ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Final Project Report

Ανδρέας Παππάς 1115201500201

Νικήτας Σακκάς 1115201400176

Κωνσταντίνα Νίκα 1115201400126

Εργασία 2020-2021

Περίληψη

Η παρακάτω αναφορά, αφορά το μάθημα του κυρίου Ιωάννη Ιωαννίδη, το οποίο ονομάζεται Ανάπτυξη Λογισμικού για Πληροφοριακά συστήματα και αφορά στην εκμάθηση υλοποίησης αλγορίθμων, δομών δεδομένων και τεχνικών για συγγραφή κώδικα με τις γλώσσες προγραμματισμού C και C++ ούτως ώστε να καταλήξουμε στην πλήρη ανάπτυξη μια εφαρμογής πληροφοριακού συστήματος.

Το project το οποίο υλοποιήθηκε στο μάθημα αυτό, αρχικά έσπασε σε 3 μικρότερα κομμάτια, τα οποία όλα μαζί αποτελούν την τελική μας εφαρμογή.

Επιβλέπων καθηγητής της ομάδας μας ήταν ο κύριος Γεώργιος Πικραμένος ο οποίος μέσα από τα μαθήματά του ήταν άκρως βοηθητικός στην όλη διεκπεραίωση της εργασίας.

Παρακάτω θα ακολουθήσει η εκτενής ανάλυση καθενός από τα 3 κομμάτια της εργασίας καθώς και θα υπάρχουν παραδείγματα, πίνακες και γραφικές παραστάσεις τα οποία θα τεκμηριώνουν τις σχεδιαστικές μας επιλογές και την αποτελεσματικότητα των επιλογών μας στην εργασία αυτή.
Τα πειράματα και οι χρόνοι εκτέλεσης αφορούν μηχάνημα με επεξεργαστή Intel(R) Core(™) i5-7300U CPU @ 2.6GHz, μνήμη RAM 16GB, m.2 ssd Samsung Evo 970 Plus 500GB, Intel(R) HD Graphics 620. Το λειτουργικό σύστημα του μηχανήματος αυτού είναι Linux και συγκεκριμένα Ubuntu 20.04 και ο μεταγλωττιστής που χρησιμοποιήθηκε είναι ο gcc, version 9.3.0

Τέλος θα κλείσουμε με τα συμπεράσματα και τις γνώσεις τις οποίες αποκτήσαμε στο μάθημα αυτό μέσω της υλοποίησης του συγκεκριμένου επαγγελματικού, ομαδικού project.

Μέρος 1ο

Το πρώτο μέρος της εργασίας αφορά στην υλοποίηση δομών δεδομένων για την αποθήκευση και διαχείριση ενός dataset 30.000 περίπου προϊόντων από ηλεκτρονικές ιστοσελίδες. Πρακτικά αυτό που έπρεπε να γίνει είναι η ανάλυση του ανωτέρω dataset ούτως ώστε να εξάγουμε σε ένα αρχείο όλα τα προϊόντα τα οποία είναι ίδια αλλά πιθανόν από διαφορετικές ιστοσελίδες.
Αρχικά το dataset δόθηκε σε μορφή Json οπότε έπρεπε με κάποιον τρόπο να φτιάξουμε κάποια συνάρτηση η οποία θα μας εξυπηρετούσε στο να εξάγουμε ότι πληροφορία χρειαζόμαστε από το dataset αυτό. Αρχικά χρησιμοποιήσαμε μια έτοιμη βιβλιοθήκη json parsing η οποία είναι η json-c. Αργότερα βέβαια η συγκεκριμένη βιβλιοθήκη δεν κάλυπτε τις ανάγκες μας τις οποίες μάθαμε στο 2ο μέρος της εργασίας οπότε έπρεπε να προχωρήσουμε στην πλήρη ανάπτυξη από το μηδέν, χωρίς χρήση εξωτερικής βιβλιοθήκης, του Json parser.

Η ιδέα που είχαμε αρχικά για την υλοποίηση των δομών μας ήταν να δημιουργήσουμε ένα AVL δέντρο όπου κάθε node να περιέχει τις απαραίτητες πληροφορίες ενός json file. Το AVL επιλέχθηκε επειδή έχει χαμηλή πολυπλοκότητα αναζήτησης, κάτι που θα απαιτείται ιδιαίτερα στην εργασία. Επίσης, κάθε node περιέχει έναν pointer σε μία μονά συνδεδεμένη λίστα από buckets οι οποίες περιέχουν πίνακες με pointers πίσω σε nodes. Όπως θα φανεί μετά, κάθε τέτοια λίστα θα αντιπροσωπεύει μία κλίκα από json files. Η μορφή των tree nodes:

```
typedef struct tree_entry{
    int ht;                                  //ύψος του node
    struct tree_entry* left;
    struct tree_entry* right;
    int json                                 //κλειδί του json file
    char* path_with_json;                    //όνομα του json file
    char* specs;                             //περιεχόμενα του json file
    struct headBucket* headbucket;           //κλίκα
```

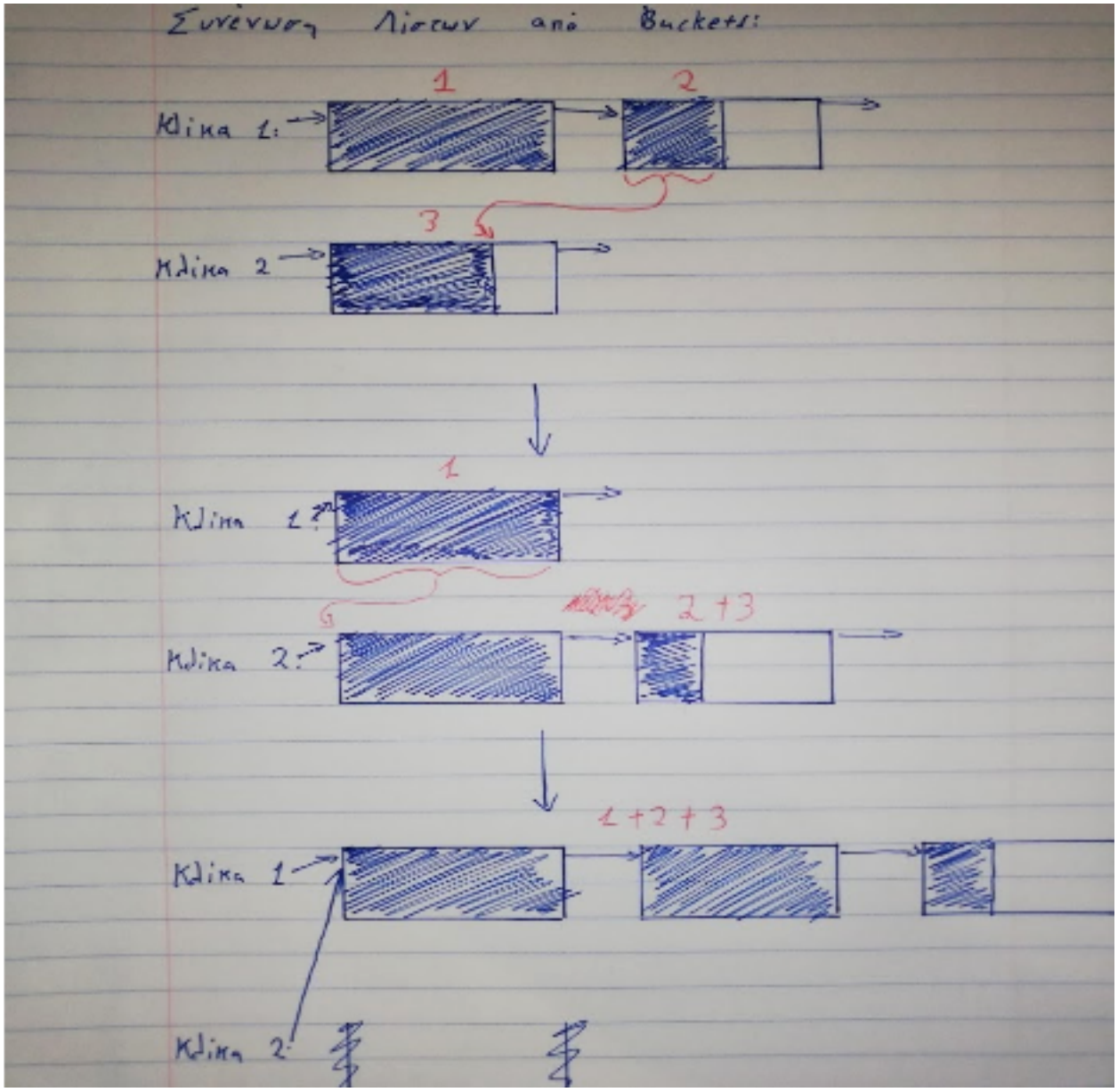
}tree_entry;

Το πρόγραμμά μας λοιπόν αρχικά ανοίγει τον φάκελο του dataset όπου μέσα βρίσκονται οι διαφορετικοί κατάλογοι με τα json αρχεία και για κάθε ένα από αυτούς τους υποκαταλόγους αναζητεί και ανοίγει τα json αρχεία. Κατόπιν εισάγουμε ένα ένα τα json αρχεία μέσα στο δέντρο. Στην εκφώνηση είχε ειπωθεί ότι στο dataset δεν υπάρχουν διπλότυπα αρχεία, συνεπώς δεν υπάρχει λόγος ελέγχου κατα την εισαγωγή.

Η δημιουργία των κλίκων γίνεται ως εξής:

Η διεργασία διαβάζει μία μία τις γραμμές του W dataset. Κάθε γραμμή ομοιότητας μεταξύ δύο json υποδηλώνει ότι οι κλίκες τους πρέπει να συνενωθούν (αν τα δύο json δεν ανήκουν ήδη στην ίδια κλίκα). Στην αρχή αυτή η συνένωση γινόταν με τη μεταφορά των στοιχείων από την μία κλίκα(όπως είπαμε, κλίκα = λίστα από buckets με pointers στα tree nodes του dataset) στην άλλη. Αυτό όμως ήταν μία πολύ χρονοβόρα διαδικασία. Τελικά, καταλήξαμε στη συνένωση των δύο κλίκων με την αλλαγή των pointers των αντίστοιχων λιστών(πρακτικά στοίχιση των 2 λιστών). Βέβαια, πριν από αυτή την αλλαγή πρέπει να γίνει η μεταφορά των στοιχείων του τελευταίου bucket της πρώτης λίστας στην δεύτερη, ώστε να μην υπάρχουν κενά στα buckets.

Για οπτική αναπαράσταση δίνεται το σχέδιο της συνένωσης 2 κλίκων σχηματικά (αν στο W dataset δίνεται ότι έχουν ένα τουλάχιστον όμοιο ζευγάρι):



Συνεπώς η ένωση 2 κλίκων γίνεται με την μεταφορά των εγγραφών μόνο του τελευταίου bucket της μίας κλίκας στην άλλη και με την αλλαγή μερικών δεικτών.

Τέλος, μετά τη δημιουργία των κλίκων, εκτυπώνονται όλα τα όμοια ζευγάρια των json files(θετικές συσχετίσεις) στο αρχείο positive_relations.csv σε μορφή:

“json5” “json3” 1

“json3” “json2” 1
“json6” “json8” 1
…

Τελικά, με βάση την τελική μας σχεδιαστική μας επιλογή πήραμε χρόνο εκτέλεσης **3 δευτερολέπτων** έναντι των **15 λεπτών** που είχαμε αρχικά και κατόπιν μερικών διορθώσεων στους δείκτες των buckets στους κόμβους των δέντρων μας, πήραμε τελικό χρόνο εκτέλεσης **1,6 δευτερολέπτων** στο μεγάλο μας dataset και χρόνο μικρότερο του **1 δευτερολέπτου** στο μικρό dataset.

Βασικές Συναρτήσεις:

```c
/* Inserts json entries inside the buckets of the tree.
 * If the bucket is full it allocates memory from the heap to create a new one
 * and adds the entry inside the newly created bucket
 */
void insert_entry(tree_entry *, bucket *);

/* Inserts into the tree nodes the relations of each and every json file.
 * i.e. if they're equal or not equal files (based on the project's description)
 */
void add_positive_relation(tree_entry *, char *, char *, int);
void add_negative_relation(tree_entry *, char *, char *, int);

/* Prints the relations of json files on each and every node of the tree. */
void print_node_positive_relations(tree_entry *,FILE*);

/* Prints the relations of each and every of the files.
 * outputs the same products onto a .csv file.
 */
void print_all_positive_relations(tree_entry *,FILE*);

void print_node_negative_relations(tree_entry *, clique_tree_entry* , FILE* );
void print_all_negative_relations(tree_entry *,FILE*);
```

Η συναρτήσεις με negative relations χρησιμοποιούνται στο 2ο μέρος της εργασίας.

Μέρος 2ο

Το δεύτερο μέρος της εργασίας αποτελείται από 4 βασικά μέρη:

1. Εύρεση αρνητικών συσχετίσεων των json files.
2. Δημιουργία Bag_of_words και tf_idf representations με βάση τα κείμενα που περιέχονται στα αντίστοιχα json files
3. “Μετάφραση” των συσχετίσεων (θετικών και αρνητικών) σε διανύσματα που προκύπτουν από την απόλυτη διαφορά των διανυσμάτων BoW ή tf_idf των αντίστοιχων json. Δημιουργία training και test set με τις παραπάνω συσχετίσεις.
4. Εκπαίδευση μοντέλου Machine Learning που να προβλέπει αν μια συσχέτιση json είναι θετική ή αρνητική και αξιολόγησή του.

1. Εύρεση αρνητικών συσχετίσεων των json files.

Αρχικά σε αυτό το κομμάτι έπρεπε να υλοποιήσουμε ακριβώς το ίδιο ζητούμενο που είχε το μέρος 1ο της εργασίας με την μόνη διαφορά, ότι αυτή την φορά μας ενδιαφέρουν τα αρχεία τα οποία είναι διαφορετικά μεταξύ τους και όχι ίδια (δηλαδή τα προϊόντα τα οποία **δεν** είναι ίδια).

Στην υλοποίηση αυτού του κομματιού χρησιμοποιήσαμε ότι είχαμε φτιάξει στο πρώτο μέρος της εργασίας με την μόνη διαφορά ότι αυτή την φορά προσθέσαμε ένα δέντρο και ένα κλειδί στην αρχή κάθε λίστας από buckets(κλίκα). Δηλαδή το “κεφάλι” κάθε λίστας είναι:

```
typedef struct headBucket{
   char* key;                                  //κλειδί της κλίκας
   struct clique_tree_entry* different_cliques_root; //δέντρο διαφορετικών κλίκων
   struct bucket* first_bucket;
}headBucket;
```

Πρακτικά, κάθε αρνητική συσχέτιση δεν δηλώνει ότι τα 2 αρχεία είναι διαφορετικά, αλλά ότι οι κλίκες τους είναι διαφορετικές. Έτσι, για κάθε αρνητική συσχέτιση προσθέτουμε(αν δεν υπάρχει ήδη) ένα node στο δέντοα των αρνητικών κλικών που περιέχει pointer που δείχνει στην καινούργια διαφορετική κλίκα. Αυτό γίνεται και στα 2 δέντρα των αντίστοιχων κλίκων.

Επιλέξαμε δέντρο για τον γρήγορο έλεγχο προυπάρχοντος node στο δέντρο και την αποφυγή διπλότυπων (κλειδί του δέντρου είναι το char* key και είναι μοναδικό για κάθε κλίκα).

Τέλος, με βάση την παραπάνω δομή, εκτυπώνονται οι αρνητικές συσχετίσεις στο αρχείο negative_relations.csv, όπως και στις θετικές συσχετίσεις.

2. Δημιουργία Bag_of_words και tf_idf representations με βάση τα κείμενα που περιέχονται στα αντίστοιχα json files.

Για τη δημιουργία των παραπάνω representations χρειάζεται αρχικά να κάνουμε **parsing**, **preprocessing** και να **δημιουργήσουμε ένα λεξιλόγιο** με όλες τις διαφορετικές λέξεις των json files.

Parsing: Ο json_parser μας “καθαρίζει” τα κείμενα των json files, αφαιρώντας τα labels των specs και τα σύμβολα που περιέχονται σε ενα json file, π.χ. Το json με περιεχόμενα:

```
{
   "<page title>": "Samsung Digital Camera WB35F - Walmart.com",
   "battery type": "Lithium Ion",
   "product in inches l x w x h": "3.97 x 2.41 x 0.81",
   "walmart no": "552574194"
}
```

...θα αποθηκευτεί στη μνήμη ώς:

“Samsung Digital Camera WB35F - Walmart.com Lithium Ion 3.97 x 2.41 x 0.81 552574194”.

Σημείωση: Επιλέξαμε να αγνοήσουμε τα labels για να αποφύγουμε την σύγχυση του ταξινομητή αργότερα που μπορεί να προκύψει από τα μοτίβα των labels που έχει κάθε ιστοσελίδα. Για παράδειγμα, η amazon μπορεί στο template παρουσίασης μίας φωτογραφικής μηχανής να απαιτεί την εισαγωγή των συγκεκριμένων specs: model, batery, resolution, zoom. Συνεπώς, η ταξινόμηση μετά θα μας προβλέπει αν τα json files είναι στην ίδια ιστοσελίδα, και όχι αν αναφέρονται στο ίδιο προϊόν.

Preprocessing: Η προεπεξεργασία των λέξεων αποτελείται από τα εξής βήματα:

- Lowercasing το κείμενο (πρακτικά ίδιες λέξεις).
- Αφαίρεση των συμβόλων, για παράδειγμα η λέξη *"camera),"* γίνεται *"camera"* (επίτηδες όμως δεν αφαιρούμε όλα τα σύμβολα, καθώς δε θέλουμε για παράδειγμα η λέξη "3.5" να γίνει "35" αφού θα αλλάξει το νόημα της).
- Αγνοούμε πολύ μεγάλες λέξεις (MAXWORDSIZE = 10 στο train_handler.h file, changeable), καθώς είναι πιθανότατα θόρυβος.
- Αφαίρεση stopwords (μετά τη δημιουργία του λεξιλογίου).

Χωρίς το preprocessing προκύπτουν πάρα πολλές λέξεις στο λεξιλόγιο (>400.000), οι οποίες θα αύξαναν και την πολυπλοκότητα της ταξινόμησης, αλλά και το θόρυβο, φέρνοντας σημαντικά χειρότερα αποτελέσματα.

Δημιουργία λεξιλογίου

Το λεξιλόγιο είναι ένα δέντρο(για γρήγορη αναζήτηση), όπου κάθε κόμβος περιέχει μία λέξη και μία στήλη από *int wordcounts[]*, η οποία πρακτικά είναι μία στήλη του πίνακα BoW που θα φτιαχτεί αργότερα. Κατά το διάβασμα των αρχείων, κάθε λέξη μετά το preprocessing αναζητείται στο λεξιλόγιο. Αν υπάρχει, αυξάνει το wordcounts[json_key] στο αντίστοιχο json

στο οποίο βρίσκεται. Αν δεν υπάρχει, φτιάχνεται καινούργιο node στο δέντρο με wordcounts[json_key] = 1 και όλα τα άλλα 0 (πρώτη φορά που βρέθηκε η λέξη).
Μετά τις παραπάνω διαδικασίες, είμαστε έτοιμοι να κατασκευάσουμε τα BoW και tf_idf arrays.

Bag_of_words array:

Η BoW array είναι πρακτικά ήδη έτοιμη όταν τελειώσει η δημιουργία του λεξιλογίου, με κάθε στήλη της να βρίσκεται σε ένα από τα nodes του dictionary tree. Οπότε διαβάζουμε αναδρομικά το δέντρο (το κλειδί του δέντρου είναι η λέξη οπότε με σωστή αναδρομή οι στήλες θα προκύψουν και αλφαβητικά) και σώζουμε τις διευθύνσεις των στηλών σε ένα διάνυσμα από pointers, δημιουργώντας τελικά τον πινακα BoW transpose(δεν υπάρχει κάποιος πρακτικός λόγος να τον γυρίσουμε, απλά θα πάρει περισσότερο χρόνο).

TF_IDF array:

Για τη δημιουργία του TF_IDF array χρειάζονται οι εξής πληροφορίες:

Για τα tf:

- Συνολικός αριθμός λέξεων ενός json file (προκύπτει από το άθροισμα των στοιχείων του διανύσματος που αντιστοιχεί στο συγκεκριμένο json file).
- Στοιχεία του BoW (τα έχουμε ήδη) .

Για τα idf:

- Αριθμός αρχείων που περιέχουν μία συγκεκριμένη λέξη τουλάχιστον μία φορα (προκύπτει με ένα iteration στον BoW).
- Συνολικος αριθμός json files(Το γνωρίζουμε ήδη).

Τα κόκκινα στοιχεία υπολογίζονται με ένα iteration του BoW, τα μαύρα τα γνωρίζουμε ήδη. Με τις παραπάνω πληροφορίες και τις σωστές πράξεις δημιουργούμε και τον πίνακα TF_IDF.

Βελτιώσεις των BoW και TF_IDF:

Σε αυτό το σημείο υλοποιήσαμε κώδικα για τη βελτίωση των πινάκων μας (παρ'ολο που δεν είχε ζητηθεί ακόμα). Αυτή η βελτίωση γίνεται βρίσκοντας *WORLDS_FOR_DATASET* (changeable) στήλες με το χαμηλότερο idf value (δηλαδή τις πιο σημαντικές λέξεις) και χρησιμοποιώντας αυτές τις στήλες από το BoW ή TF_IDF representation για τη δημιουργία των παρακάτω sets.

3. "Μετάφραση" των συσχετίσεων (θετικών και αρνητικών) σε διανύσματα που προκύπτουν από την απόλυτη διαφορά των διανυσμάτων BoW ή tf_idf των αντίστοιχων json files. Δημιουργία training και test set με τις παραπάνω συσχετίσεις.

Ακολουθεί ψευδοκώδικας της συνάρτησης δημιουργίας training, test και validation set:

- ➢ Διάβασε τη συσχέτιση από το relation file (θετική η αρνητική).
- ➢ Βρες τα κλειδιά των json αρχείων στα οποία αναφέρεται.
- ➢ Με τη χρήση των παραπάνω κλειδιών διάβασε τον πίνακα (BoW - TF_IDF) και υπολόγισε την απόλυτη διαφορά των δύο διανυσμάτων αναπαράστασης των json αρχείων.
- ➢ Γράψε το αποτέλεσμα στο αντίστοιχο αρχείο(⅗ των συσχετίσεων στο Train_Set.csv, ⅕ στο Test_Set.csv και ⅕ στο Val_Set.csv)

<u>Σημειώσεις:</u>

1. Γνωρίζουμε ότι οι αρνητικές συσχετίσεις είναι πολύ περισσότερες(10x περίπου) των θετικών, και το μοντέλο θα τείνει να προβλέπει αρνητικά. Για αυτό το λόγο επιλέξαμε να συμπεριλάβουμε στα δεδομένα ίδιο αριθμό θετικών και αρνητικών συσχετίσεων για να έχουμε καλύτερα αποτελέσματα. Άλλωστε ο αριθμός αυτων των δεδομένων θα είναι αρκετός(σχεδόν 100.000 συσχετίσεις).

2. Οι μετασχηματισμένες συσχετίσεις στα sets γράφονται εναλλάξ(μία θετική - μία αρνητικη), ώστε οι τελικοί πίνακες να είναι πρακτικά “ανακατεμένοι”.

Συναρτήσεις που υλοποιούν τα παραπάνω:

```
//create bow and tf_idf functions
void create_train_set_tfidf(float **, tree_entry *, char *, char *);
void create_train_set_bow(int **, tree_entry *, char *, char *);
int stopword_check(char *);
void add_json_keys(tree_entry *);
int *sort_idf_array();
int **improve_bow(int **);
float **improve_tf_idf(float **);
bow_tree_entry *create_bow_tree(tree_entry *);
bow_tree_entry *add_jsons_to_bow(bow_tree_entry *, tree_entry *);
bow_tree_entry *json_to_bow(bow_tree_entry *, tree_entry *);
bow_tree_entry *word_to_bow(bow_tree_entry *, char *, int);
int **create_bow_array(tree_entry *);
void get_bow_tree_entries(bow_tree_entry *, int **);
float **create_tf_idf(int **);

//dictionary functions
bow_tree_entry *bow_insert(bow_tree_entry *, int, char *);
bow_tree_entry *bow_search(bow_tree_entry *, char *);
int bow_height(bow_tree_entry *);
bow_tree_entry *bow_rotateright(bow_tree_entry *);
bow_tree_entry *bow_rotateleft(bow_tree_entry *);
bow_tree_entry *bow_RR(bow_tree_entry *);
bow_tree_entry *bow_LL(bow_tree_entry *);
bow_tree_entry *bow_LR(bow_tree_entry *);
bow_tree_entry *bow_RL(bow_tree_entry *);
int bow_BF(bow_tree_entry *);
```

4. Εκπαίδευση μοντέλου Machine Learning που να προβλέπει αν μια συσχέτιση json είναι θετική ή αρνητική και αξιολόγησή του:

Το 4ο κομμάτι λοιπόν του 2ου μέρους της εργασίας αφορά στην υλοποίηση αλγορίθμων μηχανική μάθησης και πιο συγκεκριμένα στην εκπαίδευση ενός μοντέλου το οποίο δίνοντάς του τα δεδομένα που εξάγαμε από το

προηγούμενο κομμάτι και το πρώτο μέρος, (δηλαδή τα 2 .csv αρχεία με τις θετικες και τις αρνητικές συσχετίσεις προϊόντων) θα περιμένουμε να μας πει με συγκεκριμένο ποσοστό ακριβείας ποια αρχεία είναι ίδια μεταξύ τους και ποιά όχι.

Συγκεκριμένα η υλοποίηση της μηχανικής μάθησης σπάει ως εξής, αρχικά η υλοποίησή της έχει γίνει με την γλώσσα προγραμματισμού C++ και όχι με C όπως το υπόλοιπο project. Οι βασικές συναρτήσεις υλοποίησης του κομματιού Logistic Regression της μηχανικής μάθησης βρίσκονται στο αρχείο Logistic_Rrgr.cpp και είναι οι εξής:

```
1. Matrix LogisticRegression::readFromInputFile(string input_file)
2. double sigmoid(double bias, Point4d weights, Point4d example, int
   features)
3. Point5d calculateGradient(double bias, Point4d weights, Matrix
   examples, int features,int start, int end)
4. double findError(double bias, Point4d weights, Matrix examples, int
   features)
5. Point5d LogisticRegression::gradientDesent(double bias, Point4d
   weights, Matrix examples)
```

Λίγα λόγια για την κάθε μία:

- Η readFromInputFile διαβάζει και ανοίγει το test αρχείο του dataset μας το οποίο το περνάμε ως παράμετρο στην συνάρτηση και το αποθηκεύει σε ένα matrix. Κατόπιν η sigmoid συνάρτηση επιστρέφει την τιμή που υπολογίζει με βάση την λογιστική σιγμοειδή συνάρτηση, όπως αυτή περιγράφεται στην εκφώνηση πάνω στα bias, weights και το input point το οποίο δίνεται.
- Η calculateGradient υπολογίζει και επιστρέφει το gradient της συνάρτησης λογιστικού σφάλματος για δοθέντες τιμές του bias, weights με βάση την περιγραφή της εκφώνησης επίσης.

- Στην συνέχεια η findError υπολογίζει και επιστρέφει την τιμή σφάλματος της λογιστικής συνάρτησης χρησιμοποιώντας τα μαθηματικά της εκφώνησης.

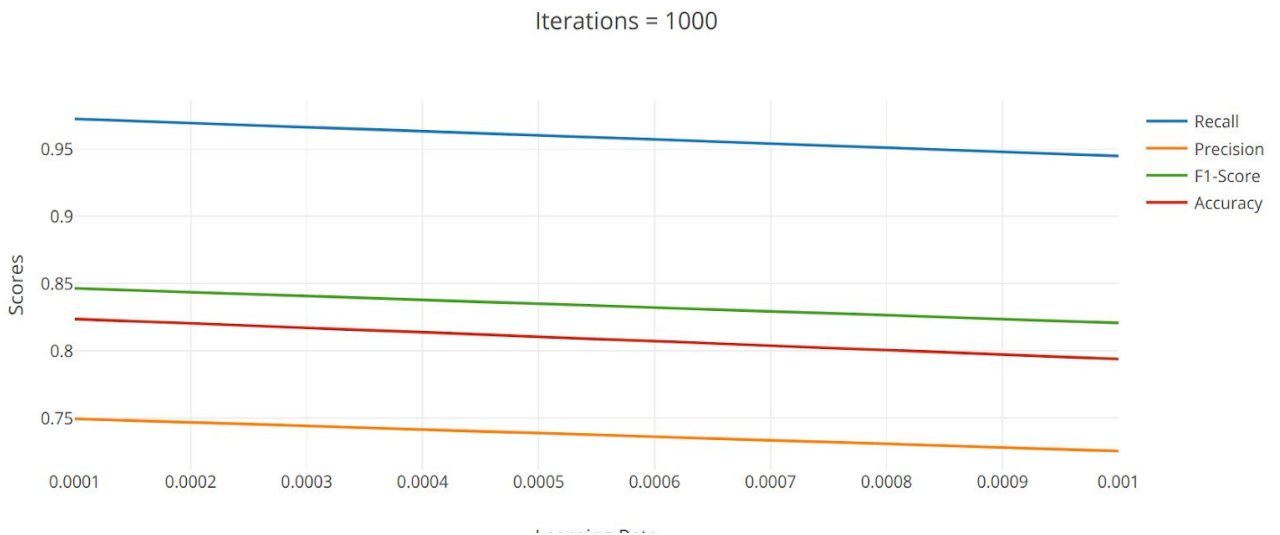
- Και τέλος η gradientDesent κάνει apply το gradient στα βάρη και το bias για συγκεκριμένο αριθμό επαναλήψεων χρησιμοποιώντας την calculateGradient που περιγράψαμε παραπάνω.

- Το testing του μοντέλου γίνεται από την παρακάτω συνάρτηση,

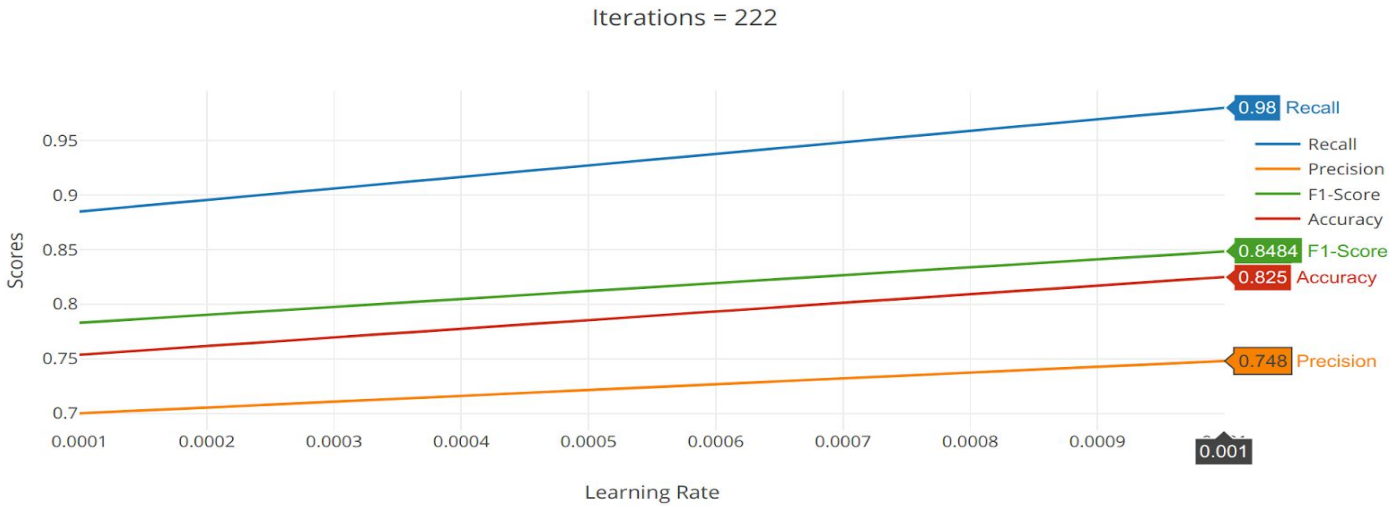
```
void LogisticRegression::testAndPrint(Point5d optimal_weights, Matrix testing_exp)
```

Η οποία δέχεται τα optimal weights τα οποία έχουν υπολογιστεί προηγουμένως και διαβάζει επίσης το testing dataset που έχει περαστεί σε ένα matrix από προηγούμενη αναφερθέντα συνάρτηση και τελικά χρησιμοποιεί κάποια πολύ απλά μαθηματικά για να ελέγξει (testing) αν τα αποτελέσματα που δίνει το μοντέλο μας είναι ορθά ή μη.

Παρακάτω βλέπουμε κάποιες γραφικές παραστάσεις οι οποίες δείχνουν την αποδοτικότητα του μοντέλου μας. Συγκεκριμένα, στην παρακάτω εικόνα μπορούμε να δούμε πως οι διαφορετικές τιμές του learning rate για αριθμό 1000 επαναλήψεων επηρεάζει την αποδοτικότητα του μοντέλου μας. Συγκεκριμένα βλέπουμε πως το καλύτερο δυνατό accuracy στις 1000 επαναλήψεις το πετυχαίνουμε με το χαμηλότερο δυνατόν learning rate. Αυτό όπως θα δούμε παρακάτω δεν ισχύει για μικρότερο αριθμό επαναλήψεων.

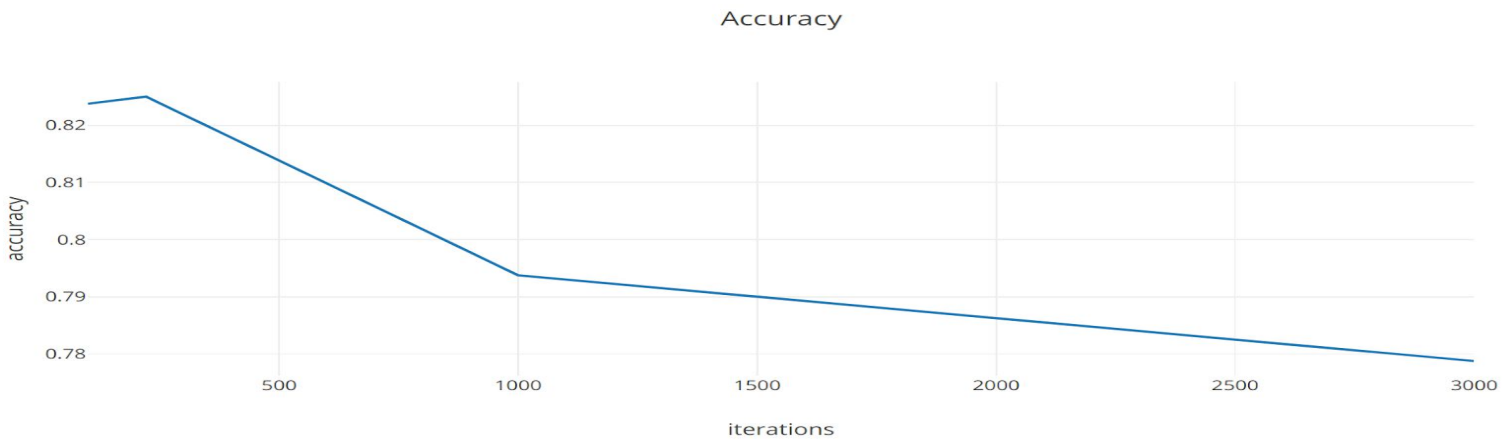


Όπως βλέπουμε στην παρακάτω εικόνα,

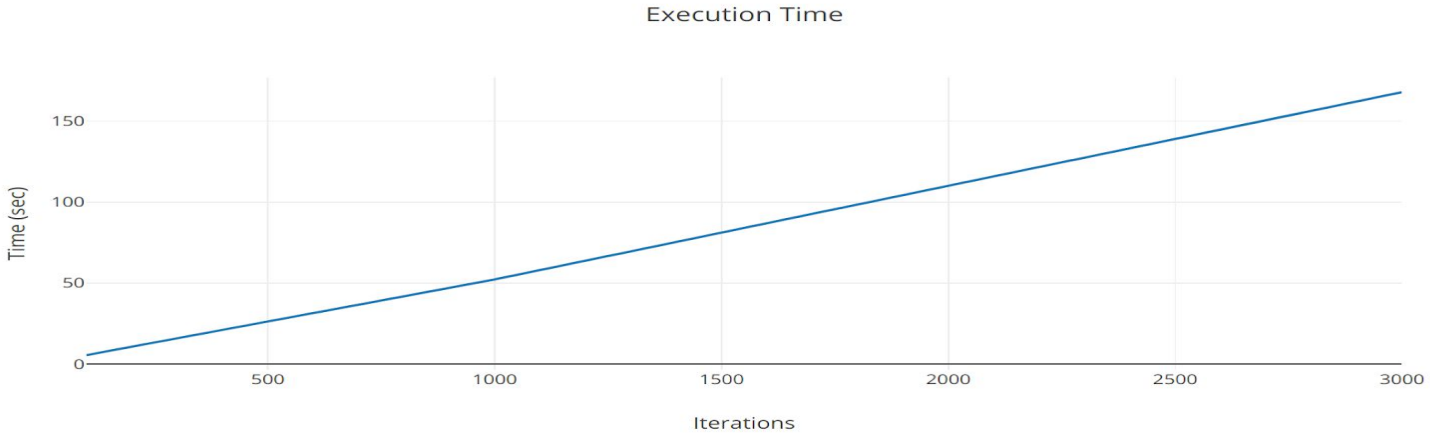


Σε αντίθεση με τις 1000 επαναλήψεις όπου το καλύτερο δυνατό accuraccy το πετυχαίναμε στο μικρότερο δυνατό learning rate, τώρα σε μικρότερο αριθμό επαναλήψεων, τα καλύτερα αποτελέσματα τα πετυχαίνουμε στο “μεγαλύτερο” δυνατό learning rate. Η συγκεκριμένη συμπεριφορά είναι απολύτως φυσιολογική, διότι το μοντέλο μας για να το πούμε απλά σε μικρότερο αριθμό επαναλήψεων εκπαίδευσης, χρειαζόμαστε μεγαλύτερα “βήματα” (learning rate) για να μπορέσει το μοντέλο να προβλέψει όσο το δυνατόν πιο ορθά τα αποτελέσματα που περιμένουμε να προβλέψει. (τις συγκρίσεις των προϊόντων μας). Το αντίθετο συμβαίνει στην περίπτωση με τις περισσότερες επαναλήψεις εκπαίδευσης.

Στις 3 παρακάτω γραφικές παραστάσεις μπορούμε να δούμε μερικές ακόμη μετρήσεις όπου κάναμε για το μοντέλο μας. Συγκεκριμένα, στις 2 παρακάτω εικόνες βλέπουμε πως διαμορφώνονται τα διάφορα scores ανάλογα με τον διαφορετικό αριθμό επαναλήψεων.



Και τέλος παρακάτω μπορούμε να δούμε πως συμπεριφέρεται ο χρόνος εκτέλεσης του προγράμματός μας, ανάλογα με τον αριθμό επαναλήψεων εκμάθησης.



Μέρος 3ο

Στο τρίτο μέρος της εργασίας έπρεπε:

1. Να βελτιώσουμε τα BoW και TF_IDF ώστε να χρησιμοποιούν μόνο τις σημαντικότερες λέξεις των κειμένων(έχει υλοποιηθεί από τη 2η εργασία).
2. Να δημιουργήσουμε μία οντότητα Job_Scheduler η οποία με τις κατάλληλες συναρτήσεις:
    a. Να παραλληλοποιεί τον υπολογισμό του gradient κατά το training του μοντέλου (mini batch gradient descent).
    b. Να παραλληλοποιεί το testing του μοντέλου.
3. Να συντάξουμε την εν λόγω αναφορά του project.

1. Η απαιτούμενη βελτίωση είχε υλοποιηθεί ήδη από την 2η Εργασία.

2-3. Η παραλληλοποίηση γίνεται ως εξής:

- ➢ Δημιουργείται ο scheduler
- ➢ Ο scheduler βάζει τα jobs που πρέπει να γίνουν σε μία ουρά(αν το training set μας έχει 10000 στοιχεία και κάθε batch αποτελείται από 500 στοιχεία, φτιάχνει 20 training jobs.
- ➢ Ο scheduler δημιουργει threads πλήθους *NUM_OF_THREADS* , τα οποία με χρήση mutex προσπαθούν να πάρουν τα jobs από την ουρά.
- ➢ Όταν ένα thread πάρει ένα job (training ή testing) το εκτελεί και αλλάζει κάποιες glodbal variables (ανάλογα με το είδος του job). Έπειτα προσπαθεί πάλι να πάρει το επόμενο job. Όταν η ουρά είναι άδεια, το thread τερματίζει.
- ➢ Όταν τερματίσουν όλα τα threads (με την pthread_join()) ο scheduler επεξεργάζεται τις αντιστοιχες global variables και υπολογίζει τα ζητούμενα (είτε το μέσο gradient για το training, είτε τα true_positives, true_negatives κλπ gia to testing).

- ➢ Στην περίπτωση του testing, δεν χρειάζεται να δημιουργήσουμε καινούργια jobs αφού το test_set έχει εξεταστεί ολόκληρο. Στο training πάλι, η διαδικασία επαναλαμβάνεται ξανά, για το επόμενο iteration του gradient descent.

Σημείωση:
Τα νήματα είναι προγραμματισμένα έτσι ώστε να μην ξέρουν πριν διαβάσουν το job τι είδους εργασία θα εκτελέσουν, όπως ζητείται στην εκφώνηση.

Η struct job:

```
Typedef struct job{
        int id;                 // id του job
        int type;               //τύπος του job(0=train, 1=test)
        int batch_size;         //μέγεθος των batches, μόνο για training
        double bias;            //bias του μοντέλου, και train και test
        Point4d *weights;       //current_weights-train, optimal_weights-test
        Matrix* array;          //training set για train, testing set για test
}job;
```

Η συνάρτηση των threads:

```
void *ThreadJob(void *sch)
```

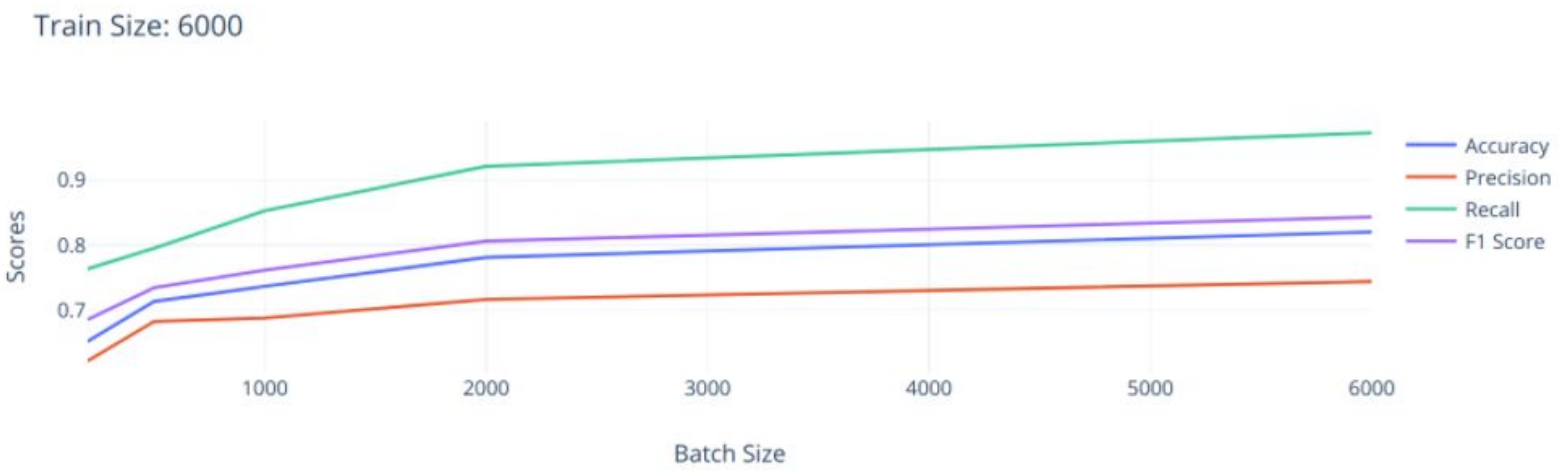
Η ThreadJob είναι στην ουσία ένα while loop (χωρίς busy waiting) που παίρνει jobs από την ουρά και τα εκτελεί. Όταν η ουρά αδειάσει, το thread τερματίζει. Σαν όρισμα δέχεται τον job_scheduler. Δεν επιστρέφει κάτι, αλλά αλλάζει τα απαραίτητα global variables που χρησιμοποιούνται για το train και το test.

Συναρτήσεις/δομές που υλοποιούν το τρίτο μέρος της εργασίας:

```
typedef struct job
{

  int id;
  int type;
  int batch_size;
  double bias;
  Point4d *weights;

  Matrix *array;
} job;

class Job_Scheduler
{
private:
public:
  int num_of_threads;
  queuestruct *queue_struct;
  pthread_t *threads;
  pthread_mutex_t *dequeue_mutex;

  Job_Scheduler(int);
  void submit_job(job *job);
  void execute_all_jobs();
};

Point5d train_model(LogisticRegression *, Matrix *, double);
void test_model(Matrix *, double, Point4d);
```

Η train_model() και η test_model() εκτελούν την εκπαίδευση και αξιολόγηση του μοντέλου αντίστοιχα, με τη χρήση παραλληλίας(δημιουργούν την ουρά από Jobs, και έπειτα threads που τρέχουν την ThreadJob() .

Scores ώς προς batch_size:



Όπως περιμέναμε, με την αύξηση του batch_size βελτιώνεται η ικανότητα πρόβλεψης του μοντέλου. Για batch_size = train_size πρέπει τα αποτελέσματα να είναι ίδια με το συνολικό gradient descent (πρακτικά ένα thread θα εκτελέσει την εκπαίδευση χρησιμοποιώντας όλο το training set).

Συμπεράσματα

Το μάθημα αυτό όντας ουσιαστικά ένα μεγάλο, επαγγελματικού επιπέδου, ομαδικό project, αποσκοπεί στην κτήση γνώσης υλοποίησης μιας μεγάλης εφαρμογής εργασιακού περιβάλλοντος σε ομαδικό επίπεδο.
Συγκεκριμένα, μάθαμε να λειτουργούμε σαν ομάδα γράφοντας κώδικα όπου όλοι μαζί ανα πάσα στιγμή μοιραζόμασταν χωρίς να επικοινωνούμε συνεχώς μέσω του εργαλείου version control GIT, στην πλατφόρμα GitHub. Μάθαμε να αξιοποιούμε σωστά και αποτελεσματικά τις δομές δεδομένων τις οποίες έχουμε διδαχθεί σε προηγούμενες χρονιές για την κατάλληλη διαχείριση και αποθήκευση μεγάλων datasets καθώς και να υλοποιούμε οτιδήποτε χρειαζόμαστε χωρίς την χρήση εξωτερικών βιβλιοθηκών το οποίο πραγματικά μας εμβαθύνει στην πλήρη κατανόηση αυτού του οποίου θέλουμε να χρησιμοποιήσουμε για την όποια περίπτωση, όπως για παράδειγμα η υλοποίηση του JSON parser που είδαμε στο μέρος 1. Αποκτήσαμε πολύ χρήσιμες γνώσεις μηχανικής μάθησης, καθώς υλοποιήθηκαν χωρίς χρήση εξωτερικών βιβλιοθηκών βασικοί αλγόριθμοι μηχανικής μάθησης όπως αυτοί περιγράφονται ανωτέρω καθώς και μέσω χρήσης πολυνηματισμού μάθαμε πως να κάνουμε τα προγράμματά μας να τρέχουν παράλληλα μειώνοντας έτσι σε μεγάλο βαθμό τον τελικό χρόνο εκτέλεσης, πράγμα πρακτικά χρήσιμο σε ένα μεγάλο project.

Τέλος, εδώ πρέπει να αναφερθεί ότι ένα αρκετά σημαντικό κομμάτι ενός Project είναι το Unit Testing. Το Unit Testing είναι ο βασικός έλεγχος που πρέπει να κάνει ένας προγραμματιστής πριν παραδώσει το έργο. Τόσο σε ακαδημαΪκό επίπεδο όσο και σε επαγγελματικό, αυτό το task είναι απαραίτητο.
Το Unit Testing της εργασίας αυτής έχει χωριστεί, ουσιαστικά, σε 2 κομμάτια. Στο πρώτο και σε ένα κομμάτι του δεύτερου μέρους χρειάστηκε να ελεγχθούν οι δομές που χρησιμοποιήσαμε (δέντρα, λίστες) καθώς και όλες οι συναρτήσεις που υλοποιήσαμε για να ολοκληρωθεί επιτυχώς το έργο. Κάποιες από τις βασικές συναρτήσεις είναι οι read_json, add_positive_relations, add_negative_relations, insert_entry, print_all_relations, create_bow_tree, create_train_set_bow κ.α.

Όλες οι παραπάνω έχουν υλοποιηθεί σε C γλώσσα, οπότε αποφασίσαμε να τις ελέγξουμε με την acutest.h. Στο υπόλοιπο κομμάτι του δεύτερου και σε όλο το τρίτο μέρος χρειάστηκε να ελεγχθεί το μοντέλο machine learning και όποια παραπάνω συνάρτηση χρειάστηκε για την παραλληλία του. Αυτά τα κομμάτια έχουν υλοποιηθεί σε C++ και αυτό είχε ως αποτέλεσμα να προτιμήσουμε διαφορετικό framework για το Unit Testing. Έτσι, χρησιμοποιήσαμε τη gtest.h (Google Tests).
Μέσα από το Unit Testing, συνειδητοποιήσαμε ότι ο κώδικάς μας είχε κάποια μικρά λαθάκια που άλλαζαν εντελώς το τελικό αποτέλεσμα. Ευτυχώς, αυτά τα λάθη βρέθηκαν έγκαιρα και επιλύθηκαν πριν το τέλος της προθεσμίας του κάθε μέρους.
Συμπερασματικά, είναι υποχρέωση του κάθε προγραμματιστή, πριν παραδώσει για SIT - UAT το έργο που έχει κληθεί να ολοκληρώσει, να έχει τρέξει ένα βασικό Unit Testing του κώδικά του. Είναι ο μόνος τρόπος για την αποφυγή βασικών λαθών.