Εργασία στην Ανάκτηση Πληροφορίας

Ονοματεπώνυμο: Εμμανουήλ Στούμπος
ΑΕΜ: 2591

Ονοματεπώνυμο: Σωκράτης Αθανασιάδης
ΑΕΜ: 2547

Η εργασία προγραμματίστηκε σε Python. Κάθε ένα από τα τρία προβλήματα έχει το δικό του .py αρχείο. Τα επιπλέον .py αρχεία είναι απαραίτητα για την λύση των προβλημάτων και εξηγούνται παρακάτω.

To Module DataManipulation

Το Module αυτό χρησιμοποιείται για την ανάγνωση των δεδομένων, την επεξεργασία τους, καθώς και για οποιαδήποτε λειτουργία σχετική με την διαχείριση τους. Γενικότερα, όπως θα φανεί και παρακάτω, ο τρόπος με τον οποίο αποφασίσαμε να αναπαραστήσουμε τα δεδομένα είναι με την μορφή διανυσμάτων των πέντε διαστάσεων, όπου η κάθε διάσταση παρέχει μια ξεχωριστή πληροφορία για τα δεδομένα. Η αρχική μας σκέψη ήταν να αναπαραστήσουμε όλα τα δεδομένα στον διανυσματικό χώρο των όρων, μα υπήρχαν προβλήματα περιορισμένης μνήμης με αυτή την μέθοδο. Στην συνέχεια εξηγούνται οι συναρτήσεις του module αναλυτικά.

extractData:

```
def extractData():

    ''' Read Data from .CSV File '''
    print("\nReading data from .csv file...")
    with open('train_original.csv', 'r', encoding='utf-8') as file:
        reader = csv.reader(file, delimiter=',', quotechar='"')
        Data = [row for row in reader]

    print("Total Entries Read: %d"%(len(Data) - 1))
    return Data[1:]  # Don't need the first row
```

Η συνάρτηση αυτή χρησιμοποιείται μονάχα για την ανάγνωση των δεδομένων από το αρχείο train_original.csv.

dataToVectors:

```
def dataToVectors(Data):

    Vectors = []
    InvertedIndex = readInvertedIndex()

    Terms = [term for term in InvertedIndex]

    for data in Data:

        blob1, blob2 = TextBlob(data[3]).lower(), TextBlob(data[4]).lower()

        d1 = queryLength(blob1, blob2)
        d2 = cosineSimilarity(blob1, blob2, InvertedIndex, Terms)
        d3, d4, d5 = compareTags(blob1, blob2)
        Vectors.append((np.array((d1, d2, d3, d4, d5)), int(data[5])))

    return Vectors
```

Η συνάρτηση dataToVectors χρησιμοποιείται για να μετατρέψει τα δεδομένα στην διανυσματική μορφή τους η οποία χαρακτηρίζεται από πέντε διαστάσεις d1, d2, d3, d4 και d5, οι οποίες θα εξηγηθούν παρακάτω. Οι μεταβλητές InvertedIndex και Terms περιέχουν τον αντεστραμμένο κατάλογο, και όλους τους όρους που περιέχονται στα queries αντίστοιχα, και χρησιμοποιούνται για τον υπολογισμό της τιμής d2, η οποία αντιπροσωπεύει το cosine similarity.

writeInvertedIndex:

```
def writeInvertedIndex(Data):

    Queries, InvertedIndex = {}, {}

    for data in Data:

        for i in range(1, 3):
            qid, query = data[i], data[i+2]
            if qid not in Queries:
                Queries.update({qid:query})

    for qid in Queries:

        words = TextBlob(Queries[qid]).lower().words

        for w in words:
            if w not in InvertedIndex:
                InvertedIndex.update({w:[qid]})
            else:
                InvertedIndex[w].append(qid)


    with open('InvertedIndex.txt', 'w', encoding='utf-8') as textFile:
        for word in InvertedIndex:
            line = ','.join(str(x) for x in InvertedIndex[word])
            textFile.write("%s::%s"%(word, line+"\n"))
```

Η συνάρτηση αυτή καλείτε με σκοπό την δημιουργία του Αντεστραμμένου Καταλόγου και την αποθήκευσή του στον δίσκο (καθώς η δημιουργία του είναι χρονοβόρα διαδικασία). Κάθε στοιχείο data της λίστας Data είναι ένα tuple έξι θέσεων της μορφής (id, qid1, qid2, q1, q2, label). Η συνάρτηση αρχικά σαρώνει τα Data για να βρει κάθε ένα από τα queries, τα οποία αποθηκεύει σε ένα Dictionary Queries έτσι ώστε να ισχύει ότι Queries[qid#] = q#. Στην συνέχεια, σαρώνεται κάθε query, το

οποίο χωρίζεται σε λέξεις, έτσι ώστε κάθε λέξη να προστίθεται στον Αντεστραμμένο Κατάλογο ή να ενημερώνεται σε περίπτωση που υπάρχει ήδη. Τέλος, ο Αντεστραμμένος Κατάλογος αποθηκεύεται στον δίσκο ως το αρχείο InvertedIndex.txt.

readInvertedIndex:

```
def readInvertedIndex():

    InvertedIndex = {}

    with open('InvertedIndex.txt', 'r', encoding='utf-8') as file:
        lines = file.readlines()
        for line in lines:
            word, queries = line.split("::")
            Q = [int(n) for n in queries.split(",")]
            InvertedIndex.update({word:Q})

    return InvertedIndex
```

Η συνάρτηση διαβάζει τον Αντεστραμμένο Κατάλογο από τον δίσκο, και τον επιστρέφει ως ένα Dictionary της μορφής:

InvertedIndex[‘word’] = [qid1, qid2, ..., qidn]

queryLength:

```
def queryLength(blob1, blob2) :

    c1, c2 = len(blob1.tokens), len(blob2.tokens)
    return 1 - abs(c1 - c2)/max(c1, c2)
```

Η συνάρτηση αυτή υπολογίζει το d1, δηλαδή την πρώτη διάσταση των εγγραφών, εκφράζοντας πληροφορία σχετική με το μήκος των queries. Πιο συγκεκριμένα, στα c1 και c2 αποθηκεύεται το πλήθος των tokens των q1 και q2 αντίστοιχα, και στην συνέχεια χρησιμοποιούμε τα c1, c2 για να υπολογίσουμε το d1 όπως φαίνεται στην παραπάνω εικόνα. Με αυτόν τον τρόπο υπολογισμού, έχουμε d1 = 1 εάν τα q1, q2 έχουν ακριβώς το ίδιο

πλήθος λέξεων, διαφορετικά το d1 τείνει προς το 0 όσο περισσότερο αυξάνεται η διαφορά ανάμεσα στα c1 και c2.

cosineSimilarity:

```
def cosineSimilarity(blob1, blob2, InvertedIndex, Terms):

    N = 537933 # Total Number of Queries

    Blobs = [blob1, blob2]

    Vectors = []

    for i in range(2):

        v = np.zeros(len(InvertedIndex)) # Initialize Vector

        Words = Blobs[i].words # Dictionary word -> frequency

        try:
            maxf = max(Words.count(w) for w in Words) # Max Frequency of a term in the query
        except:
            return 0.5 # Corrupted Data

        for w in Words:

            f, n = Words.count(w)/maxf, len(InvertedIndex[w]) # f(t), n(t)

            idf = math.log(N/n)/math.log(N) # IDF(t)

            v[Terms.index(w)] = f*idf

        Vectors.append(v)

    return (Vectors[0] @ Vectors[1])/(np.linalg.norm(Vectors[0])*np.linalg.norm(Vectors[1]))
```

Η δεύτερη διάσταση των εγγραφών d2, ορίζεται ως το cosine similarity μεταξύ των δύο queries στον διανυσματικό χώρο των όρων. Η συνάρτηση εκτελεί δύο επαναλήψεις, μία για κάθε query, έτσι ώστε να τα μετατρέψει στην διανυσματική τους μορφή, όπου κάθε διάσταση αντιπροσωπεύει και έναν όρο t του συνόλου όρων του Αντεστραμμένου Καταλόγου. Αρχικά δημιουργείται ένα μηδενικό διάνυσμα (για κάθε query q), και στην συνέχεια υπολογίζουμε τις τιμές των θέσεων που αντιστοιχούν στους όρους tq που περιέχει το κάθε query q, ως το γινόμενο των κανονικοποιημένων τιμών f(tq) και idf(tq). Τέλος επιστρέφεται το συνημίτονο της γωνίας των δύο διανυσμάτων υπολογιζόμενο ως το εσωτερικό γινόμενό τους διά το γινόμενο των μέτρων τους.

compareTags:

```
def compareTags(blob1, blob2):

    Tags, NN, VB, JJ = [blob1.tags, blob2.tags], [[], []], [[], []], [[], []]

    for i in range(2):
        for t in Tags[i]:
            if "NN" in t[1] and t[0] not in NN[i]:
                NN[i].append(t[0])
        NN[i] = set(NN[i])

    try:
        d3 = len(NN[0].intersection(NN[1]))/len(NN[0].union(NN[1]))
    except:
        d3 = 0.5
```

Μέσω της συνάρτησης αυτής υπολογίζονται οι τρεις τελευταίες διαστάσεις d3, d4 και d5. Κάθε μία από τις τρεις διαστάσεις αντιστοιχεί και σε διαφορετικό τύπο όρου:

d3 --> Ουσιαστικά
d4 --> Ρήματα
d5 --> Επίθετα

Στην παραπάνω εικόνα φαίνεται μονάχα ο υπολογισμός της τιμής d3, καθώς οι υπολογισμοί των d4 και d5 γίνονται με τον ίδιο ακριβώς τρόπο. Αρχικά σπάμε τα queries σε ζευγάρια όρος - tag. Στην συνέχεια κρατάμε μόνο τους όρους για τους οποίους το tag υποδεικνύει ότι ο όρος είναι ουσιαστικό (για το d4 ρήμα και για το d5 επίθετο). Βάσει αυτών των όρων μονάχα κατασκευάζουμε δύο σύνολα, ένα για το κάθε query, και τέλος υπολογίζουμε το d3 ως την ομοιότητα Jaccard μεταξύ αυτών των συνόλων. Σε περίπτωση που η ένωση των συνόλων είναι το κενό σύνολο, το d3 ορίζεται ως 0.5. Στις επόμενες δύο περιπτώσεις όπου ίσως προκύψει το κενό σύνολο στον παρανομαστή, ορίζουμε το d4 ίσο με το d3 και το d5 ως (d3 + d4)/2, διότι παρατηρήσαμε ότι παίρνουμε καλύτερες ακρίβειες με αυτόν τον τρόπο αντί εάν τα θέταμε όλα ως 0.5.

Αφού έχουν υπολογισθεί και οι πέντε διαστάσεις, η συνάρτηση dataToVectors αποθηκεύει κάθε εγγραφή στην λίστα Vectors ως ένα tuple δύο θέσεων όπου την πρώτη θέση κατέχει το διάνυσμα [d1 d2 d3 d4 d5] ενώ την δεύτερη θέση, το label της εγγραφής.

Αντιμετώπιση του Πρώτου Προβλήματος

Για το πρώτο πρόβλημα καλούμαστε να κατηγοριοποιήσουμε διάφορα ζεύγη queries αναλόγως εάν εκφράζουν σημασιολογικά την ίδια έννοια. Θα πρέπει δηλαδή να κατασκευάσουμε έναν κατηγοριοποιητή που κάνει αυτή ακριβώς την δουλειά.

kNearestNeighbours:

```
def kNearestNeighbors(TrainingSet, TestingSet, k):

    startTime = time.time()

    loops, Correct = 0, 0

    for test in TestingSet:

        Classes = np.zeros(2) # Possible Classes |Even or Odd| = 2
        Distances = []
        for train in TrainingSet:
            Distances.append((np.linalg.norm(train[0] - test[0]), train[1])) # (distance, label)
        Distances.sort(key = lambda tup: tup[0]) # sort the tuples by the distance
        for d in Distances[0:k]: # for k nearest neighbours
            Classes[d[1]] += 1 # find which class most neighbours belong in
        prediction = np.argmax(Classes) # get the label of the class with most k neighbours in it
        if prediction == test[1]: # if label of test is equal to label of class
            Correct += 1
        loops += 1

        if loops == 30:
            runtimeEstimation(time.time() - startTime, len(TestingSet), loops)

    totalTime = time.time() - startTime

    if totalTime >= pow(60,2):
        print("\nk Nearest Neighbors Algorithm completed in %d hours and %d minutes"%(totalTime/pow(60,2), (totalTime/60)%60))
    else:
        print("\nk Nearest Neighbors Algorithm completed in %d minutes and %d seconds"%(totalTime/60, totalTime%60))

    testSize = len(TestingSet)
    print("\nSuccessful Predictions Percentage: %.2f%% of the samples"%(Correct*100/testSize))
    print("Unsucessful Predictions Percentage: %.2f%% of the samples"%((testSize - Correct)*100/testSize))
```

Αρχικά κατασκευάσαμε τον κατηγοριοποιητή των k Κοντινότερων Γειτόνων, και χωρίσαμε τα δεδομένα σε Training και Testing σύνολα, όπου το Training περιέχει τις πρώτες 15,000 εγγραφές, ενώ το Testing περιέχει 5,000 εγγραφές (από την 15,001η έως και την 20,000η).

Παρατήρηση: Δεν πήραμε ολόκληρο το σύνολο εγγραφών καθώς η εκπαίδευση ενός κατηγοριοποιητή ακόμα και για 100,000 δεδομένα (λιγότερο από το ένα τέταρτο των συνολικών δεδομένων) είναι μία άκρως χρονοβόρα διαδικασία.

Στην συνέχεια παρουσιάζονται τα αποτελέσματα που πήραμε τρέχοντας τον αλγόριθμο των k Κοντινότερων Γειτόνων για k ίσο με 50:

```
Reading data from .csv file...
Total Entries Read: 404290

Converting 20000 Data samples to vectors...
Data to Vector Convertion completed.

Runime Estimation: 7 minutes and 46 seconds

k Nearest Neighbors Algorithm completed in 7 minutes and 23
seconds

Successful Predictions Percentage: 69.38% of the samples
Unsucessful Predictions Percentage: 30.62% of the samples
```

Παρατήρηση: Η ακρίβεια υπολογίζεται ως το πλήθος των επιτυχώς κατηγοριοποιημένων δειγμάτων διαιρούμενο με το συνολικό πλήθος των δειγμάτων. Χρησιμοποιήσαμε αυτή την τεχνική έτσι ώστε να υπάρχει κοινό μέτρο σύγκρισης με τον επόμενο κατηγοριοποιητή ο οποίος ήταν ήδη υλοποιημένος.

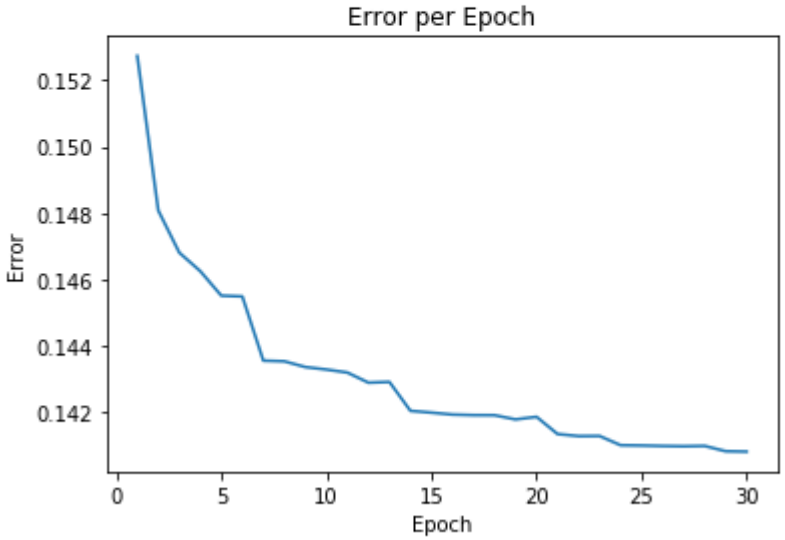
Όπως φαίνεται και από την εικόνα παραπάνω, το 69.38% από τα 5,000 δείγματα κατηγοριοποιήθηκε επιτυχώς, ενώ το 30.62% λανθασμένα. Δηλαδή για κάθε 10 δείγματα, τα 7 περίπου θα κατηγοριοποιούνται σωστά. Εάν σκεφτούμε ότι το πρόβλημα

είναι δυαδικό, που σημαίνει ότι έχουμε τυχαία πιθανότητα επιτυχούς κατηγοριοποίησης 50%, το 70% είναι σίγουρα μία βελτίωση.

Στην συνέχεια χρησιμοποιήσαμε ένα RBF Νευρωνικό Δίκτυο (κατασκευάστηκε ως εργασία για το μάθημα στα Νευρωνικά Δίκτυα), το οποίο εκτελεί K-Means για να βρει τα κέντρα του Κρυφού Επιπέδου. Χρησιμοποιήσαμε και εδώ 20,000 δείγματα (15,000 για Training και 5,000 για Testing), ορίζοντας k = 500 κέντρα, και μεταβαλλόμενο ρυθμό μάθησης αναλόγως την πορεία του δικτύου:

```
*************** Training Phase ***************

Training RBF Network for 15000 Data Samples...

Training Hidden Layer via K-Means for k = 500.

K-Means Iteration #1: Centers Mean Difference = 0.057
K-Means Iteration #2: Centers Mean Difference = 0.027
K-Means Iteration #3: Centers Mean Difference = 0.018
K-Means Iteration #4: Centers Mean Difference = 0.013
K-Means Iteration #5: Centers Mean Difference = 0.010
K-Means Iteration #6: Centers Mean Difference = 0.007
K-Means Iteration #7: Centers Mean Difference = 0.006
K-Means Iteration #8: Centers Mean Difference = 0.005
K-Means Iteration #9: Centers Mean Difference = 0.004
K-Means Iteration #10: Centers Mean Difference = 0.003
K-Means Iteration #11: Centers Mean Difference = 0.002
K-Means Iteration #12: Centers Mean Difference = 0.002
K-Means Iteration #13: Centers Mean Difference = 0.001
K-Means Iteration #14: Centers Mean Difference = 0.001
K-Means Iteration #15: Centers Mean Difference = 0.001

Saving Data...

500/500 Centers are being used.
Average Center Radius (sigma): 0.19

Hidden Layer Training completed in 11 minutes and 28 seconds.
```

```
Training Output Layer via Adaline for 30 epochs...

Epoch #1 completed: Error = 0.152739, Accuracy = 69.17%
Epoch #2 completed: Error = 0.148076, Accuracy = 70.51%
Epoch #3 completed: Error = 0.146804, Accuracy = 70.64%
Epoch #4 completed: Error = 0.146245, Accuracy = 70.87%
Epoch #5 completed: Error = 0.145505, Accuracy = 71.09%
Epoch #6 completed: Error = 0.145483, Accuracy = 71.04%
Learning rate b changing to 0.2500.
Epoch #7 completed: Error = 0.143545, Accuracy = 71.47%
Epoch #8 completed: Error = 0.143525, Accuracy = 71.63%
Epoch #9 completed: Error = 0.143352, Accuracy = 71.55%
Epoch #10 completed: Error = 0.143277, Accuracy = 71.65%
Epoch #11 completed: Error = 0.143186, Accuracy = 71.36%
Epoch #12 completed: Error = 0.142881, Accuracy = 71.48%
Epoch #13 completed: Error = 0.142903, Accuracy = 71.65%
Learning rate b changing to 0.1250.
Epoch #14 completed: Error = 0.142031, Accuracy = 72.04%
Epoch #15 completed: Error = 0.141977, Accuracy = 71.96%
Epoch #16 completed: Error = 0.141918, Accuracy = 71.96%
Epoch #17 completed: Error = 0.141901, Accuracy = 71.93%
Epoch #18 completed: Error = 0.141899, Accuracy = 71.95%
Epoch #19 completed: Error = 0.141772, Accuracy = 71.97%
Epoch #20 completed: Error = 0.141850, Accuracy = 71.95%
Learning rate b changing to 0.0625.
Epoch #21 completed: Error = 0.141331, Accuracy = 72.30%
Epoch #22 completed: Error = 0.141269, Accuracy = 72.20%
Epoch #23 completed: Error = 0.141272, Accuracy = 72.32%
Learning rate b changing to 0.0312.
Epoch #24 completed: Error = 0.140994, Accuracy = 72.18%
Epoch #25 completed: Error = 0.140985, Accuracy = 72.33%
Epoch #26 completed: Error = 0.140971, Accuracy = 72.27%
Epoch #27 completed: Error = 0.140962, Accuracy = 72.23%
Epoch #28 completed: Error = 0.140972, Accuracy = 72.25%
Learning rate b changing to 0.0156.
Epoch #29 completed: Error = 0.140812, Accuracy = 72.19%
Epoch #30 completed: Error = 0.140800, Accuracy = 72.25%
```




```
Training Phase completed in 33 minutes and 28 seconds

*************** Testing Phase ***************

Testing for 5000 samples...

Successful Predictions Percentage: 69.84% of the samples.
Unsucessful Predictions Percentage: 30.16% of the samples.
```

Όπως και φαίνεται από την εικόνα παραπάνω, το RBF Νευρωνικό Δίκτυο τα πήγε καλύτερα από το k Nearest Neighbours, βελτιώνοντας την ακρίβεια όμως μονάχα για λιγότερο από 1%.

Παρατηρήσεις & Συμπεράσματα:

Εξετάζοντας τα δεδομένα, υπήρχαν ορισμένα mislabeled δείγματα που σίγουρα θα επηρέαζαν την ακρίβεια αρνητικά. Ένα ακόμα μεγάλο πρόβλημα ήταν ορισμένες εγγραφές, των οποίων τα queries ήταν ακριβώς όμοια εκτός από μονάχα μία λέξη, η οποία άλλαζε τελείως το νόημα του κάθε query. Αποτέλεσμα ήταν, το δείγμα αυτό να πάρει σχετικά υψηλές τιμές (π.χ. υψηλό cosine similarity) ενώ ανήκει στην κλάση ‘0’. Γενικότερα προσπαθήσαμε να επινοήσουμε μία έξτρα διάσταση για να αντιμετωπίσουμε αυτό το πρόβλημα, μα πολλές φορές επηρεάζονταν και δείγματα τα οποία άνηκαν στην κλάση ‘1’ με αποτέλεσμα η ακρίβεια να παραμείνει η ίδια ή και να μειωθεί. Για αυτό τον λόγο αποφασίσαμε εν τέλει να μην την συμπεριλάβουμε στην τελική εργασία.

Σίγουρα μία ακρίβεια 70%, αν και αποτελεί πρόοδο ως προς την τυχαιότητα, δεν προσφέρει πραγματική βοήθεια σε προβλήματα πραγματικού κόσμου. Παρόλα αυτά προσθέτοντας περισσότερες διαστάσεις οι οποίες εκφράζουν διαφορετικά είδη πληροφορίας των δειγμάτων, δοκιμάζοντας νέους κατηγοριοποιητές με διαφορετικές παραμέτρους, και διαθέτοντας ισχυρότερους υπολογιστικούς πόρους, ο καθένας μπορεί σίγουρα να αγγίξει ακόμα καλύτερες ακρίβειες.

Αντιμετώπιση του Δεύτερου Προβλήματος

Όσο αφορά το δεύτερο πρόβλημα, χρησιμοποιήσαμε την ίδια ακριβώς αναπαράσταση των δεδομένων που χρησιμοποιήσαμε και στο πρώτο. Δηλαδή κάθε εγγραφή είναι ένα διάνυσμα στις πέντε διαστάσεις. Γενικά, ο τρόπος με τον οποίο δημιουργήσαμε τις διαστάσεις, είναι έτσι ώστε κάθε διάσταση να πλησιάζει την τιμή 1 αν τα queries μίας εγγραφής είναι σημασιολογικά ίδια, αλλιώς να πλησιάζει την τιμή 0. Βάσει αυτού του σκεπτικού μπορούμε να υπολογίσουμε την πιθανότητα μίας εγγραφής να ανήκει στην κλάση ‘1’ ως το μέτρο του διανύσματος της, διαιρούμενο με την ρίζα του 5, η οποία αντιστοιχεί στην μέγιστη τιμή του μέτρου που μπορεί να πάρει ένα διάνυσμα, εάν κάθε διάστασή του έχει την τιμή 1. Με αυτόν το τρόπο, διανύσματα εγγραφών των οποίων το μέτρο προσεγγίζει την τιμή ρίζα του 5, θα έχουν μεγάλη πιθανότητα να ανήκουν στην κλάση 1, δηλαδή το ζεύγος των queries τους, να είναι σημασιολογικά όμοιο. Παρακάτω εξηγούνται οι συναρτήσεις που χρησιμοποιούνται:

predict:

```
def predict(Vectors):

    accuracy, d = 0, math.sqrt(5)

    for v in Vectors:

        p = np.linalg.norm(v[0])/d
        accuracy += CrossEntropy(p, v[1])

    print("\nAccuracy: %.2f%%"%(accuracy*100/len(Vectors)))
```

Η συνάρτηση αυτή δέχεται το σύνολο των διανυσμάτων και υπολογίζει την πιθανότητα p κάθε διανύσματος να ανήκει στην κλάση ‘1’ με τον τρόπο που εξηγήθηκε παραπάνω. Στην συνέχεια

υπολογίζει την συνολική ακρίβεια, μέσω της συνάρτησης CrossEntropy, η οποία εξηγείται παρακάτω.

CrossEntropy:

```
def CrossEntropy(p, y):

    if p == 1:
        p = 0.999999

    return -(y*math.log(p) + (1-y)*math.log(1-p))
```

Η συνάρτηση αυτή δέχεται το label (κλάση) y του διανύσματος καθώς και την πιθανότητα p του διανύσματος να ανήκει στην κλάση 1, και υπολογίζει την ακρίβεια της πρόβλεψης για το συγκεκριμένο διάνυσμα μέσω της τεχνικής Cross Entropy Binary Loss.

Παρατήρηση: Αρχικά γίνεται ένας έλεγχος σε περίπτωση που η πιθανότητα p ισούται με 1, όπου σε αυτή την περίπτωση θέτουμε το p ως μία τιμή πολύ κοντά στο 1, έτσι ώστε το log(1-p) να ορίζεται καθώς για logx πρέπει x > 0.

Στην συνέχεια υπολογίζουμε την ακρίβεια, χρησιμοποιώντας ολόκληρο το σύνολο δεδομένων, καθώς αυτή τη φορά δεν υπάρχουν χωρικοί ή χρονικοί περιορισμοί:

```
Reading data from .csv file...
Total Entries Read: 404290
404290

Converting 404290 Data samples to vectors...

Data to Vector Convertion completed.
Accuracy: 69.50%
```

Όπως και φαίνεται από την εικόνα παραπάνω, η ακρίβεια υπολογίστηκε ως 69.50%, αριθμός ο οποίος βρίσκεται πολύ κοντά στις ακρίβειες που λάβαμε στο προηγούμενο ερώτημα. Βέβαια δεν μπορούμε να συγκρίνουμε απόλυτα τις ακρίβειες αυτές

μεταξύ τους καθώς στο πρώτο ερώτημα αφενός χρησιμοποιήσαμε μονάχα ένα υποσύνολο των δεδομένων, αφετέρου η ακρίβεια μετρήθηκε με διαφορετικό τρόπο.

Αντιμετώπιση του Τρίτου Προβλήματος

Για το τρίτο και τελευταίο πρόβλημα καλούμαστε να κατασκευάσουμε μία δομή για τα queries, έτσι ώστε να μπορούμε να προσδιορίσουμε για κάποιο τυχαίο query, παρόμοια queries σε γρήγορο χρόνο. Η τεχνική που ακολουθήσαμε βασίζεται στο Hashing των queries, και συγκεκριμένα είναι η SimHash. Οι συναρτήσεις που σχετίζονται με την ανάκτηση των hashes, παρουσιάζονται παρακάτω.

createQueriesDictionary:

```
def createQueriesDictionary(Data):

    InvertedIndex, Queries = dm.readInvertedIndex(), {}

    N = 537933 # Total Number of Queries

    for data in Data:

        for i in range(1, 3):
            qid, query = data[i], [data[i+2]]
            if qid not in Queries:
                Queries.update({qid:query})

    for qid in Queries:

        Words = TextBlob(Queries[qid][0]).lower().words # Dictionary word -> frequency

        Hashes, Weights = [], []

        try:
            maxf = max(Words.count(w) for w in Words) # Max Frequency of a term in the query
        except:
            continue # Corrupted Data

        for w in Words:

            Hashes.append(hashFunction(w, 64))

            f, n = Words.count(w)/maxf, len(InvertedIndex[w]) # f(t), n(t)

            idf = math.log(N/n)/math.log(N) # IDF(t)

            Weights.append(f*idf)


        queryHash = HashQuery(Hashes, Weights)
        Queries[qid].append(queryHash)
```

Σκοπός αυτής της συνάρτησης είναι να κατασκευάσουμε ένα Dictionary το οποίο για κάθε qid έχει αποθηκευμένο ένα πίνακα δύο θέσεων. Την πρώτη θέση κατέχει το ίδιο το query, ενώ την δεύτερη θέση το αντίστοιχο hash:

Queries[qid] --> [query, queryHash]

Αρχικά σαρώνουμε τα Data έτσι ώστε να αντλήσουμε όλα τα διαφορετικά queries και να φτιάξουμε το Dictionary σε αυτή την μορφή:

Queries[qid] --> [query]

Στην συνέχεια σαρώνουμε όλα τα queries, για να υπολογίσουμε τα αντίστοιχα hashes. Για κάθε query εκτελούμε την παρακάτω διαδικασία. Αρχικά σπάμε το query σε λέξεις τις οποίες αποθηκεύουμε στην λίστα Words. Έπειτα αρχικοποιούμε τις λίστες Hashes, Weights, στις οποίες θα αποθηκευτούν τα hashes των λέξεων και τα βάρη τους αντίστοιχα. Για κάθε λέξη του query καλούμε την συνάρτηση **hashFunction** για να πάρουμε το hash, και το αποθηκεύουμε στην λίστα Hashes.

hashFunction:

```
def hashFunction(word, n):

    h = (bin(hash(word))[2:])[1:]

    for i in range(len(h), n):
        h = "0" + h # Fill in the zeros

    return h
```

Η συνάρτηση αυτή χρησιμοποιεί την έτοιμη συνάρτηση hash της Python για να παράγει ένα hash μήκους n bits (64 bits στο παράδειγμά μας).

Στην συνέχεια, αφού έχουμε κατασκευάσει τον Αντεστραμμένο Κατάλογο σε προηγούμενο ερώτημα, υπολογίζουμε το βάρος της κάθε λέξης με τον γνωστό τύπο TF*IDF, και το αποθηκεύουμε στον πίνακα Weights.

Εφόσον έχουμε υπολογίσει τα hashes αλλά και τα βάρη όλων των λέξεων του query, καλούμε την συνάρτηση **HashQuery** για να υπολογίσουμε το hash του query.

HashQuery:

```
def HashQuery(Hashes, Weights):

    QueryHash = [0 for i in range(len(Hashes[0]))]

    i = 0
    while i < len(Hashes):
        weight = Weights[i]
        j = 0
        while j < len(Hashes[i]):
            if Hashes[i][j] == "1":
                QueryHash[j] += weight
            else:
                QueryHash[j] -= weight
            j += 1
        i += 1


    QueryHashString = ""

    for h in QueryHash:
        if h >= 0:
            QueryHashString += "1"
        else:
            QueryHashString += "0"

    return QueryHashString
```

Αρχικά δημιουργούμε την λίστα QueryHash η οποία έχει μέγεθος όσα και τα bits των query των λέξεων (δηλαδή 64). Στην συνέχεια σαρώνουμε όλα τα hashes των λέξεων του query και κάθε φορά αποθηκεύουμε το βάρος της αντίστοιχης λέξης στην μεταβλητή weight. Για κάθε j-οστό bit του hash μίας λέξης, αν αυτό είναι 1, τότε προσθέτουμε το weight στην j-οστή θέση της λίστας QueryHash, αλλιώς το αφαιρούμε από αυτή. Αφού τελειώσει αυτή η διαδικασία, κατασκευάζουμε το τελικό hash του query, αντιστοιχώντας τις θετικές τιμές του QueryHash σε ‘1’, ενώ τις αρνητικές σε 0.

Εφόσον έχει υπολογιστεί το hash του query, το προσθέτουμε στον αντίστοιχο πίνακα και συνεχίζουμε την ίδια διαδικασία για το επόμενο query. Τέλος αποθηκεύουμε το ολοκληρωμένο Dictionary Queries ως το αρχείο Queries.txt το οποίο μπορεί να ανακτηθεί μέσω της συνάρτησης **loadQueriesDictionary()**.

Αποφασίσαμε να δημιουργήσουμε μία δομή η οποία βασίζεται στο Local Sensitive Hashing. Συγκεκριμένα χωρίσαμε κάθε 64-bit hash, που πήραμε από την προηγούμενη διαδικασία, σε δύο bands των 32-bit. Ύστερα κάνουμε hash κάθε band των query hashes σε κάποιο Bucket. Αυτό σημαίνει ότι κάθε query θα ανήκει σε δύο Buckets, ένα για το 1ο band και ένα για το 2ο. Ουσιαστικά έχουμε δημιουργήσει μία λίστα HashBands δύο θέσεων, όπου κάθε θέση αντιστοιχεί σε ένα Dictionary (HashTable) του οποίου κάθε θέση αντιστοιχεί σε ένα Bucket στην μορφή λίστας με αποθηκευμένα τα qid των queries που είχαν collision στο συγκεκριμένο Bucket. Κάθε Bucket χαρακτηρίζεται από το 16-bit κλειδί του. Π.χ. παρακάτω κάνουμε access στο Bucket με κλειδί “10...11” του 1ου Band.

HashBands[0][“10...11”] -> [qid1, qid4, ...]

Η συνάρτηση που δημιουργεί και αποθηκεύει την δομή HashBands είναι η εξής:

createHashBands:

```
def createHashBands(Queries, b):

    QueryBands = splitToBands(Queries, b) # b is the number of bands

    HashBands = [dict() for i in range(b)]

    for qid in QueryBands:

        i = 0
        while i < len(QueryBands[qid]):

            bandHash = bandHashFunction(QueryBands[qid][i], 16)

            if bandHash in HashBands[i]: # If bucket already exists
                HashBands[i][bandHash].append(qid) # Place qid in backet
            else: # Else create Bucket
                HashBands[i].update({ bandHash: [qid] }) # Containing the qid

            i += 1

    with open('HashBands.txt', 'wb') as file:
        pickle.dump(HashBands, file)
```

Αρχικά καλούμε την συνάρτηση splitToBands με σκοπό να “σπάσουμε” κάθε hash σε b (για μας 2) bands:

splitToBands:

```
def splitToBands(Queries, b):

    r = len(Queries["1"][1])//b # Number of rows in each band

    QueryBands = {}

    for qid in Queries:

        try:
            band, qhash = [], Queries[qid][1]
        except:
            continue

        for i in range(b):
            band.append(qhash[i*r:(i+1)*r])

        QueryBands.update({qid : band})

    return QueryBands
```

Η συνάρτηση επιστρέφει το dictionary QueryBands το οποίο έχει σπάσει κάθε hash σε δύο ίσα μέρη (bands) των 32-bit. Το QueryBands έχει την εξής μορφή:

QueryBands[qid] --> [“10...01”, “11...00”]

Στην συνέχεια σαρώνουμε κάθε query και δημιουργούμε τα Buckets βρίσκοντας το 16-bit κλειδί τους βάσει της συνάρτησης **bandHashFunction**.

bandHashFunction:

```
def bandHashFunction(bitHash, n):

    toNumber = int(bitHash, 2) + 3651

    h = bin(int(toNumber % math.pow(2, n)))[2:]

    for i in range(len(h), n):
        h = "0" + h # Fill in the zeros

    return h
```

Τέλος αποθηκεύουμε την δομή μέσω του module pickle, ως το αρχείο HashBands.txt

Τελευταία συνάρτηση είναι η **findNearDuplicates**, μέσω της οποίας βρίσκουμε τα NearDuplicates του query που επιθυμούμε.

FindNearDuplicates:

```
def findNearDuplicates(qid, Queries, HashBands, threshold):

    b, r = 2, 32
    qhash = Queries[qid][1]

    band = []

    for i in range(b):
        band.append(qhash[i*r:(i+1)*r])

    i = 0

    Duplicates = set()

    while i < len(band):

        bandHash = bandHashFunction(band[i], 16)

        if bandHash in HashBands[i]:
            PossibleDuplicates = Duplicates.union(set(HashBands[i][bandHash]))

        i += 1

    NearDuplicates = []

    for dupQid in PossibleDuplicates:
        d = HammingDistance(Queries[qid][1], Queries[dupQid][1])
        if d < threshold and (qid != dupQid):
            NearDuplicates.append(dupQid)


    return NearDuplicates
```

Η συνάρτηση αυτή δημιουργεί το σύνολο PossibleDuplicates μέσα στο οποίο εμπεριέχονται όλα τα qid των queries με το οποία υπήρχε collision στο ίδιο Bucket (η ένωση των Buckets των δύο Bands), το οποίο αντιπροσωπεύει ένα πολύ μικρό ποσοστό του συνολικού πλήθος των Queries. Στην συνέχεια, για να αυξήσουμε την ακρίβεια του αλγορίθμου, υπολογίζουμε το Hamming Distance του query με τα υπόλοιπα queries στο σύνολο, βάσει των 64-bit hash που λάβαμε κατά το SimHash. Εάν η απόσταση

Hamming από ένα query στο σύνολο, είναι μικρότερη ενός threshold (το οποίο ορίζεται από τον χρήστη) τότε αποθηκεύουμε το query στην λίστα NearDuplicates την οποία στο τέλος επιστρέφουμε.

HammingDistance:

```
def HammingDistance(h1, h2):

    diff, i = 0, 0

    while i < len(h1):
        if h1[i] != h2[i]:
            diff += 1
        i += 1

    return diff/len(h1)
```

Στην συνέχεια παρουσιάζουμε τα Near Duplicates που βρέθηκαν για το query με qid = “1”, καθώς και τον χρόνο εκτέλεσης της διαδικασίας.

Κώδικας:

```
def main():

    start = time.time()

    Queries = loadQueriesDictionary()
    HashBands = loadHashBands()

    qid = "1"

    NearDuplicates = findNearDuplicates(qid, Queries, HashBands, 0.05)

    print("\nFor query:\n\n\t%s"%Queries[qid][0])

    if len(NearDuplicates) > 0:
        print("\nNear Duplicates:\n")
        for qid in NearDuplicates:
            print("\t"+Queries[qid][0])
            print()

    else:
        print("\nNo Near Duplicates found. Try increasing the threshold.")

    print("Execution Time: %.2f seconds"%(time.time() - start))
```

Αποτελέσματα:

```
For query:
        What is the step by step guide to invest in share market in india?
Near Duplicates:
        What is step by step guide to learn English?
        How would I solve this equation step by step?
        What is the step by step process to start online business?
        How do you draw a doctor step by step?
        How do draw flowers step by step?
        How can I learn digital marketing step by step for free?
        How can one learn hacking step by step?
        How do I learn English step by step?
        What is the step by step procedure to start up?
        Where do I start and what is step by step process to become a data scientist?
        What is the step by step guide to invest in share market?
        What does a step up and step down gear box do?
        When families with step kids break up do most step kids keep in contact with the step parent?
        How do I become a full stack web developer step by step?

        How can I make an Android chatting app step by step guideline?
        How do I become a web designer step by step?
        What is the step by step procedure to apply for Australian PR?
        Where do I start and what is step by step process to become a Data Scientist?
        What is the best way to learn Android development step by step?
        How do I make my own website step by step?
        How do I start learning data analytics step by step?
        William hayt circuit analysis step by step/\?
        Is there any book that is centered around the platonic relationship between a step brother and step sister?
        How can I learn Android development step by step?
  Execution Time: 1.22 seconds
```

Όπως φαίνεται παραπάνω, τα queries μοιράζονται πάρα πολλές κοινές λέξεις, που σημαίνει ότι η εύρεση των Near Duplicates ήταν επιτυχής ενώ χρειάστηκε μονάχα 1 δευτερόλεπτο για να εκτελεσθεί.

Τέλος Εργασίας