Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Κ23Α - ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

Project - Entity resolution

Μέλη:

- Θεόδωρος Χατζηιωαννίδης - 1115201600197
- Βασίλειος Πουλόπουλος - 1115201600141
- Κωνσταντίνος Χατζόπουλος - 1115201300202

Περιεχόμενα

Διαδικαστικά

Κλήση Προγραμμάτων

Για να εκτελεστούν τα πρόγραμμα από τερματικό τρέχουμε το script 'build_and_run.sh'.

Τα ορίσματα που δέχεται το εκτελέσιμο για το training είναι τα εξής:

- -dir { dataset X path }
- -csv { dataset W path }
- -sw { stopwords file path }
- -m { tfidf | bow }
- -ex {resources path }

Τα ορίσματα που δέχεται το εκτελέσιμο για το predicting είναι τα εξής:

- -dir {user dataset X path}
- -csv {user dataset W path}
- -vocabulary {vocabulary path}
- -model {model path}

Ενδεικτική εκτέλεση προγράμματος Training

```
./project \
-dir Datasets/camera_specs/2013_camera_specs \
-csv Datasets/sigmod_large_labelled_dataset.csv \
-sw resources/unwanted-words.txt \
-m bow \
-ex resources
```

Ενδεικτική εκτέλεση προγράμματος Predicting

```
./user -dir resources/user_json_files \
```

-csv resources/user_dataset.csv \ -vocabulary resources/vocabulary.csv \ -model resources/model.csv

Έξοδος προγράμματος Training

predictions και **f1 score** για το validation set

Έξοδος προγράμματος Testing

predicitons για το dataset που δίνει ο χρήστης.

Σημείωση: Για να περάσουμε το model και το vocabulary από το ένα πρόγραμμα στο άλλο, τα γράψαμε σε αρχεία.

Εισαγωγή

Στόχος της άσκησης είναι να δημιουργήσουμε κλίκες γράφων από αγγελίες με τα ίδια προϊόντα σε διάφορα sites.

Αφού δημιουργηθούν οι κλίκες, δημιουργούνται και οι αρνητικές συσχετίσεις μεταξύ των κλικών (κάθε κλίκα αναγνωρίζει ποιες κλίκες είναι διαφορετικές από αυτή).

Στη συνέχεια δημιουργούνται όλα τα ζευγάρια από **similar** και **different** αγγελίες μέσω των κλικών και χρησιμοποιούνται ως δεδομένα για να εκπαιδευτεί ένα μοντέλο το οποίο θα μαντεύει αν τα ζευγάρια που δέχεται είναι similar ή different.

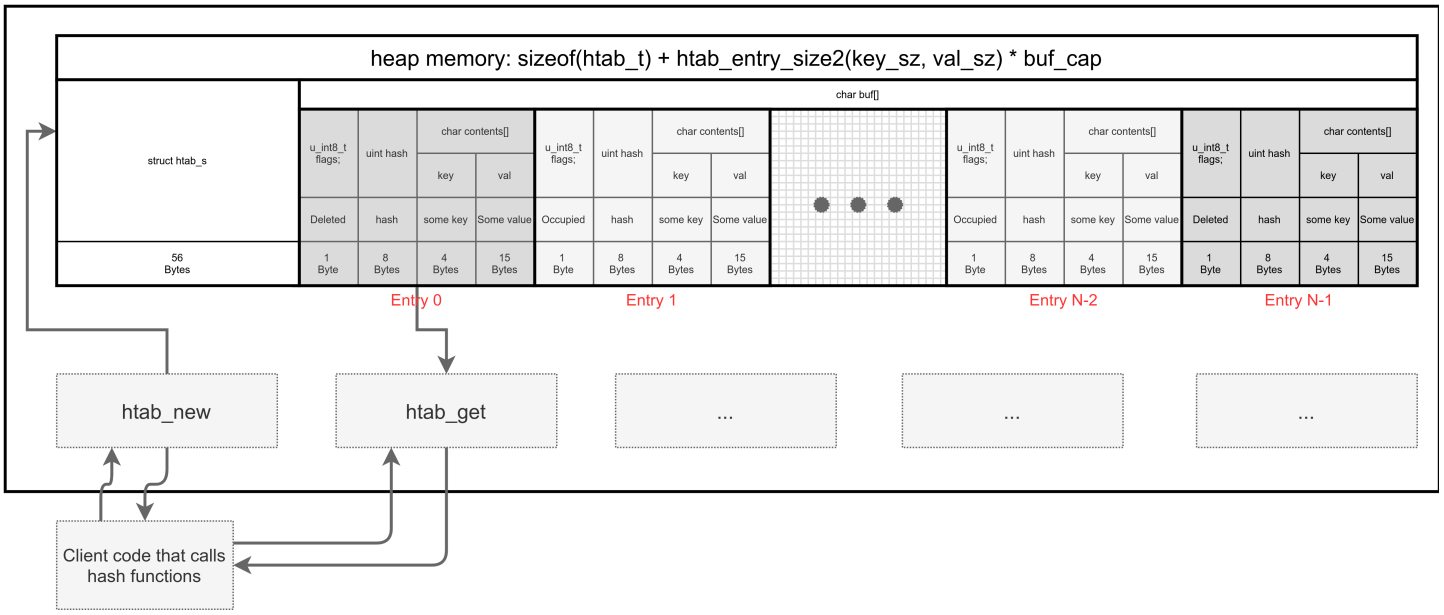
Η διαδικασία προετοιμασίας της εκπαίδευσης, καθώς και του testing, παραλληλοποιείται μέσω ενός **job scheduler**.

Για την ολοκλήρωση του project χρειάστηκε να υλοποιήσουμε τα παρακάτω **modules**:

- **hash table**
- **dictionary**
- **hash set**
- **queue**
- **spec to specs**
- **generic list**
- **job scheduler**
- **json parser**
- **tokenizer**
- **unique rand**
- **machine learning**
- **logistic regration**
- **custom semaphore**

Επιπλέον, για τον έλεγχο ορθότητας του κώδικα, χρησιμοποιήσαμε τη βιβλιοθήκη **acutest.h** όπου δέχεται μια λίστα από κατάλληλες συναρτήσεις (**tests**) που γράψαμε.

Hash table



Αναπαράσταση της δομής του hash table:

```
typedef  struct htab_s {
    ht_hash_func h;
    ht_cmp_func cmp;
    ht_key_cpy_func keycpy;
    size_t key_sz;
    size_t val_sz;
    ulong buf_cap;
    ulong buf_load;
    char buf[];
} htab_t;
```

Για την υλοποίηση του hash table, χρησιμοποιήσαμε generic **open addressing hash table** με random probing.
Πιο συγκεκριμένα, δεσμεύουμε ένα κομμάτι μνήμης το οποίο χωρίζεται σε μικροτέρα ίσα μέρη στα οποία αποθηκεύονται οι πληροφορίες του κάθε **spec**.

Αφού χρησιμοποιούμε random probing το κάθε spec ανάλογα με το key του, αν δε χωράει στην πρώτη θέση που θα υπολογιστεί μέσω του hash function, θα ψάξει κάποια "τυχαία" θέση για να αποθηκευθεί. Το seed της random συνάρτησης είναι σταθερό, οπότε πάντα μπορούμε να υπολογίσουμε σε ποιο σημείο της μνήμης βρίσκονται τα εκάστοτε δεδομένα.

Αν η μνήμη που έχουμε δεσμεύσει αρχίσει να γεμίζει θα κάνουμε **Rehashing**. Όταν θέλουμε να εισάγουμε νέα δεδομένα αλλά η πληρότητα του είναι **70%**, δεσμεύουμε τη διπλάσια μνήμη από πριν, ξαναμοιράζουμε τα δεδομένα στην καινούργια, μεγαλύτερη μνήμη και εισάγουμε τα νέα δεδομένα.

Generic List

Χρησιμοποιήσαμε μία **generic list** όπου δηλώνεται με τη χρήση του παρακάτω macro:

Αναπαράσταση της δομής του κόμβου:

```
#define  LISTOF(TYPE)   \
    struct {            \
    void *next;         \
    TYPE data;          \
}
```

Η λίστα αποτελείται από απλούς κόμβους δεδομένων `TYPE data;` οι οποίοι περιέχουν τα ίδια τα δεδομένα καθώς και έναν δείκτη στον επόμενο κόμβο. Είναι στην ουσία ένας **wrapper** ο οποίος επικαλύπτει τα δεδομένα, προσθέτοντάς τους έναν δείκτη `void *next;` για τα επόμενα δεδομένα.

Λόγω της σειράς που ορίζεται το πεδίο του επόμενου κόμβου (`void *next;`) στον wrapper, είναι εύκολο να αποκτηθεί πρόσβαση στον επόμενο κόμβο με τη χρήση του παρακάτω macro:

```
#define  NEXT(x) (*((void **)x))
```

όπου επιστρέφει έναν δείκτη στον επόμενο κόμβο πιο άμεσα.

JSON Parser

Ο json parser ακολουθεί τη γραμματική του json.org .

Πιο συγκεκριμένα, κάθε οντότητα του json περιγράφεται από την παρακάτω δομή:

```
typedef struct {
    const json_type type;
    char data[];
} JSON_ENTITY;
```

όπου το **data[]** περιέχει την ίδια την οντότητα, ενώ το **type**, περιγράφει τον τύπο της οντότητας, και μπορεί να λάβει οποιαδήποτε από τις τιμές του παρακάτω **enum**:

```
typedef enum {
    JSON_OBJ,
    JSON_ARRAY,
    JSON_NUM,
    JSON_BOOL,
    JSON_STRING,
    JSON_NULL
} json_type;
```

Spec to specs

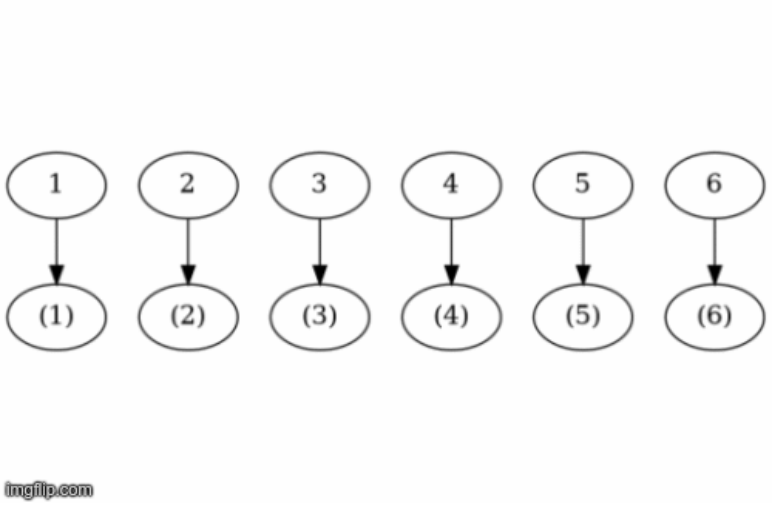
Ο τρόπος με τον οποίο συνδέουμε τα στοιχεία των κλικών μεταξύ τους, είναι μέσω μίας δενδρικής αναπαράστασης. Το πρώτο spec που θα μπει στην κλίκα είναι ο πατέρας, ενώ τα υπόλοιπα είναι τα παιδιά του. Αν δύο κλίκες γίνουν **merge**, τότε ο πατέρας της δεύτερης κλίκας αποθηκεύεται στον παππού της πρώτης. Με αυτό τον τρόπο κρατάμε το δένδρο σε χαμηλά επίπεδα.

Το **spec to specs** είναι η δομή που χρησιμοποιούμε για το ζητούμενο της άσκησης. Στην ουσία είναι ένα hash table, στο οποίο το κάθε στοιχείο που αποθηκεύεται είναι της μορφής:

```
struct SpecEntry_s {
      char *id;
      char *parent;
      StrList *similar, *similar_tail, *different, *different_tail;
      ulong similar_len, different_len;
      bool printed;
};
```

Ο **parent** είναι ο πατέρας του spec (αν δεν είναι σε κλίκα τότε έχει για πατέρα τον εαυτό του). Η **similar** είναι μία λίστα στην οποία, αν το spec είναι parent κάποιας κλίκας αποθηκεύονται τα ids όλης της κλίκας, σε αντίθετη περίπτωση της ανατίθεται η τιμή NULL, ενώ το **similar_tail** είναι δείκτης στο τελευταίο στοιχείο της similar. Το similar_len είναι το μέγεθος της similar.

Για τον υπολογισμό των διαφορετικών κλικών (**different**) βάλαμε άλλη μία λίστα στην οποία βρίσκονται οι root κάθε κλίκας που είναι διαφορετική. Αν για παράδειγμα είχαμε τη συσχέτιση "A, B, 0", ο root του A θα συμπεριλάμβανε το root του B στη different λίστα του, όπως και ο root του B θα συμπεριλάμβανε το root του A στη δική του different λίστα.



Αναπαράσταση εισαγωγής κλίκας: imgflip.com

Unique rand

Για την επιλογή τυχαίων και ταυτόχρονα μοναδικών ακέραιων τιμών στα σύνολα (**train**, **test**, **validation**) υπήρξε η ανάγκη να υλοποιηθεί ένα module (`urand`) που θα μας επιστρέφει τυχαίες τιμές καλώντας την `ur_get` εξασφαλίζοντας έτσι, ότι τις ίδιες αυτές τιμές που πήραμε δε θα τις ξαναπάρουμε σε μεταγενέστερη κλήση της συνάρτησης.

Αναπαράσταση της δομής unique_rand:

```
struct unique_rand {
    int min;
    int max;
    int *values;
    int length;
};
```

Δημιουργία

Κατά την δημιουργία αυτού του module καλώντας τη συνάρτηση `ur_create` , είναι απαραίτητο να δοθεί σε αυτήν μια ελάχιστη (min) και μια μέγιστη (max) τιμή ώστε να καθοριστεί το εύρος των τυχαίων τιμών που θα επιστρέφει η `ur_get` . Επομένως, με το εύρος τιμών γίνεται υπολογισμός του πλήθους αυτών, δεσμεύεται δυναμικά ένας πίνακας ακέραιων μεγέθους `max - min + 1` και αμέσως μετά αρχικοποιείται με τιμές που ξεκινούν από `min` .

Λειτουργικότητα

Αφού έχει δημιουργηθεί με επιτυχία το `URand *ur` αντικείμενο τότε κάθε φορά που καλείται η `ur_get` επιλέγεται τυχαία μια θέση από τον πίνακα `values` για να επιστραφεί το περιεχόμενό της στον χρήστη, η θέση αυτή γίνεται **swap** με την αντίστοιχη θέση που δείχνει εκείνη τη στιγμή η τιμή `ur->length` δηλαδή προς το τέλος. Αυτό έχει ως αποτέλεσμα, οι τιμές εκείνες που έχει πάρει ο χρήστης να μετακινούνται προς το τέλος του πίνακα `values` καθώς μειώνεται η μεταβλητή `ur->length` .

Μηχανική Μάθηση

Αρχικά έπρεπε να δημιουργηθεί το **vocabulary** σύμφωνα με το οποίο θα φτιάχνονταν τα διανύσματα για να γίνει το **training** και το **testing**. Έτσι δημιουργήσαμε το vocabulary με όλες τις διαφορετικές λέξεις των JSON αρχείων.

Για να μειωθούν οι διαστάσεις δε συμπεριλάβαμε **stopwords**, **punctuations** και **αριθμούς**, όπως και κάποιες λέξεις που θεωρούμε πως δε φέρουν χρήσιμη πληροφορία αφού υπάρχουν σε σχεδόν όλα τα αρχεία (πχ το "mm"). Στη συνέχεια αν ο χρήστης έχει δώσει την επιλογή **tfidf**, αφαιρούμε από το vocabulary τις λέξεις με μεγάλο **idf**. Για να γίνει η διαδικασία επιλογής λέξεων δοκιμάσαμε τις παρακάτω μεθόδους:

- Η πρώτη μέθοδος ήταν να χρησιμοποιήσουμε συναρτήσεις όπως η **strtok** και **strcat**.
- Η δεύτερη ήταν να τροποποιήσουμε τον **tokenizer** που έχουμε υλοποιήσει για τον **json parser** και να υλοποιήσουμε την συνάρτηση **tokenizer_nlp_sw()** που αναλαμβάνει να επιστρέψει μόνο τις λέξεις που πληρούν τα κριτήριά αγνοώντας τις

υπόλοιπες.

Παρατηρήσαμε πως η δεύτερη μέθοδος ήταν πιο γρήγορη οπότε και την προτιμήσαμε για το τελικό μας πρόγραμμα. Παρ' όλα αυτά υπάρχουν και οι δύο υλοποιήσεις καθώς και τα test τους στο repository.

Στη συνέχεια φτιάξαμε το **διάνυσμα** (bow ή tfidf ανάλογα με την προτίμηση του χρήστη) για το κάθε JSON και τα αποθηκεύσαμε σε ένα hash table για να τα έχουμε έτοιμα όταν τα χρειαστούμε.

Για να δημιουργηθεί το **dataset** για το **train**, **test** και **validation** ακολουθήθηκε η εξής διαδικασία:

Αρχικά, δημιουργήθηκαν όλα τα similar και different ζευγάρια μέσω των κλικών. Αυτό ήταν το συνολικό dataset. Στη συνέχεια κρατήσαμε τυχαία το **50% των similar** και το **50% των different** και τα ενώσαμε σε ένα πίνακα. Αυτός ο πίνακας αποτελεί το train set. Κάναμε την ίδια διαδικασία με **25% για το test set** και τα υπόλοιπα **25% για το validation set**. Έτσι, είχαμε τα τρία sets που χρειαζόμασταν.

****Για να δημιουργήσουμε το διάνυσμα κάθε ζευγαριού, χρησιμοποιούμε την απόλυτη διαφορά των διανυσμάτων json1-json2.*

Εκπαίδευση μοντέλου

Για την εκπαίδευση υλοποιήσαμε **mini batch gradient decent**. Σε κάθε **epoch** αρχικά γίνεται training για το training set και στη συνέχεια γίνεται predict για το test set ώστε να υπολογίζουμε το μέσο**loss** σε κάθε **epoch**, με το σκεπτικό αν παρατηρήσουμε συνεχή αύξηση του μέσου loss για 5 συνεχόμενα epochs, να κρατήσουμε το μοντέλο που είχε υπολογιστεί πριν 5 epochs και να σταματήσουμε τη διαδικασία του **training**. Παρατηρήσαμε όμως πως μέχρι τα 2000 **epochs** που το τεστάραμε ακόμα δεν έγινε break.

Στο τέλος του training κάνουμε **predict** για το validation set, τυπώνουμε τα predictions (με κόκκινο χρώμα τα λανθασμένα και με πράσινο τα σωστά) και στη συνέχεια υπολογίζουμε το **F1 score**

Εξαγωγή μοντέλου και λεξιλογίου σε αρχεία

Έχει υλοποιηθεί και δεύτερη main (**user.c**) η οποία κάνει μόνο **predict** ότι δώσει ο χρήστης σαν είσοδο. Χρησιμοποιώντας το **vocabulary** και το **model** που έχουν γίνει export από την πρώτη main. Για το λόγο αυτό το vocabulary και το τελικό model γίνονται export σε αρχεία ώστε να δοθούν ως είσοδο σε αυτήν.

Επαναληπτική εκμάθηση

Η διαδικασία που ακολουθούσαμε ήταν να χρησιμοποιήσουμε όλο το dataset (**train + test + val sets**) για αρχικό training set. Για threshold είχαμε αρχικά **0.0** ωστε να προστεθούν στο dataset μόνο όσα ζευγάρια γίνονταν predict me **0** ή **1** και σε κάθε επανάληψη το threshold αυξανόταν κατά **0.01**. Για όλα τα υπόλοιπα πιθανά ζευγάρια (δηλαδή όλα τα ζευγάρια απο specs για τα οποία δεν ξέρουμε τη σχέση τους, κάναμε **predict**. Αν το μοντέλο μας ήταν αρκετά "σίγουρο", δηλαδή αν ίσχυε η συνθήκη " `if (p(x,b) < threshold) or (p(x,b) > 1 - threshold)` ", προσθέταμε όλες τις συσχετίσεις που προέκυπταν από αυτό το prediction στο **training set** ώστε στην επόμενη επανάληψη να ληφθούν και αυτά στο train. Συνεχίζαμε τη διαδικασία για όσο το **threshold** ήταν **< 0,5**.

Για να ξέρουμε ποια ζευγάρια βρίσκονται στο **training set** χρησιμοποιοαμε ένα hash table του οποίου τα κλειδιά είναι τα ονόματα των ζευγαριών του **training set** άλλα και τα αντίστροφά τους, δηλαδή αν το ζευγάρι **A,B** μπει στο training set, στο hash table μπαίνει και το A,B και το B,A ώστε να μην μπει και με τους δύο τρόπους το ίδιο ζευγάρι.

Παρατηρήσαμε πως από την πρώτη επανάληψη της επαναληπτικής μάθησης, επειδή ο αριθμός των ζευγαριών που προστίθενται στο **training set** ήταν πολύ μεγάλος ο χώρος της μνήμης γέμιζε με αποτέλεσμα να τερματίζει το πρόγραμμα. Για αυτό το λόγο, δοκιμάσαμε να γράφουμε σε αρχείο το **training set** ώστε να μην γεμίζει η μνήμη. Παρόλα αυτά και πάλι γέμιζε η μνήμη λόγω του hash table.

Για αυτό το λόγο ακολουθήσαμε δύο προσεγγίσεις ώστε να μην μπαίνουν τόσα πολλά ζευγάρια στο dataset. Αρχικά σταματήσαμε να δεχόμαστε τα ζευγάρια των οποίων έστω το ένα από τα 2 specs ηταν σε κλίκα. Με αυτό τον τρόπο ουσιαστικά οι ήδη υπάρχουσες κλίκες δεν μεγάλωναν, αλλά φτιάχνονταν καινούργιες. Επίσης, βάλαμε ανώτατο όριο, να δέχεται μέχρι **500.000** καινούργια ζευγάρια.

Γενικές παρατηρήσεις για την επαναληπτική μάθηση

Μέσω της διαγραφής των λέξεων με μεγάλο **idf** κρατήσαμε τις καλύτερες **911** και όχι **1000** γιατί το αμέσως μεγαλύτερο idf κρατάει άλλες **161** λέξεις που σημαίνει πως ξεπερνάμε τις **1000**.

Παρατηρήσαμε ότι για **2000 epochs**, **0,0001 learning rate** και **batch size 2000** παίρνουμε αρκετά καλό **f1 score** της τάξης του **0,93**.

Επίσης, την πρώτη φορά που γίνεται **train** το μοντέλο, πριν την επαναληπτική μάθηση, κάθε epoch ολοκληρώνεται σε περίπου **0,4** δευτερόλεπτα. Στην επαναληπτική μάθηση, όσο μεγαλώνει το dataset, κάθε epoch γίνεται και πιο αργό, αφού πρέπει να γίνουν train περισσότερα ζευγάρια.

Παραλληλοποίηση

Για τη γρηγορότερη εκπαίδευση του μοντέλου υλοποιήθηκε ένας **Job Scheduler** που αναλαμβάνει να δημιουργήσει νέα νήματα (**workers**). Κάθε worker τρέχει πολλαπλά Jobs σειριακά τα οποία εξάγονται από μια κοινόχρηστη ουρά το ένα μετά το άλλο προσθέτοντας έτσι παραλληλία.

Βασικές δομές

Αναπαράσταση της δομής job:

```
struct job {
    long long int job_id;
    void *(*start_routine)(void *);
    int args_count;
    Argument *args;
    void *return_val;
    bool complete;
    /* sync */
    sem_t sem_complete;
};
```

Κάθε νέο **job** που δημιουργείται μέσω της συνάρτησής **js_create_job()** έχει τα παρακάτω χαρακτηριστικά:

- μοναδικό id (**job_id**)
- δείκτη σε συνάρτηση (**start_routine**) για τη ρουτίνα εκτέλεσης
- πίνακα με τα ορίσματα της ρουτίνας (**args**)
- τιμή επιστροφής της ρουτίνας (**return_val**)
- flag (**complete**) που υποδηλώνει αν το job ολοκληρώθηκε ή όχι
- semaphore (**sem_complete**) ώστε αν κάποιο νήμα ζητήσει να δει την τιμή επιστροφής του, να είναι εφικτό να μπλοκάρει μέχρι αυτό να ολοκληρωθεί.

Αναπαράσταση της δομής job_scheduler:

```
struct job_scheduler {
     uint execution_threads;
     pthread_t *tids;
     Queue waiting_queue;
     Queue running_queue;
     bool working;
     bool exit;
     int ready;
     /* sync */
     pthread_cond_t condition_wake_up;
     pthread_cond_t condition_wake_up_submitter;
     pthread_mutex_t mutex;
     pthread_mutex_t mutex_submitter;
     sem_t_ *sem_barrier;
};
```

Η δομή του Job Scheduler έχει τα παρακάτω χαρακτηριστικά:

- Αριθμός διαθέσιμων νημάτων (**execution_threads**) για εκτέλεση εργασιών.
- Πίνακας με τα αναγνωριστικά των νημάτων (**tids**) που χρησιμοποιείται για την εύρεσή τους με σκοπό την ένωσή τους με το αρχικό νήμα κάνοντάς κλήση της συνάρτησής **pthread_join**.
- Ουρά αναμονής (**waiting_queue**) προς εκτέλεση εργασιών.
- Ουρά εργασιών που ξεκίνησαν να τρέχουν (**running_queue**) η οποία χρησιμοποιείται για να τα συλλέξει με τη σωστή σειρά το master thread όταν θα έχουν ολοκληρωθεί όλα.
- Flag (**working**) που χρησιμοποιείται για να υποδηλώσει ένα ενεργό **work cycle** όταν έχει την τιμή true και **barrier ready** όταν έχει την τιμή false.
- Flag (**exit**) που υποδηλώνει ότι το master thread αποφάσισε να μη χρησιμοποιήσει άλλο τον scheduler. Αν λάβει την τιμή true, τότε όταν το κάθε νήμα "ξυπνήσει" θα καλέσει την **pthread_exit**.
- Μετρητής έτοιμων νημάτων (**ready**) που δεν τρέχουν κάποια εργασία αυτή τη στιγμή.

Ενώ για τον συγχρονισμό:

- Mutex (**mutex**) που χρησιμοποιείται για τον αμοιβαίο αποκλεισμό των άλλων νημάτων στις περιπτώσεις που χρειάζεται.
- Condition variable (**condition_wake_up**) που χρησιμοποιείται για να μπλοκάρουν τα νήματα που είναι έτοιμα και δεν υπάρχει ακόμα διαθέσιμο job.
- Mutex αποστολέα (**mutex_submitter**) για τον αμοιβαίο αποκλεισμό των υπόλοιπων νημάτων που προσπαθούν εκείνη τη στιγμή ταυτόχρονα να βάλουν μία νέα εργασία στην ουρά.
- Condition variable (**condition_wake_up_submitter**) που χρησιμοποιείται για να μπλοκάρει μέχρι να υπάρξει κάποιο διαθέσιμο νήμα.
- Custom semaphore (**sem_barrier**) που χρησιμοποιείται για να ενημερώσει το master thread που έχει καλέσει τη **join_threads** ότι όλα τα νήματα είναι έτοιμα.

Αντιγραφή ορισμάτων στην ρουτίνα

Για την αντιγραφή των ορισμάτων που δίνει ο χρήστης από το master νήμα στη ρουτίνα του αντίστοιχου νήματος μέσω της συνάρτησης `submit_job` κάνουμε χρήση των Variadic arguments για να τα παίρνουμε με δυναμικό τρόπο χωρίς να είναι απαραίτητο να δημιουργηθεί από το χρήστη κάποια δομή από ορίσματα. Τέλος, τα αντιγράφουμε σε έναν πίνακα από `Argument` το μέγεθος του οποίου, αυξάνεται δυναμικά με realloc ανάλογα με τον αριθμό των ορισμάτων που δίνει ο χρήστης.

Αναπαράσταση της διαδικασίας αντιγραφής των ορισμάτων σε πίνακα (`submit_job`):

```
va_start(vargs, start_routine);
FOREACH_ARG(arg, vargs) {
    size_t type_sz = va_arg(vargs, size_t);
  (*job)->args = realloc((*job)->args, (i + 1) * sizeof(Argument));
  (*job)->args[i].arg = malloc(type_sz);
  assert((*job)->args[i].arg != NULL);
  memcpy((*job)->args[i].arg, arg, type_sz);
  (*job)->args[i].type_sz = type_sz;
  (*job)->args_count++;
};
va_end(vargs);
```

Για την αντίστροφη διαδικασία, όπου ο χρήστης θέλει να πάρει μέσα από τη ρουτίνα τα ορίσματα που έδωσε στην `submit_job` το μόνο που έχει να κάνει είναι να καλέσει την `js_get_args` και να δώσει σε αυτήν το job της ρουτίνας καθώς και τη διεύθυνση από τις μεταβλητές που θέλει να αρχικοποιήσει με τα αντίστοιχα ορίσματα. Τα ορίσματα πρέπει να τα δώσει με την ίδια σειρά που τα έγραψε και στη `submit_job` .

Αναπαράσταση της διαδικασίας αρχικοποίησης μεταβλητών :

```
js_get_args(job, &js, &sum, &computations, &mtx, NULL);
```

Αναπαράσταση της διαδικασίας αντιγραφής των ορισμάτων από τον πίνακα ορισμάτων στην αντίστοιχη μεταβλητή `arg` :

```
void js_get_args(Job job, ...) {
    va_list vargs;
```

```
  va_start(vargs, job);
  FOREACH_ARG(arg, vargs) {
        memcpy(arg, job->args[i].arg, job->args[i].type_sz);
  };
  va_end(vargs);
}
```

Δεξαμενή νημάτων

Ο Job scheduler αναλαμβάνει να τρέξει εργασίες (jobs) σε διαφορετικά νήματα με **FIFO** (First In First Out) σειρά και αυτό το επιτυγχάνει με μια ουρά αναμονής (waiting_queue).

Κύκλος εκτέλεσης εργασιών

Αναπαράσταση της συνάρτησης thread:

```
void *thread(JobScheduler js) {
    int jobs_count = 0;
    while (true) {
        LOCK_;
        while ((!js->working && !js->exit) || (!queue_size(js->waiting_queue) && !js->exit)) {
            js->ready++;
            if (js->ready == js->execution_threads) {
                js->working = false;
            }
            pthread_cond_signal(&js->condition_wake_up_submitter);
            NOTIFY_BARRIER_;
            WAIT_;
            js->ready--;
        }
        if (js->exit && !queue_size(js->waiting_queue)) {
            UNLOCK_;
            EXIT_;
        }
        Job job = NULL;
        if (queue_dequeue(js->waiting_queue, &job, false)) {
            jobs_count++;
            queue_enqueue(js->running_queue, &job, false);
            queue_unblock_enqueue(js->waiting_queue);
            UNLOCK_;
            RUN_ROUTINE_;
            job->complete = true;
            NOTIFY_JOB_COMPLETE_;
            continue;
        }
        UNLOCK_;
    }
}
```

Όταν ξεκινάει να λειτουργεί ένα νήμα, εισέρχεται σε έναν ατέρμον βρόγχο, κλειδώνει την πρόσβαση από τα υπόλοιπα νήματα με τη χρήση `LOCK_` (ή αντίστοιχα περιμένει μέχρι η πρόσβαση για αυτό να είναι εφικτή) και κάνει τους παρακάτω ελέγχους ώστε να αποφασίσει αν πρέπει να μπει σε κατάσταση `WAIT_` :

- `(!js->working && !js->exit)` : Αν δεν έχει ενεργοποιηθεί ακόμα το flag **working** και **exit** τότε σημαίνει ότι δεν έχει ληφθεί ακόμα κάποιο σήμα εκκίνησης ή τερματισμού.
- `(!queue_size(js->waiting_queue) && !js->exit)` : Αν η ουρά δεν έχει jobs και δεν έχει ληφθεί κάποιο σήμα τερματισμού.

Όταν "ξυπνήσει" το νήμα λαμβάνοντας κάποιο `BROADCAST_WAKEUP_` ή `SIGNAL_WAKE_UP_` signal, θα ελέγξει εκ νέου αν ισχύουν οι δύο παραπάνω συνθήκες ή όχι (**busy waiting**) και θα πράξει αναλόγως.

Στην περίπτωση που δεν ισχύει το condition στο while: `(!js->working && !js->exit) || (!queue_size(js->waiting_queue) && !js->exit)` τότε θα ακολουθήσει ο έλεγχος: `js->exit && !queue_size(js->waiting_queue)` ο οποίος θα ελέγξει αν ο λόγος που έγινε **wake up** το νήμα είναι για να τερματίσει (ή όχι) κάνοντας `UNLOCK_` και καλώντας τη `pthread_exit` (`EXIT_`).

Στη συνέχεια, αν το νήμα δεν τερματίσει τότε οι λόγοι που "ξύπνησε" (**wake up**) είναι οι παρακάτω:

- έλαβε κάποιο signal `BROADCAST_WAKEUP_` από τη συνάρτηση `js_execute_all_jobs` που αναλαμβάνει να ξυπνήσει όλα τα νήματα για να ξεκινήσουν να τρέχουν τα jobs ή
- έλαβε κάποιο signal `SIGNAL_WAKEUP_` από τη συνάρτηση `js_submit_job`

Σε επόμενη φάση αν υπάρχουν στην ουρά αναμονής διαθέσιμες προς εκτέλεση εργασίες, εξάγεται από την ουρά η πρώτη διαθέσιμη, ο μετρητής των εργασιών (`jobs_count`) που έχει αναλάβει το συγκεκριμένο νήμα αυξάνεται κατά ένα και η εργασία εισέρχεται στη running queue, ενώ αμέσως μετά καλείται η `queue_unblock_enqueue` ώστε να ξεμπλοκάρει η waiting queue και να μπορέσει να χρησιμοποιηθεί από κάποιο άλλο νήμα. Κατόπιν γίνεται `UNLOCK_` για να μπορέσουν και άλλα νήματα να έχουν πρόσβαση στους κοινόχρηστους πόρους και εκτελείται η ρουτίνα (`RUN_ROUTINE_`) με όρισμα το ίδιο το job (`job->return_val = (job->start_routine)(job)`).

Όταν επιστρέψει η συνάρτηση της ρουτίνας, αποθηκεύεται η τιμή επιστροφής (**return_val**) στη δομή του job ώστε να είναι διαθέσιμη στον client κώδικα.

Στη συνέχεια ενημερώνεται το flag `job->complete` σε true και καλείται η `sem_post(&job->sem_complete)` (`NOTIFY_JOB_COMPLETE_`) με σκοπό να ενημερωθεί το master νήμα πως το job ολοκληρώθηκε. Ο λόγος που γίνεται αυτό είναι γιατί μπορεί εκείνη τη στιγμή να είχε μπλοκάρει στις συναρτήσεις `js_wait_job` ή και `js_get_return_val` αντίστοιχα.

Χρήση του Job Scheduler

Στην υλοποίηση μας για τη μηχανική μάθηση, οι συναρτήσεις train και test περιμένουν έναν buffer με όλα τα διανύσματα που θα χρησιμοποιηθούν για αυτές τις διαδικασίες. Το να γεμίσει αυτός ο buffer κάθε φορά που χρειαζόταν, έπαιρνε αρκετό χρόνο, οπότε η πρώτη διαδικασία που παραλληλίσαμε ήταν αυτή. Πιο συγκεκριμένα, κάθε job αναλάμβανε να βάλει ένα διάνυσμα στο σωστό σημείο του buffer. Με αυτό τον τρόπο επιτυγχάνουμε ταυτόχρονο γέμισμα του buffer.

Άλλο ένα σημείο στο οποίο παραλληλοποιήσαμε ήταν ο υπολογισμός των Deltas στη διαδικασία του training. Κάθε job αναλαμβάνει να υπολογίσει από ένα στοιχείο του Deltas array.

Τέλος, παραλληλοποιήσαμε τη διαδικασία του predict, έτσι ώστε κάθε job να αναλαμβάνει να εκτελέσει το prediction ενός ζευγαριού και να το τοποθετεί στο σωστό σημείο του πίνακα με τα συνολικά predictions.

Σκοπεύαμε αν προλαβαίναμε, για να παρατηρήσουμε αν θα υπάρξει διαφορά στο χρόνο εκτέλεσης του προγράμματος, να αλλάζαμε τη διαδικασία της παραλληλοποίησης και να δίναμε στα jobs περισσότερες απο μία επαναλήψεις ώστε να γίνονταν περισσότερες πράξεις σε κάθε job.

Unit tests

Κάθε module (**list**, **hash table**, **hset**, **job_scheduler**, **json_parser**, **ml**, **logreg**, **queue**, κτλ) έχει αντίστοιχα και τα δικά του unit tests τα οποία βρίσκονται σε διαφορετικά αρχεία το καθένα. Για την υλοποίησή τους, κάνουμε χρήση της βιβλιοθήκης Acutest. Όταν επιχειρούμε να κάνουμε complile (make), commit ή push εκτελούνται τα **unit tests** και αν τερματίσουν επιτυχώς, πραγματοποιείται με επιτυχία η αντίστοιχη διαδικασία.

Επιπλέον, τα test αυτά εκτελούνται remotely μέσω των **github actions (CI)**.

Documentation

Στον σύνδεσμο Documentation που βρίσκεται και στην πρώτη σελίδα του report βρίσκεται το **documentation** (**man pages**) των βιβλιοθηκών που έχουν υλοποιηθεί. Πρόσβαση επιτρέπεται μόνο στους **contributors** του project (δηλαδή την ομάδα μας και το βοηθό του τμήματός μας) και πρέπει να γίνει **authentication** μέσω **github**, ώστε να μην μπορεί κάποιος άλλος να αποκτήσει πρόσβαση.

Το Documentation υλοποιήθηκε μέσω του Doxygen. Τα man pages αυτά ανανεώνονται κάθε φορά που προστίθεται νέο commit στο master. Τα output αρχεία του **Doxygen** γίνονται push στο branch "**docs**" και απο εκεί ενημερώνεται το documentation site μας.