ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα *- Τελική Αναφορά*

ΥΠΕΥΘΥΝΟΣ ΚΑΘΗΓΗΤΗΣ : Ιωαννίδης Ιωάννης

ΥΠΕΥΘΥΝΟΣ ΕΡΓΑΣΤΗΡΙΟΥ : Πικραμένος Γεώργιος

ΜΕΛΗ ΟΜΑΔΑΣ

Χίου Ρίτα Άννα 1115201700192

Χουσιανίτη Αικατερίνη 1115201700194

Μπούρα Τατιάνα 1115201700100

Github repository : https://github.com/tatiana-boura/K23-Assignment-I

Αθήνα 2021

Περιγραφή

Στην παρούσα αναφορά θα περιγραφούν οι σχεδιαστικές επιλογές στα πρώτα δύο παραδοτέα που οδήγησαν στην τελική μορφή της εργασίας όπως αυτή φαίνεται στο τρίτο. Οι σχεδιαστικές επιλογές θα τεκμηριωθούν με παραδείγματα και αναφορές στον χώρο αλλά και στον χρόνο εκτέλεσης.

Γενικότερες Παρατηρήσεις

1. Σε αρκετές δομές μας φάνηκε χρήσιμη η χρήση λίστας. Συνεπώς, υλοποιήσαμε μια δομή *generic* λίστας, η οποία περιέχει πλήθος διαφορετικών συναρτήσεων για την διαχείρισή της και έτσι η λίστα μπορεί να είναι ταξινομημένη, να λειτουργεί σαν ουρά (με push back και pop συναρτήσεις) ή σαν στοίβα (με push front και pop συναρτήσεις) κλπ. ή ως συνδυασμός των παραπάνω προκειμένου να μπορούμε να ανταποκριθούμε στα ζητούμενα της άσκησης με ιδανικό τρόπο. Συνεπώς, όταν θα αναφερόμαστε από εδώ και στο εξής σε λίστα, θα εννοούμε αυτήν.
2. Οι χρόνοι εκτέλεσης αφορούν το μεγάλο dataset W και όχι το dataset Y.

Αποθήκευση JSON αρχείων

Κάθε αρχείο JSON χαρακτηρίζεται από ένα μοναδικό path το οποίο απεικονίζει τον υποφάκελο στον οποίο βρίσκεται το αρχείο αυτό, δηλαδή το e-shop, και το όνομά του. Συνεπώς, αυτό αποτελεί το κλειδί βάσει του οποίου γίνονται αναζητήσεις και αναφορές στο αρχείο JSON. Ειδικά οι αναζητήσεις είναι βασικό να γίνονται γρήγορα. Έτσι, αποφασίσαμε να αποθηκεύουμε τα JSON αρχεία κωδικοποιημένα σε ένα πίνακα κατακερματισμού (hash table). Η κωδικοποίηση θα αναλυθεί σε δεύτερο χρόνο.

Συγκεκριμένα, ο πίνακας κατακερματισμού περιέχει ως κάθε στοιχείο μία λίστα από buckets, όπου το bucket περιέχει σταθερό πλήθος εγγραφών ώστε να μπορούμε να δημιουργούμε το πλήθος που επαρκεί για την φύλαξη των JSON, αλλά παράλληλα να διατηρούμε ως ένα βαθμό και την ευκολία και ταχύτητα πρόσβασης που δίνουν οι

πίνακες. Με τη χρήση, λοιπόν, πίνακα κατακερματισμού με μία καλή συνάρτηση κατακερματισμού, η οποία κάνει καλή διασπορά και δε στέλνει πολλά δεδομένα σε ίδια bucket, η πολυπλοκότητα χρόνου αναζήτησης των JSON αρχείων γίνεται για *n := πλήθος JSON αρχείων*, *O(logn)* αντί για *O(n)* που θα ήταν με τη χρήση μιας απλής λίστας που ήταν η αρχική μας ιδέα .

Όσον αφορά τη κωδικοποίηση της πληροφορίας των JSON αρχείων, στο πρώτο παραδοτέο (όπου δε γινόταν η χρήση των δεδομένων εντός του JSON), αποθηκεύαμε τα χαρακτηριστικά κάθε προϊόντος σε μία λίστα από tuples, δηλαδή μιάς δομής ζευγών (propertyName, λίστα από τιμές propertyValue ή propertyValueList>). Η propertyValueList χρησίμευε στην αποθήκευση των δεδομένων propertyValue όταν το JSON αρχείο περιείχε πίνακα και ήταν μία λίστα από strings.

Στο δεύτερο παραδοτέο, όμως, εισήχθη η έννοια του vocabulary και έτσι η παραπάνω αναπαράσταση δεν ήταν βολική, αφού μας ενδιέφεραν και άλλα χαρακτηριστικά των δεδομένων των JSON αρχείων, όπως πόσες φορές εμφανίζεται η λέξη αυτή στο συγκεκριμένο JSON αλλά και στο σύνολο των JSON. Έτσι, υλοποιήσαμε την δομή (struct) wordInfo που αποτελείται από μία λέξη και τη συχνότητα εμφάνισής της. Χρησιμοποιήσαμε τη δομή αυτή προκειμένου να αποθηκεύουμε την κάθε λέξη του κάθε JSON και τη πληθικότητα εμφάνισής της στο αρχείο σε μια λίστα. Η λίστα αυτή αποθηκεύεται στην εγγραφή του πίνακα κατακερματισμού που αντιστοιχεί στο αρχείο και θα αναφερόμαστε σε αυτήν ως local vocabulary. Επίσης, χρησιμοποιήσαμε μία global λίστα από wordInfo η οποία είχε την έννοια του vocabulary και διατηρούσε μετρητή για το πλήθος των αρχείων στα εμφανίζεται η κάθε λέξη του vocabulary.

Επειδή η δομή vocabulary άλλαξε αρκετές μορφές για να είναι εύχρηστη και να δημιουργεί αποδοτικά τα διανύσματα BoW ή/και tfidf του κάθε αρχείου, θα την αναλύσουμε ευθύς αμέσως σε συνδυασμό με τα προαναφερθέντα διανύσματα.

Vocabulary global - local / tfidf - BoW vectors

Μια λειτουργία που ήταν απαραίτητη για την ορθή χρήση του προγράμματος και εν προκειμένη περιπτώσει για την ορθή λειτουργία του global και local vocabulary ήταν η εξασφάλιση ότι κάθε λίστα δεν περιείχε διπλότυπα. Για να γίνει αυτό κάναμε τις εξής αλλαγές στο αρχικό πρόγραμμα που χρησιμοποιούσε απλές αταξινόμητες λίστες από wordInfo:

- Για την global λίστα δημιουργήσαμε έναν πίνακα hash για εύκολη αναζήτηση (*O(logn)* με την καλή συνάρτηση κατακερματισμού), στον οποίο κάθε εγγραφή έδειχνε στο vocabulary στην αντίστοιχη λέξη. Καθ' αυτόν τον τρόπο ο έλεγχος των διπλοτύπων ήταν εύκολος. Επιπλέον, επειδή λόγω του γεγονότος ότι κάνουμε αναζήτηση μέσω του hash δεν μας απασχολούσε η σειρά αποθήκευσης των λέξεων στην global λίστα η εισαγωγή γινόταν από μπροστά με πολυπλοκότητα *O(1)*.
- Η τοπική λίστα του κάθε αρχείου εξασφαλίσαμε να είναι ταξινομημένη, ώστε η αναζήτηση για διπλότυπα να είναι κατά μέσο όρο *O(n/2)*. Στην συγκεκριμένη περίπτωση δεν μπορούσαμε να χρησιμοποιήσουμε hash για κάθε αρχείο διότι το τίμημα που θα πληρώναμε για γρήγορη αναζήτηση θα ήταν πολύς χώρος.

Πρέπει να αναφερθεί ότι το μέγεθος του τοπικού vocabulary είναι σημαντικά μικρότερο του καθολικού, οπότε γι' αυτό οι τεχνικές αποφυγής διπλοτύπων διαφέρουν για τα δύο παραπάνω bullet.

Τα vocabulary αυτά, τα χρησιμοποιήσαμε στην δημιουργία των tfidf ή BoW vectors για κάθε json αρχείο. Στην αρχή, δημιουργήσαμε τους πίνακες αυτούς με την αρχική διαστατικότητα του vocabulary. Η πληθικότητα του vocabulary, όμως, αρχικά ήταν ~50.000 λέξεις και σημεία στίξης γεγονός που οδήγησε σε καθυστερήσεις κατά το training και διακοπή (kill) της διεργασίας λόγω μεγάλης χρήσης μνήμης. Έτσι, προβήκαμε στις εξής ενέργειες:

- Κατά το διάβασμα των αρχείων κάναμε preprocessing και "κόβαμε" κάποιες λέξεις, δηλαδή δεν τις αφήναμε στο vocabulary καθώς δεν έδιναν μεγάλη γνώση για την εφαρμογή μας, με τη χρήση stopwords. Συγκεκριμένα δεν αφήναμε σύμβολα (παρέμεναν μόνο αριθμοί και γράμματα), τους σκέτους αριθμούς, τις λέξεις μήκους 1 χαρακτήρα αλλά και τις συχνά χρησιμοποιούμενες λέξεις της αγγλικής γλώσσας. Επίσης, για να περιορίσουμε τα διπλότυπα κάναμε όλες τις λέξεις να περιέχουν πεζά γράμματα. Τέλος, δεν αποθηκεύαμε τα πεδία property name διότι δεν προσέδιδαν κάποια περεταίρω πληροφόρηση, ενώ εκτός εάν αυτά είχαν ως property value 'yes', άρα και χαρακτήριζαν την φωτογραφική μηχανή.
  Με αυτόν τον τρόπο πέσαμε στις *24482* λέξεις.
- Επειδή όμως και το πλήθος αυτό είναι μεγάλο για μια εφαρμογή machine learning , μειώσαμε την διάσταση των διανυσμάτων tfidf / BoW με τη χρήση των tfidf τιμών . Συγκεκριμένα κρατήσαμε τις *1000 λέξεις που έχουν το μεγαλύτερο μέσο tfidf*. Γενικά, δοκιμάσαμε δύο εναλλακτικές στην τεχνική αυτή. Η πρώτη είναι να γίνεται επανυπολογισμός των tfidf (για τα BoW δε χρειάστηκε κάτι τέτοιο) μετά

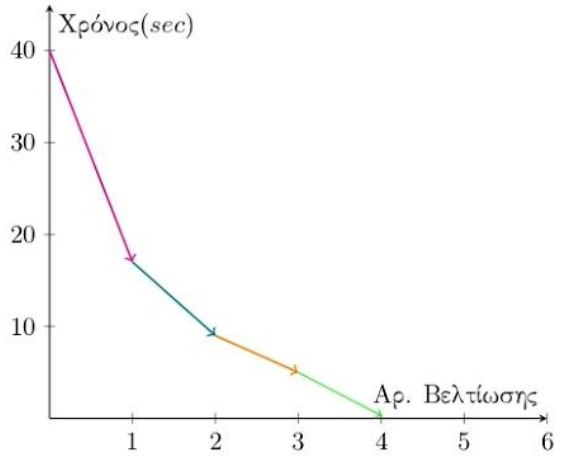
την διαγραφή κάποιων λέξεων με χαμηλό tfidf, και η δεύτερη είναι απλά η διαγραφή των αντίστοιχων τιμών στο διάνυσμα. Η πρώτη τεχνική μας έδινε κατά μέσο όρο *5% καλύτερο accuracy στο validation set*.

Για να γίνει γρήγορα ο υπολογισμός των tfidf / BoW vectors, σημαντικό ρόλο έπαιξε ο πίνακας κατακερματισμού για το global vocabulary.

Τώρα, θα παραθέσουμε έναν πίνακα με χρόνους εκτέλεσης μετά από κάθε βελτίωση.
Οι χρόνοι αναφέρονται στη διαδικασία διάβασμα και αποθήκευση JSON και δημιουργία tfidf / bow vectors. Οι βελτιώσεις εφαρμόστηκαν με τη σειρά που δίνονται σειριακά.

Αρχική Κατάσταση: Καμία βελτιωμένη λίστα και απουσία hash στα vocabulary, και 3 φορές διάβασμα των πινάκων στα JSON για επεξεργασία ***Μέσος Χρόνος Αρχικής Κατάστασης:*** 40 min	
Βελτίωση	**Μέσος Χρόνος Διαδικασίας**
Διάβασμα πινάκων στα JSON 2 φορές	17 min
Διάβασμα πινάκων στα JSON 1 φορά	9 min
Ταξινομημένες λίστες και στα 2 είδη vocabulary	5 min
Τελικές μορφές για τα vocabulary όπως στα bullets	**18 sec**

Δίνεται και γραφική αναπαράσταση:



Σχήμα 1: Γραφική αναπαράσταση βελτίωσης χρόνου

* Οι χρόνοι μετρήθηκαν σε Dell Inspiron 15 5000 Series με CORE i7 και RAM 8GB dual boot σε περιβάλλον linux.

Προτού συνεχίσουμε με την επεξήγηση δημιουργίας των train-test-validation sets, απαραίτητο είναι να εξηγήσουμε την φύλαξη και δημιουργία κλικών-αντικλικών.

Κλίκες - Αντικλίκες

Για την αποθήκευση των κλικών και των αρνητικών συσχετίσεων (αντικλικών) χρησιμοποιήσαμε λίστα. Συγκεκριμένα, κάθε εγγραφή του hash table περιείχε έναν δείκτη σε λίστα (την κλίκα στην οποία ανήκει). Η λίστα ήταν ίδια για όλα τα στοιχεία της κλίκας, αφού ο δείκτης τους έδειχνε στην ίδια θέσης μνήμης, δηλαδή δεν υπήρχαν διπλότυπα. Ομοίως για τις αντικλίκες, ανάμεσα στα στοιχεία της ίδιας αντικλίκας δεν υπήρχαν διπλότυπα.

Μία βελτίωση που μας μείωσε τον χρόνο ήταν η εισαγωγή από μπροστά στη λίστα κλικών και αντικλικών αντί για append που ήταν πριν, δηλαδή από *O(n)* σε *O(1)*.

Επίσης, μια ακόμη βελτίωση που μείωσε τον χρόνο του πρώτου παραδοτέου από *4 sec* σε *3 sec* ήταν τα data ενός κόμβου της λίστας να είναι δείκτης στην εγγραφή αντί για δείκτης στο κλειδί της εγγραφής όπως ήταν αρχικά, διότι έτσι δε χρειάζεται να ψάχνουμε την εγγραφή μέσω του κλειδιού, αλλά έχουμε απευθείας πρόσβαση.

Η πιο σημαντική βελτίωση όμως ήταν αυτή που προαναφέραμε με τον χώρο, αφού δεν σπαταλάμε χώρο σε φύλαξη πολλαπλών ίδιων λιστών ή τα data των κόμβων να είναι διπλότυπα. Ο χώρος που δεσμεύεται με τον τρόπο αυτό μειώνεται κατακόρυφα.

Δημιουργία train-validation-test sets

Ένα πρόβλημα που αντιμετωπίσαμε στην εργασία και μας προβλημάτισε είναι το γεγονός ότι το αρχικό dataset περιείχε 42535 ζεύγη που άνηκαν στην ίδια κλίκα και 299394 ζεύγη που θεωρούνταν αντικλίκες, δηλαδή τα μηδενικά ήταν πολλά περισσότερα από τους άσους και το training set ήταν unbalanced. Για να λύσουμε το πρόβλημα αυτό αποφασίσαμε, στο δεύτερο παραδοτέο, να προσθέτουμε κάθε ζεύγος άσων 7 φορές προκειμένου το πλήθος των μηδενικών και το πλήθος των άσσων να είναι περίπου ίσο. Προτού γίνει αυτό, το accuracy score ήταν αρκετά υψηλό , ενώ το f1

score χαμηλό, ενώ με την αλλαγή αυτή συνέβη το αντίθετο. Στο τρίτο παραδοτέο δεν εφαρμόζεται η συγκεκριμένη τεχνική λόγω του retraining, δηλαδή δουλέψαμε με το αρχικό dataset.

Όπως προτάθηκε στα φροντιστήρια, κάθε παρατήρηση ήταν η απόλυτη διαφορά δύο διανυσμάτων tfidf ή BoW και γινόταν mark σαν 1 ή 0 ανάλογα αν μιλάμε για στοιχεία εντός της ίδιας κλίκας ή όχι. Έτσι, δεν αυξήθηκε η διάσταση των παρατηρήσεων και έγινε εξοικονόμηση χώρου.

Στο δεύτερο παραδοτέο χωρίσαμε 80%-20% train και validation set και θεωρήσαμε ότι το testing-inference θα γίνει από τελείως ξένα data, ενώ στο τρίτο 60-20-20 train-validation-test από τα ίδια data λόγω της εκφώνησης. Και στα δύο παραδοτέα αυτό το split γίνεται με shuffling των δεδομένων προκειμένου να εξασφαλίσουμε μια σχετική ανεξαρτησία στα δεδομένα.

Επειδή η διαδικασία αυτή ήταν σχετικά χρονοβόρα, κάναμε και εδώ χρήση ενός πίνακα κατακερματισμού με σκοπό την αναζήτηση ζευγών προκειμένου να μην έχουμε διπλότυπα ζεύγη και καταφέραμε να μειώσουμε τον χρόνο σημαντικά.

(Re)Training - Gradient Descent

Στο δεύτερο παραδοτέο αρχικά προσπαθήσαμε να υλοποιήσουμε full-batch gradient descent αλλά καθυστερούσε πολύ να ολοκληρώσει, πάνω από 1 ώρα. Συνεπώς, οδηγηθήκαμε στην ανάγκη υλοποίησης stochastic gradient descent η οποία επέστρεφε αποτέλεσμα μετά από το πολύ 2 sec. Στο δεύτερο παραδοτέο είχαμε *accuracy score ~50%* για το validation set με την χρήση της stochastic και την επιπλέον πρόσθεση των θετικών συσχετίσεων όπως αναφέραμε προηγουμένως.

Επίσης, στο δεύτερο παραδοτέο ως συνθήκη τερματισμού του training είχαμε την απειροελάχιστη μεταβολή των βαρών από epoch σε epoch, ενώ στο τρίτο καταλήξαμε στα 5 epochs και πλήθος thread 100, διότι η σύγκλιση αργεί πολύ. Συγκεκριμένα για αυτόν τον αριθμό epochs:

10 νήματα → ~3,30 min/epoch
50 νήματα → ~3,00 min/epoch
100 νήματα → ~2,00 min/epoch

Επιπλέον, καταλήξαμε σε μέγεθος batch = 1024 , διότι δίνει κατα 5% γρηγορότερα αποτελέσματα από μέγεθος batch ≃ 500.

Πριν εφαρμόσουμε retraining, στο τρίτο παραδοτέο , με τα παραπάνω δεδομένα είχαμε accuracy score 0.87 το οποίο βέβαια είναι αναμενόμενο λόγω του unbalanced set όπως αναφέραμε παραπάνω. Μετά το retraining το accuracy στο validation δεν μεταβάλλεται ιδιαίτερα λόγω του unbalanced set και του γεγονότος ότι (λόγω έλλειψης χρόνου) δεν υλοποιήσαμε το επίλυση των transitivity.

Συμπληρωματικά σχόλια περί RAM / CPU usage

Σε συνδυασμό με όσα αναφέρθηκαν παραπάνω για τη μνήμη, θα σχολιάσουμε κάποια άλλα επιμέρους σημεία αναφορικά με τη χρήση της CPU (σε πολυπύρηνο μηχάνημα άρα μιλάμε για ποσοστά εν δυνάμει >100% όπως θα δούμε και έπειτα) και τη RAM στο training.

Γενικότερα, προτού ξεκινήσει το training η CPU usage ανέρχεται

I. στο *60%-70%* στη φάση του διαβάσματος των JSON και δημιουργίας vocabulary
II. στο *80%-100%* στη φάση δημιουργίας tfidf/ BoW vectors και train-valid-test sets

ενώ, η RAM

I. περίπου *0,1 GB* στη φάση του διαβάσματος των JSON και δημιουργίας vocabulary
II. περίπου *1 GB* στη φάση δημιουργίας tfidf/ BoW vectors και train-valid-test sets

Κατά τη διάρκεια του training/(ή re-training) έχουμε κατακόρυφη αύξηση των παραπάνω με τη RAM να ανέρχεται στα περίπου *3,5 GB* και η CPU usage κοντά στο *345%* .

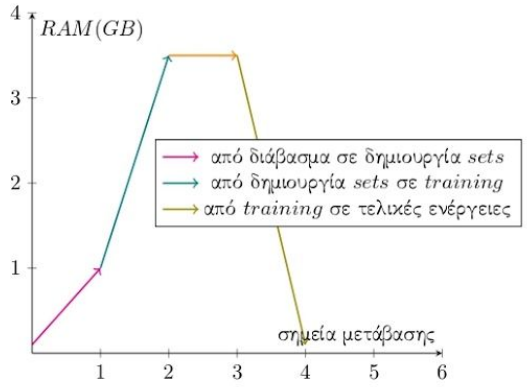
Τα 3,5 GB είναι ένα καλό μέγεθος, διότι αν δούμε μόνο τα data που χρησιμοποιούνται στο training είναι περίπου 1,5 αφού

(A) 350,000 pairs × 1000 dim × 4 bytes/float = 1,4 GB για τα X στο training set συν
(B) 350,000 pairs × 4 bytes/int = 0,002 GB για τα Y στο training set

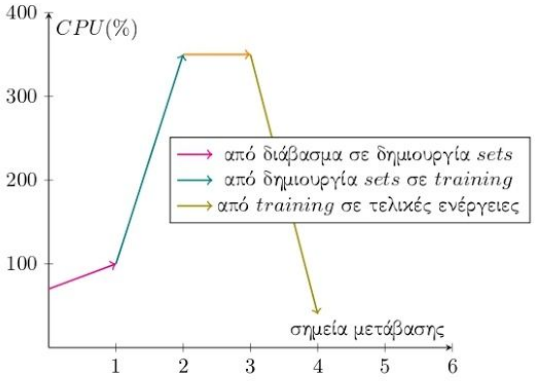
Η μεταβολή αυτή σε RAM και CPU usage είναι αναμενόμενη, διότι το training διαχειρίζεται μεγάλο πλήθος data.

Θα μπορούσε να μειωθεί η χρήση της RAM με τη χρήση sparse matrices, αλλά θα έπρεπε να τροποποιηθούν για τη χρήση τους στο training και στο inference το οποίο δεν είναι ιδιαίτερα βολικό. Συνεπώς, επιλέξαμε αυτό το t*radeoff μνήμης - ευκολίας διαχείρισης δεδομένων*.

Θα δείξουμε γραφικά στην αύξηση της χρήσης CPU και RAM.



Σχήμα 1: Γραφική αναπαράσταση αύξησης χρήσης *RAM*



Σχήμα 1: Γραφική αναπαράσταση αύξησης χρήσης *CPU*

Συμπληρωματικά σχόλια περί χρόνου εκτέλεσης / Ρόλος των threads

Στο δεύτερο μας παραδοτέο ο συνολικός χρόνος εκτέλεσης ήταν *3 min 37 sec*.

Στο τελικό μας παραδοτέο οι χρόνοι έχουν ως εξής:

Διάβασμα αρχείων-δημιουργία vocabulary	9 sec
Δημιουργία tfidf/BoW vectors ∀ JSON file	9 sec
Δημιουργία κλικών-αρνητικών συσχετίσεων	1 sec
Δημιουργία train-valid-test sets με shuffling	3 min
Re-training - validation	2 min/epoch × 5 epochs × 4 iterations re-training
Result-memory free	1 sec

Η μεγάλη διαφορά όσον αφορά τον χρόνο ανάμεσα στα δύο παραδοτέα οφείλεται σε δύο παράγοντες : (1) στη χρήση stochastic gradient descent στο δεύτερο παραδοτέο που είναι δίχως αμφιβολία πιο γρήγορη από την batch gradient descent του τρίτου και (2) στο retraining της τρίτης εργασίας το οποίο πέραν του γεγονότος ότι εκτελεί πολλαπλές φορές την ίδια μη-σύντομη εργασία batch gradient descent, εκτελεί και άλλες ενέργειες (π χ εισαγωγή παρατηρήσεων στο training set).

Εξ ορισμού, λοιπόν, η διαδικασία του training (και πόσο μάλλον του retraining) είναι χρονοβόρα και είναι πρακτικά αδύνατο να μειωθεί σε χρόνο σαν διαδικασία στο full batch gradient descent. Στο συγκεκριμένο project ένας πιθανός τρόπος μείωσης του χρόνου πέρα από τη χρήση stochastic gradient descent όπως στο δεύτερο παραδοτέο, είναι μία διαφορετική διαχείριση των threads. Όμως, λόγω έλλειψης χρόνου ως προς τη διεκπεραίωση της τρίτης εργασίας δεν είχαμε τον απαραίτητο χρόνο να πειραματιστούμε αρκετά στο κομμάτι αυτό. Ωστόσο, καταλήξαμε στην τελική υλοποίηση ως εξής:

Σε κάθε επανάληψη της επαναληπτικής μάθησης, δημιουργούνται threads και το κάθε ένα αναλαμβάνει ένα (ή περισσότερα) batch(-es) μεγέθους 1024 προκειμένου να υπολογίσει την αντίστοιχη παράγωγο. Αρχικά, επιλέξαμε τα threads να ξεκινούν τη δουλειά τους αφότου δημιουργηθούν όλες οι δουλειές και ανατεθούν στα threads. Όμως κάτι τέτοιο δεν είναι αποδοτικό από δύο απόψεις: 1) δε δίνει τη δυνατότητα σε κάποιο thread να αρχίσει να εκτελεί τις απαιτούμενες ενέργειες πριν από κάποιο άλλο (όταν δηλαδή δημιουργηθεί μία εργασία και εισαχθεί στην ουρά) και 2) ο job-scheduler καταναλώνει άσκοπο χρόνο στο διαμοιρασμό εργασιών εξ αρχής και όχι σε κάτι πιο αποδοτικό όπως κάθε thread να αναλαμβάνει μια εργασία όσο υπάρχουν εργασίες. Επομένως, κάναμε απαλοιφή των δύο αυτών προβλήματων και έτσι τα thread “ξυπνούσαν” μόλις εμφανιζόταν εργασία για αυτά στην ουρά και λειτουργούσαν όσο υπήρχαν δουλειές στην ουρά. Τέλος, όταν τερμάτιζαν όλα τα νήματα λόγω λήξης των εργασιών τους και απουσίας νέων, συνένωναν τα δεδομένα τους και έκαναν simultaneous update των βαρών το οποίο εξ ορισμού είναι και πιο αποδοτικό.

Συμπεράσματα

Η εφαρμογή που μας ζητήθηκε να υλοποιήσουμε είναι μία σχετικά “βαριά” εφαρμογή, δηλαδή με μεγάλες απαιτήσεις σε χώρο και υπολογιστική ισχύ. Αυτό συνεπάγεται ότι λανθασμένες επιλογές αναφορικά με δομές που χρησιμοποιήθηκαν για τη φύλαξη των δεδομένων και αλγοριθμικές διαδικασίες θα καθιστούσαν αδύνατον να υλοποιηθεί και να τεσταριστεί σε οικιακούς υπολογιστές. Γι’αυτό κατά τη διάρκεια υλοποίησής της έγιναν πολλές αλλαγές και βελτιστοποιήσεις προκειμένου να πετύχουμε το ζητούμενο αποτέλεσμα. Το παραπάνω αποτελεί το λόγο που η υλοποίηση της εφαρμογής από μία ομάδα ήταν αναγκαία, καθώς το κάθε μέλος της προσέφερε μια διαφορετική προσέγγιση στα τυχόν προβλήματα και με ανταλλαγή απόψεων οδηγηθήκαμε στην ιδανική λύση για αυτά.

Καθόλη τη διαδικασία ανάπτυξης, ενημέρωσης και βελτίωσης των τριών παραδοτέων εργασιών του μαθήματος εξοικειωθήκαμε με την πλατφόρμα του github. Αναγνωρίσαμε και εκτιμήσαμε ιδιαίτερα τις δυνατότητες που προσφέρει για την ανάπτυξη εκτενούς πληροφοριακού λογισμικού και για την υποστήριξη της ομαδικής παράλληλης εργασίας. Η εύκολη πρόσβαση σε προηγούμενες εκδόσεις του κώδικα καθώς και η αξιοποίηση της διεξαγωγής test με χρήση των github actions μας διευκόλυνε στην αντικειμενική αξιολόγηση του χρόνου εκτέλεσης του προγράμματός μας.