εφαρμόστηκαν στον τρισδιάστατο πίνακα. Σε κάθε περίπτωση, παρατίθενται επιλεκτικά κάποια διαγράμματα των συστάδων, καθώς και τα αντίστοιχα κεντροειδή με κατάλληλες μπάρες σφάλματος (error bars) τυπικής απόκλισης [58].

Συγκεκριμένα, πριν την εφαρμογή των k-Means αλγορίθμων εκτελέσαμε τη μέθοδο $f(K)$ κι έπειτα χρησιμοποιήσαμε τις 5-10 καλύτερες τιμές που απέδωσε. Με μόνη εξαίρεση τον τρισδιάστατο k-Means στην τομή του άξονα γονιδίων, διότι ήταν αδύνατο να συγκλίνει για $k > 100$, οπότε και χρησιμοποιήσαμε ενδεικτικά τις τιμές $k = 80, 100$. Στα επόμενα σχήματα φαίνονται τα αποτελέσματα της μεθόδου $f(K)$ για κάθε τομή.

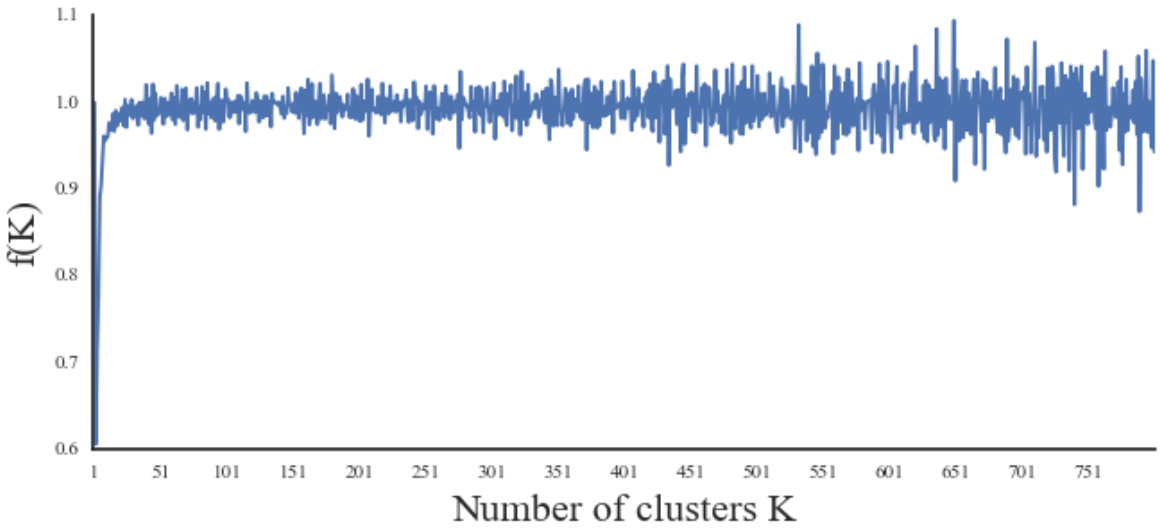
- <u>Τομή στον άξονα των χρωμοσωμάτων</u>



Σχήμα 23: Οι τιμές της μεθόδου f(K) για $1 \leq k \leq 23$

Στην περίπτωση αυτή, η μέθοδος $f(K)$ ανέδειξε τις τιμές $k = 23, 22, 21, 2, 17, 19, 14, 20, 3, 4, 10$.

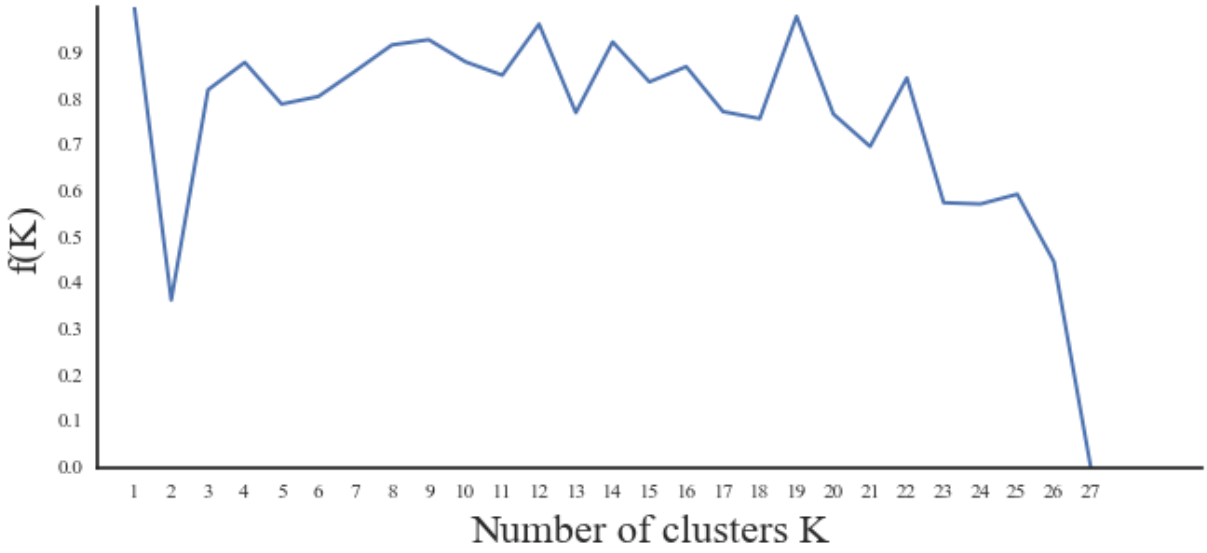
- Τομή στον άξονα των γονιδίων



Σχήμα 24: Οι τιμές της μεθόδου f(K) για 1 ≤ k ≤ 800

Η μέθοδος *f*(*K*) ανέδειξε τις τιμές *k* = 2, 3, 4, 789, 740, 5, 758, 6, 650, 726, 736, 746, 672, 762, 434 ως υποψήφιες για την ανακάλυψη υψηλής ποιότητας συσταδοποιήσεων.

- Τομή στον άξονα των δειγμάτων



Σχήμα 25: Οι τιμές της μεθόδου f(K) για 1 ≤ k ≤ 27

Σύμφωνα με το διάγραμμα, η μέθοδος *f*(*K*) ανέδειξε τις τιμές *k* = 27, 2, 26, 24, 23, 25, 21.

Τρισδιάστατος k-Means

Η πρώτη μας απόπειρα συσταδοποίησης αφορούσε την εφαρμογή της επέκτασης του κλασικού k-Means αλγορίθμου σε δεδομένα τριών διαστάσεων με νόρμες πινάκων. Μία συνάρτηση $\|o\| : M_n \to \mathbb{R}$ ονομάζεται νόρμα πινάκων (matrix norm) [59] αν για κάθε πίνακα A , $B \in M_n$ ικανοποιεί τις ακόλουθες ιδιότητες:

i. $\|A\| \geq 0$ (Μη αρνητική)

ii. $\|A\| = 0$ *αν και μόνο αν* $A = 0$ (Θετική)

iii. $\|cA\| = |c| \|A\|$, *για κάθε* $c \in \mathbb{C}$ (Ομογενής)

iv. $\|A + B\| \leqslant \|A\| + \|B\|$ (Τριγωνική ανισότητα)

v. $\|AB\| \leqslant \|A\| \, \|B\|$ (Υποπολλαπλασιαστικότητα)

Οι πιο γνωστές νόρμες πινάκων είναι οι λεγόμενες l_p-νόρμες, για p = 1, 2, ∞. Πιο συγκεκριμένα:

- **Η l_1-νόρμα ή πυρηνική νόρμα:**

$$\|A\|_1 = \sum_{i\,,\,j=1}^{n} |a_{ij}| \qquad \textbf{(23)}$$

- **Η l_2-νόρμα ή νόρμα Fröbenius:**

$$\|A\|_F = \left(\sum_{i\,,\,j=1}^{n} |a_{ij}|^2 \right)^{1/2} = \sqrt{tr\left(\bar{A}^T A\right)} \qquad \textbf{(24)}$$

όπου *tr(A)* το ίχνος του πίνακα *A*, δηλαδή το άθροισμα των στοιχείων της κύριας διαγωνίου του.

- **Η l_∞-νόρμα:**

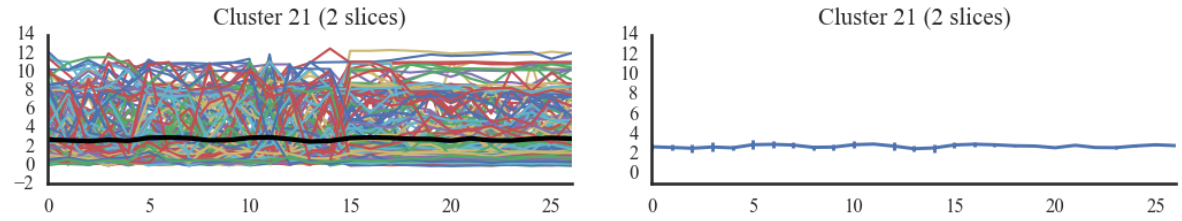
$$\|A\|_\infty = \max_{1 \leqslant i\,,\,j \leqslant n} |a_{ij}| \qquad \textbf{(25)}$$

Στην παραλλαγή του k-Means κάναμε διαδοχικά χρήση όλων των παραπάνω νορμών, χωρίς τελικά κάποια από αυτές να δίνει αισθητά καλύτερα αποτελέσματα. Στα παρακάτω σχήματα φαίνονται ενδεικτικά κάποιες συστάδες που δημιουργήθηκαν με τον τρισδιάστατο αυτό αλγόριθμο, εκτελώντας τον 15 φορές για κάθε τιμή του *k*, με μέγιστο αριθμό επαναλήψεων τις 50 σε κάθε εκτέλεση. Χρησιμοποιήσαμε τις βέλτιστες τιμές του *k* που

απέδωσε η μέθοδος *f*(*K*), και τη νόρμα Fröbenius ως μετρική, αλλά είναι εμφανές ότι οι τελικές συστάδες απέχουν πολύ από τη βέλτιστη λύση.

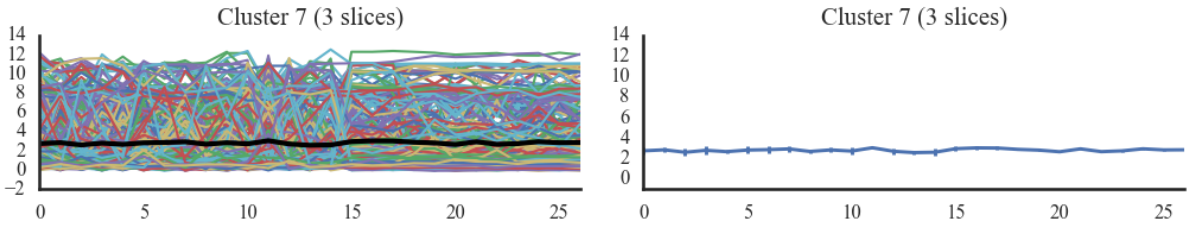
- Τομή στον άξονα των χρωμοσωμάτων

1. ***k* = 22**



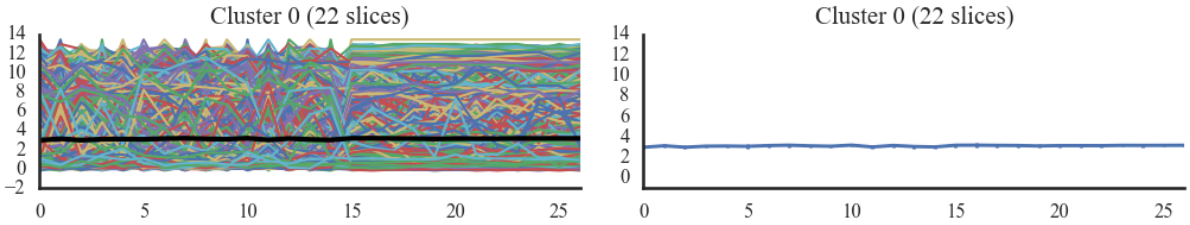
Σχήμα 26: Η σημαντική συστάδα που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 22.

2. ***k* = 21**



Σχήμα 27: Η σημαντική συστάδα που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 21.

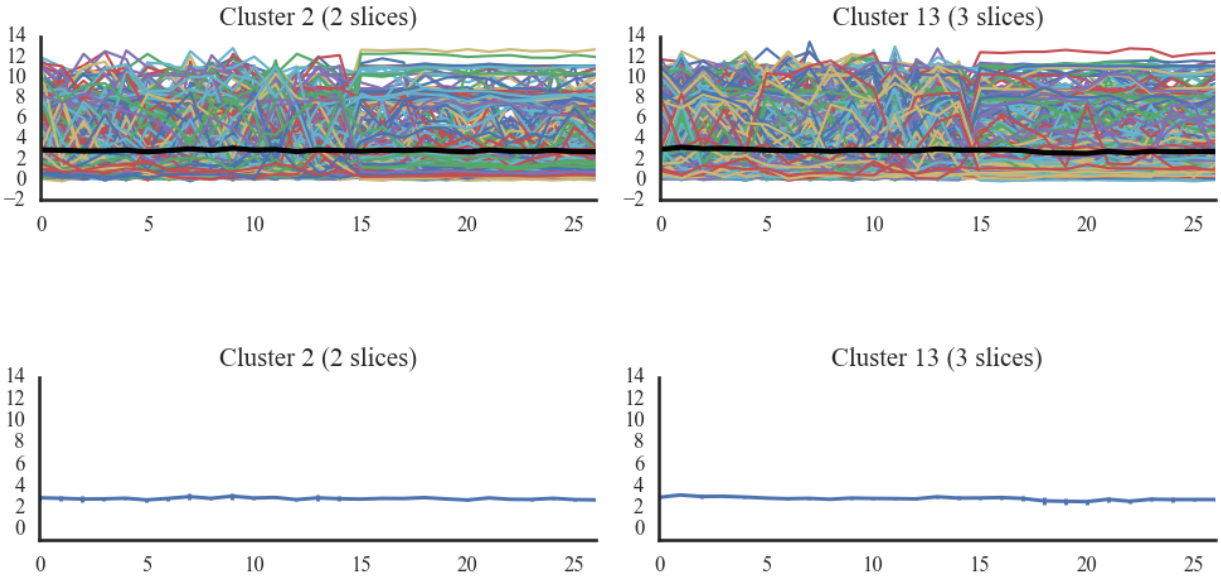
3. ***k* = 2**



Σχήμα 28: Η σημαντική συστάδα που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα

των χρωμοσωμάτων για k = 2.

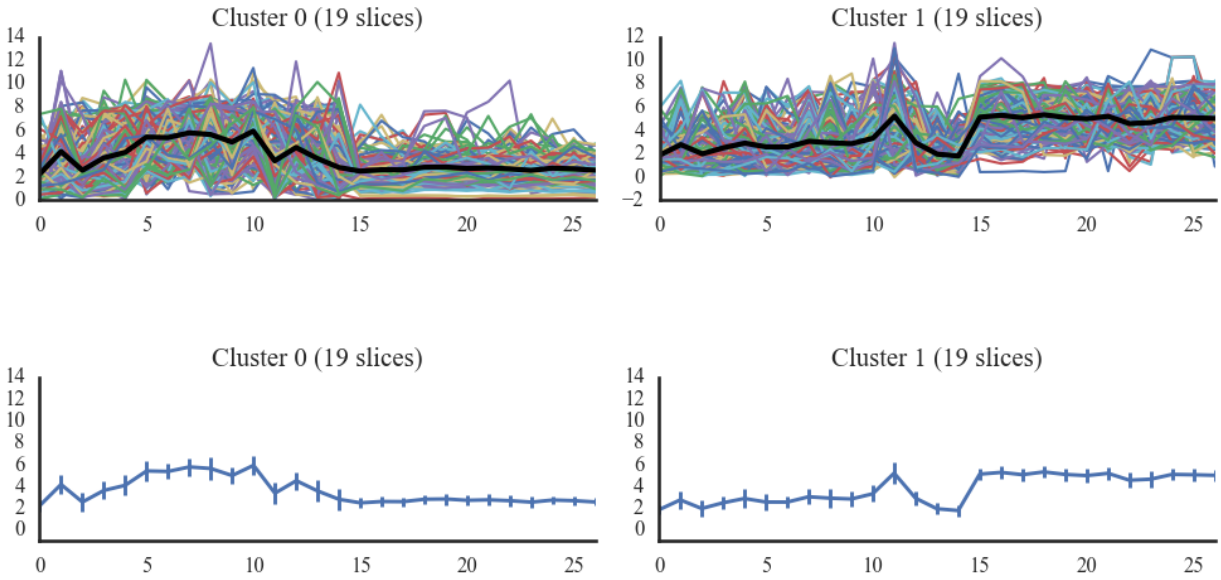
4. ***k* = 17**

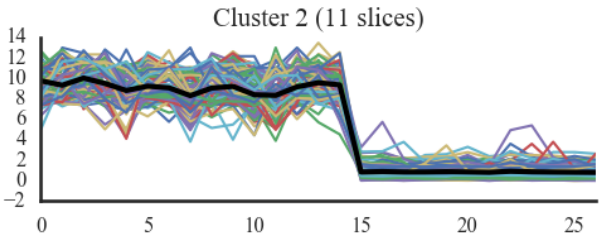


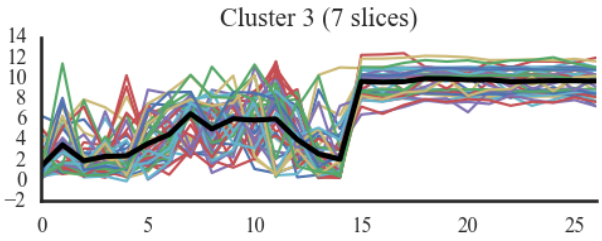
Σχήμα 29: Κάποιες σημαντικές συστάδες που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 17.

- <u>Τομή στον άξονα των γονιδίων</u>

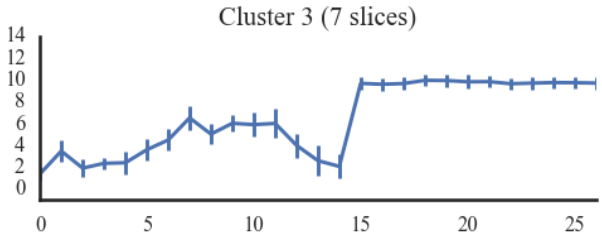
1. ***k* = 80**

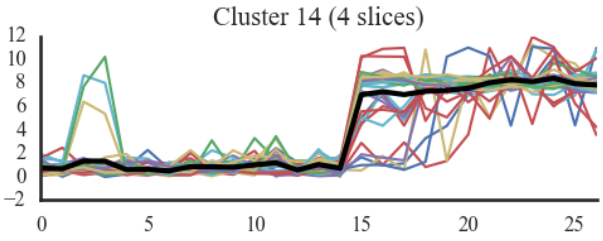


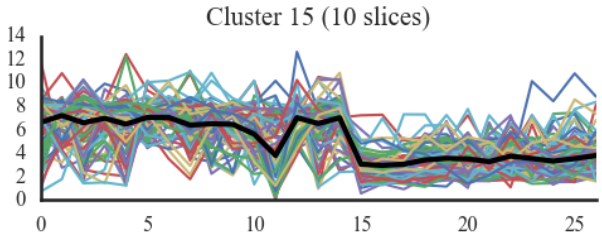

Cluster 2 (11 slices)

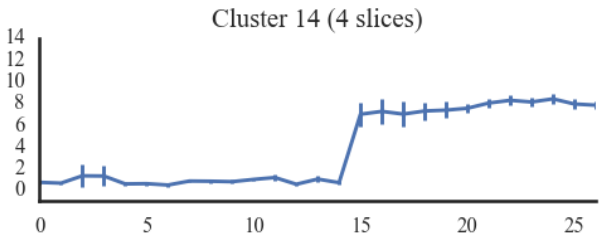

Cluster 3 (7 slices)

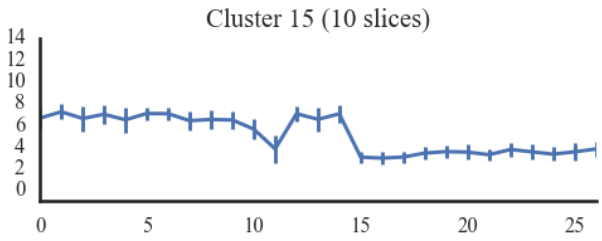

Cluster 2 (11 slices)

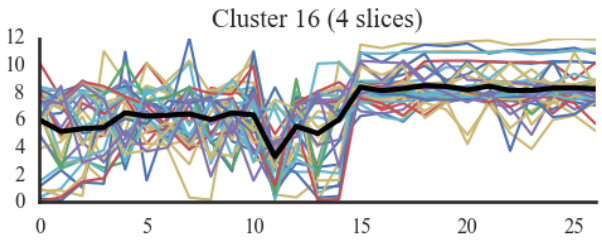

Cluster 3 (7 slices)

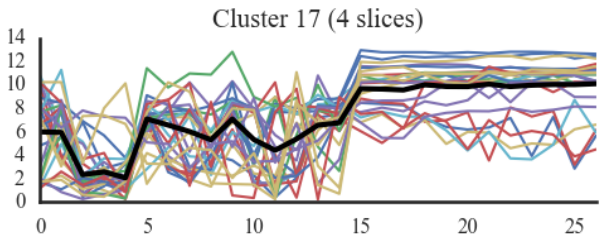

Cluster 14 (4 slices)

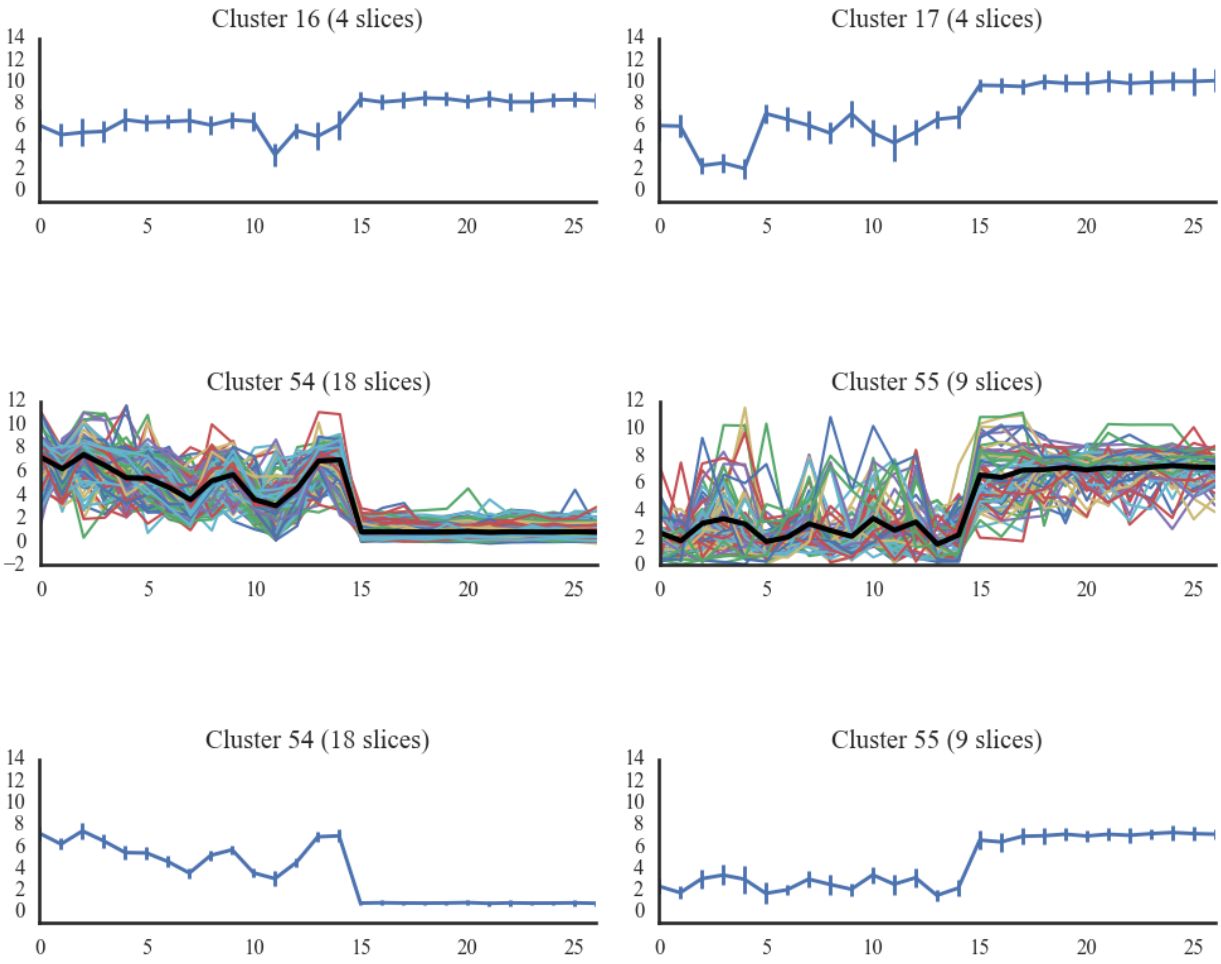

Cluster 15 (10 slices)


Cluster 14 (4 slices)


Cluster 15 (10 slices)

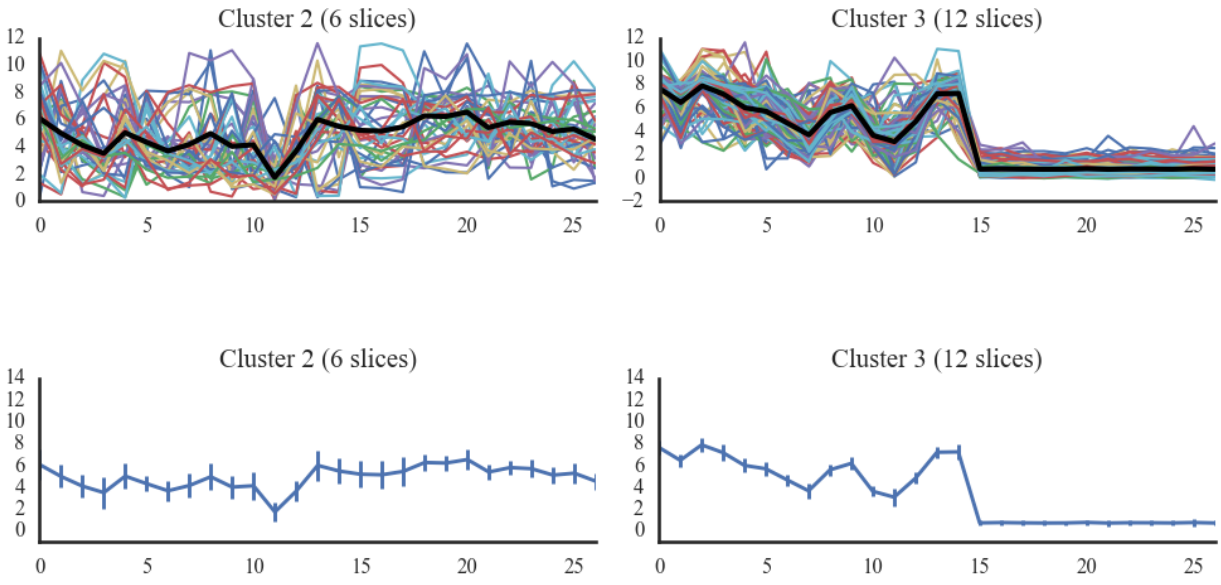

Cluster 16 (4 slices)

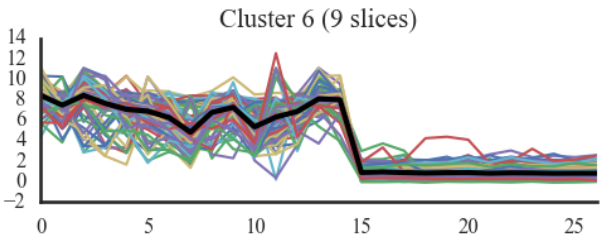

Cluster 17 (4 slices)

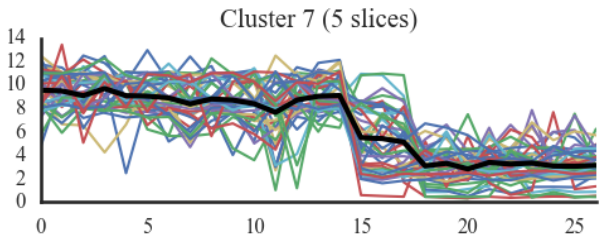


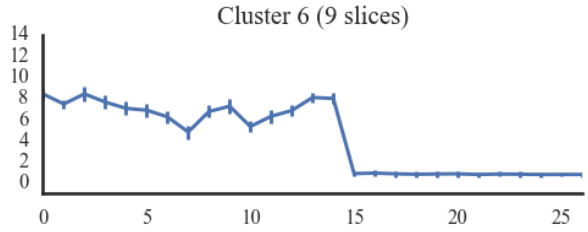
Σχήμα 30: Κάποιες σημαντικές συστάδες που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των γονιδίων για k = 80.

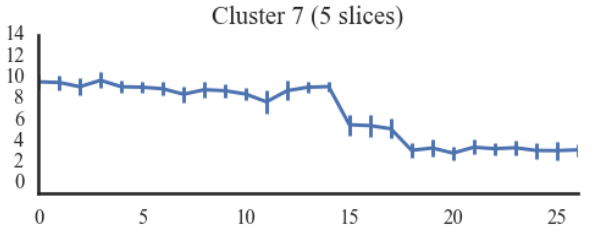
2. *k* = 100

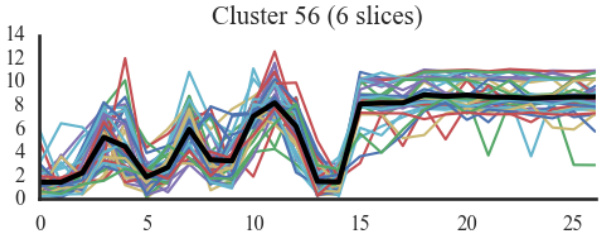


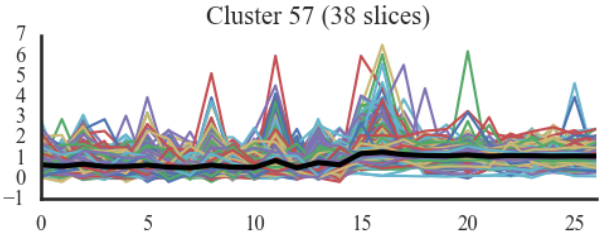

Cluster 6 (9 slices)

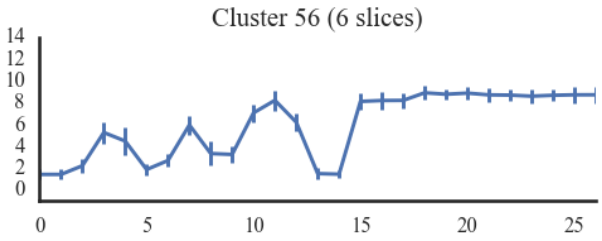

Cluster 7 (5 slices)

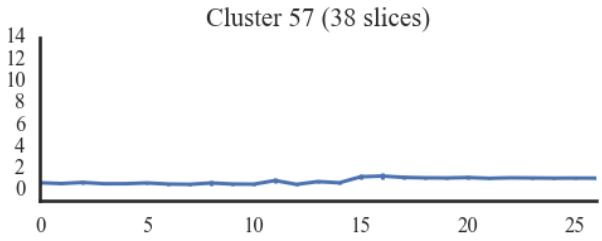

Cluster 6 (9 slices)

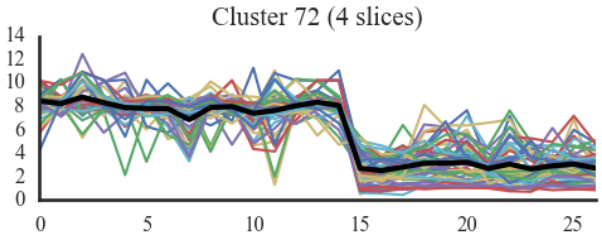

Cluster 7 (5 slices)

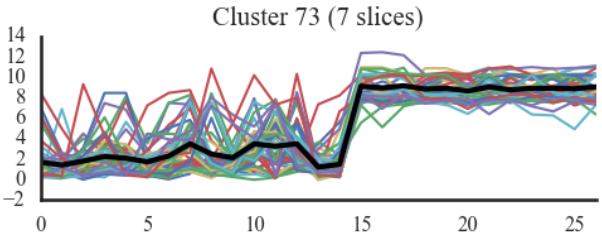

Cluster 56 (6 slices)

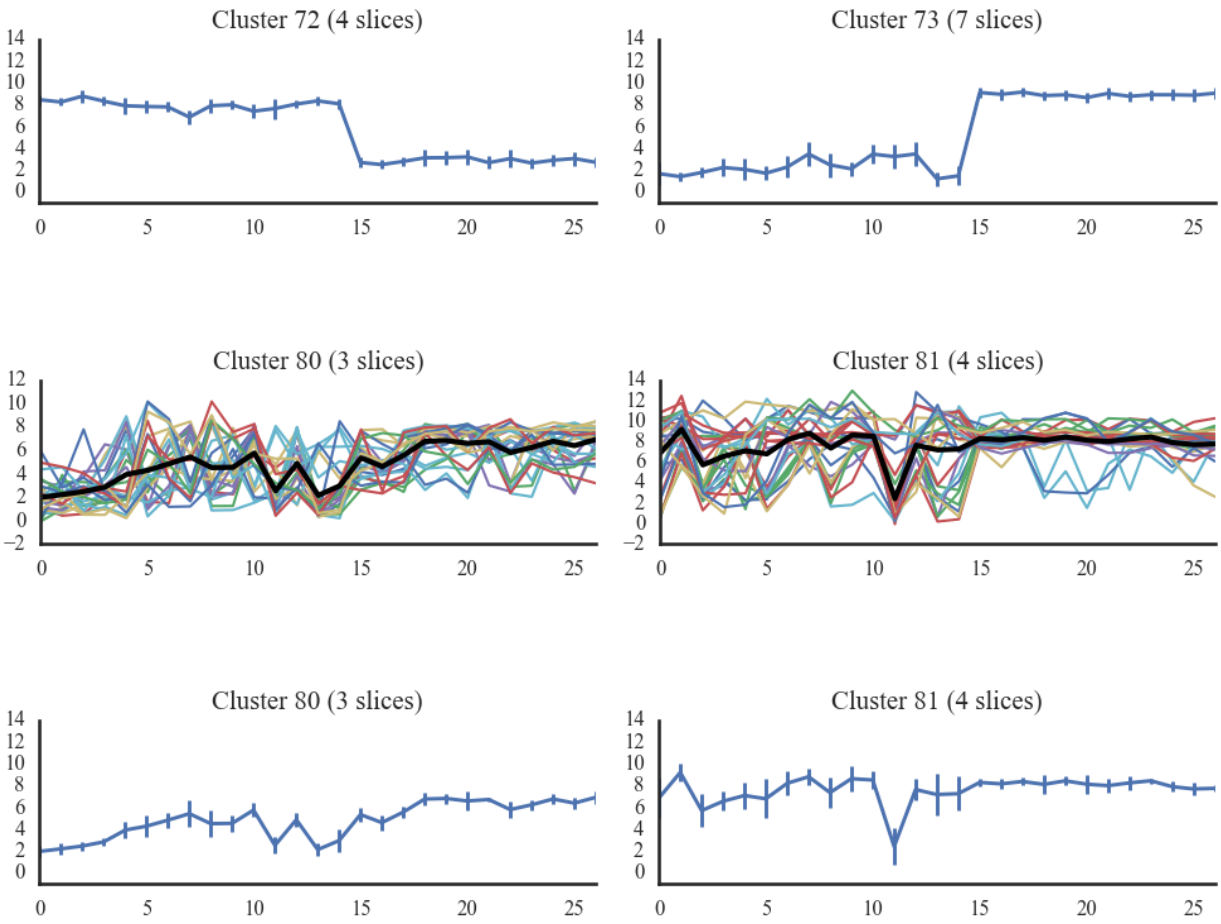

Cluster 57 (38 slices)


Cluster 56 (6 slices)


Cluster 57 (38 slices)


Cluster 72 (4 slices)

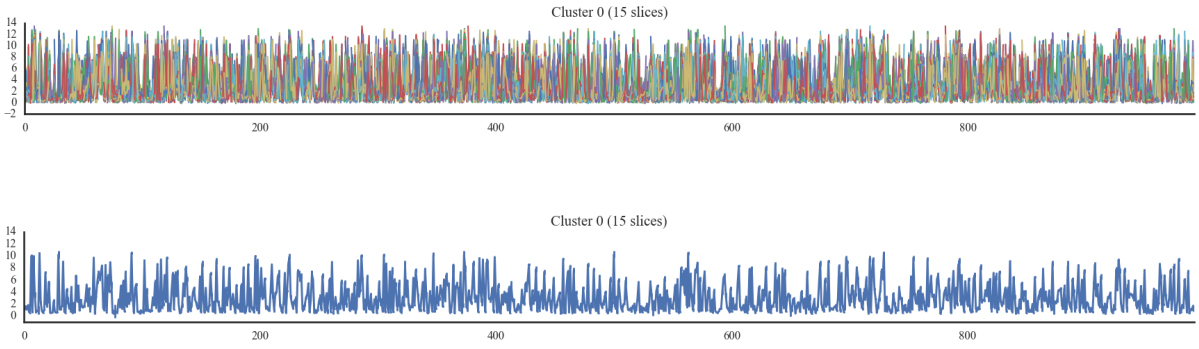

Cluster 73 (7 slices)

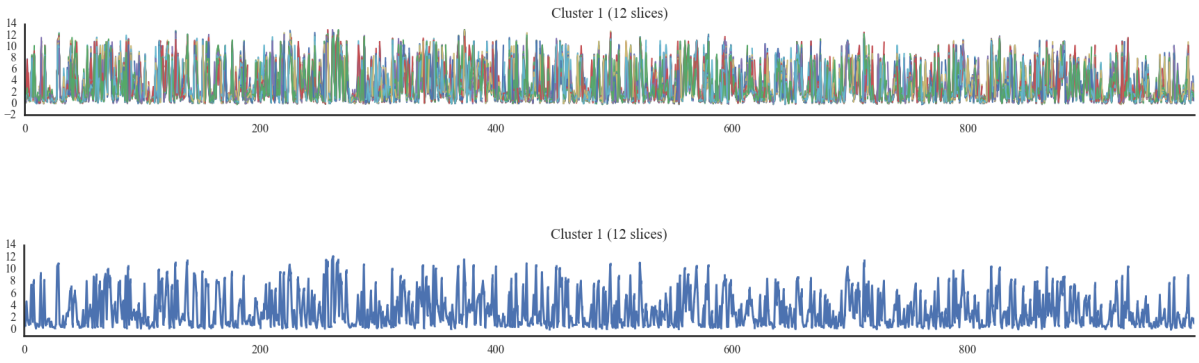


Σχήμα 31: Κάποιες σημαντικές συστάδες που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των γονιδίων για k = 100.

- <u>Τομή στον άξονα των δειγμάτων</u>

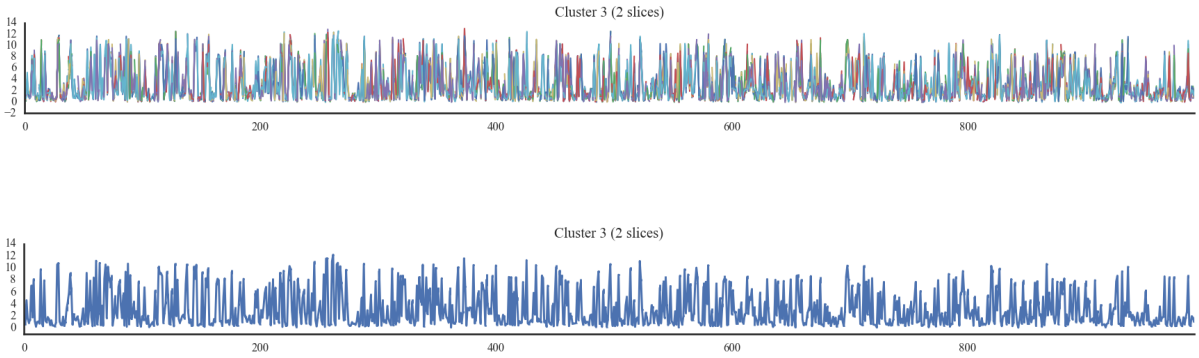
1. *k* = 2





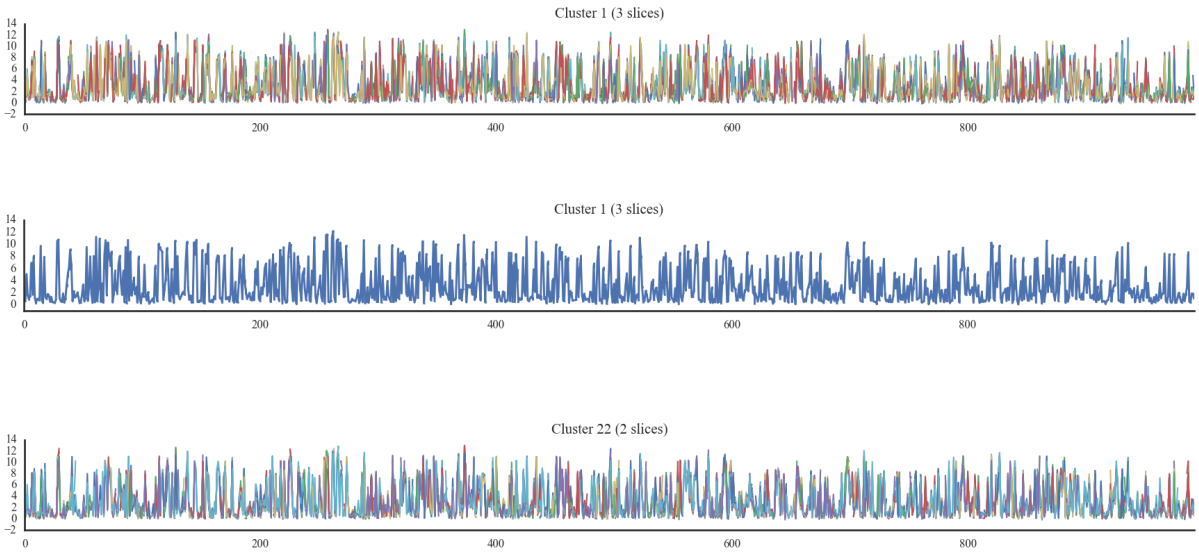
Σχήμα 32: Οι σημαντικές συστάδες που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των δειγμάτων για k = 2.

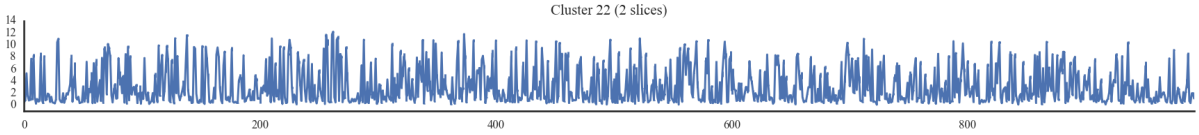
2. *k* = 26



Σχήμα 33: Η σημαντική συστάδα που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των δειγμάτων για k = 26.

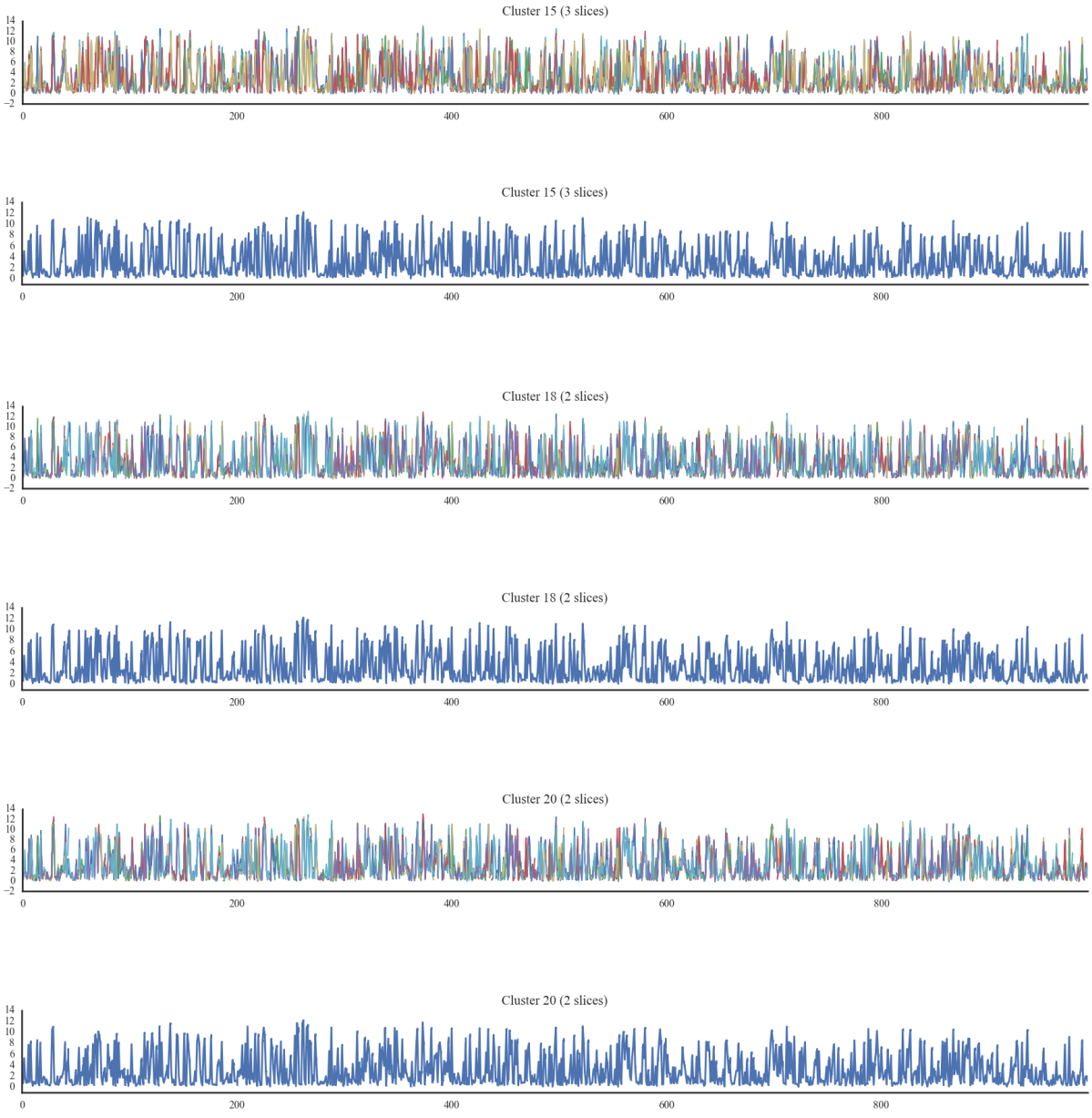
3. *k* = 24





Σχήμα 34: Οι σημαντικές συστάδες που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των δειγμάτων για k = 24.

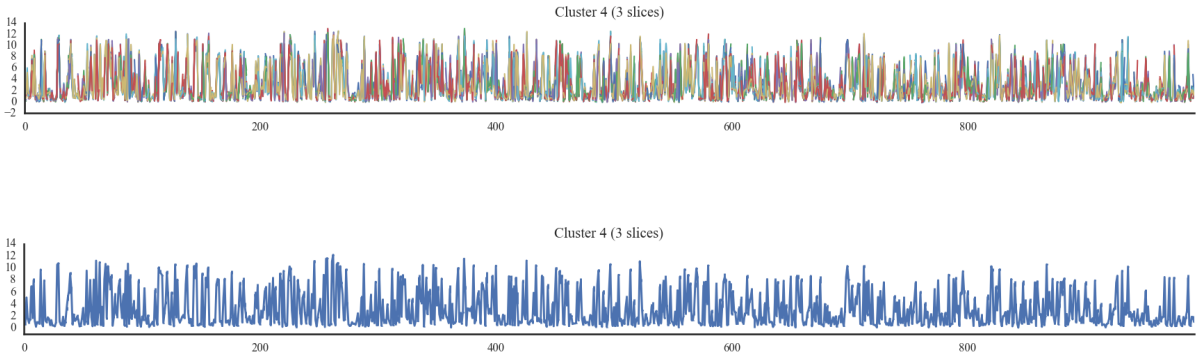
4. ***k* = 23**



Σχήμα 35: Οι σημαντικές συστάδες που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των δειγμάτων για k = 23

.

5. $k = 25$



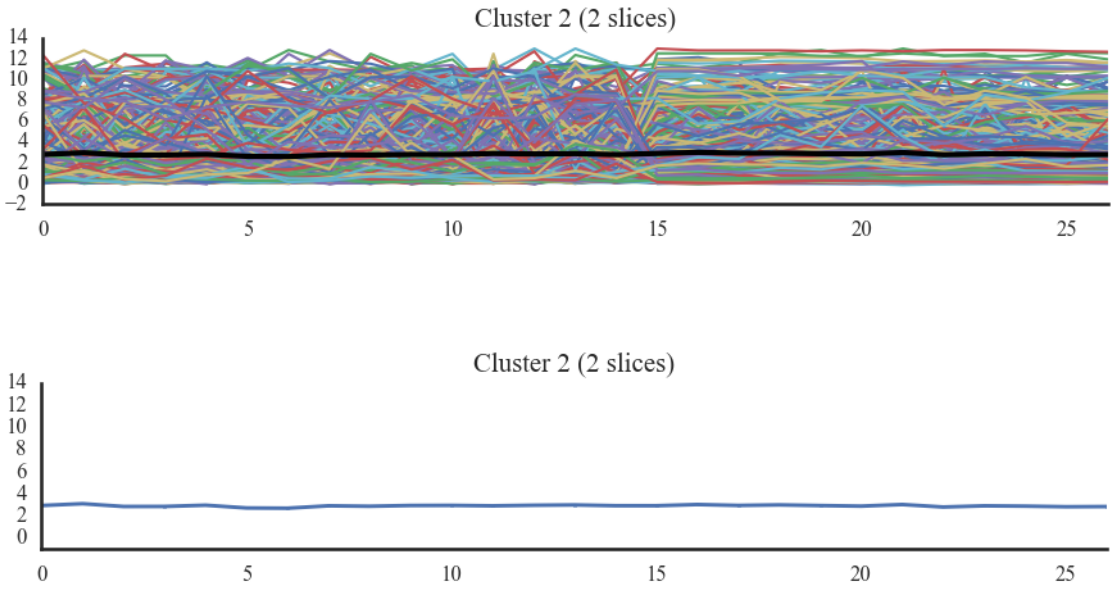
Σχήμα 36: Η σημαντική συστάδα που απέφερε ο τρισδιάστατος k-Means στην τομή του άξονα των δειγμάτων για k = 25.

Κλασικός k-Means στα κεντροειδή του πίνακα

Στη συνέχεια επιχειρήσαμε να εφαρμόσουμε τον κλασικό k-Means αλγόριθμο στα κεντροειδή της τρισδιάστατης δομής. Χρησιμοποιήσαμε την Ευκλείδεια απόσταση ως μετρική ομοιότητας και τον αλγόριθμο k-Means++ για αρχικοποίηση των κέντρων. Σε κάθε τιμή του *k*, εφαρμόσαμε τον k-Means 10 φορές, με μέγιστο αριθμό επαναλήψεων τις 300, και κρατήσαμε τη βέλτιστη λύση. Παρακάτω φαίνονται ενδεικτικά κάποιες από τις συστάδες που ανακαλύφθηκαν από τον αλγόριθμο.

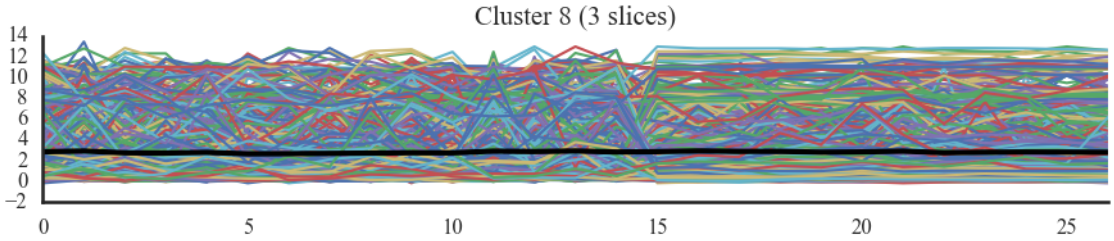
- Τομή στον άξονα των χρωμοσωμάτων

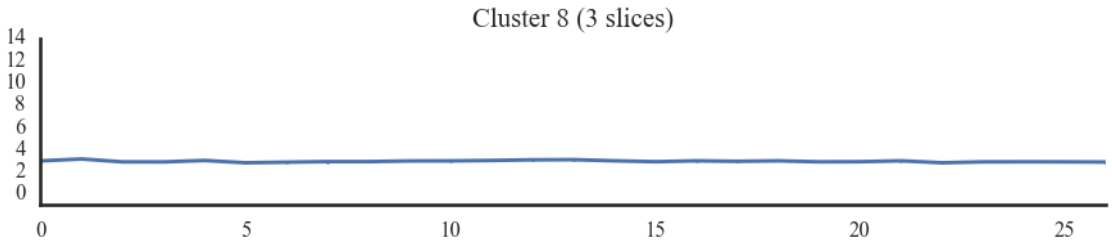
1. ***k* = 22**



Σχήμα 37: Η σημαντική συστάδα που απέφερε ο κλασικός k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 22.

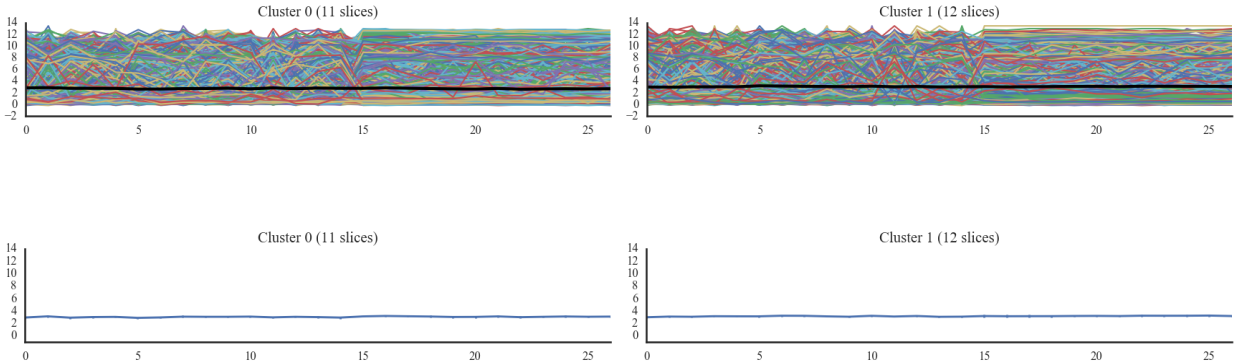
2. ***k* = 21**





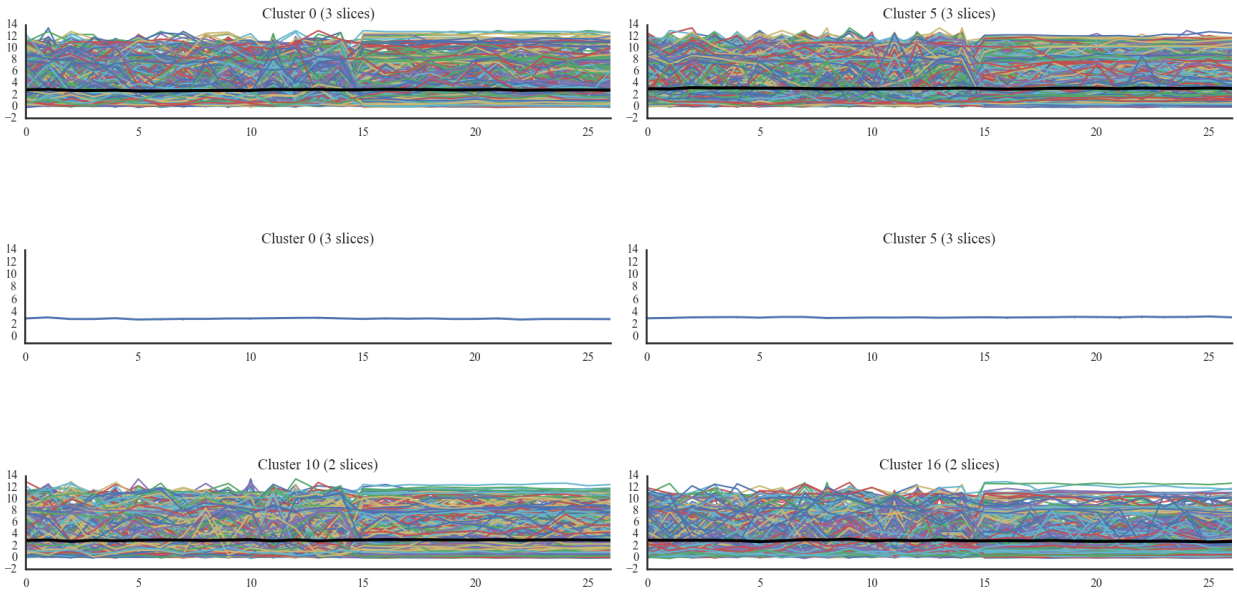
Σχήμα 38: Η σημαντική συστάδα που απέφερε ο κλασικός k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 21.

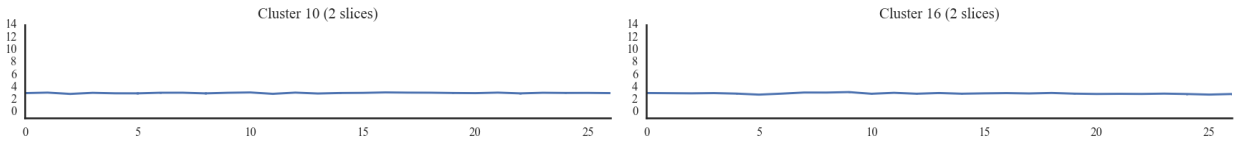
3. *k* = 2



Σχήμα 39: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 2.

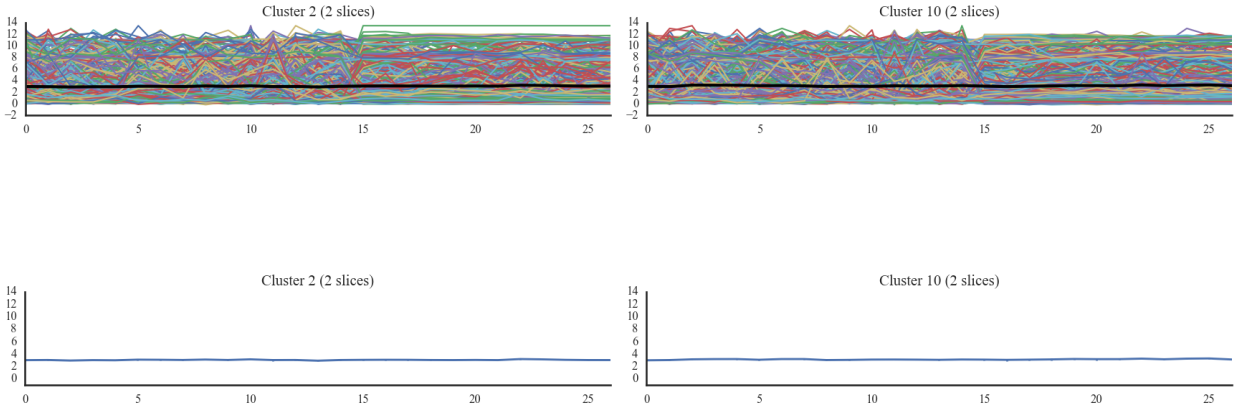
4. *k* = 17

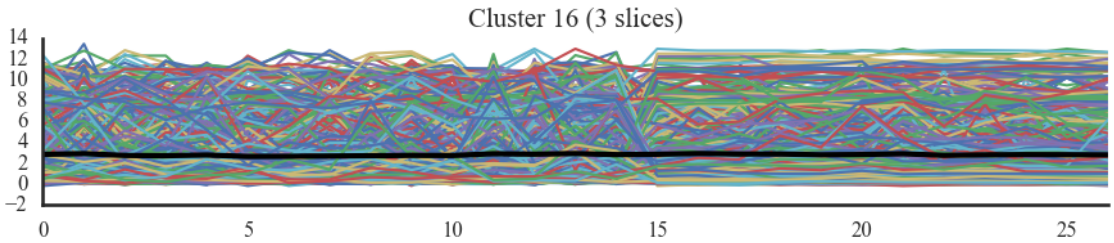


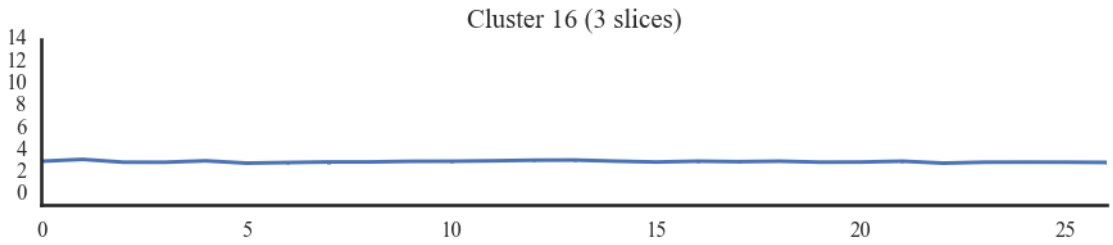


Σχήμα 40: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 17.

5. *k* = 19



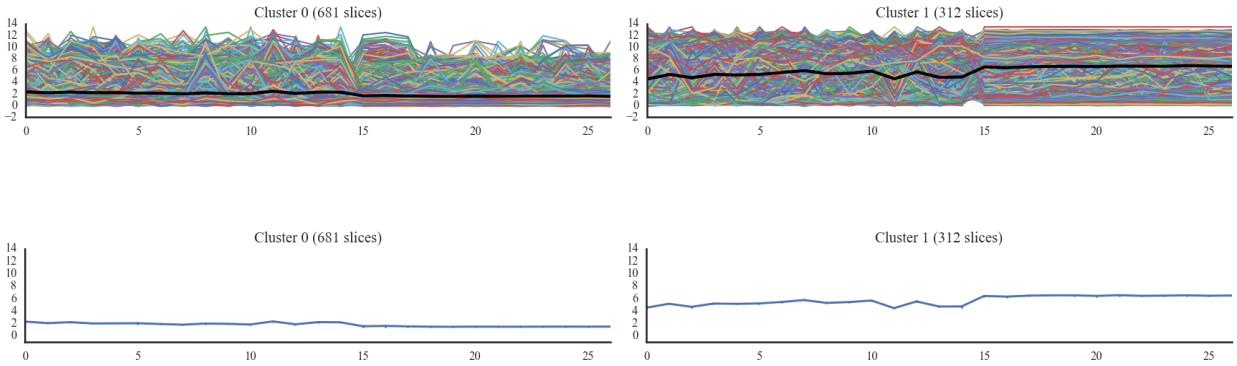




Σχήμα 41: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των χρωμοσωμάτων για k = 19.

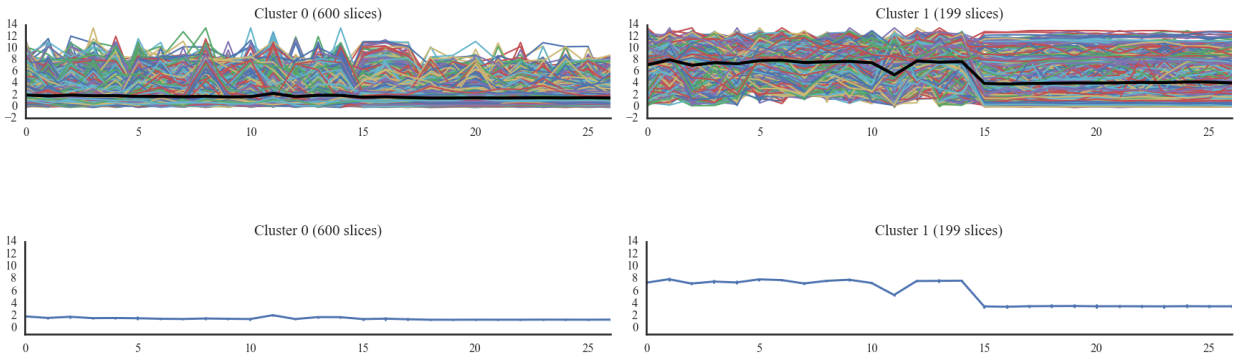
- <u>Τομή στον άξονα των γονιδίων</u>

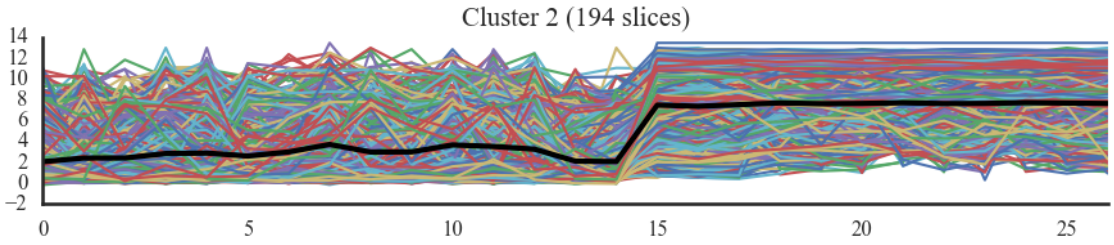
1. *k* = 2

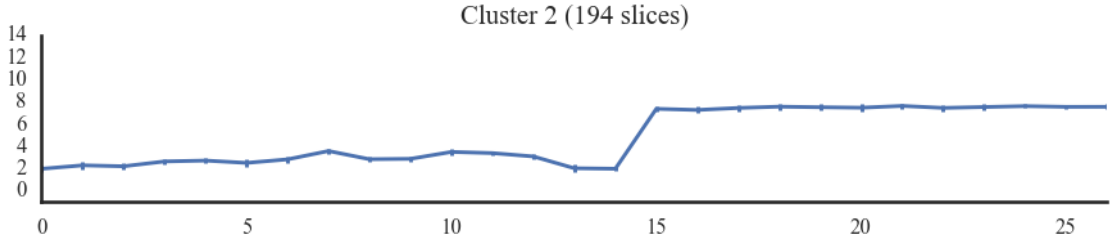


Σχήμα 42: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 2.

2. *k* = 3

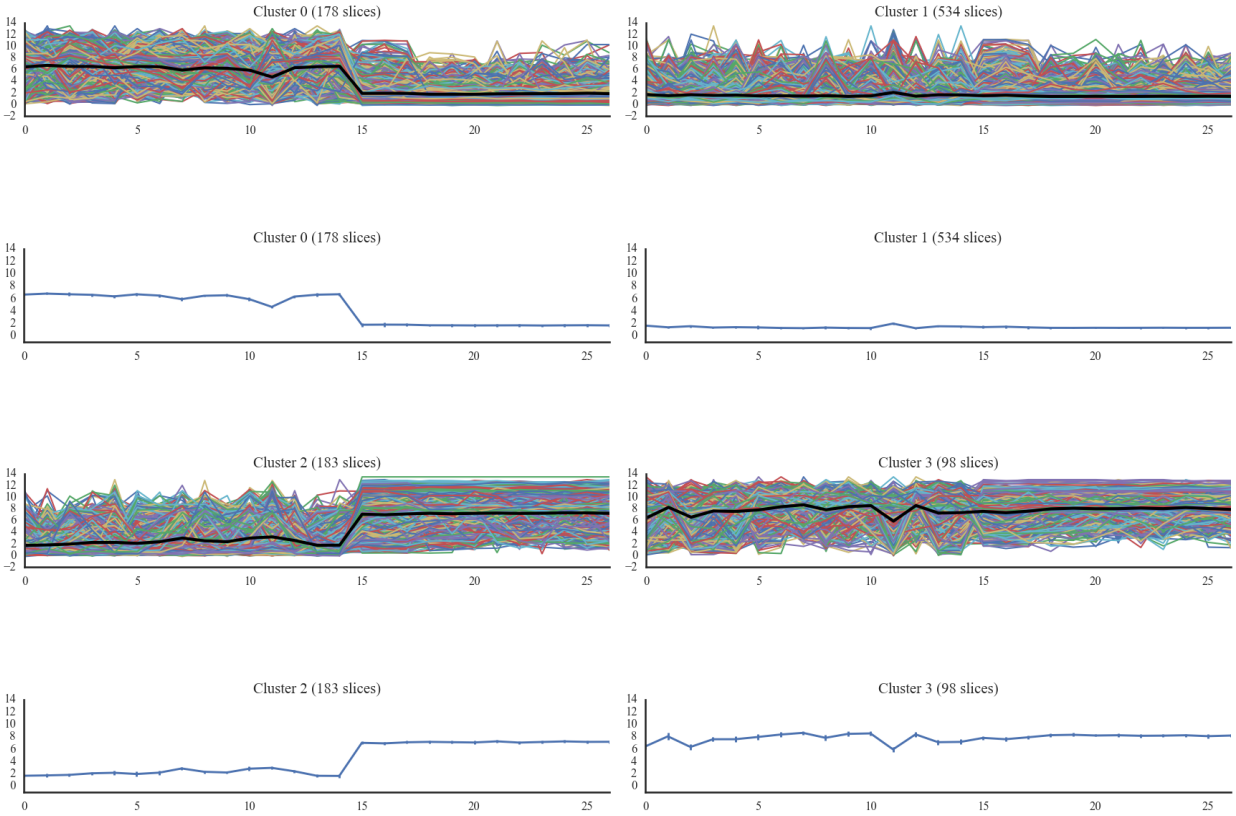






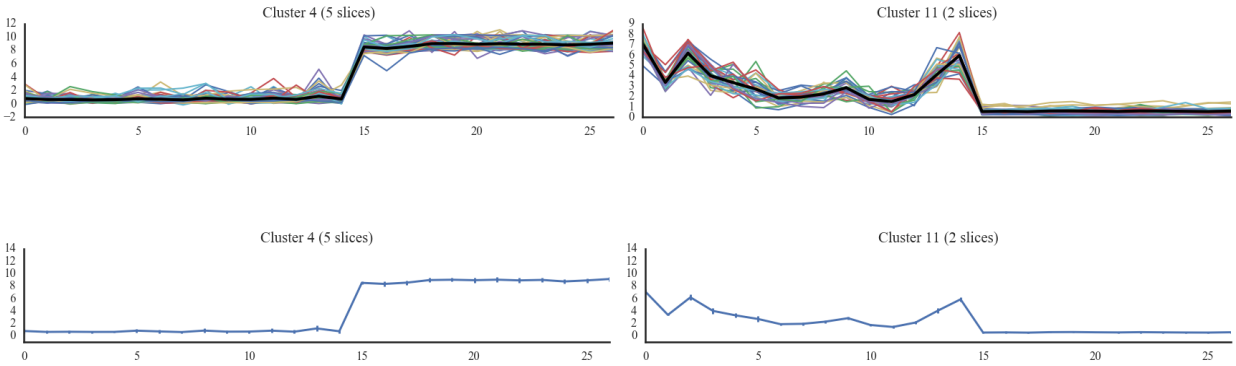
Σχήμα 43: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 3.

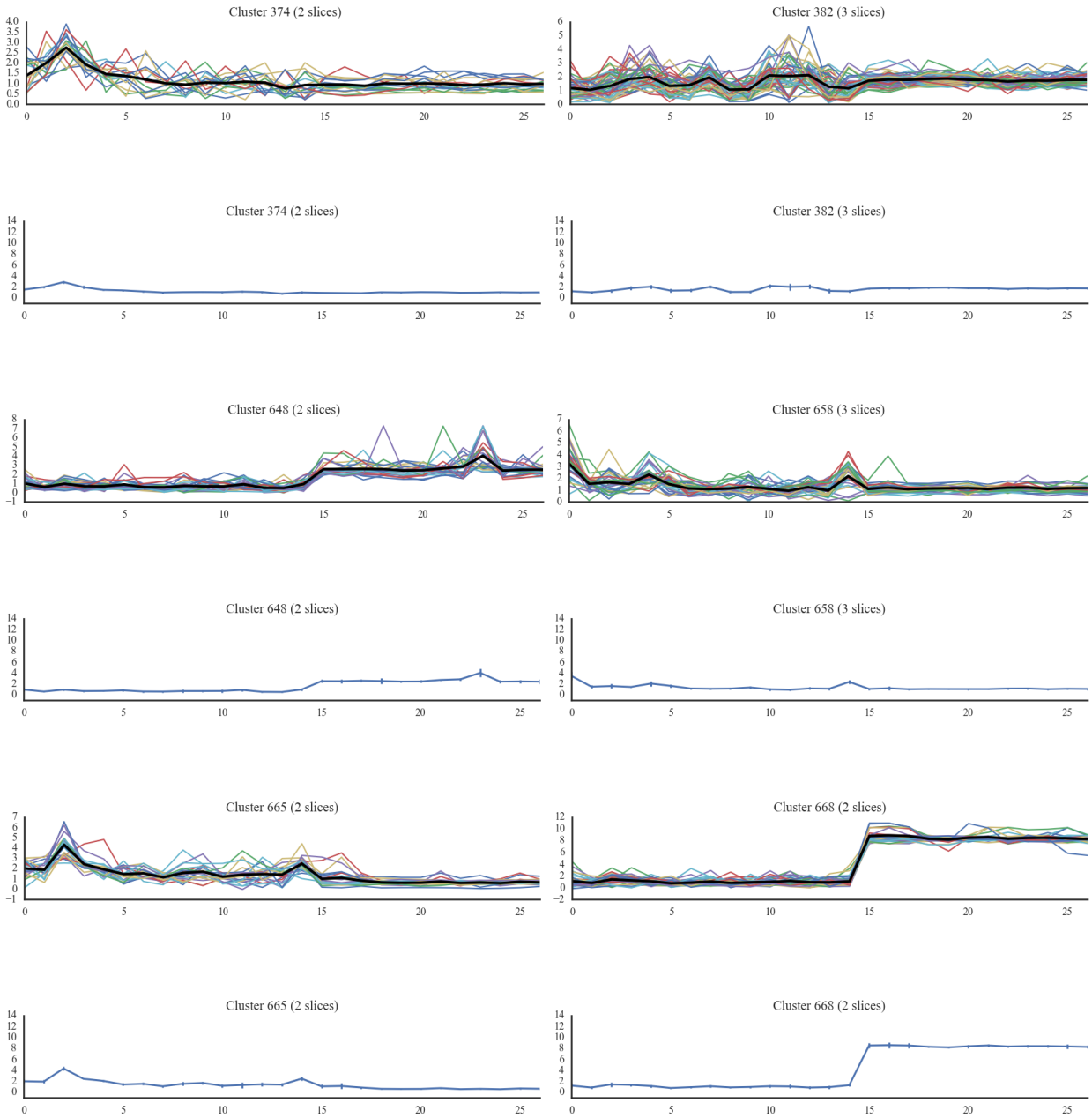
3. *k* = 4



Σχήμα 44: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 4.

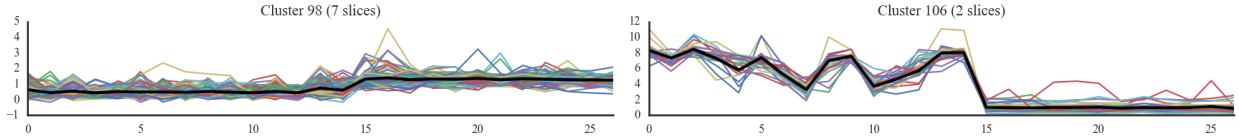
4. *k* = 789

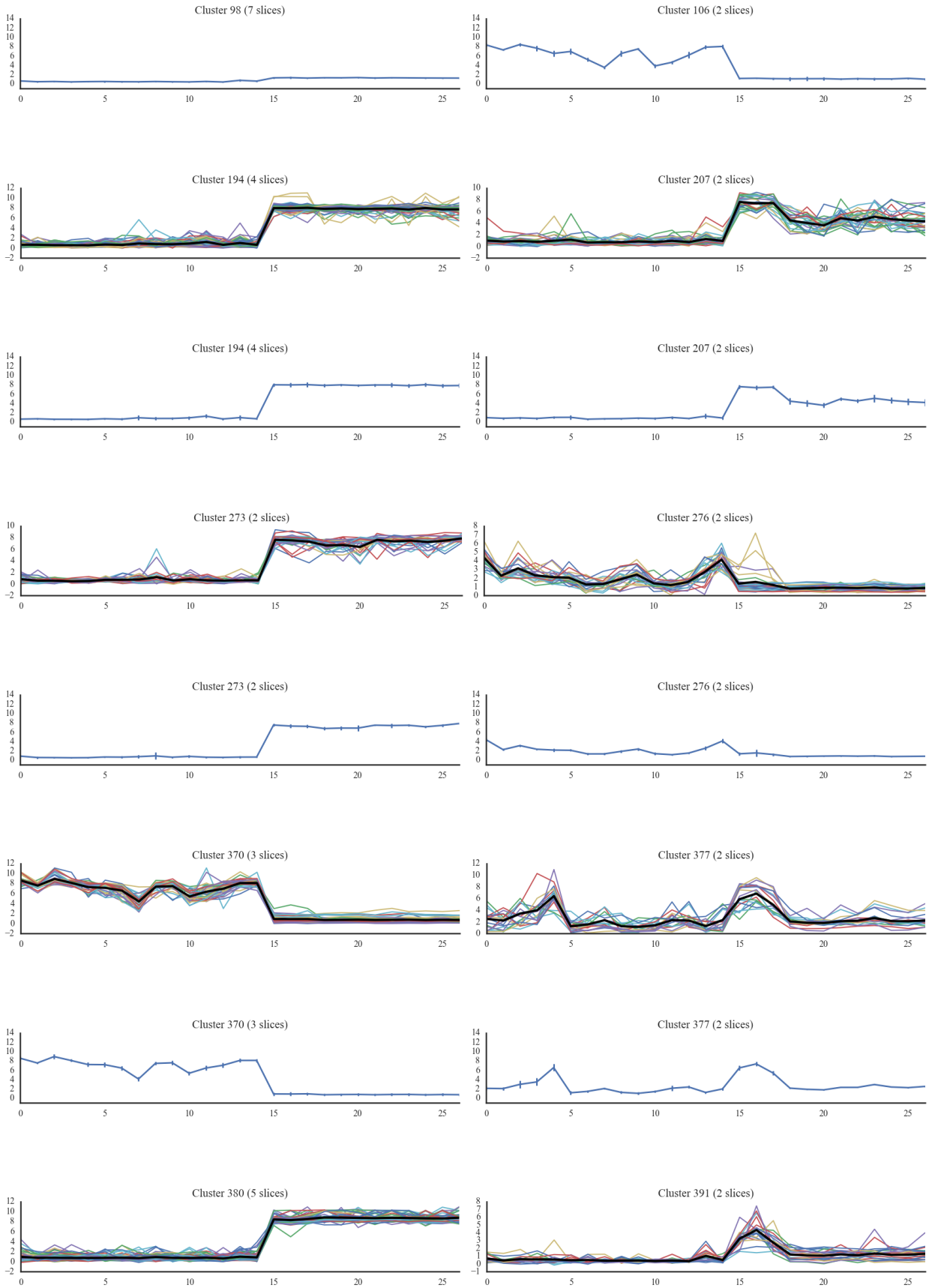


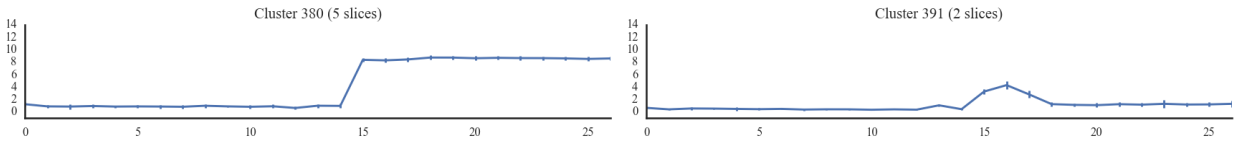


Σχήμα 45: Κάποιες σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 789.

5. $k = 740$

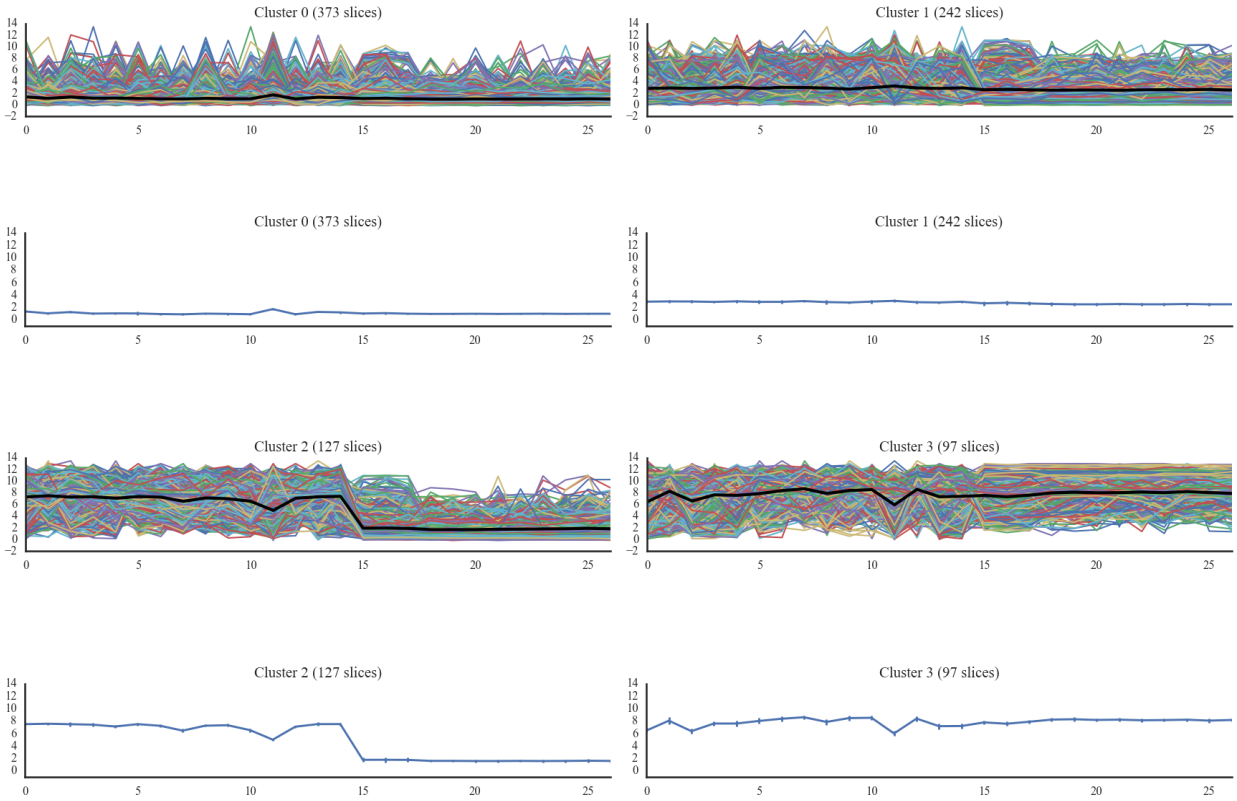


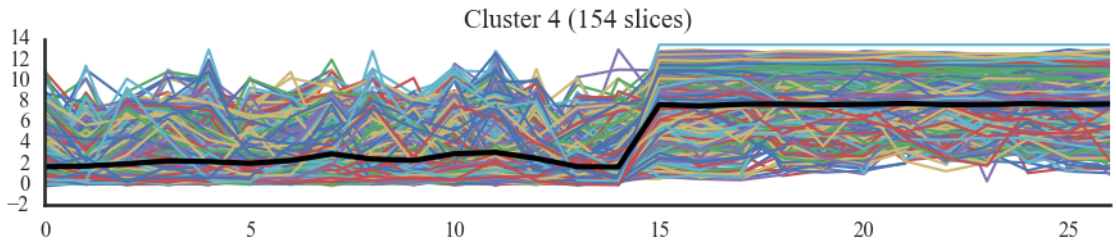

Cluster 98 (7 slices)
Cluster 106 (2 slices)
Cluster 194 (4 slices)
Cluster 207 (2 slices)
Cluster 194 (4 slices)
Cluster 207 (2 slices)
Cluster 273 (2 slices)
Cluster 276 (2 slices)
Cluster 273 (2 slices)
Cluster 276 (2 slices)
Cluster 370 (3 slices)
Cluster 377 (2 slices)
Cluster 370 (3 slices)
Cluster 377 (2 slices)
Cluster 380 (5 slices)
Cluster 391 (2 slices)

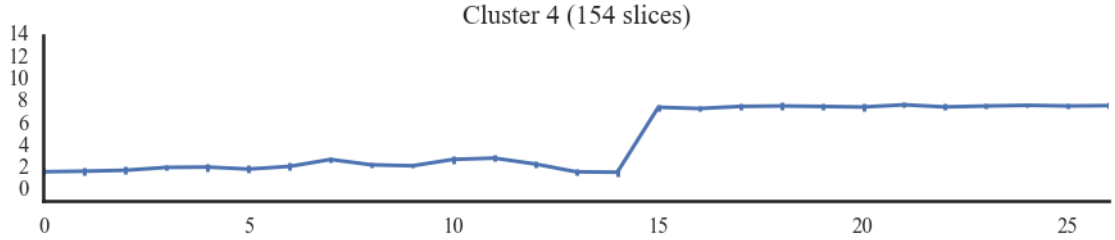


Σχήμα 46: Κάποιες σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 740.

6. $k = 5$

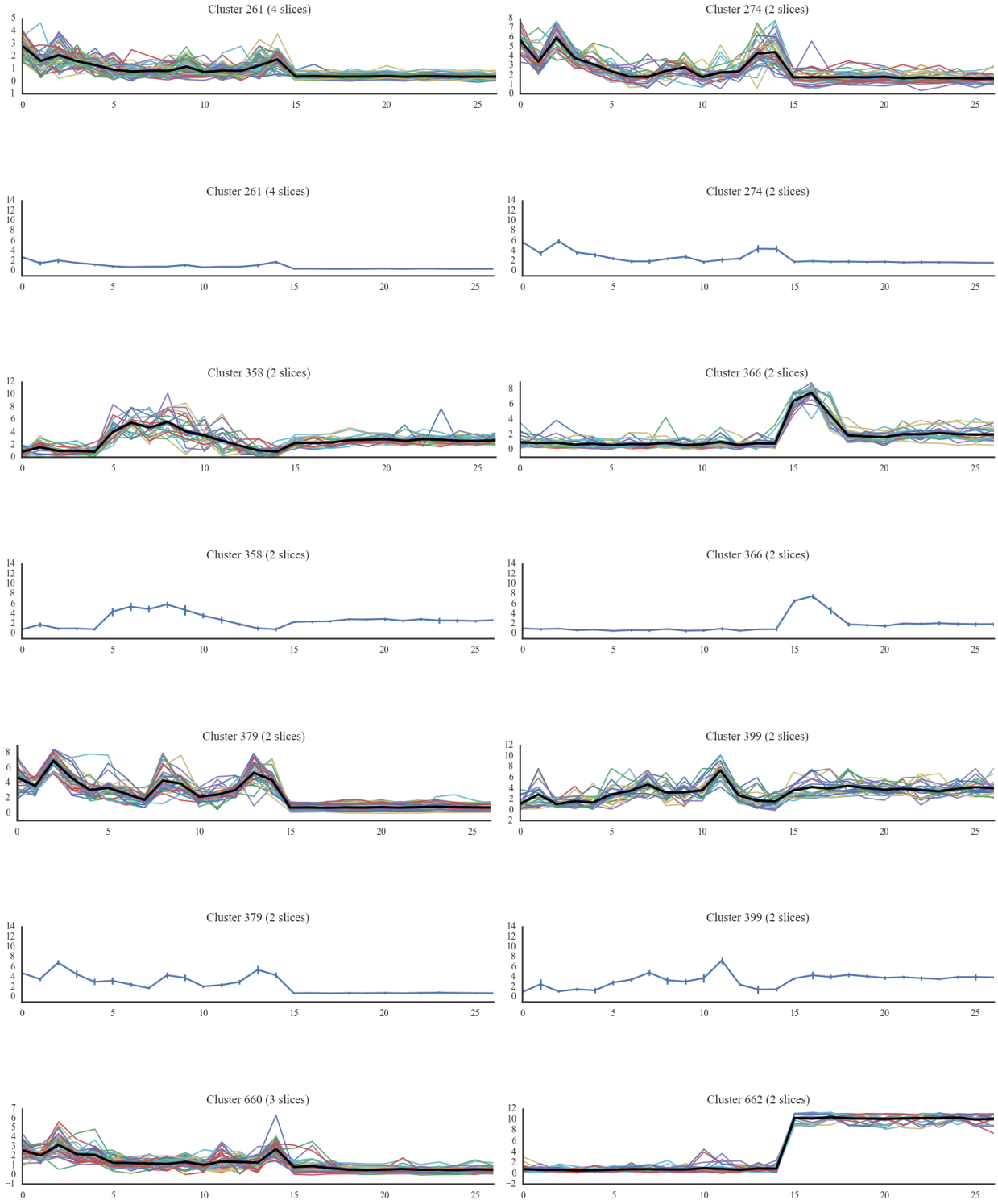


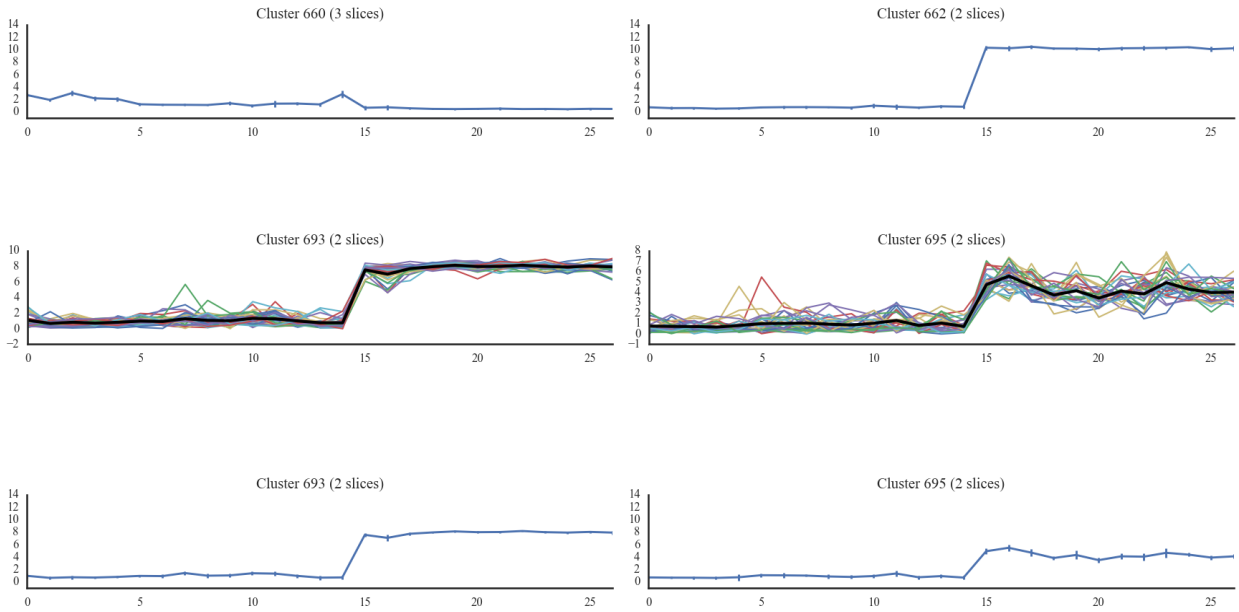




Σχήμα 47: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 5

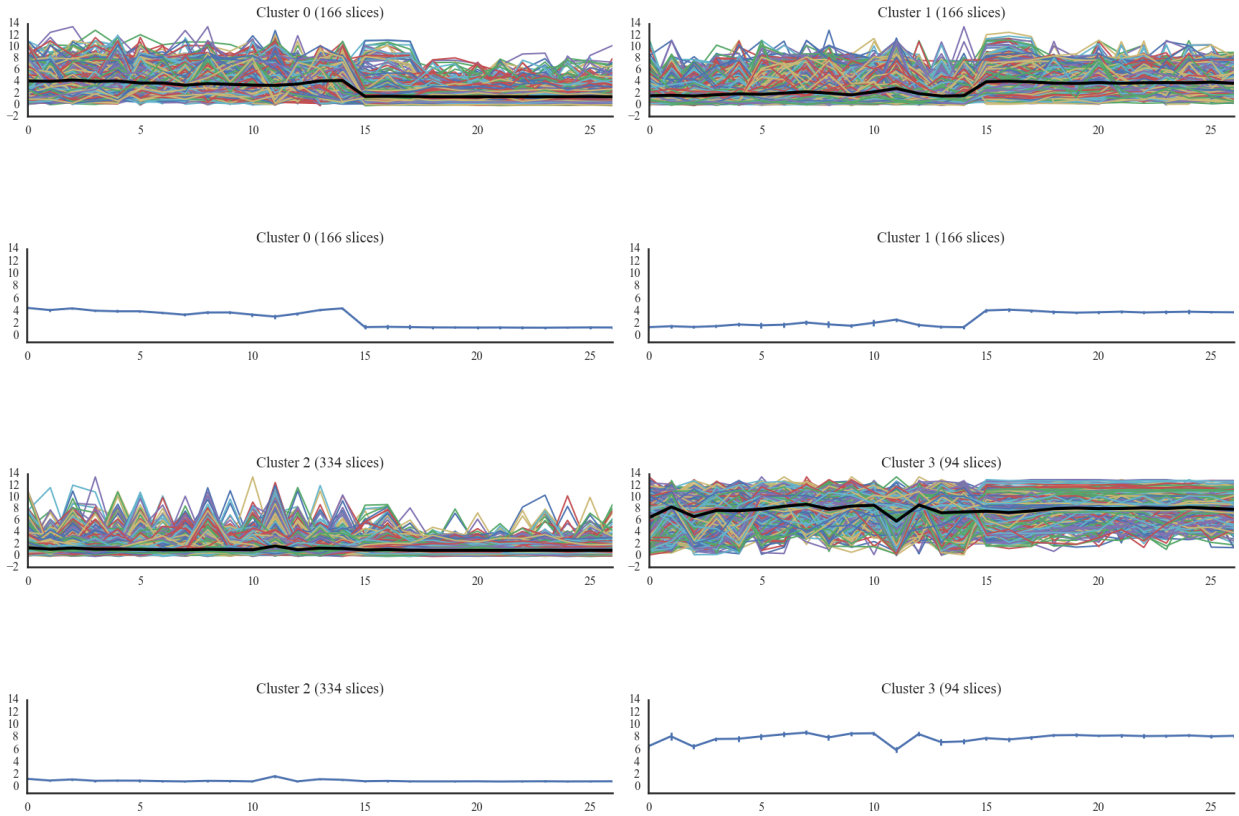
7. *k* = 758

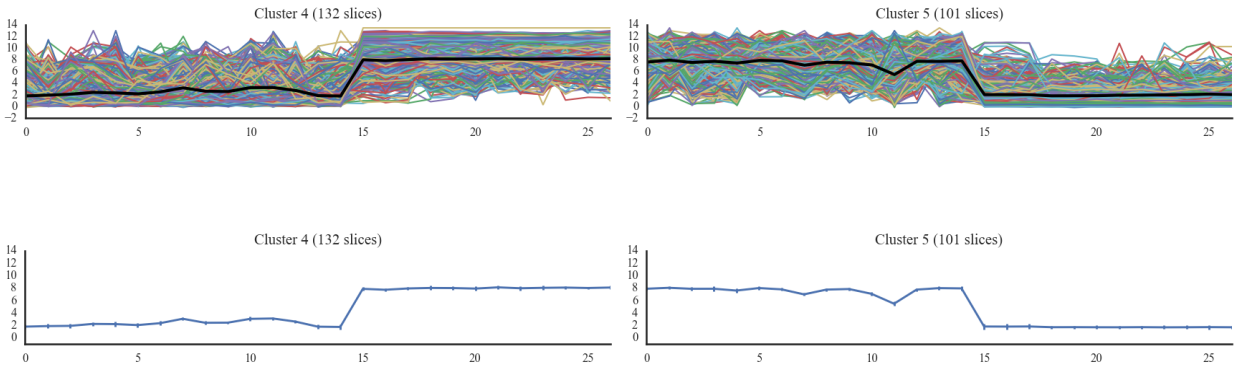




Σχήμα 48: Κάποιες σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 758.

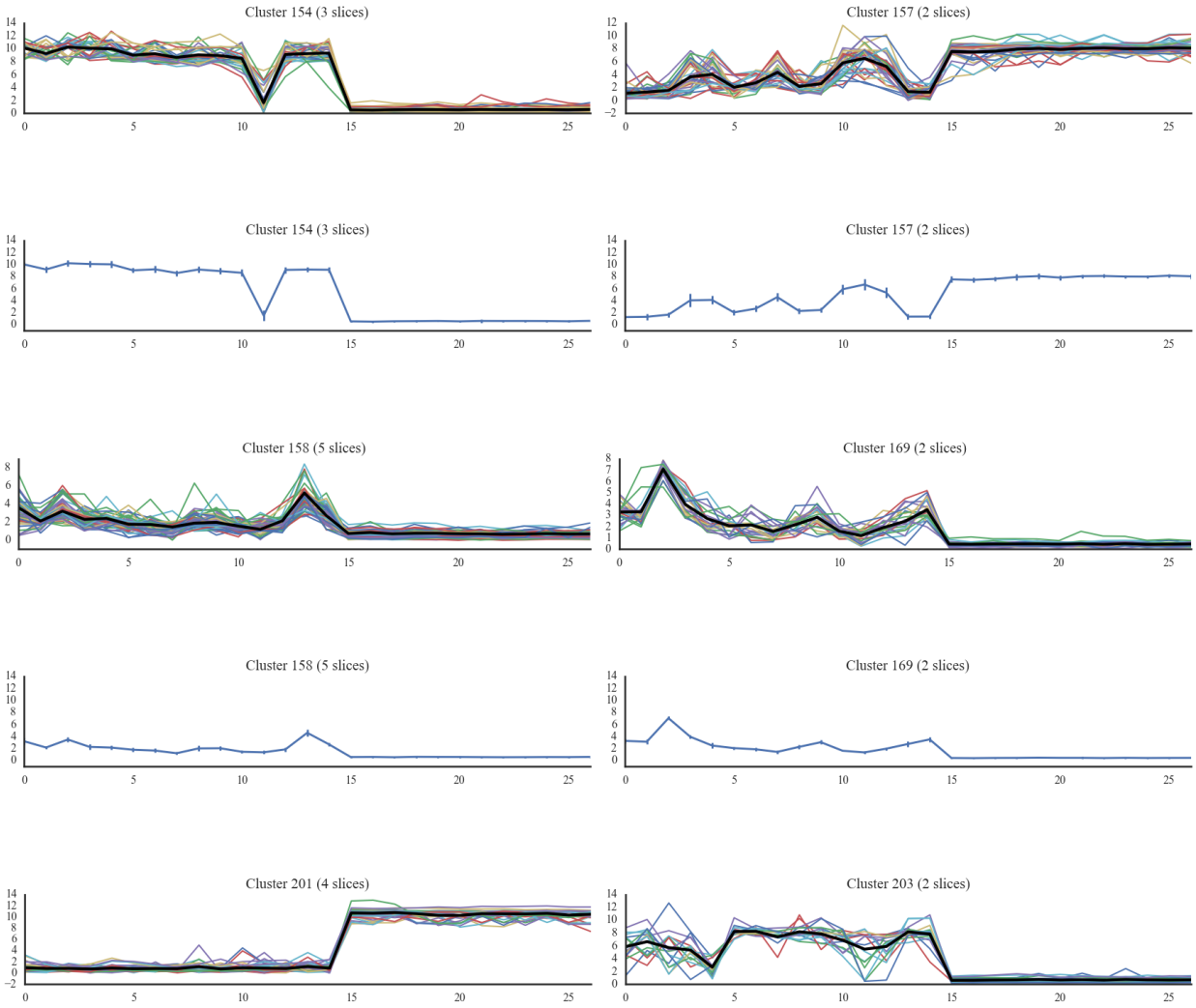
8. $k = 6$

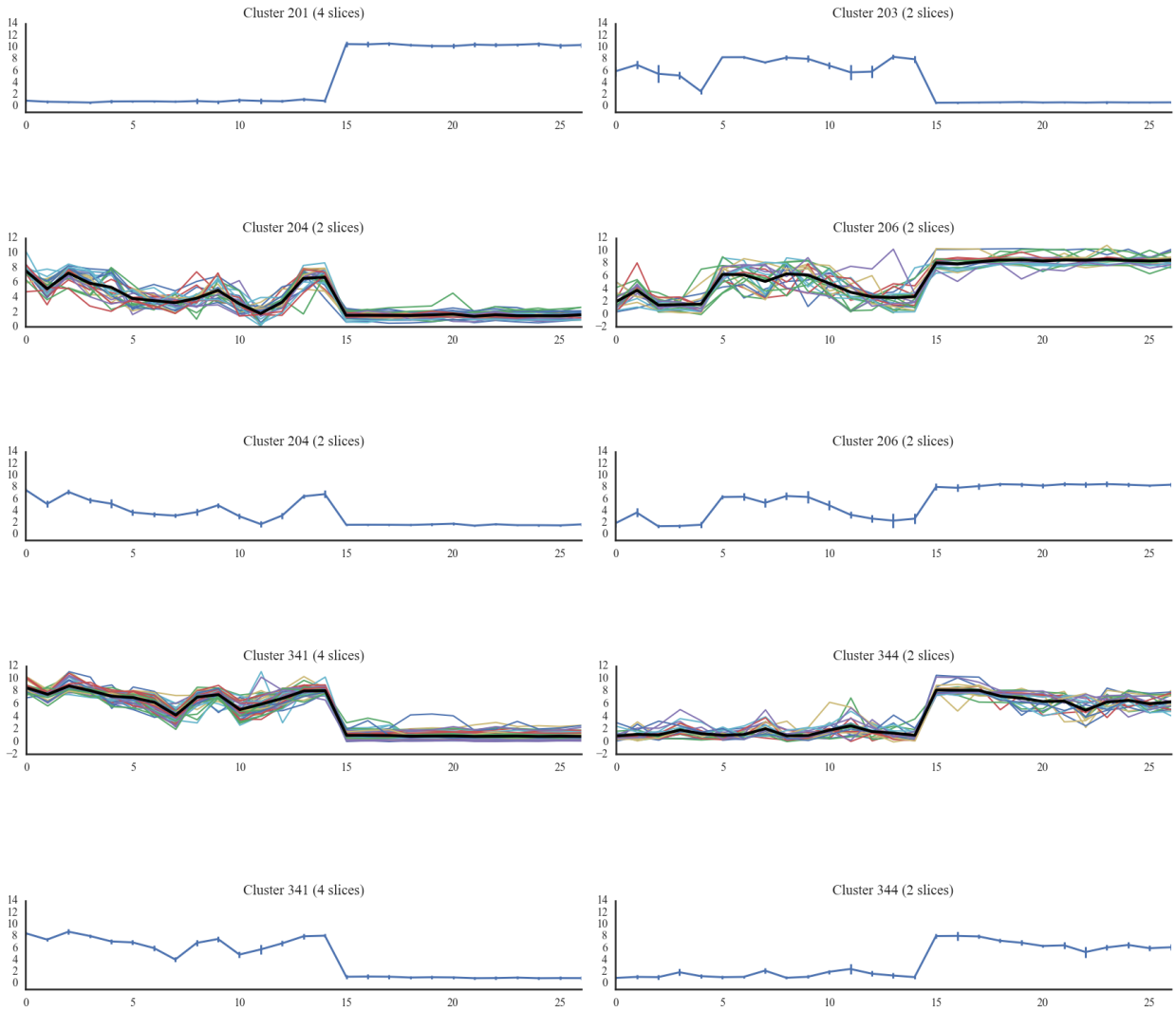




Σχήμα 49: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 6.

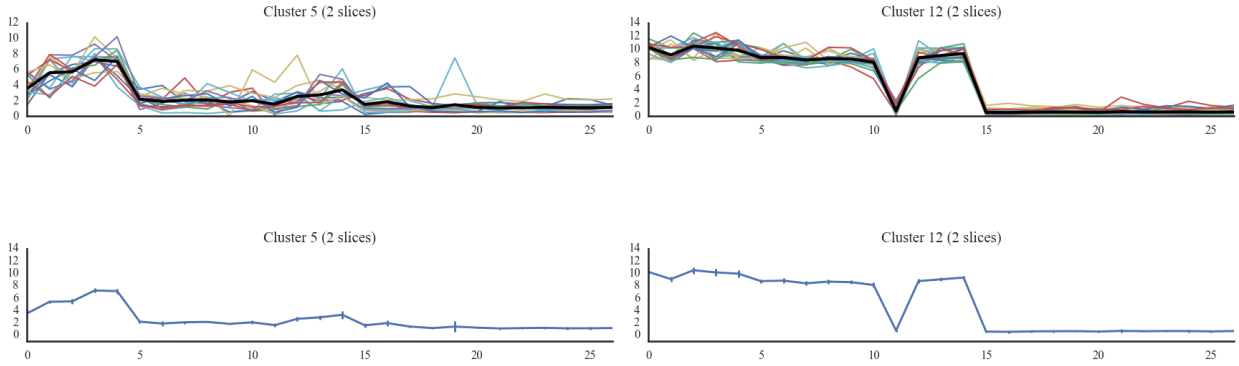
9. *k* = 650

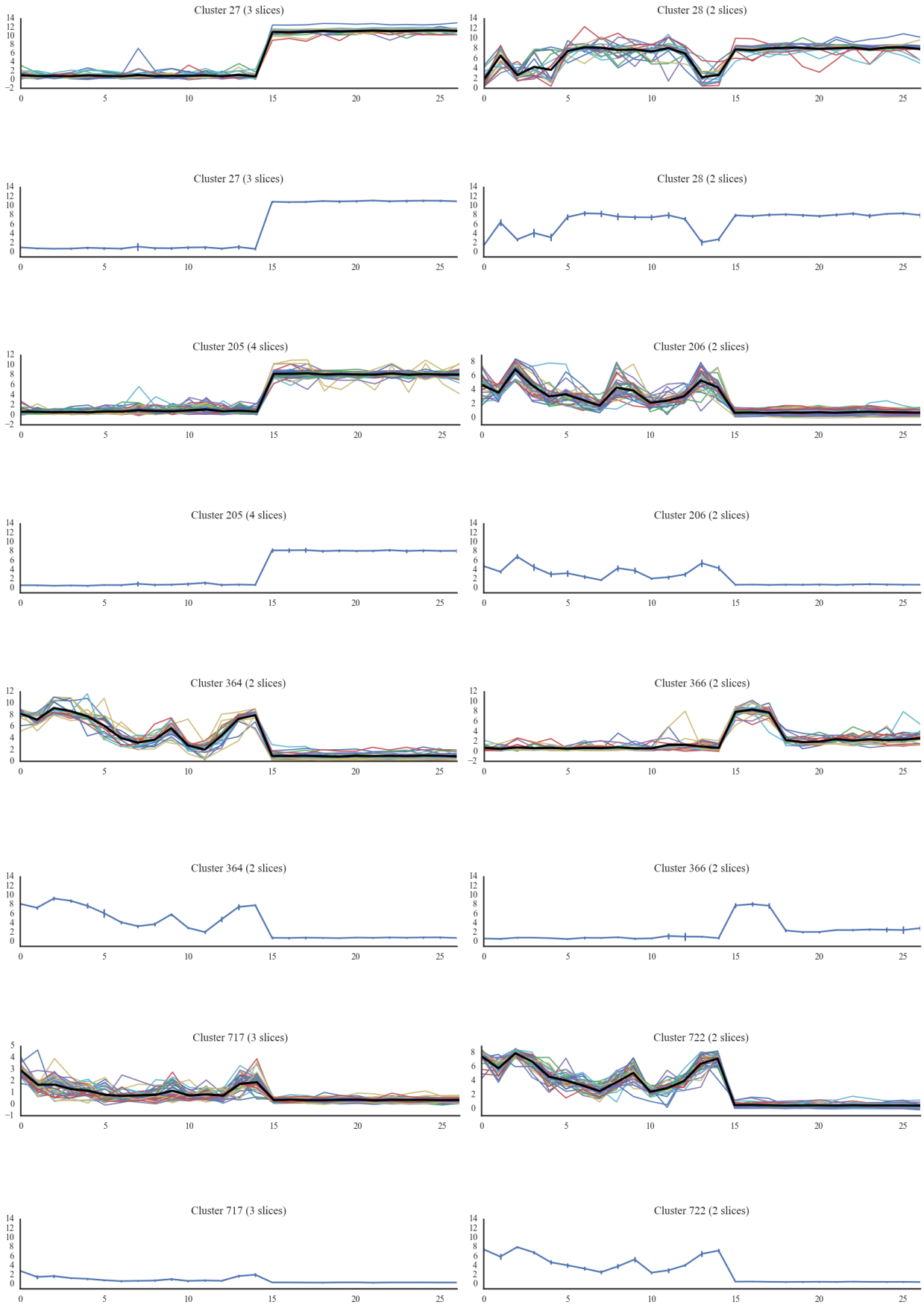




Σχήμα 50: Κάποιες σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των γονιδίων για k = 650.

10. k = 726



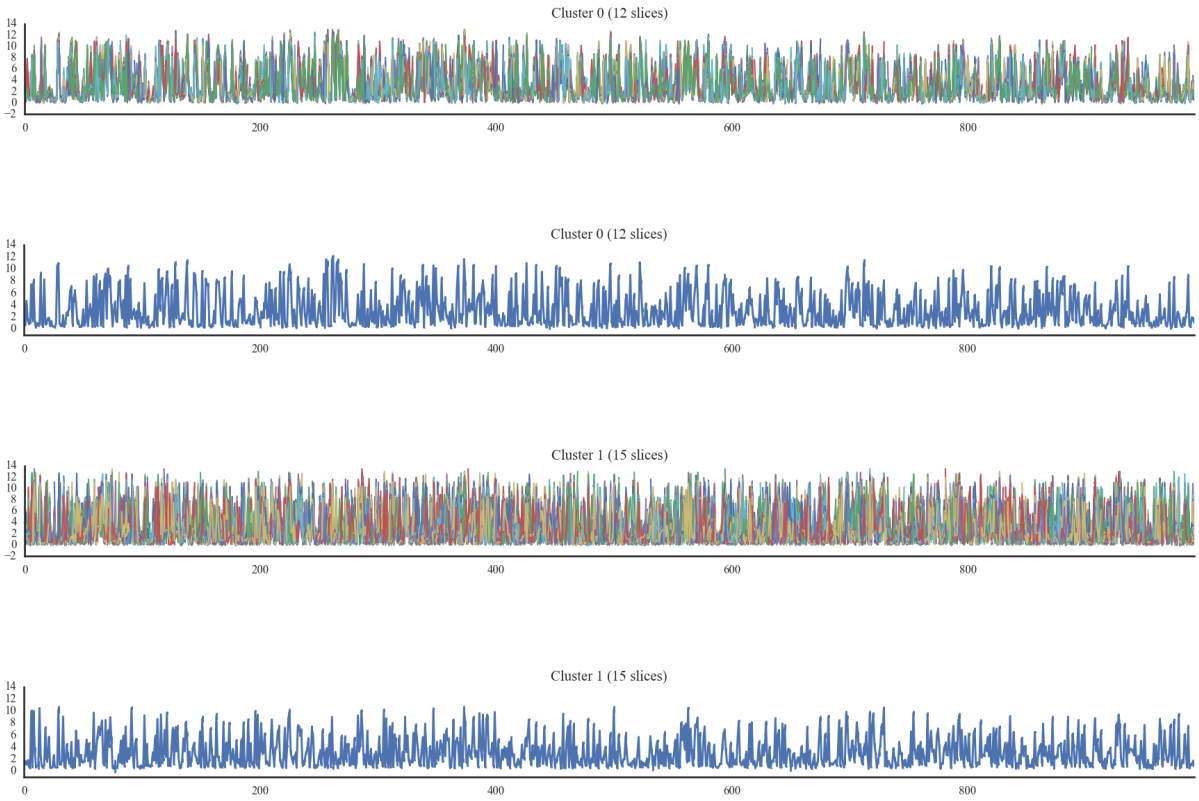


Σχήμα 51: Κάποιες σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του

άξονα των γονιδίων για k = 726.

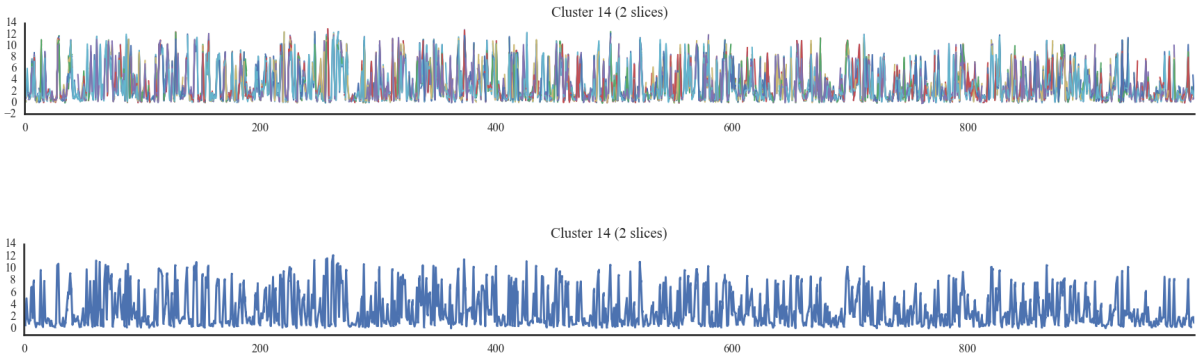
- <u>Τομή στον άξονα των δειγμάτων</u>

1. *k* = 2



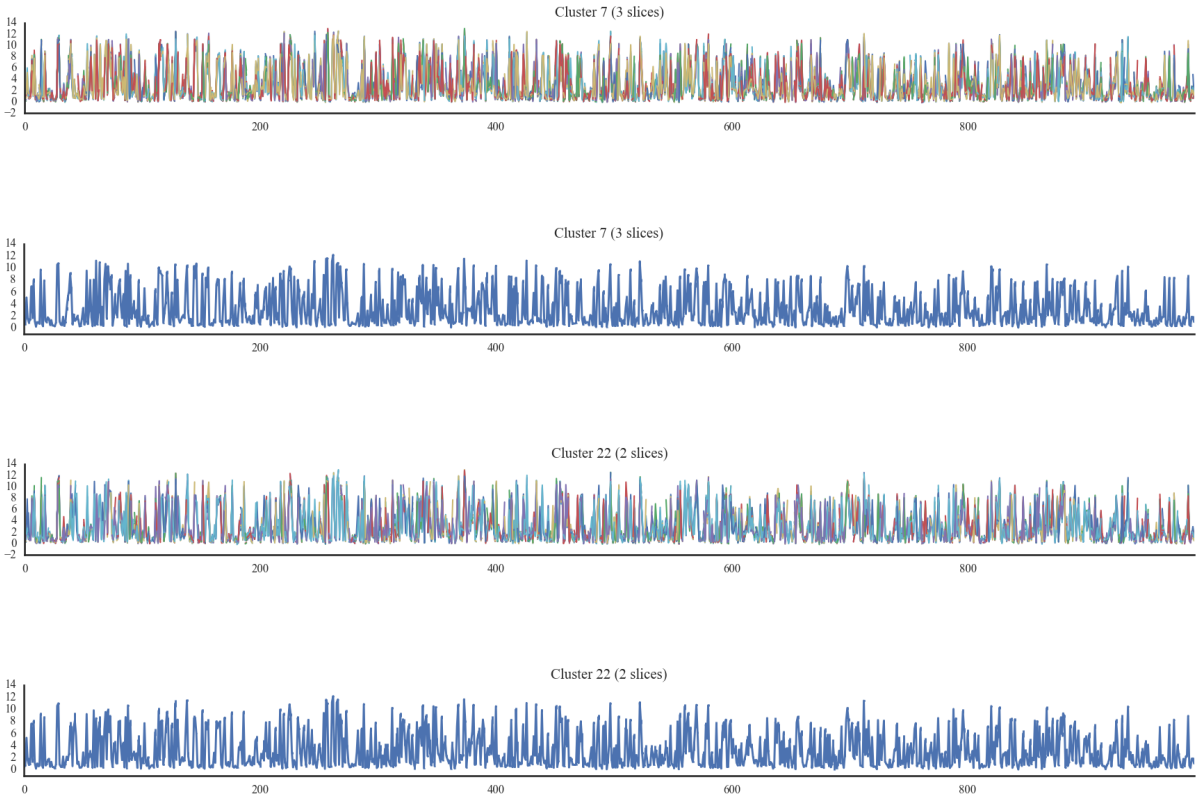
Σχήμα 52: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των δειγμάτων για k = 2.

2. *k* = 26



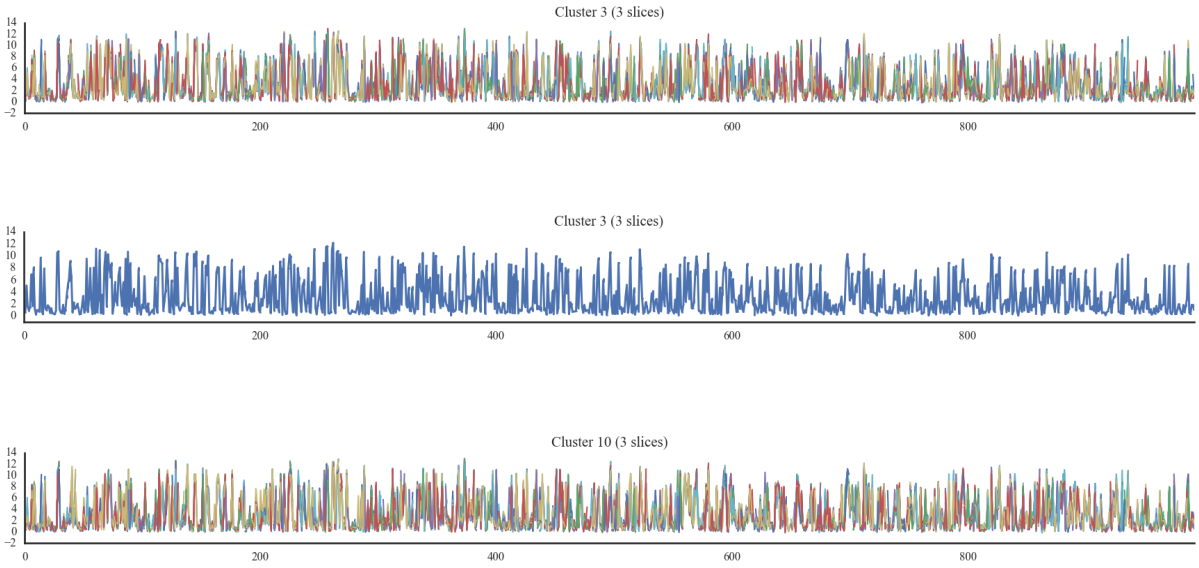
Σχήμα 53: Η σημαντική συστάδα που απέφερε ο κλασικός k-Means στην τομή του άξονα των δειγμάτων για k = 26.

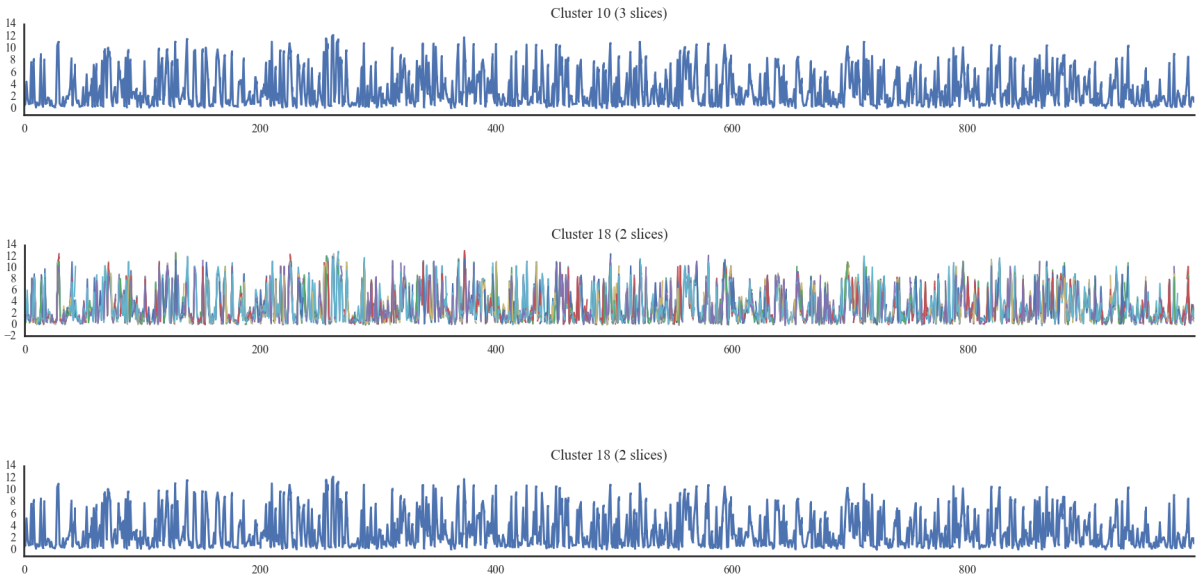
3. $k = 24$



Σχήμα 54: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των δειγμάτων για k = 24.

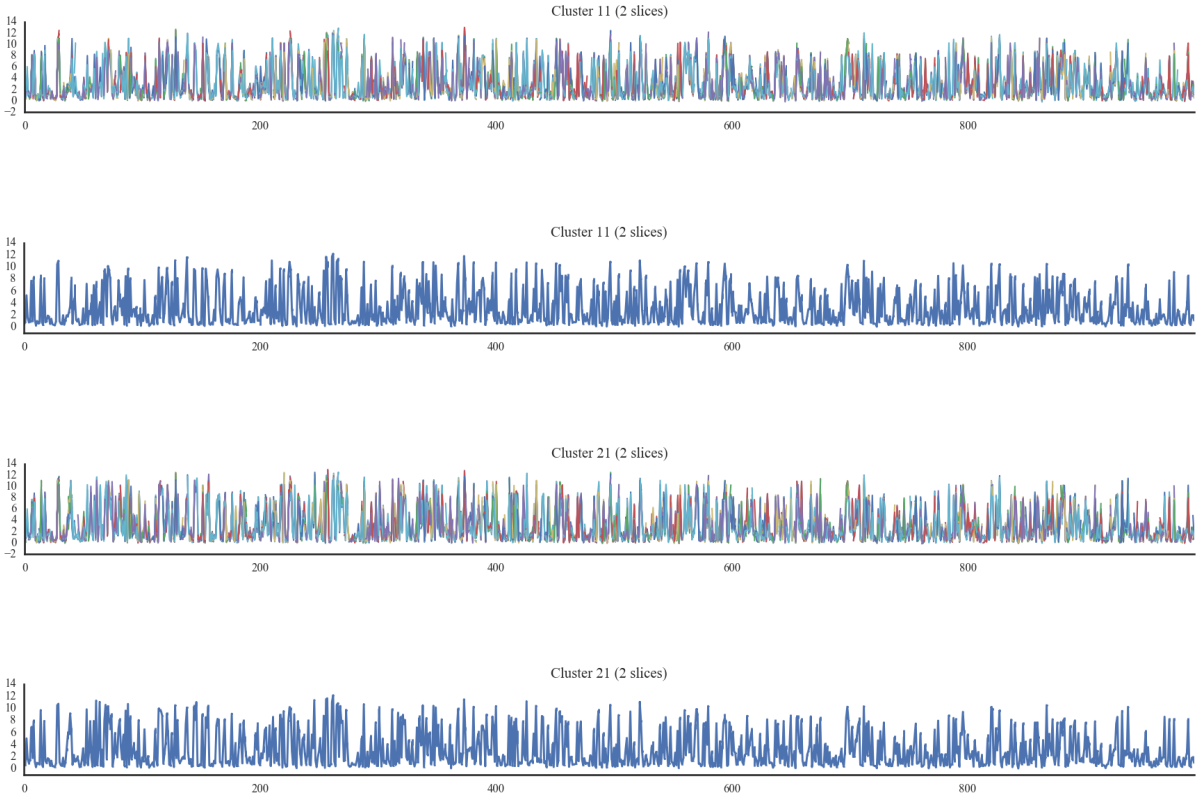
4. $k = 23$





Σχήμα 55: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των δειγμάτων για k = 23.

5. ***k*** **= 25**



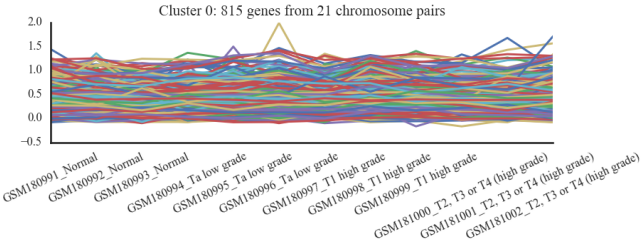
Σχήμα 56: Οι σημαντικές συστάδες που απέφερε ο κλασικός k-Means στην τομή του άξονα των δειγμάτων για k = 25.

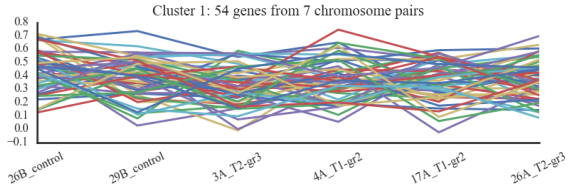
Είναι εμφανές ότι στον άξονα των χρωμοσωμάτων δεν είχαμε ιδιαίτερα καλές συσταδοποιήσεις, γεγονός που δικαιολογείται απόλυτα αφού κάθε ζεύγος χρωμοσωμάτων περιέχει ≤ 993 γονίδια με πολύ διαφορετικό προφίλ έκφρασης το καθένα, κι έτσι το κεντροειδές που λαμβάνει υπ' όψη ο αλγόριθμος δεν αποτελεί αξιόπιστο αντιπρόσωπο. Αντιθέτως, στους άξονες των γονιδίων και των δειγμάτων είχαμε αρκετά καλές ομαδοποιήσεις.

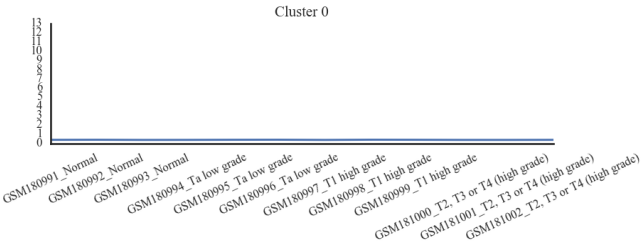
δ-TRIMAX

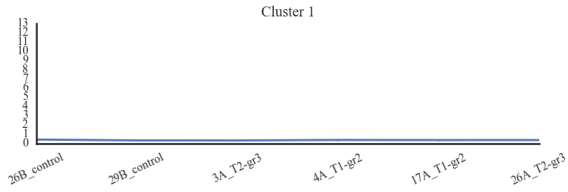
Η τελευταία τεχνική που εφαρμόσαμε ήταν η συσταδοποίηση υποχώρου με χρήση του αλγορίθμου δ-TRIMAX, δοκιμάζοντας διάφορες τιμές των μεταβλητών δ και λ στα εύρη [0.005, 0.05] και [1.5, 2.5] αντίστοιχα. Στα παρακάτω σχήματα παρουσιάζονται ενδεικτικά οι συστάδες που ανακάλυψε ο αλγόριθμος για κάποιους συνδυασμούς τιμών. Σε κάθε περίπτωση θέσαμε την τιμή του κατωφλίου για την πολλαπλή διαγραφή χρωμοσωμάτων στο 2, των γονιδίων στο 10 και των δειγμάτων στο 3.

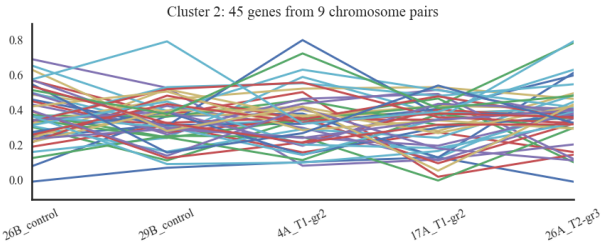
- **δ = 0.02, λ = 2.5**

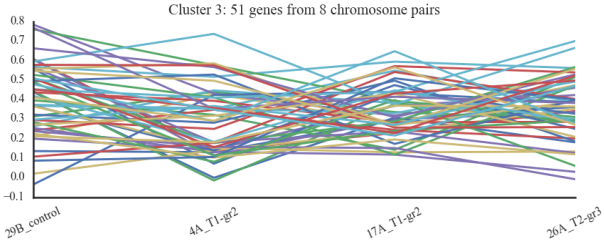


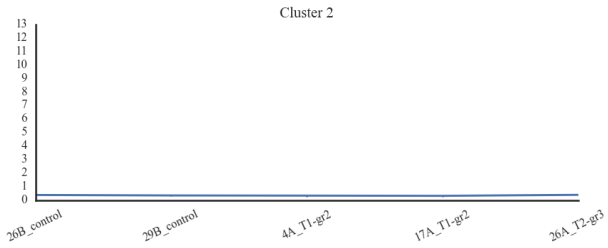


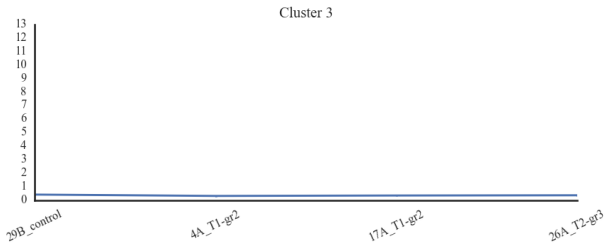


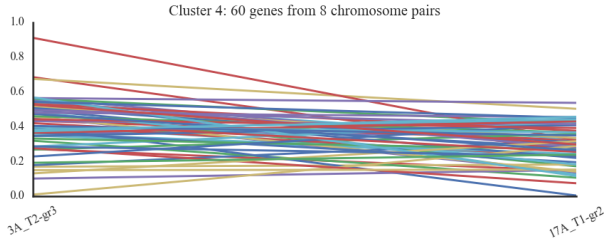


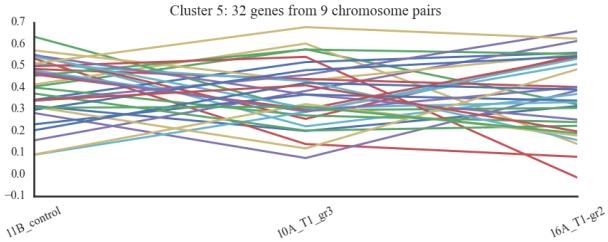


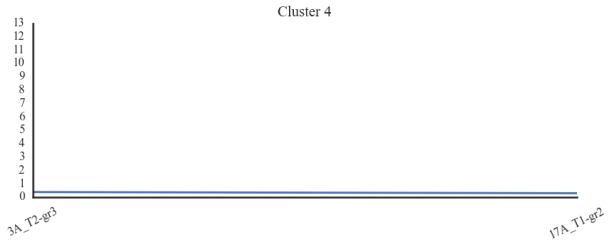


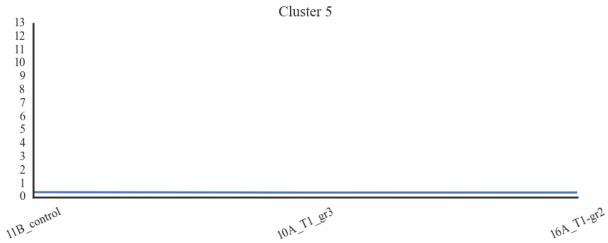

Cluster 2
26B_control
29B_control
4A_T1-gr2
17A_T1-gr2
26A_T2-gr3

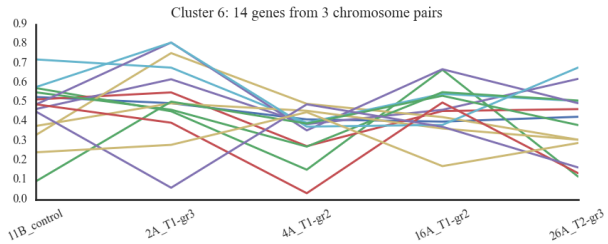

Cluster 3
29B_control
4A_T1-gr2
17A_T1-gr2
26A_T2-gr3

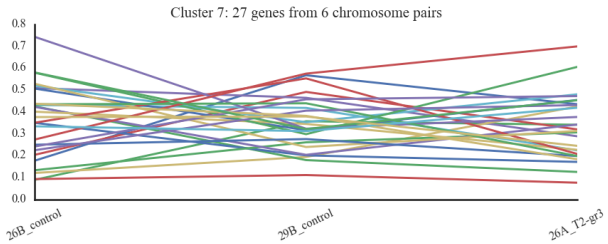

Cluster 4: 60 genes from 8 chromosome pairs
3A_T2-gr3
17A_T1-gr2

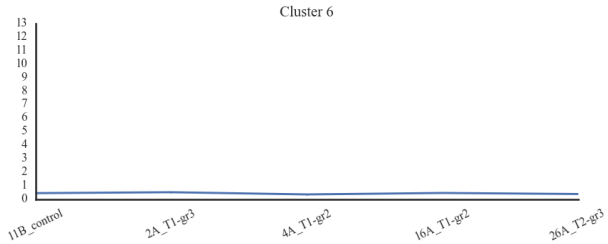

Cluster 5: 32 genes from 9 chromosome pairs
11B_control
10A_T1_gr3
16A_T1-gr2

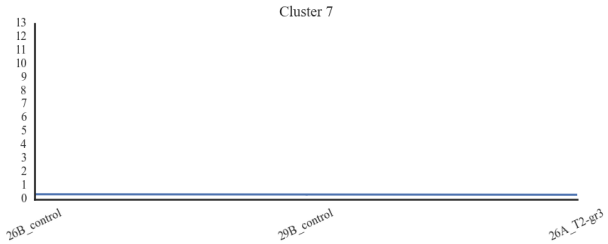

Cluster 4
3A_T2-gr3
17A_T1-gr2

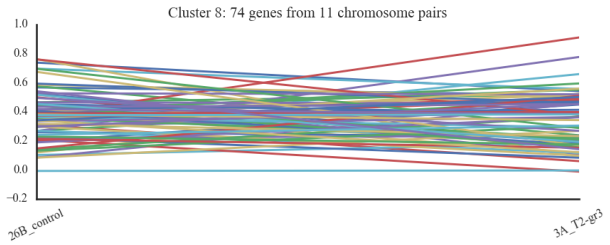

Cluster 5
11B_control
10A_T1_gr3
16A_T1-gr2

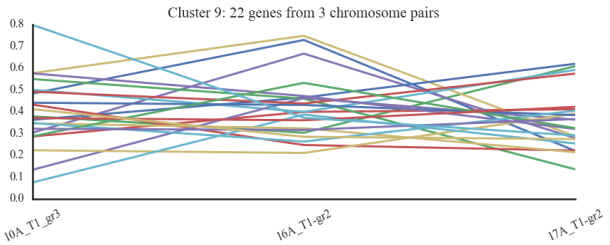

Cluster 6: 14 genes from 3 chromosome pairs
11B_control
2A_T1-gr3
4A_T1-gr2
16A_T1-gr2
26A_T2-gr3

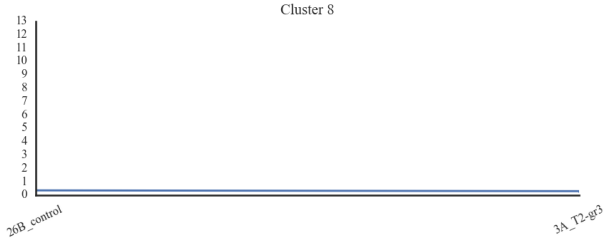

Cluster 7: 27 genes from 6 chromosome pairs
26B_control
29B_control
26A_T2-gr3

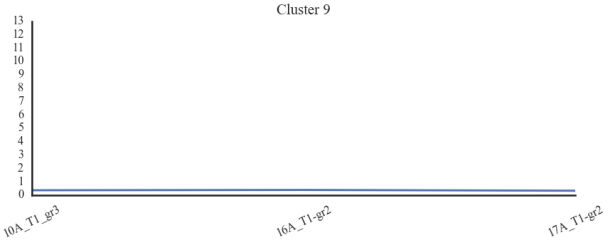

Cluster 6
11B_control
2A_T1-gr3
4A_T1-gr2
16A_T1-gr2
26A_T2-gr3

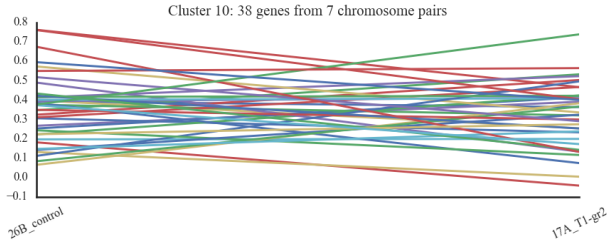

Cluster 7
26B_control
29B_control
26A_T2-gr3

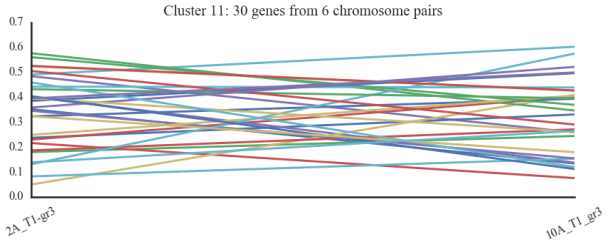

Cluster 8: 74 genes from 11 chromosome pairs
26B_control
3A_T2-gr3

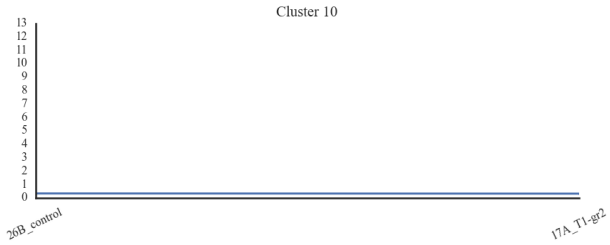

Cluster 9: 22 genes from 3 chromosome pairs
10A_T1_gr3
16A_T1-gr2
17A_T1-gr2

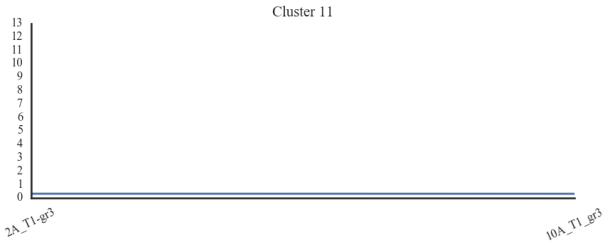

Cluster 8
26B_control
3A_T2-gr3

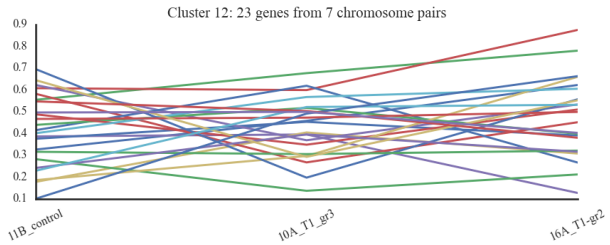

Cluster 9
10A_T1_gr3
16A_T1-gr2
17A_T1-gr2

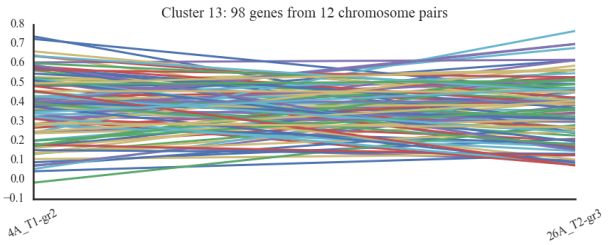

Cluster 10: 38 genes from 7 chromosome pairs
26B_control
17A_T1-gr2

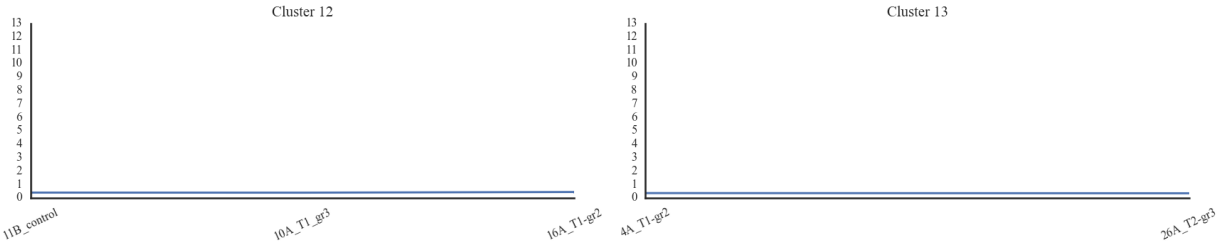

Cluster 11: 30 genes from 6 chromosome pairs
2A_T1-gr3
10A_T1_gr3


Cluster 10
26B_control
17A_T1-gr2


Cluster 11
2A_T1-gr3
10A_T1_gr3

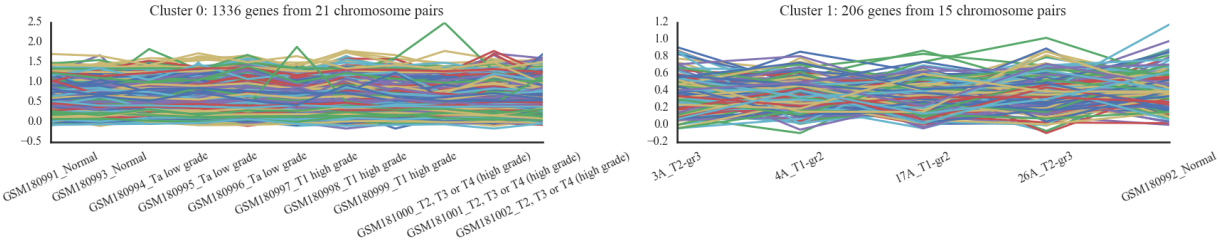

Cluster 12: 23 genes from 7 chromosome pairs
11B_control
10A_T1_gr3
16A_T1-gr2

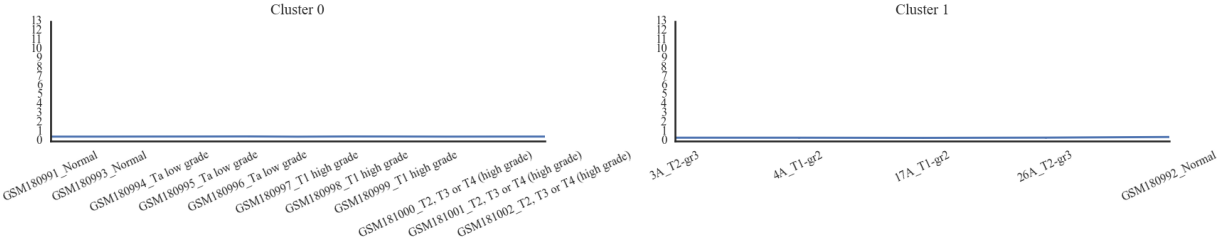

Cluster 13: 98 genes from 12 chromosome pairs
4A_T1-gr2
26A_T2-gr3

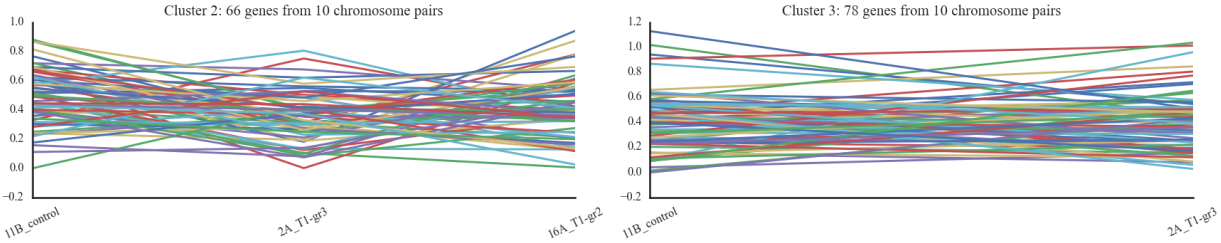


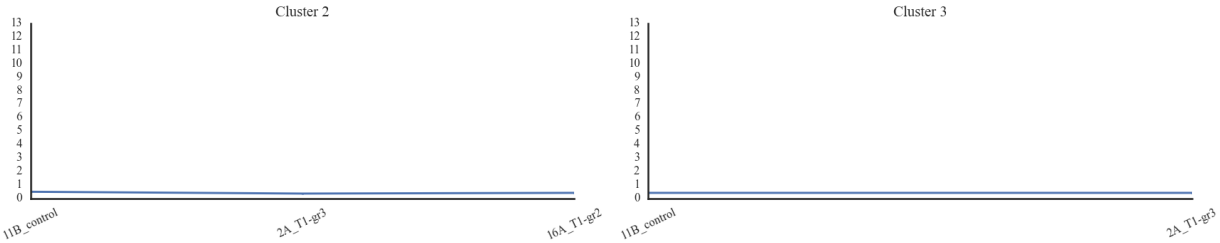
Σχήμα 57: Οι συστάδες που επέστρεψε ο δ-TRIMAX για δ=0.02 και λ=2.5.

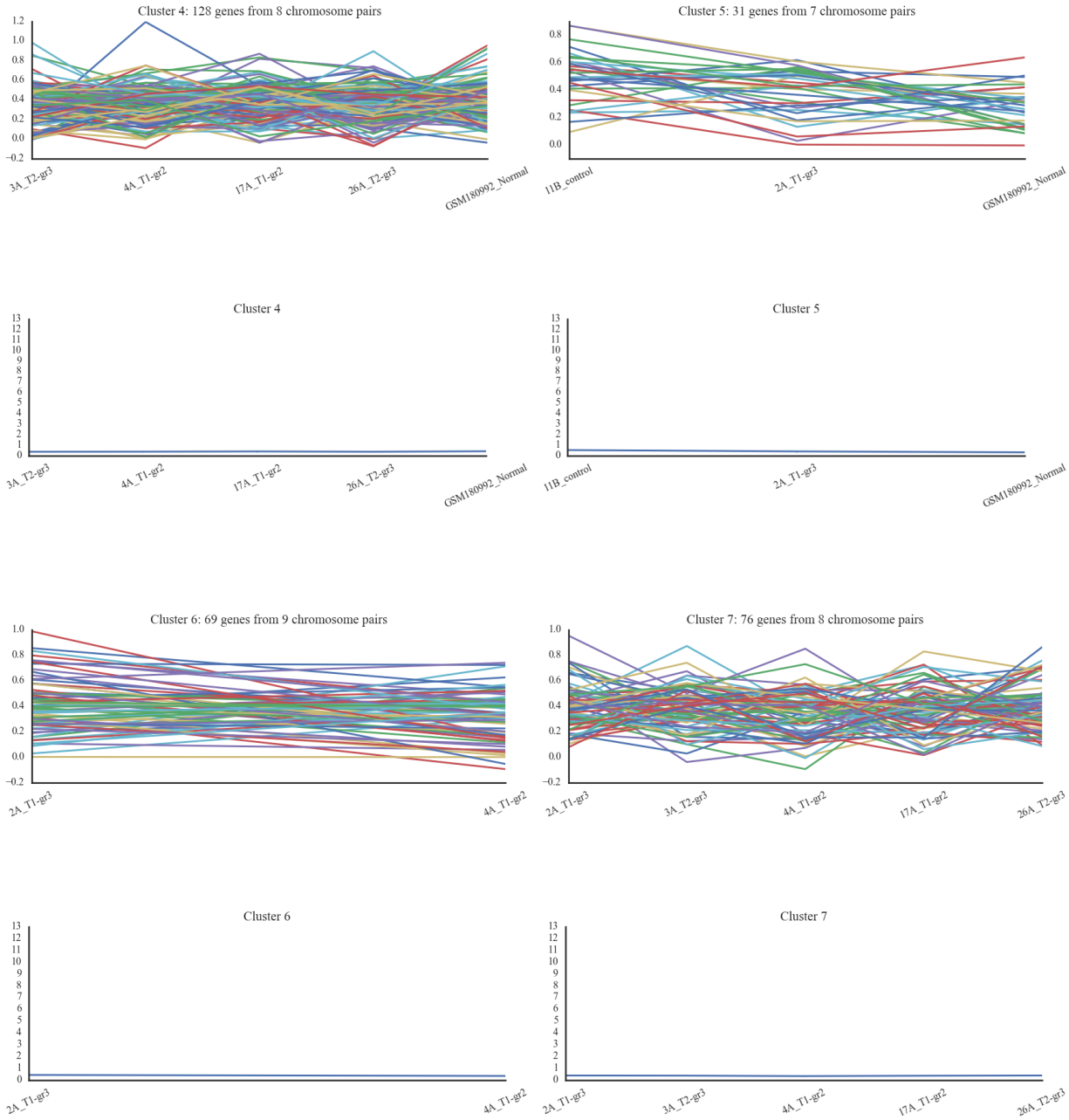
- **δ = 0.03, λ = 1.5**







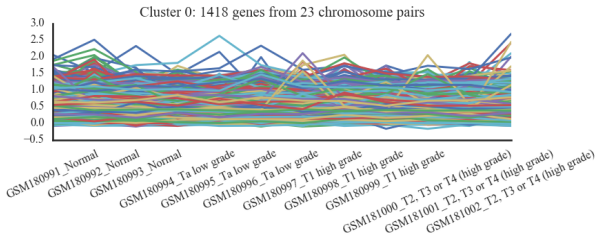


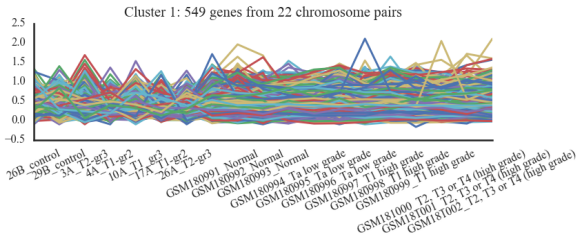


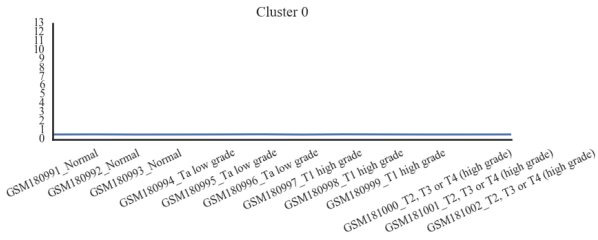
Σχήμα 58: Οι συστάδες που επέστρεψε ο δ-TRIMAX για δ=0.03 και λ=1.5.

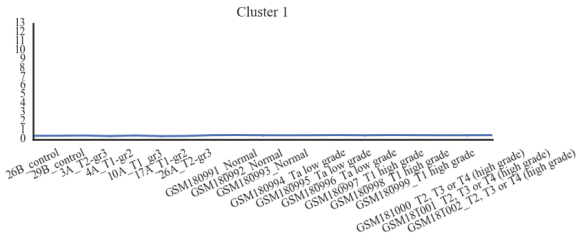
- ***δ* = 0.05, *λ* = 2**

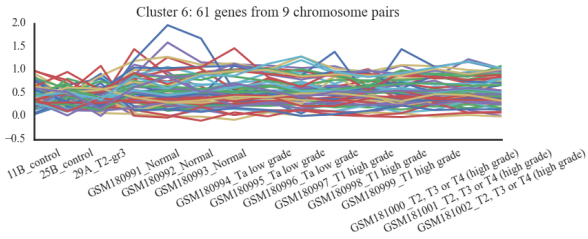
  Με τις συγκεκριμένες τιμές παραμέτρων, ο δ-TRIMAX ανακάλυψε 98 triclusters και κατόρθωσε να ομαδοποιήσει 19 δείγματα σε μια συστάδα. Παρακάτω δίνουμε ενδεικτικά κάποια από αυτά τα triclusters που επέστρεψε.

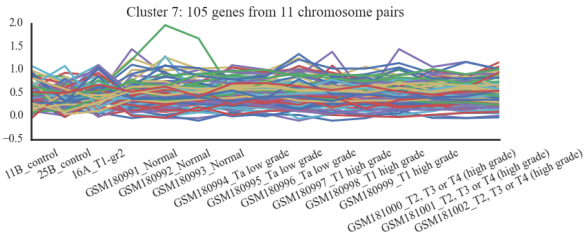

Cluster 0: 1418 genes from 23 chromosome pairs

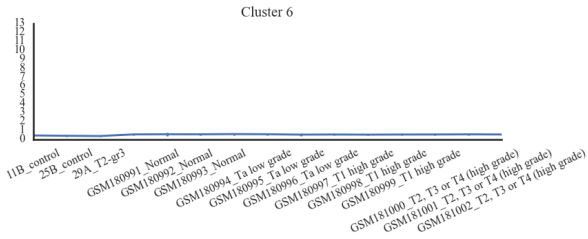

Cluster 1: 549 genes from 22 chromosome pairs

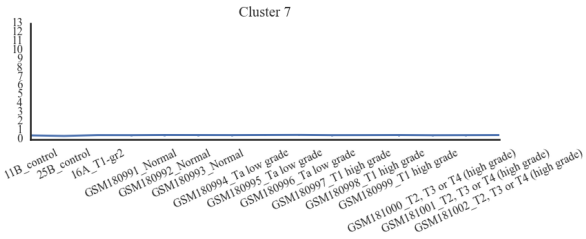

Cluster 0

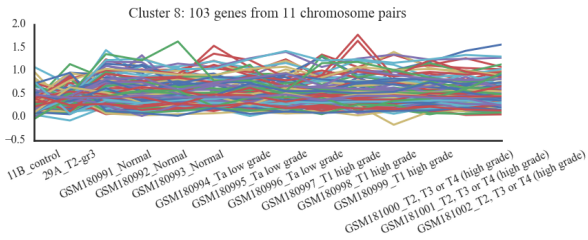

Cluster 1

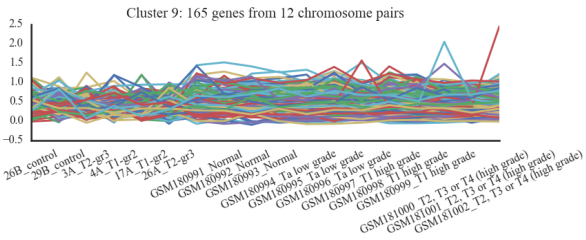

Cluster 6: 61 genes from 9 chromosome pairs

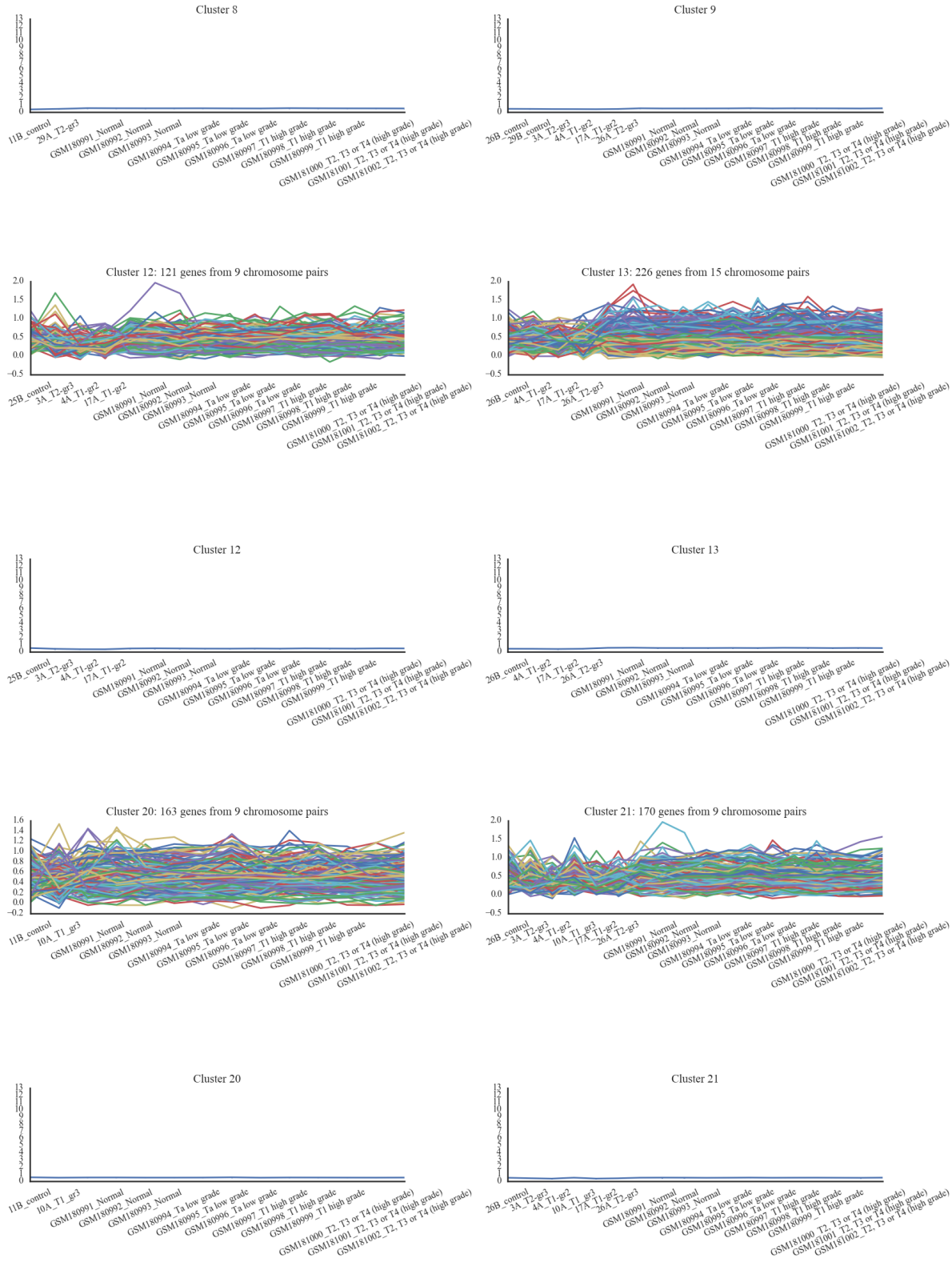

Cluster 7: 105 genes from 11 chromosome pairs


Cluster 6


Cluster 7


Cluster 8: 103 genes from 11 chromosome pairs


Cluster 9: 165 genes from 12 chromosome pairs



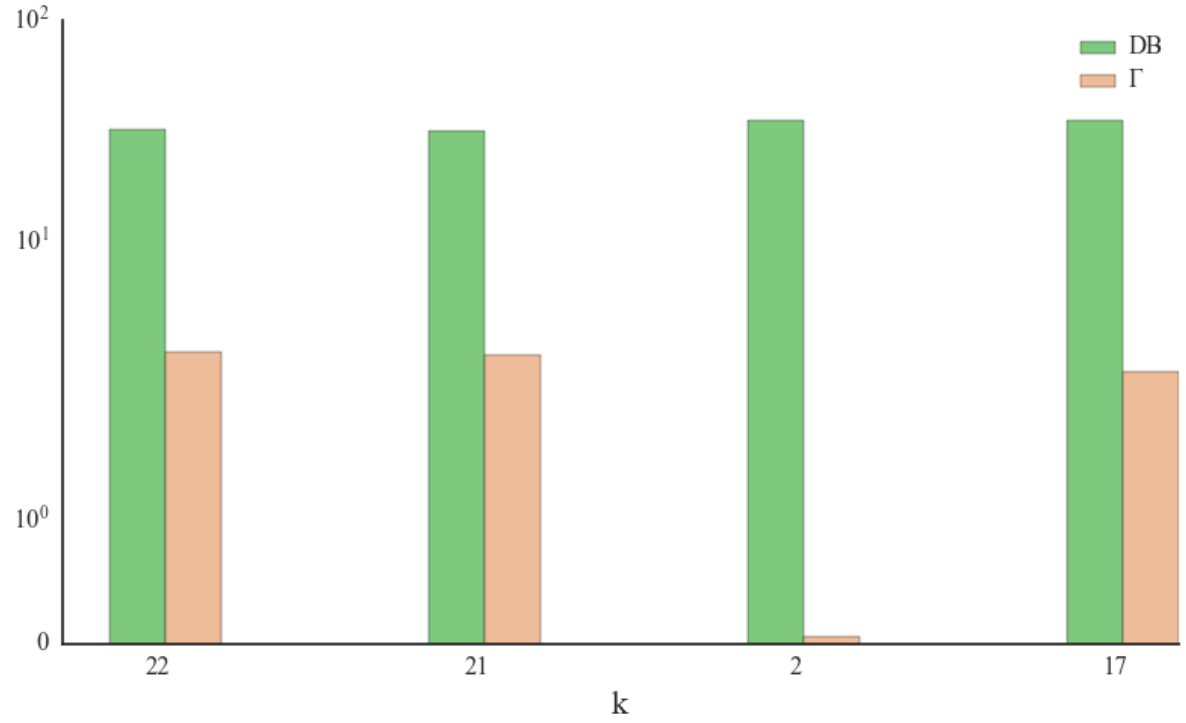
Σχήμα 59: Κάποιες συστάδες που επέστρεψε ο δ-TRIMAX για δ=0.05 και λ=2.

Δ.4) Αξιολογηση Αποτελεσματων

Στα επόμενα διαγράμματα παρουσιάζονται οι τιμές του δείκτη Davies-Bouldin καθώς και του τροποποιημένου δείκτη Γ του Hubert για τις διάφορες τιμές του *k* στην εκτέλεση των k-Means αλγορίθμων. Στη συνέχεια παρουσιάζεται ένα ραβδόγραμμα με τον αριθμό των συστάδων που ανακάλυψε ο δ-TRIMAX κατά την εκτέλεσή του με διάφορες τιμές των παραμέτρων *δ* και *λ*.

Τρισδιάστατος k-Means

- Τομή στον άξονα των χρωμοσωμάτων

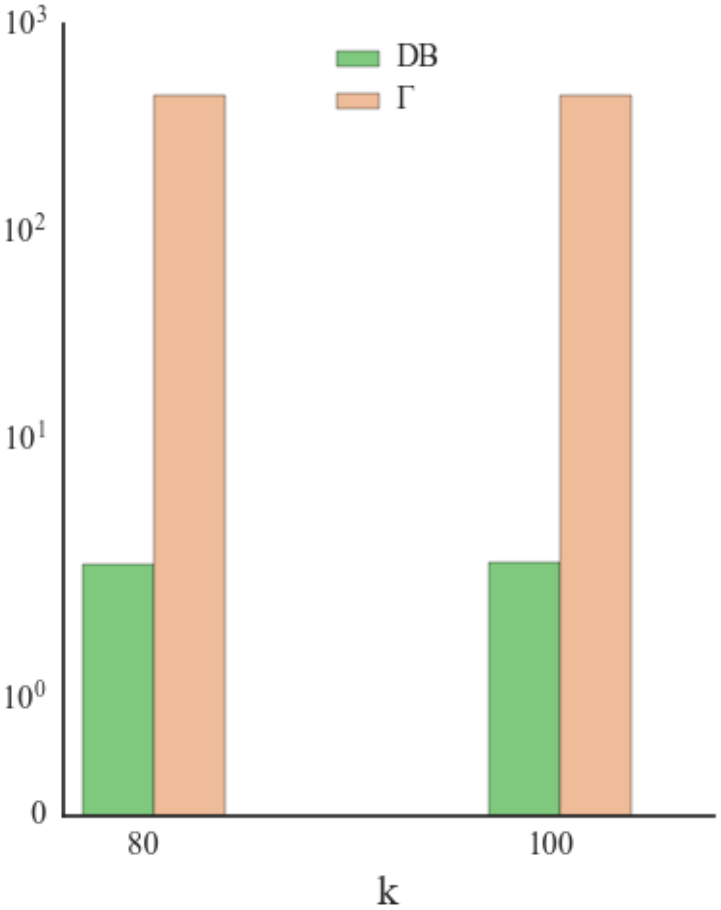


Σχήμα 60: Οι τιμές των δεικτών DB και Γ του τρισδιάστατου k-Means για την τομή στον άξονα των χρωμοσωμάτων.

Όπως φαίνεται στο Σχήμα 60, οι τιμές του δείκτη DB είναι αρκετά υψηλές ενώ του δείκτη Γ αρκετά χαμηλές κατά τη συσταδοποίηση πάνω στον άξονα των χρωμοσωμάτων, ενώ παίρνουν τις χειρότερες τιμές τους για $k = 2$, κάτι που είναι αναμενόμενο, αφού στην

περίπτωση αυτή έχουμε δύο συστάδες γονιδίων με αρκετά μεγάλη διακύμανση των προφίλ έκφρασης. Σε γενικές γραμμές, πάντως, βλέπουμε ότι η επίδοση του αλγορίθμου στην τομή αυτή δεν είναι ιδιαίτερα καλή, όπως άλλωστε διαπιστώνουμε και με μια απλή επισκόπηση των τελικών συστάδων (Σχήματα 26-29).

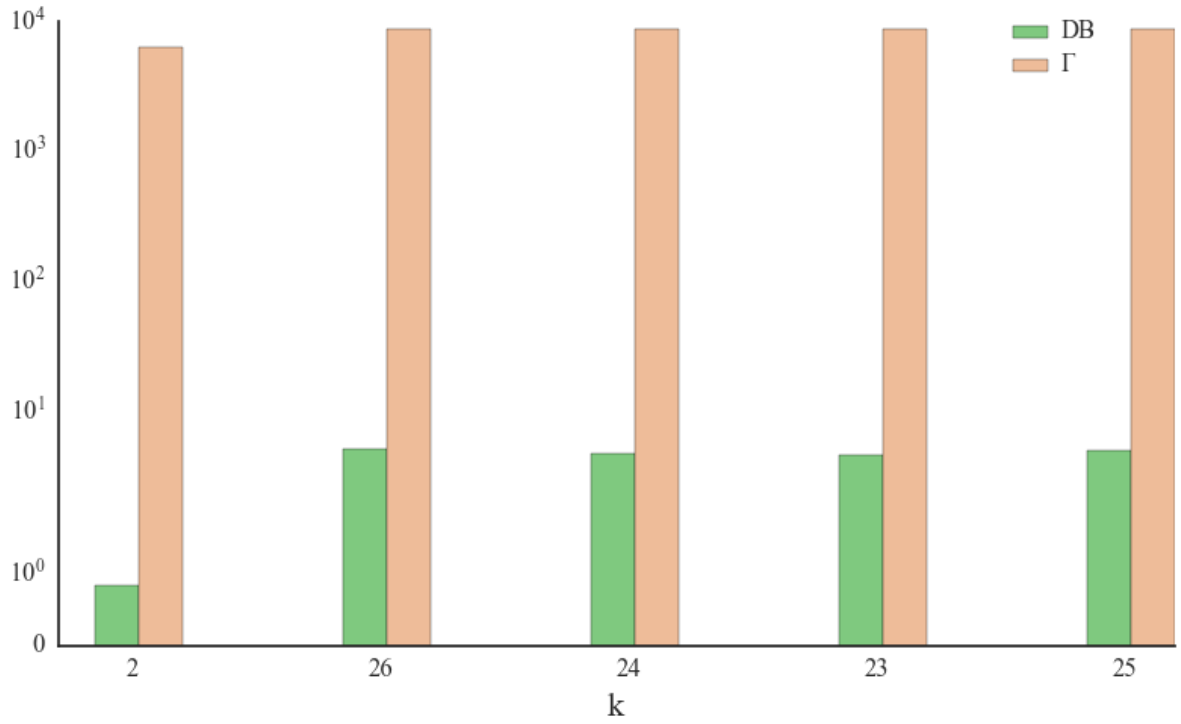
- Τομή στον άξονα των γονιδίων



Σχήμα 61: Οι τιμές των δεικτών DB και Γ του τρισ-διάστατου k-Means για την τομή στον άξονα των γονιδίων.

Στην περίπτωση αυτή οι τιμές των δεικτών βελτιώνονται αρκετά, με τον δείκτη DB να πέφτει χαμηλότερα και τον δείκτη Γ να αυξάνεται. Αξίζει να τονίσουμε ότι δεν παρατηρούνται μεγάλες διαφορές ανάμεσα στις δύο τιμές του *k*, κάτι που είναι επίσης αναμενόμενο αφού και οι δύο συσταδοποιήσεις είναι αρκετά χαμηλής ποιότητας.

- Τομή στον άξονα των δειγμάτων

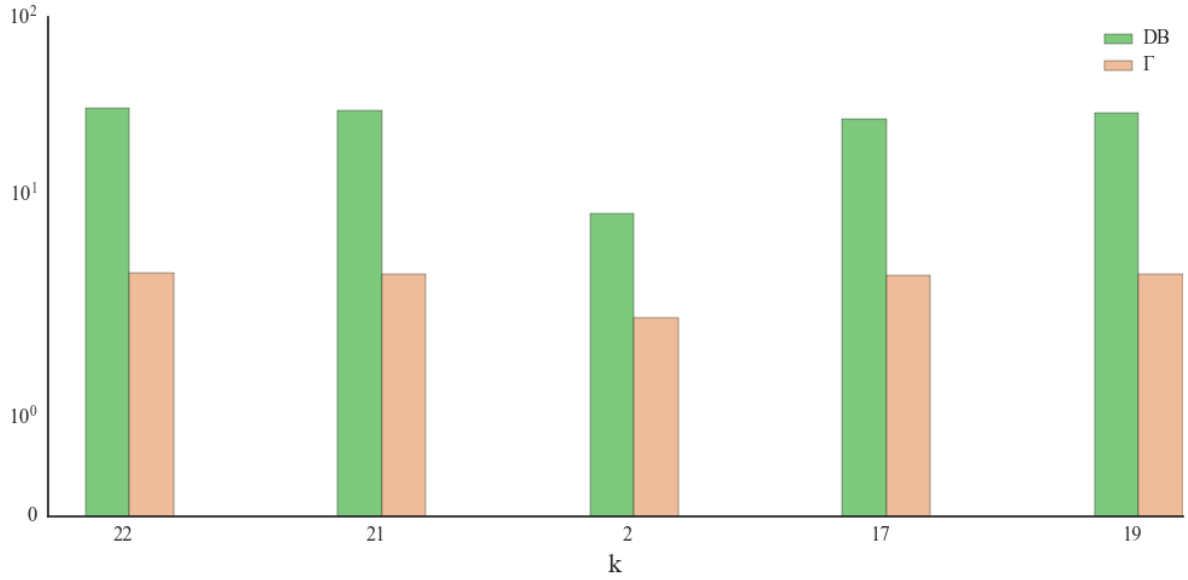


Σχήμα 62: Οι τιμές των δεικτών DB και Γ του τρισδιάστατου k-Means για την τομή στον άξονα των δειγμάτων.

Στην τελευταία αυτή τομή φαίνεται πως έχουμε τα καλύτερα αποτελέσματα, αφού ο δείκτης DB λαμβάνει τις χαμηλότερες και αντίστοιχα ο δείκτης Γ τις υψηλότερες τιμές του. Ιδιαίτερα για k = 2 έχουμε το βέλτιστο συνδυασμό τιμών, κάτι που είναι ιδιαίτερα ενδιαφέρον, αφού ελέγχοντας τις συστάδες διαπιστώνουμε ότι ο αλγόριθμος έχει ξεχωρίσει ακριβώς τα δύο σύνολα δεδομένων. Ομοίως, για k = 26, ο αλγόριθμος έχει ομαδοποιήσει σε μία συστάδα τα δείγματα ελέγχου του CodeLink, για k = 24 έχει δημιουργήσει μια συστάδα με τα δείγματα ελέγχου του GSE7476 και άλλη μία με δύο από τα low grade δείγματά του, για k = 23 έχει φτιάξει πάλι μια συστάδα με τα δείγματα ελέγχου του GSE7476, μία με δύο high grade και μία με δύο low grade δείγματα του ίδιου dataset και τέλος, για k = 25 ο k-Means έχει διαμορφώσει μια συστάδα με τα δείγματα ελέγχου του GSE7476. Παρατηρούμε λοιπόν ότι υπάρχει μια εγγενής ομοιότητα ανάμεσα στο είδος των δειγμάτων που βρίσκονται σε κάθε συστάδα.

Κλασικός k-Means στα κεντροειδή του τρισδιάστατου πίνακα

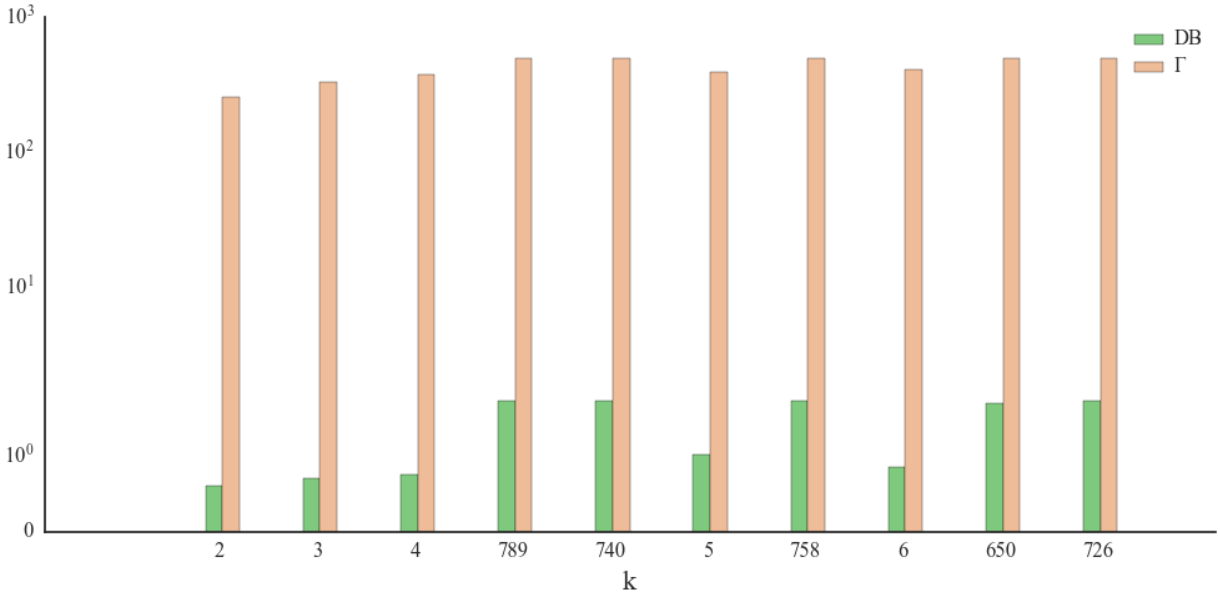
- Τομή στον άξονα των χρωμοσωμάτων



Σχήμα 63: Οι τιμές των δεικτών DB και Γ του κλασικού k-Means για την τομή στον άξονα των χρωμοσωμάτων.

Όπως και στην περίπτωση του τροποποιημένου k-Means, έτσι κι εδώ παρατηρούμε ότι οι δείκτες DB και Γ αποκτούν εξαιρετικά άσχημες τιμές για την τομή στον άξονα των χρωμοσωμάτων. Σύμφωνα με τα όσα εξηγήσαμε παραπάνω, αλλά και με μία απλή επισκόπηση των συστάδων που προκύπτουν, καταλαβαίνουμε ότι το γεγονός αυτό οφείλεται κυρίως στη μεγάλη διακύμανση που παρουσιάζουν τα προφίλ έκφρασης των γονιδίων μέσα στο ίδιο ζεύγος χρωμοσωμάτων, με αποτέλεσμα το κεντροειδές αυτού του ζεύγους να μην αποτελεί αξιόπιστο αντιπρόσωπο για την περαιτέρω συσταδοποίηση.

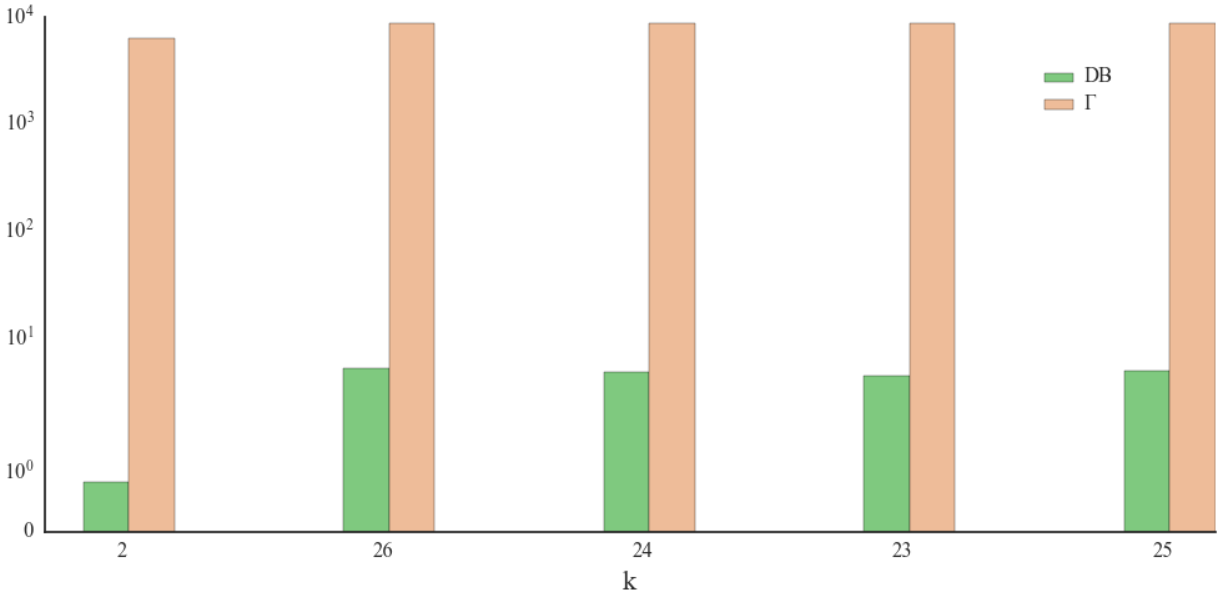
- Τομή στον άξονα των γονιδίων



Σχήμα 64: Οι τιμές των δεικτών DB και Γ του κλασικού k-Means για την τομή στον άξονα των γονιδίων.

Σε αυτή την τομή, οι τιμές των δεικτών είναι αρκετά βελτιωμένες, κάτι που άλλωστε φαίνεται και με μια απλή ματιά στις συστάδες που προκύπτουν.

- Τομή στον άξονα των δειγμάτων

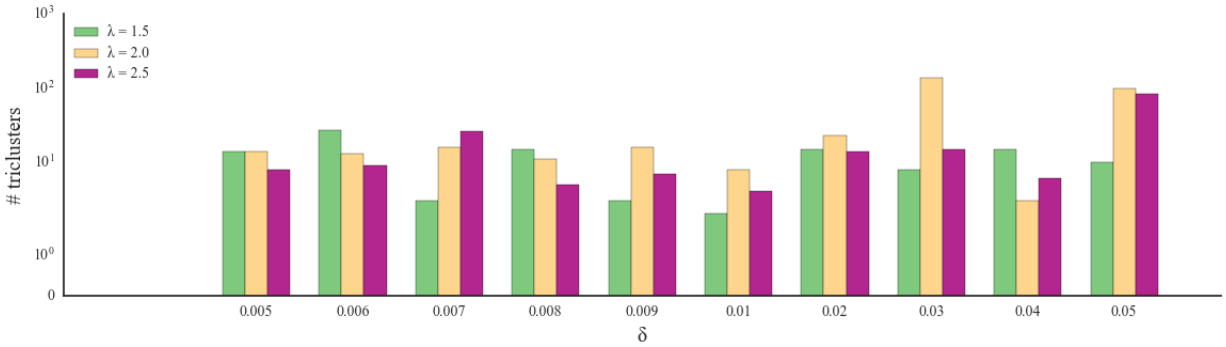


Σχήμα 65 Οι τιμές των δεικτών DB και Γ του κλασικού k-Means για την τομή στον άξονα των δειγμάτων.

Εδώ ο κλασικός k-Means πέτυχε παρόμοιο διαχωρισμό των δειγμάτων με τον τροποποιημένο k-Means που εφαρμόσαμε πρωτύτερα. Συγκεκριμένα, για $k = 2$ ο αλγόριθμος κατόρθωσε να ξεχωρίσει τα δύο dataset σε διαφορετικές συστάδες, ενώ στις υπόλοιπες τιμές του k ομαδοποίησε δείγματα ελέγχου, δείγματα low grade ή high grade, κάτι που διαθέτει ιδιαίτερο ενδιαφέρον από βιολογικής άποψης.

δ-TRIMAX

Στο επόμενο σχήμα φαίνεται ο αριθμός των συστάδων που ανακάλυψε ο δ-TRIMAX σε κάθε συνδυασμό των παραμέτρων που χρησιμοποιήσαμε.

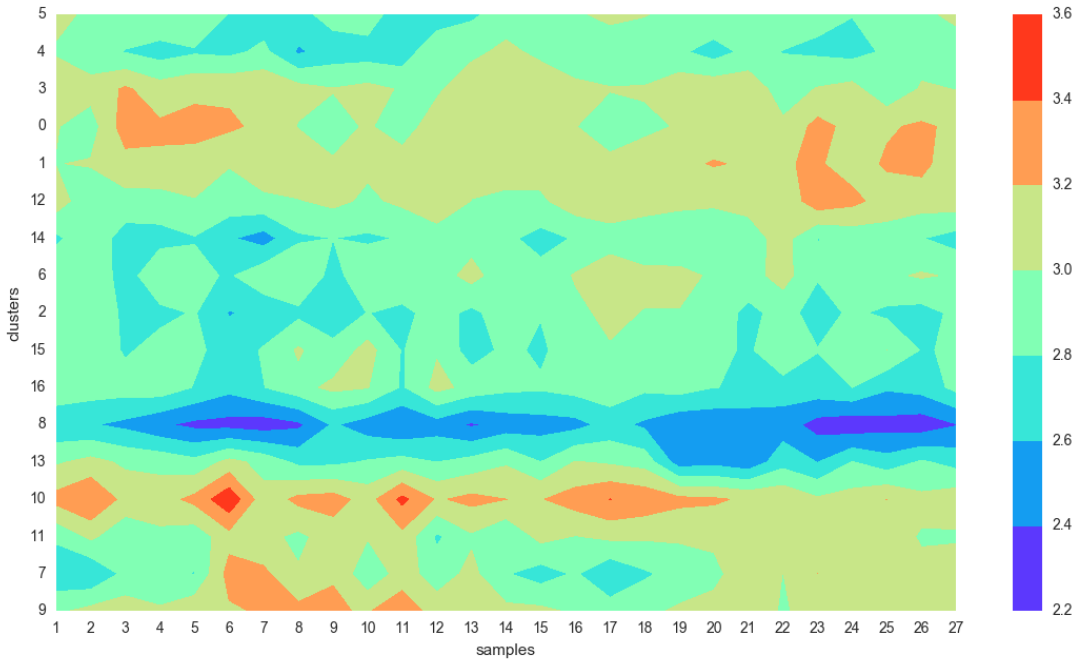


Σχήμα 66: Ο αριθμός των συστάδων που επέστρεψε ο δ-TRIMAX σε διάφορους συνδυασμούς των παραμέτρων δ και λ.

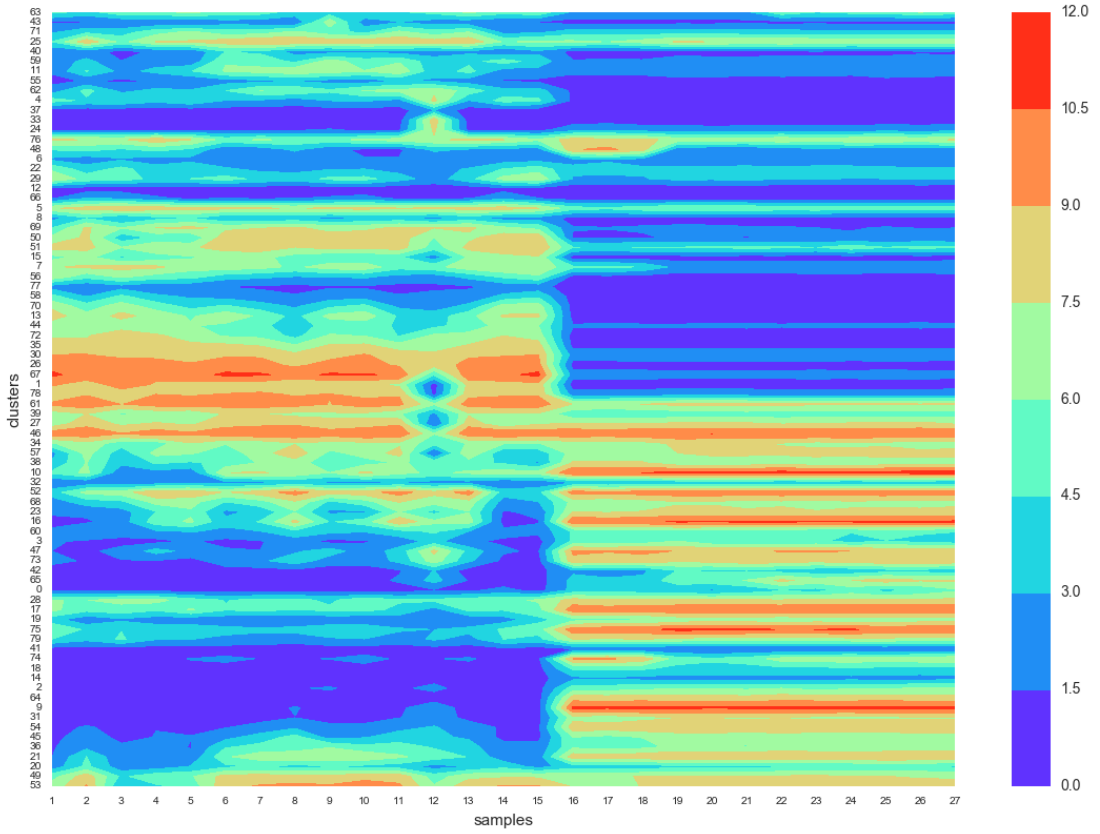
Είναι εμφανές ότι αυξάνοντας τις τιμές της παραμέτρου δ ο αλγόριθμος εντόπιζε όλο και περισσότερες συστάδες, με μεγαλύτερη όμως διακύμανση στις τιμές τους. Με άλλα λόγια, χαλαρώνει σταδιακά η συνθήκη να ανήκει ένα κελί του πίνακα σε κάποια συστάδα. Όσον αφορά το εύρος τιμών που επιλέξαμε, θεωρήσαμε ότι για $\delta > 0.05$ οι συστάδες έπαυαν να είναι ικανοποιητικά συμπαγείς.

Επιφάνειες κεντροειδών

Μία ακόμη ενδιαφέρουσα προσέγγιση των αποτελεσμάτων είναι η απεικόνιση των ισοϋψών καμπυλών (contour lines) που παράγονται αν τοποθετήσουμε τα κεντροειδή κάθε συστάδας σε μια επιφάνεια. Παραδείγματα τέτοιων γραφημάτων φαίνονται στα σχήματα 67-69 για κάποιες από τις περιπτώσεις που μελετήσαμε στην ενότητα Δ.



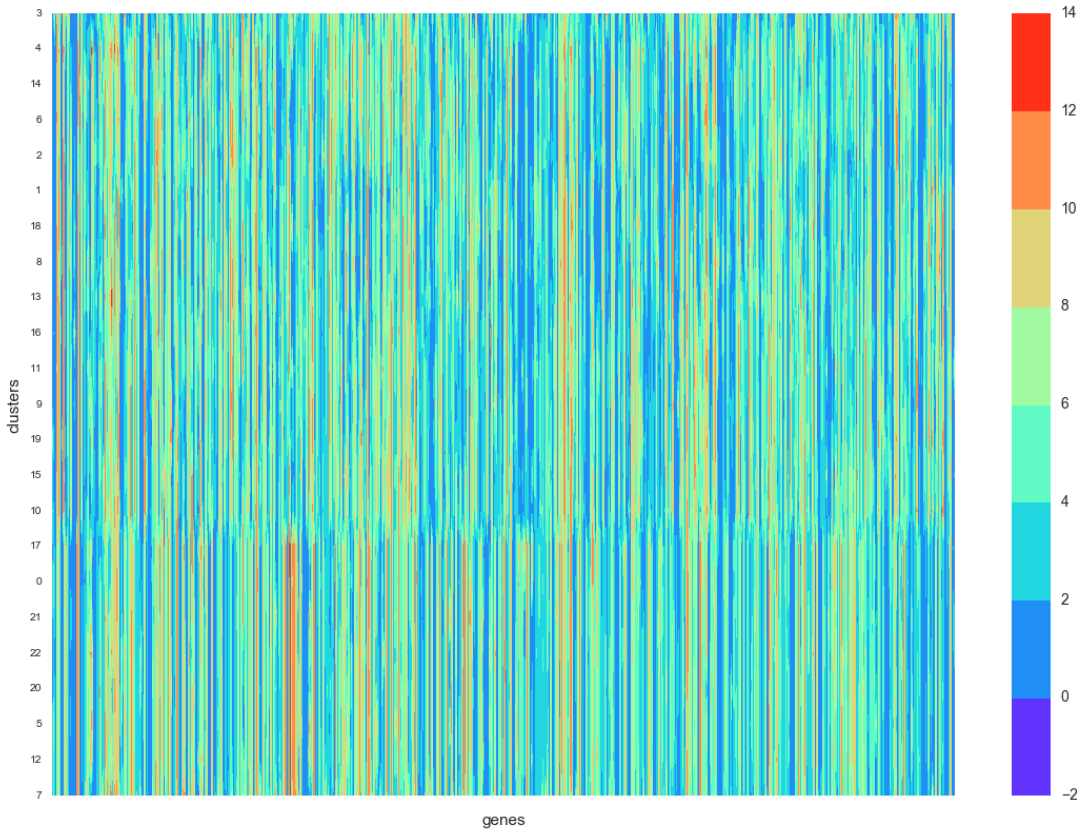
Σχήμα 67: Contour plot της επιφάνειας που δημιουργήθηκε από τα κεντροειδή των συστάδων κατά μήκος του άξονα των χρωμ/των για k = 17.



Σχήμα 68: Contour plot της επιφάνειας που δημιουργήθηκε από τα κεντροειδή των συστάδων κατά μήκος του άξονα των γονιδίων για k = 80.

Η δημιουργία αυτών των διαγραμμάτων έγινε με υπολογισμό της συσχέτισης κατά Pearson ανάμεσα στα κεντροειδή των συστάδων που επέστρεψε σε κάθε περίπτωση ο τροποποιημένος k-Means αλγόριθμος. Στη συνέχεια υπολογίστηκε μια βέλτιστη διάταξη των κεντροειδών με χρήση του αλγορίθμου *repetitive k-NN* [60] ώστε αυτά να έχουν τη μέγιστη μεταξύ τους συσχέτιση, κι έτσι να σχηματίζουν συμπαγή βουνά και κοιλάδες στο τελικό διάγραμμα. Στον κατακόρυφο άξονα των παραγόμενων contour plots αντιστοιχούν τα κεντροειδή των συστάδων και στον οριζόντιο τα δείγματα ή τα γονίδια, ενώ οι χρωματισμοί αντιπροσωπεύουν τα επίπεδα γονιδιακής έκφρασης σύμφωνα με το υπόμνημα.

Παρατηρούμε, λοιπόν, ότι τα τελικά διαγράμματα εμφανίζουν στην πλειοψηφία των περιπτώσεων ευδιάκριτα μοτίβα, γεγονός που καθιστά ιδιαίτερα ενδιαφέρουσα την περαιτέρω μελέτη και διερεύνησή τους.



Σχήμα 69: Contour plot της επιφάνειας που δημιουργήθηκε από τα κεντροειδή των συστάδων κατά μήκος του άξονα των δειγμάτων για k = 23.

Ε. ΣΥΖΗΤΗΣΗ

Στην παρούσα εργασία έγινε προσπάθεια ανάπτυξης μιας νέας μεθόδου για τη συσταδοποίηση δεδομένων γονιδιακής έκφρασης από μικροσυστοιχίες DNA, η οποία λαμβάνει υπ' όψη τη χρωμοσωμική ταυτότητα των γονιδίων. Αρχικά κατασκευάστηκε μια τρισδιάστατη δομή της μορφής *χρωμοσωμικά_ζεύγη* x *γονίδια* x *δείγματα* και στη συνέχεια εφαρμόστηκαν σε αυτή δυο διαφορετικοί αλγόριθμοι συσταδοποίησης.

Ο πρώτος αλγόριθμος ήταν μια παραλλαγή του k-Means, ο οποίος διαθέτει την ικανότητα να εξετάζει και να ομαδοποιεί δισδιάστατους πίνακες αντί διανυσμάτων. Εφαρμόζοντάς τον στα δεδομένα μας διαπιστώσαμε ότι αντιμετώπιζε αρκετά μεγάλη δυσκολία στη σύγκλιση, ενώ οι συστάδες που επέστρεφε απείχαν πολύ από τις βέλτιστες, ανεξάρτητα της τομής κατά την οποία έγινε η συσταδοποίηση. Το γεγονός αυτό μπορεί να εξηγηθεί αν σκεφτούμε ότι οι νόρμες πινάκων που χρησιμοποιεί ο αλγόριθμος ως μετρικές απόστασης εμπεριέχουν στους υπολογισμούς πολύ περισσότερες τιμές απ' ό,τι οι αντίστοιχες διανυσματικές νόρμες, αυξάνοντας έτσι τη διαστατικότητα και δυσχεραίνοντας την ορθή σύγκριση. Για παράδειγμα, έστω ότι έχουμε δύο διανύσματα (γονίδια) *a,b* διαστάσεων *n* (για *n* δείγματα) το καθένα. Η Ευκλείδεια απόσταση που χρησιμοποιεί ο κλασικός k-Means θα είχε τη μορφή:

$$\|a-b\| \;=\; \sqrt{\sum_{i=1}^{n}(a_i-b_i)^2}$$

Αντιθέτως, η νόρμα Fröbenius που χρησιμοποιεί αντίστοιχα ο τροποποιημένος k-Means για δύο πίνακες *A, B* διαστάσεων *m* x *n* ο καθένας, θα είχε τη μορφή:

$$\|A-B\|_F \;=\; \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}-b_{ij}|^2}$$

Αυτό έχει ως αποτέλεσμα ο δεύτερος αλγόριθμος να λαμβάνει υπ' όψη πολύ περισσότερες τιμές έκφρασης από τον πρώτο σε κάθε υπολογισμό, κι έτσι τελικά δυσκολεύεται η σύγκριση και η εκτίμηση της απόστασης μεταξύ δύο datapoints.

Στη συνέχεια εφαρμόσαμε τον κλασικό k-Means στα κεντροειδή του τρισδιάστατου πίνακα. Η μέθοδος αυτή απέφερε αρκετά καλύτερα αποτελέσματα από την προηγούμενη κυρίως στον άξονα των γονιδίων, αφού κατά την κατασκευή του πίνακα είχε ήδη γίνει μια πρώτη

ομαδοποίηση των γονιδίων που παρουσιάζουν παρόμοιο προφίλ έκφρασης, κι έτσι το κεντροειδές της κάθε ομάδας αποτέλεσε ικανό αντιπρόσωπο. Αντιθέτως, στην τομή του άξονα των χρωμοσωμάτων οι συσταδοποιήσεις δεν ήταν εξίσου καλές, κάτι που ήταν εν μέρει αναμενόμενο αφού η μέθοδος *f*(*K*) είχε επιστρέψει ως πρώτη τιμή τον πλήρη αριθμό των χρωμ/κών ζευγών. Όσον αφορά τον άξονα των δειγμάτων, ενώ η μέθοδος *f*(*K*) είχε επιστρέψει πάλι ως πρώτη τιμή τον πλήρη αριθμό δειγμάτων, ο k-Means ανακάλυψε ενδιαφέρουσες συσταδοποιήσεις με διαφορετικά *k*, κατορθώνοντας να διαχωρίσει καρκινικά από υγιή δείγματα, ή να συγκεντρώσει σε μια ομάδα δείγματα ίδιου βαθμού επιθετικότητας.

Τέλος, εφαρμόσαμε τον αλγόριθμο συσταδοποίησης υποχώρου δ-TRIMAX, ο οποίος διαθέτει την ικανότητα να εντοπίζει συστάδες ταυτόχρονα και στις τρεις διαστάσεις του πίνακα. Τα αποτελέσματα της μεθόδου αυτής ήταν ιδιαίτερα εντυπωσιακά, αφού οι συστάδες που επέστρεφε κάθε φορά ήταν αρκετά υψηλής ποιότητας.

Έχει αναφερθεί σε προηγούμενες μελέτες ότι η πιθανή συσχέτιση κατά Pearson στη γονιδιακή έκφραση μπορεί να υποδηλώνει κοινή γονιδιακή ρύθμιση [61]. Υπενθυμίζουμε ότι στην κλασική ανάλυση της γονιδιακής έκφρασης σε πειράματα μικροσυστοιχιών, τόσο η ταξινόμηση όσο και η γραμμική συσχέτιση αφορούσε πίνακες της μορφής *γονίδια* x *δείγματα*. Ως εκ τούτου, μια πιθανή συσχέτιση μεταξύ δύο γονιδίων σε όλα τα υπό διερεύνηση δείγματα υποδήλωνε και κοινή ρύθμιση. Ταυτοχρόνως, είναι επίσης αποδεκτό ότι η συσχέτιση δεν αποτελεί απαραίτητα και αιτιότητα (*correlation does not mean causality*). Σε προηγούμενη μελέτη είχε γίνει προσπάθεια εξέτασης της γονιδιακής έκφρασης σε δείγματα νεοπλασιών του μαστού με βάση τη χρωμοσωμική κατανομή, όπου είχε δειχθεί ότι υπάρχει συσχέτιση της γονιδιακής έκφρασης ως προς το κάθε χρωμόσωμα και μάλιστα μπορούσε να ταξινομήσει με μεγαλύτερη ακρίβεια τα γονιδιακά μοτίβα στην υπό διερεύνηση νεοπλασία [62]. Στην παρούσα εργασία, η προσπάθεια μελέτης της γονιδιακής έκφρασης σε τρεις διαστάσεις αναφέρεται για πρώτη φορά στη βιβλιογραφία και συνδράμει προς την κατεύθυνση της ταυτοποίησης αιτιότητας στη γονιδιακή έκφραση.

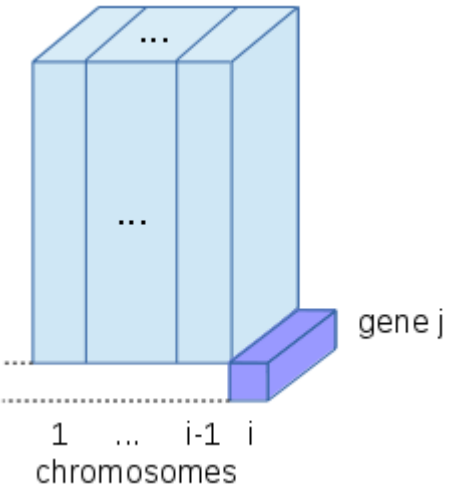
Επίσης, η προσέγγιση που ακολουθήσαμε στην παρούσα εργασία παρέχει περισσότερες πληροφορίες. Ειδικότερα, η ταξινόμηση ως προς τη διάσταση των γονιδίων (Σχήμα 11α) δίνει πιο ακριβή πληροφορία ως προς την κοινή έκφραση μεταξύ των χρωμοσωμάτων σε όλα τα υπό διερεύνηση δείγματα. Επίσης το ίδιο επιτυγχάνεται με την ταξινόμηση ως προς τη διάσταση των χρωμοσωμάτων (Σχήμα 11γ), η οποία μάλιστα προσομοιάζει την αρχική δισδιάστατη προσέγγιση, με τη διαφορά ότι γίνεται για κάθε ζεύγος χρωμοσωμάτων μεμονωμένα, τμηματίζοντας το πρόβλημα σε μικρότερες οντότητες. Τέλος, η ανάλυση ως

προς τη διάσταση των δειγμάτων (Σχήμα 11β) επιτρέπει τη μεμονωμένη μελέτη της γονιδιακής έκφρασης σε κάθε δείγμα.

Πέραν των παραπάνω, η παρούσα προσέγγιση επιτρέπει και μια ακόμη ανάλυση της γονιδιακής έκφρασης, δηλαδή τη συσχέτιση των επιφανειών των τριών διαστάσεων. Πιο συγκεκριμένα, κάθε σχηματιζόμενη επιφάνεια αποτελεί τη γονιδιακή έκφραση της αντίστοιχης διάστασης, η οποία μπορεί να δώσει περαιτέρω πληροφορίες για τη γονιδιακή ρύθμιση στις υπό διερεύνηση μεταβλητές (Σχήματα 67-69).

Κλείνοντας, θα πρέπει να τονίσουμε ότι στην προκείμενη εργασία πραγματοποιήθηκε μια σχετικά σύντομη ανάλυση πάνω στα χαρακτηριστικά και τις ιδιότητες που διέπουν τη νέα αυτή μέθοδο που αναπτύξαμε. Είναι όμως εμφανές ότι η εν λόγω μέθοδος επιδέχεται πολλών ακόμη βελτιώσεων, ενώ παράλληλα προσφέρει άπλετο χώρο για περαιτέρω έρευνα και πειραματισμό. Στις επόμενες παραγράφους θα προτείνουμε κάποιες ιδέες για μελλοντική επέκταση ή/και παραλλαγές της.

Κατασκευή τρισδιάστατου πίνακα με επέκταση προς τα κάτω



Σχήμα 70: Παράδειγμα τοποθέτησης του j γονιδίου στο τέλος του i-στού ζεύγους χρωμ/των.

Ένας εναλλακτικός τρόπος δημιουργίας της τρισδιάστατης δομής είναι η επέκτασή της προς τα κάτω, χωρίς περιορισμό στο μήκος. Πιο συγκεκριμένα, κάθε φορά που εξετάζουμε τον πίνακα για την τοποθέτηση ενός νέου γονιδίου και δεν εντοπίζουμε κάποιο γονίδιο ή κεντροειδές “αρκετά κοντά” στο εξεταζόμενο (εδώ μπορεί να γίνει χρήση κάποιας τιμής κατωφλίου), τότε θα υπάρχει η δυνατότητα τοποθέτησής του στο τέλος του συγκεκριμένου ζεύγους χρωμ/των.

Στο Σχήμα 70 δίνεται ένα παράδειγμα αυτής της τεχνικής. Υποθέτουμε ότι κατά την εξέταση του γονιδίου *j* που ανήκει στο *i*-στό ζεύγος χρωμ/των διαπιστώνουμε ότι το προφίλ έκφρασής του δεν ταιριάζει με τα προφίλ των γονιδίων που έχουμε μέχρι στιγμής εισάγει στον πίνακα (στα προηγούμενα *i*-1 ζεύγη χρωμ/των), κι έτσι το τοποθετούμε στο τέλος.

Χρήση διαφορετικών μετρικών απόστασης

Μια ακόμη ιδέα θα ήταν η χρήση εναλλακτικών μετρικών απόστασης στους αλγορίθμους συσταδοποίησης που εφαρμόσαμε. Για παράδειγμα, θα ήταν εφικτό να χρησιμοποιηθούν άλλες αποστάσεις στον κλασικό k-Means αντί της Ευκλείδειας,, π.χ. απόσταση Manhattan, Chebyshev, Mahalanobis, κλπ. Επίσης, στον τρισδιάστατο k-Means θα μπορούσαν να χρησιμοποιηθούν άλλες, μη τετριμμένες νόρμες πινάκων οι οποίες ίσως και να έδιναν καλύτερα αποτελέσματα και ταχύτερη σύγκλιση του αλγορίθμου.

Χρήση διαφορετικών αλγορίθμων συσταδοποίησης

Θα ήταν ακόμη αρκετά ενδιαφέρον να μελετηθεί η συμπεριφορά εναλλακτικών αλγορίθμων συσταδοποίησης πλην των k-Means και δ-TRIMAX πάνω στην τρισδιάστατη δομή, όπως για παράδειγμα αλγορίθμων ασαφούς συσταδοποίησης ή Γκαουσιανών Μοντέλων. Επίσης, θα μπορούσε να γίνει χρήση κάποιου διαφορετικού αλγόριθμου συσταδοποίησης υποχώρου σε τρεις διαστάσεις, ο οποίος θα παρέχει τη δυνατότητα εξεύρεσης διαφορετικού τύπου συστάδων, όπως επικαλυπτόμενων (overlapping) ή ιεραρχικών (hierarchical) [24]. Θα μπορούσε ακόμη και να εφαρμοστεί ο βελτιωμένος αλγόριθμος EMOA-δ-TriMax [63], ο οποίος συνδυάζει τη μέθοδο δ-TRIMAX με διαδικασίες γενετικής προσομοίωσης και επιτυγχάνει αρκετά καλύτερα αποτελέσματα.

Ανεύρεση αιτιολογικών σχέσεων

Περαιτέρω μελέτη θα μπορούσε να επικεντρωθεί στην ανεύρεση αιτιολογικών σχέσεων μεταξύ των μεταβλητών και ειδικότερα η ταξινόμηση ενός διανύσματος σε μία διάσταση θα μπορούσε να βοηθήσει στη διερεύνηση της παλινδρόμησης μεταξύ των υπόλοιπων διανυσμάτων. Πιο συγκεκριμένα, αν υποθέσουμε ότι θα θέλαμε να διερευνήσουμε την πιθανή αιτιολογική σχέση στη διάσταση των γονιδίων, τότε η ταξινόμησή τους για το πρώτο ζεύγος χρωμοσωμάτων θα οδηγούσε σε αναδιάταξη του πίνακα στην ίδια διάσταση και θα επέτρεπε την παλινδρόμηση των υπό διερεύνηση μεταβλητών. Να σημειώσουμε ότι στα πλαίσια της γονιδιακής έκφρασης η συσχέτιση δύο γονιδίων δεν οδηγεί πάντα στην παλινδρόμησή τους, αφού οι παραγόμενες απεικονίσεις δεν είναι *ένα προς ένα* ή/και *επί*. Για να μελετηθεί μια τέτοια συσχέτιση απαιτείται η τροποποίηση των δεδομένων. Στο βαθμό που μας είναι γνωστό δεν έχουν προταθεί στη βιβλιογραφία τέτοιου είδους προσεγγίσεις για τη γονιδιακή ανάλυση.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] J. D. Watson και F. H. C. Crick, ‘Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid’, *Nature*, τ. 171, τχ. 4356, σσ 737–738, Απριλίου 1953.

[2] F. Crick, ‘Central Dogma of Molecular Biology’, *Nature*, τ. 227, τχ. 5258, σσ 561–563, Αυγούστου 1970.

[3] Genetic Science Learning Center, University of Utah, ‘DNA Microarray’.

[4] ‘FGED: MIAME’. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: http://fged.org/projects/miame/. [Ημερομηνία πρόσβασης: 15-Μαρτίου-2016].

[5] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[6] S. S. Haykin και S. S. Haykin, *Neural networks and learning machines*, 3rd ed. New York: Prentice Hall, 2009.

[7] S. Theodoridis και K. Koutroumbas, *Pattern recognition*, 4. ed. Amsterdam: Elsevier Acad. Press, 2009.

[8] S. Lloyd, ‘Least squares quantization in PCM’, *IEEE Trans. Inf. Theory*, τ. 28, τχ. 2, σσ 129–137, Μαρτίου 1982.

[9] G. H. Ball και D. J. Hall, ‘A clustering technique for summarizing multivariate data’, *Behav. Sci.*, τ. 12, τχ. 2, σσ 153–155, Μαρτίου 1967.

[10] J. MacQueen, ‘Some methods for classification and analysis of multivariate observations’, *Proc Fifth Berkeley Symp Math Stat. Prob Univ Calif Press 1967*, τ. 1, σσ 281–297, 1967.

[11] M. Mahajan, P. Nimbhorkar, και K. Varadarajan, ‘The Planar k-Means Problem is NP-Hard’, στο *WALCOM: Algorithms and Computation*, τ. 5431, S. Das και R. Uehara, Επιμ. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, σσ 274–285.

[12] Arthur, D και Vassilvitskii, S., ‘k-means++: the advantages of careful seeding’, *Proc. Eighteenth Annu. ACM-SIAM Symp. Discrete Algorithms Soc. Ind. Appl. Math. Phila. PA USA*, σσ 1027–1035.

[13] D. T. Pham, S. S. Dimov, και C. D. Nguyen, ‘Selection of K in K-means clustering’, *Proc. Inst. Mech. Eng. Part C J. Mech. Eng. Sci.*, τ. 219, τχ. 1, σσ 103–119, Ιανουαρίου 2005.

[14] K. Beyer, J. Goldstein, R. Ramakrishnan, και U. Shaft, ‘When Is “Nearest Neighbor” Meaningful?’, στο *Database Theory — ICDT’99*, τ. 1540, C. Beeri και P. Buneman, Επιμ.

Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, σσ 217–235.

[15] Y. Cheng και G. M. Church, ‘Biclustering of expression data’, *Proc. Int. Conf. Intell. Syst. Mol. Biol. ISMB Int. Conf. Intell. Syst. Mol. Biol.*, τ. 8, σσ 93–103, 2000.

[16] A. K. Jain και R. C. Dubes, *Algorithms for clustering data*. Englewood Cliffs, N.J: Prentice Hall, 1988.

[17] L. Hubert και J. Schultz, ‘QUADRATIC ASSIGNMENT AS A GENERAL DATA ANALYSIS STRATEGY’, *Br. J. Math. Stat. Psychol.*, τ. 29, τχ. 2, σσ 190–241, Νοεμβρίου 1976.

[18] D. L. Davies και D. W. Bouldin, ‘A Cluster Separation Measure’, *IEEE Trans. Pattern Anal. Mach. Intell.*, τ. PAMI-1, τχ. 2, σσ 224–227, Απριλίου 1979.

[19] O. Arbelaitz, I. Gurrutxaga, J. Muguerza, J. M. Pérez, και I. Perona, ‘An extensive comparative study of cluster validity indices’, *Pattern Recognit.*, τ. 46, τχ. 1, σσ 243–256, Ιανουαρίου 2013.

[20] Lambrou, G. I., Adamaki, M., Koultouki, E., και Moschovi, M., ‘Systems Biology Methodologies for the Understanding of Common Oncogenetic Mechanisms in Childhood Leukemic and Rhabdomyosarcoma Cells’, στο *Quality Assurance in Healthcare Service Delivery, Nursing and Personalized Medicine: Technologies and Processes*, IGI Global, 2012, σσ 111–168.

[21] Jiang, D., Tang, C., και Zhang, A., ‘Cluster Analysis for Gene Expression Data: A Survey’, *EEE Trans. Knowl. Data Eng.*, τ. 16, σσ 370–1386, 2004.

[22] Z. R. Yang, *Machine learning approaches to bioinformatics*. Hackensack, NJ: World Scientific, 2010.

[23] Aidong Zhang, *Advanced analysis of gene expression microarray data*, 1η έκδ., τ. 1. World Scientific Publishing Co. Pte. Ltd., 2006.

[24] S. C. Madeira και A. L. Oliveira, ‘Biclustering algorithms for biological data analysis: a survey’, *IEEEACM Trans. Comput. Biol. Bioinforma. IEEE ACM*, τ. 1, τχ. 1, σσ 24–45, Μαρτίου 2004.

[25] Y. Kluger, ‘Spectral Biclustering of Microarray Data: Coclustering Genes and Conditions’, *Genome Res.*, τ. 13, τχ. 4, σσ 703–716, Απριλίου 2003.

[26] L. Yin, C.-H. Huang, και J. Ni, ‘Clustering of gene expression data: performance and similarity analysis’, *BMC Bioinformatics*, τ. 7, τχ. Suppl 4, σ S19, 2006.

[27] P. D haeseleer, ‘How does gene expression clustering work’, *Nat. Biotechnol.*, τ. 23, τχ. 12, σσ 1499–1501, Δεκεμβρίου 2005.

[28] P. Mahanta, H. A. Ahmed, D. K. Bhattacharyya, και J. K. Kalita, ‘Triclustering in gene

expression data analysis: A selected survey', 2011, σσ 1–6.

[29] L. Zhao και M. J. Zaki, 'TRICLUSTER: an effective algorithm for mining coherent clusters in 3D microarray data', 2005, σ 694.

[30] A. Bhar, M. Haubrock, A. Mukhopadhyay, U. Maulik, S. Bandyopadhyay, και E. Wingender, 'Coexpression and coregulation analysis of time-series gene expression data in estrogen-induced breast cancer cell', *Algorithms Mol. Biol.*, τ. 8, τχ. 1, σ 9, 2013.

[31] A. Ciaramella, S. Cocozza, F. Iorio, G. Miele, F. Napolitano, M. Pinelli, G. Raiconi, και R. Tagliaferri, 'Interactive data analysis and clustering of genomic data', *Neural Netw.*, τ. 21, τχ. 2–3, σσ 368–378, Μαρτίου 2008.

[32] D. Gutierrez-Aviles, C. Rubio-Escudero, και J. C. Riquelme, 'Triclustering on temporary microarray data using the TriGen algorithm', 2011, σσ 877–881.

[33] R. Braga Araújo, G. H. Trielli Ferreira, G. H. Orair, W. Meira, R. A. Celso Ferreira, D. Olavo Guedes Neto, και M. J. Zaki, 'The ParTriCluster Algorithm for Gene Expression Analysis', *Int. J. Parallel Program.*, τ. 36, τχ. 2, σσ 226–249, Απριλίου 2008.

[34] D. Jiang, J. Pei, M. Ramanathan, C. Tang, και A. Zhang, 'Mining coherent gene clusters from gene-sample-time microarray data', 2004, σ 430.

[35] A. B. Tchagang, S. Phan, F. Famili, H. Shearer, P. Fobert, Y. Huang, J. Zou, D. Huang, A. Cutler, Z. Liu, και Y. Pan, 'Mining biological information from 3D short time-series gene expression data: the OPTricluster algorithm', *BMC Bioinformatics*, τ. 13, τχ. 1, σ 54, 2012.

[36] S. Mankad και G. Michailidis, 'Biclustering Three-Dimensional Data Arrays With Plaid Models', *J. Comput. Graph. Stat.*, τ. 23, τχ. 4, σσ 943–965, Οκτωβρίου 2014.

[37] A. Li και D. Tuck, 'An effective tri-clustering algorithm combining expression data with gene regulation information', *Gene Regul. Syst. Biol.*, τ. 3, σσ 49–64, 2009.

[38] 'History and License — Python 3.5.1 documentation'. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://docs.python.org/3.5/license.html. [Ημερομηνία πρόσβασης: 27-Μαρτίου-2016].

[39] Eric Jones, Travis Oliphant, Pearu Peterson, και et al., *SciPy: Open Source Scientific Tools for Python*. 2001.

[40] S. van der Walt, S. C. Colbert, και G. Varoquaux, 'The NumPy Array: A Structure for Efficient Numerical Computation', *Comput. Sci. Eng.*, τ. 13, τχ. 2, σσ 22–30, Μαρτίου 2011.

[41] Wes McKinney, 'Data Structures for Statistical Computing in Python', *Proc. 9th Python Sci. Conf.*, σσ 51–56, 2010.

[42] F. Perez και B. E. Granger, ‘IPython: A System for Interactive Scientific Computing’, *Comput. Sci. Eng.*, τ. 9, τχ. 3, σσ 21–29, 2007.

[43] J. D. Hunter, ‘Matplotlib: A 2D Graphics Environment’, *Comput. Sci. Eng.*, τ. 9, τχ. 3, σσ 90–95, 2007.

[44] Michael Waskom και et al., *seaborn: v0.7.0*. 2016.

[45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, και Édouard Duchesnay, ‘Scikit-learn: Machine Learning in Python’, *J. Mach. Learn. Res.*, τ. 12, σσ 2825–2830, 2011.

[46] Pierre Raybaut, Carlos Cordoba, και et al., *Spyder - The Scientific PYthon Development EnviRonment*. .

[47] A. Zaravinos, G. I. Lambrou, I. Boulalas, D. Delakas, και D. A. Spandidos, ‘Identification of Common Differentially Expressed Genes in Urinary Bladder Cancer’, *PLoS ONE*, τ. 6, τχ. 4, σ e18135, Απριλίου 2011.

[48] L. Mengual, M. Burset, E. Ars, J. J. Lozano, H. Villavicencio, M. J. Ribal, και A. Alcaraz, ‘DNA Microarray Expression Profiling of Bladder Cancer Allows Identification of Noninvasive Diagnostic Markers’, *J. Urol.*, τ. 182, τχ. 2, σσ 741–748, Αυγούστου 2009.

[49] D. Diez, R. Alvarez, και A. Dopazo, ‘Codelink: an R package for analysis of GE healthcare gene expression bioarrays’, *Bioinformatics*, σσ 1168–1169, Μαρτίου 2007.

[50] D. Amaratunga και J. Cabrera, ‘Analysis of Data From Viral DNA Microchips’, *J. Am. Stat. Assoc.*, τ. 96, τχ. 456, σσ 1161–1170, Δεκεμβρίου 2001.

[51] ‘Quantile normalization’, *Wikipedia, the free encyclopedia*. 23-Μαρτίου-2016.

[52] B. Bolstad, ‘Probe Level Quantile Normalization of High Density Oligonucleotide Array Data’.

[53] B. M. Bolstad, R. . Irizarry, M. Astrand, και T. P. Speed, ‘A comparison of normalization methods for high density oligonucleotide array data based on variance and bias’, *Bioinformatics*, τ. 19, τχ. 2, σσ 185–193, Ιανουαρίου 2003.

[54] W. Wu, N. Dave, G. C. Tseng, T. Richards, E. P. Xing, και N. Kaminski, ‘Comparison of normalization methods for CodeLink Bioarray data’, *BMC Bioinformatics*, τ. 6, σ 309, 2005.

[55] T. Hastie, R. Tibshirani, G. Sherlock, M. Eisen, P. Brown, και D. Botstein, ‘Imputing Missing Data for Gene Expression Arrays’, Division of Biostatistics, Stanford University,

Technical report, 1999.

[56] O. Troyanskaya, M. Cantor, G. Sherlock, P. Brown, T. Hastie, R. Tibshirani, D. Botstein, και R. B. Altman, ‘Missing value estimation methods for DNA microarrays’, *Bioinformatics*, τ. 17, τχ. 6, σσ 520–525, Ιουνίου 2001.

[57] R. Malarvizhi και A. S. Thanamani, ‘K-Nearest Neighbor in Missing Data Imputation’, *Int. J. Eng. Res. Dev.*, τ. 5, τχ. 1, Νοεμβρίου 2012.

[58] G. Cumming, F. Fidler, και D. L. Vaux, ‘Error bars in experimental biology’, *J. Cell Biol.*, τ. 177, τχ. 1, σσ 7–11, Απριλίου 2007.

[59] Ανέτα Ν. Νούσια, ‘Νόρμες Πινάκων και Διανυσμάτων σε Χώρους Πεπερασμένης Διάστασης’, Σχολή Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών, Εθνικό Μετσόβιο Πολυτεχνείο, Αθήνα, 2011.

[60] ‘Approximation of maximum weight hamiltonian paths’. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: http://www.ctl.ua.edu/math103/hamilton/quick.htm. [Ημερομηνία πρόσβασης: 18-Ιανουαρίου-2016].

[61] B. A. Cohen, R. D. Mitra, J. D. Hughes, και G. M. Church, ‘A computational analysis of whole-genome expression data reveals chromosomal domains of gene expression’, *Nat. Genet.*, τ. 26, τχ. 2, σσ 183–186, Οκτωβρίου 2000.

[62] F. Reyal, ‘Visualizing Chromosomes as Transcriptome Correlation Maps: Evidence of Chromosomal Domains Containing Co-expressed Genes--A Study of 130 Invasive Ductal Breast Carcinomas’, *Cancer Res.*, τ. 65, τχ. 4, σσ 1376–1383, Φεβρουαρίου 2005.

[63] A. Bhar, M. Haubrock, A. Mukhopadhyay, και E. Wingender, ‘Multiobjective triclustering of time-series transcriptome data reveals key genes of biological processes’, *BMC Bioinformatics*, τ. 16, τχ. 1, Δεκεμβρίου 2015.

[64] ‘The MIT License (MIT) | Open Source Initiative’. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://opensource.org/licenses/MIT. [Ημερομηνία πρόσβασης: 08-Ιουνίου-2016].

ΠΑΡΑΡΤΗΜΑ

Παραθέτουμε τα βασικά τμήματα του κώδικα που αναπτύξαμε για τις ανάγκες της παρούσας εργασίας, ο οποίος διατίθεται υπό άδεια MIT [64] ως ελεύθερο, ανοικτού κώδικα λογισμικό. Σημειώνουμε ότι, όπως κάθε έργο προγραμματισμού, έτσι και ο παρών κώδικας βρίσκεται σε διαρκή αναθεώρηση και επιδιόρθωση, επομένως για την τελευταία έκδοσή του παραπέμπουμε στο δημόσιο αποθετήριο: **https://github.com/paren8esis/thesis**

Προεπεξεργασία δεδομένων

```
# -*- coding: utf-8 -*-

import numpy as np
from sklearn import neighbors
import pandas as pd


def quantile_normalization(data, method='mean'):
    """
    Quantile normalization of data.
    """
    data_norm = data.copy()
    I = np.argsort(data_norm.ix[:, 1:], axis=0)
    if (method == 'median'):
       data_medians = np.float64(np.nanmedian(data_norm.ix[:, 1:].values[I,
                          np.arange(data_norm.shape[1]-1)], axis=1)[:, np.newaxis])
       data_norm.ix[:, 1:].values[I, np.arange(data_norm.shape[1]-1)] = data_medians
    else:
       data_means = np.float64(np.nanmean(data_norm.ix[:, 1:].values[I,
                          np.arange(data_norm.shape[1]-1)],axis=1)[:, np.newaxis])
       data_norm.ix[:, 1:].values[I, np.arange(data_norm.shape[1]-1)] = data_means

    return data_norm


def kNN_imputation(a, k=1, imp_method='mean', metric='euclidean',
                   **metric_params):
    """
    Performs value imputation using the k Nearest Neighbors algorithm.

    For all missing (i.e. NaN) values in data, we find the k Nearest
    Neighbors and then replace the NaN value with a weighted average of the
    found neighbors.

    data is of the form: genes x samples
    Suppose gene i contains a NaN value in sample j. This function chooses k
    genes with non-missing values in sample j, nearest to gene i (i.e. those
    genes that have the closest expression profiles to gene i in the remaining
    samples). Also, only genes with complete columns can be neighbors. Then, it
    uses the average value of those k neighbors in sample j to fill in the
    NaN value.

    Parameters
    ----------
    a : pandas.DataFrame
        The data in which we will perform the iputation.
```

```
k : int, optional
    The number of nearest neighbors to take into account.
imp_method : {'mean', 'median'}, optional
            The method of imputation.
metric : {'cityblock', 'cosine', 'euclidean', 'l1', 'l2',
          'manhattan', 'braycurtis', 'canberra', 'chebyshev',
          'correlation', 'dice', 'hamming', 'jaccard',
          'kulsinski', 'mahalanobis', 'matching', 'minkowski',
          'rogerstanimoto', 'russellrao', 'seuclidean',
          'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'}, optional
          The distance metric to be used in the k Nearest Neighbors
          algorithm.
metric_params : dict, optional
                Additional keyword arguments for the metric function.

Returns
-------
pandas.DataFrame
    The given DataFrame with the NaN values imputed.

References
----------
.. [1] P. Jonsson, C. Wohlin, 2004. An Evaluation of kNearest Neighbour
       Imputation Using Likert Data, Proceedings of the 10th International
       Symposium on Software Metrics, Chicago, IL, (USA), pp. 108 – 118

.. [2] O. Troyanskaya, M. Cantor, G. Sherlock, P. Brown, T. Hastie,
       R. Tibshirani, D. Botstein, and R. B. Altman, (2001). Missing value
       estimation methods for DNA microarrays, Bioinformatics,
       17 (6): 520-525 doi:10.1093/bioinformatics/17.6.520
"""

# Check if given parameters are correct
if (imp_method not in ['mean', 'median']):
    print("Error: Invalid method")
    return
if (metric not in ['cityblock', 'cosine', 'euclidean', 'l1', 'l2',
                   'manhattan', 'braycurtis', 'canberra', 'chebyshev',
                   'correlation', 'dice', 'hamming', 'jaccard',
                   'kulsinski', 'mahalanobis', 'matching', 'minkowski',
                   'rogerstanimoto', 'russellrao', 'seuclidean',
                   'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']):
    print("Error: Invalid metric")
    return

# Get a copy of the original array
a_imputed = a.values.copy()

# Find nans in original array
nanVals = np.isnan(a_imputed)

# Find rows that do not contain any nans
noNans = np.logical_not(np.any(nanVals, axis=1))

# Keep rows from the original array that do not contain any nans
dataNoNans = a_imputed[noNans, :]

# If there are no such rows, print error message
if (dataNoNans.size == 0):
    print("Error: There are no rows without NaN values.")
    return

# Find indices of nan values in original matrix
(nan_rows, nan_cols) = np.nonzero(nanVals)

rowWarn = np.zeros((a_imputed.shape[0], 1))

nan_rows, slices = np.unique(nan_rows, return_index=True)
```

```
    nan_cols = np.split(nan_cols, slices[1:])

    knn = neighbors.NearestNeighbors(n_neighbors=k+1,
                                     metric=metric,
                                     metric_params=metric_params)

    # For each nan row
    for nan_row in range(nan_rows.size):
        # Check if the row contains only nans
        if (np.all(np.isnan(a_imputed[nan_rows[nan_row], :]))):
            if (rowWarn[nan_rows[nan_row]] == 0):
                print("Warning: row {0} contains only " +
                + "NaN values.".format(nan_rows[nan_row]))
               rowWarn[nan_rows[nan_row]] = 1
            continue

        # Find all columns that do not correspond to any nan values of the
        # nan row
        complete_cols = [x for x in range(a_imputed.shape[1]) if x not in
                                  nan_cols[nan_row]]

        knn.fit(np.vstack((a_imputed[nan_rows[nan_row], complete_cols],
                           dataNoNans[:, complete_cols])))
        neighs = knn.kneighbors(a_imputed[nan_rows[nan_row],
                                          complete_cols].reshape(1, -1),
                                n_neighbors=k+1,
                                return_distance=False)

        # We ignore the first neighbor - it's the reference vector itself
        neighs = neighs[0][1:]
        # Impute values
        for nan_col in nan_cols[nan_row]:
            if (imp_method == 'mean'):
               a_imputed[nan_rows[nan_row], nan_col] = np.mean(a_imputed[neighs-1,
                                                                        nan_col])
            else:
               a_imputed[nan_rows[nan_row], nan_col] = np.median(a_imputed[neighs-1,
                                                                          nan_col])

    a[:] = a_imputed
    return a
```

Κατασκευή τρισδιάστατου πίνακα

```
# -*- coding: utf-8 -*-

import numpy as np
from scipy.spatial import distance


class Array3D():
    """
    A 3-dimensional array structure of the form
        n_chromosomes x n_genes x n_samples

    Attributes
    ----------
    X : list of ndarray
        A list containing all genes grouped by chromosome.
    indices : list of lists
        A list containing n_chromosomes lists with the indices of the genes
        in the corresponding chromosomes in X.
    prev_chrom : int
        The index of the previous chromosome examined.
    metric : str or callable, optional
```

```
    The distance metric to use.  If a string, the distance function can be
    'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation',
    'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski',
    'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao',
    'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
    'wminkowski', 'yule'.
method : {'onebyone', 'centroids'}, default 'centroids'
    The method to be used in order to group together the genes.
    'centroids' means that every time we examine the genes in
    chromosome i, we compute the distances of every gene to the set
    centroids we've already found.
    'onebyone' means that every time we examine the genes in
    chromosome i, we compute the distances of every gene to those in
    chromosome i-1.
sets : ndarray
    This attribute contains either the last genes of each set (if method is
    'onebyone') or the centroids of each set (if method is 'centroids').
centroids : dict, int->list of tuples
    This dictionary holds the set id as key and a list containing tuples
    of the form (chromosome, gene_index) for each gene in the set.
array3D : ndarray
    The final 3D array structure.
indices3D : ndarray
    The gene labelsof the final 3D array structure.
array3D_size : int
    The final 3D array's size of the gene axis.

Methods
-------
create(X, metric='euclidean', method='centroids'):
    Create the 3D array from the data in X.
"""

def __init__(self, X, indices=None):
    """
    Parameters
    ----------
    X : list of ndarray
        A list containing all genes grouped by chromosome.
    indices : list of lists
        A list containing n_chromosomes lists with the indices of the genes
        in the corresponding chromosomes in X.
    """
    if (X is None):
        raise Exception("Error: No data provided!")
    else:
        self.X = X.copy()   # list of ndarray
        self.array3D_size = max([self.X[x].shape[0] for x in
                                 range(len(self.X))])
        self.indices = indices.copy()

def _group_genes(self, chrom):
    """
    Group genes in given chrom to sets.

    Parameters
    ----------
    chrom : int
        The chromosome of which to group the genes.
    """
    distances = np.ma.masked_array(distance.cdist(self.X[chrom],
                                                  self.sets,
                                                  self.metric))

    n_genes_chrom = self.X[chrom].shape[0]
    n_genes_prev = self.sets.shape[0]
    genes_chrom = np.array([-1] * n_genes_chrom)
    genes_prev = np.array([-1] * n_genes_prev)
```

```
        n_iterations = min(n_genes_chrom, n_genes_prev)
        for i in range(n_iterations):
            closest = np.unravel_index(distances.argmin(), distances.shape)
            distances[closest[0], :] = np.ma.masked
            distances[:, closest[1]] = np.ma.masked

            # Assign new gene to the appropriate set
            if (self.method == 'centroids'):
                self.centroids[closest[1]].append((chrom, closest[0]))
            self.sets[closest[1]] = self.X[chrom][closest[0]]
            genes_chrom[closest[0]] = closest[1]
            genes_prev[closest[1]] = closest[0]

        if (n_genes_chrom > n_genes_prev):
            # Add the rest of the genes of the current chromosome
            genes_left = np.where(genes_chrom == -1)[0]
            self.sets = np.vstack((self.sets,
                                   self.X[chrom].take(genes_left, axis=0)))
            if (self.method == 'centroids'):
                set_id = n_genes_prev
                for gene_left in genes_left:
                    self.centroids[set_id].append((chrom, gene_left))
                    set_id += 1

        # Update the 3D array
        new_slice = self.sets.copy()
        new_slice[np.where(genes_prev == -1)] = [np.nan] * self.X[chrom].shape[1]
        self.array3D = np.dstack((self.array3D, new_slice))

        # Update the indices of the 3D array
        if (self.indices is not None):
            if (n_genes_chrom > n_genes_prev):
                chrom_ind = np.append(genes_prev, genes_left).tolist()
            else:
                chrom_ind = genes_prev.tolist()
            for i in range(len(chrom_ind)):
                if (chrom_ind[i] != -1):
                    chrom_ind[i] = self.indices[chrom][chrom_ind[i]]
                else:
                    chrom_ind[i] = None
            self.indices3D += [chrom_ind]

    def _reevaluate_centroids(self):
        """
        Calculates the centroids of every slice.
        """
        new_centroids = []

        for set_id in self.centroids.keys():
            set_centroid = []
            for set_gene in self.centroids[set_id]:
                set_centroid.append(self.X[set_gene[0]][set_gene[1]])
            new_centroids.append(np.mean(np.asarray(set_centroid), axis=0))

        self.sets = np.asarray(new_centroids)

    def create(self, metric='euclidean', method='centroids'):
        """
        Create the 3D array with the data in X.

        Parameters
        ----------
        metric : {'euclidean', 'cityblock', 'cosine', 'correlation',
              'hamming', 'jaccard', 'chebyshev', 'canberra',
              'braycurtis', 'yule', 'matching', 'dice',
              'kulsinski', 'rogerstanimoto', 'russellrao',
              'sokalsneath', 'wminkowski', 'fractional'}, default 'euclidean'
              The distance metric to be used
```

```
        method : {'centroids', 'onebyone'}, default 'centroids'
            The method to be used in order to group together the genes.
            'centroids' means that every time we examine the genes in
            chromosome i, we compute the distances of every gene to the set
            centroids we've already found.
            'onebyone' means that every time we examine the genes in
            chromosome i, we compute the distances of every gene to those in
            chromosome i-1.

        Returns
        -------
        array3D : ndarray
            The final 3D array structure.
        indices3D : ndarray
            The gene labelsof the final 3D array structure.
        """
        self.metric = metric
        self.method = method

        # Initialize 3D array
        self.sets = np.array(self.X[0])
        self.array3D = self.X[0]
        if (self.indices is not None):
            self.indices3D = [self.indices[0]]
        else:
            self.indices3D = None

        # Add padding to both array3D and indices
        padding = np.array(self.X[0].shape[1] * [np.nan])[np.newaxis, :]
        for i in range(self.array3D_size - self.X[0].shape[0]):
            self.array3D = np.vstack((self.array3D, padding))

        if (method == 'onebyone'):
            for i in range(1, len(self.X)):
                self._group_genes(i)
        else:
            self.centroids = {}
            for i in range(self.array3D_size):
                if (i < self.X[0].shape[0]):
                    self.centroids[i] = [(0, i)]
                else:
                    self.centroids[i] = []
            for i in range(1, len(self.X)):
                self._group_genes(i)
                self._reevaluate_centroids()

        self.array3D = self.array3D.swapaxes(0, 2)
        self.array3D = self.array3D.swapaxes(1, 2)
        # Referece as: [chrom, gene, sample]

        return self.array3D, self.indices3D
```

Τροποποιημένος k-Means

```
# -*- coding: utf-8 -*-

import numpy as np
import random
from scipy.spatial import distance
from sympy import floor

import warnings
```

```
class KMeans3D():
    """
    The k-means clustering algorithm.

    Attributes
    ----------
    X : ndarray
        The data to be clustered
    n_chroms : int
        The number of chromosomes
    n_genes : int
        The number of genes
    n_samples : int
        The number of samples
    K : int
        The number of clusters to be determined
    order : {non-zero int, inf, -inf, 'fro', 'nuc'}, default 2
        Order of the norm to be used as distance metric.
        See numpy.linalg.norm documentation for more details.
    mode : {0, 1, 2}, default 1
        Determines the axis along which to do the clustering.
        mode=i means that the clustering will be done along axis i.
    mu : ndarray
        The cluster centers
    oldmu : ndarray
        The previous values of cluster centers
    clusters : dict
        Cluster id mapped to a list of the cluster's elements
    method : {'random', 'k-means++'}
        The method of cluster centers initialization
    tol : float
        The algorithm's tolerance
    best_inertia : float
        The algorithm's inertia
    best_clusters : dict
        Same as clusters, but keeps the best clustering we've found so far
    best_labels : ndarray
        Same as labels but keeps the best clustering we've found so far
    best_mu : ndarray
        Same as mu but keeps the best clustering we've found so far
    distances : ndarray
        A 2xm array containg for each data point its closest cluster center
        and the distance from it
    fs : list of float
        The f(k) values for each k
    D2 : ndarray
        Vector of the D^2 weighting for the k-means++ algorithm.


    Methods
    -------
    find_centers(method='random', order=2, mode=1, n_times=10,
                 tol=1e-4, max_iter=300, K=1, verbose=False, init_centers=None)
        Run the k-means3D algorithm with the given parameters.
    get_clusters()
        Return a dictionary cluster_id->list_of_cluster_elements
    get_labels()
        Return an ndarray containing the cluster id for each element
    get_best_k()
        Return a list of the most appropriate values of k
    get_centers()
        Return an ndarray containing the cluster centers
    get_centroids()
        Return an ndarray containing the cluster centroids
    fk()
        Run the f(k) method in order to determine suitable values for k

    References
```

```
    ----------
    .. [1] Lloyd, Stuart P. "Least Squares Quantization in PCM."
           IEEE Transactions on Information Theory. Vol. 28, 1982, pp. 129–137
    .. [2] Arthur, David, and Sergi Vassilvitskii. "K-means++: The Advantages
           of Careful Seeding." SODA ‘07: Proceedings of the Eighteenth Annual
           ACM-SIAM Symposium on Discrete Algorithms. 2007, pp. 1027–1035
    .. [3] Pham D. T., Dimov S. S., and Nguyen C. D. "Selection of K in
           k-means clustering." DOI: 10.1243/095440605X8298, 2004
    .. [4] Davies D., Bouldin D. "A cluster separation measure", IEEE
           Transactions on Pattern Analysis and Machine Intelligence, Vol. 1,
           pp. 224-227, doi: 10.1109/TPAMI.1979.4766909
    """

    def __init__(self, X):
        """
        Parameters
        ----------
        X : ndarray (3D Array)
            The data to be clustered

        Notes
        -----
        Updates the X, n_chroms, n_genes, n_samples, N, mu, clusters
        and method attributes.
        """
        if (X is None):
            raise Exception("Error: No data provided!")
        else:
            self.X = X

        self.mu = None  # ndarray
        self.clusters = None
        self.method = None

        self.n_chroms = X.shape[0]
        self.n_genes = X.shape[1]
        self.n_samples = X.shape[2]

    def _cluster_points(self):
        """
        Compute the distance of each data point from all cluster centers
        and find the nearest.

        Returns
        -------
        bool
            True if an empty cluster was found, False otherwise.

        Notes
        -----
        Updates the clusters, labels and distances attributes.
        """
        clusters = {}

        X = self.X
        mu = self.mu

        distances = np.empty((1, 2))
        for i in range(X.shape[self.mode]):  # For each slice of X
            if (self.mode == 0):
                distances_x =
np.linalg.norm(np.absolute(np.nan_to_num(X[i,:,:].reshape(1, self.n_genes,
self.n_samples)-mu)), ord=self.order, axis=(1,2))
            elif (self.mode == 1):
                distances_x =
np.linalg.norm(np.absolute(np.nan_to_num(X[:,i,:].reshape(self.n_chroms, 1,
self.n_samples)-mu)), ord=self.order, axis=(0,2))
            else:
```

```
                distances_x =
np.linalg.norm(np.absolute(np.nan_to_num(X[:,:,i].reshape(self.n_chroms,
self.n_genes, 1)-mu)), ord=self.order, axis=(0,1))

            distances = np.vstack((distances, [np.nanmin(distances_x),
                                               np.nanargmin(distances_x)]))

        distances = distances[1:, :]
        empty_cluster_found = False
        for clust in range(self.K):
            clusters[clust] = np.where(distances[:, 1] == clust)[0].tolist()
            if (len(clusters[clust]) == 0):
                empty_cluster_found = True

        self.clusters = clusters
        self.labels = np.asarray(distances[:, 1], dtype=int)
        self.distances = distances

        return empty_cluster_found

    def _reevaluate_centers(self):
        """
        Re-evaluates the cluster centers for the Update phase of the algorithm.

        Notes
        -----
        Updates the mu attribute.
        """
        clusters = self.clusters
        if (self.mode == 0):
            newmu = np.empty((1, self.n_genes, self.n_samples))
        elif (self.mode == 1):
            newmu = np.empty((self.n_chroms, 1, self.n_samples))
        else:
            newmu = np.empty((self.n_chroms, self.n_genes, 1))

        keys = sorted(self.clusters.keys())
        for k in keys:
            with warnings.catch_warnings():  # We expect mean of NaNs here
                warnings.simplefilter("ignore", category=RuntimeWarning)
                newmu_i = np.nanmean(self.X.take(clusters[k], axis=self.mode),
                                     axis=self.mode)  # Could contain NaNs

            if (self.mode == 0):
                newmu_i = newmu_i.reshape((1, self.n_genes, self.n_samples))
                newmu = np.vstack((newmu, newmu_i))
            elif (self.mode == 1):
                newmu_i = newmu_i.reshape((self.n_chroms, 1, self.n_samples))
                newmu = np.hstack((newmu, newmu_i))
            else:
                newmu_i = newmu_i.reshape((self.n_chroms, self.n_genes, 1))
                newmu = np.dstack((newmu, newmu_i))

        if (self.mode == 0):
            self.mu = newmu[1:, :, :]
        elif (self.mode == 1):
            self.mu = newmu[:, 1:, :]
        else:
            self.mu = newmu[:, :, 1:]

    def _has_converged(self):
        """
        Returns True if the algorithm has converged, i.e. the difference
        between new cluster centers and previous ones is less than the
        given tolerance.

        Returns
        -------
```

```
        bool
            True if the algorithm has converged, False otherwise.

        """
        mu = np.nan_to_num(self.mu)
        oldmu = np.nan_to_num(self.oldmu)
        d = np.ravel(oldmu - mu)
        return np.dot(d, d) <= self.tol

    def find_centers(self, method='random', order=2, mode=1, n_times=10,
                     tol=1e-4, max_iter=300, K=1, verbose=False,
                     init_centers=None):
        """
        Run the k-means3D algorithm with the given parameters.

        Parameters
        ----------
        method : {'random', 'k-means++'}, default 'random'
            The method to be used for cluster centers initialization
        order : {non-zero int, inf, -inf, 'fro', 'nuc'}, default 2
            Order of the norm to be used as distance metric.
            See numpy.linalg.norm documentation for more details.
        mode : {0, 1, 2}, default 1
            Determines the axis along which to do the clustering.
            mode=i means that the clustering will be done along axis i.
        n_times : int, default 10
            Number of times for the k-means algorithm to be run
        tol : float, default 1e-4
            The tolerance. If the difference between the previous cluster
            centers and the new ones is less than the tolerance, then the
            algorithm converges
        max_iter : int, default 300
            The maximum number of iterations for the algorithm
        K : int, default 1
            The number of clusters
        verbose : bool, default False
            Verbose mode for debugging
        init_centers : ndarray
            The user can give an initialization of the cluster centers to be
            used by the algorithm.

        Notes
        -----
        Updates the K, method, metric, tol, best_inertia, best_clusters,
        best_labels, best_mu, distances, labels, clusters and oldmu attributes.
        """
        self.K = K
        self.method = method
        self.order = order
        self.mode = mode
        self.tol = tol

        # Check the parameters given
        if (len(self.X) == 0):
            raise ValueError("Please provide valid data to the algorithm.")
        if (floor(self.K) != self.K) or (self.K <= 0):
            raise ValueError("'k' must be an integer > 0, but its value"
                             " is {}".format(self.K))
        if (self.method not in ['random', 'k-means++']):
            raise ValueError("'method' must be either 'random' or 'k-means++',"
                             " but its value is {}".format(self.delta))
        if (floor(max_iter) != max_iter) or (max_iter < 0):
            raise ValueError("'max_iter' must be an integer > 0, but its value"
                             " is {}".format(self.K))

        self.best_inertia = None
        self.best_clusters = None
        self.best_labels = None
```

```
        self.best_mu = None

        self.distances = np.asarray([[None], [None]])
        self.labels = None
        self.clusters = None

        # Run k-means algorithm n_times
        for i in range(n_times):
            if (verbose):
                print(i)

            self.oldmu = self.X.take(np.random.randint(self.X.shape[self.mode],
                                                       size=self.K),
                                     axis=self.mode)  # Could contain NaNs

            # Initialize centers
            if (init_centers is None):
                self._init_centers()
            else:
                self.mu = init_centers

            # E-M steps
            for j in range(max_iter):
                if (not self._has_converged()):
                    self.oldmu = self.mu
                    # Assign all points in X to clusters
                    empty_cluster_found = self._cluster_points()
                    if (empty_cluster_found):
                        print("   Empty cluster found! Exiting...")
                        break
                    # Reevaluate centers
                    self._reevaluate_centers()
                else:
                    if (verbose):
                        print("   Converged in iteration {}".format(j))
                    break

            # Calculate inertia (Sum of distances of samples to their closest
            # cluster center)
            if (not empty_cluster_found):
                inertia = np.sum(self.distances[:, 0])
                if (self.best_inertia is None) or (inertia < self.best_inertia):
                    self.best_inertia = inertia
                    self.best_labels = self.labels
                    self.best_clusters = self.clusters
                    self.best_mu = self.mu

    def get_clusters(self):
        """
        Return the clusters formed.

        Returns
        -------
            best_clusters : dict
                Dictionary of the form: cluster_id->list_of_points_of_cluster
        """
        return self.best_clusters

    def get_labels(self):
        """
        Return the labels found.

        Returns
        -------
        best_labels : ndarray
            The id of the cluster that each data point belongs to
        """
        return np.array(self.best_labels)
```

```
    def get_centers(self):
        """
        Return the values of the cluster centers.

        Returns
        -------
        best_mu : ndarray
            The values of the cluster centers found
        """
        return self.best_mu

    def get_centroids(self):
        """
        Return the cluster centroids.

        Returns
        -------
        centroids : ndarray
            The cluster centroids
        """
        if (self.best_mu is None):
            return None
        with warnings.catch_warnings():  # We expect mean of NaNs here
            warnings.simplefilter("ignore", category=RuntimeWarning)
            if (self.mode == 0):
                return np.nanmean(self.best_mu, axis=1)
            elif (self.mode == 1):
                return np.nanmean(self.best_mu, axis=0)
            else:
                return np.rollaxis(np.nanmean(self.best_mu, axis=0), 1)

    def get_best_k(self):
        """
        Return the most suitable values for k.

        Returns
        -------
        best_k : list of int
            The values of k in priority order (those with the smallest f(k) are
            listed first)
        """
        return [i+1 for i in np.argsort(self.fs)]

    def _dist_from_centers(self):
        """
        Calculate the distance of each data point from each cluster center.

        Notes
        -----
        Updates the D2 attribute.
        """
        X = self.X
        cent = self.mu
        D2 = np.array([])

        for i in range(X.shape[self.mode]):  # For each slice of X
            if (self.mode == 0):
                D2_x =
np.power(np.linalg.norm(np.absolute(np.nan_to_num(X[i,:,:].reshape(1,self.n_genes,se
lf.n_samples)-cent)), ord=self.order, axis=(1,2)), 2)
            elif (self.mode == 1):
                D2_x =
np.power(np.linalg.norm(np.absolute(np.nan_to_num(X[:,i,:].reshape(self.n_chroms,1,s
elf.n_samples)-cent)), ord=self.order, axis=(0,2)), 2)
            else:
                D2_x =
np.power(np.linalg.norm(np.absolute(np.nan_to_num(X[:,:,i].reshape(self.n_chroms,
```

```
self.n_genes, 1)-cent)), ord=self.order, axis=(0,1)), 2)
            D2 = np.hstack((D2, np.nanmin(D2_x)))

        self.D2 = np.nan_to_num(D2)

    def _choose_next_center(self):
        """
        Compute and update the values of the cluster centers.

        Returns
        -------
        ndarray of float
            An array containing the new cluster centers
        """
        probs = self.D2/self.D2.sum()
        cumprobs = probs.cumsum()
        while True:
            r = random.random()
            ind = np.where(cumprobs >= r)[0][0]
            if (probs[ind] != 0):
                break

        if (self.mode == 0):
            return self.X[ind,:,:].reshape((1, self.n_genes, self.n_samples))
        elif (self.mode == 1):
            return self.X[:,ind,:].reshape((self.n_chroms, 1, self.n_samples))
        else:
            return self.X[:,:,ind].reshape((self.n_chroms, self.n_genes, 1))

    def _init_centers(self):
        """
        Initialize the cluster centers according to given method.

        Notes
        -----
        Updates the mu attribute.
        Caution: Mu might contain NaNs!
        """
        if (self.method == 'k-means++'):   # k-means++ initialization
            self.mu = self.X.take(np.random.randint(self.X.shape[self.mode]),
                                  axis=self.mode)

            if (self.mode == 0):
                self.mu = self.mu.reshape((1, self.mu.shape[0], self.mu.shape[1]))
            elif (self.mode == 1):
                self.mu = self.mu.reshape((self.mu.shape[0], 1, self.mu.shape[1]))
            else:
                self.mu = self.mu.reshape((self.mu.shape[0], self.mu.shape[1], 1))
            i = 1
            while (i < self.K):
                self._dist_from_centers()
                if (self.mode == 0):
                    self.mu = np.vstack((self.mu, self._choose_next_center()))
                elif (self.mode == 1):
                    self.mu = np.hstack((self.mu, self._choose_next_center()))
                else:
                    self.mu = np.dstack((self.mu, self._choose_next_center()))
                i += 1
        else:  # random initialization
            self.mu = self.X.take(np.random.randint(self.X.shape[self.mode],
                                                    size=self.K),
                                  axis=self.mode)

    def _memoize(func):
        """
        Memoization wrapper function, for  optimization purposes.
        """
        memo = {}
```

```
    def helper(*args):
        if args not in memo:
            memo[args] = func(*args)
        return memo[args]

    return helper

@_memoize
def _ak(self, k, Nd):
    """
    Compute the weight factor for f(k).

    Parameters
    ----------
    k : int
        The current number of clusters
    Nd : int
        The dimensions of the data

    Returns
    -------
    a_k : float
        The weight factor
    """
    if (k == 2):
        a_k = 1.0 - (3.0 / (4.0 * Nd))
    else:
        a_k = self._ak(k-1, Nd) + (1.0 - self._ak(k-1, Nd)) / 6.0
    return a_k

def fk(self, maxk, method='random', order=2, mode=1, n_times=10,
       max_iter=300, tol=1e-4, init_fs=None, verbose=False):
    """
    Compute the evaluation function f(k) for different values of k, in
    order to determine the most suitable number of clusters.

    Parameters
    ----------
    maxk : int
        The maximum value of `k` to be considered
    method : {'random', 'k-means++'}, default 'random'
        The method to be used for cluster centers initialization
    order : {non-zero int, inf, -inf, ‘fro’, ‘nuc’}, default 2
        Order of the norm to be used as distance metric.
        See numpy.linalg.norm documentation for more details.
    mode : {0, 1, 2}, default 1
        Determines the axis along which to do the clustering.
        mode=i means that the clustering will be done along axis i.
    n_times : int, default 10
        Number of times for the k-means algorithm to be run
    max_iter : int, default 300
        The maximum number of iterations for the algorithm
    tol : float, default 1e-4
        The tolerance. If the difference between the previous cluster
        centers and the new ones is less than the tolerance, then the
        algorithm converges
    init_fs : ndarray of float
        The user can give a list of f(k) values for k, 1 <= k <= i, and
        the function will proceed to find the next f(k) values for
        i < k <= maxk
    verbose : bool, default False
        Verbose mode for debugging

    Notes
    -----
    Updates the metric, method, K and fs attributes.
    """
```

```
        fs = np.zeros(maxk)

        X = self.X
        if (mode == 0) or (mode == 1):
            Nd = self.n_genes
        else:
            Nd = self.n_samples

        if (init_fs is None):
            start_k = 1
            Skm1 = 0
        else:
            start_k = len(init_fs)
            fs[:start_k] = init_fs

            # Compute Skm1
            self.K = start_k-1
            self.find_centers(K=start_k-1, order=order, max_iter=max_iter,
                              mode=mode, n_times=n_times, tol=tol,
                              method=method)
            centroids = self.get_centroids()
            clusters = self.best_clusters

            Skm1 = 0
            for i in sorted(clusters.keys()):
                i_data = X.take(clusters[i], axis=self.mode)

                if (self.mode == 0) or (self.mode == 1):
                    # Convert i_data to 2-D array by flattening the chromosomes
                    # dimension
                    i_data = np.nan_to_num(np.reshape(i_data,
                                                      (i_data.shape[0]*i_data.shape[1],
i_data.shape[2])))

                    # Remove the NaN rows
                    i_data = i_data.take(np.unique(np.nonzero(i_data)[0]), axis=0)
                else:
                    # Take the mean of each sample for all chromosomes
                    i_data = np.nan_to_num(np.nanmean(i_data, axis=0))

                    # Swap axes 1 and 2
                    i_data = np.swapaxes(i_data, 0, 1)

                # Compute Skm1
                Skm1 = Skm1 +
np.sum(np.square(distance.cdist(np.nan_to_num(centroids[i, np.newaxis]),
                                                                i_data)))

        # Compute fs and Sk for the next k values
        for k in range(start_k, maxk+1):
            if (verbose):
                print("k = {}".format(k))

            self.K = k
            self.find_centers(K=k, order=order, max_iter=max_iter, mode=mode,
                              n_times=n_times, tol=tol, method=method)
            centroids = self.get_centroids()
            clusters = self.best_clusters

            Sk = 0
            for i in sorted(clusters.keys()):

                i_data = X.take(clusters[i], axis=self.mode)

                if (self.mode == 0) or (self.mode == 1):
                    # Convert i_data to 2-D array by flattening the chromosomes
                    # dimension
                    i_data = np.nan_to_num(np.reshape(i_data,
```

```
                                                (i_data.shape[0]*i_data.shape[1],
i_data.shape[2])))
                    # Remove the NaN rows
                    i_data = i_data.take(np.unique(np.nonzero(i_data)[0]), axis=0)
                else:
                    # Take the mean of each sample for all chromosomes
                    i_data = np.nan_to_num(np.nanmean(i_data, axis=0))

                    # Swap axes 1 and 2
                    i_data = np.swapaxes(i_data, 0, 1)

                # Compute Sk
                Sk = Sk + np.sum(np.square(distance.cdist(np.nan_to_num(centroids[i,
                                          np.newaxis]), i_data)))

            if (k == 1):
                fs[0] = 1
            elif (Skm1 == 0):
                fs[k-1] = 1
            else:
                fs[k-1] = Sk / (self._ak(k, Nd)*Skm1)

            Skm1 = Sk

        self.fs = fs
```

Κλασικός k-Means

```
# -*- coding: utf-8 -*-

import numpy as np
import random
from scipy.spatial import distance
from scipy import floor


class KMeans():
    """
    The k-means clustering algorithm.

    Attributes
    ----------
    X : ndarray
        The data to be clustered
    K : int
        The number of clusters to be determined
    N : int
        The dimension of the data
    mu : ndarray
        The cluster centers
    oldmu : ndarray
        The previous values of cluster centers
    clusters : dict
        Cluster id mapped to a list of the cluster's elements
    method : {'random', 'k-means++'}
        The method of cluster centers initialization
    metric : str or callable, optional
        The distance metric to use.  If a string, the distance function can be
        'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation',
        'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski',
        'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao',
        'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
        'wminkowski', 'yule', 'fractional'.
```

```
tol : float
    The algorithm's tolerance
best_inertia : float
    The algorithm's inertia
best_clusters : dict
    Same as clusters, but keeps the best clustering we've found so far
best_labels : ndarray
    Same as labels but keeps the best clustering we've found so far
best_mu : ndarray
    Same as mu but keeps the best clustering we've found so far
distances : ndarray
    A 2xm array containg for each data point its closest cluster center
    and the distance from it
f : float, optional
    The order of the fractional norm
fs : list of float
    The f(k) values for each k
D2 : ndarray
    Vector of the D^2 weighting for the k-means++ algorithm.
    It is a (1 x n_genes) vector containing the shortest (squared )distance
    of each gene from a centroid.

Methods
-------
find_centers(method='random', metric='euclidean', f=None,
             n_times=10, tol=1e-4, max_iter=300, K=1, verbose=False)
    Run the k-means algorithm with the given parameters.
get_clusters()
    Return a dictionary cluster_id->list_of_cluster_elements
get_labels()
    Return an ndarray containing the cluster id for each element
get_best_k()
    Return a list of the most appropriate values of k
get_centers()
    Return an ndarray containing the cluster centers
get_centroids()
    Return an ndarray containing the cluster centroids
fk()
    Run the f(k) method in order to determine suitable values for k

References
----------
.. [1] Lloyd, Stuart P. "Least Squares Quantization in PCM."
       IEEE Transactions on Information Theory. Vol. 28, 1982, pp. 129–137
.. [2] Arthur, David, and Sergi Vassilvitskii. "K-means++: The Advantages
       of Careful Seeding." SODA ‘07: Proceedings of the Eighteenth Annual
       ACM-SIAM Symposium on Discrete Algorithms. 2007, pp. 1027–1035
.. [3] Pham D. T., Dimov S. S., and Nguyen C. D. "Selection of K in
       k-means clustering." DOI: 10.1243/095440605X8298, 2004
.. [4] Davies D., Bouldin D. "A cluster separation measure", IEEE
       Transactions on Pattern Analysis and Machine Intelligence, Vol. 1,
       pp. 224-227, doi: 10.1109/TPAMI.1979.4766909
"""

def __init__(self, X):
    """
    Parameters
    ----------
    X : ndarray
        The data to be clustered

    Notes
    -----
    Updates the X, N, mu, clusters and method attributes.
    """
    if (X is None):
        raise Exception("Error: No data provided!")
    else:
```

```
            self.X = X   # ndarray
            self.N = len(X)  # Number of genes

        self.mu = None  # ndarray
        self.clusters = None
        self.method = None

    def _cluster_points(self):
        """
        Compute the distance of each data point from all cluster centers
        and find the nearest.

        Notes
        -----
        Updates the clusters, labels and distances attributes.
        """
        mu = self.mu
        clusters = {}
        X = self.X

        if (self.metric == 'fractional'):
            distances = np.asarray(distance.cdist(X,
                                                  mu,
                                                  lambda x1, x2: np.linalg.norm(x1-
                                                                                x2,
                                                                       ord=self.f)))
        else:
            distances = np.asarray(distance.cdist(X, mu, self.metric))

        distances = np.asarray((np.amin(distances, axis=1),
                                np.argmin(distances, axis=1)))

        for clust in range(self.K):
            clusters[clust] = np.where(distances[1] == clust)[0].tolist()

        self.clusters = clusters
        self.labels = np.asarray(distances[1], dtype=int)
        self.distances = distances

    def _reevaluate_centers(self):
        """
        Re-evaluates the cluster centers for the Update phase of the algorithm.

        Notes
        -----
        Updates the mu attribute.
        """
        clusters = self.clusters
        newmu = []
        keys = sorted(self.clusters.keys())
        for k in keys:
            newmu.append(np.mean(np.take(self.X, clusters[k], axis=0), axis=0))
        self.mu = np.asarray(newmu)

    def _has_converged(self):
        """
        Returns True if the algorithm has converged, i.e. the difference
        between new cluster centers and previous ones is less than the
        given tolerance.

        Returns
        -------
        bool
            True if the algorithm has converged, False otherwise.
        """
        mu = self.mu
        oldmu = self.oldmu
        d = np.ravel(oldmu - mu)
```

```
    return np.dot(d, d) <= self.tol

def find_centers(self, method='random', metric='euclidean', f=0.5,
                 n_times=10, tol=1e-4, max_iter=300, K=1, verbose=False,
                 init_centers=None):
    """
    Runs the k-means algorithm with the given parameters.

    Parameters
    ----------
    method : {'random', 'k-means++'}, default 'random'
        The method to be used for cluster centers initialization
    metric : str or callable, optional
        The distance metric to use.  If a string, the distance function can
        be 'braycurtis', 'canberra', 'chebyshev', 'cityblock',
        'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard',
        'kulsinski', 'mahalanobis', 'matching', 'minkowski',
        'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener',
        'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule', 'fractional'.
    f : float, default 0.5
        The order of the norm in case of 'fractional' distance metric
    n_times : int, default 10
        Number of times for the k-means algorithm to be run
    tol : float, default 1e-4
        The tolerance. If the difference between the previous cluster
        centers and the new ones is less than the tolerance, then the
        algorithm converges
    max_iter : int, default 300
        The maximum number of iterations for the algorithm
    K : int, default 1
        The number of clusters
    verbose : bool, default False
        Verbose mode for debugging
    init_centers : ndarray
        The user can give an initialization of the cluster centers to be
        used by the algorithm. Ndarray should be of the form (n_clusters,
        n_features)

    Notes
    -----
    Updates the K, method, metric, tol, best_inertia, best_clusters,
    best_labels, best_mu, distances, labels, clusters and oldmu attributes.
    """
    self.K = K
    self.method = method
    self.metric = metric
    X = self.X.tolist()
    self.tol = tol

    # Check the parameters given
    if (len(self.X) == 0):
        raise ValueError("Please provide valid data to the algorithm.")
    if (floor(self.K) != self.K) or (self.K <= 0):
        raise ValueError("'k' must be an integer > 0, but its value"
                         " is {}".format(self.K))
    if (self.method not in ['random', 'k-means++']):
        raise ValueError("'method' must be either 'random' or 'k-means++',"
                         " but its value is {}".format(self.delta))
    if (floor(max_iter) != max_iter) or (max_iter < 0):
        raise ValueError("'max_iter' must be an integer > 0, but its value"
                         " is {}".format(self.K))

    if (metric == 'fractional'):
        self.f = f

    self.best_inertia = None
    self.best_clusters = None
    self.best_labels = None
```

```
        self.best_mu = None

        self.distances = np.asarray([[None], [None]])
        self.labels = None
        self.clusters = None

        # Run k-means algorithm n_times
        for i in range(n_times):
            if (verbose):
                print(i)

            self.oldmu = np.asarray(random.sample(X, K))

            # Initialize centers
            if (init_centers is None):
                self._init_centers()
            else:
                self.mu = init_centers

            # E-M steps
            for j in range(max_iter):
                if (not self._has_converged()):
                    self.oldmu = self.mu
                    # Assign all points in X to clusters
                    self._cluster_points()
                    # Reevaluate centers
                    self._reevaluate_centers()
                else:
                    if (verbose):
                        print("   Converged in iteration {}".format(j))
                    break

            # Calculate inertia (Sum of distances of samples to their closest
            # cluster center)
            inertia = np.sum(self.distances[0])
            if (self.best_inertia is None) or (inertia < self.best_inertia):
                self.best_inertia = inertia
                self.best_labels = self.labels
                self.best_clusters = self.clusters
                self.best_mu = self.mu

    def get_clusters(self):
        """
        Return the clusters formed.

        Returns
        -------
            best_clusters : dict
                Dictionary of the form: cluster_id->list_of_points_of_cluster
        """
        return self.best_clusters

    def get_labels(self):
        """
        Return the labels found.

        Returns
        -------
        best_labels : ndarray
            The id of the cluster that each data point belongs to
        """
        return np.array(self.best_labels)

    def get_best_k(self):
        """
        Return the most suitable values for k.

        Returns
```

```
        -------
        best_k : list of int
            The values of k in priority order (those with the smallest f(k) are
            listed first)
        """
        return [i+1 for i in np.argsort(self.fs) if self.fs[i] < 0.85]

    def get_centers(self):
        """
        Return the values of the cluster centers.

        Returns
        -------
        best_mu : ndarray
            The values of the cluster centers found
        """
        return self.best_mu

    def get_centroids(self):
        """
        Return the cluster centroids.

        Returns
        -------
        centroids : ndarray
            The cluster centroids
        """
        return np.vstack(self.X.take(self.best_clusters[x], axis=0).mean(axis=0)
                         for x in self.best_clusters.keys())

    def _dist_from_centers(self):
        """
        Calculate the distance of each data point from each cluster center.

        Notes
        -----
        Updates the D2 attribute.
        """
        cent = self.mu
        X = self.X

        if (self.metric == 'fractional'):
            D2 = np.amin(np.power(distance.cdist(X,
                                                 cent,
                                                 lambda x1, x2: np.linalg.norm(x1-
                                                                               x2,
                                                                               ord=self.f)), 2),
                         axis=1).flatten(order='A')
        else:
            D2 = np.amin(np.power(distance.cdist(X, cent, self.metric), 2),
                         axis=1).flatten(order='A')

        self.D2 = D2

    def _choose_next_center(self):
        """
        Compute and update the values of the cluster centers.

        Returns
        -------
        list of float
            A list containing the new cluster centers
        """
        probs = self.D2/self.D2.sum()
        cumprobs = probs.cumsum()
        r = random.random()
        ind = np.where(cumprobs >= r)[0][0]
        return self.X[ind].tolist()
```

```
def _init_centers(self):
    """
    Initialize the cluster centers according to given method.

    Notes
    -----
    Updates the mu attribute.
    """
    if (self.method == 'k-means++'):   # k-means++ initialization
        self.mu = np.asarray(random.sample(self.X.tolist(), 1))
        while (len(self.mu) < self.K):
            self._dist_from_centers()
            self.mu = np.vstack((self.mu, self._choose_next_center()))
    else:  # random initialization
        self.mu = np.asarray(random.sample(self.X.tolist(), self.K))

def _memoize(func):
    """
    Memoization wrapper function, for  optimization purposes.
    """
    memo = {}

    def helper(*args):
        if args not in memo:
            memo[args] = func(*args)
        return memo[args]

    return helper

@_memoize
def _ak(self, k, Nd):
    """
    Compute the weight factor for f(k).

    Parameters
    ----------
    k : int
        The current number of clusters
    Nd : int
        The dimensions of the data

    Returns
    -------
    a_k : float
        The weight factor
    """
    if (k == 2):
        a_k = 1.0 - (3.0 / (4.0 * Nd))
    else:
        a_k = self._ak(k-1, Nd) + (1.0 - self._ak(k-1, Nd)) / 6.0
    return a_k

def fk(self, maxk, metric='euclidean', method='random', f=0.5, n_times=10,
       max_iter=300, tol=1e-4, verbose=False):
    """
    Compute the evaluation function f(k) for different values of k, in
    order to determine the most suitable number of clusters.

    Parameters
    ----------
    maxk : int
        The maximum value of `k` to be considered
    metric : str or callable, optional
        The distance metric to use.  If a string, the distance function can
        be 'braycurtis', 'canberra', 'chebyshev', 'cityblock',
        'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard',
        'kulsinski', 'mahalanobis', 'matching', 'minkowski',
```

```
        'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener',
        'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule', 'fractional'.
    method : {'random', 'k-means++'}, default 'random'
        The method to be used for cluster centers initialization
    f : float, default 0.5
        The order of the norm in case of 'fractional' distance metric
    n_times : int, default 10
        Number of times for the k-means algorithm to be run
    max_iter : int, default 300
        The maximum number of iterations for the algorithm
    tol : float, default 1e-4
        The tolerance. If the difference between the previous cluster
        centers and the new ones is less than the tolerance, then the
        algorithm converges
    verbose : bool, default False
        Verbose mode for debugging

    Notes
    -----
    Updates the metric, method, K and fs attributes.
    """

    X = self.X
    Nd = self.N

    fs = np.zeros(maxk)

    self.K = 1
    Skm1 = 0

    for k in range(1, maxk+1):
        if (verbose):
            print("k = {}".format(k))

        self.K = k

        self.find_centers(K=k, max_iter=max_iter, n_times=n_times, tol=tol,
                          method=method)
        best_clusters = self.best_clusters
        best_mu = self.best_mu

        Sk = 0
        for clust in best_clusters.keys():
            if (self.metric == 'fractional'):
                Sk = Sk + \
np.sum(np.square(distance.cdist(X.take(best_clusters[clust], axis=0),
                                                best_mu[clust, np.newaxis],
                                                lambda x1, x2:
                                        np.linalg.norm(x1-x2, ord=f))))
            else:
                Sk = Sk + \
np.sum(np.square(distance.cdist(X.take(best_clusters[clust], axis=0),
                                                best_mu[clust, np.newaxis],
                                                metric)))
        if (k == 1):
            fs[0] = 1
        elif (Skm1 == 0):
            fs[k-1] = 1
        else:
            fs[k-1] = Sk / (self._ak(k, Nd)*Skm1)

        Skm1 = Sk

    self.fs = fs
```

δ-TRIMAX

```
# -*- coding: utf-8 -*-

import numpy as np

import warnings


class EmptyTriclusterException(Exception):
    pass


class DeltaTrimax():
    """
    The delta-TRIMAX clustering algorithm.

    Attributes
    ----------
    D : ndarray
        The data to be clustered
    delta : float
        The delta parameter of the algorithm. Must be > 0.0
    l : float
        The lambda parameter of the algorithm. Must be >= 1.0
    chrom_cutoff : int
        The deletion threshold for the chromosome axis
    gene_cutoff : int
        The deletion threshold for the gene axis
    sample_cutoff : int
        The deletion threshold for the sample axis
    tol : float
        The algorithm's tolerance
    mask_mode : {'random', 'nan'}
        The masking method for the clustered values. If 'random', the values
        are replaced by random floats. If 'nan', they are replaced by nan
        values.
    n_chroms : int
        The number of chromosome pairs
    n_genes : int
        The number of genes
    n_samples : int
        The number of samples
    reuslt_chroms : list of ndarray
        A list of length #triclusters, containg a boolean ndarray for each
        tricluster. The boolean array is of length #chromosomes and contains
        True if the respective chromosome is contained in the tricluster,
        False otherwise.
    result_genes : list of ndarray
        A list of length #triclusters, containg a boolean ndarray for each
        tricluster. The boolean array is of length #genes and contains
        True if the respective gene is contained in the tricluster,
        False otherwise.
    result_samples : list of ndarray
        A list of length #triclusters, containg a boolean ndarray for each
        tricluster. The boolean array is of length #samples and contains
        True if the respective sample is contained in the tricluster,
        False otherwise.
    MSR : float
        The Mean Squared Residue of each cell.
    MSR_chrom : float
        The Mean Squared Residue of each chromosome.
    MSR_gene : float
        The Mean Squared Residue of each gene.
    MSR_sample : float
```

```
    The Mean Squared Residue of each sample.

Methods
-------
fit(self, delta=2.5, l=1.005, chrom_cutoff=50, gene_cutoff=50,
        sample_cutoff=50, tol=1e-5, mask_mode='nan', verbose=False)
    Run the delta-TRIMAX algorithm for the given parameters.
get_triclusters()
    Return the triclusters found by the algorithm.

References
----------
.. [1] A. Bhar, M. Haubrock, A. Mukhopadhyay, U. Maulik, S. Bandyopadhyay,
       and E. Wingender, 'Coexpression and coregulation analysis of
       time-series gene expression data in estrogen-induced breast cancer
       cell', Algorithms Mol. Biol., τ. 8, τχ. 1, σ 9, 2013.
"""

def __init__(self, D):
    """
    Parameters
    ----------
    D : ndarray
        The data to be clustered
    """
    self.D = D.copy()

def _check_parameters(self):
    """
    Checks the parameters given by the user. If the values are not valid,
    a ValueError is raised.
    """
    if (self.delta < 0):
        raise ValueError("'delta' must be > 0.0, but its value"
                         " is {}".format(self.delta))
    if (self.l < 1):
        raise ValueError("'lambda' must be >= 1.0, but its"
                         " value is {}".format(self.l))
    if (self.gene_cutoff < 1):
        raise ValueError("'gene deletion cutoff' must be > 1.0, but its"
                         " value is {}".format(self.gene_cutoff))
    if (self.sample_cutoff < 1):
        raise ValueError("'sample deletion cutoff' must be > 1.0, but its"
                         " value is {}".format(self.sample_cutoff))
    if (self.chrom_cutoff < 1):
        raise ValueError("'chromosomes deletion cutoff' must be > 1.0, but"
                         " its value is {}".format(self.chrom_cutoff))
    if (self.mask_mode not in ['nan', 'random']):
        raise ValueError("'mask mode' must be either 'nan' or 'random',"
                         " but its value is {}".format(self.mask_mode))

def _compute_MSR(self, chroms, genes, samples):
    """
    Computes the Mean Squared Residue (MSR) for the algorithm.

    Parameters
    ----------
    chroms : ndarray
        Contains 1 for a chromosome pair that belongs to the tricluster
        currently examined, 0 otherwise.
    genes : ndarray
        Contains 1 for a gene that belongs to the tricluster currently
        examined, 0 otherwise.
    samples : ndarray
        Contains 1 for a sample that belongs to the tricluster currently
        examined, 0 otherwise.

    Note
```

```
        ----
        Updates the n_chorms, n_genes, n_samples, MSR, MSR_chrom, MSR_gene and
        MSR_sample attributes.
        """
        chrom_idx = np.expand_dims(np.expand_dims(np.nonzero(chroms)[0], axis=1),
                                                  axis=1)
        gene_idx = np.expand_dims(np.expand_dims(np.nonzero(genes)[0], axis=0),
                                                  axis=2)
        sample_idx = np.expand_dims(np.expand_dims(np.nonzero(samples)[0], axis=0),
                                                  axis=0)

        if (not chrom_idx.size) or (not gene_idx.size) or (not sample_idx.size):
            raise EmptyTriclusterException()

        subarr = self.D[chrom_idx, gene_idx, sample_idx]
        self.n_chroms = subarr.shape[0]
        self.n_genes = subarr.shape[1]
        self.n_samples = subarr.shape[2]

        with warnings.catch_warnings():  # We expect mean of NaNs here
            warnings.simplefilter("ignore", category=RuntimeWarning)
            # Computation of m_iJK
            m_iJK = np.nanmean(np.nanmean(subarr, axis=2), axis=1)
            m_iJK = np.expand_dims(np.expand_dims(m_iJK, axis=1), axis=1)

            # Computation of m_IjK
            m_IjK = np.nanmean(np.nanmean(subarr, axis=2), axis=0)
            m_IjK = np.expand_dims(np.expand_dims(m_IjK, axis=0), axis=2)

            # Computation of m_IJk
            m_IJk = np.nansum(np.nansum(subarr, axis=0, keepdims=1), axis=1,
                                        keepdims=1)
            m_IJk = m_IJk / ((subarr.shape[0] * subarr.shape[1]) -
                                    np.count_nonzero(np.isnan(subarr[:,:,0])))

            # Computation of m_IJK
            m_IJK = np.nanmean(subarr)

            # Computation of MSR
            residue = subarr - m_iJK - m_IjK - m_IJk + (2*m_IJK)
            SR = np.square(residue)

            self.MSR = np.nanmean(SR)
            self.MSR_chrom = np.nanmean(np.nanmean(SR, axis=2), axis=1)
            self.MSR_gene = np.nanmean(np.nanmean(SR, axis=2), axis=0)
            self.MSR_sample = np.nanmean(np.nanmean(SR, axis=0), axis=0)

            # Check tolerance
            self.MSR_chrom[self.MSR_chrom < self.tol] = 0
            self.MSR_gene[self.MSR_gene < self.tol] = 0
            self.MSR_sample[self.MSR_sample < self.tol] = 0
            self.MSR = 0 if (self.MSR < self.tol or np.isnan(self.MSR)) else
                                    self.MSR

    def _single_node_deletion(self, chroms, genes, samples):
        """
        The single node deletion routine of the algorithm.

        Parameters
        ----------
        chroms : ndarray
            Contains 1 for a chromosome pair that belongs to the tricluster
            currently examined, 0 otherwise.
        genes : ndarray
            Contains 1 for a gene that belongs to the tricluster currently
            examined, 0 otherwise.
        samples : ndarray
            Contains 1 for a sample that belongs to the tricluster currently
```

```
            examined, 0 otherwise.

        Returns
        -------
        chroms : ndarray
            Contains 1 for a chromosome pair that belongs to the tricluster
            examined, 0 otherwise.
        genes : ndarray
            Contains 1 for a gene that belongs to the tricluster examined,
            0 otherwise.
        samples : ndarray
            Contains 1 for a sample that belongs to the tricluster examined,
            0 otherwise.
        """
        self._compute_MSR(chroms, genes, samples)

        while (self.MSR > self.delta):
            chrom_idx = np.nanargmax(self.MSR_chrom)
            gene_idx = np.nanargmax(self.MSR_gene)
            sample_idx = np.nanargmax(self.MSR_sample)

            with warnings.catch_warnings():  # We expect mean of NaNs here
                warnings.simplefilter("ignore", category=RuntimeWarning)
                if (self.MSR_chrom[chrom_idx] > self.MSR_gene[gene_idx]):
                    if (self.MSR_chrom[chrom_idx] > self.MSR_sample[sample_idx]):
                        # Delete chrom
                        nonz_idx = chroms.nonzero()[0]
                        chroms.put(nonz_idx[chrom_idx], 0)
                    else:
                        # Delete sample
                        nonz_idx = samples.nonzero()[0]
                        samples.put(nonz_idx[sample_idx], 0)
                else:
                    if (self.MSR_gene[gene_idx] > self.MSR_sample[sample_idx]):
                        # Delete gene
                        nonz_idx = genes.nonzero()[0]
                        genes.put(nonz_idx[gene_idx], 0)
                    else:
                        # Delete sample
                        nonz_idx = samples.nonzero()[0]
                        samples.put(nonz_idx[sample_idx], 0)

            self._compute_MSR(chroms, genes, samples)

        return chroms, genes, samples

    def _multiple_node_deletion(self, chroms, genes, samples):
        """
        The multiple node deletion routine of the algorithm.

        Parameters
        ----------
        chroms : ndarray
            Contains 1 for a chromosome pair that belongs to the tricluster
            currently examined, 0 otherwise.
        genes : ndarray
            Contains 1 for a gene that belongs to the tricluster currently
            examined, 0 otherwise.
        samples : ndarray
            Contains 1 for a sample that belongs to the tricluster currently
            examined, 0 otherwise.

        Returns
        -------
        chroms : ndarray
            Contains 1 for a chromosome pair that belongs to the tricluster
            examined, 0 otherwise.
        genes : ndarray
```

```
        Contains 1 for a gene that belongs to the tricluster examined,
        0 otherwise.
    samples : ndarray
        Contains 1 for a sample that belongs to the tricluster examined,
        0 otherwise.
    """
    self._compute_MSR(chroms, genes, samples)

    while (self.MSR > self.delta):
        deleted = 0

        with warnings.catch_warnings():  # We expect mean of NaNs here
            warnings.simplefilter("ignore", category=RuntimeWarning)
            if (self.n_chroms > self.chrom_cutoff):
                chroms_to_del = self.MSR_chrom > (self.l * self.MSR)
                nonz_idx = chroms.nonzero()[0]
                if (chroms_to_del.any()):
                    deleted = 1
                chroms.put(nonz_idx[chroms_to_del], 0)

            if (self.n_genes > self.gene_cutoff):
                genes_to_del = self.MSR_gene > (self.l * self.MSR)
                nonz_idx = genes.nonzero()[0]
                if (genes_to_del.any()):
                    deleted = 1
                genes.put(nonz_idx[genes_to_del], 0)

            if (self.n_samples > self.sample_cutoff):
                samples_to_del = self.MSR_sample > (self.l * self.MSR)
                nonz_idx = samples.nonzero()[0]
                if (samples_to_del.any()):
                    deleted = 1
                samples.put(nonz_idx[samples_to_del], 0)

        if (not deleted):
            break

        self._compute_MSR(chroms, genes, samples)

    return chroms, genes, samples

def _node_addition(self, chroms, genes, samples):
    """
    The single node addition routine of the algorithm.

    Parameters
    ----------
    chroms : ndarray
        Contains 1 for a chromosome pair that belongs to the tricluster
        currently examined, 0 otherwise.
    genes : ndarray
        Contains 1 for a gene that belongs to the tricluster currently
        examined, 0 otherwise.
    samples : ndarray
        Contains 1 for a sample that belongs to the tricluster currently
        examined, 0 otherwise.

    Returns
    -------
    chroms : ndarray
        Contains 1 for a chromosome pair that belongs to the tricluster
        examined, 0 otherwise.
    genes : ndarray
        Contains 1 for a gene that belongs to the tricluster examined,
        0 otherwise.
    samples : ndarray
        Contains 1 for a sample that belongs to the tricluster examined,
        0 otherwise.
```

```
    """

    while True:
        self._compute_MSR(chroms, genes, samples)
        n_chroms = np.count_nonzero(chroms)
        n_genes = np.count_nonzero(genes)
        n_samples = np.count_nonzero(samples)

        with warnings.catch_warnings():  # We expect mean of NaNs here
            warnings.simplefilter("ignore", category=RuntimeWarning)
            elems_to_add = self.MSR_chrom <= self.MSR
            nonz_idx = chroms.nonzero()[0]
            chroms.put(nonz_idx[elems_to_add], 1)

            elems_to_add = self.MSR_gene <= self.MSR
            nonz_idx = genes.nonzero()[0]
            genes.put(nonz_idx[elems_to_add], 1)

            elems_to_add = self.MSR_sample <= self.MSR
            nonz_idx = samples.nonzero()[0]
            samples.put(nonz_idx[elems_to_add], 1)

        if (n_chroms == np.count_nonzero(chroms)) and \
           (n_genes == np.count_nonzero(genes)) and \
           (n_samples == np.count_nonzero(samples)):
            break

    return chroms, genes, samples

def _mask(self, chroms, genes, samples, minval, maxval):
    """
    Masks the values of the array that have been used in triclusters
    with either random float numbers, or nan.

    Parameters
    ----------
    chroms : ndarray
        Contains 1 for a chromosome pair that belongs to the tricluster
        currently examined, 0 otherwise.
    genes : ndarray
        Contains 1 for a gene that belongs to the tricluster currently
        examined, 0 otherwise.
    samples : ndarray
        Contains 1 for a sample that belongs to the tricluster currently
        examined, 0 otherwise.
    minval : float
        Lower boundary of the output interval for the random generator.
    maxval : float
        Upper boundary of the output interval for the random generator.
    """
    c = np.expand_dims(np.expand_dims(chroms.nonzero()[0], axis=1), axis=1)
    g = np.expand_dims(np.expand_dims(genes.nonzero()[0], axis=0), axis=2)
    s = np.expand_dims(np.expand_dims(samples.nonzero()[0], axis=0), axis=0)
    if (self.mask_mode == 'random'):
        shape = np.count_nonzero(chroms), np.count_nonzero(genes),
                      np.count_nonzero(samples)
        mask_vals = np.random.uniform(minval, maxval, shape)
        self.D[c, g, s] = mask_vals
    else:
        self.D[c, g, s] = np.nan

def fit(self, delta=2.5, l=1.005, chrom_cutoff=50, gene_cutoff=50,
        sample_cutoff=50, tol=1e-5, mask_mode='nan', verbose=False):
    """
    Runs the delta-TRIMAX algorithm with the given parameters.

    Parameters
    ----------
```

```
delta : float, default 2.5
    The delta parameter of the algorithm. Must be > 0.0
l : float, default 1.005
    The lambda parameter of the algorithm. Must be >= 1.0
chrom_cutoff : int, default 50
    The deletion threshold for the chromosome axis
gene_cutoff : int, default 50
    The deletion threshold for the gene axis
sample_cutoff : int, default 50
    The deletion threshold for the sample axis
tol : float, default 1e-5
    The algorithm's tolerance
mask_mode : {'random', 'nan'}, default 'nan'
    The masking method for the clustered values. If 'random', the values
    are replaced by random floats. If 'nan', they are replaced by nan
    values.
verbose : bool, default False
    Verbose mode for debugging.
"""
self.delta = delta
self.l = l
self.chrom_cutoff = chrom_cutoff
self.gene_cutoff = gene_cutoff
self.sample_cutoff = sample_cutoff
self.tol = tol
self.mask_mode = mask_mode
self._check_parameters()

n_chroms, n_genes, n_samples = self.D.shape
minval, maxval = np.nanmin(self.D), np.nanmax(self.D)

result_chroms = []
result_genes = []
result_samples = []

i = 1
while True:
    if (verbose):
        print(i)
    chroms = np.ones(n_chroms, dtype=np.bool)
    genes = np.ones(n_genes, dtype=np.bool)
    samples = np.ones(n_samples, dtype=np.bool)

    # Multiple node deletion
    chroms, genes, samples = self._multiple_node_deletion(chroms,
                                                          genes,
                                                          samples)

    # Single node deletion
    chroms, genes, samples = self._single_node_deletion(chroms,
                                                        genes,
                                                        samples)

    # Node addition
    chroms, genes, samples = self._node_addition(chroms,
                                                 genes,
                                                 samples)

    # Check for trivial tricluster
    if (chroms.sum() == 1) or (genes.sum() == 1) or (samples.sum() == 1):
        break  # trivial bicluster
    # Check if the aren't any unused values in D
    if ((mask_mode == 'nan') and (np.isnan(self.D).all())):
        break

    # Mask values
    self._mask(chroms, genes, samples, minval, maxval)
```

```
            result_chroms.append(chroms)
            result_genes.append(genes)
            result_samples.append(samples)

            if (verbose):
                print("--- MSR = " + str(self.MSR))

            i += 1

        self.result_chroms = result_chroms
        self.result_genes = result_genes
        self.result_samples = result_samples

    def get_triclusters(self):
        """
        Returns the triclusters found by the algorithm.
        """
        return self.result_chroms, self.result_genes, self.result_samples
```

Μέθοδοι αξιολόγησης συσταδοποιήσεων

```
# -*- coding: utf-8 -*-

import numpy as np
from scipy.spatial import distance


def DB_index2D(centroids, clusters, X, metric='euclidean'):
    """
    Return the Davies-Bouldin index for cluster evaluation for 2D data.

    The index is calculated using the formula:
    DB = (1/k) * sum(max((sigma_i + sigma_j)/distance(c_i, c_j)))
    where:
        k is the number of clusters
        sigma_i is the average distance of all elements in cluster i to
            centroid c_i
        c_i is the centroid of cluster i
        distance(c_i, c_j) is the distance between cluster centroids i and
            j, using the same metric as in the algorithm

    The smaller the value of the DB index, the better the clustering.

    Parameters
    ----------
    centroids : ndarray of float
        The centroids of the clusters
    clusters : dict
        Cluster id mapped to a list of the cluster's elements
    X : ndarray (2D)
        The data array
    metric : str or callable, optional
        The distance metric to use.  If a string, the distance function can be
        'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation',
        'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski',
        'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao',
        'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
        'wminkowski', 'yule'.

    Returns
    -------
    DB_ind : float
        The Davies-Bouldin index
```

```
    """
    K = len(clusters.keys())
    mean_distances = np.vstack(distance.cdist(centroids[x, np.newaxis],
                                              X.take(clusters[x], axis=0),
                                              metric=metric).mean()
                               for x in clusters.keys())
    D = np.zeros((K, K))
    for i in range(K):
        for j in range(i+1, K):
            D[i, j] = (mean_distances[i] + mean_distances[j]) /
distance.cdist(centroids[i, np.newaxis], centroids[j, np.newaxis], metric=metric)

    return (1/K) * np.sum(D.max(axis=1))

def DB_index3D(centroids, clusters, X, metric='euclidean', mode=1):
    """
    Return the Davies-Bouldin index for cluster evaluation for 3D data.

    The index is calculated using the formula:
    DB = (1/k) * sum(max((sigma_i + sigma_j)/distance(c_i, c_j)))
    where:
        k is the number of clusters
        sigma_i is the average distance of all elements in cluster i to
            centroid c_i
        c_i is the centroid of cluster i
        distance(c_i, c_j) is the distance between cluster centroids i and
            j, using the same metric as in the algorithm

    The smaller the value of the DB index, the better the clustering.

    Parameters
    ----------
    centroids : ndarray of float
        The centroids of the clusters
    clusters : dict
        Cluster id mapped to a list of the cluster's elements
    X : ndarray (3D)
        The data array
    metric : str or callable, optional
        The distance metric to use.  If a string, the distance function can be
        'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation',
        'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski',
        'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao',
        'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
        'wminkowski', 'yule'.
    mode : {0, 1, 2}, default 1
        Determines the axis along which to do the clustering.
        mode=i means that the clustering will be done along axis i.

    Returns
    -------
    DB_ind : float
        The Davies-Bouldin index
    """
    centroids = np.nan_to_num(centroids)
    K = len(clusters.keys())

    mean_distances = []
    non_empty_clusters = []

    for i in sorted(clusters.keys()):
        # Ignore current cluster if empty
        if (len(clusters[i]) == 0):
            mean_distances.append(0)
            continue

        non_empty_clusters.append(i)
        # Choose only the elements of cluster i
```

```
        i_data = np.nan_to_num(X.take(clusters[i], axis=mode))

        if (mode == 0) or (mode == 1):
            # Convert i_data to 2-D array by flattening the chromosomes
            # dimension
            i_data = np.reshape(i_data, (i_data.shape[0]*i_data.shape[1],
                                         i_data.shape[2]))
            # Remove the NaN rows
            i_data = i_data.take(np.unique(np.nonzero(i_data)[0]), axis=0)
        else:
            # Take the mean of each sample for all chromosomes
            i_data = np.nan_to_num(np.nanmean(i_data, axis=0))

            i_data = np.swapaxes(i_data, 0, 1)

        # Compute the mean distances of each cluster element to the
        # cluster's centroid
        mean_distances.append(distance.cdist(centroids[i, np.newaxis],
                                             i_data).mean())

    # Compute the distortion of each cluster
    D = np.zeros((K, K))
    for i in range(K):
        if (i not in non_empty_clusters):
            continue
        for j in range(i+1, K):
            if (j not in non_empty_clusters):
                continue
            D[i, j] = (mean_distances[i] + mean_distances[j]) /
distance.cdist(centroids[i, np.newaxis], centroids[j, np.newaxis])

    return (1/len(non_empty_clusters)) * np.sum(D.max(axis=1))

def modified_Gamma_index(centroids, clusters, labels, X, metric='euclidean'):
    """
    Return the Modified Hubert Gamma statistic.

    This statistic is calculated using the formula:
    Gamma = (1/M) * sum(sum(P(i,j)*Q(i,j)))
    where:
        M is the number of all possible pairs of X datapoints
        P is the similarity (distance) matrix of X datapoints
        Q is the matrix whose element Q(i,j) holds the distances between the
        centroids of the clusters that xi and xj belong to

    High values of the Modified Hubert Gamma statistic indicate the existence
    of compact clusters.

    Parameters
    ----------
    centroids : ndarray of float
        The centroids of the clusters
    clusters : dict
        Cluster id mapped to a list of the cluster's elements
    labels: ndarray of int
        The label of each datapoint, i.e. the cluster id it belongs to
    X : ndarray (2D)
        The data array
    metric : str or callable, optional
        The distance metric to use.  If a string, the distance function can be
        'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation',
        'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski',
        'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao',
        'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
        'wminkowski', 'yule'.

    Returns
    -------
```

```
    Gamma : float
        The Modified Hubert Gamma index
    """
    # Compute the matrix Q where Q(i,j) is equal to the distance between the
    # representatives of the clusters where xi and xj belong.
    representatives = centroids.take(labels, axis=0)
    Q = distance.cdist(representatives, representatives, metric=metric)

    # Compute the similarity matrix P of X
    P = distance.pdist(X, metric=metric)
    P = distance.squareform(P)

    # Compute the total number of all possible pairs of X
    Gamma = 0
    for i in range(X.shape[0]-1):
        for j in range(i+1, X.shape[0]):
            Gamma += (P[i,j] * Q[i,j])
    M = (X.shape[0] * (X.shape[0] - 1)) / 2

    # Compute and return the statistic
    return (1 / M) * Gamma

def modified_Gamma_index3D(centroids, clusters, labels, X, metric='euclidean',
                           mode=1):
    """
    Return the Modified Hubert Gamma statistic for the 3D structure.

    This statistic is calculated using the formula:
    Gamma = (1/M) * sum(sum(P(i,j)*Q(i,j)))
    where:
        M is the number of all possible pairs of X datapoints
        P is the similarity (distance) matrix of X datapoints
        Q is the matrix whose element Q(i,j) holds the distances between the
        centroids of the clusters that xi and xj belong to

    High values of the Modified Hubert Gamma statistic indicate the existence
    of compact clusters.

    Parameters
    ----------
    centroids : ndarray of float
        The centroids of the clusters
    clusters : dict
        Cluster id mapped to a list of the cluster's elements
    labels: ndarray of int
        The label of each datapoint, i.e. the cluster id it belongs to
    X : ndarray (3D)
        The data array
    metric : str or callable, optional
        The distance metric to use.  If a string, the distance function can be
        'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation',
        'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski',
        'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao',
        'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
        'wminkowski', 'yule'.
    mode : {0, 1, 2}, default 1
        Determines the axis along which to do the clustering.
        mode=i means that the clustering will be done along axis i.

    Returns
    -------
    Gamma : float
        The Modified Hubert Gamma index
    """
    # Compute the matrix Q where Q(i,j) is equal to the distance between the
    # representatives of the clusters where xi and xj belong.
    representatives = centroids.take(labels, axis=0)
    Q = distance.cdist(representatives, representatives, metric=metric)
```

```
# Compute the similarity matrix P of X
if (mode == 0):
    X_mean = np.nanmean(X, axis=1)
elif (mode == 1):
    X_mean = np.nanmean(X, axis=0)
else:
    X_mean= np.swapaxes(np.nanmean(X, axis=0), 0, 1)
P = distance.pdist(X_mean, metric=metric)
P = distance.squareform(P)

# Compute the total number of all possible pairs of X
Gamma = 0
for i in range(X_mean.shape[0]-1):
    for j in range(i+1, X_mean.shape[0]):
        Gamma += (P[i,j] * Q[i,j])
M = (X_mean.shape[0] * (X_mean.shape[0] - 1)) / 2

# Compute and return the statistic
return (1 / M) * Gamma
```