Αναγνώριση Προτύπων

ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ, 2017-2018

Παναγιώτα Θωμοπούλου | Π14053
Γιωργός Κονσουλάκης | Π14078
Γεράσιμος Δεληβοριάς | Π14029

Εισαγωγή

Για την επίλυση της εργασίας χρησιμοποιήθηκε η γλώσσα python 3.6 και οι βιβλιοθήκες:

- Numpy
- Matplotlib
- Scikit-learn
- Scipy

Προετοιμασία dataset – saveasnumpy.py

Για την χρήση των δεδομένων, χρειάστηκε πρώτα η κατάλληλη επεξεργασία τους. Οι προτιμήσεις των χρηστών θεωρήθηκαν οι πρώτες 18 στήλες με τα είδη των ταινιών (Action, Adventure, ..., Western), και η 19η τα ratings.

Για να τοποθετηθούν τα δεδομένα στο πρόγραμμα, χρησιμοποιήθηκε το αρχείο python saveasnumpy.py.

Αρχικά, αρχικοποιήθηκε ο πίνακας ratings

```
ratings = np.arange(400000).reshape(100000, 4)
```

που περιέχει τα ids και τα ratings. Διαβάστηκαν από το αρχείο όλα τα ratings και τοποθετήθηκαν στον πίνακα ανά γραμμή.

```
fileStream = open('u.data', 'r')
lines = fileStream.readlines()
i= 0
for line in lines:
    ratings[i]= line.split("\t")
    i += 1
fileStream.close()
```

Ομοίως για τους χρήστες και τις ταινίες.

Το τελικό dataset τοποθετήθηκε στο array dataset, μεγέθους 100.000*19.

```
dataset = np.zeros((100000, 19), dtype=float)
```

Στις πρώτες 18 στήλες του dataset τοποθετήθηκαν οι 18 στήλες των ειδών στο movies,

```
for j in range(18):
    dataset[i][j] = movies[movieId][j + 6]
```

ενώ στην τελευταία στήλη τοποθετήθηκε το rating της ταινίας του κάθε χρήστη κανονικοποιημένο, για να βρίσκεται σε τιμές μεταξύ 0 και 1

```
dataset[i][18] = rating[2]*0.25-0.25
```

Τέλος, γίνεται αποθήκευση του αρχείου σε μορφή .npy, για πιο εύκολη και γρήγορη εκτέλεση του υπόλοιπου κώδικα.

```
np.save('dataset.npy', dataset)
```

Basic Sequential Algorithmic Scheme (BSAS)

Για τον αλγόριθμο BSAS χρειάστηκε να οριστούν 2 σταθερές. Τον μέγιστο αριθμό ομαδοποιήσεων και τον δείκτη διαφοροποίησης.

```
# maximum number of clusters to create
MAX_CLUSTER_NUMBER = 20
# threshold for how different 2 vectors can be based on their distance
THRESHOLD = 2.45
```

Επιπλέον, ορίζεται η συνάρτηση FindDistance, που παίρνει 2 διανύσματα -μονοδιάστατα arrays- και επιστρέφει την ευκλείδεια απόστασή τους,

```
# finds the euclidean distance between 2 vectors
def FindDistance(vec1, vec2):
    distance = 0
    for i in range(vec1.size):
        distance += (vec1[i]-vec2[i])**2
    distance = math.sqrt(distance)
    return distance
```

και η συνάρτηση FindClosestCluster, η οποία δέχεται σαν ορίσματα ένα διάνυσμα, ένα array με όλες τις θέσεις των υπαρχόντων clusters και τον αριθμό τους.

```
# finds the closest possible cluster from a vector, given the vector
itself,
# the current clusters locations and the number of existing clusters
def FindClosestCluster(vec1, clusterPositions, currentClustersNumber):
    numberOfCluster = 0
    minumunClusterDistance = FindDistance(vec1, clusterPositions[0])
    for i in range(currentClustersNumber):
        currentClusterDistance = FindDistance(vec1,
clusterPositions[i])
        if (minumunClusterDistance> currentClusterDistance):
            minumunClusterDistance = currentClusterDistance
            numberOfCluster = i
    return numberOfCluster
```

Η συνάρτηση επιστρέφει τον αριθμό του cluster που βρίσκεται πιο κοντά στο διάνυσμα.

Για να τρέξει ο αλγόριθμος, χρειάζεται να φορτωθεί ο πίνακας dataset από το αρχείο dataset.npy:

```
# loads the dataset from saveasnumpy
dataset = np.load('dataset.npy')
```

Οι μεταβλητές που θα χρησιμοποιηθούν είναι αρχικά, ο πίνακας ClusterPositions, μεγέθους MAX_CLUSTER_NUMBER * Αριθμός διαστάσεων dataset (20x18). Στον πίνακα αυτόν θα τοποθετηθεί η θέση του κάθε cluster στον διανυσματικό χώρο.

Ο επόμενος πίνακας assignedCluster, περιέχει τον αριθμό του cluster στον οποίο ανήκει (label).

Τέλος, ο ακέραιος currentClustersNumber δείχνει τον αριθμό των υπαρχόντων cluster.

Μετά την αρχικοποίηση των μεταβλητών,

```
clusterPositions[0] = dataset[0]
assignedCluster[0] = 0
currentClustersNumber = 1
```

γίνεται προσπέλαση όλων των διανυσμάτων του dataset.

```
for i in range(dataset.shape[0]):
```

Για κάθε ένα dataset βρίσκεται το πιο κοντινό cluster σε αυτό,

```
currentClosestCluster =
FindClosestCluster(dataset[i],clusterPositions,currentClustersNumber)
```

και γίνεται έλεγχος για το αν η απόσταση από αυτό είναι μεγαλύτερη από το threshold, και αν δεν έχει συμπληρωθεί ο μέγιστος αριθμός cluster.

```
if(FindDistance(clusterPositions[currentClosestCluster],
dataset[i])>THRESHOLD and currentClustersNumber<MAX_CLUSTER_NUMBER):
```

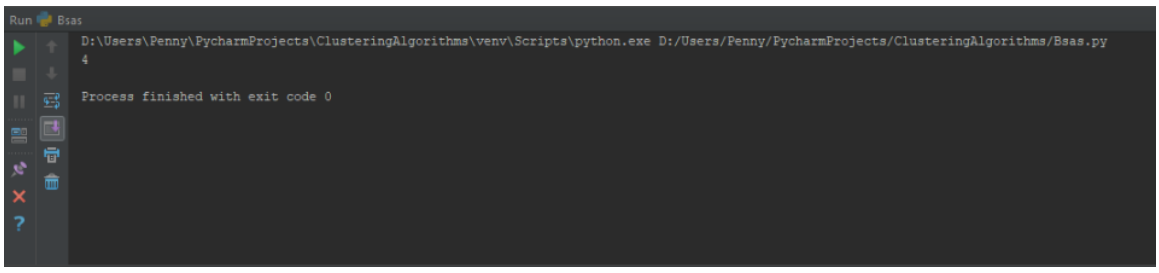
Αν όντως γίνει αυτό, δημιουργείται το καινούργιο cluster, και ανανεώνονται οι κατάλληλοι πίνακες και μεταβλητές, αλλιώς το σημείο τοποθετείται σε ένα ήδη υπάρχον cluster, αυτό που επέστρεψε η συνάρτηση.

```
clusterPositions[currentClustersNumber] = dataset[i]
    assignedCluster[i] = currentClustersNumber
    currentClustersNumber += 1
else:
    assignedCluster[i] = currentClosestCluster
```

Όταν τελειώσει η επανάληψη, γίνεται εκτύπωση του αριθμού των cluster.

```
print(currentClustersNumber)
```

ΠΑΡΑΔΕΙΓΜΑ ΕΚΤΕΛΕΣΗΣ



ΕΠΕΞΗΓΗΣΗ THRESHOLD ΚΑΙ MAX_NUMBER_OF_CLUSTERS

Στον παραπάνω κώδικα, τέθηκαν οι σταθερές για τον μέγιστο αριθμό cluster και τον δείκτη διαφοροποίησης αυθαίρετα. Στην πραγματικότητα, θα έπρεπε να γίνει εκτέλεση του αλγόριθμου με for από την μικρότερη απόσταση μεταξύ των cluster, μέχρι την μεγαλύτερη απόσταση, με ένα βήμα, και το πιο συνηθισμένο θ που θα προκύψει, να επιλεγεί.

Το θ = 2,45 επιλέχθηκε με βάση τους επόμενους αλγορίθμους ομαδοποίησης. Ο λόγος που δεν υλοποιήθηκε ο παραπάνω τρόπος, είναι επειδή για το μεγάλο dataset της εργασίας, παίρνει πάρα πολύ χρόνο, οπότε δεν ήταν δυνατό να ελεγχθεί το αν έτρεχε σωστά.

Αλγόριθμος k-means

Για τον αλγόριθμο k-means, χρησιμοποιήθηκε η βιβλιοθήκη scikit-learn.

Αφού γίνει φόρτωση του dataset από το npy αρχείο,

```
# loads the dataset from saveasnumpy
dataset = np.load('dataset.npy')
```

Ορίζεται ο αριθμός των cluster που θα δημιουργηθούν.

```
# total cluster number based on the result of BSAS
k = 4
```

Ο αριθμός που επιλέχθηκε είναι 4, με βάση το αποτέλεσμα που εμφάνισε ο αλγόριθμος BSAS.

Στη συνέχεια εφαρμόζεται ο αλγόριθμος k-means της βιβλιοθήκης.

```
# apply k-means with sk learn library, based on the number of clusters
we found via BSAS
kmeans = KMeans(n_clusters=k, random_state=0).fit(dataset)
```

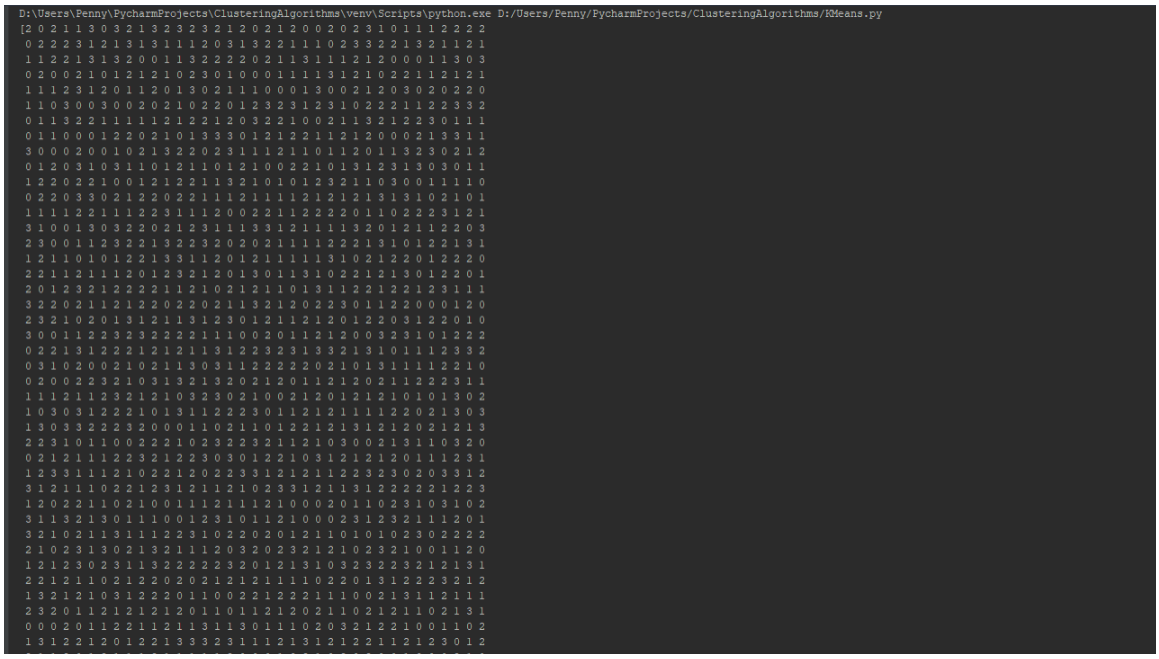
Από εκεί αρκεί να βρεθούν τα labels του κάθε διανύσματος, δηλαδή τον αριθμό του cluster του οποίου ανήκουν.

```
# get the labels for each rating, basically the number of cluster every
rating belongs to
kmeansLabels = kmeans.labels_
print(kmeans.labels_)
```

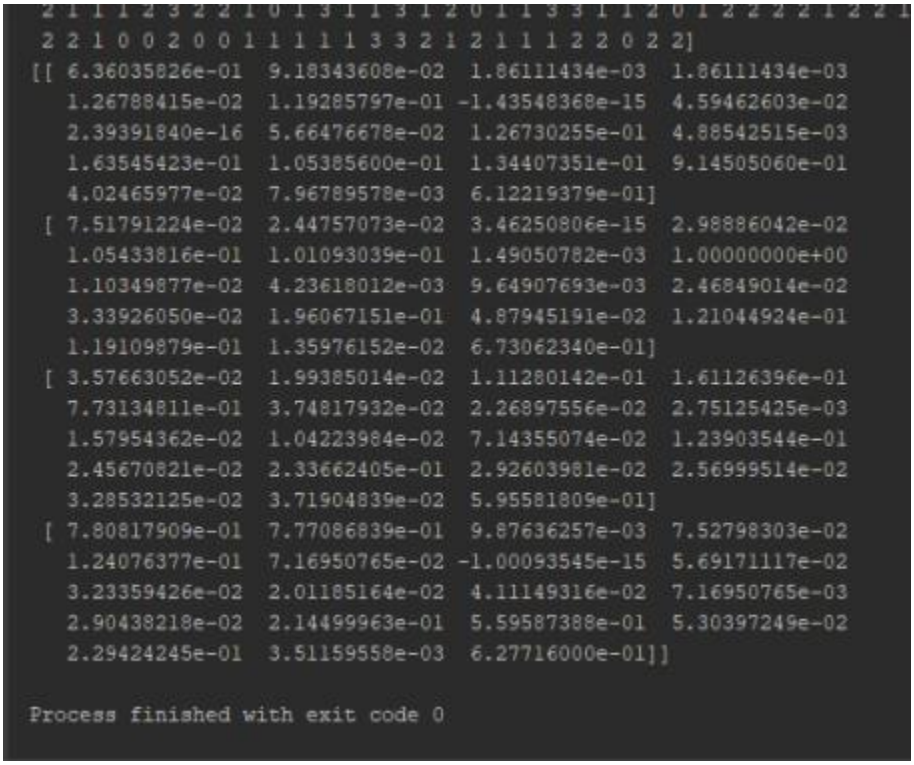
Τέλος, εμφανίζονται τα κέντρα του κάθε cluster:

```
# get the position of every cluster center
clusterCenters = kmeans.cluster_centers_
print(clusterCenters)
```

ΠΑΡΑΔΕΙΓΜΑ ΕΚΤΕΛΕΣΗΣ ΚΩΔΙΚΑ



Labels κάθε διανύσματος



Θέσεις κέντρων

Ιεραρχικός αλγόριθμος ομαδοποίησης

Για τον αλγόριθμο αυτό, χρησιμοποιήθηκε η βιβλιοθήκη scipy, και για την αναπαράσταση του δενδρογράμματος η βιβλιοθήκη matplotlib.

Επειδή οι αλγόριθμοι ιεραρχικής ομαδοποίησης δεν έχουν καλή κλιμάκωση με μεγάλα δεδομένα, χρησιμοποιήθηκε ένα μικρότερο μέρος του dataset. Αν χρησιμοποιηθεί ολόκληρο το dataset, θα υπάρξει πρόβλημα μη επαρκής μνήμης.

Έτσι λοιπόν, φορτώνουμε τον πίνακα dataset, και παίρνουμε μόνο τα πρώτα 1000 δεδομένα:

```
# importing dataset from saveasnumpy
dataset = np.load('dataset.npy')

# reducing the size of dataset to the first 1000 ratings
X = dataset[:-99000]
```

Χρησιμοποιώντας την βιβλιοθήκη, κάνουμε την ένωση των διανυσμάτων.

```
# creating links between ratings(vectors) based on ward linking
Z = linkage(X, 'ward')
```

Ο τρόπος ένωσης που επιλέχθηκε ήταν ο ward, επειδή είναι πιο ακριβής στις αποστάσεις μεταξύ των διανυσμάτων και των clusters.

Χρησιμοποιώντας την βιβλιοθήκη matplotlib, δημιουργούμε και εμφανίζουμε το δενδρόγραμμα:

```
# calculating full dendrogram
plt.figure(figsize=(25, 10))
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('sample index')
plt.ylabel('distance')
dendrogram(
    Z,
    leaf_rotation=90.,  # rotates the x axis labels
    leaf_font_size=8.,  # font size for the x axis labels
)
plt.show()
```

Με βάση τον αριθμό των cluster που βρήκαμε από τον BSAS, επιλέγουμε τα clusters του dataset.

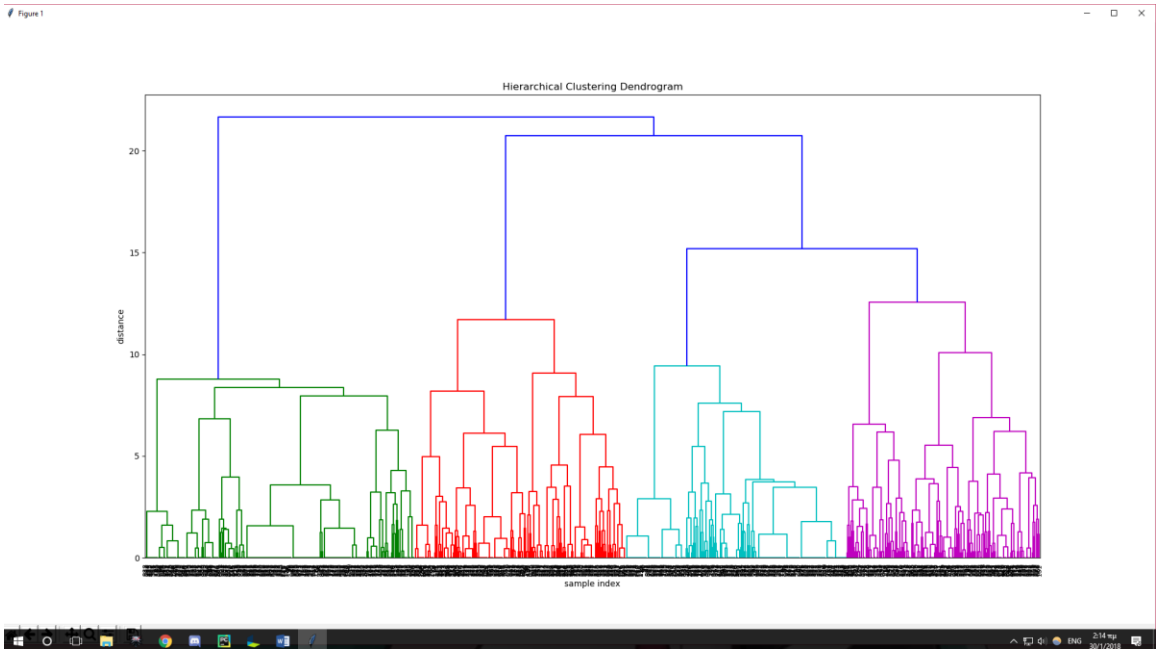
```
# maximum number of clusters, based on BSAS algorithm
k = 4

# get clusters based on k
clusters = fcluster(Z, k, criterion='maxclust')
```

Τέλος, εμφανίζουμε σε ποιον αριθμό cluster ανήκει το κάθε διάνυσμα (labels):

```
# print on which number of clusters each rating belongs to
print(clusters)
```

ΠΑΡΑΔΕΙΓΜΑΤΑ ΕΚΤΕΛΕΣΗΣ ΚΩΔΙΚΑ



Δενδρόγραμμα

Labels του dataset

ΣΥΜΠΕΡΑΣΜΑ – ΣΗΜΑΣΙΑ ΟΜΑΔΟΠΟΙΗΣΗΣ

Το συμπέρασμα που μπορούμε να βγάλουμε από την ομαδοποίηση είναι ότι οι προτιμήσεις των χρηστών, χωρίζονται κυρίως σε 4 κατηγορίες. Δηλαδή, κάποιος μπορεί να πει, ότι οι διαφοροποίηση των χρηστών δεν είναι μεγαλύτερη από τις 4 και ότι οι χρήστες και οι προτιμήσεις τους είναι σχετικά όμοιες με το ¼ των υπόλοιπων χρηστών.

Ερώτημα 4

Σύμφωνα με το 5-fold σχήμα δημιουργήσαμε 2 ταξινομητες. 'Εναν νευρωνικού δικτύου και εναν ελαχίστων τετραγώνων.

ΝΕΥΡΩΝΙΚΟ ΔΙΚΤΥΟ

Το νευρωνικό δίκτυο που σχεδιάσαμε περιεχει 2 κρυφά επίπεδα (hidden layers) των 10 επιπέδων το καθένα.

```
hl1 = 10
hl2 = 10
```

Γενικά, ο σκοπός του δικτύου μας είναι να δέχεται ως είσοδο έναν πίνακα με βαθμολογίες και να βγάζει ως έξοδο τον ίδιο πίνακα.

Η συνάρτηση της αρχικοποίησης του αντικειμένου είναι η εξής:

```
def __init__(self, inputs, outputs): #neural network constructor
    random.seed(1) #takes desired inputs and outputs for testing

    self.input = inputs
    self.output = outputs

    self.layer_1_nodes = self.hl1
    self.layer_2_nodes = self.hl2

    self.input_shape = (1, 1682)
    self.output_shape = (1, 1682)

    self.weights_1 = 2 * random.random((self.input_shape[1], self.layer_1_nodes)) - 1
    self.weights_2 = 2 * random.random((self.layer_1_nodes, self.layer_2_nodes)) - 1

    self.out_weights = 2 * random.random((self.layer_2_nodes, self.output_shape[1])) - 1
```

όπου αρχικοποιούνται τα βάρη με τυχαίες τιμές μεταξύ του -1 και του 1 αλλά και οι επιθυμητές διαστάσεις των εισόδων και εξόδων όπως είναι λογικό εφόσον κάθε χρήστης βαθμολογεί 1682 διαφορετικές ταινίες το πολύ.

Μετά, ορίσαμε την συνάρτηση "sigmoid" η οποία κάνει κανονικοποίηση των δεδομένων ώστε να μην παίρνουν ποτέ τιμές μικρότερες του -10,

```
def sigmoid(self, x): #sigmoid function used
    for i in range(len(x)):
        for j in range(np.shape(x)[1]):
            if (x[i][j] < -10): x[i][j] = -5

    return 1 / (1 + exp(-x))
```

και την "sigmoid_derivative" μέσω της οποίας υπολογίζεται η παράγωγος της απόστασης "X" απο τα δεδομένα μας.

```
def sigmoid_derivative(self, x): #calculates sigmoid dirivative
    return x * (1 - x)
```

Στην συνέχεια ορίσαμε την συνάρτηση "think". Η συνάρτηση αυτή κάνει μία 'πρόβλεψη' των αποτελεσμάτων του δικτύου. Με βάση αυτά τα αποτελέσματα βλέπουμε κατά πόσο είναι ακριβές το δίκτυό μας.

```
def think(self, x, outs):
    # Inputs are multiplied with weights and then passed to next layer
    layer1 = self.sigmoid(dot(x, self.weights_1))
    layer2 = self.sigmoid(dot(layer1, self.weights_2))
    output = self.sigmoid(dot(layer2, self.out_weights))

    for i in range (943):
        for j in range(1682):
            if (output[i][j] >=  0.5):
                output[i][j] = 1.0 #activation function
            else:
                output[i][j] = 0.0
    return x - output
```

Η τελευταία συνάρτηση που ορίζεται, είναι η "train". Αυτή στην ουσία πραγματοποιεί τον σκοπό που πρέπει εφόσον είναι αυτή η οποία υπολογίζει τα αποτελέσματα τα οποία συγκρίνει με τα αποτελέσματα που εικάζονται να έρθουν, και με βάση τους υπολογισμούς αυτούς αλλάζει τις τιμές των βαρών με σκοπό να κινηθεί προς την σωστή κατεύθυνση.

```
def train(self): #training function
    layer1 = self.sigmoid(dot(self.input, self.weights_1))

    layer2 = self.sigmoid(dot(layer1, self.weights_2))

    output = self.sigmoid(dot(layer2, self.out_weights))

    outputError = self.output - output

    delta = outputError * self.sigmoid_derivative(output) #calculates delta of output

    out_weights_adjustment = dot(layer2.T, delta) #adjusts all the weights using backpropagation algorithm

    self.out_weights += out_weights_adjustment

    delta = dot(delta, self.out_weights.T) * self.sigmoid_derivative(layer2)

    weight_2_adjustment = dot(layer1.T, delta)
    self.weights_2 += weight_2_adjustment

    delta = dot(delta, self.weights_2.T) * self.sigmoid_derivative(layer1)
    weight_1_adjustment = dot(self.input.T, delta)
    self.weights_1 += weight_1_adjustment
```

Τέλος, σας παρουσιάζουμε των κώδικα που χρησιμοποιεί τις παραπάνω συναρτήσεις στα δεδομένα του αρχείου "u1.base":

```
if __name__ == '__main__': #testing and initializations
    base = np.arange(4 * 80000).reshape(80000, 4)

    # Read all from base file 1
    fileStream = open('u1.base', 'r')
    lines = fileStream.readlines()
    i = 0
    for line in lines:
        base[i] = line.split("\t")
        i += 1
    fileStream.close()

    usersMovies = np.zeros((943, 1682), dtype='float32')

    for i in range(80000):
        usersMovies[base[i][0] - 1][base[i][1] - 1] = 1



    neuralN = NN(usersMovies, usersMovies)

    neuralN.train() #trains neural network

    test = np.arange(4 * 20000).reshape(20000, 4)
```

```
# Read all from test file 1
fileStream = open('u1.test', 'r')
lines = fileStream.readlines()
i = 0
for line in lines:
    test[i] = line.split("\t")
    i += 1
fileStream.close()

tests = np.zeros((943, 1682), dtype='float32')

for i in range(20000):
    tests[test[i][0] - 1][test[i][1] - 1] = 1

results = neuralN.think(tests,tests) #tests the erros made for test file
errors = 0
for j in range (943):
    for i in range (1682):
        if(results[j][i] != 0.0):
            errors += 1
print (errors)
```

ΤΑΞΙΝΟΜΗΤΗΣ ΕΛΑΧΙΣΤΩΝ ΤΕΤΡΑΓΩΝΩΝ

Για την υλοποίηση του εξής ταξινομητή κάναμε χρήση μιας αρχιτεκτονικής όμοιας με τον αλγόριθμο perceptron. Πιο λεπτομερώς, αφού έχουν ορισθεί και αρχικοποιηθεί τα βάρη, γίνεται μια πρόβλεψη των αποτελεσμάτων με βάση τον perceptron και τα λάθη που προκύπτουν διορθώνεται με βάση την απόσταση ελαχίστων τετραγώνων. Συγκεκριμένα, η αρχικοποίηση γινεται ως εξής:

```
class leastSquare: #the class for least square algorithm
    def __init__(self, inputs, custom): #constructor
        if (custom == 0): #sets weights using 1 more input as 1 and its weight as bias
            self.weights = 2 * random.random((len(inputs[0]) + 1, len(inputs[0]))) - 1
        else: #takes the weights already calculated for quicker testing
            self.weights = inputs
```

Ενώ η εκπαίδευση:

```
def train(self, sampleArray, expectedResults, numOfInputs, Lrate): #finds the weights that give  least square
    MSEofOutputGate = np.zeros(1682)
    newArray = np.ones((len(sampleArray), 1))
    sampleArray = np.append(newArray, sampleArray, axis=1)
    for j in range (len(MSEofOutputGate)): #for better testing the success of each individual output gate is calculated indi
        while(True):
            miniSquareErrors = np.zeros(numOfInputs)
            MSEofOutputGate[j] = 0
            for i in range(numOfInputs):
                miniSquareErrors[i] = self.predict(sampleArray,i, j) - expectedResults[i][j]
                MSEofOutputGate[j] += miniSquareErrors[i]**2
            if (MSEofOutputGate[j] < 0.0):
                print("MSE of gate: ", j, "fixed") # prints if the gate has a success rate close to 0.0
                break #breaks only if the gate gives the least square
            for i in range(1683): #calculates the partial dirivative of the output  with respect to each weight
                partialDir = 0
                for k in range(numOfInputs):
                    partialDir += 2 * miniSquareErrors[k] * sampleArray[k][i] #changes the weights
                self.weights[i,j] -= Lrate * partialDir
    MSE = 0
    for i in range(1682):
        MSE += MSEofOutputGate[i]
```

Για την αποτίμηση του ταξινομητή "γράψαμε" την συνάρτηση test:

```
def test(self, x , y): #tests the leastSquare but doesn't update it's weights
    newArray = np.ones((len(x), 1))
    x = np.append(newArray, x, axis=1)
    rightAnswers = 0 #counts the right answers the algorithm makes
    outOf = 0
    for k in range (943):
        for j in range (1682):
            bias = 0
            for i in range (1683):
                bias += x[k][i] * self.weights[i,j] #predicts outcome of ALL gates using perceptron
            if (bias >= 0):
                bias = 1
            else:
                bias = 0
            if (bias == y[k,j]):
                rightAnswers += 1
            outOf += 1
            print(rightAnswers,outOf,"percentage= ",rightAnswers/outOf) #prints right answers as percentage of all answers
```

Και φυσικά όλα τα παραπάνω τα κάναμε πράξη στην main συνάρτηση της τάξης.

```
if __name__ == '__main__':
    base = np.arange(4 * 80000).reshape(80000, 4)

    # Read all from base file 1
    fileStream = open('u1.base', 'r')
    lines = fileStream.readlines()
    i = 0
    for line in lines:
        base[i] = line.split("\t")
        i += 1
    fileStream.close()
    #convert results to a 943 * 1682 array
    usersMovies = np.zeros((943, 1682), dtype='float32')

    for i in range(80000):
        usersMovies[base[i][0] - 1][base[i][1] - 1] = 1
    '''
    squares = leastSquare(usersMovies, 0)
    squares.train(usersMovies,usersMovies,10,0.1)

    np.save('doneWeights.npy', squares.weights)
    '''
```

```
customedWeights = np.load('doneWeights.npy')

squares = leastSquare(customedWeights, 1)

test = np.arange(4 * 20000).reshape(20000, 4)

# Read all from test file 1
fileStream = open('u1.test', 'r')
lines = fileStream.readlines()
i = 0
for line in lines:
    test[i] = line.split("\t")
    i += 1
fileStream.close()
#convert it to an array of shape (943, 1682)
tests = np.zeros((943, 1682), dtype='float32')

for i in range(20000):
    tests[test[i][0] - 1][test[i][1] - 1] = 1

rightAnswers = squares.test(tests,tests)
```

ΠΑΡΑΔΕΙΓΜΑ ΕΚΤΕΛΕΣΗΣ ΚΩΔΙΚΑ

```
29061
```

Νευρωνικό δίκτυο

```
1 1 percentage=  1.0
1 2 percentage=  0.5
2 3 percentage=  0.6666666666666666
2 4 percentage=  0.5
3 5 percentage=  0.6
3 6 percentage=  0.5
4 7 percentage=  0.5714285714285714
5 8 percentage=  0.625
6 9 percentage=  0.6666666666666666
6 10 percentage=  0.6
6 11 percentage=  0.5454545454545454
7 12 percentage=  0.5833333333333334
8 13 percentage=  0.6153846153846154
8 14 percentage=  0.5714285714285714
9 15 percentage=  0.6
10 16 percentage=  0.625
10 17 percentage=  0.5882352941176471
11 18 percentage=  0.6111111111111112
12 19 percentage=  0.631578947368421
```

```
1586 1984 percentage=  0.7993951612903226
1586 1985 percentage=  0.798992443324937
1587 1986 percentage=  0.7990936555891238
1588 1987 percentage=  0.7991947659788626
1589 1988 percentage=  0.7992957746478874
1589 1989 percentage=  0.7988939165409754
1589 1990 percentage=  0.7984924623115578
1589 1991 percentage=  0.7980914113510799
1589 1992 percentage=  0.7976907630522089
1590 1993 percentage=  0.7977922729553437
1590 1994 percentage=  0.7973921765295887
1590 1995 percentage=  0.7969924812030075
1590 1996 percentage=  0.7965931863727455
1590 1997 percentage=  0.7961942914371557
1590 1998 percentage=  0.7957957957957958
1591 1999 percentage=  0.7958979489744873
1591 2000 percentage=  0.7955
1592 2001 percentage=  0.7956021989005497
```

Ταξινομητής Ελαχίστων Τετραγώνων