ΒΑΣΙΛΕΙΟΣ ΠΑΠΑΣΑΚΕΛΛΑΡΙΟΥ 1762
ΙΩΑΝΝΗΣ ΠΑΠΑΣΑΚΕΛΛΑΡΙΟΥ 2520
ΕΥΑΓΓΕΛΟΣ ΤΡΙΑΝΤΑΦΥΛΟΥ 1777

ΜΕΘΟΔΟΙ ΟΜΑΔΟΠΟΙΗΣΗΣ

Έχουμε υλοποιήσει τους εξής αλγόριθμους ομαδοποίησης: **K_Means**, **Agglomerative Hierarchical Clustering** και **Spectral Clustering** χρησιμοποιόντας gaussian kernel για τον τελευταίο. Όλοι οι αλγόριθμοι έχουν υλοποιηθεί από το χέρι χωρίς να χρησιμοποιηθούν έτοιμες συναρτήσεις.

Βοηθητικές κλάσεις που χρησιμοποιούνται:

Το αρχείο **File_loader** περιέχει την κλάση **load_data** που χρησιμοποιείται για να φορτώσει ένα αρχείο δεδομενων καθώς και να εκτυπώσει τα δεδομένα στην οθόνη και αν χρειαστεί να γίνει shuffle των φορτομένων δεδομένων.

Το αρχείο **Classify.py** περιέχει την κλάση **classify**. Η κλάση αυτή παίρνει σαν όρισμα τα clusters που έχουν δημιουργηθεί απο τον αλγόριθμο ομαδοποίησης και κατηγοροποιεί το κάθε cluster στην κατηγορία spam ή not_spam αναλογα με την πλειοψηθούσα κατηγορία των δεδομένων.

Το αρχείο **Evaluation.py** περιέχει την κλάση **Evaluation_Metrics** η οποία υπολογίζει τις μετρικές purity και F_measure που χρησημοποιούνται για την αξιολόγηση των μεθόδων ομαδοποίησης. Επίσης υπάρχουν και κάποιες βοηθητικές μέθοδοι που υπολογίζουν που χρησιμοποιούνται για τον υπολογισμό του **Total F_measure**.

Το αρχείο **Information_Class** περιέχει την κλάση **Information** η οποία απλά εκτυπώνει στην οθόνη διάφορες πληροφορίες για τον αλγόριθμο.

Κλάσεις Αλγορίθμων Ομαδοποίησης:

K_means

Το αρχείο **K_means.py** περιέχει την κλάση **K_means** η οποία υλοποιεί τον αλγόριθμο K_means. Παίρνει σαν όρισμα το σύνολο δεδομένων, το σύνολο των cluster στο οποίο ο αλγόριθμος θέλουμε να ομαδοποιήσει τα δεδομένα(το default είναι 2), το πλήθος των επαναλήψεων που θέλουμε να τρέξει ο αλγόριθμος και μια επιπλέον βοηθητική μεταβλητή που δηλώνει απλά αν τρέχουμε τον K_means ή τον K_means σαν τελευταίο βήμα για τον Spectral Clustering .
-Η μέθοδος

```
def choose_centers(self):
    random_positions = random.sample(range(len(self.dataset)), self.k)
    centers = dict()
    for i in range(self.k):
        centers[i] = self.dataset[random_positions[i]]

    return centers
```

επιλέγει τυχαία κ δεδομένα απο το σύνολο δεδομένων τα οποία θα είναι τα αρχικά μας κέντρα και τα επιστρέφει.

-Η μέθοδος
```
def euclidean_distance(self, data1, data2, data_length):
        distance = 0
        for attribute in range(data_length):
                distance += pow((data1[attribute] - data2[attribute]), 2)
        return float(math.sqrt(distance))
```
υπολογίζει την ευκλίδεια απόσταση μεταξύ δεδομένων.

-Η
```
def algorithm_converged(self, previus_centers, new_centers):
    for center in new_centers:
      previus_center = previus_centers[center]
      new_center = new_centers[center]

      if self.euclidean_distance(previus_center, new_center, len(previus_center)-1) < 0.0001:
        return True
    return False
```

ελέγχει αν εχει επιτεχθεί σύγκλιση του αλγορίθμου. Ελέγχει κάθε φορά αν τα καινούργια κέντρα(τα οποία τα έχουμε πάρει παίρνοντας τον μέσο όρο των δεδομένων της κάθε ομάδας) σε σχέση με τα παλιά η διαφορά τους ειναι αμελητέα χρησιμοποιώντας την ευκλίδεια απόσταση. Αν η διαφορά τους είναι < 0.0001 τότε σημαίνει οτι τα κέντρα των ομάδων δεν μετακινούνται πλέον και ο αλγορίθμος έχει συγκλίνει.

-Η
```
def implementation(self):
    results = list()#contain 10 dict of final clusterizations after 10 times run of kmeans algorithm
    for i in range(self.iterations):#run the kmeans 10 times with different centers each time
      centers = self.choose_centers()
      #print(centers)
      print("Running the K-Means algorithm %dth time" % (i+1))
      start = time.time()
      while True:
        clusters = dict()

        for cluster in range(self.k):
          clusters[cluster] = list()

        for data in self.dataset:
          distances = [self.euclidean_distance(data, centers[center],len(data) -1) for center in
centers]
          clusters[distances.index(min(distances))].append(data)

        previus_centers = dict(centers)

        for cluster in clusters:
          centers[cluster] = np.average(clusters[cluster], axis=0)
        #print(len(clusters[0]), len(clusters[1]))

        if (self.algorithm_converged(previus_centers, centers)):
          results.append(clusters)
          break
      #print(clusters)
```

```
end = time.time()
execution_time = self.compute_execution_time(start, end)
classification = classify(clusters, len(self.dataset))
classes = classification.classification()
metrics = Evaluation_Metrics(classes, len(self.dataset))
purity = metrics.Purity()
totalF_measure = metrics.TotalF_measure()
information = Information(purity, totalF_measure, execution_time, self.alg)
information.print_information()
```

υλοποιεί τον αλγόριθμο. Εξωτερικά υπάρχει μια for η οποία τρέχει 10 φορές τον αλγόριθμο παίρνοντας κάθε φορα διαφορετικά αρχικά κέντρα ομάδων. Έπειτα έχουμε μια while true απο την οποία βγαίνουμε μόνο οταν εχει επιτεχθεί σύγκλιση. Μέσα σε αυτήν την while υπολογίζουμε τα cluster χρησιμοποιώντας λεξικό στο οποίο κάθε κλειδί δηλώνει το cluster και η τιμή είναι τα δεδομένα του κάθε cluster. Για κάθε δεδομένο του συνόλου δεδομένων ελέγχουμε την αποστασή του απο το κέντρο κάθε cluster χρησιμοποιώντας την ευκλίδεια απόσταση. Το κάθε δεδομένο καταλήγει σε εκείνη την ομάδα στην οποία η απόσταση του απο το κέντρο της ομάδας είναι η μικρότερη σε σχέση με τις άλλες αποστάσεις απο τα κέντρα των άλλων ομάδων. Μόλις κάθε δεδομένο έχει μπεί σε κάποιο cluster σύμφωνα με την παραπάνω διαδικασία σαν επόμενο βήμα υπολογίζουμε τα καινούργια κέντρα των ομάδων παίρνοντας τον μέσο όρο των διανυσμάτων της κάθε ομάδας. Έπειτα καθώς έχουμε βρεί τα καινούρια κέντρα ελέγχουμε για σύγκλιση.
Τέλος αν ο αλγόριθμος σύγκλινε κατηγοριοποιούμε κάθε cluster με την πλειοψηφεία των δεδομένων και υπολογίζουμε έπειτα τις μετρικές για αξιολόγηση.

Agglomerative Hierarchical Clustering

Το αρχείο **Agglomerative_Hierarchical_Clustering** περιέχει την κλάση **Agglomerative_Hierarchical_Clustering** η οποία υλοποιεί την ιεραρχική ομαδοποίηση.
Τα ορισματά της είναι το σύνολο των δεδομένων, το πλήθος των δεδομένων και ο αριθμός των cluster(default 2).

Έχουμε πάλι την μέθοδο euclidean_distance που υπολογίζει την απόσταση δύο δεδομένων.

-Η μέθοδος
```
def initialization(self):
    for i in range(self.number_of_data):
        self.clusters[i] = list()
        self.clusters[i].append(self.dataset[i])
```
φτιάχνει μια ομάδα για κάθε δεδομένο του dataset. Άρα μετα την αρχικοποίηση αυτή το κάθε δεδομένο θα περιέχεται στο δικό(ξεχωριστό) του cluster. Δηλαδή ξεκινάμε την ομαδοποίηση με κάθε δεδομένο σε διαφορετικό cluster.

-Η μέθοδος
```
def find_middle_representative(self, cluster_data):
    If len(cluster_data) == 1:#an to cluster periexei mono ena dianusma tote profanws to
                                meso einai to idio to dianusma
        return cluster_data[0]
    else:
        return np.average(cluster_data, axis=0)
```

βρίσκει τον αντιπρόσωπο του καθενός cluster.

-Η μέθοδος
```
def distance_of_clusters(self, fcluster_data, scluster_data):
    r1 = self.find_middle_representative(fcluster_data)
```

```
            r2 = self.find_middle_representative(scluster_data)

            return self.euclidean_distance(r1, r2, len(r1)-1)
```
βρίσκει την απόσταση δύο cluster υπολογίζοντας την ευκλίδεια απόσταση των αντιπροσώπων του κάθε cluster.

-Η μέθοδος
```
def delete_cluster(self, cluster_id):
            del self.clusters[cluster_id]
```
απλά διαγράφει το cluster me κλειδί cluster_id απο το λεξικό των cluster.

-Η μέθοδος
```
def merge_clusters(self, fcluster_id, scluster_id):
            self.clusters[fcluster_id].extend(self.clusters[scluster_id])
            self.delete_cluster(scluster_id)
```
συγχωνεύει δύο cluster παίρνοντας τα δεδομένα του ενός και τα αναθέτει στο αλλο. Έπειτα διαγράφει το άλλο cluster.

-Η
```
def implementation(self):
    number_of_clusters = len(self.clusters)
      level = 0
    self.initialization()
    start = time.time()
    print("Running Agglomerative Hierarchical Clustering algorithm...")
    sleep(2.0)
    while len(self.clusters) != self.number_of_clusters:
      c = []
      print("Level %d..."% level)
      for i, j in itertools.combinations(self.clusters, 2):
        distance = self.distance_of_clusters(self.clusters[i], self.clusters[j])
        c.append([distance, i, j])
      min_distance = min(c, key=lambda c: c[0])
      #print(min_distance)
      #print(len(self.clusters))
      self.merge_clusters(min_distance[1], min_distance[2])
      level += 1
      #print(self.clusters)
    end = time.time()
    execution_time = self.compute_execution_time(start, end)
    classification = classify(self.clusters, self.number_of_data)
    classes = classification.classification()
    metrics = Evaluation_Metrics(classes, self.number_of_data)
    purity = metrics.Purity()
    totalF_measure = metrics.TotalF_measure()
    information = Information(purity, totalF_measure, execution_time,
"Agglomerative_Hierarchical_Clustering")
    information.print_information()
```
υλοποιεί τον αλγόριθμο της ιεραρχικής ομαδοποίησης. Έχουμε μια while η οποία συνεχίζεται έως ότου το δέντρο να φτάσει στο επίπεδο στο οποίο θελουμε ανάλογα με τον αριθμό των cluster που θελουμε να καταλήξουμε. Κάθε φορά συγκρίνουμε το κάθε cluster με όλα τα άλλα βρίσκοντας την απόσταση μεταξύ τους.(Η απόσταση βρίσκεται υπολογίζοντας πρώτα τους αντιπρόσωπους των cluster και έπειτα παίρνοντας την ευκλίδεια απόσταση μεταξύ τους). Από τις αποστάσεις αυτές κρατάμε την ελάχιστη μεταξύ των δύο ομάδων και τις συγχωνεύουμε και επιπλέον διαγράφουμε το δεύτερο cluster. Αυτό έχει ως αποτέλεσμα σε κάθε επανάληψη ο αριθμός των τρέχοντων cluster να

μειώνεται κατα ένα κάθε φορά(σχηματίζουμε ουσιαστικά ένα δέντρο).Όλο αυτό γίνεται έως ότου να φτάσουμε στο επιθυμητο επίπεδο το οποίο θα περιέχει όσα cluster θέλουμε να έχει η τελική ομαδοποίηση. Τέλος κατηγοριοποιούμε τα τελικά cluster και υπολογίζουμε τις μετρικές.

Spectral Clustering

Το αρχείο **Spectral_Clustering.py** περιέχει την κλάση **Spectral_Clustering_GK**. Ο contructor της παίρνει σαν ορίσματα τα δεδομένα, το πλήθος των δεδομένων, την ελάχιστη απόσταση που πρέπει αν υπάρχει μεταξύ των δεδομένων έτσι ώστε να κρατηθεί στον affinity_matrix και το sigma το οποίο χρείαζεται στον gaussian kernel .
-Η μέθοδος
```
def gaussian_kernel(self, distance):
    return math.exp(-distance/2*self.sigma**2)
```
υπολογίζει τον gaussian kernel.
-Η
```
def Affinity_Matrix(self):
    affinity_matrix = np.zeros((self.number_of_data, self.number_of_data))#pinakas opou kathe timh tou einai h apostash tou shmeioi i apo to j
    for i in range(self.number_of_data):
        for j in range(i):
            if i == j:#to dedomeno apo ton eauto tou exei profanws mhdenikh apostash
                continue
            distance = self.euclidean_distance(self.dataset[i], self.dataset[j], len(self.dataset[i])-1)
            if distance < self.data_distance_limit:
                value = self.gaussian_kernel(distance)
                affinity_matrix[i][j] = value
                affinity_matrix[j][i] = value

            else:
                affinity_matrix[i][j] = 0
                affinity_matrix[j][i] = 0

    return affinity_matrix
```
υπολογιζει τον πίνακα ομοιότητας. Κάθε γραμμή του πίνακα συμβολίζει τον βαθμό ομοιότητας των δεδομένων . Αν ο βαθμός ομοιότητας ξεπερνάει το ελάχιστο όριο τότε δεμ υπάρχει ομοιότητα μεταξύ των δεδομένων και η αντίστοιχη τιμή του πινακα παίρνει τιμή 0. Αν δεν το ξεπερνάει αποθηκεύουμε τον βαθμό ομοιότητας αφού περάσουμε πρώτα την απόσταση απο τον gaussian πυρήνα. Επίσης ο πίνακας είναι συμμετρικός που σημαίνει ότι η ομοιότητα πχ του δεδομένου 1 απο το 2 είναι ίδια με την ομοιότητα του 2 από το 1.
-Η
```
def Degree_Matrix(self, affinity_matrix):
    D = np.zeros((self.number_of_data, self.number_of_data))#pinakas opou kathe diagwnio stoixeio periexei to athroisma ths kathe grammhs tou affinity matrix
    sum_of_each_row = np.sum(affinity_matrix, axis=1)
    for i in range(self.number_of_data):
        D[i][i] = sum_of_each_row[i]
    return D
```

υπολογίζει τον degree matrix ο οποίος είναι ένας διαγώνιος πίνακας όπου στην κύρια διαγωνιό του περιέχει τον άθροισμα των αποστάσεων των δεδομένων από τα άλλα δεδομένα. Οπότε χρησημοποιούμε τον affinity matrix που υπολογίσαμε προηγουμένως και κάθε στοιχείο της κύριας διαγωνίου του degree matrix θα είναι το άθροισμα των στοιχείων της αντίστοιχης γραμμής του affinity matrix.

-H
```
def Laplacian_Matrix(self, A, D):
    return D-A
```
υπολογίζει τον Laplacian πίνακα χρησιμοποιώντας τον affinity και τον degree matrix.
-H
```
def Normalized_Laplacian_Matrix(self, A, D):
    DM = np.copy(D)
    L = self.Laplacian_Matrix(A, D)
    for i in range(len(D)):
        DM[i][i] = 1.0/(D[i][i]*float(0.5))

    return DM.dot(L).dot(DM)
```
υπολογίζει τον normalized Laplacian.
-H
```
def find_Eigenvalues_Eigenvectors(self, L):
    print(np.isnan(L).any())
    L = np.nan_to_num(L)
    eigenvalues, eigenvectors = np.linalg.eig(L)
    return eigenvalues.real, eigenvectors.real
```

υπολογίζει τις ιδιοτιμές και τα ιδιοδιανύσματα του Laplacian πίνακα.
-H
```
def choose_k(self, eigenvalues):

    eigengap = eigenvalues[2] - eigenvalues[1]
    for i in range(3, eigenvalues.shape[0]):
        if eigenvalues[i] - eigenvalues[i-1] > eigengap:
            eigengap = eigenvalues[i] - eigenvalues[i-1]
            self.k = i
```
υπολογίζει το κατάλληλο κ επιλέγοντας την μέγιστη διαφορά μεταξύ των ιδιοτιμών.
-H
```
def Spectral_Analysis_Transformation(self, L, eigenvalues, eigenvectors):
    #self.choose_k(eigenvalues)
    #print(self.k)
    idx = np.argsort(eigenvalues)
    eigenvalues = eigenvalues[idx]
    print(eigenvalues)
    eigenvectors = eigenvectors[:,idx]
    new_data = eigenvectors[:,-self.k:]

    return new_data
```
υλοποιεί την φασματική ανάλυση. Ταξινομεί τις ιδιοτιμές και τα ιδιοδιανύσματα και επιλέγει τα top k ιδιοδιανύσματα.
-H
```
def label_data(self, transformed_data):
    labels = list()
        for i in range(self.number_of_data):
        labels.append([self.dataset[i][len(self.dataset[i])-1]])
    return np.append(transformed_data, labels, axis=1)
```
κατηγοριοποιεί τα καινούργια δεδομένα.
-H
```
def plot_eigenvalues(self, eigenvalues):
    plt.title('Eigenvalues of Laplace Matrix')
    plt.scatter(np.arange(len(eigenvalues)), eigenvalues)
    plt.grid()
    plt.show()
```
πλοτάρει τις ιδιοτιμές του Laplacian matrix.

-H
```
def implementation(self):
    print("Running Spectral Clustering algorithm...")
    sleep(2.0)
    start = time.time()
    A = self.Affinity_Matrix()
    D = self.Degree_Matrix(A)
    #L = self.Laplacian_Matrix(A, D)
    NL = self.Normalized_Laplacian_Matrix(A, D)
    eigenvalues, eigenvectors = self.find_Eigenvalues_Eigenvectors(NL)
    self.plot_eigenvalues(eigenvalues)
    transformed_data = self.Spectral_Analysis_Transformation(NL, eigenvalues, eigenvectors)
    #print(transformed_data)
       new_data = self.label_data(transformed_data)
       #print(new_data)
    classify = K_means(new_data, self.k , 1, "Spectral_Clustering")
    classify.implementation()
    end = time.time()
```

καλεί όλες τις παραπάνω συναρτήσεις και σαν τελευταίο βήμα χρησιμοποιεί τον K_means στα καινούρια δεδομένα.

Αποτελέσματα μεθόδων

Για το αρχείο δεδομένων spambase.data τα αποτελέσματα του K_means και του Spectral Cluster καθώς δεν ήταν δυνατό να ελέξουμε τον Hierarchical για 4096 δεδομένα καθώς υπολογιστικά ήταν πολύ βαρή.

K_means(k=2)		**Spectral_Clustering**(k=2)	
Purity	**Total F_measure**	**Purity**	**Total F_measure**
0.635949	1.649453	0.607042	1.448374

K_means(k=4)		**Spectral_Clustering**(k=4)	
Purity	**Total F_measure**	**Purity**	**Total F_measure**
0.676375	3.306079	0.662247	3.116158

Για ένα τεστ αρχείο spambasetext.data με λιγότερα δεδομένα τα αποτελέσματα ήταν τα εξής:

K_means(k=2)		**Agglomerative_H_Clustering**(k=2)		**Spectral_Clustering**(k=2)	
Purity	**Total F_measure**	**Purity**	**Total F_measure**	**Purity**	**Total F_measure**
0.592058	1.632820	0.593261	1.744327	0.592058	1.605157

PS: Όλες τις μεθόδους τις υλοποιήσαμε απο το χέρι σε πολύ χαμηλό επίπεδο και δεν πήραμε τίποτα έτοιμο γιαυτό οι μέθοδοι αργούνε να τερματίσουν.Επίσης στην Spectral _Clustering μέθοδο έχουμε βάλει σταθερό κ καθώς υπολογίζοντας το κ με το μέγιστο eigengap βγαίνει μεγάλο με αποτέλεσμα κάποια cluster να μην παίρνουν καθόλου δεδομένα.
Επίσης όπως αναφέραμε η ιεραρχική ομαδοποίηση είναι παρα πολυ αργή με τα αρχικά δεδομένα όπως και επίσης με τα δεδομένα που δημιουργήσαμε εμείς(κάνει 55 λεπτά να τελειώσει με 830 δεδομένα) και δεν μπορέσαμε να συγκρίνουμε καλά με τις άλλες μεθόδους!
Επίσης αν τρέξετε τον Spectral_Clustering επειδή χρησιμοποιεί αντικείμενο K_means στο τέλος και για κάποιο λόγο τρέχει ο K_means ο κανονικός, αν μπορείται να βάλετε σε σχόλια τις τελευταίες γραμμές (123-126) στο αρχείο K_means.py.