Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Ταξινόμηση πηγαίου κώδικα σε αλγορίθμους με τη χρήση Νευρωνικών Δικτύων.

Διπλωματική Εργασία

Ελευθέριος Καναβάκης

Αθήνα,

Σεπτέμβριος 2020

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Ταξινόμηση πηγαίου κώδικα σε αλγορίθμους με τη χρήση Νευρωνικών Δικτύων.

Διπλωματική Εργασία

Ελευθέριος Καναβάκης

Επιβλέπων καθηγητής: Γεώργιος Γκούμας

Εγκρίθηκε από την τριμελή επιτροπή στις 9 Σεπτεμβρίου 2020.

Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής, Ε.Μ.Π.	Επίκουρος Καθηγητής, Ε.Μ.Π.	Καθηγητής, Ε.Μ.Π.

Αθήνα,

Σεπτέμβριος 2020

Ελευθέριος Καναβάκης
Διπλωματούχος Εθνικού Μετσοβίου Πολυτεχνείου

Copyright @ Ελευθέριος Καναβάκης, 2020. Με επιφύλαξη παντός δικαιώματος. All rights reserved. Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσοβίου Πολυτεχνείου.

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι η μελέτη του προβλήματος της ταξινόμησης πηγαίου κώδικα με την χρήση νευρωνικών δικτύων. Πιο συγκεκριμένα, στο πρόβλημα αυτό ένα κομμάτι κώδικα ταξινομείται σε μια αλγοριθμική κλάση με βάση την λειτουργία που επιτελεί. Η προϋπάρχουσα έρευνα έχει δείξει πως τα νευρωνικά δίκτυα είναι ένας αποτελεσματικός τρόπος για την μοντελοποιήσει του πηγαίου κώδικα και την επίλυση του προβλήματος της ταξινόμησης αυτού. Βέβαια, παρόλο που τα αποτελέσματα της βιβλιογραφίας είναι ενθαρρυντικά παρατηρούνται περιορισμοί τόσο σχετικά με τα σύνολα των δεδομένων όσο και με τις τεχνικές προ-επεξεργασίας και τα μοντέλα μηχανικής μάθησης. Για τον σκοπό αυτό, δημιουργήσαμε ένα σύστημα το οποίο αρχικά κατασκευάζει ποιοτικά σύνολα δεδομένων (Datasets), τα οποία δεν είναι δέσμια προκαταλήψεων και θορύβου. Στην συνέχεια, με την βοήθεια μεταγλωττιστών (compilers) επεξεργάζεται τα σύνολα αυτά και τέλος, με χρήση νευρωνικών δικτύων τα ταξινομεί στην αλγοριθμική κλάση που ανήκουν. Στα πλαίσια της βελτιστοποίησης του παραπάνω συστήματος, μελετήσαμε τόσο διαφορετικές τεχνικές προ-επεξεργασίας όσο και διαφορετικά μοντέλα νευρωνικών δικτύων.

Λέξεις κλειδιά: Ταξινόμηση Πηγαίου Κώδικα; Μεταγλωττιστές (Compilers); Αφηρημένο Συντακτικό Δέντρο (AST); Τεχνικές προ-επεξεργασίας Πηγαίου Κώδικα; Αναδρομικά Νευρωνικά Δίκτυα (RNN); Long Short Term Memory (LSTM); Deep Averaging Network (DAN); Hierarchical Attention Network (HAN);

Abstract

The purpose of this dissertation is to study the problem of source code classification using neural networks. More specifically, in this problem, a piece of code is classified into an algorithmic class-based on the function it performs. Pre-existing research has shown that neural networks are an effective way of modeling source code and solving such classification problems. Although literature results are encouraging, there are limitations related not only to datasets and preprocessing techniques but also to machine learning models. To this end, we propose a system that initially builds quality datasets, which are free of biases and noise. It then uses compilers to process these sets and finally uses neural networks to classify them into an algorithmic class. In the context of optimizing the system above, we studied a variety of preprocessing techniques and machine learning models.

Keywords: Source code classification; Compilers; Abstract Syntax Tree (AST); Source code preprocessing; Recurrent neural network (RNN); Long Short Term Memory (LSTM); Deep Averaging Network (DAN); Hierarchical Attention Network (HAN);

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων (Cslab) της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου.

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή Γεώργιο Γκούμα, για την εμπιστοσύνη που μου έδειξε με την ανάθεση της συγκεκριμένης διπλωματικής εργασίας καθώς επίσης και τον υποψήφιο διδάκτορα Πέτρο Αναστασιάδη για την εξαιρετική συνεργασία και καθοδήγηση που μου παρείχε κατά την εκπόνηση της παρούσας εργασίας.

Από καρδίας θα ήθελα να ευχαριστήσω τους γονείς μου, Γιώργο και Άννα, για την αγάπη, την υπομονή και την συμπαράστασή τους σε κάθε στάδιο της ζωής μου .

Ευχαριστώ από καρδίας την κοπέλα μου, Δανάη, για την αγάπη, την υπομονή και την στήριξη της.

Τέλος, από καρδιάς ευχαριστώ τους φίλους μου, παλιούς και νέους, για τη στήριξη και τη συνεργασία.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Η ανάγκη για συστήματα κατανόησης κώδικα

Στις μέρες μας, η τεχνολογία και οι εφαρμογές της διαδραματίζουν έναν ιδιαίτερα σημαντικό ρόλο στην καθημερινότητα των ανθρώπων. Είναι χαρακτηριστικό, πως η πλειοψηφία των ανθρώπων χρησιμοποιούν καθημερινά συσκευές όπως οι Ηλεκτρονικοί Υπολογιστές και τα Κινητά Τηλέφωνα στην προσπάθεια τους είτε να διεκπεραιώσουν προσωπικές τους υποχρεώσεις είτε να ψυχαγωγηθούν μέσω της πληθώρας των παρεχομένων υπηρεσιών. Φυσικά, όλα αυτά είναι αποτέλεσμα των εξελίξεων στον τομέα της επιστήμης των υπολογιστών.

Όμως, οι εξελίξεις αυτές έχουν επιφέρει πρόσθετες αλλαγές στην καθημερινότητα των ανθρώπων καθώς τα τελευταία χρόνια παρατηρούμε ένα ιδιαίτερο ενδιαφέρον φαινόμενο κατά το οποίο ο προγραμματισμός αποτελεί ένα απαραίτητο εργαλείο για όλους τους κλάδους της ανθρώπινης δραστηριότητας. Μάλιστα, είναι χαρακτηριστικό, πως ακόμη και στα παραδοσιακά επαγγέλματα, είναι επιθυμητές πλέον οι δεξιότητες προγραμματισμού για την πραγματοποίηση καθημερινών εργασιών. Θα μπορούσαμε λοιπόν να πούμε ότι απαιτείται σχεδόν όλοι μας να γίνουμε ως ένα βαθμό προγραμματιστές.

Το γεγονός όμως αυτό αποτελεί προβληματική εξέλιξη καθώς για να γίνει κάποιος προγραμματιστής συχνά απαιτείται εξειδικευμένη γνώση καθώς και πολύς κόπος και χρόνος. Για την επίλυση του προβλήματος αυτού θα μπορούσαμε να βελτιστοποιήσουμε τα εργαλεία συγγραφής κώδικα. Πιο συγκεκριμένα, κρίνεται απαραίτητη η δημιουργία εξυπνότερων εργαλείων ανάπτυξης κώδικα τα οποία όχι μόνο να προσφέρουν ένα φιλικό περιβάλλον εργασίας και απλές χρήσιμες λειτουργίες (όπως γίνεται μέχρι σήμερα) αλλά και να μπορούν να "κατανοήσουν" τον κώδικα και να προτείνουν διορθώσεις και βελτιστοποιήσεις. Για παράδειγμα, είναι επιθυμητό οι σύγχρονοι $IDEs$ να μπορούν να εντοπίζουν μόνοι τους τόσο απλά όσο και σύνθετα προβλήματα (bugs) στον κώδικα (είτε αυτά είναι συντακτικά είτε νοηματικά) και γιατί όχι να έχουν την δυνατότητα να τα διορθώνουν. Επιπρόσθετα, είναι επιθυμητό οι σύγχρονοι $IDEs$ να έχουν την δυνατότητα να συμπληρώνουν αυτόματα κομμάτια κώδικα (code auto-completion), τόσο με συντακτικά όσο και νοηματικά κριτήρια, ή και να γράφουν κώδικα ο οποίος να επιλύει ένα πρόβλημα δεδομένης της περιγραφής του προβλήματος (code synthesis). Τέλος, κρίνεται επίσης σημαντικό οι σύγχρονοι

$IDEs$ να μπορούν να κατανοούν την λειτουργικότητα του κάθε κώδικα και είτε να προτείνουν είτε να πραγματοποιούν βελτιστοποιήσεις, οι οποίες θα μπορούσαν να σχετίζονται με την μείωση της πολυπλοκότητας της μεθόδου ή την παραλληλοποίηση αυτής έτσι ώστε να μειωθεί σημαντικά ο χρόνος εκτέλεσης.

Επιπρόσθετα, στα πλαίσια της επίλυσης του προβλήματος κρίνεται απαραίτητη η βελτιστοποίηση της διαδικασίας εκπαίδευσης νέων προγραμματιστών. Για τον σκοπό αυτό, η δημιουργία συστημάτων που κατανοούν την λειτουργικότητα του κάθε κώδικα κρίνεται απαραίτητη έτσι ώστε να συμβάλει στην απλοποίηση και την αυτοματοποίηση της διαδικασίας εκμάθησης συγγραφής κώδικα. Πιο συγκεκριμένα, τέτοια συστήματα θα μπορούσαν να χρησιμοποιηθούν στα συστήματα αυτόματης αξιολόγησης κώδικα που διαθέτουν τα σύγχρονα μαθήματα πληροφορικής έτσι ώστε να μην αξιολογούν μόνο την αποδοτικότητα της λύσης του εκπαιδευόμενου αλλά και το επίπεδο των γνώσεων του και την ποιότητα της υλοποίησης του. Μάλιστα, ένα τέτοιο συστήματα θα μπορούσε κατανοώντας το επίπεδο του χρήστη, με βάση την υλοποίηση του, να προτείνει αυτόματα είτε εξειδικευμένο διδακτικό υλικό είτε εξειδικευμένα προβλήματα προς λύση. Παράλληλα, συστήματα αυτόματης ανίχνευσης προβλημάτων στον κώδικα , όπως αυτά που αναφέραμε παραπάνω, θα μπορούσαν να φανούν ιδιαίτερα χρήσιμα και στον τομέα της εκπαίδευσης δίνοντας στους εκπαιδευόμενους προτάσεις για την αποφυγή των συχνότερων λαθών τους κτλ.

Τέλος, η ανάπτυξη έξυπνων συστημάτων κατανόησης κώδικα κρίνεται απαραίτητη και για άλλες πτυχές της επιστήμης των υπολογιστών. Για παράδειγμα, τέτοια συστήματα θα μπορούσαν να χρησιμοποιηθούν από χρονοδρομολογητές οι οποίοι πλέον θα μπορούν να αποφασίσουν τους υπολογιστικούς πόρους που θα αναθέτουν σε κάθε διεργασία όχι μόνο με χρήση ευριστικών μεθόδων αλλά και με χρήση τέτοιων συστημάτων τα οποία θα επιτρέπουν στους χρονοδρομολογητές να γνωρίζουν την υπολογιστική πολυπλοκότητα της κάθε διεργασίας.

Συνοψίζοντας, θεωρούμε ότι η ανάπτυξη έξυπνων συστημάτων που κατανοούν κώδικα κρίνεται απαραίτητη για την βελτίωση τόσο της εργασίας των προγραμματιστών και επιστημόνων όσο και της διαδικασίας εκπαίδευσης νέων επιστημόνων και προγραμματιστών. Επιπρόσθετα, τέτοια συστήματα θα μπορούσαν να βοηθήσουν και άλλους τομείς της επιστήμης των υπολογιστών όπως είναι οι χρονοδρομολογητές που χρησιμοποιούνται συχνά σε διάφορους servers.

1.2 Αντικείμενο Διπλωματικής

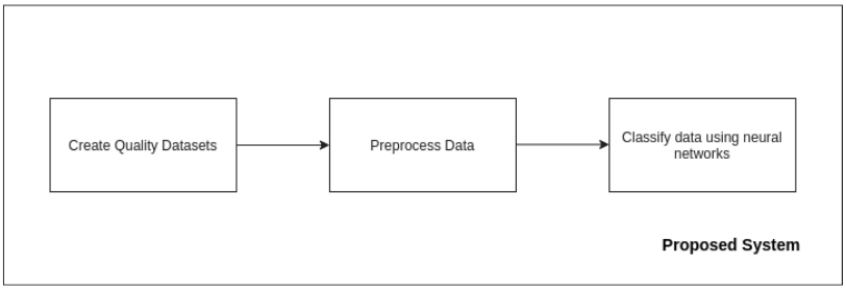
1.2.1 Το πρόβλημα

Όπως σχολιάσαμε και παραπάνω, στις μέρες μας, κρίνεται επιτακτική η ανάγκη ανάπτυξης έξυπνων συστημάτων τα οποία μπορούν να καταλάβουν τον κώδικα και τις λειτουργίες που επιτελεί. Για τον σκοπό αυτό, στην παρούσα διπλωματική εργασία θα μελετήσουμε το πρόβλημα της ταξινόμησης πηγαίου κώδικα σε αλγορίθμους. Πιο συγκεκριμένα, δεδομένου ενός κώδικα γραμμένου σε μία γνωστή γλώσσα προγραμματισμού, προσπαθούμε να ταξινομήσουμε τον κώδικα αυτό στην αλγοριθμική κλάση στην οποία ανήκει. Η ταξινόμηση αυτή γίνεται με βασικό γνώμονα την λειτουργία ή τις λειτουργίες που επιτελεί ο εκάστοτε κώδικας. Για την καλύτερη κατανόηση του προβλήματος, θα παραθέσουμε ένα παράδειγμα.

Σχήμα 1.1: Παράδειγμα ορισμού προβλήματος

1.2.2 Συνεισφορά

Στην παρούσα διπλωματική εργασία αποφασίσαμε να προσεγγίσουμε την λύση του παραπάνω προβλήματος με την χρήση Νευρωνικών Δικτύων, γεγονός που συνάδει και με τις προτάσεις της σύγχρονης βιβλιογραφίας. Αρχικά, δημιουργήσαμε δύο νέα σύνολα δεδομένων (Datasets) τα οποία πληρούν αυστηρά πρωτόκολλα ποιότητας. Παράλληλα, στα πλαίσια της δημιουργίας των συνόλων αυτών προτείναμε μία νέα προσέγγιση για την δημιουργία συνόλων δεδομένων με γνωστή κλάση σε μεγάλη κλίμακα. Επιπρόσθετα, στα πλαίσια της εργασίας αυτής μελετήσαμε διάφορες τεχνικές προ-επεξεργασίας καθώς και μοντέλα μηχανικής μάθησης. Μάλιστα, αξίζει να σημειώσουμε ότι τα πειραματικά αποτελέσματα της εργασίας αυτής κρίνονται ως ιδιαίτερα ενθαρρυντικά καθώς οι τεχνικές και τα μοντέλα που προτείναμε οδήγησαν σε αποτελεσματικές λύσεις του προβλήματος. Τέλος, μέσα από την εργασία αυτή αναδείξαμε πιθανούς περιορισμούς των υπαρχουσών μεθόδων και συνόλων δεδομένων της σύγχρονης βιβλιογραφίας. Στην συνέχεια, παραθέτουμε ένα σχήμα που συνοψίζει την δομή του συστήματος που θα μελετήσουμε στην παρούσα διπλωματική εργασία.



Σχήμα 1.2: Προτεινόμενο σύστημα

1.3 Επισκόπηση της σύγχρονης βιβλιογραφίας

Στην ενότητα αυτή θα παραθέσουμε μία επισκόπηση της σύγχρονης βιβλιογραφίας :

1. Στο [43] αναπτύσσεται ένα $Language\ Model$ με $stacked\ LSTMs$ για την δημιουργία συνθετικού κώδικα σε γλώσσα $OpenCL$. Για την δημιουργία του συνόλου δεδομένων κατέβασαν αρχεία από το Github. Στην συνέχεια, με χρήση μεταγλωττιστών πραγματοποίησαν την προ-επεξεργασία των αρχείων. Κατά την προ-επεξεργασία αυτή, επέβαλαν έναν ενιαίο τρόπο γραφής σε όλους τους κώδικες και κανονικοποίησαν τα ονόματα μεταβλητών και συναρτήσεων με έναν σχετικά "απλό" τρόπο (διαφορετικά ονόματα για μεταβλητές και διαφορετικά για συναρτήσεις). Με τα δεδομένα αυτά εκπαίδευσαν τα $stacked\ LSTMs$, τα οποία και χρησιμοποίησαν για την δημιουργία dummy κωδίκων και τον εντοπισμό πιθανών προβλημάτων σε μεταγλωττιστές. Είναι χαρακτηριστικό, πως όχι μόνο εντόπισαν πολλά bugs αλλά και τύπους bugs τα οποία οι προηγούμενες μέθοδοι αδυνατούσαν να εντοπίσουν.

2. Στο [34] αναπτύσσονται $CNNs$ για την επίλυση του προβλήματος της ταξινόμησης πηγαίου κώδικα σε αλγορίθμους. Πιο συγκεκριμένα, χρησιμοποιούν το OJ System dataset[61], το οποίο περιλαμβάνει 52.000 προγράμματα και 104 κλάσεις και προέρχεται από μία διαδικτυακή πλατφόρμα υποβολής ασκήσεων προγραμματισμού. Στην συνέχεια, για την προ-επεξεργασία των δεδομένων αγνοούν τα ονόματα μεταβλητών και συναρτήσεων κρατώντας μόνο τις δεσμευμένες λέξεις της γλώσσας. Τις λέξεις αυτές, αφού τις ομαδοποιούν κατάλληλα τις αναπαριστούν σαν one-hot vectors. Μετά την προ-επεξεργασία αυτή εκπαιδεύουν $CNNs$ τα οποία μάλιστα πετυχαίνουν 93.7% στην μετρική f1.

3. Στο [51] προτείνεται μία νέα αρχιτεκτονική για την αναπαράσταση κώδικα η οποία βασίζεται στην χρήση $ASTs$ και το όνομα αυτής είναι, $ASTnn$. Στην προσέγγιση αυτή γίνεται εμπλουτισμός των ακμών του AST, σε μια προσπάθεια αύξησης της αποτελεσματικότητας. Μία από τις μεθόδους αξιολόγησης του μοντέλου θα είναι και το πρόβλημα της ταξινόμησης πηγαίου κώδικα σε αλγόριθμους όπου και πάλι θα χρησιμοποιηθεί το OJ System dataset[61]. Μάλιστα κατά την αξιολόγηση το μοντέλο πέτυχε, 98.2% $accuracy$, το οποίο θεωρείται πολύ υψηλή τιμή, αν και προβληματίζει η χρήση της μετρικής $accuracy$ για μη ισορροπημένα σύνολα δεδομένων.

4. Στο [36] μελετάται το φαινόμενο των αρχείων αντιγράφων και σχεδόν αντιγράφων στα σύνολα δεδομένων που χρησιμοποιούνται στην βιβλιογραφία. Για την εύρεση αρχείων αντιγράφων αναπτύχθηκε μία ευριστική μέθοδος που χρησιμοποιεί την εξίσωση της ομοιότητας συνημιτόνου. Είναι χαρακτηριστικό πως η μέθοδος αυτή εντόπισε σε όλα τα σύνολα δεδομένων αρχεία αντίγραφα και πολλές φορές μάλιστα τα ποσοστά αυτών ξεπερνούσαν το 20%. Επιπρόσθετα, παρουσιάζονται εκτενώς και οι επιπτώσεις της ύπαρξης αρχείων αντιγράφων στην επίδοση των μοντέλων.

Κεφάλαιο 2

Απαραίτητο θεωρητικό υπόβαθρο

2.1 Μεταγλωττιστής (Compiler)

Μεταγλωττιστής ή μεταφραστής (compiler)[3] ονομάζεται ένα πρόγραμμα υπολογιστή που διαβάζει κώδικα γραμμένο σε μια γλώσσα προγραμματισμού (την πηγαία γλώσσα) και τον μεταφράζει σε ισοδύναμο κώδικα σε μια άλλη γλώσσα προγραμματισμού (τη γλώσσα στόχο). Το κείμενο της εισόδου ονομάζεται πηγαίος κώδικας (source code), ενώ η έξοδος του προγράμματος, η οποία συχνά έχει δυαδική μορφή, αντικειμενικός κώδικας (object code).

Ο όρος «μεταγλωττιστής» χρησιμοποιείται κυρίως για προγράμματα που μεταφράζουν μια γλώσσα προγραμματισμού υψηλού επιπέδου σε μια γλώσσα χαμηλότερου επιπέδου (όπως η συμβολική γλώσσα ή η γλώσσα μηχανής). Αν το μεταγλωττισμένο πρόγραμμα πρόκειται να εκτελεστεί σε έναν υπολογιστή που έχει διαφορετικό επεξεργαστή ή λειτουργικό σύστημα σε σχέση με την πλατφόρμα που εκτελείται ο μεταγλωττιστής, ο τελευταίος τότε ονομάζεται cross-compiler. Ένα πρόγραμμα που μεταφράζει από μια γλώσσα χαμηλού επιπέδου σε μια υψηλότερου επιπέδου ονομάζεται decompiler. Ένα πρόγραμμα που μεταφράζει από μια γλώσσα υψηλού επιπέδου σε μια άλλη, επίσης υψηλού επιπέδου, ονομάζεται συνήθως γλωσσικός μεταφραστής, μεταφραστής από πηγαίο κώδικα σε πηγαίο κώδικα (source to source translator) ή μετατροπέας γλωσσών. Ένα πρόγραμμα που μεταφράζει τη μορφή εκφράσεων σε άλλη μορφή, διατηρώντας την ίδια γλώσσα, ονομάζεται language rewriter.

Ένας μεταγλωττιστής μπορεί να περιλαμβάνει οποιαδήποτε από τις εξής λειτουργίες: λεκτική ανάλυση, προ-επεξεργασία, συντακτική ανάλυση, σημασιολογική ανάλυση (μετάφραση καθοδηγούμενη από τη σύνταξη), παραγωγή κώδικα και βελτιστοποίηση κώδικα.

Τα σφάλματα προγραμμάτων που προκύπτουν από λανθασμένη μεταγλώττιση είναι πολύ δύσκολο να εντοπιστούν και να αντιμετωπιστούν. Για αυτόν τον λόγο οι κατασκευαστές μεταγλωττιστών κάνουν σημαντικές προσπάθειες για να βεβαιώσουν την ορθότητα λειτουργίας του λογισμικού τους.

2.1.1 Η δομή ενός μεταγλωττιστή

Ένας μεταγλωττιστής αποτελεί τη γέφυρα μεταξύ προγραμμάτων πηγαίου κώδικα σε κάποια γλώσσα υψηλού επιπέδου και του υλικού. Μάλιστα, αξίζει να σημειώσουμε ότι ένας μεταγλωττιστής πρέπει :

- να επαληθεύσει ότι τα προγράμματα έχουν σωστή σύνταξη

- να παράγει σωστό και γρήγορο αντικειμενικό κώδικα
- να οργανώσει το πώς εκτελείται το πρόγραμμα
- να δώσει μορφή στην έξοδο που να είναι κατάλληλη για τον συμβολομεταφραστή ή τον συνδέτη

Τέλος, αξίζει να σημειώσουμε ότι ένας μεταγλωττιστής αποτελείται από τρία κύρια μέρη: το εμπρόσθιο, το ενδιάμεσο και το οπίσθιο τμήμα.

Το εμπρόσθιο τμήμα (front end) ελέγχει αν το πρόγραμμα είναι σωστά γραμμένο με βάση τη σύνταξη και τη σημασιολογία της γλώσσας προγραμματισμού. Σε αυτό το σημείο φαίνεται ποιο πρόγραμμα είναι έγκυρο και ποιο όχι, και εμφανίζονται μηνύματα σφάλματος. Εδώ επίσης ελέγχονται οι τύποι συλλέγοντας πληροφορία τύπων. Το εμπρόσθιο τμήμα παράγει επίσης μια ενδιάμεση αναπαράσταση (intermediate representation ή IR) του πηγαίου κώδικα, η οποία πρόκειται να δοθεί στο ενδιάμεσο τμήμα για επεξεργασία. Στο εμπρόσθιο τμήμα μπορούν να ανήκουν μεταξύ άλλων και τα εξής στάδια:

ένα προ-επεξεργαστή που αναλαμβάνει να επεξεργαστεί κάποιες ειδικές εντολές ή άλλα χαρακτηριστικά του πηγαίου κώδικα, ώστε να είναι σε κατάλληλη μορφή για τη μεταγλώττιση, ένα λεκτικό αναλυτή (lexical analyzer ή lexer) που τεμαχίζει τον πηγαίο κώδικα σε λεκτικές μονάδες (tokens), ξεχωρίζοντας για παράδειγμα τις λέξεις-κλειδιά, τις εντολές της γλώσσας και τις τιμές του προγράμματος, ένα συντακτικό αναλυτή (parser) που συνθέτει τις λεκτικές μονάδες με βάση τη σύνταξη της γλώσσας, ώστε να προκύψει μια αφηρημένη μορφή του προγράμματος (συντακτικό δέντρο), κατάλληλη για περαιτέρω επεξεργασία. Τα τμήματα του λεκτικού αναλυτή και του συντακτικού αναλυτή είναι καθιερωμένο να υλοποιούνται με ειδικά εργαλεία για αυτό το σκοπό, τις γεννήτριες λεκτικών και συντακτικών αναλυτών. Στις γεννήτριες της πρώτης κατηγορίας, όπως το Lex, ο προγραμματιστής του μεταγλωττιστή ορίζει τις λεκτικές μονάδες που μπορεί να συναντηθούν στον πηγαίο κώδικα (όπως οι δεσμευμένες λέξεις και τα αλφαριθμητικά) και η γεννήτρια αναλαμβάνει να παράγει το αντίστοιχο τμήμα του μεταγλωττιστή. Αντίστοιχα, στις γεννήτριες της δεύτερης κατηγορίας, όπως το Yacc, ο προγραμματιστής ορίζει τη γραμματική της πηγαίας γλώσσας σε μια κατάλληλη μορφή (όπως η μορφή Backus–Naur (BNF)) και στη συνέχεια παράγεται ο συντακτικός αναλυτής που διαβάζει αυτή τη γραμματική.

Στο ενδιάμεσο τμήμα (middle end) γίνονται οι βελτιστοποιήσεις. Συνηθισμένοι μετασχηματισμοί βελτιστοποίησης είναι η αφαίρεση άχρηστου ή απρόσιτου κώδικα, ο εντοπισμός και η διάδοση των σταθερών (constant propagation), η μεταφορά υπολογισμών εκτός συχνά χρησιμοποιούμενων τμημάτων (για παράδειγμα, μετακίνηση έξω από μια δομή επανάληψης), ή η εξειδίκευση ενός υπολογισμού ανάλογα με τον κώδικα που τον περιβάλλει. Το ενδιάμεσο τμήμα παράγει στη συνέχεια μια άλλη ενδιάμεση αναπαράσταση, για το οπίσθιο τμήμα. Οι περισσότερες βελτιστοποιήσεις έχουν ήδη γίνει στο ενδιάμεσο τμήμα.

Το οπίσθιο τμήμα (back end) είναι υπεύθυνο για τη μετάφραση της ενδιάμεσης αναπαράστασης του ενδιάμεσου τμήματος σε γλώσσα μηχανής, συμβολική γλώσσα, γλώσσα προγραμματισμού (όπως η C) ή κώδικα για κάποια αφηρημένη μηχανή (abstract machine) όπως ο κώδικας byte (bytecode). Κάθε εντολή της ενδιάμεσης αναπαράστασης αντιστοιχεί σε κάποιες συμβολικές εντολές. Η κατανομή καταχωρητών αντιστοιχεί καταχωρητές στις μεταβλητές του προγράμματος. Το οπίσθιο τμήμα χρησιμοποιεί το υλικό με τέτοιο τρόπο ώστε να χρησιμοποιούνται όλες οι λειτουργικές μονάδες του υλικού με αποδοτικό τρόπο. Αν και οι περισσότεροι αλγόριθμοι βελτιστοποίησης για αυτά

τα προβλήματα είναι πολυπλοκότητας NP, έχουν αναπτυχθεί και αρκετά προχωρημένες ευριστικές τεχνικές.

2.1.2 Έξοδος του μεταγλωττιστή

Οι μεταγλωττιστές μπορούν να χωριστούν σε κατηγορίες ανάλογα με την πλατφόρμα στην οποία πρόκειται να εκτελεστεί ο παραγόμενος κώδικας (target platform).

Ένας μεταγλωττιστής ονομάζεται native ή hosted αν παράγει κώδικα που πρόκειται να εκτελεστεί στον ίδιο τύπο υπολογιστή και λειτουργικού συστήματος με αυτά στα οποία εκτελείται ο ίδιος ο μεταγλωττιστής. Αν ο παραγόμενος κώδικας πρόκειται να εκτελεστεί σε διαφορετική πλατφόρμα, τότε αναφερόμαστε σε cross compiler. Οι cross-compilers χρησιμοποιούνται συχνά στην ανάπτυξη λογισμικού για ενσωματωμένα συστήματα, τα οποία δεν προτίθεντε τα ίδια να υποστηρίξουν κάποιο περιβάλλον ανάπτυξης λογισμικού.

Η έξοδος ενός μεταγλωττιστή που παράγει κώδικα για μια εικονική μηχανή (virtual machine - VM) μπορεί να εκτελεστεί ή να μην εκτελεστεί στην ίδια πλατφόρμα με το μεταγλωττιστή που την παρήγαγε. Για αυτόν τον λόγο, οι μεταγλωττιστές αυτού του τύπου δεν θεωρείται ότι ανήκουν σε μια από τις προηγούμενες κατηγορίες.

Η γλώσσα χαμηλού επιπέδου στην οποία παράγει κώδικα ο μεταγλωττιστής μπορεί η ίδια να είναι μια γλώσσα υψηλού επιπέδου. Η C, η οποία συχνά θεωρείται ένα είδος φορητής συμβολικής γλώσσας, μπορεί επίσης να είναι η γλώσσα στόχος του μεταγλωττιστή. Για παράδειγμα, το Cfront, ο πρωτότυπος μεταγλωττιστής της C++, χρησιμοποιούσε τη C σαν γλώσσα παραγόμενου κώδικα. Ο κώδικας C που παράγεται από μεταγλωττιστές αυτού του τύπου συνήθως δεν προορίζεται για ανάγνωση και συντήρηση από ανθρώπους. Αυτό σημαίνει ότι μπορεί να μην ακολουθεί κανόνες στοίχισης ή άλλου τύπου για ευανάγνωστο κώδικα. Κάποια χαρακτηριστικά της C την κάνουν καλή γλώσσα στόχο, για παράδειγμα μπορεί να παραχθεί κώδικας C με οδηγίες $\#line$ ώστε να βοηθήσει στην αποσφαλμάτωση του αρχικού πηγαίου κώδικα.

2.1.3 Αφηρημένο συντακτικό δέντρο (AST)

Στην επιστήμη των υπολογιστών, ένα αφηρημένο συντακτικό δέντρο (AST)[2], ή απλά ένα δέντρο σύνταξης, είναι μια δεντρική αναπαράσταση της αφηρημένης συντακτικής δομής του πηγαίου κώδικα που είναι γραμμένη σε μια γλώσσα προγραμματισμού. Κάθε κόμβος του δέντρου δηλώνει μια κατασκευή που εμφανίζεται στον πηγαίο κώδικα. Η σύνταξη είναι "αφηρημένη" με την έννοια ότι δεν αντιπροσωπεύει κάθε λεπτομέρεια που εμφανίζεται στην πραγματική σύνταξη, αλλά μόνο τις δομικές ή τις σχετικές με το περιεχόμενο λεπτομέρειες. Για παράδειγμα, οι παρενθέσεις ομαδοποίησης (δηλαδή οι χαρακτήρες "()") είναι έμμεσες στη δομή του δέντρου, επομένως αυτές δεν χρειάζεται να εκπροσωπούνται ως ξεχωριστοί κόμβοι. Παρομοίως, μια συντακτική κατασκευή όπως μια έκφραση if-condition-then μπορεί να δηλωθεί μέσω ενός μόνο κόμβου με τρεις κλάδους. Το γεγονός αυτό, διακρίνει τα αφηρημένα συντακτικά δέντρα από τις υπόλοιπες μορφές συντακτικών δέντρων, όπως είναι τα concrete syntax trees και τά designated parse trees. Τα δέντρα τύπου parse δημιουργούνται συνήθως από έναν αναλυτή (parser) κατά τη διάρκεια της διαδικασίας μετάφρασης και σύνταξης του πηγαίου κώδικα. Μόλις δημιουργηθεί, πρόσθετες πληροφορίες προστίθενται στο AST μέσω μεταγενέστερης επεξεργασίας, π.χ., ανάλυση με βάση τα συμφραζόμενα. Τέλος, αξίζει να σημειώσουμε ότι τα αφηρημένα συντακτικά δέντρα χρησιμοποιούνται επίσης στην ανάλυση

προγραμμάτων (program analysis) και στα συστήματα μετασχηματισμού προγραμμάτων (program transformation systems).

2.1.3.1 Η χρήση του AST στους μεταγλωττιστές

Τα αφηρημένα συντακτικά δέντρα είναι δομές δεδομένων που χρησιμοποιούνται ευρέως σε μεταγλωττιστές για να αντιπροσωπεύουν τη δομή του πηγαίου κώδικα. Ένα AST είναι συνήθως το αποτέλεσμα της φάσης ανάλυσης σύνταξης ενός μεταγλωττιστή. Συχνά χρησιμεύει ως ενδιάμεση αναπαράσταση του προγράμματος σε διάφορα στάδια που απαιτεί ο μεταγλωττιστής και έχει ισχυρό αντίκτυπο στην τελική έξοδο του μεταγλωττιστή. Στην συνέχεια, θα παραθέσουμε μερικές από τις ιδιότητες που έχουν τα AST και βοηθούν την διαδικασία μεταγλώττισης :

- Ένα AST μπορεί να επεξεργαστεί και να βελτιωθεί με πληροφορίες όπως ιδιότητες και σχολιασμούς για κάθε στοιχείο που περιέχει. Μάλιστα, τέτοια επεξεργασία και σχολιασμός είναι αδύνατο να επιτευχθεί με τον πηγαίο κώδικα ενός προγράμματος, δεδομένου ότι συνεπάγεται αλλαγές στον κώδικα αυτό.
- Σε σύγκριση με τον πηγαίο κώδικα, ένα AST δεν περιλαμβάνει τα μη απαραίτητα σημεία στίξης κτλ (για παράδειγμα οι χαρακτήρες “(),[]”).
- Ένα AST περιέχει συνήθως επιπλέον πληροφορίες σχετικά με το πρόγραμμα, χάρη στα διαδοχικά στάδια ανάλυσης από τον μεταγλωττιστή. Για παράδειγμα, μπορεί να αποθηκεύσει τη θέση κάθε στοιχείου στον πηγαίο κώδικα, επιτρέποντας στον μεταγλωττιστή να εμφανίζει χρήσιμα μηνύματα σφάλματος.

Τα AST είναι απαραίτητα λόγω τόσο της εγγενούς φύσης των γλωσσών προγραμματισμού όσο και της τεκμηρίωσής τους. Βέβαια, είναι γνωστό πως οι γλώσσες προγραμματισμού είναι συχνά διφορούμενες από τη φύση τους. Προκειμένου να αποφευχθεί αυτή η ασάφεια, οι γλώσσες προγραμματισμού συχνά ορίζονται ως γραμματικές χωρίς συμφραζόμενα (CFG). Ωστόσο, υπάρχουν συχνά πτυχές των γλωσσών προγραμματισμού που μία CFG δεν μπορεί να εκφράσει, αλλά αποτελούν μέρος της γλώσσας και τεκμηριώνονται στις προδιαγραφές της. Αυτές είναι λεπτομέρειες που απαιτούν ένα πλαίσιο για τον προσδιορισμό της εγκυρότητας και της συμπεριφοράς τους. Για παράδειγμα, εάν μια γλώσσα επιτρέπει τη δήλωση νέων τύπων, μία CFG δεν μπορεί να προβλέψει ούτε τα ονόματα αυτών των τύπων ούτε τον τρόπο με τον οποίο θα πρέπει να χρησιμοποιούνται. Ακόμα κι αν μια γλώσσα έχει ένα προκαθορισμένο σύνολο τύπων, η επιβολή της σωστής χρήσης αυτών συνήθως απαιτεί κάποιο ειδικό πλαίσιο.

Αν και υπάρχουν άλλες δομές δεδομένων που εμπλέκονται στην εσωτερική λειτουργία ενός μεταγλωττιστή, το AST εκτελεί μια μοναδική λειτουργία. Κατά το πρώτο στάδιο, το στάδιο της συντακτικής ανάλυσης, ένας μεταγλωττιστής παράγει ένα δέντρο ανάλυσης (parse tree). Το δέντρο αυτό μπορεί να χρησιμοποιηθεί για την εκτέλεση σχεδόν όλων των λειτουργιών ενός μεταγλωττιστή μέσω της συντακτικά κατευθυνόμενης μετάφρασης. Αν και η μέθοδος αυτή μπορεί να οδηγήσει σε έναν πιο αποτελεσματικό μεταγλωττιστή, αντιτίθεται στις αρχές της τεχνολογίας λογισμικού για τη σύνταξη και τη συντήρηση προγραμμάτων. Ένα άλλο πλεονέκτημα που έχουν τα AST έναντι των δέντρων ανάλυσης είναι το μέγεθος. Πιο συγκεκριμένα, είναι γνωστό ότι τα AST έχουν τόσο μικρότερο βάθος όσο και αριθμό κόμβων από τα αντίστοιχα parse trees.

2.1.3.2 Η σχεδίαση ενός AST

Ο σχεδιασμός ενός AST συχνά συνδέεται στενά με το σχεδιασμό ενός μεταγλωττιστή και τα αναμενόμενα χαρακτηριστικά του.

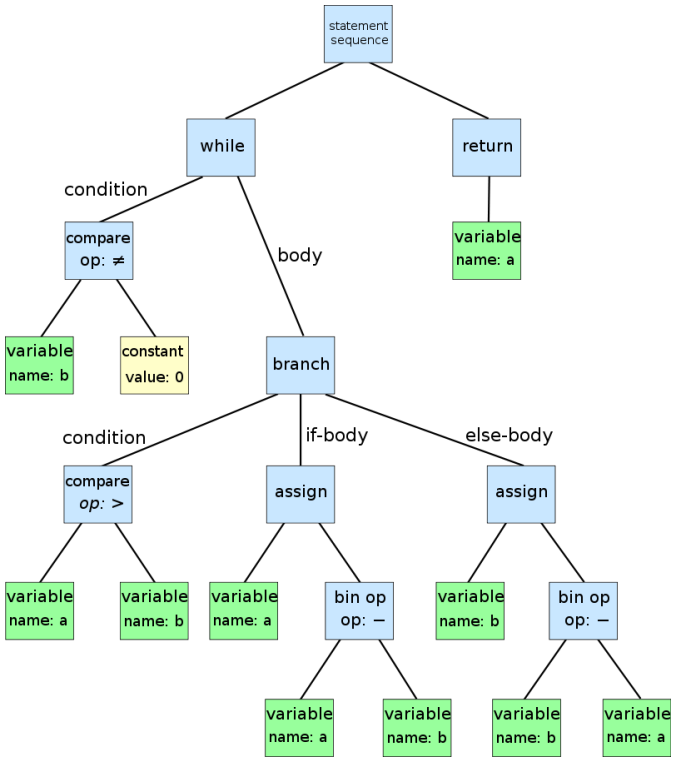
Οι βασικές απαιτήσεις περιλαμβάνουν τα ακόλουθα:

- Πρέπει να διατηρηθούν οι τύποι των μεταβλητών, καθώς και η θέση κάθε δήλωσης στον πηγαίο κώδικα.
- Η σειρά των εκτελέσιμων δηλώσεων πρέπει να αντιπροσωπεύεται ρητά και να ορίζεται καλά.
- Τα στοιχεία που βρίσκονται αριστερά και δεξιά σε δυαδικές πράξεις ή λειτουργίες πρέπει να αποθηκεύονται και να αναγνωρίζονται σωστά.
- Τα αναγνωριστικά και οι καταχωρημένες τιμές τους πρέπει να αποθηκεύονται για δηλώσεις ανάθεσης.

Στην συνέχεια, για την καλύτερη κατανόηση του των αφηρημένων συντακτικών δέτρων (AST) θα παραθέσουμε ένα παράδειγμα υλοποίησης του αλγόριθμου του Ευκλείδη (gcd).

```
while b ≠ 0
  if a > b
    a := a − b
  else
    b := b − a
return a
```

Σχήμα 2.1: Αλγόριθμος GCD [4]



Σχήμα 2.2: Παράδειγμα AST για τον αλγόριθμο GCD [4]

2.1.4 Ο μεταγλωττιστής Clang

Το Clang[4] είναι μια διεπαφή μεταγλωττιστή για τις γλώσσες προγραμματισμού C, C++, Objective-C και Objective-C++, καθώς και τα framework OpenMP, OpenCL, RenderScript, CUDA και HIP. Χρησιμοποιεί την υποδομή μεταγλωττιστή LLVM ως back end και αποτελεί μέρος του κύκλου κυκλοφορίας LLVM από την έκδοση LLVM 2.6. Έχει σχεδιαστεί για να λειτουργεί ως υποκατάστατο της συλλογής GNU Compiler Collection (GCC), υποστηρίζοντας τις περισσότερες από τις σημαίες μεταγλώττισης (compilation flags) και τις ανεπίσημες επεκτάσεις γλώσσας. Οι συνεργάτες του περιλαμβάνουν τις εταιρίες Apple, Microsoft, Google, ARM, Sony, Intel και AMD. Πρόκειται για λογισμικό ανοιχτού κώδικα, με τον πηγαίο κώδικα του να κυκλοφορεί υπό την άδεια του Πανεπιστημίου του Ιλλινόις (NCSA), μια ανεκτική άδεια ελεύθερου λογισμικού. Από την έκδοση 9.0.0 και μετά, παραχωρήθηκε στο Apache License 2.0 με εξαιρέσεις LLVM. Στο σημείο αυτό, αξίζει να σημειώσουμε ότι από προεπιλογή, το Clang μεταγλωττίζει την γλώσσα C++ χρησιμοποιώντας το πρότυπο C++98 (με πολλές επεκτάσεις C++11 και GNU GCC, ενώ η προεπιλογή του GCC αφορά μεταγενέστερο πρότυπο της C++), αλλά ανάλογα με την έκδοση του μεταγλωττιστή υποστηρίζει και νεότερά πρότυπα της γλώσσας C++. Τέλος, το πακέτο του Clang περιλαμβάνει τόσο το Clang front end (στατικός αναλυτής) όσο και διάφορα εργαλεία ανάλυση κώδικα (για παράδειγμα ένα τέτοιο εργαλείο, που μάλιστα θα χρησιμοποιηθεί στην παρούσα εργασία είναι το Clang tidy).

Όπως αναφέραμε και παραπάνω, το Clang προορίζεται να δουλέψει πάνω από το LLVM. Ο συνδυασμός Clang και LLVM παρέχει την πλειοψηφία των εργαλείων, που επιτρέπει την αντικατάσταση του stack του GCC. Μάλιστα, δεδομένου ότι είναι κατασκευασμένο με ένα library-based σχεδιασμό, όπως το υπόλοιπο LLVM, το Clang είναι εύκολο να ενσωματωθεί σε άλλες εφαρμογές. Αυτός είναι ένας από τους λόγος για τον οποίο οι περισσότερες εφαρμογές OpenCL έχουν δημιουργηθεί με Clang και LLVM.

Ένας από τους κύριους στόχους του Clang είναι να παρέχει μια library-based αρχιτεκτονική, για να επιτρέπει στον μεταγλωττιστή να συνδέεται εύκολα με εργαλεία που αλληλεπιδρούν με τον πηγαίο κώδικα, όπως ένα γραφικό περιβάλλον ανάπτυξης κώδικα (IDE). Αντίθετα, το GCC έχει σχεδιαστεί για να λειτουργεί σε μια ροή εργασιών compile-link-debug και η ενσωμάτωσή του σε εξωτερικά εργαλεία δεν είναι πάντα εύκολη. Για παράδειγμα, το GCC χρησιμοποιεί ένα βήμα που ονομάζεται fold που είναι το κλειδί για τη συνολική διαδικασία μεταγλώττισης, έχει όμως και την παρενέργεια της μετάφρασης του δέντρου κώδικα σε μια μορφή που δεν μοιάζει ιδιαίτερα με τον αρχικό πηγαίο κώδικα. Επιπρόσθετα, το Clang έχει σχεδιαστεί για να διατηρεί περισσότερες πληροφορίες κατά τη διαδικασία μεταγλώττισης, σε σύγκριση με το GCC, και να διατηρεί τη συνολική μορφή του αρχικού κώδικα. Ο στόχος αυτού είναι να διευκολύνει την αντιστοίχιση σφαλμάτων στον αρχικό πηγαίο κώδικα. Παράλληλα, οι αναφορές σφαλμάτων που προσφέρονται από το Clang στοχεύουν στο να είναι πιο λεπτομερείς και συγκεκριμένες, καθώς και αναγνώσιμες από μηχανή, έτσι ώστε οι IDE να μπορούν να αναζητήσουν την έξοδο του μεταγλωττιστή κατά τη μεταγλώττιση. Τέλος, το δέντρο ανάλυσης (parse tree) είναι επίσης καταλληλότερο για υποστήριξη αυτόματης αναδιαμόρφωσης κώδικα, καθώς αντιπροσωπεύει άμεσα τον αρχικό πηγαίο κώδικα.

2.2 Αλγόριθμοι μηχανικής μάθησης

Στην ενότητα αυτή θα ασχοληθούμε με την περιγραφή αλγορίθμων μηχανικής μάθησης οι οποίοι κατά κύριο λόγω χρησιμοποιήθηκαν στις παραδοσιακές προσεγγίσεις επίλυσης προβλημάτων στον τομέα της Επεξεργασίας Φυσικής Γλώσσας. Με τον όρο παραδοσιακές προσεγγίσεις εννοούμε τις προσεγγίσεις εκείνες που χρησιμοποιούνταν πριν την επικράτηση των βαθιών νευρωνικών δικτύων . Οι αλγόριθμοι που θα ασχοληθούμε στην ενότητα αυτή είναι :

- Αλγόριθμος k Nearest Neighbors (kNN) [9]
- Αλγόριθμος Μηχανών Διανυσμάτων Υποστήριξης (SVM) [8]

2.2.1 Αλγόριθμος k Nearest Neighbors (kNN)

Στον τομέα της αναγνώρισης προτύπων, ο αλγόριθμος k-Nearest Neighbours (kNN)[9] είναι μια μη-παραμετρική μέθοδος που χρησιμοποιείται για classification και regression και προτάθηκε από τον Thomas Cover. Τόσο στην περίπτωση του classification όσο και του regression η είσοδος αποτελείται από τα k κοντινότερα δείγματα στον χώρο χαρακτηριστικών (feature space). Η έξοδος εξαρτάται από το κατά πόσο ο αλγόριθμος kNN χρησιμοποιείται για classification ή regression:

- Στην περίπτωση του kNN για classification, η έξοδος είναι η κλάση στην οποία ταξινομείται το αντικείμενο. Η ταξινόμηση ενός αντικειμένου πραγματοποιείται βάσει των κλάσεων που ανήκουν οι γείτονες του. Πιο συγκεκριμένα, ανατίθεται στην κλάση που ανήκουν οι περισσότεροι από τους k κοντινότερους γείτονές του (όπου το k είναι θετικός, σχετικά μικρός ακέραιος). Αν το $k = 1$ τότε το αντικείμενο ταξινομείται στην κλάση που ανήκει ο κοντινότερος γείτονας του.
- Στην περίπτωση του kNN regression, η έξοδος είναι η τιμή ιδιότητας για το αντικείμενο. Η τιμή αυτή είναι ο μέσος όρος των τιμών των k κοντινότερων γειτόνων.

Ο αλγόριθμος kNN[6],[9] αποτελεί έναν αλγόριθμο της κατηγορίας lazy learning, καθώς η συνάρτηση εκτιμάται μόνο τοπικά και όλοι οι υπολογισμοί πραγματοποιούνται κατά την φάση της αποτίμησης της συνάρτησης. Δεδομένου ότι ο αλγόριθμος βασίζεται στην απόσταση μεταξύ των σημείων για την ταξινόμηση αυτών, η κανονικοποιήση του συνόλου εκπαίδευσης θα μπορούσε να οδηγήσει σε δραματική βελτίωση των μετρικών αναφοράς. Μάλιστα, δεδομένου ότι αλγόριθμος kNN είναι ιδιαίτερα απλός μια πιθανά χρήσιμη προέκταση του είναι η ανάθεση διαφορετικών βαρών στην συμβολή των γειτόνων, έτσι ώστε οι πλησιέστεροι γείτονες να συνεισφέρουν περισσότερο από αυτούς που εντοπίζονται πιο μακριά. Για παράδειγμα, ένα συχνό σχήμα ανάθεσης βαρών αποτελείται από το να δίνεται σε κάθε γείτονα το βάρος $\frac{1}{d}$, όπου d είναι η απόσταση του γείτονα από το αντικείμενο. Επιπρόσθετα, θεωρούμε σκόπιμο να διευκρινίσουμε ότι οι γείτονες λαμβάνονται από ένα σύνολο αντικειμένων στο οποίο η κλάση (όταν μιλάμε για classification) ή η τιμή ιδιότητας του αντικειμένου (όταν μιλάμε για regression) είναι γνωστά. Η διαδικασία αυτή μπορεί να θεωρηθεί ως η διαδικασία εκπαίδευσης του αλγορίθμου, αν και πρέπει να διευκρινίσουμε ότι δεν χρειάζεται κάποια ειδική διαδικασία εκπαίδευσης. Τέλος, μία ιδιαιτερότητα του αλγορίθμου αυτού είναι ότι είναι ευαίσθητος στην τοπικότητα που παρουσιάζουν τα δεδομένα του συνόλου εκπαίδευσης.

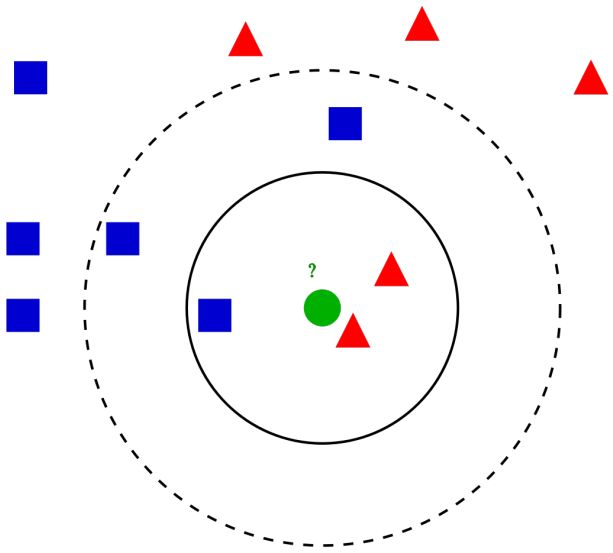
2.2.1.1 Περιγραφή αλγορίθμου kNN

Τα δείγματα του συνόλου δεδομένων εκπαίδευσης είναι πολυδιάστατα διανύσματα, όπου το καθένα διαθέτει μια ετικέτα κλάσης. Η φάση εκπαίδευσης του αλγορίθμου αποτελείται μόνο από την αποθήκευση των πολυδιάστατων διανυσμάτων και ετικετών κλάσεων των δεδομένων του συνόλου εκπαίδευσης. Κατά την φάση της ταξινόμησης (classification), η μεταβλητή k είναι σταθερή και ορισμένη από τον χρήστη, ένα "άγνωστο" διάνυσμα ταξινομείται με βάση την συχνότερα εμφανιζόμενη ετικέτα κλάσης ανάμεσα στους k κοντινότερους γείτονες (από το σημείο προς ταξινόμηση) του συνόλου εκπαίδευσης.

Στο σημείο αυτό θεωρούμε σκόπιμο να δώσουμε περαιτέρω πληροφορίες σχετικά με τις μεθόδους υπολογισμού απόστασης μεταξύ δύο σημείων. Μία συχνά χρησιμοποιούμενη μετρική απόστασης για συνεχείς μεταβλητές είναι η ευκλείδεια απόσταση. Για διακριτές μεταβλητές, όπως παρατηρούνται κατά την ταξινόμηση φυσικής γλώσσας, μπορεί να χρησιμοποιηθεί η μετρική των επικαλύψεων (Hamming distance). Τέλος, η επίδοση του αλγόριθμου kNN στην ταξινόμηση ετικετών θα μπορούσε να βελτιωθεί σημαντικά αν η μετρική απόστασης προκύψει από ειδικούς αλγορίθμους όπως είναι ο *Large Margin Nearest Neighbor* ή ο *Neighbourhood components analysis*.

Βέβαια, αξίζει να σημειώσουμε ότι ένα μειονέκτημα της ταξινόμησης με βάση την πλειοψηφία προκύπτει αν ένα σύνολο δεδομένων είναι μη ισορροπημένο αναφορικά με τις κλάσεις του. Πιο συγκεκριμένα, τέτοια συμπεριφορά μπορεί να παρατηρηθεί όταν παραδείγματα μίας πιο συχνά εμφανιζόμενης κλάσης τείνουν να κυριαρχούν στην πρόβλεψη ενός νέου παραδείγματος, καθώς λόγω του ότι έχουν πολύ μεγάλο πλήθος εμφανίσεων κυριαρχούν ανάμεσα στους k εμφανιζόμενους γείτονες. Ένας τρόπος για να αντιμετωπιστεί το πρόβλημα αυτό, όπως αναφέραμε και παραπάνω, είναι η χρήση βαρών απόστασης ανάμεσα στα σημεία, έτσι ώστε να δίνεται μεγαλύτερη σημασία στα κοντινότερα σημεία και μικρότερη στα σημεία που βρίσκονται πιο μακριά.

Στην συνέχεια, για την καλύτερη κατανόηση του αλγορίθμου αυτού θα παρουσιάσουμε ένα παράδειγμα χρήσης του αλγορίθμου kNN για την ταξινόμηση (classification) ετικετών.



Σχήμα 2.3: Παράδειγμα του αλγόριθμου kNN για ταξινόμηση [6]

Από το παραπάνω σχήμα παρατηρούμε ότι το σημείο προς ταξινόμηση (πράσινη βούλα) μπορεί να ταξινομηθεί είτε στην κλάση των μπλε τετραγώνων είτε των κόκκινων τριγώνων. Αν η τιμή του $k = 3$ (μικρός κύκλος) το πράσινο σημείο ταξινομείται στα κόκκινα τρίγωνα καθώς υπάρχουν 2 κόκκινα τρίγωνα και μόλις 1 μπλε τετράγωνο εντός του μικρού κύκλου. Αν η τιμή του $k = 5$ (μεγάλος κύκλος), τότε το πράσινο σημείο ταξινομείται στα μπλε τετράγωνα καθώς υπάρχουν 3 μπλε τετράγωνα και μόλις 2 κόκκινα τρίγωνα εντός του μεγάλου κύκλου.

2.2.1.2 Επιλογή βέλτιστων παραμέτρων του αλγορίθμου kNN

Στην υποενότητα αυτή θα ασχοληθούμε με την επιλογή των βέλτιστων παραμέτρων για τον αλγόριθμο kNN. Η βέλτιστη επιλογή αυτή, κατά κύριο λόγο εξαρτάται από το σύνολο δεδομένων που θα χρησιμοποιηθεί στον αλγόριθμο αυτό. Γενικά, μεγάλες τιμές του k, μειώνουν το φαινόμενο του θορύβου στο πρόβλημα της ταξινόμησης (classificaiton), αλλά κάνουν τα σύνορα μεταξύ των διαφορετικών κλάσεων λιγότερο διαχωρίσιμα. Μία καλή τιμή της παραμέτρου k μπορεί να επιλεχθεί με την χρήση διάφορων ευριστικών τεχνικών. Επιπρόσθετα, αξίζει να σημειώσουμε ότι η επίδοση του αλγορίθμου μπορεί να μειωθεί δραματικά από την ύπαρξη θορύβου ή μη σχετικών χαρακτηριστικών, ή αν το εύρος τιμών των χαρακτηριστικών δεν είναι συνεκτικό με την σημασία του εκάστοτε χαρακτηριστικού. Ως εκ τούτου, έχει υπάρξει σημαντική προσπάθεια έρευνας προς αυτή την κατεύθυνση (επιλογή ή επεξεργασία τιμής χαρακτηριστικών) έτσι ώστε να βελτιωθεί η επίδοση του αλγορίθμου. Μία ιδιαίτερα γνωστή τεχνική είναι η χρήση εξελικτικών αλγόριθμων για την βελτιστοποίηση της επεξεργασία της κλίμακας τιμών των χαρακτηριστικών. Τέλος, αξίζει να σημειώσουμε ότι στο πρόβλημα της δυαδική ταξινόμησης (υπάρχουν μόλις δύο κλάσεις), είναι βοηθητικό να επιλέξουμε μία περιττή τιμή σαν τιμή του k έτσι ώστε να αποφευχθούν οι ισοπαλίες. Για την επιλογή του βέλτιστου k σε αυτού του τύπου τα προβλήματα, χρησιμοποιείται συχνά η μέθοδος bootstrap.

2.2.2 Αλγόριθμος Μηχανών Διανυσμάτων Υποστήριξης (SVM)

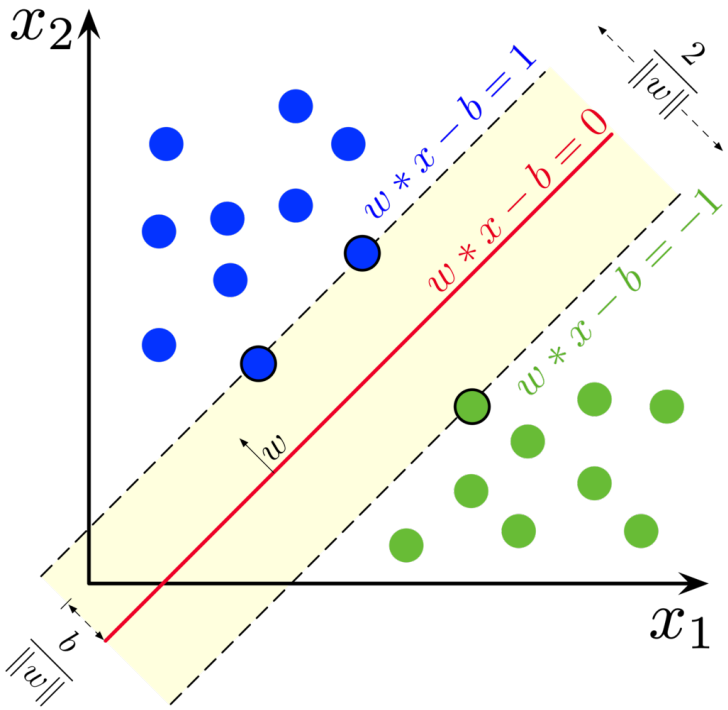
Έστω οτι έχουμε δύο γραμμικά διαχωρίσιμες κλάσεις και θέλουμε να σχεδιάσουμε ένα γραμμικό ταξινομητή [8],[30]. Στόχος μας είναι να σχεδιάσουμε ένα υπερεπίπεδο της μορφής :

$$f(x) = w^T b + x \tag{2.1}$$

όπου $x \in R^d$ τα προς εκπαίδευση δεδομένα και $w \in R^d$, $b \in R^d$ οι παράμετροι του ταξινομητή. Ακόμα, συμβολίζουμε με $y \in -1, 1$ τις ετικέτες των δύο κλάσεων.

Το πλήθος των υπερεπιπέδων που μπορούν να ταξινομήσουν όλα τα δεδομένα εκπαίδευσης x με επιτυχία είναι άπειρο. Παρόλα αυτά, η επιλογή θα πρέπει να γίνει με τέτοιο τρόπο έτσι ώστε ο ταξινομητής να είναι αποδοτικός ως προς τη γενίκευση δεδομένων εκτός του συνόλου εκμάθησης. Το βέλτιστο υπερεπίπεδο (optimal hyperplane) είναι αυτό που μεγιστοποιεί το περιθώριο (margin) ανάμεσα στις δύο κλάσεις. Σε αυτό το σημείο σημειώνεται πως η απόσταση ενός σημείου x από το υπερεπίπεδο δίνεται από την σχέση :

$$z = \frac{|f(x)|}{||w||} \tag{2.2}$$



Σχήμα 2.4: Support Vector Machine (SVM) [7]

Δεδομένου ότι οι δύο κλάσεις είναι γραμμικά διαχωρίσιμες, μπορούν να επιλεγούν παράμετροι w και b τέτοιοι ώστε :

$$f(x) > 0 \; \forall x \; while \; y = +1 \tag{2.3}$$

$$f(x) < 0 \; \forall x \; while \; y = -1 \tag{2.4}$$

Κλιμακώνοντας το w μπορούμε να απαιτήσουμε η f(x) στα πλησιέστερα σημεία των δύο κλάσεων να είναι +1 και -1 για κάθε κλάση αντίστοιχα. Με αυτή την απαίτηση το μέγιστο μήκος περιθωρίου γίνεται $\frac{2}{||w||}$ και έχουμε

$$f(x) \geq 1 \; \forall x \; while \; y = +1 \tag{2.5}$$

$$f(x) \leq -1 \; \forall x \; while \; y = -1 \tag{2.6}$$

Το πρόβλημα πλέον ανάγεται στην εύρεση των παραμέτρων w και b ούτως ώστε να ελαχιστοποιείται η συνάρτηση

$$J(w, b) = \frac{2}{||w||} \tag{2.7}$$

υπό τους περιορισμούς

$$y_i(w^T x_i + b) \geq 1, i = 1, 2, ..., N \tag{2.8}$$

Η λύση του τετραγωνικού προβλήματος γραμμικού προγραμματισμού (Linear Programming) δίνεται με τη χρήση πολλαπλασιαστών Lagrange

2.3 Διανυσματικές Αναπαραστάσεις Λέξεων

Στην ενότητα αυτή, θα ασχοληθούμε με έναν ιδιαίτερα σημαντικό τομέα της Επεξεργασίας Φυσικής Γλώσσας. Ο τομέας αυτός σχετίζεται με την εύρεση μίας κατάλληλης μορφής αναπαράστασης των λέξεων έτσι ώστε να μπορούν να γίνουν κατανοητές από υπολογιστικά συστήματα και κατ' επέκταση να χρησιμοποιηθούν σε νευρωνικά δίκτυα. Αρχικά, η βιβλιογραφία προσπάθησε να προσεγγίσει το πρόβλημα αυτό με την δημιουργία ηλεκτρονικών εγκυκλοπαιδικών θησαυρών (WordNet[20]) στους οποίους προσπαθούσαν να αποθηκεύσουν πληροφορίες όπως η σημασία, τα συνώνυμα και τα αντώνυμα όλων των λέξεων μίας γλώσσας. Όπως εύκολα καταλαβαίνει κανείς, η προσπάθεια αυτή, αν και αρχικά γνώρισε μία κάποια αποδοχή, εν τέλει εγκαταλείφθηκε καθώς είχε σημαντικά μειονεκτήματα. Μερικά από τα μειονεκτήματα αυτά σχετίζονται με την πολύ δύσκολη συντήρηση μίας τέτοιας βάσης δεδομένων, τόσο γιατί προκύπτουν πολύ συχνά νέες λέξεις όσο και γιατί προκύπτουν συχνά νέες σημασίες μία λέξης, καθώς και με το γεγονός ότι δεν είναι εύκολο κανείς να υπολογίσει την ομοιότητα μεταξύ δύο λέξεων. Στην συνέχεια, θα παραθέσουμε διάφορες προσεγγίσεις που δοκιμάστηκαν για να επιλύσουν το πρόβλημα αυτό.

2.3.1 One-hot vectors

Στην υποενότητα αυτή, θα σχολιάσουμε την προσέγγιση η οποία είναι γνωστή ως one-hot vectors η the localist represantation. Κατά την προσέγγιση αυτή, κάθε λέξη ενός λεξιλογίου αναπαρίσταται από ένα διάνυσμα το οποίο αποτελείται από τιμές 0 και 1. Το διάνυσμα αυτό είναι διαστάσεων $|V| \times 1$, όπου V είναι το πλήθος των μοναδικών λέξεων του λεξιλογίου. Κάθε μοναδική λέξη του λεξιλογίου, αναπαρίσταται από ένα μοναδικό διάνυσμα το οποίο έχει μόνο μία τιμή του ίση με 1 και όλες τις υπόλοιπες τιμές ίσες με 0. Για την καλύτερη κατανόηση της προσέγγισης αυτής θεωρούμε σκόπιμο να παραθέσουμε ένα παράδειγμα. Έχοντας ένα λεξιλόγιο που αποτελείται από τις λέξεις $hotel, motel, pansion$ τα μοναδικά διανύσματα των λέξεων αυτών φαίνονται παρακάτω :

$$x_{hotel} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, x_{motel} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, x_{pansion} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

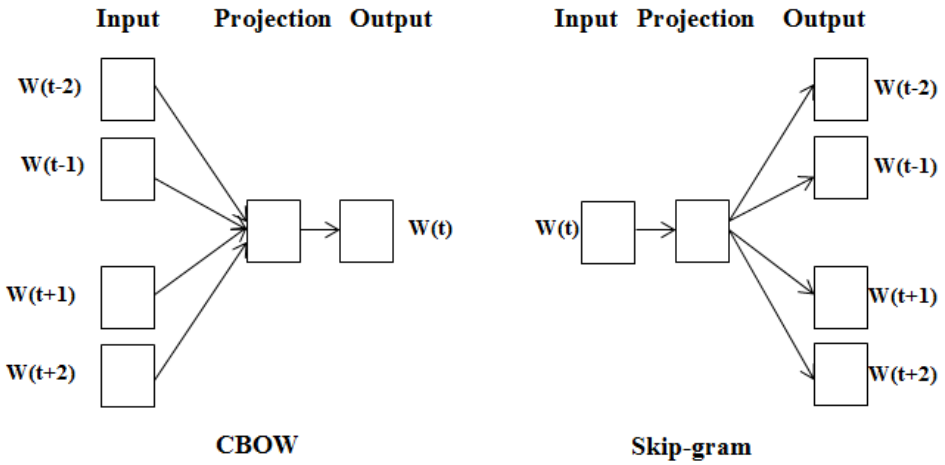
Δυστυχώς, αν και η μέθοδος αυτή είναι ιδιαίτερη απλή, παρουσιάζει κάποια πολύ σημαντικά μειονεκτήματα. Αρχικά, οι διαστάσεις των διανυσμάτων εξαρτώνται από το πλήθος των μοναδικών λέξεων της φυσικής γλώσσας και όπως είναι γνωστό οι φυσικές γλώσσες αποτελούνται από εκατοντάδες έως και χιλιάδες λέξεις γεγονός που συνεπάγεται πολύ μεγάλες διαστάσεις διανυσμάτων. Ως εκ τούτου, η μέθοδος αυτή χαρακτηρίζεται μη αποδοτική καθώς απαιτεί πολλούς υπολογιστικούς πόρους. Υπάρχει όμως ένα σοβαρότερο μειονέκτημα από την αποδοτικότητα της μεθόδου. Πιο συγκεκριμένα, οι λέξεις που παραθέσαμε στο παραπάνω παράδειγμα είναι συνώνυμες, ή έστω κατά κάποιο τρόπο μοιάζουν νοηματικά. Με τα διανύσματα όμως αυτά η νοηματική αυτή συγγένεια δεν διατηρείται καθώς αν υπολογίσουμε το γινόμενο μεταξύ δύο διανυσμάτων η τιμή τους θα είναι πάντα 0 (πχ ${x_{hotel}}^T \times x_{motel} = 0$). Το πρόβλημα αυτό είναι πολύ σημαντικό και αποτελεί το κίνητρο της επόμενης προσέγγισης.

2.3.2 Word Embeddings

Τα *word embeddings* αποτελούν διανυσματικές αναπαραστάσεις λέξεων. Πιο συγκεκριμένα, η κάθε λέξη αναπαρίσταται στον χώρο σαν ένα διάνυσμα συνεχών τιμών (το διάνυσμα αυτό συνήθως είναι είτε 100 είτε 300 διαστάσεων[32]). Το γεγονός αυτό επιτρέπει την πραγματοποίηση πράξεων μεταξύ των λέξεων, οι οποίες επιτρέπουν την εξαγωγή χρήσιμων συμπερασμάτων. Για παράδειγμα, μπορεί κανείς να υπολογίσει την ομοιότητα μεταξύ δύο λέξεων με βάση τα διανύσματα αυτά. Επιπρόσθετα, οι διαφορετικές διαστάσεις από τις οποίες αποτελείται ένα τέτοιο διάνυσμα θεωρείται ότι δίνουν μεγαλύτερες δυνατότητες εκφραστικότητας. Τα διανύσματα αυτά δημιουργούνται με διάφορους τρόπους όπως είναι η χρήση Νευρωνικών δικτύων[21] και ο συνδυασμός Νευρωνικών δικτύων και στατιστικών μεθόδων[31].

2.3.3 Ο αλγόριθμος Word2Vec

Στη βιβλιογραφία παρατηρούνται διάφορες προσεγγίσεις αλγορίθμων που δημιουργούν word embeddings[30] αλλά αυτή που στιγμάτισε τον κλάδο της Επεξεργασίας Φυσικής Γλώσσας είναι το Word2Vec[21] του Tomas Mikolov. Ο αλγόριθμος αυτός βασίζεται σε μία πολύ απλή ιδιότητα της φυσικής γλώσσας. Πιο συγκεκριμένα, γνωρίζουμε ότι αν ξέρουμε τα συμφραζόμενα γύρω από τα οποία χρησιμοποιείται μία λέξη τότε γνωρίζουμε και την σημασία της λέξης αυτής. Για την καλύτερη κατανόησή του παρουσιάζονται τα τέσσερα δομικά του στοιχεία. Το Continuous Bag of Words (CBOW) και το skip-gram αποτελούν τις δύο διαφορετικές εκδοχές του αλγόριθμου εκπαίδευσης ενώ το Negative Sampling[22] και Hierarchical Softmax αποτελούν δύο προεκτάσεις του αλγορίθμου αυτού για αποτελεσματικότερη εκπαίδευση.



Σχήμα 2.5: Οι δύο διαφορετική τρόποι εκπαίδευσης του αλγορίθμου Word2Vec [21]

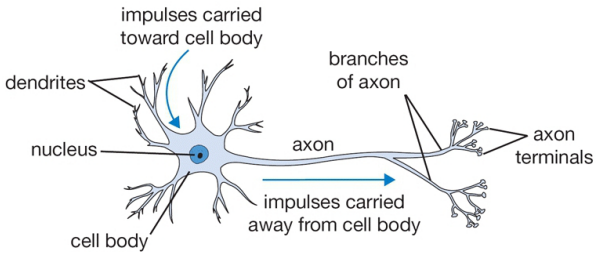
Όπως παρατηρείται στο παραπάνω σχήμα, ο αλγόριθμος CBOW δεδομένου του περιεχομένου (context) μιας λέξης, προσπαθεί να προβλέψει τη κεντρική αυτή λέξη. Αντιθέτως, στο Skip-gram μοντέλο, δεδομένης μιας κεντρικής λέξης επιχειρείται να προβλεφθεί η κατανομή των λέξεων που συνιστούν το περιεχόμενο (context) της κεντρικής λέξης. Επιπλέον, το Negative Sampling βασίζεται στη δειγματοληψία "αρνητικών" παραδειγμάτων ενώ το Hierarchical Softmax προτείνει μια αποδοτική δενδρική δομή για τον υπολογισμό των πιθανοτήτων των λέξεων του λεξικού.

2.4 Νευρωνικά Δίκτυα

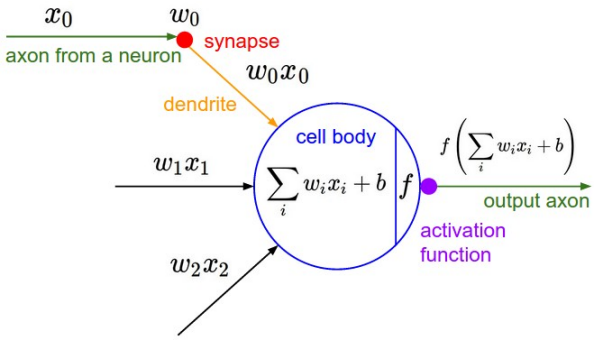
2.4.1 Ο Νευρώνας (Perceptron)

Βασική υπολογιστική μονάδα του εγκεφάλου αποτελεί ο νευρώνας[30]. Ένας νευρώνας δέχεται ένα ερέθισμα μέσω των δενδριτών και παράγει ένα σήμα εξόδου στον άξονά του. Ο άξονάς του με τη σειρά του συνδέεται μέσω συνάψεων με δενδρίτες άλλων νευρώνων. Στο υπολογιστικό μοντέλο, το σήμα των αξόνων διέρχεται από τις συνάψεις όπου και πολλαπλασιάζεται με κάποιο βάρος. Τα νέα σήματα έχοντας δεχθεί αυτή την επεξεργασία διέρχονται στο σώμα του νευρώνα όπου και αθροίζονται και το αποτέλεσμα διέρχεται από μία συνάρτηση ενεργοποίησης (activation function) η οποία αναπαριστά τη συχνότητα των σημάτων αυτών στον άξονα. Η παραπάνω διαδικασία συνοψίζεται από την φόρμουλα :

$$a = f(\sum_{i=1}^{n} w_i * x_i + b_i) \tag{2.9}$$

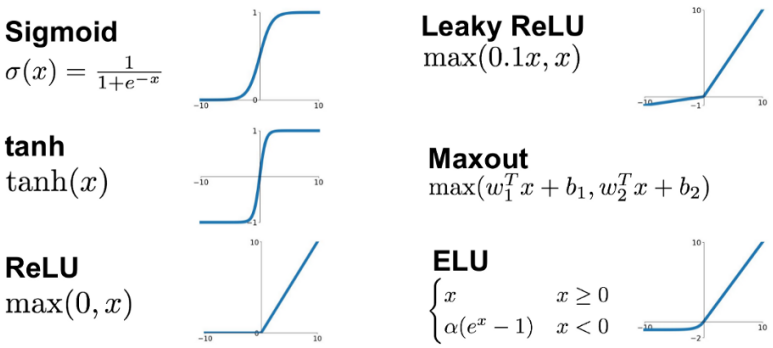


Σχήμα 2.6: Ο νευρώνας (Perceptron) [24]



Σχήμα 2.7: Το μοντέλο νευρώνας [24]

Όσον αφορά τις συναρτήσεις ενεργοποίησης (activation functions) συνήθως χρησιμοποιούνται η σιγμοειδής συνάρτηση, η υπερβολική εφαπτομένη, η Rectified Linear Unit (ReLU)[19] ,η Leaky ReLU[25], ή η ELU[26]. Αξίζει να σημειώσουμε, ότι χάρη στις συναρτήσεις ενεργοποίησης δίνεται η δυνατότητα στον νευρώνα να μάθει ενδιαφέρουσες συσχετίσεις. Στην συνέχεια, για την καλύτερη κατανόηση αυτών παραθέτουμε μία σύνοψη που τις παρουσιάζει.



Σχήμα 2.8: Συναρτήσεις Ενεργοποίησης [23]

2.4.2 Πολυστρωματικοί Νευρώνες (Multilayer Perceptron)

Σύνδεση μεταξύ δύο νευρώνων υφίσταται, όταν η έξοδος του ενός χρησιμοποιείται σαν είσοδος του άλλου. Η σύνδεση πολλών νευρώνων μεταξύ τους μπορεί να δημιουργήσει μια συνάρτηση που να έχει τη δυνατότητα να λύση μη γραμμικά διαχωρίσιμα προβλήματα. Μια από τις συνηθισμένες κατασκευές είναι το Perceptron Πολλών Επιπέδων (Multiple Layer Perceptron, MLP). Στην τοπολογία αυτού του τύπου, οι νευρώνες διαμερίζονται σε επίπεδα (layers), και τα γειτονικά επίπεδα συνδέονται με συνδέσεις μονής κατεύθυνσης. Ουσιαστικά, κάθε νευρώνας του i επιπέδου δέχεται σαν είσοδο τις εξόδους από κάθε νευρώνα του i − 1 επιπέδου. Η έξοδός του, τροφοδοτείται στους νευρώνες του επόμενου επιπέδου, ή στην έξοδο του δικτύου αν ο νευρώνας ανήκει στο τελευταίο επίπεδο. Για ένα δίκτυο με m επίπεδα, για την έξοδο του i-οστού επιπέδου, όπου: 0 < i < m,ισχύει:

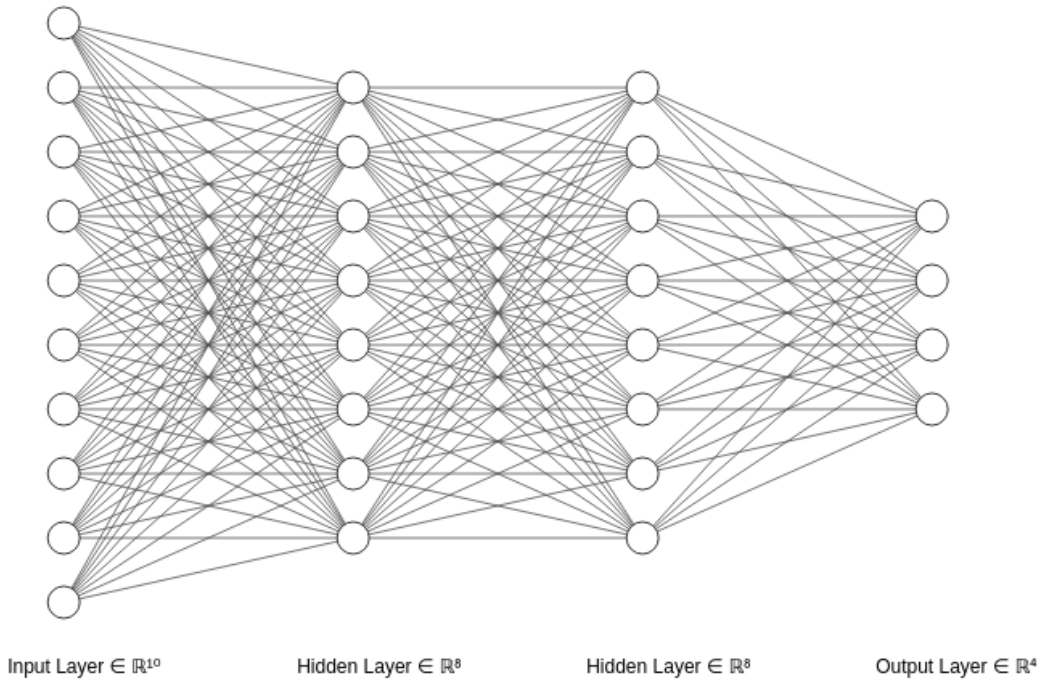
$$h^{(i)} = g^{(i)}(W^{(i)}h^{(i-1)} + b^{(i)}) = g^{(i)}(h^{(i-1)}; j^{(i)}) \tag{2.10}$$

όπου: $h^{(i)}$ η έξοδος του επιπέδου, $g^{(i)}$ η συνάρτηση ενεργοποίησης των νευρώνων του επιπέδου, $W^{(i)}$ ο πίνακας των βαρών των συνδέσεων και $b^{(i)}$ το διάνυσμα της προκατάληψης.

Όμοια,

$$h^{(0)} = g^{(0)}(x; \theta^{(0)}) \, and \, h^{(m-1)} = g^{(0)}(h^{(m-2)}; \theta^{(m-1)}) \tag{2.11}$$

Τα επίπεδα 0 < i < m − 1 ονομάζονται κρυφά επίπεδα (hidden layers) αφού η συμβολή τους στο δίκτυο δεν είναι εμφανής εκτός του μοντέλου. Τα επίπεδα i = 0 και i = m − 1 ονομάζονται επίπεδα εισόδου και εξόδου, αντίστοιχα. Γενικά, ένα δίκτυο ονομάζεται βαθύ (deep) αν $m > 3$, δηλαδή έχει τουλάχιστον ένα κρυφό επίπεδο.



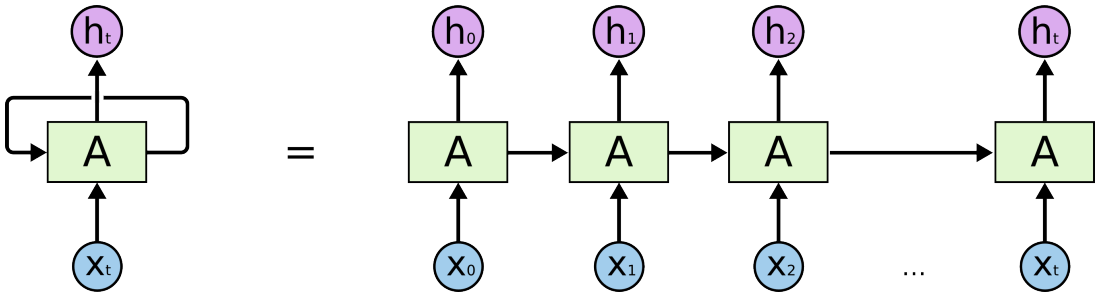
Σχήμα 2.9: Παράδειγμα Multilayer Perceptron με δύο hidden layer [27]

2.4.3 Αναδρομικά Νευρωνικά δίκτυα (RNN)

Τα αναδρομικά νευρωνικά δίκτυα είναι μία ειδική κατηγορία νευρωνικών δικτύων που επεξεργάζονται αποτελεσματικά κάθε είδους ακολουθιακά δεδομένα, με κύριο γνώρισμα τους την ιδιότητα τους να συνδέουν προηγούμενες εισόδους με την εκάστοτε τρέχουσα. Με αυτό τον τρόπο, ένα κόμβος του δικτύου έχει τη δυνατότητα σε κάθε χρονική στιγμή να παίρνει τα τρέχοντα δεδομένα εισόδου, παράλληλα με τις τιμές των κρυμμένων κόμβων, συλλέγοντας έτσι πληροφορίες από προηγούμενες χρονικές στιγμές. Παραδείγματα ακολουθιακών δεδομένων αποτελούν η φωνή, η γραφή, η οπτική πληροφορία που προκύπτει από μία κίνηση ή μία δράση ή ακόμα και τα pixel[29] μίας εικόνας αν τα διατρέξουμε με κάποιο δομημένο τρόπο. Στο πλαίσιο της παρούσας εργασίας θα χρησιμοποιήσουμε τα RNNs για προβλήματα που σχετίζονται με κείμενο. Αξίζει να σημειώσουμε ότι στον τομέα αυτό τα RNNs έχουν δείξει αποδειχθεί ιδιαίτερα αποτελεσματικά. Στην συνέχεια, θα μελετήσουμε τις εξισώσεις[30] που εκφράζουν τα RNNs την χρονική στιγμή t:

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}X_t + b_h) \tag{2.12}$$

όπου,$W_{hh} \in R^{H\times H}$ είναι ο πίνακας βαρών των hidden units (H είναι το μέγεθος των κρυφών διαστάσεων), $h_{t-1} \in R^{N\times H}$ είναι η κρυφή κατάσταση της προηγούμενης χρονικής στιγμής (N είναι το μέγεθος του batch), $W \times h \in R^{E\times H}$ είναι ο πίνακας βαρών των στοιχείων της εισόδου (E είναι οι διαστάσεις των $embeddings$), $X_t \in R^{N\times E}$ είναι η είσοδος την χρονική στιγμή t, $b_h \in R^{H\times 1}$ είναι το hidden units bias και τέλος h_t είναι η έξοδος του Αναδρομικού Νευρωνικού Δικτύου την χρονική στιγμή t.



Σχήμα 2.10: Recurrent Neural Networks (RNN) [5]

Η στιγμή t = 0 είναι η μοναδική χρονική στιγμή όπου το δίκτυο δέχεται μία μόνο είσοδο. Ως h_{t-1} θα μπορούσαμε να χρησιμοποιήσουμε ένα μηδενικό διάνυσμα (χωρίς εκμάθηση) ή κάποια αρχικοποιημένη τιμή (με εκμάθηση). Οι παράμετροι που θα πρέπει να μάθει το Αναδρομικό Νευρωνικό δίκτυο είναι οι πίνακες W_{hh} και $W \times h$ (και b_h).

2.4.4 Long Short Term Memory (LSTM)

Τα LSTMs[10] αποτελούν μια προέκταση των αναδρομικών Νευρωνικών δικτύων και αποσκοπούν στην επίλυση ενός πολύ σημαντικού περιορισμού που παρουσιάζουν τα RNNs. Πιο συγκεκριμένα, τα RNNs αντιμετωπίζουν προβλήματα στο να διατηρούν πληροφορία σε μεγάλες ακολουθίες. Έχει παρατηρηθεί πως η τελική έξοδος του RNN επηρεάζεται κυρίως από τα τελευταία timesteps της ακολουθίας και ελάχιστα ή καθόλου από αυτά που βρίσκονται στην αρχή αυτής. Το φαινόμενο αυτό είναι γνωστό στην βιβλιογραφία ως $vanishing\ gradient\ problem$. Τα LSTMs, λειτουργούν και αυτά όπως τα αναδρομικά δίκτυα που περιγράψαμε παραπάνω με μόνη διαφορά ότι το cell του μοντέλου είναι αρκετά πιο περίπλοκο από αυτό των RNNs. Ο περίπλοκος αυτός σχεδιασμός του cell είναι και ο λόγος για τον οποίο τα LSTMs καταφέρνουν να διατηρούν πληροφορίες κατά μήκος μίας μεγάλης ακολουθίας. Στην συνέχεια, παραθέτουμε αναλυτικά τις εξισώσεις που εκφράζουν το cell ενός LSTM.

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)\ \ (Forget\ Gate) \tag{2.13}$$

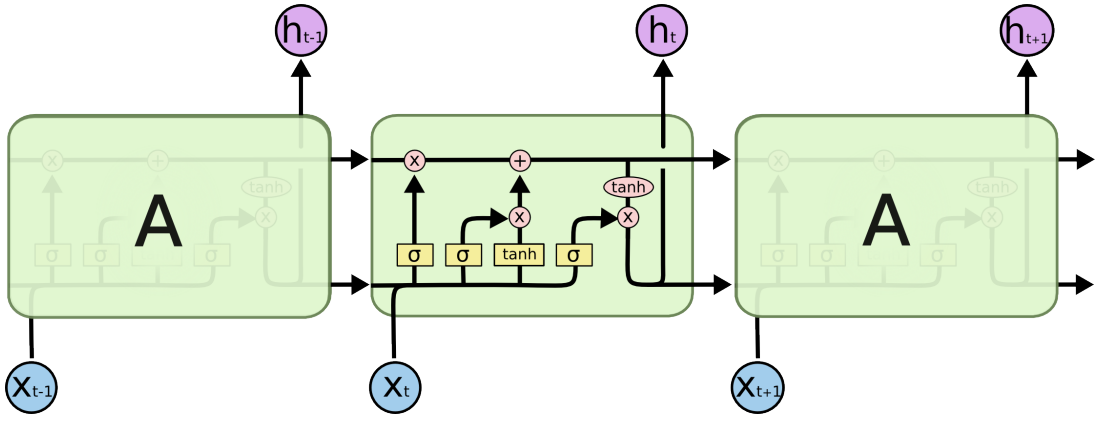
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)\ \ (Input\ Gate) \tag{2.14}$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)\ \ (Output\ Gate) \tag{2.15}$$

$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)\ \ (Cell\ input) \tag{2.16}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t\ \ (Cell\ State) \tag{2.17}$$

$$h_t = o_t \circ \sigma_h(c_t)\ \ (Hidden\ State) \tag{2.18}$$



Σχήμα 2.11: Long Short Term Memory (LSTM) [5]

2.4.5 Στρώμα Προσοχής (Attention Layer)

Όπως αναφέραμε παραπάνω, τα RNNs μαστίζονται από ένα πρόβλημα γνωστό στην βιβλιογραφία ως $vanishing\ gradient\ problem$. Το πρόβλημα αυτό επιλύεται ως ένα βαθμό από τα LSTMs τα οποία όντως λειτουργούν πολύ καλύτερα από τα RNNs σε ακολουθίες μεγάλου μήκους. Δυστυχώς, ακόμα και τα LSTMs όμως δεν καταφέρουν να λύσουν πλήρως το πρόβλημα αυτό καθώς ακόμα και αυτά δυσκολεύονται να μάθουν συσχετίσεις σε μεγάλες ακολουθίες. Για τον σκοπό αυτό, στην βιβλιογραφία προτάθηκε η χρήση ενός στρώματος μετά από ένα στρώμα LSTM το οποίο λύνει ακριβώς αυτό το πρόβλημα. Το στρώμα αυτό δέχεται όλες τις εξόδους του LSTM, για κάθε χρονική στιγμή t της ακολουθίας, και με βάση τις εξόδους αυτές κρίνει σε ποία ή ποίες χρονικές στιγμές πρέπει να δώσει προσοχή καθώς και μπορεί να εντοπίσει συσχετίσεις οι οποίες απέχουν μεταξύ τους πολλά ακολουθιακά βήματα. Το στρώμα αυτό είναι γνωστό ως στρώμα προσοχής (Attention Layer)[28] και στην βιβλιογραφία έχουν προταθεί διάφορες παραλλαγές του. Στην παρούσα εργασία, θα χρησιμοποιήσουμε το Attention Layer που προτάθηκε μαζί με το Hierarchical Attention Model[17]. Στην συνέχεια, για την καλύτερη κατανόηση του στρώματος προσοχής θα παραθέσουμε τις μαθηματικές σχέσεις που το εκφράζουν.

Αρχικά, μας δίνεται η έξοδος του LSTM σε κάθε χρονική στιγμή (h_{it}). Με βάση αυτή επιθυμούμε να υπολογίσουμε το σκορ προσοχής κάθε λέξης (a_{it}). Για τον υπολογισμό αυτό, θα ορίσουμε τρεις εκπαιδευόμενες παραμέτρους : έναν πίνακα βαρών (W_w), ένα διάνυσμα προκατάληψης (b_w) καθώς και ένα διάνυσμα συμφραζομένων (u_w).

$$u_{it} = tanh(W_w h_{it} + b_w) \tag{2.19}$$

$$a_{it} = \frac{exp(u_{it} u_w)}{\sum_t exp(u_{it} u_w)} \tag{2.20}$$

Τέλος, θα υπολογίσουμε το σταθμισμένο διάνυσμα μέσων όρων (s_i):

$$s_i = \sum_t a_{it} h_{it} \tag{2.21}$$

Στο σημείο αυτό θεωρούμε σκόπιμο να σημειώσουμε μερικούς από τους λόγους που το στρώμα προσοχής έχει επικρατήσει στον τομέα της Επεξεργασίας Φυσικής Γλώσσας. Αρχικά, ο υπολογισμός του μπορεί να πραγματοποιηθεί παράλληλα γεγονός που επιτρέπει την επιτάχυνση του από κάρτες γραφικών. Επιπρόσθετα, λύνει σε μεγάλο βαθμό το πρόβλημα του $vanishing\ gradient\ problem$. Τέλος, επιτρέπει την ερμηνεία του μοντέλου από τον προγραμματιστή καθώς με βάση τα σκορ a_{it} που υπολογίσαμε παραπάνω μπορεί κανείς εύκολα να "δει" την σημασία της κάθε λέξης μίας ακολουθίας για την επίτευξη του τελικού στόχου. Το κομμάτι της ερμηνείας των νευρωνικών δικτύων είναι από τα πλέον σημαντικά καθώς είναι επιτακτικό να μπορεί κανείς να ερμηνεύσει τους λόγους για τους οποίους ένα μοντέλο είναι αποτελεσματικό.

2.4.6 Κανονικοποιήση Παρτίδας (Batch Normalization)

Η κανονικοποιήση παρτίδας (Batch Normalization)[1],[11] (επίσης γνωστή ως Batch Norm) είναι μια μέθοδος που χρησιμοποιείται για να κάνει τα τεχνητά νευρωνικά δίκτυα ταχύτερα και πιο σταθερά μέσω της κανονικοποίησης του στρώματος εισόδου. Κατά την κανονικοποιήση αυτή αλλάζει τόσο η κατανομή των δεδομένων όσο και η κλίμακα τιμών τους. Ενώ το αποτέλεσμα της batch normalization είναι εμφανές, οι λόγοι πίσω από την αποτελεσματικότητά της παραμένουν υπό συζήτηση. Αρχικά, η βιβλιογραφία[11] υποστήριξε ότι η μέθοδος αυτή μπορεί να μετριάσει το πρόβλημα της εσωτερικής μεταβλητής συνδιακύμανσης (internal covariance shift), όπου η αρχικοποίηση των παραμέτρων και οι αλλαγές στην κατανομή των εισόδων κάθε επιπέδου επηρεάζουν το ρυθμό εκμάθησης του δικτύου. Πρόσφατα, η θεωρία αυτή αμφισβητήθηκε από ένα μέρος της βιβλιογραφίας[12] καθώς υποστηρίχτηκε ότι η batch normalization δεν μετριάζει το πρόβλημα της εσωτερικής μεταβλητής συνδιακύμανσης, αλλά απλά λειαίνει την αντικειμενική συνάρτηση του δικτύου οδηγώντας έτσι σε βελτίωση της επίδοσης.

Σε ένα νευρωνικό δίκτυο, η batch normalization επιτυγχάνεται μέσω ενός βήματος κανονικοποίησης που καθορίζει τα μέσα και τις διακυμάνσεις των εισόδων κάθε στρώματος. Στην ιδανική περίπτωση, η κανονικοποιήση θα μπορούσε να πραγματοποιηθεί σε ολόκληρο το σύνολο εκπαίδευσης, αλλά η χρησιμοποίηση του βήματος αυτού σε συνδυασμό με τις στοχαστικές μεθόδους βελτιστοποίησης είναι μη πρακτική. Έτσι, η κανονικοποιήση περιορίζεται σε κάθε mini batch κατά την διαδικασία εκπαίδευσης.

Θα χρησιμοποιήσουμε την μεταβλητή B για να ορίσουμε ένα mini batch μεγέθους m του συνολικού συνόλου εκπαίδευσης. Ο εμπειρικός μέσος όρος και η διακύμανση του B μπορούν έτσι να οριστούν ως εξής :

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.22}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.23}$$

Για ένα στρώμα (Layer) του δικτύου με είσοδο d διαστάσεων, $x = (x^{(1)}, ..., x^{(d)})$, η κάθε διάσταση της εισόδου κανονικοποιείται ξεχωριστά, $\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}}$, όπου $k \in [1, d]$ και $i \in [1, m]$; $\mu_B^{(k)}$ και $\sigma_B^{(k)^2}$ είναι ο μέσος όρος και η διακύμανση κάθε διάστασης αντίστοιχα. Η τιμή ϵ προστίθεται στον παρανομαστή έτσι ώστε να διασφαλίσει την περίπτωση της διαίρεσης με την τιμή 0. Η τελική κανονικοποιημένη ενεργοποίηση $\hat{x}^{(k)}$ έχει μηδενικό μέσο όρο και μοναδιαία διακύμανση, αν το ϵ δεν ληφθεί υπόψιν στους υπολογισμούς. Τέλος, η μετατροπή ολοκληρώνεται με την εξίσωση $y_i^{(k)} = \gamma^{(k)}\hat{x}_i^{(k)} + \beta^{(k)}$, όπου οι παράμετροι $\gamma^{(k)}$ και $\beta^{(k)}$ είναι παράμετροι που μαθαίνει το δίκτυο κατά την διαδικασία εκπαίδευσης.

2.5 Επιλογή μετρικών αξιολόγησης

Στην ενότητα αυτή θα ασχοληθούμε με την μετρικές αξιολόγησης που θα χρησιμοποιηθούν στην παρούσα εργασία. Αξίζει να σημειώσουμε ότι οι μετρικές αυτές είναι ιδιαίτερα σημαντικές καθώς η λάθος επιλογή αυτών θα μπορούσε να οδηγήσει σε λανθασμένη αξιολόγηση των πειραμάτων μας. Η επιλογή των μετρικών αξιολόγησης οφείλει να γίνεται με γνώμονα την φύση του προβλήματος[38]. Ως εκ τούτου, στην παρούσα εργασία ασχοληθήκαμε ιδιαίτερα με την επιλογή αυτών και καταλήξαμε στην επιλογή των παρακάτω μετρικών αξιολόγησης. Πριν προχωρήσουμε στην περιγραφή των μετρικών αξιολόγησης, θεωρούμε σκόπιμο να σημειώσουμε ότι η επιλογή αυτή έχει γίνει με κύριο γνώμονα το γεγονός ότι τα σύνολα δεδομένων που δημιουργήσαμε είναι μη ισορροπημένα (imbalanced data)[39] και διαθέτουν πολλαπλές κλάσεις (multiclass)[39].

2.5.1 Ακρίβεια (Precision) και Ανάκληση (Recall)

Στον τομέα της μηχανικής μάθησης, η ακρίβεια (ονομάζεται επίσης θετική προγνωστική τιμή) και η ανάκληση (γνωστή και ως ευαισθησία) είναι μετρικές που χρησιμοποιούνται συχνά για προβλήματα ταξινόμησης.

Η ακρίβεια (Precision)[16] είναι το κλάσμα των θετικών ετικετών που ταξινομήθηκαν σωστά στην θετική κλάση (True Positive) προς το πλήθος των ετικετών που ταξινομήθηκαν στην θετική κλάση και είτε όντως άνηκαν σε αυτή (True Positive) είτε άνηκαν στην αρνητική κλάση (False Positive). Πιο συγκεκριμένα :

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{2.24}$$

Από τα παραπάνω καταλαβαίνουμε ότι η ακρίβεια εκφράζει το ποσοστό των επιτυχημένων προβλέψεων της θετικής κλάσης ως προς τα δεδομένα που ταξινομήθηκαν στην θετική κλάση.

Η ανάκληση (Recall)[16] είναι το κλάσμα των θετικών ετικετών που ταξινομήθηκαν σωστά στην θετική κλάση (True Positive) προς το άθροισμα του πλήθους των ετικετών που ταξινομήθηκαν σωστά στην θετική κλάση (True Positive) και του πλήθους των ετικετών που ταξινομήθηκαν λάθος στην αρνητική κλάση (False Negative). Πιο συγκεκριμένα :

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{2.25}$$

Από τα παραπάνω καταλαβαίνουμε ότι η ανάκληση εκφράζει το ποσοστό των επιτυχημένων προβλέψεων της θετικής κλάσης ως προς το πλήθος των δεδομένων που ανήκουν στην κλάση αυτή.

Στο σημείο αυτό, αξίζει να σημειώσουμε ότι στην περίπτωση που το πρόβλημα έχει περισσότερες από 2 κλάσεις τότε ο υπολογισμός της ακρίβειας και της ανάκλησης γίνεται ξεχωριστά για κάθε κλάση. Πιο συγκεκριμένα, θεωρώντας την κάθε κλάση σαν την θετική κλάση και τις υπόλοιπες σαν αρνητικές υπολογίζουμε τις τιμές των Precision και Recall. Τέλος, η μέγιστη τιμή των Precision και Recall είναι η τιμή 1 ενώ η ελάχιστη η τιμή 0.

2.5.2 F1-score

Η μετρική F1[15] είναι ο αρμονικός μέσος όρος των μετρικών της ακρίβειας και της ανάκλησης. Η μετρική αυτή χρησιμοποιείται συχνά σε προβλήματα ταξινόμησης για να εκφράσει την επίδοση ενός μοντέλου σε ένα σύνολο δεδομένων. Μάλιστα, αξίζει να σημειώσουμε ότι το γεγονός ότι χρησιμοποιείται αρμονικός μέσος όρος και όχι απλός μέσος όρος γίνεται έτσι ώστε να "τιμωρείται" περισσότερο η μικρότερη τιμή μεταξύ της ακρίβειας και της ανάκλησης. Στην συνέχεια, παραθέτουμε την γενική φόρμουλα που υπολογίζει την τιμή της μετρικής αυτής :

$$F1\text{-}score = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall} \tag{2.26}$$

Η παράμετρος β είναι αυτή που ορίζει την σημασία που δίνεται στις τιμές της ανάκλησης και ακρίβειας. Στην γενική περίπτωση, η τιμή αυτή ισούται με την τιμή 1 έτσι ώστε να δίνεται ίση σημασία μεταξύ της ανάκλησης και της ακρίβειας. Ως εκ τούτου, η παραπάνω φόρμουλα παίρνει την μορφή :

$$F1\text{-}score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.27}$$

Η παραπάνω φόρμουλα εκφράζει τον υπολογισμό της μετρικής F1 για προβλήματα δυαδικής ταξινόμησης. Βέβαια, το πρόβλημα μας αποτελείται από περισσότερες από δύο κλάσεις γεγονός που συνεπάγεται ότι πρέπει να επεκτείνουμε τον υπολογισμό της μετρικής αυτής. Για το υπολογισμό της μετρικής F1 για προβλήματα ταξινόμησης πολλών κλάσεων υπάρχουν δύο γνωστές τεχνικές, η τεχνική macro averaging και η micro averaging. Στην παρούσα εργασία χρησιμοποιήσαμε την τεχνική macro averaging κατά την οποία αρχικά υπολογίζουμε τις τιμές της ανάκλησης και της ακρίβειας, υπολογίζοντας τον μέσο όρο των τιμών τους για κάθε κλάση, και στην συνέχεια με βάση τις τιμές αυτές υπολογίζουμε την τιμή της μετρικής. Ο λόγος που επιλέξαμε την τεχνική macro έναντι της micro, είναι γιατί δίνει την ίδια σημασία σε όλες τις κλάσεις σε αντίθεση με την micro που δίνει περισσότερη σημασία στις κλάσεις με περισσότερα στοιχεία. Τέλος, αξίζει να σημειώσουμε ότι η μέγιστη τιμή της μετρικής F1 είναι η τιμή 1 ενώ η ελάχιστη η τιμή 0.

2.5.3 Matthews correlation coefficient (MCC)

Η μετρική MCC [13, 14] χρησιμοποιείται συχνά στον τομέα της μηχανικής μάθησης σαν μία μετρική αξιολόγησης σε προβλήματα στα οποία τα σύνολα δεδομένων είναι ιδιαίτερα μη ισορροπημένα (imbalanced). Ένα τέτοιο παράδειγμα, είναι ένα σύνολο δεδομένων με ακτινογραφίες ασθενών με πνευμονία. Το 99% των ακτινογραφιών θα είναι από υγιείς ασθενείς ενώ μόλις το 1% αυτών (ή και λιγότερο) θα έχει πνευμονία γεγονός που καθιστά αναγκαία ειδική μέριμνα καθώς σε άλλη περίπτωση αν έναν μοντέλο μας λέει ότι όλες οι ακτινογραφίες ανήκουν σε υγιείς ανθρώπους το μοντέλο θα έχει 99% επιτυχία στις προβλέψεις. Η μαθηματική φόρμουλα που εκφράζει την μετρική (MCC) για το πρόβλημα της δυαδικής ταξινόμησης δίνεται παρακάτω :

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.28}$$

Βέβαια, δεδομένου ότι το πρόβλημα μας αποτελείται από περισσότερες από δύο κλάσεις είναι απαραίτητο να επεκτείνουμε την παραπάνω φόρμουλα για ταξινόμηση πολλαπλών κλάσεων (multi-class classification). Στην συνέχεια, παραθέτουμε την γενικευμένη φόρμουλα της μετρικής MCC που εφαρμόζεται σε τέτοιου τύπου προβλήματα :

$$MCC = \frac{\sum_k \sum_l \sum_m C_{kk} C_{lm} - C_{kl} C_{mk}}{\sqrt{\sum_k (\sum_l C_{kl})(\sum_{k'|k' \neq k} \sum_{l'} C_{k'l'})} \sqrt{\sum_k (\sum_l C_{lk})(\sum_{k'|k' \neq k} \sum_{l'} C_{l'k'})}} \tag{2.29}$$

Τέλος, αξίζει να σημειώσουμε ότι η ελάχιστη τιμή της μετρικής MCC (για multiclass classification) κυμαίνεται μεταξύ του -1 και 0, ανάλογα με την κατανομή των δεδομένων, ενώ η μέγιστη τιμή αυτής είναι η τιμή 1. Μάλιστα, όταν η τιμή της MCC ισούται με 0 μπορούμε να συμπεράνουμε ότι το μοντέλο δεν προβλέπει καλύτερα από έναν τυχαίο ταξινομητή.

Κεφάλαιο 3

Δημιουργία συνόλου δεδομένων (*Dataset*)

Στο κεφάλαιο αυτό θα ασχοληθούμε με την δημιουργία του συνόλου των δεδομένων (*Dataset*) που θα χρησιμοποιήσουμε για την επίλυση του προβλήματος της ταξινόμησης πηγαίου κώδικα σε αλγορίθμους. Για την δημιουργία του συνόλου αυτού θα πραγματοποιήσουμε τα εξής βήματα:

- Κατέβασμα δεδομένων από το *Github*
- Καθαρισμός δεδομένων από αρχεία αντίγραφα
- Απομόνωση αλγορίθμου από τον υπόλοιπο κώδικα

3.1 Κατέβασμα δεδομένων από το *Github*

Για την δημιουργία του συνόλου των δεδομένων μας αρχικά θα χρειαστεί να αναζητήσουμε και να βρούμε πολλά αρχεία που περιλαμβάνουν κώδικα. Δεδομένου ότι οι κώδικες που υπάρχουν στην βιβλιογραφία είναι περιορισμένοι και απαιτείται αρκετός κόπος για να μετατραπούν σε ψηφιακή μορφή αποφασίσαμε να αναζητήσουμε αρχεία κώδικα σε ιστοσελίδες στο διαδίκτυο. Για τον σκοπό αυτό, επιλέξαμε την ιστοσελίδα *Github*. Η ιστοσελίδα αυτή λειτουργεί σαν ένα εργαλείο στο οποίο οι προγραμματιστές μπορούν όχι μόνο να μοιραστούν αρχεία κώδικα με άλλους προγραμματιστές αλλά και να επεξεργαστούν τον κώδικα αυτό χωρίς να δημιουργηθούν προβλήματα συγχρονισμού ανάμεσα στις διάφορες εκδόσεις που βλέπει μια ομάδα προγραμματιστών ταυτόχρονα. Παράλληλα, η συγκεκριμένη ιστοσελίδα παρουσιάζει περαιτέρω προτερήματα. Πιο συγκεκριμένα, είναι μία από τις πιο γνωστές ιστοσελίδες τέτοιου τύπου και διαθέτει εκατομμύρια αρχεία κώδικα, στα οποία μπορεί να έχει ελεύθερη πρόσβαση οποιοσδήποτε χρήστης της πλατφόρμας. Το γεγονός αυτό, δημιουργεί αρκετές προσδοκίες για το ζητούμενο μας καθώς όσο περισσότερα αρχεία υπάρχουν διαθέσιμα τόσο πιο πιθανό είναι να βρούμε και πολλά παραδείγματα των αλγορίθμων που αναζητούμε.

Επιπρόσθετα, υπάρχει ένα ακόμα πλεονέκτημα το οποίο θεωρούμε σκόπιμο να σημειώσουμε. Αυτό, δεν είναι άλλο από το γεγονός ότι το $Github$ διαθέτει μια ειδική διεπαφή μέσα από την οποία μπορεί κανείς να αναζητήσει αρχεία κώδικα χρησιμοποιώντας συγκεκριμένα κριτήρια. Για παράδειγμα, μπορεί κανείς να αναζητήσει αρχεία με βάση το όνομα τους, την γλώσσα προγραμματισμού στην οποία έχουν γραφτεί, λέξεις κλειδιά που περιέχονται μέσα στο αρχείο, το μέγεθος του αρχείου καθώς και την ημερομηνία κατά την οποία γράφτηκε το αρχείο αυτό.Φυσικά, η διεπαφή αυτή μας επέτρεψε να έχουμε μεγάλη ευχέρεια και άνεση στις αναζητήσεις τις οποίες πραγματοποιήσαμε. Παράλληλα, η διεπαφή αυτή μας επέτρεψε να αναζητήσουμε και να κατεβάσουμε αρχεία σε πολύ μεγάλη κλίμακα. Χαρακτηριστικά, αξίζει να αναφέρουμε ότι κατεβάσαμε πάνω από 500.000 αρχεία τα οποία επεξεργαστήκαμε με διάφορους τρόπους. Αδιαμφισβήτητα, κάτι τέτοιο θα ήταν ανέφικτο αν δεν ήταν διαθέσιμη η διεπαφή αυτή.

Αφού λοιπόν εξηγήσαμε τους λόγους για τους οποίους επιλέξαμε την ιστοσελίδα $Github$, στο σημείο αυτό θα δώσουμε περισσότερες λεπτομέρειες σχετικά με τις αναζητήσεις που θα πραγματοποιήσουμε. Αρχικά, όπως αναφέραμε και παραπάνω χρησιμοποιήσαμε την διεπαφή την οποία προσφέρει το $Github$ για προγραμματιστές. Μέσα από την διεπαφή αυτή μας δόθηκε η δυνατότητα να πραγματοποιήσουμε διάφορες αναζητήσεις με διαφορετικά σημεία αναζήτησης. Το γεγονός αυτό απαίτησε από την μεριά μας να πάρουμε σχεδιαστικές αποφάσεις σχετικά με τα σημεία αναζήτησης τα οποία θέλουμε να χρησιμοποιήσουμε. Όπως αναφέραμε και παραπάνω, η διεπαφή αυτή προσφέρει διάφορες δυνατότητες αναζήτησης. Βέβαια, ενώ κάτι τέτοιο δίνει μεγαλύτερη ελευθέρια στον χρήστη δημιουργεί και κάποια προβλήματα καθώς μια πιθανή εξαντλητική αναζήτηση μπορεί να χρειαστεί πολύ χρόνο για να πραγματοποιηθεί. Για παράδειγμα, αν αναζητήσουμε όλους τους κώδικες που είναι γραμμένοι στην γλώσσα προγραμματισμού C μπορεί να χρειαστούν μήνες για να κατεβάσουμε τα αρχεία αυτά από την ιστοσελίδα του $Github$. Αυτό συμβαίνει, όχι μόνο γιατί τα αρχεία έχουν συνολικά μεγάλο μέγεθος (της τάξεως των GB) αλλά και γιατί η διεπαφή αυτή ορίζει περιορισμούς στην ποσότητα των αναζητήσεων που πραγματοποιεί ένας χρήστης ανά λεπτό έτσι ώστε να αποφεύγετε πιθανή υπερφόρτωση του συστήματος και να είναι οι πόροι διαθέσιμοι για όλους τους χρήστες δικαιότερα. Παράλληλα, ακόμα και αν δεν υπήρχαν οι περιορισμοί αυτοί υπάρχει ίσως ένα σημαντικότερος περιοριστικός παράγοντας ο οποίος κάνει επιτακτική την ανάγκη της πραγματοποίησης έξυπνης αναζήτησης. Ο παράγοντας αυτός δεν είναι άλλος από την περιορισμένη υπολογιστική ισχύ που διαθέτουμε για να επεξεργαστούμε όλα τα αποτελέσματα που μπορούν να προκύψουν. Πιο συγκεκριμένα, αν κατεβάσουμε αρχεία με την χρήση εξαντλητικής αναζήτησης πιθανόν να χρειαστεί μήνες για την επεξεργασία των αρχείων αυτών. Φυσικά, το γεγονός αυτό είναι απαγορευτικό για την πραγματοποίηση του πειράματος μας. Στην συνέχεια, παραθέτουμε σε μορφή πίνακα τα σημαντικότερα κριτήρια αναζήτησης της διεπαφής αυτής.

Όνομα αρχείου
Μέγεθος αρχείου
Γλώσσα προγραμματισμού
Λέξη κλειδί μέσα στο αρχείο
Λέξη κλειδί στο μονοπάτι του αρχείου
Οργανισμός στον οποίο ανήκει το αρχείο
Όνομα χρήστη στον οποίο ανήκει το αρχείο

Πίνακας 3.1: Κριτήρια αναζήτησης στο API του $Github$

Εν συνεχεία, προσπαθήσαμε πειραματικά να αξιολογήσουμε την επίπτωση του κάθε κριτηρίου αναζήτησης, που προσφέρεται από την διεπαφή, στο πλήθος και την ποιότητα των αποτελεσμάτων της κάθε αναζήτησης. Παρατηρήσαμε, ότι το πιο σημαντικό κριτήριο αναζήτησης είναι το όνομα του αρχείου καθώς και κάποιες λέξεις κλειδιά που μπορεί να περιέχει το αρχείο αυτό. Ως εκ τούτου, αποφασίσαμε να χρησιμοποιήσουμε τα δύο αυτά κριτήρια, τόσο μεμονωμένα όσο και συνδυαστικά. Ως λέξεις κλειδιά των αναζητήσεων μας, επιλέξαμε κυρίως το όνομα του εκάστοτε αλγόριθμου καθώς και συνώνυμα και συντομογραφίες αυτού, καθώς παρατηρήσαμε ότι οι λέξεις αυτές είχαν τα πιο ποιοτικά αποτελέσματα. Παράλληλα, παρατηρήσαμε ότι τα αρχεία με μέγεθος μεγαλύτερο από 30 Mb πιθανόν να μας δημιουργήσουν προβλήματα κλιμάκωσης σε επόμενα βήματα. Έτσι, αποφασίσαμε οι αναζητήσεις μας να περιοριστούν στο διάστημα από 0 Mb έως 30 Mb.

Τέλος, πριν προχωρήσουμε παρακάτω θεωρούμε σκόπιμο να σχολιάσουμε έναν τεχνικό περιορισμό που συναντήσαμε χρησιμοποιώντας την διεπαφή αυτή, καθώς και την λύση την οποία εφαρμόσαμε για να αντιμετωπίσουμε το πρόβλημα αυτό. Παρόλο που η διεπαφή του $Github$ μας παρέχει όλες αυτές τις δυνατότητες, επιστρέφει για κάθε αναζήτηση τα καλύτερα 1000 αποτελέσματα (με την βοήθεια ενός κριτηρίου ομοιότητας) χωρίς να δίνει την δυνατότητα πρόσβασης στα υπόλοιπα αποτελέσματα. Για να γίνει περισσότερο κατανοητή η σημασία της λεπτομέρειας αυτής θα δώσουμε ένα παράδειγμα με πραγματικά νούμερα. Αν λοιπόν αναζητήσουμε την λέξη κλειδί $Gemm$ (αντιστοιχεί στον αλγόριθμο του πολλαπλασιασμού πίνακα επί πίνακα), για αρχεία που έχουν γραφτεί στην γλώσσα προγραμματισμού $C++$, το αποτέλεσμα της διεπαφής είναι 1000 αποτελέσματα ενώ συνολικά υπάρχουν 29073 αρχεία που αποτελούν πιθανό στόχο της αναζήτησης αυτής (σύμφωνα και πάλι με την διεπαφή). Είναι προφανές, ότι ο περιορισμός αυτός μπορεί να έχει πολύ αρνητικές επιπτώσεις στην προσπάθεια μας να δημιουργήσουμε ένα μεγάλο και ποιοτικό σύνολο δεδομένων. Ως εκ τούτου, δοκιμάσαμε διάφορους τρόπους για να λύσουμε το πρόβλημα αυτό και καταλήξαμε στο ότι η καλύτερη λύση είναι να κάνουμε πολλαπλές αναζητήσεις για διάφορα μεγέθη αρχείου. Πιο συγκεκριμένα, χωρίσαμε το διάστημα από 0 Mb έως 30 Mb σε υποδιαστήματα μεγέθους 500 Kb το καθένα και διασχίσαμε το καθένα από αυτά. Παράλληλα, σε περίπτωση που κάποιο από τα υποδιαστήματα αυτά είχε περισσότερα αποτελέσματα από 1000, χωρίσαμε και πάλι το διάστημα αυτό σε υποδιαστήματα των 50 Kb. Παρατηρήσαμε ότι περαιτέρω σπάσιμο σε υποδιαστήματα δεν ωφελούσε ιδιαίτερα και απλά αύξανε τον χρόνο εκτελέσεις των αναζητήσεων. Έτσι, σε περίπτωση που η συνθήκη των 1000 αποτελεσμάτων έσπασε σε διάστημα των 50 Kb δεν πραγματοποιήσαμε περαιτέρω σπάσιμο του διαστήματος. Στην συνέχεια, παραθέτουμε ένα μικρό κομμάτι κώδικα που περιγράφει τον τρόπο με τον οποίο αντιμετωπίσαμε τον περιορισμό που αναφέραμε.

Αλγόριθμος 1: Δυναμική αναζήτηση αρχείων στο API του $Github$

Δεδομένα Εισόδου: Κριτήρια αναζήτησης $query$

Έξοδος: Λίστα αποτελεσμάτων αναζήτησης

```
1  for i = 0; i < 30000; i+=500 do
2      results = postGithubApi(query);
3      if results < 1000 then
4          subquery = CreateSubquery(query, i);
5          FetchInfo(subquery);
6      else
7          for j = 0; j < 1000 j+= 50 do
8              subquery = CreateSubquery(query, i + j);
9              FetchInfo(subquery);
10         end
11 end
```

Αφού λοιπόν εξηγήσαμε τις επιλογές μας σχετικά με τα κριτήρια αναζήτησης του API του $Github$ στο σημείο αυτό καλούμαστε να σχολιάσουμε τις επιλογές μας σχετικά με το ποία από τα αποτελέσματα της αναζήτησης θα αποθηκεύσουμε. Πιο συγκεκριμένα, μετά από κάθε αναζήτηση επιστρέφεται μία λίστα από αποτελέσματα στην οποία υπάρχουν διάφορες πληροφορίες σχετικά με το αρχείο κώδικα στο οποίο αντιστοιχεί το κάθε αποτέλεσμα. Δυστυχώς, δεδομένου ότι οι πληροφορίες αυτές είναι πολλές καλούμαστε να αποθηκεύσουμε μόνο πληροφορίες οι οποίες μπορεί να μας χρειαστούν στην συνέχεια, καθώς αν τις αποθηκεύσουμε όλες θα δημιουργήσουμε πολύ μεγάλα αρχεία τα οποία πιθανώς να μας δημιουργήσουν προβλήματα στην μετέπειτα επεξεργασία τους. Στην συνέχεια, παραθέτουμε σε μορφή πίνακα τα σημαντικότερα σημεία πληροφορίας που περιέχονται σε κάθε αποτέλεσμα μίας αναζήτησης.

Όνομα αρχείου
Μέγεθος αρχείου
Γλώσσα προγραμματισμού
Μονοπάτι αρχείου στο $repository$
Μοναδικό αναγνωριστικό αρχείου(sha)
Σύνδεσμος στον οποίο βρίσκεται το αρχείο
Όνομα χρήστη στον οποίο ανήκει το αρχείο
Βαθμός ταιριάσματος με βάση το κριτήριο αναζήτησης
Λοιπές πληροφορίες σχετικά με το $repository$ του αρχείου
Λοιπές πληροφορίες σχετικά με αρχείο

Πίνακας 3.2: Σημεία πληροφορίας αποτελέσματος αναζήτησης στο API του $Github$

Αφού λοιπόν παραθέσαμε εν συντομία κάποια από τα στοιχεία που επιστρέφει η διεπαφή για κάθε αποτέλεσμα μίας αναζήτησης θα σχολιάσουμε ποία από τα σημεία αυτά θεωρήσαμε ότι είναι χρήσιμα για να αποθηκεύσουμε. Αρχικά, χρησιμοποιήσαμε τον σύνδεσμο του αρχείου για να τα κατεβάσουμε το ίδιο το αρχείο (η μεθοδολογία θα εξηγηθεί παρακάτω). Στην συνέχεια, αποθηκεύσαμε στοιχεία όπως το μέγεθος του αρχείου και την γλώσσα προγραμματισμού στην οποία αυτό έχει γραφτεί. Ένα άλλο πολύ σημαντικό στοιχείο το οποίο αποθηκεύσαμε είναι το μοναδικό αναγνωριστικό του αρχείου (sha). Το μοναδικό αναγνωριστικό αυτό θα το χρησιμοποιήσουμε έτσι ώστε να αποφύγουμε το κατέβασμα αρχείων που είναι γνωστά αντίγραφα. Τέλος, αποφασίσαμε να κρατήσουμε και πληροφορίες σχετικά με το $repository$ του κάθε αρχείου καθώς είναι πιθανό αργότερα στην διαδικασία να χρειαστεί να κατεβάσουμε ολόκληρο το $repository$. Πιο συγκεκριμένα, μία τέτοια πληροφορία θα μπορούσε να φανεί χρήσιμη στην περίπτωση όπου το αρχείο δεν μεταγλωττίζεται καθώς λείπουν συναρτήσεις που έχουν οριστεί σε εξωτερικά βοηθητικά αρχεία εντός του $repository$.

Στο σημείο αυτό, θα εξηγήσουμε το τελευταίο και πιο σημαντικό κομμάτι αυτής της ενότητας. Πιο συγκεκριμένα, θα εξηγήσουμε τον τρόπο με τον οποίο όλα τα παραπάνω συνδέονται μεταξύ τους σε μία διαδικασία η οποία μας επιτρέπει να συλλέξουμε τα αρχεία κώδικα από το $Github$. Αρχικά, ορίζουμε μία σειρά από ονόματα και λέξεις κλειδιά για τα οποία επιθυμούμε να πραγματοποιηθεί η αναζήτηση. Στην συνέχεια, δίνουμε την σειρά αυτή σε έναν αλγόριθμο ο οποίος για κάθε πιθανό συνδυασμό αναζήτησης που υπάρχει στην σειρά κάνει αιτήματα αναζήτησης στην διεπαφή του $Github$ (λαμβάνοντας υπόψιν δυναμικά το μέγεθος του αρχείου όπως εξηγήθηκε παραπάνω). Για κάθε αποτέλεσμα που λαμβάνει, κρατάει τις πληροφορίες που σχολιάσαμε παραπάνω ενώ ταυτόχρονα κατεβάζει το αρχείο κώδικα με βάση το δοσμένο σύνδεσμο. Στο σημείο αυτό, αξίζει να σημειώσουμε ότι πριν το κατέβασμα και την αποθηκεύσει ενός αρχείου ελέγχουμε αν το αρχείο αυτό υπάρχει ήδη στο σύνολο των δεδομένων με την χρήση του μοναδικού αναγνωριστικού (sha). Μάλιστα, κατά την διαδικασία παρατηρήσαμε ότι σε πολλές αναζητήσεις τα αντίγραφα αυτά είχαν ακόμα και ποσοστό κοντά στο 80%, γεγονός που μας οδήγησε στο συμπέρασμα ότι κρίνεται επιτακτική η ανάγκη αποκοπής αντιγράφων με την χρήση του μοναδικού αναγνωριστικού. Παράλληλα, έχει υλοποιηθεί ένα ειδικός μηχανισμός ο οποίος διαχειρίζεται το σενάριο στο οποίο ο αλγόριθμος μας ξεπερνάει τα επιτρεπόμενα όρια χρησιμοποίησης της συγκεκριμένης διεπαφής (αφού όπως αναφέραμε προηγουμένως υπάρχουν περιορισμοί τόσο ανά ώρα όσο και ανά λεπτό). Πιο συγκεκριμένα, στην περίπτωση που η διεπαφή επιστρέψει μήνυμα υπέρβασης χρήσης ο αλγόριθμος μας κάνει παύση για 1 λεπτό και προσπαθεί ξανά έως ότου το αποτέλεσμα της αναζήτησης είναι επιτυχές. Τέλος, αφού η διαδικασία αυτή ολοκληρωθεί για όλους τους συνδυασμούς αναζήτησης τα συνολικά αποτελέσματα αποθηκεύονται σε ένα αρχείο της μορφής $json$. Στην συνέχεια, παραθέτουμε ένα μικρό κομμάτι κώδικα που περιγράφει την παραπάνω διαδικασία όπως αυτή υλοποιήθηκε.

Αλγόριθμος 2: Κατέβασμα αρχείων με βάση κριτήρια από το API του $Github$

Δεδομένα Εισόδου: Ονόματα αναζήτησης ($names$), Λέξεις κλειδιά ($keywords$), Γλώσσα Προγραμματισμού ($language$), Όνομα αρχείου αποθήκευσης ($filename$)

Έξοδος: Αποτελέσματα αναζήτησης αποθηκευμένα σε αρχείο της μορφής $json$

```
1  queries = CreateAllSearchCombinations(names, keywords, language);
2  memory = dict();
3  for (query in queries) do
4      information, response = DynamicSearch(query);
5      while (repsonse == "abuse") do
6          sleep(60);
7          information, response = DynamicSearch(query);
8      end
9      if (memory[information["sha"]] == None) then
10         code = FetchCode(information["codeurl"]);
11         results.append(information, code);
12         memory[information["sha"]] = 1;
13 end
14 SaveToJson(results, filename);
```

3.2 Καθαρισμός δεδομένων από αρχεία αντίγραφα

Αφού λοιπόν κατεβάσαμε ένα σύνολο αρχείων από το *Github*, στην συνέχεια επιθυμούμε να επεξεργαστούμε κατάλληλα τα αρχεία αυτά έτσι ώστε να μπορέσουμε να τα χρησιμοποιήσουμε στα μοντέλα που θα κατασκευάσουμε. Πριν όμως συμβεί αυτό, θεωρούμε πολύ σημαντικό να συζητήσουμε την επίδραση του φαινομένου των αρχείων αντιγράφων (ή των αρχείων που είναι σχεδόν αντίγραφα). Πιο συγκεκριμένα, είναι πιθανό στο σύνολο δεδομένων που κατεβάσαμε από το *Github* να υπάρχουν αρχεία τα οποία είτε είναι ακριβώς ίδια είτε σχεδόν ίδια με κάποιο άλλο αρχείο[36],[40]. Αυτό για παράδειγμα θα μπορούσε να συμβεί αν ένας χρήστης αντιγράψει τον κώδικα ενός άλλου χρήστη ή αν κατεβάσουμε αρχεία κώδικα τα οποία ανήκουν στο ίδιο εγχείρημα αλλά σε διαφορετικές εκδόσεις αυτού. Στην συνέχεια, για την καλύτερη κατανόηση του αναγνώστη παραθέτουμε ένα παράδειγμα δύο τέτοιων αρχείων όπως εντοπίστηκε στα δεδομένα που κατεβάσαμε από το *Github*.

```c
#include <stdio.h>

int main()
{
  int array[100], n, c, d, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      /* For decreasing order use < */
      if (array[d] > array[d+1])
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }

  printf("Sorted list in ascending order:\n");

  for ( c = 0 ; c < n ; c++ )
     printf("%d\n", array[c]);

  return 0;
}
```

Σχήμα 3.1: Αντίγραφο 1 για τον αλγόριθμο *bubblesort*

```c
#include <stdio.h>
int main()
{
  //make sure to allocate enough space
  int array[100], n, c, d, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);
  //take in values from the command line
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0 ; c < ( n - 1 ); c++)
  {
    //inner loop for doing the swaps

    for (d = 0 ; d < n - c - 1; d++)
    {
      /* For decreasing order use < */
      if (array[d] > array[d+1])
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }

  printf("Sorted list in ascending order:\n");
  //printing the results
  for ( c = 0 ; c < n ; c++ )
     printf("%d\n", array[c]);

  return 0;
}
```

Σχήμα 3.2: Αντίγραφο 2 για τον αλγόριθμο *bubblesort*

Παρατηρώντας τα δύο παραπάνω σχήματα βλέπουμε ότι το πρώτο παράδειγμα (Σχήμα 3.1) διαφέρει από το δεύτερο (Σχήμα 3.2) ως προς τον σχολιασμό καθώς όπως εύκολα παρατηρεί κανείς, το παράδειγμα του Σχήματος 3.2 παρουσιάζει εκτενή σχολιασμό σε αντίθεση με το παράδειγμα του Σχήματος 3.1. Φυσικά, ο σχολιασμός δεν είναι ο μόνος λόγος για τον οποίο παρατηρήσαμε φαινόμενα αντιγράφων στο σετ αρχείων που δημιουργήσαμε. Πιο συγκεκριμένα, παρατηρήσαμε ότι υπήρχαν αρχεία τα οποία είτε δεν διέφεραν καθόλου μεταξύ τους, είτε παρουσίαζαν μικρές διαφορές μεταξύ τους ως προς τα σχόλια, τα κενά και τον τρόπο γραφής, τα ονόματα των μεταβλητών, τις τιμές των σταθερών καθώς και μικρές διαφορές υλοποίησης οι οποίες όπως δεν επέφεραν ουσιαστική αλλαγή στην λειτουργία του τελικού κώδικα. Τέτοια αρχεία, θα μπορούσαν να δημιουργήσουν σημαντικά προβλήματα στην συνέχεια, τόσο κατά την διαδικασία εκμάθησης όσο και κατά την διαδικασία αξιολόγησης του μοντέλου μας. Πιο συγκεκριμένα, είναι σημαντικό να αποφανθούμε σχετικά με το εάν κάθε αρχείο ανήκει στην αληθινή κατανομή των δεδομένων[36]. Κάτι τέτοιο είναι απαραίτητο καθώς τα αρχεία αντίγραφα θα υπάρχουν τόσο στο σύνολο εκπαίδευσης του μοντέλου μας, όσο και στο σύνολο αξιολόγησης. Ως εκ τούτου, είναι προφανές ότι τα αρχεία αυτά θα επηρεάσουν αρκετά τα πειράματα μας. Για παράδειγμα, αν ένα αρχείο υπάρχει τόσο στο σύνολο εκπαίδευσης όσο και στο σύνολο αξιολόγησης τότε το μοντέλο μας θα προβλέψει τον αλγόριθμο του αρχείου αυτού πολύ εύκολα . Αν αυτό συμβεί σε μεγάλη κλίμακα είναι πιθανό να παρατηρήσουμε ότι διάφορες μετρικές, που μπορεί να χρησιμοποιούμε κατά την αξιολόγηση, παρουσιάζουν υπερβολικά υψηλές τιμές . Το γεγονός αυτό συνεπάγεται την δημιουργία λανθασμένων εντυπώσεων σχετικά με την ποιότητα των προβλέψεων του μοντέλου μας.

Όπως αναφέραμε και παραπάνω είναι σημαντικό να αποφασίσουμε αν δύο αρχεία τα οποία είναι είτε αντίγραφα είτε σχεδόν αντίγραφα ανήκουν στην αληθινή κατανομή των δεδομένων[36]. Για το πρόβλημα μας, θεωρήσαμε ότι δύο τέτοια αρχεία δεν ανήκουν στην κατανομή αυτή καθώς εμείς αναζητούμε πολλές αναπαραστάσεις του ίδιου κώδικα, οι οποίες όμως δεν έχουν γραφτεί από τον ίδιο χρήστη. Στην περίπτωση των αρχείων αυτών, παρατηρήσαμε ότι τα περισσότερα αρχεία προέρχονταν από ένα πολύ δημοφιλές αρχείο. Ως εκ τούτου, θεωρήσαμε επιτακτική την ανάγκη να αποκόψουμε τα αρχεία αυτά από το σύνολο των δεδομένων μας. Για τον σκοπό αυτό, ερευνήσαμε διάφορους τρόπους με τους οποίους θα μπορούσαμε να εντοπίσουμε τα αρχεία αυτά και να τα αποκόψουμε από τα υπόλοιπα. Αρχικά, δοκιμάσαμε απλές τεχνικές όμως το να διαγράψουμε τα σχόλια και και τα τα κενά ($whitespaces$) σε κάθε αρχείο και στην συνέχεια να τα συγκρίνουμε μεταξύ τους. Οι τεχνικές αυτές, αν και έβρισκαν αρκετά αρχεία αντίγραφα παρατηρήσαμε ότι δεν ήταν όσο αποδοτικές όσο θα θέλαμε καθώς έχαναν τις περιπτώσεις στις οποίες ένας χρήστης είχε απλά αλλάξει τα ονόματα των μεταβλητών σε ένα αρχείο (φαινόμενο το οποίο παρατηρείται πολύ συχνά κατά την αντιγραφή ενός αρχείου κώδικα). Έτσι, θεωρήσαμε απαραίτητο να χρησιμοποιήσουμε μια πιο έξυπνη τεχνική για τον καθαρισμό του συνόλου των δεδομένων από αρχεία αντίγραφα[36].

Για τον σκοπό αυτό, δημιουργήσαμε για κάθε αρχείο μία λίστα από όλες τις λέξεις που περιέχει, χωρίς να λάβουμε υπόψιν τα σχόλια και τις λέξεις κλειδιά της γλώσσας προγραμματισμού. Στην συνέχεια, με την χρήση των λέξεων της λίστας αυτής δημιουργήσαμε ένα σύνολο μοναδικών λέξεων (T_0) καθώς και ένα σύνολο λέξεων το οποίο όμως λαμβάνει υπόψιν και τον αριθμό εμφανίσεων της κάθε λέξης στο αρχείο (T_1). Στο σημείο αυτό, θα παραθέσουμε ένα παράδειγμα των συνόλων αυτών για τον κώδικα του Σχήματος 3.2.

main, array, 100, n, c, d, swap, Enter, number, of, elements, in, ascending, order, integers, 0, 1, Sorted, list

Πίνακας 3.3: Σύνολο μοναδικών λέξεων (T_0) του Σχήματος 3.2

(main,1), (array,9), (100,1) , (n,7), (c,13), (d,10), (swap,2), (Enter,2), (number,1), (of,1), (elements,1), (in,1), (ascending,1), (order,1), (integers,1), (0,5), (1,5), (Sorted,1), (list,1)

Πίνακας 3.4: Σύνολο λέξεων με πλήθος εμφανίσεων (T_1) του Σχήματος 3.2

Στην συνέχεια, για κάθε σύνολο (T_0 και T_1) κάθε αρχείου υπολογίσαμε την ομοιότητα $Jaccard$ με τα σύνολα όλων των υπόλοιπων αρχείων. Αν η ομοιότητα αυτή είναι μεγαλύτερη από μία ελάχιστη τιμή τότε θεωρούμε ότι τα αρχεία αυτά είναι αντίγραφα και αποκόπτουμε το ένα από τα δύο. Στην συνέχεια, παραθέτουμε την μαθηματική έκφραση της ομοιότητας $Jaccard$ για τα σύνολα (T_0 και T_1) των αρχείων f1 και f2.

$$Jaccard = \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|}$$

Σχήμα 3.3: Εξίσωση ομοιότητας $Jaccard$ για τα σύνολα λέξεων δύο αρχείων f_1 και f_2

Αφού λοιπόν πραγματοποιήσουμε την παραπάνω διαδικασία για όλα τα αρχεία κάθε αλγορίθμου διαγράψαμε τα αρχεία αντίγραφα και δημιουργήσαμε ένα σύνολο αρχείων καθαρό από αρχεία αντίγραφα. Το γεγονός αυτό, μας εξασφαλίζει ότι όλα τα αρχεία κώδικα στο σύνολο των δεδομένων μας ανήκουν στην αληθινή κατανομή των δεδομένων. Στην συνέχεια, παραθέτουμε ένα μικρό κομμάτι κώδικα που περιγράφει την διαδικασία αυτή.

Αλγόριθμος 3: Διαγραφή αντιγράφων από το σύνολο δεδομένων

Δεδομένα Εισόδου: Σύνολο δεδομένων ($algorithms$)
Έξοδος: Σύνολο δεδομένων χωρίς αρχεία αντίγραφα ($results$)

```
for (algorithm in algorithms) do
    for code in algorithm do
        set_1, set_2 = CreateTokenSets(code);
        memory[code] = (set_1, set_2);
    end
    index_outer = 0;
    visited = dict();
    for (code_1 in algorithm) do
        index_inner = 0;
        for (code_2 in algorithm) do
            if (code_1 == code_2) then
                continue;
            if (visited[str(index_outer) + str(index_inner)]) == 1; then
                continue;
            similarity_1 = JaccardSimilarity(memory[code_1][0], memory[code_2][0]);
            similarity_2 = JaccardSimilarity(memory[code_1][1], memory[code_2][1]);
            if (similarity_1 > 0.7 and similary_2 > 0.8) then
                duplicates.append(index);
                visited[str(index_inner) + str(index_outer)] = 1;
            index_inner += 1;
        end
        index_outer += 1;
    end
    deduplicated = RemoveDuplicates(algorithm, duplicates);
    results.append(deduplicated);
end
```

Όπως μπορεί εύκολα να παρατηρήσει κανείς, ο Αλγόριθμος 3 έχει πολυπλοκότητα $O(n^2)$ για κάθε αλγόριθμο του συνόλου δεδομένων. Το γεγονός αυτό σε συνδυασμό με το πολύ μεγάλο πλήθος αρχείων που έχουμε για τον κάθε αλγόριθμο, καθιστούν την παραπάνω διαδικασία όχι και τόσο πρακτική ακόμα και αν την εκτελέσουμε παράλληλα σε περισσοτέρους από έναν πυρήνες. Ως εκ τούτου, αποφασίσαμε να χρησιμοποιήσαμε ένα προσεγγιστικό εργαλείο[36] το οποίο εκτελεί την παραπάνω διαδικασία κάνοντας κάποιες "έξυπνες" απλοποιήσεις. Για παράδειγμα, το εργαλείο αυτό υποθέτει ότι η ομοιότητα μεταξύ αρχείων μεταφέρεται, δηλαδή αν ένα αρχείο f_1 είναι όμοιο με ένα αρχείο f_2 και το αρχείο f_2 είναι όμοιο με ένα αρχείο f_3 τότε και τα 3 αρχεία είναι όμοια ασχέτως αν το f_1 είναι όμοιο με το f_3 . Η συγκεκριμένη υπόθεση προφανώς δεν είναι απόλυτα σωστή αλλά έχει αποδειχθεί ότι δεν επηρεάζει την ποιότητα των προβλέψεων του συστήματος ομοιότητας. Με την χρήση του εργαλείου αυτού καταφέραμε να υλοποιήσουμε την διαδικασία αυτή σε μόλις το $\frac{1}{10}$ του χρόνου που θα χρειαζόμασταν αν είχαμε τρέξει την μη προσεγγιστική υλοποίηση.

Τέλος, στο σημείο αυτό θεωρούμε σκόπιμο να σχολιάσουμε τα πειραματικά αποτελέσματα που παρατηρήσαμε μετά την εκτέλεση της διαδικασίας αυτής για όλους τους αλγορίθμους του σύνολου δεδομένων που δημιουργήσαμε. Αφού λοιπόν τρέξαμε το παραπάνω κριτήριο ομοιότητας για όλα τα αρχεία παρατηρήσαμε ότι έπρεπε να φιλτράρουμε περίπου το 35% των αρχείων που έχουμε. Πιο συγκεκριμένα , παρατηρήσαμε ότι υπάρχει ένα 10% στο οποίο κάθε αρχείο είναι ακριβώς ίδιο με κάποιο άλλο και στο μόνο που διαφέρουν είναι η μορφοποίηση ή τα σχόλια. Το υπόλοιπο 25% παρουσίαζε τόσο αλλαγές στα σχόλια και τη μορφοποίηση όσο και μικρές αλλαγές σε διάφορα σημεία του κώδικα, όπως για παράδειγμα τα ονόματα των μεταβλητών ή την χρήση διαφορετικών εντολών που πραγματοποιούν την ίδια λειτουργία. Επιπρόσθετα, παρατηρήσαμε ότι στις περισσότερες περιπτώσεις τα αρχεία αντίγραφα ήταν είτε διαφορετικές εκδόσεις του ίδιου κώδικα που κάποιος είχε κάνει $fork$ κάποια στιγμή, είτε εκδόσεις κώδικα τις οποίες χρήστες είχαν κάνει αντιγραφή-επικόλληση με το χέρι και είχαν πραγματοποιήσει μικρές ή και καθόλου αλλαγές. Από τα παραπάνω αποτελέσματα, εύκολα καταλαβαίνει κανείς πως η διαδικασία αυτή ήταν επιτακτική έτσι ώστε να διασφαλιστεί η ποιότητα του πειράματος μας, καθώς σε αντίθετη περίπτωση το $\frac{1}{3}$ των δεδομένων μας θα ήταν αρχεία αντίγραφα οδηγώντας στην δραματική μείωση της ποιότητας του συνόλου των δεδομένων μας.

3.3 Απομόνωση αλγορίθμου από τον υπόλοιπο κώδικα

Αφού λοιπόν δημιουργήσαμε ένα σύνολο αρχείων κώδικα το οποίο δεν περιλαμβάνει αντίγραφα, στο σημείο αυτό θα ασχοληθούμε με τον τρόπο με τον οποίο θα απομονώσουμε από κάθε αρχείο μόνο το κομμάτι κώδικα που μας ενδιαφέρει. Πιο συγκεκριμένα, αυτή την στιγμή γνωρίζουμε ότι στα αρχεία κώδικα που διαθέτουμε υπάρχει πιθανώς ο αλγόριθμος που αναζητούμε. Όμως στα αρχεία αυτά, μπορεί να υπάρχουν και αρκετές γραμμές κώδικα οι οποίες μπορεί να μην σχετίζονται με τον αλγόριθμο που αναζητούμε. Ως εκ τούτου, κρίνεται επιτακτική η ανάγκη της απομόνωσης του αλγορίθμου στόχου από το υπόλοιπο αρχείο. Ο ασφαλέστερος τρόπος για να πραγματοποιηθεί αυτό είναι να ελεγχθεί κάθε αρχείο ξεχωριστά από έμπειρους προγραμματιστές. Φυσικά, κάτι τέτοιο είναι πρακτικά αδύνατο καθώς τα αρχεία που θέλουμε να επεξεργαστούμε είναι εκατοντάδες χιλιάδες. Έτσι, δημιουργήσαμε μία ειδική αυτοματοποιημένη διαδικασία η οποία απομονώνει τον αλγόριθμο από το αρχικό αρχείο στο οποίο περιέχεται. Στην συνέχεια, θα συζητήσουμε ξεχωριστά την δημιουργία :

- Ενός συνόλου δεδομένων με επαναληπτικούς βρόγχους (*loops*)
- Ενός συνόλου δεδομένων με συναρτήσεις

3.3.1 Δημιουργία συνόλου δεδομένων με επαναληπτικούς βρόγχους

Στην ενότητα αυτή θα ασχοληθούμε αναλυτικά με τον τρόπο απομόνωσης ενός επαναληπτικού βρόγχου στόχου από ένα αρχείο κώδικα. Πιο συγκεκριμένα, για ένα γνωστό αλγόριθμο ο οποίος γνωρίζουμε ότι συνήθως υλοποιείται με χρήση επαναληπτικών βρόγχων (*loops*) επιθυμούμε να απομονώσουμε από ένα αρχείο κώδικα (το οποίο πιθανολογούμε ότι περιέχει τον αλγόριθμο αυτό) μόνο το κομμάτι με τους επαναληπτικούς βρόγχους που περιέχει τον αλγόριθμο αυτό. Φυσικά, η συγκεκριμένη διαδικασία είναι πολύ περίπλοκη καθώς στην ουσία επιθυμούμε από ένα αρχείο κώδικα το οποίο μπορεί να περιέχει και 1000 γραμμές κώδικα να απομονώσουμε μόνο λίγες από αυτές, οι οποίες μάλιστα δεν αρκεί να είναι απλά επαναληπτικοί βρόγχοι αλλά πρέπει και να αποτελούν υλοποίηση του αλγορίθμου στόχου που αναζητούμε. Για τον σκοπό αυτό θα χρησιμοποιήσουμε τόσο τα αρχεία κώδικα που κατεβάσαμε από το *Github* όσο και θα δημιουργήσουμε συνθετικά κομμάτια κώδικα για τις περιπτώσεις στις οποίες δεν μπορούμε να συλλέξουμε δεδομένα.

3.3.1.1 Δημιουργία συνθετικών δεδομένων

Αρχικά, θα προσπαθήσουμε να δημιουργήσουμε συνθετικά δεδομένα για τις κατηγορίες εκείνες οι οποίες είναι δύσκολο να απομονώσουμε το κομμάτι του κώδικα από το υπόλοιπο αρχείο. Στην περίπτωση των επαναληπτικών βρόγχων παρατηρήσαμε ότι η μεγαλύτερη δυσκολία εντοπίζεται στους αλγορίθμους για τους οποίους αρκεί μόλις ένας βρόγχος για την υλοποίηση τους. Αυτό συμβαίνει κυρίως γιατί, υπάρχουν πολλές άλλες περιπτώσεις που αναπαρίστανται με ένα βρόγχο (όπως η δημιουργία μονοδιάστατου πίνακα και η εκτύπωση αυτού). Ως εκ τούτου, αποφασίσαμε να δημιουργήσουμε κυρίως συνθετικά δεδομένα για τις περιπτώσεις αυτές.

Αφού λοιπόν, εντοπίσαμε όλες τις "προβληματικές" περιπτώσεις ανάμεσα στους αλγορίθμους στόχους, πραγματοποιήσαμε εκτενή ανάλυση για τους πιθανούς τρόπους με τους οποίες μπορεί να γραφτεί ο καθένας από τους αλγορίθμους αυτούς. Βέβαια, δεδομένου ότι όπως αναφέραμε και παραπάνω οι αλγόριθμοι τις περιπτώσεις αυτοί είναι αρκετά απλοί (υλοποιούνται από ένα μόνο βρόγχο) δεν υπήρχαν πολλές "ουσιαστικές" εναλλακτικές με τις οποίες μπορεί να γραφτεί καθένας από τους αλγορίθμους αυτούς. Στην συνέχεια, για κάθε αλγόριθμο που ανήκει στην ομάδα αυτή δημιουργήσαμε ένα αρχείο στην γλώσσα προγραμματισμού $C++$ το οποίο περιέχει μόνο επαναληπτικούς βρόγχους με τους διαφορετικούς τρόπους υλοποίησης του εκάστοτε αλγορίθμου. Έπειτα, για κάθε αρχείο που δημιουργήσαμε χρησιμοποιήσαμε τον μεταγλωττιστή $Clang$ ο οποίος μας επέστρεψε το αφηρημένο συντακτικό δέντρο (AST) του κάθε αρχείου. Στην συνέχεια, διασχίσαμε το δέντρο αυτό και όποτε συναντήσαμε έναν κόμβο ο οποίος αποτελεί βρόγχο απομονώσαμε τον κώδικα που περικλείει αφού γνωρίζουμε οτι αλγόριθμοι στόχοι υλοποιούνται από ένα μόνο επαναληπτικό βρόγχο. Στην συνέχεια, παραθέτουμε ένα μικρό κομμάτι κώδικα που περιγράφει την διαδικασία αυτή.

Αλγόριθμος 4: Απομόνωση συνθετικών δεδομένων από αρχεία κώδικα

Δεδομένα Εισόδου: Σύνολα αρχείων με συνθετικά δεδομένα ($files$), Όνομα αρχείου αποθήκευσης ($dataset$)

Έξοδος: Αποτελέσματα αναζήτησης αποθηκευμένα σε αρχείο της μορφής pkl

```
1  memory = list();
2  for (file in files) do
3      AST = ParseCode(file);
4      for (node in AST) do
5          if (node == "forloop" or node == "whileloop") then
6              code = FetchCodeContainedInNode(node);
7              results.append((code, file));
8      end
9  end
10 SaveToPkl(results, dataset);
```

3.3.1.2 Απομόνωση βρόγχων από αρχεία κώδικα

Στο σημείο αυτό θα ασχοληθούμε με την δημιουργία μιας αυτοματοποιημένης διαδικασίας για την εξαγωγή επαναληπτικών βρόγχων. Ο λόγος για τον οποίο θεωρήσαμε σκόπιμο να δημιουργήσουμε την μέθοδο αυτή είναι γιατί η διαδικασία της δημιουργίας συνθετικών δεδομένων είναι πολύ επίπονη και απαιτεί πολύ χρόνο. Ως εκ τούτου, είναι προφανές ότι η η διαδικασία αυτή δεν μπορεί να γενικευτεί για την δημιουργία συνόλων δεδομένων με περισσότερα αρχεία και απαιτείται μία αυτοματοποίηση της διαδικασίας αυτής. Για την απομόνωση των βρόγχων αυτών αρχικά πρέπει να κάνουμε μία υπόθεση σχετικά με το σημείου του κώδικα στο οποίο πιθανόν να εντοπίζεται ο βρόγχος στόχος. Για τον σκοπό αυτό, θεωρήσαμε "έξυπνο" να αναζητήσουμε τις συναρτήσεις που εμπεριέχονται στο αρχείο κώδικα τον οποίο το όνομα εμπεριέχει μέσα κάποια λέξη κλειδί η οποία σχετίζεται με τον αλγόριθμο στόχο και ταυτόχρονα δεν εμπεριέχει κάποιες λέξεις οι οποίες θεωρούνται ως λέξεις προς αποφυγή. Στην περίπτωση που βρούμε μία τέτοια συνάρτηση θεωρούμε ότι ο βρόγχος στόχος θα μπορούσε να εμπεριέχεται μέσα στην συνάρτηση αυτή. Στην συνέχεια, παραθέτουμε ένα μικρό κομμάτι κώδικα που περιγράφει την διαδικασία αυτή.

Αλγόριθμος 5: Εντοπισμός συνάρτησης στόχου με βάση λέξεις κλειδιά

Δεδομένα Εισόδου: Αφηρημένο συντακτικό δένδρο κώδικα (AST), Λέξεις κλειδιά προς αναζήτηση ($keywords$), Λέξεις κλειδιά προς αποφυγή ($stopwords$)

Έξοδος: Συναρτήσεις στων οποίων το όνομα εντοπίζονται οι λέξεις κλειδιά $functions$

```
1  functions = list();
2  for (node in AST) do
3      if (node == "funtion") then
4          FunctionName = FetchFunctioName(node);
5          for (keyword in keywords) do
6              if (keyword in FunctionName) then
7                  flag = 0;
8                  for (stopword in stopwords) do
9                      if (stopword in FunctionName) then
10                         flag = 1;
11                         break;
12                 end
13                 if (flag == 0) then
14                     functions.append(node);
15                     break;
16         end
17 end
```

Αφού λοιπόν εντοπίσαμε τα πιθανά σημεία στα οποία μπορεί να βρίσκεται ο βρόγχος στόχος, απομένει να εντοπίσουμε το ακριβές σημείο στο οποίο βρίσκεται ο βρόγχος αυτός. Βέβαια, η διαδικασία αυτή αποτελεί πρόκληση καθώς ναι μεν περιορίσαμε τα κομμάτια κώδικα στα οποία θα επιστήσουμε την προσοχή μας αλλά υπάρχει ακόμα η πιθανότητα να υπάρχουν περισσότεροι από έναν βρόγχοι στα κομμάτια αυτά. Για τον σκοπό αυτό, επιθυμούμε να αναπτύξουμε έναν "έξυπνο" τρόπο έτσι ώστε να επιλέξουμε τον βρόγχο στόχο. Ένα ασφαλές και ταυτόχρονα αποδοτικό κριτήριο αναζήτησης είναι το βάθος του εκάστοτε βρόγχου. Ως βάθος ενός βρόγχου ορίζουμε τον μέγιστο αριθμό εμφωλευμένων βρόγχων σε ένα κομμάτι κώδικα. Για τον υπολογισμό του βάθος ενός κόμβου στόχου, εκτελούμε μία απλή διάσχιση κατά βάθος (DFS). Συνοψίζοντας, για την απομόνωση του στόχου βρόγχου θεωρούμε ένα ελάχιστο βάθος που απαιτείται για την υλοποίηση ενός αλγορίθμου (η συγκεκριμένη γνώση βασίζεται στην βιβλιογραφία) και κρατάμε όλους τους βρόγχους που ικανοποιούν το κριτήριο αυτό. Παρόλο που, το κριτήριο αυτό μοιάζει απλό, ο συνδυασμός του με την απομόνωση μίας συνάρτησης στόχου με βάσει το όνομα οδηγεί σε πολύ ενθαρρυντικά αποτελέσματα. Στην συνέχεια, παραθέτουμε ένα μικρό κομμάτι κώδικα που περιγράφει την διαδικασία αυτή.

Αλγόριθμος 6: Εντοπισμός βρόγχου στόχου με βάση το βάθος

Δεδομένα Εισόδου: Συναρτήσεις που απομονώθηκαν από τον Αλγόριθμο 5 ($nodes$), Ελάχιστο επιθυμητό βάθος ($desiredDepth$)

Έξοδος: Βρόγχοι στόχοι $results$

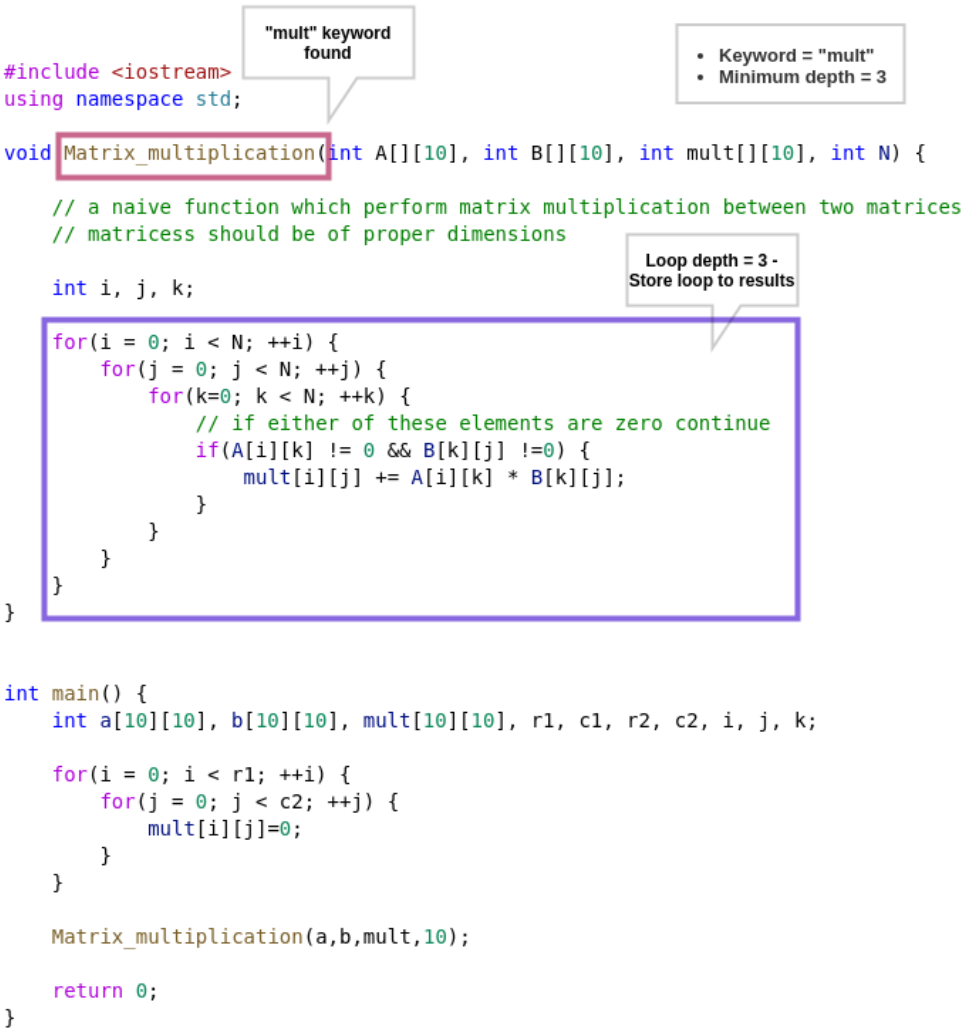
```
1  Function checkDepth(node, depth):
2      if (node == "forloop" or node == "whileloop") then
3          depth+ = 1;
4          result = depth;
5      for (child in node.getChildren()) do
6          result = max(checkDepth(node, depth), result);
7      end
8      return result;
9  End Function
10 Function findLoops(nodes, desiredDepth):
11     results = list();
12     for (node in nodes) do
13         if (node == "forloop" or node == "whileloop") then
14             if (checkDepth(node, 0) >= desiredDepth) then
15                 results.append(node);
16     end
17     return results;
18 End Function
```

Στο σημείο αυτό, θα επαναλάβουμε σύντομα την διαδικασία με την οποία απομονώνουμε τους βρόγχους στόχους για κάθε αλγόριθμο. Αρχικά, για κάθε αλγόριθμο θέτουμε κάποια ονόματα στόχους. Με βάση τα ονόματα αυτά απομονώνουμε όλες τις συναρτήσεις οι οποίες περιέχουν τουλάχιστον ένα από τα ονόματα αυτά στο όνομα τους (η διαδικασία αυτή περιγράφεται αναλυτικά στον Αλγόριθμο 5). Στην συνέχεια, για κάθε αλγόριθμο θέτουμε ένα ελάχιστο βάθος εμφωλευμένων βρόγχων το οποίο απαιτείται για την υλοποίηση του αλγορίθμου αυτού. Για κάθε συνάρτηση που απομονώσαμε στο προηγούμενο βήμα βρίσκουμε όλους εκείνους του βρόγχους που έχουν κατ' ελάχιστο το απαιτούμενο βάθος εμφωλευμένων βρόγχων (όπως περιγράφεται στον Αλγόριθμο 6). Με βάση την διαδικασία αυτή δημιουργήσαμε το παρακάτω σύνολο από κομμάτια κώδικα που υλοποιούνται με επαναληπτικούς βρόγχους.

Αλγόριθμος	Πλήθος δεδομένων
dot	1897
swap	905
axpy	1233
scal	997
fft	1833
gemm	4062
floyd warshall	2502
bubble sort	9183
insertion sort	7025
selection sort	5471

Πίνακας 3.5: Σύνολο δεδομένων με βρόγχους

Τέλος, πριν προχωρήσουμε παρακάτω θεωρούμε χρήσιμο να παραθέσουμε και ένα παράδειγμα στο οποίο αναπαρίσταται ο τρόπος που εφαρμόζεται η παραπάνω διαδικασία. Με το παράδειγμα αυτό επιθυμούμε να γίνει περισσότερο κατανοητή η διαδικασία με την οποία απομονώνεται ένας βρόγχος, που υλοποιεί έναν αλγόριθμο, από ένα αρχείο.



Σχήμα 3.4: Παράδειγμα απομόνωσης βρόγχου από αρχείο

3.3.2 Δημιουργία συνόλου δεδομένων με συναρτήσεις

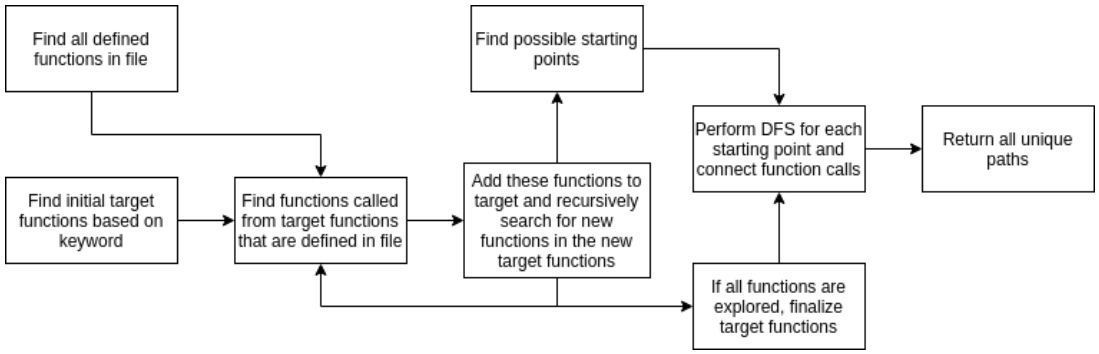
Στην ενότητα αυτή θα ασχοληθούμε αναλυτικά με τον τρόπο απομόνωσης μίας συνάρτησης στόχου από ένα αρχείο κώδικα. Πιο συγκεκριμένα, για ένα γνωστό αλγόριθμο (ή μέθοδο) επιθυμούμε να απομονώσουμε από ένα αρχείο κώδικα (το οποίο πιθανολογούμε ότι περιέχει τον αλγόριθμο αυτό) μόνο την συνάρτηση που περιέχει τον αλγόριθμο αυτό. Φυσικά, η συγκεκριμένη διαδικασία είναι αρκετά περίπλοκη καθώς όπως και στην προηγούμενη περίπτωση υπάρχουν πολλές πιθανές συναρτήσεις μέσα στο κάθε αρχείο κώδικα. Παράλληλα, η διαδικασία γίνεται ακόμα πιο περίπλοκη από το γεγονός ότι στην ουσία αναζητούμε περισσότερες από μία συναρτήσεις καθώς μία συνάρτηση είναι συχνό να καλεί άλλες συναρτήσεις κατά την εκτέλεση της. Για τον σκοπό αυτό αναπτύξαμε μία αυτοματοποιημένη μέθοδο η οποία απομονώνει όλες τις συναρτήσεις που σχετίζονται με τον αλγόριθμο στόχο από ένα αρχείο κώδικα. Στην συνέχεια, θα περιγράψουμε αναλυτικά την διαδικασία με την οποία λειτουργεί η μέθοδος αυτή.

Αρχικά η μεγάλη δυσκολία και σε αυτή την περίπτωση έγκειται στο γεγονός ότι δεν γνωρίζουμε σε ποίο σημείο του κώδικα πρέπει να κοιτάξουμε για να βρούμε μία από τις συναρτήσεις που αναζητούμε. Για τον σκοπό αυτό, όπως και προηγουμένως αποφασίσαμε να αναζητήσουμε τις συναρτήσεις που εμπεριέχονται στο αρχείο κώδικα των οποίων το όνομα εμπεριέχει μέσα κάποια λέξη κλειδί η οποία σχετίζεται με τον αλγόριθμο στόχο και ταυτόχρονα δεν εμπεριέχει κάποιες λέξεις οι οποίες θεωρούνται ως λέξεις προς αποφυγή (η διαδικασία αυτή περιγράφεται αναλυτικά από τον Αλγόριθμο 5). Ο λόγος για τον οποίο επιλέξαμε και πάλι την στρατηγική αυτή είναι γιατί θεωρήσαμε ότι η υπόθεση, ότι τουλάχιστον μία από τις συναρτήσεις που υλοποιούν έναν αλγόριθμο ή μέθοδο περιέχουν στο όνομα τους τουλάχιστον μία λέξη που σχετίζεται με τον αλγόριθμο αυτό, είναι αρκετά ισχυρή. Παράλληλα, η επιλογή της μεθόδου αυτής σχετίζεται και με την πειραματική επιτυχία που επέδειξε κατά την δημιουργία του συνόλου δεδομένων του Σχήματος 3.5.

Αφού λοιπόν εντοπίσαμε τα πιθανά σημεία τα οποία μπορεί να συνδέονται με την υλοποίηση του αλγορίθμου στόχου, απομένει να εντοπίσουμε και να συνδέσουμε τις υπόλοιπες συναρτήσεις που σχετίζονται με την υλοποίηση του αλγορίθμου αυτού. Για τον σκοπό αυτό, κατά την διαδικασία εκτέλεσης του αλγορίθμου 5 αποθηκεύουμε όλες τις συναρτήσεις που εντοπίσαμε μέσα στο αρχείο που προσπελάσαμε. Πιο συγκεκριμένα, για κάθε συνάρτηση που εντοπίσαμε αποθηκεύουμε το όνομα της, το ακριβές σημείο στο οποίο την εντοπίσαμε καθώς και τον κόμβο του AST που αντιστοιχεί στην συνάρτηση αυτή. Στην συνέχεια, για όλα τα αποτελέσματα που μας επέστρεψε η εκτέλεση του αλγορίθμου 5 (τα οποία και είναι μεμονωμένες συναρτήσεις) αναζητούμε και αποθηκεύουμε τα ονόματα των συναρτήσεων που καλούνται στο εσωτερικό της κάθε συνάρτησης.

Στο σημείο αυτό, γνωρίζουμε τόσο το σύνολο των συναρτήσεων που ορίζονται σε ένα αρχείο όσο και το σύνολο των συναρτήσεων που καλούνται μέσα από τα αποτελέσματα της εκτέλεσης του αλγορίθμου 5. Στην συνέχεια, αρκεί να πραγματοποιήσουμε δύο ακόμα διαδικασίες για να δημιουργήσουμε το σύνολο των δεδομένων. Αρχικά, για όλες τις συναρτήσεις, που γνωρίζουμε ότι καλούνται από τις πιθανές συναρτήσεις στόχους, εντοπίζουμε τις συναρτήσεις που καλούνται μέσα από αυτές. Μάλιστα, η διαδικασία αυτή πραγματοποιείται αναδρομικά καθώς από κάθε συνάρτηση μπορεί να προκύψουν νέες συναρτήσεις που απαιτούν διερεύνηση. Αφού λοιπόν έχουμε

αποθηκεύσει όλες τις συναρτήσεις που σχετίζονται με τον αλγόριθμο στόχο απομένει να συνδέσουμε τις συναρτήσεις αυτές μεταξύ τους για την δημιουργία μίας τελικής μεθόδου. Πιο συγκεκριμένα, από τις συναρτήσεις αυτές απομονώσαμε εκείνες οι οποίες δεν καλούνται από καμία άλλη συνάρτηση που ανήκει στις συναρτήσεις στόχους. Τις συναρτήσεις αυτές τις θεωρήσαμε σαν τον αρχικό κόμβο ενός μονοπατιού. Στην συνέχεια, για κάθε αρχικό κόμβο εκτελέσαμε μία απλή διάσχιση κατά βάθος (DFS) κατά την οποία σαν ακμές του γράφου θεωρήσαμε τις κλήσεις συναρτήσεων. Στην συνέχεια, παραθέτουμε ένα διάγραμμα το οποίο περιγράφει την λειτουργία του κώδικα μας. Επιλέξαμε, να περιγράψουμε την λειτουργία του κώδικα μας με διάγραμμα και όχι με έναν Αλγόριθμο καθώς η λειτουργία αυτή απαιτεί κάποιες εκατοντάδες γραμμές κώδικα για να υλοποιηθεί.



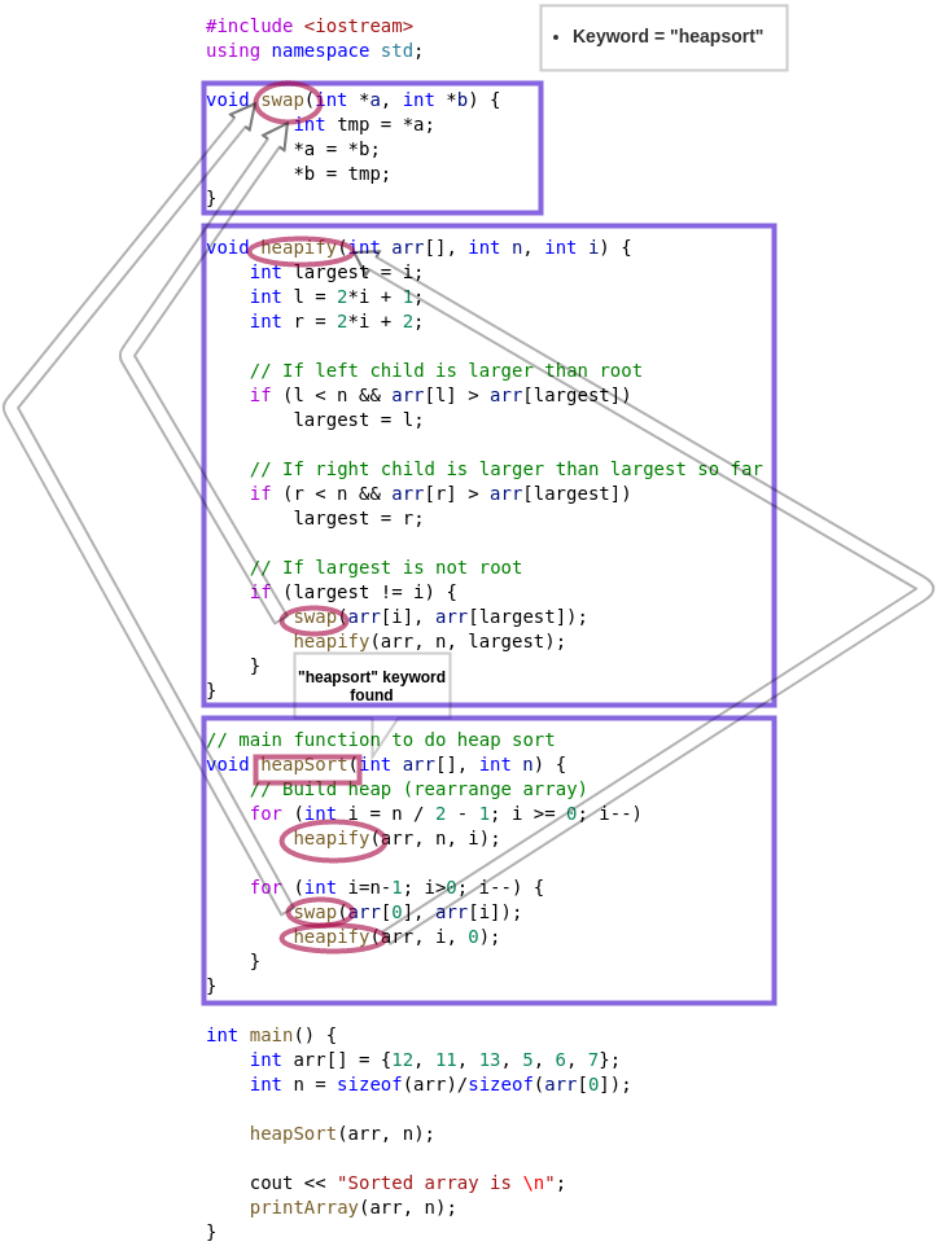
Σχήμα 3.5: Περιγραφή απομόνωσης μεθόδου από αρχείο

Με βάση την διαδικασία που περιγράψαμε παραπάνω δημιουργήσαμε ένα σύνολο δεδομένων που αποτελείται από μεθόδους. Στην συνέχεια παραθέτουμε έναν πίνακα στον οποίο αναφέρονται αναλυτικά το πλήθος δεδομένων για κάθε αλγόριθμο του συνόλου δεδομένων.

Αλγόριθμος	Πλήθος δεδομένων
bfs	9018
dfs	13064
reverse list	7537
heap sort	9502
radix sort	3864
quicksort	17594
mergesort	18839
union find	4118
bubble sort	10653
insert in list	2366
insertion sort	8550
selection sort	6743
binary search	13004
height of tree	6479

Πίνακας 3.6: Σύνολο δεδομένων με συναρτήσεις

Τέλος, πριν προχωρήσουμε παρακάτω θεωρούμε χρήσιμο να παραθέσουμε και ένα παράδειγμα στο οποίο αναπαρίσταται ο τρόπος που εφαρμόζεται η παραπάνω διαδικασία. Με το παράδειγμα αυτό επιθυμούμε να γίνει περισσότερο κατανοητή η διαδικασία με την οποία απομονώνεται μία μέθοδος, που υλοποιεί έναν αλγόριθμο, από ένα αρχείο.




```
#include <iostream>
using namespace std;

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i=n-1; i>0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

Σχήμα 3.6: Παράδειγμα απομόνωσης μεθόδου από αρχείο

Κεφάλαιο 4

Επεξεργασία συνόλου δεδομένων

Στο κεφάλαιο αυτό θα ασχοληθούμε με την επεξεργασία των συνόλων δεδομένων που δημιουργήσαμε στην προηγούμενη ενότητα. Πιο συγκεκριμένα, είναι συνήθης πρακτική η προ-επεξεργασία ενός συνόλου δεδομένων πριν χρησιμοποιηθεί σαν είσοδο σε ένα μοντέλο μηχανικής μάθησης έτσι ώστε να διασφαλιστεί η ποιότητα των πειραμάτων που θα πραγματοποιήσουμε με το μοντέλο αυτό. Για τον σκοπό αυτό, θα πραγματοποιήσουμε τα εξής βήματα:

- Καθαρισμός συνόλου δεδομένων από αντίγραφα
- Προ-επεξεργασία συνόλου δεδομένων

Στην συνέχεια, θα κάνουμε κάποιες πειραματικές παρατηρήσεις σχετικά με τα σύνολα δεδομένων που δημιουργήσαμε έτσι ώστε να επιτευχθεί μία καλύτερη κατανόηση των συνόλων αυτών.

4.1 Καθαρισμός συνόλων δεδομένων από αντίγραφα

Στο προηγούμενο κεφάλαιο ασχοληθήκαμε με τα αντίγραφα σε επίπεδο αρχείου. Πιο συγκεκριμένα, εντοπίσαμε και απομονώσαμε τα αρχεία εκείνα τα οποία θεωρήθηκαν αντίγραφα από την μέθοδο που υλοποιήσαμε. Στην ενότητα αυτή, θα ασχοληθούμε με τα αντίγραφα σε επίπεδο μεθόδου. Αυτό πρακτικά σημαίνει ότι μας ενδιαφέρει να διαχειριστούμε τις μεθόδους αντίγραφα που εντοπίζονται σε ένα από τα δύο σύνολα δεδομένων που δημιουργήσαμε. Αρχικά, οφείλουμε και πάλι να αποφασίσουμε αν τα αντίγραφα αυτά ανήκουν στην αληθινή κατανομή των δεδομένων ή όχι. Η αλήθεια είναι ότι στην περίπτωση αυτή είναι αρκετά πιθανό τα αντίγραφα αυτά να ανήκουν στην αληθινή κατανομή των δεδομένων καθώς είναι συχνό φαινόμενο διαφορετικοί προγραμματιστές να υλοποιούν με ίδιο ή παρόμοιο τρόπο μία συγκεκριμένη μέθοδο. Βέβαια, πρέπει να συνυπολογίσουμε και το γεγονός ότι η ύπαρξη αντιγράφων σε ένα σύνολο δεδομένων είναι πιθανόν να οδηγήσει σε εσφαλμένα συμπεράσματα τόσο κατά την εκπαίδευση όσο και κατά την αξιολόγηση ενός μοντέλου. Πιο συγκεκριμένα, αν δεδομένα του συνόλου εκπαίδευσης βρίσκονται στο σύνολο αξιολόγησης είναι δεδομένο ότι το μοντέλο μας θα προβλέψει με επιτυχία την ετικέτα των δεδομένων αυτών. Παράλληλα, κάτι ανάλογο θα συμβεί και κατά την διάρκεια της εκπαίδευσης αν δεδομένα του συνόλου εκπαίδευσης βρίσκονται και στο σύνολο επαλήθευσης. Ως εκ τούτου, αποφασίσαμε ότι παρόλο που η πλειοψηφία των δεδομένων αυτών ανήκουν στην αληθινή κατανομή των δεδομένων δεν

είναι επιθυμητό να βρίσκονται στο σύνολο δεδομένων κατά την εκπαίδευση καθώς όπως αναφέρεται και στην βιβλιογραφία[36] το γεγονός αυτό θα μπορούσε να οδηγήσει σε εσφαλμένα συμπεράσματα σχετικά με την επιτυχία του μοντέλου μας. Όμως, αποφασίσαμε να διατηρήσουμε τα δεδομένα αυτά σε ένα ξεχωριστό σύνολο αξιολόγησης έτσι ώστε να επαληθεύσουμε κατά τα πειράματα μας ότι το μοντέλο αναγνωρίζει όλα τα αντίγραφα του συνόλου δεδομένων με μεγάλη ακρίβεια.

Αρχικά, για την απομόνωση των αντιγράφων προσπαθήσαμε να χρησιμοποιήσουμε την μέθοδο του Σχήματος 3.3. Παρατηρήσαμε όμως πειραματικά ότι η μέθοδος αυτή δεν λειτουργεί με την ίδια επιτυχία με την οποία λειτουργούσε σε επίπεδο αρχείου. Έτσι, αναζητήσαμε στην βιβλιογραφία μία νέα μέθοδο η οποία υλοποιεί ότι και η μέθοδος του Σχήματος 3.3 αλλά σε επίπεδο μεθόδου. Κατά την μέθοδο αυτή[37], μετατρέπουμε ένα κώδικα σε μία λίστα συνόλου εμφανίσεων ανάλογη με το παράδειγμα του Πίνακα 3.4. Στην συνέχεια, για να διαπιστώσουμε αν δύο κώδικες είναι όμοιοι μεταξύ τους υπολογίζουμε την ομοιότητα σύμφωνα με την σχέση του Σχήματος 4.1. Αν η ομοιότητα τους είναι μεγαλύτερη ή ίση από ένα ελάχιστο όριο ομοιότητας, η τιμή του ορίου είναι 0.8 σύμφωνα με την βιβλιογραφία[37], τότε οι δύο κώδικες θεωρούνται αντίγραφα. Τέλος, αξίζει να σημειώσουμε ότι η υλοποίηση της μεθόδου αυτής είναι παρόμοια με αυτή του Αλγορίθμου 3, με την κύρια διαφορά να είναι ότι αλλάζει η εξίσωση ομοιότητας.

$$Overlap = \frac{|f_1 \cap f_2|}{max(f_1, f_2)}$$

Σχήμα 4.1: Εξίσωση ομοιότητας $Overlap$ για τα σύνολα λέξεων δύο αρχείων f_1 και f_2

Στην συνέχεια, για την καλύτερη κατανόηση της μεθόδου αυτής[37] από τον αναγνώστη θα παραθέσουμε ένα παράδειγμα αντιγράφων όπως εντοπίστηκε κατά την εκτέλεση της μεθόδου στο σύνολο δεδομένων με συναρτήσεις. Αξίζει να σημειώσουμε ότι η ομοιότητα μεταξύ των δύο αντιγράφων υπολογίστηκε στην τιμή 0.91.

```
void radixsort ( int * a , int n ) {
 int i , b [ MAX ] , m = a [ 0 ] , exp = 1 ;
 for ( i = 0 ; i < n ; i ++ ) {
   if ( a [ i ] > m ) {
     m = a [ i ] ;
   }
 }
 while ( m / exp > 0 ) {
   int bucket [ 10 ] = {
     0 };
   for ( i = 0 ; i < n ; i ++ ) {
     bucket [ a [ i ] / exp % 10 ] ++ ;
   }
   for ( i = 1 ; i < 10 ; i ++ ) {
     bucket [ i ] += bucket [ i - 1 ] ;
   }
   for ( i = n - 1 ; i >= 0 ; i -- ) {
     b [ -- bucket [ a [ i ] / exp % 10 ] ] = a [ i ] ;
   }
   for ( i = 0 ; i < n ; i ++ ) {
     a [ i ] = b [ i ] ;
   }
   exp *= 10 ;
 }
}
```

Σχήμα 4.2: Αντίγραφο 1 του αλγόριθμου $radixsort$

```
void radixsort ( int a [ 6 ] ) {
 int i , b [ 6 ] , m = 0 , exp = 1 , n = 6 ;
 int bucket [ 10 ] = {
   0 };
 for ( i = 0 ; i < n ; i ++ ) {
   if ( a [ i ] > m ) {
     m = a [ i ] ;
   }
 }
 while ( m / exp > 0 ) {
   bucket [ 0 ] = 0 ;
   for ( i = 0 ; i < n ; i ++ ) {
     bucket [ a [ i ] / exp % 10 ] ++ ;
   }
   for ( i = 1 ; i < 10 ; i ++ ) {
     bucket [ i ] += bucket [ i - 1 ] ;
   }
   for ( i = n - 1 ; i >= 0 ; i -- ) {
     b [ -- bucket [ a [ i ] / exp % 10 ] ] = a [ i ] ;
   }
   for ( i = 0 ; i < n ; i ++ ) {
     a [ i ] = b [ i ] ;
   }
   exp *= 10 ;
 }
}
```

Σχήμα 4.3: Αντίγραφο 2 του αλγόριθμου $radixsort$

4.1.0.1 Καθαρισμός συνόλου δεδομένων με επαναληπτικούς βρόγχους

Στο σημείο αυτό, θα εφαρμόσουμε την μέθοδο που περιγράψαμε παραπάνω στο σύνολο δεδομένων με επαναληπτικούς βρόγχους που δημιουργήσαμε στο προηγούμενο κεφάλαιο. Πριν όμως προχωρήσουμε στην εκτέλεση της μεθόδου αυτής θεωρήσαμε απαραίτητο να αποφασίσουμε τόσο την οριακή τιμή που ορίζει πότε δύο κώδικες αποτελούν αντίγραφα όσο και αν όλα τα αρχεία είναι κατάλληλα για την εφαρμογή της μεθόδου αυτής. Αρχικά, όσον αναφορά την καταλληλότητα όλων των αρχείων είναι προφανές πως εφόσον η μέθοδος αυτή εξαρτάται από τα ονόματα μεταβλητών και συναρτήσεων τα συνθετικά δεδομένα που έχουμε κατασκευάσει δεν είναι κατάλληλα για την μέθοδο αυτή. Ως εκ τούτου, για τις κλάσεις που περιλαμβάνουν συνθετικά δεδομένα (dot, swap, axpy, scal) δεν θα εφαρμόσουμε την μέθοδο αυτή. Πάραυτα, θεωρούμε σκόπιμο να μεριμνήσουμε έστω για την μη ύπαρξη κωδίκων οι οποίοι είναι ακριβώς ίδιοι με άλλος κώδικες. Έτσι, για τις κλάσεις αυτές θα απομονώσουμε μόνο τους κώδικες που είναι ακριβή αντίγραφα. Για τις υπόλοιπες κλάσεις αλγορίθμων αποφασίσαμε να εφαρμόσουμε την μέθοδο που περιγράφηκε στην προηγούμενη παράγραφο χρησιμοποιώντας ως όριο ομοιότητας την τιμή 0.9. Ο λόγος που αποφασίσαμε να αποκλίνουμε από την θεωρητική τιμή η οποία είναι 0.8 βασίζεται σε καθαρά πειραματικά δεδομένα. Πιο συγκεκριμένα, παρατηρήσαμε ότι για τιμές ομοιότητας στο διάστημα από 0.8 έως 0.9 απομονωνόταν πολλοί βρόγχοι οι οποίοι για τα δεδομένα του προβλήματος δεν ήταν και τόσο όμοιοι. Αυτό πρακτικά σημαίνει ότι λόγω της φύσης του συνόλου δεδομένων, το οποίο αποτελείται από βρόγχους μικρού μήκους, είναι φυσικό να παρατηρείται μεγάλη ομοιότητα ανάμεσα στα δεδομένα και έτσι αποφασίσαμε να επιλέξουμε μία πιο συντηρητική τιμή ορίου ομοιότητας. Στην συνέχεια παραθέτουμε έναν πίνακα στον οποίο αναφέρονται αναλυτικά το πλήθος δεδομένων για κάθε αλγόριθμο του συνόλου δεδομένων.

Αλγόριθμος	Πλήθος δεδομένων	Ποσοστό επί του συνόλου(%)
dot	1185	7.0
swap	1748	10.3
axpy	643	3.8
scal	928	5.4
fft	1096	6.4
gemm	2277	13.4
floyd warshall	1264	7.4
bubble sort	3236	19.0
insertion sort	2461	14.4
selection sort	2200	12.9

Πίνακας 4.1: Σύνολο δεδομένων με βρόγχους

Συγκρίνοντας τον πίνακα αυτό με τον πίνακα 3.5 παρατηρούμε ότι "χάσαμε" περίπου το 50% των αρχείων από την διαδικασία αυτή. Το γεγονός αυτό είναι ιδιαίτερα ανησυχητικό καθώς τα αρχεία αυτά είναι σίγουρο πως θα φούσκωναν τις μετρικές αξιολόγησης κατά την διαδικασία αξιολόγησης του μοντέλου μας, οδηγώντας έτσι σε αρκετά αισιόδοξα συμπεράσματα.

4.1.0.2 Καθαρισμός συνόλου δεδομένων με συναρτήσεις

Στο σημείο αυτό, θα εφαρμόσουμε την μέθοδο που περιγράψαμε παραπάνω στο σύνολο δεδομένων με συναρτήσεις που δημιουργήσαμε στο προηγούμενο κεφάλαιο. Πριν όμως προχωρήσουμε στην εκτέλεση της μεθόδου της απομόνωσης αντιγράφων από το σύνολο δεδομένων θεωρήσαμε απαραίτητο να αποφασίσουμε την οριακή τιμή που ορίζει πότε δύο κώδικες αποτελούν αντίγραφα. Αφού λοιπόν πειραματιστήκαμε αρκετά σχετικά με την επιλογή της κατάλληλης τιμής του ορίου ομοιότητας καταλήξαμε ότι για το συγκεκριμένο σύνολο δεδομένων η καλύτερη τιμή φαίνεται να είναι η τιμή 0.82. Όπως παρατηρούμε η τιμή αυτή είναι αρκετά κοντά στην τιμή που ορίζει η βιβλιογραφία η οποία είναι 0.8[37]. Κατά την εκτέλεση των πειραμάτων παρατηρήσαμε ότι οι κώδικες που είχαν ομοιότητα μεγαλύτερη του 0.9 ήταν στην ουσία όμοιοι με ελάχιστες διαφορές. Οι κώδικες όμως που είχαν τιμή ομοιότητας κοντά στο 0.85 παρουσίαζαν σε μερικές περιπτώσεις ουσιαστικές διαφορές από τον κώδικα τον οποίο τις συγκρίναμε. Δεδομένου του ότι όπως προείπαμε οι κώδικες αυτοί ανήκουν στην αληθινή κατανομή των δεδομένων μας αποφασίσαμε η οριακή τιμή της ομοιότητας να είναι λίγο πιο κοντά στην τιμή 0.85. Στην συνέχεια παραθέτουμε έναν πίνακα στον οποίο αναφέρονται αναλυτικά το πλήθος δεδομένων για κάθε αλγόριθμο του συνόλου δεδομένων.

Αλγόριθμος	Πλήθος δεδομένων	Ποσοστό επί του συνόλου(%)
bfs	7654	8.8
dfs	10566	12.1
reverse list	5786	6.6
heap sort	6870	7.9
radix sort	2933	3.4
quicksort	11983	13.8
mergesort	13159	15.1
union find	3207	3.7
bubble sort	4991	5.7
insert in list	2033	2.3
insertion sort	4080	4.7
selection sort	3681	4.2
binary search	7066	8.1
height of tree	3082	3.5

Πίνακας 4.2: Σύνολο δεδομένων με συναρτήσεις

Συγκρίνοντας τον πίνακα αυτό με τον πίνακα 3.6 παρατηρούμε ότι "χάσαμε" περίπου 50000 κώδικες από την διαδικασία αυτή. Το γεγονός αυτό είναι αρκετά ανησυχητικό καθώς οι κώδικες αυτοί είναι σχεδόν σίγουρο ότι θα έδιναν μία λάθος εντύπωση όσον αφορά την ποιότητα των προβλέψεων του μοντέλου μας. Από την άλλη, αξίζει να σημειώσουμε ότι ίσως αξίζει να διερευνηθεί περαιτέρω η χρησιμότητα αυτών των κωδίκων καθώς μπορεί να μην είναι ιδιαίτερα χρήσιμοι κατά την εκπαίδευση ενός μοντέλου αλλά θα μπορούσαν πιθανώς να χρησιμοποιηθούν κατά την αξιολόγηση ενός μοντέλου, σαν μίας μορφής επαλήθευση ότι το μοντέλο μαθαίνει να μοντελοποιεί σωστά τον χώρο.

4.2 Προ-επεξεργασία συνόλου δεδομένων

Στην ενότητα αυτή θα ασχοληθούμε με την προ-επεξεργασία του συνόλου δεδομένων εφαρμόζοντας διάφορες τεχνικές επεξεργασίας σε κάθε κώδικα του συνόλου δεδομένων έτσι ώστε να μεγιστοποιήσουμε τόσο την επίδοση όσο και την απόδοση των μοντέλων μας. Πιο συγκεκριμένα, στην ενότητα αυτή αρχικά θα επιβάλουμε σε κάθε κώδικα του συνόλου δεδομένων έναν ενιαίο και ομοιόμορφο τρόπο γραφής. Με την αλλαγή αυτή επιθυμούμε, να επιτύχουμε τόσο καλύτερη μοντελοποίηση του προβλήματος όσο και ευκολότερη διαχείριση του συνόλου δεδομένων από τον προγραμματιστή. Στην συνέχεια, για κάθε κώδικα θα δημιουργήσουμε διαφορετικές εκδοχές, οι οποίες θα διαφέρουν ως προς τα ονόματα μεταβλητών, συναρτήσεων καθώς και τον συνολικό τρόπο παρουσίασης, έτσι ώστε να μελετήσουμε την επίδραση των στοιχείων αυτών στην απόδοση ενός μοντέλου κατά την επίλυση του προβλήματος της ταξινόμησης πηγαίου κώδικα σε αλγορίθμους.

4.2.1 Επιβολή ενιαίας μορφής σε κάθε κώδικα

Αρχικά, θεωρούμε σημαντικό να επιβάλλουμε μία ενιαία μορφή σε κάθε κώδικα των συνόλων δεδομένων καθώς πιστεύουμε ότι μία τέτοια ομοιομορφία όχι μόνο θα διευκολύνει την υλοποίηση των πειραμάτων μας αλλά και θα αυξήσει την απόδοση αυτών. Πιο συγκεκριμένα, είναι πολύ σημαντικό κατά την δημιουργία ενός συνόλου δεδομένων να υπάρχει μία ομοιόμορφη συμπεριφορά ανάμεσα στα δεδομένα. Ένα τέτοιο χαρακτηριστικό, επιτρέπει την εύκολη διαχείριση του συνόλου δεδομένων από τον προγραμματιστή. Για παράδειγμα, είναι σημαντικό να υπάρχει ένας σαφής τρόπος με τον οποίο διαχωρίζονται οι λέξεις μεταξύ τους έτσι ώστε να μπορεί ο προγραμματιστής να αποσυνθέσει έναν κώδικα στο σύνολο των λέξεων από το οποίο αποτελείται. Στην περίπτωση που το χαρακτηριστικό αυτό δεν υπάρχει στο σύνολο των δεδομένων τότε απαιτούνται ειδική χειρισμοί έτσι ώστε να αποφασιστούν τα κριτήρια με τα οποία θα γίνει η παραπάνω αποσύνθεση. Παράλληλα, η επιβολή μίας τέτοιας ομοιόμορφης συμπεριφοράς στο σύνολο των δεδομένων θα μπορούσε να έχει ευεργετικές ιδιότητες και στην μοντελοποίηση του προβλήματος από τα νευρωνικά δίκτυα. Μάλιστα, τα βήματα που θα περιγράψουμε παρακάτω έχουν ταυτόχρονα και έναν σκοπό καθαρισμό του συνόλου δεδομένων από θόρυβο ο οποίος πιθανόν να υπάρχει.

Τέλος, θα θέλαμε να επισημάνουμε ότι για την επιβολή αυτής της ενιαίας μορφής σε κάθε κώδικα του συνόλου δεδομένων χρησιμοποιήσαμε εκτεταμένα κάποιες λειτουργίες του μεταγλωττιστή $clang$ καθώς πολλές από τις τροποποιήσεις που θα περιγράψουμε παρακάτω μπορεί σε ένα πρώτο άκουσμα να μοιάζουν απλές όμως στην πραγματικότητα είναι ιδιαίτερα σύνθετες και απαιτούν την ανάπτυξη ιδιαίτερα περίπλοκων εργαλείων για την επίλυση τους. Για παράδειγμα, ο εντοπισμός των σχολίων σε έναν κώδικα που έχει γραφτεί στην γλώσσα προγραμματισμού C ή $C++$ είναι μια διαδικασία η οποία μπορεί να υλοποιηθεί σχετικά απλά από έναν προγραμματιστή. Δυστυχώς όμως, λόγω του ότι ο κάθε κώδικας έχει διαφορετική μορφή υπάρχει πιθανότητα η απλή διαδικασία αυτή να αποτύχει σε αρκετές περιπτώσεις. Αντιθέτως, παρατηρήσαμε ότι το εργαλείο του μεταγλωττιστή δούλευε ακόμα και σε περιπτώσεις οι οποίες ήταν αρκετά δύσκολο να εντοπιστούν.

4.2.1.1 Διαγραφή σχολίων και σταθερών τύπου string

Όπως είναι γνωστό, οι προγραμματιστές χρησιμοποιούν κατά κόρον σχόλια καθώς και σταθερές τύπου $string$ στην προσπάθεια τους να επικοινωνήσουν τον κώδικα τους σε άλλους προγραμματιστές. Πιο συγκεκριμένα, τέτοια κομμάτια κώδικα δεν συνεισφέρουν στην ουσία ενός κώδικα και έχουν κυρίως επεξηγηματικό χαρακτήρα. Στο σημείο αυτό, θεωρήσαμε σκόπιμο να αποφανθούμε αν τα στοιχεία αυτά είναι επιθυμητό να ανήκουν στους κώδικες των σύνολων δεδομένων. Η συγκεκριμένη λεπτομέρεια μας δημιούργησε αρκετό προβληματισμό καθώς από την μία τέτοια στοιχεία όπως ήδη είπαμε δεν συνεισφέρουν ουσιαστικά σε ένα αρχείο κώδικα. Από την άλλη όμως, είναι πιθανό να περιέχουν πληροφορία ικανή να βοηθήσει στην επίλυση του προβλήματος μας. Αξίζει μάλιστα να σημειώσουμε ότι και στην βιβλιογραφία[42] υπάρχουν διαφορετικές προσεγγίσεις σχετικά με το ζήτημα αυτό.

Αφού λοιπόν σκεφτήκαμε τα παραπάνω, αποφασίσαμε ότι τα στοιχεία αυτά δεν είναι επιθυμητό να ανήκουν στο σύνολο των δεδομένων παρόλο που πιθανώς να βοηθούσαν στην επίλυση του προβλήματος μας. Η συγκεκριμένη απόφαση πάρθηκε με γνώμονα τρία πολύ βασικά κριτήρια. Αρχικά, στην παρούσα εργασία προσπαθούμε να δημιουργήσουμε μοντέλα τα οποία να "κατανοούν" τον κώδικα και τις ιδιότητες του. Ως εκ τούτου, τυχόν βοήθεια από ανθρώπινα σχόλια για την επίλυση του προβλήματος δεν θεωρείται επιθυμητή. Παράλληλα, συχνά τα στοιχεία αυτά όχι μόνο δεν περιέχουν χρήσιμη πληροφορία αλλά είναι πιθανόν η πληροφορία αυτή να μπερδεύει το μοντέλο που δημιουργήσαμε. Για παράδειγμα, συναντήσαμε έναν μικρό αριθμό αρχείων στον οποίο τα σχόλια ήταν στην κινέζικη γλώσσα, γεγονός που θα μπορούσε να μπερδέψει το μοντέλο μας όταν η πλειοψηφία των αρχείων που έχει δει είναι γραμμένα στην αγγλική γλώσσα. Τέλος, δεν πρέπει να ξεχνάμε πως είναι συχνό πολλά από τα στοιχεία του τύπου αυτού να αποτελούνται από ένα μεγάλο πλήθος λέξεων. Το γεγονός αυτό, θα μπορούσε να οδηγήσει στην εκτίναξη του μήκους της ακολουθίας αυξάνοντας έτσι δραματικά τον χρόνο εκπαίδευσης του RNN που θα χρησιμοποιήσουμε αργότερα στα μοντέλα μας. Στην συνέχεια, έχοντας ως γνώμονα την καλύτερη κατανόηση της λειτουργίας αυτής από τον αναγνώστη θα παραθέσουμε ένα απλό παράδειγμα στο οποίο φαίνεται η μετατροπή ενός κώδικα από την αρχική του κατάσταση στην κατάσταση στην οποία έχουμε αφαιρέσει στοιχεία τα οποία αφορούν σχόλια και σταθερές τύπου $string$.

```
void Matrix_multiplication(int A[][10],int B[][10],int mult[][10],int N) {
    // a naive function which perform matrix multiplication between two matrices
    // matricess should be of proper dimensions
    int i, j, k ;
    for ( i = 0 ; i < N ; ++ i ) {
        for ( j = 0 ; j < N ; ++ j ) {
            for ( k = 0 ; k < N ; ++ k ) {
                // if either of these elements are zero continue
                if ( A[i][k] != 0 &&B[k][j] != 0 ) {
                    mult[i][j] += A[i][k]*B[k][j] ;
                }
            }
        }
    }
    cout << "Multiplication completed!" << endl;
}
```

Σχήμα 4.4: Κώδικας με σχόλια και string

```
void Matrix_multiplication(int A[][10],int B[][10],int mult[][10],int N) {
    int i, j, k ;
    for ( i = 0 ; i < N ; ++ i ) {
        for ( j = 0 ; j < N ; ++ j ) {
            for ( k = 0 ; k < N ; ++ k ) {
                if ( A[i][k] != 0 &&B[k][j] != 0 ) {
                    mult[i][j] += A[i][k]*B[k][j] ;
                }
            }
        }
    }
    cout << <STRING> << endl;
}
```

Σχήμα 4.5: Κώδικας χωρίς σχόλια και string

4.2.1.2 Διαγραφή εντολών εισόδου και εξόδου (I/O)

Στην υποενότητα αυτή θα ασχοληθούμε με τις εντολές εισόδου και εξόδου (όπως είναι το $printf$ και το $scanf$) που μπορεί να υπάρχουν σε ένα κομμάτι κώδικα. Πιο συγκεκριμένα, είναι αρκετά συχνό φαινόμενο οι προγραμματιστές να χρησιμοποιούν τέτοιες εντολές στους κώδικες τους έτσι ώστε να επιτύχουν την ταυτόχρονη παρακολούθηση της εξέλιξης του κώδικα καθώς και της ορθής λειτουργίας αυτού. Στο σημείο αυτό, όπως και προηγουμένως θεωρήσαμε σκόπιμο να αποφανθούμε αν τα στοιχεία αυτά είναι επιθυμητό να ανήκουν στους κώδικες των συνόλων δεδομένων. Η απόφαση αυτή δεν είναι ξεκάθαρη καθώς είναι πιθανόν οι εντολές αυτές να περιέχουν ικανή πληροφορία έτσι ώστε να βοηθήσουν στην επίλυση του προβλήματος μας. Για παράδειγμα, είναι πιθανό η πλειοψηφία των προγραμματιστών να γράφει παρόμοιες εντολές εξόδου (τύπου $print$) για την παρακολούθηση της εκτέλεσης του ίδιου αλγορίθμου. Από την άλλη όμως, όπως και με τα σχόλια και τα $string$ (που σχολιάσαμε στην προηγούμενη ενότητα), τα στοιχεία αυτά τις περισσότερες φορές δεν συμμετέχουν στην ουσιαστική λειτουργία ενός κώδικα.

Αφού λοιπόν αναλογιστήκαμε τα υπέρ και τα κατά της ύπαρξης των στοιχείων αυτών σε ένα σύνολο δεδομένων αποφασίσαμε πως τέτοια στοιχεία δεν είναι επιθυμητά στο σύνολο των δεδομένων μας. Οι λόγοι για τους οποίους πήραμε την απόφαση αυτή είναι σχεδόν ταυτόσημοι με τους λόγους που παρουσιάσαμε στην προηγούμενη ενότητα. Ο σημαντικότερος λόγος πίσω από την απόφαση αυτή είναι πως τα στοιχεία αυτά κατά πάσα πιθανότητα θα ωθήσουν τα μοντέλα μας να "μάθουν" χαρακτηριστικά τα οποία δεν έχουν άμεση σχέση με την υλοποίηση του αλγόριθμου. Σε μία πρώτη ανάγνωση, το γεγονός αυτό μπορεί να μην φαίνεται τόσο προβληματικό καθώς στην ουσία για την επίλυση του προβλήματος μας αρκεί να εκπαιδεύσουμε μοντέλα τα οποία να μπορούν αποφασίσουν ποίος αλγόριθμος υλοποιείται σε ένα κομμάτι κώδικα. Βέβαια, αν τα μοντέλα μας μάθουν να λαμβάνουν αποφάσεις επηρεαζόμενα από στοιχεία τέτοιου τύπου τι θα γίνει όταν προσπαθήσουν να πάρουν μία απόφαση για έναν κώδικα ο οποίος δεν περιέχει τέτοια στοιχεία ; Το γεγονός αυτό είναι αρκετά ανησυχητικό, καθώς στην ουσία είναι πιθανό το μοντέλο μας να επηρεαστεί υπερβολικά από τις εντολές αυτές και να μην μπορεί να κάνει σωστή πρόβλεψη όταν οι εντολές αυτές δεν υπάρχουν. Τέλος, πριν προχωρήσουμε παρακάτω αξίζει να σχολιάσουμε ότι η μέθοδος που αναπτύξαμε δεν διαγράφει απλά τις εντολές αυτές αλλά στην περίπτωση που οι εντολές αυτές περικλείονται μέσα σε επαναληπτικούς βρόγχους (και δεν υπάρχουν εντολές άλλου τύπου μέσα σε αυτούς) διαγράφονται ολόκληροι οι επαναληπτικοί βρόγχοι. Για την καλύτερη κατανόηση της λειτουργίας αυτής θα παραθέσουμε ένα απλό παράδειγμα στο οποίο φαίνονται τα αποτελέσματα της διαδικασίας αυτής.

```
void Matrix_multiplication(int A[][10],int B[][10],
                           int mult[][10],int N) {
   int i, j, k ;
   for(i = 0; i < N; ++i ){
       for(j = 0; j < N ; ++j) {
           for(k = 0; k < N; ++k) {
               if(A[i][k] != 0 && B[k][j] != 0){
                   mult[i][j] += A[i][k]*B[k][j] ;
               }
           }
       }
   }
   for(i = 0; i < N ; ++i){
       for(j = 0; j < N; ++j){
           cout << mult[i][j];
       }
       cout << endl;
   }
}
```

Σχήμα 4.6: Κώδικας με εντολές I/0

```
void Matrix_multiplication(int A[][10],int B[][10],
                           int mult[][10],int N) {
   int i, j, k ;
   for(i = 0; i < N; ++i ){
       for(j = 0; j < N ; ++j) {
           for(k = 0; k < N; ++k) {
               if(A[i][k] != 0 && B[k][j] != 0){
                   mult[i][j] += A[i][k]*B[k][j] ;
               }
           }
       }
   }
}
```

Σχήμα 4.7: Κώδικας χωρίς εντολές I/0

4.2.1.3 Επιβολή συγκεκριμένου τρόπου γραφής (style) κώδικα

Στην υποενότητα αυτή θα ασχοληθούμε με τον τρόπο συγγραφής κώδικα και την σημασία αυτού στην επίλυση του προβλήματος μας. Αρχικά, όταν λέμε τρόπο συγγραφής αναφερόμαστε σε χαρακτηριστικά τα οποία σχετίζονται με την παρουσίαση του κώδικα (κενά (whitespaces), στοίχιση (indedantaion), αγκύλες, παρενθέσεις και άλλα). Προφανώς, τα χαρακτηριστικά αυτά δεν σχετίζονται με την ουσιαστική λειτουργία την οποία επιτελεί ένας κώδικας. Αν λοιπόν κανείς αναλογιστεί και τις αποφάσεις που πήραμε προηγουμένως εύκολα θα μπορούσε να θεωρήσει ότι τα στοιχεία αυτά δεν έχουν ιδιαίτερο ενδιαφέρον για την επίλυση του προβλήματος μας. Όμως, παρόλο που τα στοιχεία αυτά όντως δεν συνεισφέρουν στην λειτουργία ενός κώδικα συνεισφέρουν στην αποδοτική επίλυση του προβλήματός μας.

Πιο συγκεκριμένα, είναι πολύ σημαντικό να επιβάλουμε μία ενιαία πολιτική στον τρόπο γραφής ενός κώδικα. Αρχικά, αποφασίσαμε ότι κάθε "λέξη" (ή χαρακτήρας που όμως αποτελεί λέξη) θα διαχωρίζεται από τις υπόλοιπες "λέξεις" με ακριβώς ένα κενό (whitespaces). Η λεπτομέρεια αυτή είναι πολύ σημαντική καθώς με τον τρόπο αυτό για να χωρίσουμε έναν κώδικα σε όλες τις "λέξεις" από τις οποίες αποτελείται αρκεί να διαχωρίσουμε τον κώδικα στα σημεία στα οποία εμφανίζεται ο κενός χαρακτήρας και να διαγράψουμε τυχόν εμφανίσεις του κενού χαρακτήρα από το σύνολο λέξεων που θα προκύψει. Στην συνέχεια, αποφασίσαμε να επιβάλουμε μία ομοιόμορφη στοίχιση σε όλα τα κομμάτια κώδικα των συνόλων δεδομένων. Πιο συγκεκριμένα, για κάθε εντολή που συνηθίζεται να δημιουργεί μία νέα περιοχή στοίχισης (όπως είναι οι εντολές τύπου for και $while$, η έναρξη συνάρτησης, οι εντολές τύπου if και άλλα) επιβάλουμε την δημιουργία μίας περιοχής στοίχισης, στοιχίζοντας προς τα μέσα όλες τις εντολές που περικλείονται στην περιοχή αυτή. Η επιλογή αυτή αν και δεν έχει άμεση συσχέτιση με τα πειράματα μας θα χρησιμοποιηθεί έμμεσα κατά την δημιουργία της χρωματισμένης εκδοχής κώδικα, η οποία θα παρουσιαστεί παρακάτω. Τέλος, αποφασίσαμε να επιβάλλουμε μία συγκεκριμένη πολιτική σχετικά με τους χαρακτήρες όπως είναι οι αγκύλες και οι παρενθέσεις. Υπάρχουν περιπτώσεις στις οποίες ο μεταγλωττιστής επιτρέπει την απουσία των στοιχείων αυτών. Στην προσπάθεια μας όμως, να δημιουργήσουμε κώδικες οι οποίοι ακολουθούν μία ενιαία πολιτική τρόπου γραφής αποφασίσαμε να συμπληρώσουμε τα στοιχεία αυτά στα σημεία στα οποία έχει παραληφθεί η χρήση τους. Δεδομένου ότι, η διαδικασία αυτή ήταν αρκετά περίπλοκη για να την υλοποιήσουμε μόνοι μας χρησιμοποιήσαμε μία λειτουργία του εργαλείου $clang-tidy$ το οποίο εντοπίζει τα σημεία στα οποία λείπουν οι αγκύλες και τις συμπληρώνει. Για την καλύτερη κατανόηση της λειτουργίας αυτής θα παραθέσουμε ένα απλό παράδειγμα στο οποίο φαίνονται τα αποτελέσματα της διαδικασίας αυτής.

```
void Matrix_multiplication(int A[][10],int B[][10],
                           int mult[][10],int N) {
    int i, j, k ;
    for(i = 0; i < N; ++i ){
        for(j = 0; j < N ; ++j) {
            for(k = 0; k < N; ++k) {
                if(A[i][k] != 0 && B[k][j] != 0)
                mult[i][j] += A[i][k]*B[k][j] ;
            }
        }
    }
}
```

Σχήμα 4.8: Αρχικός κώδικας

```
void Matrix_multiplication (int A[][ 10 ] , int B [ ] [ 10 ] ,
                            int mult [ ] [ 10 ] , int N ) {
    int i , j , k ;
    for ( i = 0 ; i < N ; ++i ) {
        for( j = 0 ; j < N ; ++ j ) {
            for( k = 0 ; k < N ; ++ k) {
                if( A [ i ][ k ] != 0 && B [ k ] [ j ] != 0 ) {
                    mult[ i ] [ j ] += A [ i ] [ k ] * B [ k ][ j ] ;
                }
            }
        }
    }
}
```

Σχήμα 4.9: Κώδικας με συγκεκριμένο coding style

4.2.2 Δημιουργία διαφορετικών εκδοχών κώδικα για πειραματισμό

Όπως αναφέραμε και παραπάνω, αφού επιβάλλαμε μια ενιαία σύνταξη και μορφή σε όλους τους κώδικες των συνόλων δεδομένων θα δημιουργήσουμε διαφορετικές εκδοχές για τον κάθε κώδικα έτσι ώστε να αξιολογήσουμε στα πειράματα μας την επίπτωση που έχουν χαρακτηριστικά όπως είναι τα ονόματα μεταβλητών και η στοίχιση στην απόδοση ενός μοντέλου που προσπαθεί να μοντελοποιήσει αρχεία κώδικα.

Αρχικά, στις δύο πρώτες υποενότητες θα ασχοληθούμε με την επεξεργασία των ονομάτων που εμφανίζονται σε ένα κομμάτι κώδικα. Όταν μιλάμε για ονόματα εννοούμε τις λέξεις εκείνες οι οποίες δεν είναι δεσμευμένες λέξεις της γλώσσας προγραμματισμού (για παράδειγμα οι λέξεις for και if), δεν είναι ειδικοί χαρακτήρες τις γλώσσας προγραμματισμού (για παράδειγμα οι χαρακτήρες "!" και ";") καθώς και αριθμοί ή σταθερές. Στην ουσία μας ενδιαφέρουν κυρίως τα ονόματα των μεταβλητών καθώς και των συναρτήσεων τα οποία ορίζονται συνήθως από τον προγραμματιστή. Μάλιστα, όπως είναι γνωστό ο προγραμματιστής έχει το περιθώριο να ονομάσει μια μεταβλητή ή συνάρτηση με οποιοδήποτε όνομα αυτός επιθυμεί. Η συγκεκριμένη ελευθερία που δίνεται από τις γλώσσες προγραμματισμού είναι συχνά ευεργετική καθώς δίνει μεγαλύτερη ελευθερία έκφρασης στον προγραμματιστή, οδηγώντας έτσι σε ποιοτικότερα αρχεία κώδικα. Ταυτόχρονα όμως, το γεγονός αυτό είναι ιδιαίτερα ανησυχητικό για την επίλυση του προβλήματός μας καθώς το πλήθος των δυνατών συνδυασμών που μπορούν να προκύψουν για την ονοματοδοσία των μεταβλητών και συναρτήσεων είναι πρακτικά άπειρο. Ως εκ τούτου, είναι δεδομένο ότι κάποια στιγμή θα υπάρξουν λέξεις που δεν έχει δει το μοντέλο μας, δημιουργώντας έτσι μία αβεβαιότητα στις προβλέψεις του. Παράλληλα, το μεγάλο πλήθος λέξεων μας προβληματίζει ιδιαίτερα και για το γεγονός ότι αυξάνει ιδιαίτερα την υπολογιστική ισχύ που απαιτείται για να το επεξεργαστεί κατά την διάρκεια εκπαίδευσης του μοντέλου μας (η συγκεκριμένη λεπτομέρεια θα σχολιαστεί εκτενώς αργότερα).

Στην συνέχεια, αποφασίσαμε ότι είναι ιδιαίτερα ενδιαφέρον να μελετήσουμε την επίπτωση της στοίχισης στην απόδοση και την επίδοση ενός μοντέλου. Πιο συγκεκριμένα, στην παρούσα εργασία έχουμε αποφασίσει να χειριστούμε τον κώδικα σαν αρχείο κειμένου[41, 44, 45] και όχι σαν δένδρο ή γράφο (που παρατηρείται σε ένα μέρος της βιβλιογραφίας). Το γεγονός αυτό, πιθανόν να δημιουργεί κάποιους περιορισμούς σχετικά με το πόσο ποιοτικά εκφράζεται η στοίχιση και η δομή του κάθε κώδικα. Πιο συγκεκριμένα, για να εκφραστεί σωστά η δομή και η στοίχιση είναι απαραίτητο να συμπεριλάβουμε στην είσοδο του μοντέλου μας και όλους τους κενούς χαρακτήρες που εκφράζουν την στοίχιση. Βέβαια, κάτι τέτοιο δεν είναι πολύ καλή ιδέα καθώς θα αυξήσει δραματικά το μήκος της ακολουθίας εισόδου γεγονός που θα έχει επιπτώσεις τόσο στην απόδοση όσο και στην επίδοση του μοντέλου μας. Ως εκ τούτου, αποφασίσαμε να δημιουργήσουμε μία εκδοχή κώδικα, που θα παρουσιαστεί παρακάτω, η οποία επιθυμούμε να χειριστεί και να επιλύσει τον περιορισμό αυτό.

4.2.2.1 Αντικατάσταση ονομάτων με υποονόματα

Στην προσπάθεια μας να αντιμετωπίσουμε το πρόβλημα που περιγράφηκε παραπάνω αποφασίσαμε να εξετάσουμε μία λύση η οποία χρησιμοποιείται συχνά σύμφωνα με την βιβλιογραφία[41, 42]. Πιο συγκεκριμένα, οι προγραμματιστές όταν θέλουν να δώσουν σαν όνομα μεταβλητής (ή συνάρτησης) μία σύνθετη λέξη συνδυασμός λέξεων) χρησιμοποιούν συνήθως δύο συμβάσεις. Κατά την πρώτη σύμβαση, η οποία ονομάζεται $Snakecase$, οι προγραμματιστές συνδέουν τις διαφορετικές λέξεις μεταξύ τους με την χρήση του ειδικού χαρακτήρα "_". Κατά την δεύτερη σύμβαση, η οποία ονομάζεται $Camelcase$, οι προγραμματιστές συνδέουν τις διαφορετικές λέξεις μεταξύ τους κάνοντας κεφαλαίο το πρώτο γράμμα κάθε λέξης. Στο σημείο αυτό, αξίζει να σημειώσουμε ότι αν και οι συμβάσεις αυτές θεωρούνται καλές πρακτικές και η χρήση τους συνηθίζεται δεν υπάρχουν εγγυήσεις ότι όντως κάποιος προγραμματιστής θα τις χρησιμοποιήσει. Ως εκ τούτου, κάνουμε την θεώρηση ότι ο κώδικας που μας δίνεται πληρεί κατ' ελάχιστο κάποια ποιοτικά κριτήρια.

Αφού λοιπόν κάναμε την παραδοχή ότι οι πρακτικές αυτές τηρούνται, αποφασίσαμε ότι μία καλή λύση που αξίζει να δοκιμάσουμε είναι να σπάσουμε τις σύνθετες λέξεις αυτές στις λέξεις από τις οποίες αποτελούνται. Προφανώς, δεδομένου ότι γνωρίζουμε την σύμβαση με την οποία έχει γίνει η σύνθεση των σύνθετων αυτών λέξεων, η αποδομήσει τους σε απλούστερες λέξεις είναι αρκετά εύκολη, καθώς αρκεί να χωρίσουμε τις λέξεις όπου εμφανίζεται κεφαλαίος χαρακτήρας ή ο ειδικός χαρακτήρας "_" (θεωρούμε ότι μπορεί να ισχύει μία από τις δύο συμβάσεις για κάθε σύνθετη λέξη). Με την λύση αυτή θεωρούμε ότι επιτυγχάνεται η μείωση του συνολικού αριθμού των λέξεων από το οποίο αποτελείται ένα σύνολο δεδομένων. Το γεγονός αυτό συμβαίνει, καθώς όταν μία σύνθετη λέξη σπάει σε απλούστερες λέξεις είναι ευκολότερο οι λέξεις αυτές να υπάρχουν ξανά στο σύνολο δεδομένων (σε αντίθεση με μία πολύ σύνθετη λέξη που είναι αρκετά δύσκολο να υπάρχει όμοια της). Φυσικά, το γεγονός αυτό, θα μπορούσε να λειτουργήσει ευεργετικά και στην απόδοση του μοντέλου μας καθώς μειώνεται η πιθανότητα να μην έχει δει μία λέξη στο παρελθόν. Ακόμα όμως και αν δεν έχει δει κάποια από τις λέξεις που συνθέτουν την σύνθετη λέξη, παίρνει αρκετή πληροφορία από τις λέξεις που ήδη γνωρίζει και περιλαμβάνονται στην λέξη αυτή. Τέλος, για την καλύτερη κατανόηση της εκδοχής αυτής θα παραθέσουμε ένα απλό παράδειγμα στο οποίο φαίνεται η μετατροπή ενός κώδικα από την αρχική του κατάσταση στην κατάσταση στην οποία σπάσαμε τα σύνθετα ονόματα σε απλούστερα.

```
void MatrixMultiplication ( int A [ ] [ 10 ] , int B [ ] [ 10 ] ,
                            int mult [ ] [ 10 ] , int N ) {
    int loop_i , j , k ;
    for ( loop_i = 0 ; loop_i < N ; ++ loop_i ) {
        for ( j = 0 ; j < N ; ++ j ) {
            for ( k = 0 ; k < N ; ++ k ) {
                if ( A [ loop_i ] [ k ] != 0 && B [ k ] [ j ] != 0 ) {
                    mult [ loop_i ] [ j ] += A [ loop_i ] [ k ] * B [ k ] [ j ] ;
                }
            }
        }
    }
}
```

Σχήμα 4.10: Αρχικός κώδικας

```
void Matrix Multiplication ( int A [ ] [ 10 ] , int B [ ] [ 10 ] ,
                             int mult [ ] [ 10 ] , int N ) {
    int loop i , j , k ;
    for ( loop i = 0 ; loop i < N ; ++ loop i ) {
        for ( j = 0 ; j < N ; ++ j ) {
            for ( k = 0 ; k < N ; ++ k ) {
                if ( A [ loop i ] [ k ] != 0 && B [ k ] [ j ] != 0 ) {
                    mult [ loop i ] [ j ] += A [ loop i ] [ k ] * B [ k ] [ j ] ;
                }
            }
        }
    }
}
```

Σχήμα 4.11: Κώδικας με υποονόματα

4.2.2.2 Αντικατάσταση ονομάτων με κανονικοποιημένα ονόματα

Στην υποενότητα αυτή θα ασχοληθούμε και πάλι με το ζήτημα των ονομάτων των μεταβλητών και των συναρτήσεων. Παρατηρήσαμε ότι ένα μικρό μέρος της βιβλιογραφίας[43, 44, 45] έχει επιλέξει να αντικαταστήσει τα ονόματα αυτά με κανονικοποιημένα ονόματα. Πιο συγκεκριμένα, όλα τα ονόματα μεταβλητών και συναρτήσεων αντικαθίστανται από ονόματα που ανήκουν σε ένα περιορισμένο σύνολο λέξεων. Όπως είναι φυσικό, μία τέτοια λύση οδηγεί στην δραματική μείωση του πλήθους των λέξεων του συνόλου δεδομένων. Μία τέτοια μείωση όμως δεν είναι απαραίτητα θετική και είναι σίγουρα κάτι το οποίο χρίζει διερεύνησης.

Στην εργασία αυτή, αποφασίσαμε να δημιουργήσουμε μια δική μας παραλλαγή σχετικά με τον τρόπο που θα γίνει η αντικατάσταση των ονομάτων. Στην βιβλιογραφία[43, 44, 45], κατά κύριο λόγο δεν γίνεται διαχωρισμός του τύπου των μεταβλητών ή συναρτήσεων και η ονοματοδοσία γίνεται κατά κύριο λόγο τυχαία. Βέβαια, το γεγονός αυτό δεν συνάδει ιδιαίτερα με την λογική με την οποία ένας προγραμματιστής γράφει κώδικα. Πιο συγκεκριμένα, είναι πολύ συχνό μεταξύ των προγραμματιστών να χρησιμοποιούνται συγκεκριμένα ονόματα για συγκεκριμένους τύπους μεταβλητών. Ένα παράδειγμα είναι ο μετρητής ενός βρόγχου όπου πολύ συχνά χρησιμοποιείται ο χαρακτήρας “i” και αν αυτός είναι δεσμευμένος για άλλη χρήση τότε χρησιμοποιούνται οι χαρακτήρες “j” ή “k”. Το παράδειγμα αυτό είναι μεν χαρακτηριστικό, αλλά πρέπει να τονίσουμε ότι είναι ένα από τα πολλά παραδείγματα που υπάρχουν καθώς γενικά θεωρείται καλή πρακτική να ακολουθείται μία λογική διαδικασία επιλογής ονομάτων των μεταβλητών και συναρτήσεων. Με βάση την παρατήρηση αυτή, θεωρήσαμε ότι θα ήταν πιο σωστό να δώσουμε κανονικοποιημένα ονόματα στις μεταβλητές ανάλογα με τον τύπο τους. Έτσι, αρχικά διαχωρίσαμε τις μεταβλητές του κάθε αρχείου στις παρακάτω κατηγορίες :

- Μεταβλητή τύπου πίνακα
- Μεταβλητή τύπου μετρητή βρόγχου
- Μεταβλητή τύπου όριο βρόγχου
- Μεταβλητή γενικής χρήσης
- Όνομα συνάρτησης

Στην συνέχεια, αντικαταστήσαμε τα ονόματα όλων των μεταβλητών με ένα κανονικοποιημένο όνομα το οποίο όμως σχετίζεται με την κατηγορία μεταβλητής στην οποία ανήκουν. Παράλληλα, αξίζει να σημειώσουμε ότι η αντικατάσταση του ονόματος μίας μεταβλητής ή συνάρτησης πρέπει να γίνει με ιδιαίτερη προσοχή καθώς πρέπει το κανονικοποιημένο όνομα να εμφανίζεται σε όλες τις εμφανίσεις του αρχικού ονόματος. Τέλος, για την καλύτερη κατανόηση της εκδοχής αυτής θα παραθέσουμε ένα απλό παράδειγμα στο οποίο φαίνεται η μετατροπή ενός κώδικα από την αρχική του κατάσταση στην κατάσταση στην οποία εισάγαμε τα κανονικοποιημένα ονόματα.

```
void Matrix_multiplication ( int Q [ ] [ 10 ] , int Z [ ] [ 10 ] ,
                             int mult [ ] [ 10 ] , int limit) {
    int loop_i , j , k ;
    for ( loop_i = 0 ; loop_i < limit; ++ loop_i ) {
        for ( j = 0 ; j < limit ; ++ j ) {
            for ( k = 0 ; k < limit ; ++ k ) {
                if ( Q [ loop_i ] [ k ] != 0 && Z [ k ] [ j ] != 0 ) {
                    mult [ loop_i ] [ j ] += Q [ loop_i ] [ k ] * Z [ k ] [ j ] ;
                }
            }
        }
    }
}
```

Σχήμα 4.12: Αρχικός κώδικας

```
void func ( int A [ ] [ 10 ] , int B [ ] [ 10 ] ,
            int C [ ] [ 10 ] , int N ) {
    int i , j , k ;
    for ( i = 0 ; i < N ; ++ i ) {
        for ( j = 0 ; j < N ; ++ j ) {
            for ( k = 0 ; k < N ; ++ k ) {
                if ( A [ i ] [ k ] != 0 && B [ k ] [ j ] != 0 ) {
                    C [ i ] [ j ] += A [ i ] [ k ] * B [ k ] [ j ] ;
                }
            }
        }
    }
}
```

Σχήμα 4.13: Κώδικας με κανονικοποιημένα ονόματα

4.2.2.3 Δημιουργία χρωματισμένης έκδοσης κώδικα

Στην υποενότητα αυτή θα ασχοληθούμε με το πρόβλημα της στοίχισης και της δομής του κώδικα. Πιο συγκεκριμένα, όπως αναφέραμε και παραπάνω είναι σχετικά δύσκολο να δώσουμε στο μοντέλο μας πληροφορία σχετικά με την δομή του κώδικα. Ένας τρόπος που έχει μελετηθεί από την βιβλιογραφία[42] είναι να συμπεριληφθούν οι κενοί χαρακτήρες (*whitespaces*) στην είσοδο του μοντέλου. Βέβαια, το γεγονός αυτό θα οδηγούσε στην εκτινάξει του μήκους της ακολουθίας εισόδου γεγονός το οποίο θα έχει ιδιαίτερα σημαντικές επιπτώσεις τόσο στην απόδοση όσο και στην επίδοση ενός μοντέλου. Ως εκ τούτου, αποφασίσαμε να προτείνουμε μία μέση λύση η οποία ναι μεν περιέχει όλη την πληροφορία των κενών χαρακτήρων αλλά δεν αυξάνει δραματικά το μήκος της ακολουθίας εισόδου. Σύμφωνα με την δική μας λύση, γραμμές κώδικα που βρίσκονται σε διαφορετικό βάθος, όπου το βάθος εξαρτάται από το πλήθος των κενών χαρακτήρων που υπάρχουν πριν τον πρώτο αλφαριθμητικό χαρακτήρα της γραμμής, χρωματίζονται με διαφορετικό χρώμα, ενώ οι γραμμές που βρίσκονται στο ίδιο βάθος χρωματίζονται με το ίδιο χρώμα. Στο σημείο αυτό, θα θέλαμε να τονίσουμε ότι η λειτουργία αυτή υλοποιήθηκε αρκετά εύκολα μετά και την επιβολή συγκεκριμένου τρόπου γραφής σε όλους τους κώδικες του συνόλου δεδομένων. Τέλος, για την καλύτερη κατανόηση της εκδοχής αυτής θα παραθέσουμε ένα απλό παράδειγμα στο οποίο φαίνεται η μετατροπή ενός κώδικα από την αρχική του κατάσταση στην κατάσταση στην οποία έχουμε εισάγει τον χρωματισμό των γραμμών κώδικα ανάλογα με το βάθος στο οποίο βρίσκονται.

```
void Matrix_multiplication ( int A [ ] [ 10 ] , int B [ ] [ 10 ] ,
                             int mult [ ] [ 10 ] , int N ) {
    int i , j , k ;
    for ( i = 0 ; i < N ; ++ i ) {
        for ( j = 0 ; j < N ; ++ j ) {
            for ( k = 0 ; k < N ; ++ k ) {
                if ( A [ i ] [ k ] != 0 && B [ k ] [ j ] != 0 ) {
                    mult [ i ] [ j ] += A [ i ] [ k ] * B [ k ] [ j ] ;
                }
            }
        }
    }
}
```

Σχήμα 4.14: Αρχικός κώδικας

```
void Matrix_multiplication ( int A [ ] [ 10 ] , int B [ ] [ 10 ] ,
                             int mult [ ] [ 10 ] , int N ) {
    int i , j , k ;
    for ( i = 0 ; i < N ; ++ i ) {
        for ( j = 0 ; j < N ; ++ j ) {
            for ( k = 0 ; k < N ; ++ k ) {
                if ( A [ i ] [ k ] != 0 && B [ k ] [ j ] != 0 ) {
                    mult [ i ] [ j ] += A [ i ] [ k ] * B [ k ] [ j ] ;
                }
            }
        }
    }
}
```

Σχήμα 4.15: Χρωματισμένη έκδοση κώδικα

4.3 Πειραματική αξιολόγηση της προ-επεξεργασίας στα σύνολα δεδομένων

Στην ενότητα αυτή θα προσπαθήσουμε να αξιολογήσουμε πειραματικά την επίδραση της παραπάνω επεξεργασίας στην ποιότητα των συνόλων δεδομένων. Πιο συγκεκριμένα, αν και έχουμε αναφέρει εκτενώς τους λόγους που κρύβονται πίσω από τις επιλογές μας σχετικά με την προ-επεξεργασία, θεωρούμε ότι υπάρχουν πτυχές των λόγων αυτών που είναι σημαντικό να αξιολογηθούν και πειραματικά εκτός από θεωρητικά. Τέλος, στο σημείο αυτό αξίζει να σημειώσουμε ότι τα παρακάτω πειραματικά αποτελέσματα αφορούν το σύνολο δεδομένων με συναρτήσεις καθώς παρατηρήσαμε ότι τα δύο σύνολα δεδομένων παρουσίαζαν αντίστοιχη πειραματική συμπεριφορά και έτσι δεν υπάρχει ουσιαστικός λόγος να παρουσιάσουμε και τα δύο.

4.3.1 Επίδραση της προ-επεξεργασίας στο πλήθος των λέξεων

Στην υποενότητα αυτή, θα ασχοληθούμε με τον τρόπο που επιδρά η προ-επεξεργασία στο πλήθος των μοναδικών λέξεων του συνόλου δεδομένων. Πιο συγκεκριμένα, επιθυμούμε να μελετήσουμε την επίδραση τόσο των σχολίων και στοιχείων τύπου $string$ όσο και της επιλογής της μορφής της ονοματοδοσίας των μεταβλητών και συναρτήσεων. Στην συνέχεια, παραθέτουμε έναν πίνακα με τα πειραματικά αποτελέσματα της επίδρασης των διαφορετικών μορφών επεξεργασίας στο πλήθος των μοναδικών λέξεων του συνόλου δεδομένων.

Τύπος προ-επεξεργασίας	Πλήθος μοναδικών λέξεων
Καμία επεξεργασία	365257
Διαγραφή σχολίων-$string$	164119
Διαγραφή σχολίων-$string$ και σπάσιμο ονομάτων	55441
Διαγραφή σχολίων-$string$ και κανονικοποιημένα ονόματα	2403

Πίνακας 4.3: Πλήθος μοναδικών λέξεων στο σύνολο δεδομένων με συναρτήσεις

Αρχικά, παρατηρούμε ότι τα στοιχεία τα οποία ανήκουν στην κατηγορία των σχολίων και $string$ αποτελούν περίπου το 55% των μοναδικών λέξεων του συνόλου δεδομένων. Το γεγονός αυτό, μας οδηγεί στο συμπέρασμα ότι τα στοιχεία τέτοιου τύπου θα μπορούσαν να οδηγήσουν σε δραματική αύξηση του χρόνου εκπαίδευσης ενός μοντέλου. Στην συνέχεια, παρατηρούμε ότι το σπάσιμο των ονομάτων των μεταβλητών και συναρτήσεων στα επιμέρους ονόματα τους οδηγεί σε μία περαιτέρω μείωση της τάξεως του 66%. Το γεγονός αυτό χαρακτηρίζεται ως αρκετά ενθαρρυντικό καθώς ταυτίζεται και με τον ουσιαστικό λόγο της τεχνικής αυτής. Πιο συγκεκριμένα, υπενθυμίζουμε ότι με την αποσύνθεση μίας λέξης σε επιμέρους απλούστερες λέξεις επιθυμούμε να αντλήσουμε πληροφορία από τις ήδη γνωστές απλούστερες λέξεις. Τέλος, όπως ήταν και αναμενόμενο παρατηρούμε ότι η κανονικοποίηση των ονομάτων μεταβλητών και συναρτήσεων οδηγεί στην δραματική συρρίκνωση του πλήθους των μοναδικών λέξεων του συνόλου δεδομένων. Φυσικά, για να αξιολογήσουμε την αποτελεσματικότητα της μεθόδου πρέπει να δούμε αν η πληροφορία που χάνεται λόγω της κανονικοποίησης οδηγεί σε μείωση ή αύξηση της αποτελεσματικότητας των μοντέλων μας.

4.3.2 Επίδραση της προ-επεξεργασίας στο μήκος της ακολουθίας

Στην ενότητα αυτή, θα ασχοληθούμε με τον τρόπο που επιδρά η προ-επεξεργασία στο μήκος της ακολουθίας του συνόλου δεδομένων. Πιο συγκεκριμένα, σαν μήκος ακολουθίας ενός κώδικα ορίζουμε τον αριθμό λέξεων στον κώδικα αυτό. Στο σημείο αυτό, αξίζει να σχολιάσουμε ότι ο λόγος που αποφασίσαμε να μελετήσουμε το χαρακτηριστικό αυτό είναι ότι το μήκος της ακολουθίας είναι ιδιαίτερα σημαντικό τόσο στην απόδοση όσο και στην επίδοση των Αναδρομικών Νευρωνικών Δικτύων (RNN).

4.3.2.1 Η επίδραση της επεξεργασίας ονομάτων και της αφαίρεσης σχολίων-string

Στην υποενότητα αυτή, θα ασχοληθούμε με τον τρόπο που επηρεάζουν τα ονόματα μεταβλητών και συναρτήσεων καθώς και τα σχόλια και $string$ στο μήκος της ακολουθίας του συνόλου δεδομένων. Στην συνέχεια, παραθέτουμε 4 πίνακες με τα πειραματικά αποτελέσματα της επίδρασης των διαφορετικών μορφών επεξεργασίας στο μήκος της ακολουθίας του συνόλου δεδομένων.

Κλίμακα πλήθους λέξεων	Πλήθος δεδομένων	Ποσοστό επί του συνόλου (%)
0-250	90779	69.1
250-500	33683	25.6
500-1000	5438	4.1
1000-2000	1143	0.9
2000-7966	288	0.2

Πίνακας 4.4: Μελέτη μήκους ακολουθίας χωρίς επεξεργασία

Κλίμακα πλήθους λέξεων	Πλήθος δεδομένων	Ποσοστό επί του συνόλου (%)
0-250	95350	72.6
250-500	30710	23.4
500-1000	4214	3.2
1000-2000	857	0.7
2000-7966	200	0.2

Πίνακας 4.5: Μελέτη μήκους ακολουθίας μετά από διαγραφή σχολίων-$string$

Κλίμακα πλήθους λέξεων	Πλήθος δεδομένων	Ποσοστό επί του συνόλου (%)
0-250	92769	70.6
250-500	32332	24.6
500-1000	4877	3.7
1000-2000	1069	0.8
2000-7966	284	0.2

Πίνακας 4.6: Μελέτη μήκους ακολουθίας μετά από διαγραφή σχολίων-$string$ και σπάσιμο ονομάτων

Κλίμακα πλήθους λέξεων	Πλήθος δεδομένων	Ποσοστό επί του συνόλου (%)
0-250	95350	72.6
250-500	30710	23.4
500-1000	4214	3.2
1000-2000	857	0.7
2000-7966	200	0.2

Πίνακας 4.7: Μελέτη μήκους ακολουθίας μετά από διαγραφή σχολίων-$string$ και κανονικοποίηση ονομάτων

Αρχικά, παρατηρώντας τους Πίνακες 4.4 και 4.5 παρατηρούμε ότι η διαγραφή των σχολίων και $string$ οδηγεί όπως ήταν αναμενόμενο στην μείωση του μήκους της ακολουθίας. Πιο συγκεκριμένα, στον Πίνακα 4.4 το 5.2% των δεδομένων έχουν μήκος ακολουθίας μεγαλύτερο από 500 ενώ στον Πίνακα 4.5 η τιμή αυτή μειώθηκε στο 4.1%. Στην συνέχεια, παρατηρώντας τους Πίνακες 4.5 και 4.6 παρατηρούμε ότι το σπάσιμο των ονομάτων επιφέρει μία μικρή αύξηση του μήκους ακολουθίας και το ποσοστό των δεδομένων με μήκος μεγαλύτερο του 500 είναι στο 4.7%. Φυσικά, το γεγονός αυτό είναι απολύτως αναμενόμενο καθώς σπάζοντας τις σύνθετες λέξεις σε απλούστερες στην ουσία αυξάνουμε το πλήθος των λέξεων. Τέλος, παρατηρώντας τους Πίνακες 4.5 και 4.7 βλέπουμε, όπως ήταν αναμενόμενο, ότι είναι ταυτόσημοι.

Στο σημείο αυτό, και παρατηρώντας τους παραπάνω πίνακες θεωρούμε σκόπιμο να σχολιάσουμε το γεγονός ότι πιθανός θα μπορούσαμε να αποκόψουμε κάποια από τα δεδομένα του συνόλου έχοντας ως κριτήριο το μήκος της ακολουθίας. Πιο συγκεκριμένα, παρατηρούμε ότι κατά μέσο όρο (ανεξαρτήτως προ-επεξεργασίας) περίπου το 1% των δεδομένων έχουν μήκος ακολουθίας μεγαλύτερο του 1000 και μόλις το 0.2% μήκος μεγαλύτερο του 2000. Μάλιστα, αξίζει να σημειώσουμε ότι εξαιτίας του μέγιστου μήκους ακολουθίας, το οποίο στην περίπτωση αυτή είναι 7966, η εκπαίδευση του μοντέλου μας θα χρειαστεί αρκετές ώρες για να πραγματοποιηθεί. Στην περίπτωση όμως που το μέγιστο μήκος ακολουθίας ήταν κοντά στην τιμή 1000 τότε θα χρειάζονταν αρκετά λεπτά για να ολοκληρωθεί η εκπαίδευση αυτή. Παράλληλα, αξίζει να σημειώσουμε ότι τα δεδομένα που έχουν υπερβολικά μεγάλο μήκος ακολουθίας είναι κατά πάσα πιθανότητα αρκετά θορυβώδη ή και λάθος. Το γεγονός αυτό εξηγείται τόσο πειραματικά όσο και θεωρητικά. Πιο συγκεκριμένα, παρατηρώντας τα δεδομένα εκείνα τα οποία έχουν υπερβολικά μεγάλο μήκος ακολουθίας είδαμε ότι το μήκος αυτό οφειλόταν κατά κύριο λόγο σε θόρυβο. Επιπρόσθετα, η παρατήρηση αυτή εξηγείται και θεωρητικά καθώς βλέπουμε ότι το ποσοστό των δεδομένων που έχουν τόσο μεγάλο μήκος είναι πάρα πολύ μικρό. Ως εκ τούτου, αποφασίσαμε να κόψουμε τα δεδομένα τα οποία έχουν μήκος ακολουθίας μεγαλύτερο του 1000. Βέβαια, δεδομένου ότι έχουμε δημιουργήσει αρκετές εκδοχές επεξεργασίας του κάθε κώδικα οφείλουμε να τονίσουμε ότι η διαδικασία αυτή δεν γίνεται κεντρικά αλλά γίνεται διαφορετικά για κάθε ξεχωριστή εκδοχή επεξεργασίας του κώδικα.

4.3.2.2 Η επίδραση των κενών χαρακτήρων

Στην υποενότητα αυτή, θα μελετήσουμε τον τρόπο με τον οποίο επηρεάζει η παρουσία των κενών χαρακτήρων το μήκος της ακολουθίας του συνόλου δεδομένων. Πιο συγκεκριμένα, στα πειράματα της προηγούμενης υποενότητας αγνοήσαμε τους κενούς χαρακτήρες καθώς γνωρίζουμε από την βιβλιογραφία[42] ότι αυξάνουν δραματικά το μήκος της ακολουθίας οδηγώντας έτσι στα προβλήματα που περιγράψαμε παραπάνω. Παρόλο όμως, που το γεγονός αυτό αναφέρεται στην βιβλιογραφία[42] θεωρήσαμε σκόπιμο να το διερευνήσουμε και πειραματικά. Στην συνέχεια, παραθέτουμε ένα πίνακα με τα πειραματικά αποτελέσματα της επίδρασης των κενών χαρακτήρων στο μήκος της ακολουθίας.

Κλίμακα πλήθους λέξεων	Πλήθος δεδομένων	Ποσοστό επί του συνόλου (%)
0-250	11383	8.7
250-500	50860	38.7
500-1000	46793	35.6
1000-2000	17692	13.5
2000-31280	4603	3.5

Πίνακας 4.8: Μελέτη μήκους ακολουθίας, συμπεριλαμβανομένου των κενών χαρακτήρων, μετά από διαγραφή σχολίων-$string$ και σπάσιμο ονομάτων

Παρατηρώντας τους Πίνακες 4.6 και 4.8, οι οποίοι αφορούν την ίδια τεχνική προ-επεξεργασίας, παρατηρούμε ότι η παρουσία των κενών χαρακτήρων οδηγεί στην εκτίναξη του μήκους της ακολουθίας. Είναι χαρακτηριστικό πως στον Πίνακα 4.6 (όπου δεν προσμετρώνται οι κενοί χαρακτήρες) μόλις το 4.7% του συνόλου των δεδομένων έχει μήκος ακολουθίας μεγαλύτερο του 500. Αντίθετα, στον Πίνακα 4.8 (όπου και προσμετρώνται οι κενοί χαρακτήρες) παρατηρούμε ότι το αντίστοιχο ποσοστό κυμαίνεται στο 52.6%, γεγονός που στην ουσία αποδεικνύει με τον πιο εμφατικό τρόπο την επίδραση των κενών χαρακτήρων στο μήκος της ακολουθίας. Μετά το πείραμα αυτό, θεωρούμε ότι είναι προφανείς οι λόγοι για τους οποίους επιλέξαμε να αγνοήσουμε τους κενούς χαρακτήρες από το σύνολο δεδομένων. Βέβαια, όπως αναφέραμε και παραπάνω έχουμε προτείνει μία νέα τεχνική κατά την οποία κάθε γραμμή κώδικα χρωματίζεται ανάλογα με το βάθος στο οποίο βρίσκεται. Η τεχνική αυτή θεωρούμε ότι περιέχει όλη την πληροφορία που θα περιείχαν οι κενοί χαρακτήρες και ταυτόχρονα αυξάνει το μήκος της ακολουθίας κατά ακριβώς το πλήθος των γραμμών του κάθε κώδικα (Σχήμα 4.15). Για παράδειγμα, ένας κώδικας ο οποίος αποτελείται από 30 γραμμές και το μήκος ακολουθίας του είναι ίσο με 200 μετά την προσθήκη του χαρακτηριστικού του χρωματισμού το μήκος της ακολουθίας του θα είναι ίσο με 230. Φυσικά, παρόλο που θεωρητικά το χαρακτηριστικό αυτό φαίνεται πολλά υποσχόμενο, καθώς συνδυάζει μικρό μήκος ακολουθίας και όλη την πληροφορία που περιέχουν οι κενοί χαρακτήρες, απομένει και η πειραματική αξιολόγηση αυτού, καθώς πρέπει να αξιολογήσουμε κατά πόσο αυξάνει η μειώνει την αποτελεσματικότητα του μοντέλου μας.

4.3.3 Επίδραση της προ-επεξεργασίας στο πλήθος των αντιγράφων

Τέλος, στην υποενότητα αυτή θα ασχοληθούμε με τον τρόπο που επιδρά η προ-επεξεργασία στο πλήθος αντιγράφων του σύνολου των δεδομένων. Η συγκεκριμένη μελέτη κατά κάποιον τρόπο φαίνεται παράδοξη, καθώς έχουμε απομονώσει ήδη σε δύο διαφορετικές φάσεις αντίγραφα από το σύνολο δεδομένων και έτσι περιμένουμε να μην υπάρχουν άλλα αντίγραφα μέσα σε αυτό. Δυστυχώς, η πραγματικότητα είναι κάπως διαφορετική καθώς παρατηρήσαμε ότι για την εκδοχή επεξεργασίας κατά την οποία κανονικοποιήσαμε τα ονόματα μεταβλητών και συναρτήσεων υπάρχουν αντίγραφα τα οποία μάλιστα είναι και ακριβή, δηλαδή είναι ακριβώς ίδια μεταξύ τους. Πιο συγκεκριμένα, στο σύνολο δεδομένων με συναρτήσεις παρατηρήσαμε ότι το 1.74% (1494 αρχεία) του συνόλου των δεδομένων (85902 αρχεία) έχουν τουλάχιστον ένα ακριβές αντίγραφο επίσης μέσα στο σύνολο δεδομένων. Το γεγονός αυτό εξηγείται καθώς με την κανονικοποίηση των ονομάτων των μεταβλητών και συναρτήσεων επιβλήθηκε μία ομοιομορφία ανάμεσα στα δεδομένα. Σύμφωνα με την ομοιομορφία αυτή, κώδικες οι οποίοι διέφεραν μόνο ως προς τα ονόματα των μεταβλητών και συναρτήσεων πλέον είναι ακριβώς ίδιοι. Φυσικά, η πειραματική απόδειξη αυτή μας χαροποιεί ιδιαίτερα καθώς στην ουσία επιβεβαιώθηκε το κίνητρο σύμφωνα με το οποίο αναπτύξαμε την τεχνική αυτή. Επιπρόσθετα, αξίζει να σημειώσουμε ότι η μέθοδός μας στην ουσία βελτίωσε την καλύτερη έως τώρα μέθοδο στον εντοπισμό αντιγράφων κώδικα κατά σχεδόν 2%. Βέβαια, αξίζει να σχολιάσουμε και ένα μειονέκτημα της μεθόδου αυτής το οποίο επίσης παρατηρήθηκε κατά την παρούσα πειραματική αξιολόγηση. Παρατηρήσαμε ότι, ένα πολύ μικρό μέρος των αντιγράφων που εντοπίσαμε ανήκαν σε διαφορετική αλγοριθμική κλάση. Το γεγονός αυτό εξηγείται επίσης από το αρχικό μας κίνητρο, καθώς αλγοριθμικές υλοποιήσεις οι οποίες εκ φύσεως μοιάζουν θα μπορούσαν μετά από μία κανονικοποίηση ονομάτων να είναι ακριβώς οι ίδιες. Το φαινόμενο αυτό παρατηρήθηκε κυρίως για τους αλγορίθμους BFS και DFS οι οποίοι όπως είναι γνωστό έχουν παρόμοιες υλοποιήσεις. Η παρατήρηση αυτή, δεν προκαλεί ιδιαίτερη ανησυχία καθώς εντοπίστηκε περιορισμένα, αλλά πάραυτα θεωρήσαμε ότι αξίζει να σημειωθεί για την πληρότητα της μελέτης αυτής. Τέλος, λόγω των πιθανών προβλημάτων που μπορεί να προκύψουν, αποφασίσαμε να διαγράψουμε τα αντίγραφα αυτά.

Κεφάλαιο 5

Περιγραφή μοντέλων βαθιάς μηχανικής μάθησης

Στο κεφάλαιο αυτό επιθυμούμε να περιγράψουμε τα μοντέλα βαθιάς μηχανικής μάθησης που θα χρησιμοποιήσουμε στα πειράματα μας. Μάλιστα, θεωρούμε το κεφάλαιο αυτό πολύ σημαντικό μέρος της παρούσας εργασίας καθώς στην ουσία τα μοντέλα αυτά θα μας επιτρέψουν να συγκρίνουμε και να αξιολογήσουμε τις διάφορες τεχνικές προ-επεξεργασίας που υλοποιήσαμε και περιγράψαμε στο προηγούμενο κεφάλαιο. Παράλληλα, μέσω του κεφαλαίου αυτού επιθυμούμε να δώσουμε τόσο μία αναλυτική περιγραφή του μοντέλου όσο και να εξηγήσουμε τον σκοπό του κάθε μοντέλου καθώς και τυχόν προσθήκες που πραγματοποιήσαμε σε μοντέλα τα οποία έχουμε εμπνευστεί από την βιβλιογραφία. Στην συνέχεια, παραθέτουμε μία λίστα με τα μοντέλα βαθιάς μηχανικής μάθησης που θα περιγράψουμε στο κεφάλαιο αυτό :

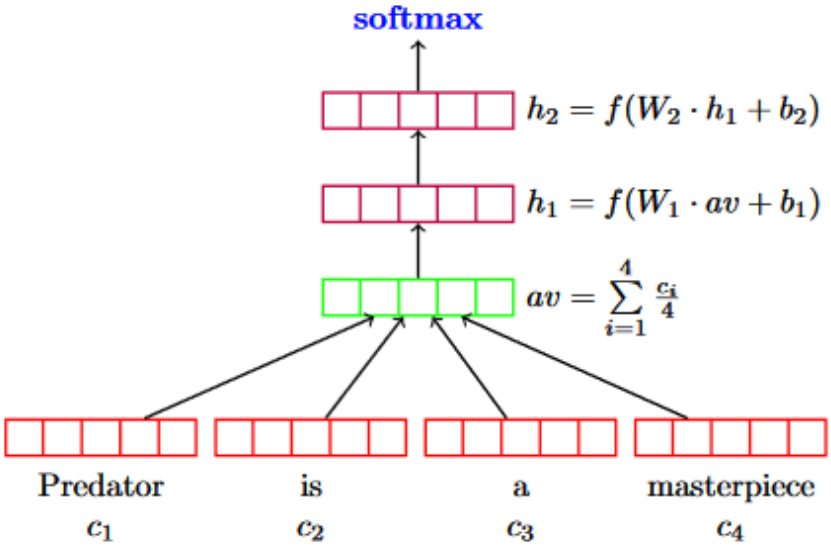
- Deep Averaging Network (DAN)[18]
- Αμφίδρομο LSTM[10] με Attention Layer[17]
- Hierarchical Attention Network (HAN)[17]

Τέλος, θεωρούμε σκόπιμο να σχολιάσουμε ότι τα μοντέλα αυτά σχετίζονται με το κομμάτι της βιβλιογραφίας εκείνο που αφορά μοντέλα βαθιάς μηχανικής μάθησης στην επεξεργασία φυσικής γλώσσας. Ως εκ τούτου, καταλαβαίνει κανείς εύκολα ότι στην παρούσα εργασία προσομοιάζουμε τον κώδικα με την φυσική γλώσσα. Το γεγονός αυτό, είναι σωστό υπό το πρίσμα ότι η φυσική γλώσσα είναι ο τρόπος επικοινωνίας μεταξύ ανθρώπων ενώ ο κώδικας ο τρόπος επικοινωνίας ανθρώπου μηχανής. Βέβαια, είναι γνωστό ότι κώδικας δεν μοιράζεται όλες τις ιδιότητες της φυσικής γλώσσας και αντίστροφα. Ως εκ τούτου, υπάρχει πιθανότητα τα μοντέλα αυτά να μην μοντελοποιούν τον κώδικα κατάλληλα. Φυσικά, θα αξιολογήσουμε την ορθότητα της σκέψης αυτής κατά την πραγματοποίηση των πειραμάτων μας.

5.1 Deep Averaging Network (DAN)

Στην ενότητα αυτή, θα ασχοληθούμε με το μοντέλο *Deep Averaging Network* (DAN)[18]. Το μοντέλο αυτό παρουσιάζει ιδιαίτερο ενδιαφέρον καθώς είναι μάλλον ένα από τα απλούστερα μοντέλα που θα συζητήσουμε στο κεφάλαιο αυτό. Πάραυτα, το μοντέλο αυτό είναι γνωστό ότι είναι ιδιαίτερα αποτελεσματικό σε διάφορα προβλήματα στον τομέα της επεξεργασίας φυσικής γλώσσας. Μάλιστα, είναι χαρακτηριστικό πως χάριν στην απλότητα του μοντέλου η διαδικασία της εκπαίδευσης ολοκληρώνεται σε πολύ λίγο χρόνο σε έναν απλό φορητό υπολογιστή σε αντίθεση με τα υπόλοιπα δίκτυα που θα συζητήσουμε τα οποία απαιτούν εξειδικευμένο υλικό (κάρτες γραφικών τελευταίας τεχνολογίας) για να ολοκληρωθεί η διαδικασία εκπαίδευσης σε σύντομο χρονικό διάστημα.

Στο σημείο αυτό, θα πραγματοποιήσουμε μία σύντομη περιγραφή του μοντέλου αυτού. Στο συγκεκριμένο μοντέλο, αρχικά υπάρχει ένα *Embedding Layer* μέσα από το οποίο δίνεται η είσοδος στο νευρωνικό δίκτυο. Στην συνέχεια, από το *Layer* αυτό για κάθε λέξη της εισόδου επιστρέφεται ένα διάνυσμα το οποίο κωδικοποιεί την σημασία της λέξης αυτής. Έπειτα, για να κωδικοποιήσουμε το νόημα ολόκληρου του κειμένου υπολογίζουμε τον μέσο όρο των διανυσμάτων που επιστρέφει το *Embedding Layer* για την ίδια είσοδο. Το διάνυσμα που προκύπτει, θεωρούμε ότι περιλαμβάνει την κωδικοποιημένη πληροφορία ολόκληρης της εισόδου. Στην συνέχεια, περνάμε το διάνυσμα αυτό από ένα *Layer* κανονικοποίησης έτσι ώστε να υπάρχει ομοιόμορφη συμπεριφορά ανάμεσα σε όλα τα διανύσματα ανεξαρτήτου εισόδου. Τέλος, περνάμε το κάθε διάνυσμα από 2 διαδοχικά *Dense Layers* και προβλέπουμε την κλάση στην οποία ανήκει. Στην συνέχεια, παραθέτουμε ένα σχήμα (5.1) που εξηγεί την διαδικασία που περιγράψαμε.



Σχήμα 5.1: Deep Averaging Network [18]

Στο σημείο αυτό θεωρούμε σκόπιμο να δώσουμε περισσότερες πληροφορίες σχετικά με το μοντέλο που περιγράψαμε παραπάνω. Αρχικά, η είσοδος του μοντέλου δίνεται σε ένα $Embedding\ Layer$. Έχοντας λοιπόν έναν πίνακα με $embeddings\ W_e$ και την είσοδο w_{it} θα υπολογίσουμε την έξοδο του $Embedding\ Layer\ x_{it}$:

$$x_{it} = W_e w_{it} \tag{5.1}$$

Στην συνέχεια, θα δώσουμε το x_{it} σαν είσοδο στο $Layer$ που είναι υπεύθυνο για τον υπολογισμό του μέσου όρου των $embedding$ διανυσμάτων μίας εισόδου. Πιο συγκεκριμένα, έχουμε :

$$x_{it} = \frac{1}{m} \sum_{t=0}^{m} x_{it} \tag{5.2}$$

Στην συνέχεια, θα περάσουμε το x_{it} από ένα $Batch\ Normalization\ Layer$[11] το οποίο επιθυμούμε να αλλάξει την κλίμακα των δεδομένων. Πιο συγκεκριμένα, πιστεύουμε ότι το $Layer$ αυτό θα αυξήσει αρκετά την αποτελεσματικότητα του μοντέλου καθώς όλα τα δεδομένα θα έχουν την ίδια κλίμακα.

$$x_{it} = BatchNormalization(x_{it}) \tag{5.3}$$

Στην συνέχεια, θα περάσουμε το x_{it} από ένα $Dense\ Layer$. Πιο συγκεκριμένα, θα χρησιμοποιήσουμε δύο μεταβλητές τις οποίες θα μαθαίνει το νευρωνικό δίκτυο όσο εκπαιδεύεται. Οι μεταβλητές αυτές είναι ένας πίνακας βαρών W_d και ένα διάνυσμα προκατάληψης b_d. Στην συνέχεια, θα περάσουμε την έξοδο του $Layer$ και πάλι από ένα $Batch\ Normalization\ Layer$. Τέλος, θα εφαρμόσουμε στην έξοδο του $Batch\ Normalization\ Layer$ μία μη γραμμική συνάρτηση ενεργοποίησης η οποία στην περίπτωση αυτή ονομάζεται $Relu$[19].

$$\begin{aligned} z_{it} &= W_d x_{it} + b_d \\ z_{it} &= BatchNormalization(z_{it}) \\ a_{it} &= max(0, z_{it}) \end{aligned} \tag{5.4}$$

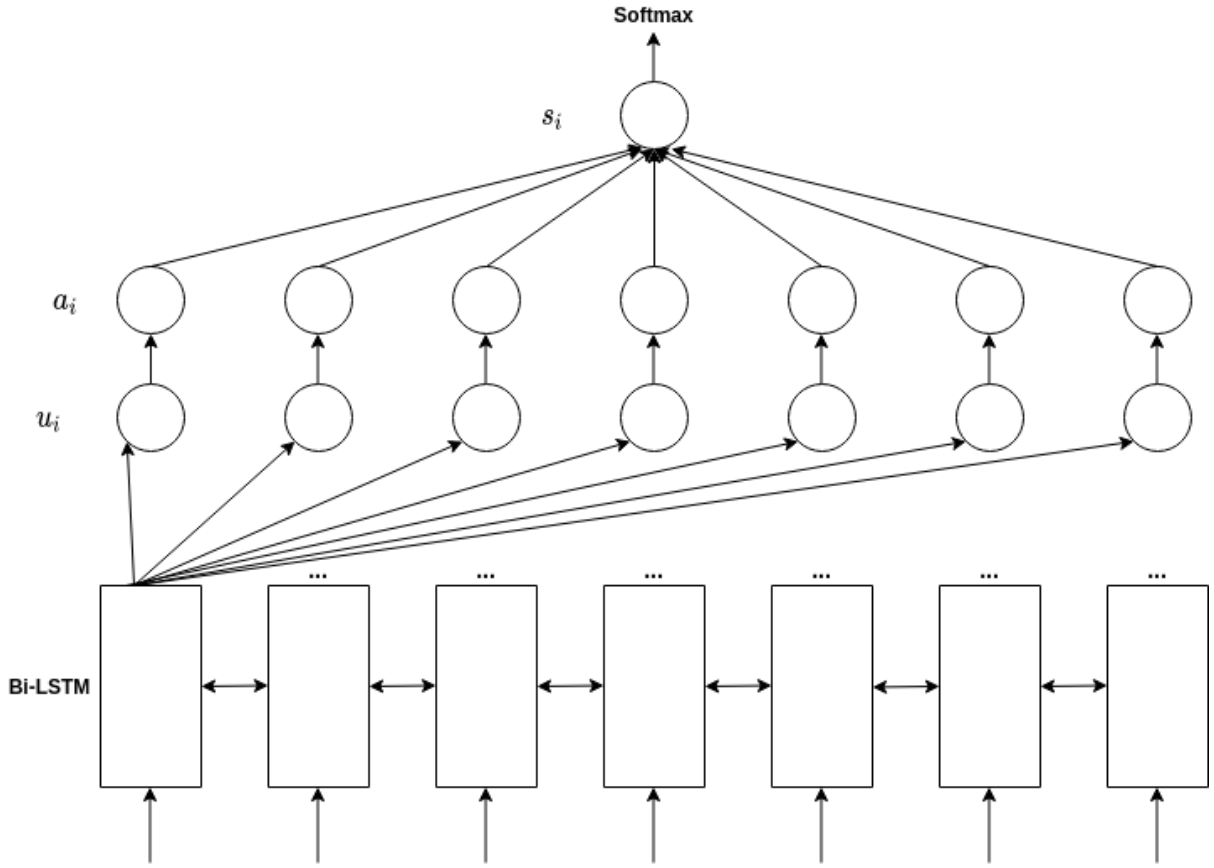
Στο σημείο αυτό, θα δώσουμε το a_{it} σε ένα $Dropout\ Layer$ έτσι ώστε να επιτύχουμε μία στιβαρότερη μοντελοποίηση. Επιπρόσθετα, στην προσπάθεια μας να αυξήσουμε την περιγραφικότητα του μοντέλου θα δώσουμε το a_{it} σαν είσοδο σε ακόμα ένα $Dense\ Layer$ και ένα $Dropout\ Layer$, τα οποία περιλαμβάνουν ακριβώς ότι περιγράφηκε παραπάνω. Παράλληλα, αξίζει να σημειώσουμε ότι τόσο τα $Dropout\ Layers$ όσο και τα $Batch\ Normalization\ Layers$ δεν υπήρχαν στην αρχική δημοσίευση αλλά παρατηρήσαμε ότι πειραματικά βελτιώνουν την αποτελεσματικότητα του μοντέλου.

Τέλος, θα χρειαστούμε ένα ακόμη $Dense\ Layer$ το οποίο θα λειτουργήσει σαν $Classification\ Layer$. Πιο συγκεκριμένα, θα υπολογίσουμε την πιθανότητα που έχει μία είσοδος να ανήκει σε μία συγκεκριμένη αλγοριθμική ετικέτα. Για να το επιτύχουμε αυτό, θα χρησιμοποιήσουμε δύο μεταβλητές τις οποίες θα μαθαίνει το νευρωνικό δίκτυο όσο εκπαιδεύεται. Οι μεταβλητές αυτές είναι ένας πίνακας βαρών W_c και ένα διάνυσμα προκατάληψης b_c

$$prob = \frac{exp(W_c a_{it} + b_c)}{\sum_{classes} exp(W_c a_{it} + b_c)} \tag{5.5}$$

5.2 Αμφίδρομο LSTM με Attention Layer

Στην ενότητα αυτή θα ασχοληθούμε με ένα μοντέλο το οποίο αποτελείται από ένα αμφίδρομο $LSTM$[10] και ένα $Attention\ Layer$[17]. Το μοντέλο αυτό, είναι ένα αρκετά απλό μοντέλο το οποίο βασίζεται στις τελευταίας εξελίξεις στον κλάδο της επεξεργασίας φυσικής γλώσσας. Είναι συχνό να χρησιμοποιούνται $LSTM$ για την κωδικοποίηση μίας διαδοχικής εισόδου καθώς και η χρήση ενός $Attention\ Layer$ μετά την κωδικοποίηση που πραγματοποιείται από το $LSTM$. Πιο συγκεκριμένα, στο δίκτυο αυτό δίνεται σαν είσοδος όλη η ακολουθία λέξεων που συνθέτουν έναν κώδικα. Στην συνέχεια, η κάθε λέξη κωδικοποιείται από ένα $Embedding\ Layer$ και η ακολουθία δίνεται στο $LSTM$. Σε κάθε βήμα του $LSTM$ δίνεται μία έξοδος η οποία αποτελεί την κωδικοποίηση του συγκεκριμένου βήματος σεβόμενη τα γειτονικά βήματα. Αφού ολοκληρωθεί η κωδικοποίηση από το $LSTM$ δίνουμε την έξοδο κάθε χρονικού βήματος στο $Attention\ Layer$ το οποίο προσδίδει διαφορετική βαρύτητα σε κάθε κωδικοποίηση ανάλογα με την σημασία που έχει για την κατανόηση του νοήματος του αλγορίθμου. Τέλος, αθροίζουμε τις διαφορετικές βαρύτητες αυτές και θεωρούμε ότι έχουμε κωδικοποιήσει όλον τον κώδικα σε ένα διάνυσμα. Με χρήση του διανύσματος αυτού ταξινομούμε τον κάθε κώδικα στον αλγόριθμο που αντιστοιχεί. Στην συνέχεια, παραθέτουμε ένα σχήμα (5.2) που εξηγεί την διαδικασία που περιγράψαμε.



Σχήμα 5.2: Bidirectional LSTM with Attention Layer

Στο σημείο αυτό θεωρούμε σκόπιμο να περιγράψουμε τις μαθηματικές σχέσεις που συνδέουν τα επιμέρους κομμάτια του δικτύου αυτού καθώς πιστεύουμε ότι έτσι θα επέλθει η μέγιστη δυνατή κατανόηση του δικτύου από τον αναγνώστη. Όπως είδαμε και παραπάνω, το μοντέλο μας αποτελείται από ένα $Embedding\ Layer$, ένα αμφίδρομο $LSTM$, ένα $Batch\ Normalization\ Layer$[11], ένα $Attention\ Layer$ και ένα $Classification\ Layer$.
Αρχικά, η είσοδος του μοντέλου δίνεται σε ένα $Embedding\ Layer$. Έχοντας λοιπόν έναν πίνακα με $embeddings\ W_e$ και την είσοδο w_{it} θα υπολογίσουμε την έξοδο του $Embedding\ Layer\ x_{it}$:

$$x_{it} = W_e w_{it} \tag{5.6}$$

Στην συνέχεια, θα δώσουμε το x_{it} σαν είσοδο σε ένα αμφίδρομο $LSTM$ έτσι ώστε να το επεξεργαστεί διαδοχικά.

$$\begin{aligned} \overrightarrow{h_{it}} &= \overrightarrow{LSTM}(x_{it}) \\ \overleftarrow{h_{it}} &= \overleftarrow{LSTM}(x_{it}) \\ h_{it} &= [\overrightarrow{h_{it}}, \overleftarrow{h_{it}}] \end{aligned} \tag{5.7}$$

Έπειτα, θα περάσουμε το h_{it} από ένα $Batch\ Normalization\ Layer$ και ένα $Dropout\ Layer$. Η αλήθεια είναι ότι επιλογή αυτή δεν είναι απόλυτα σωστή με βάση την θεωρία καθώς το $Batch\ Normalization$ έχει σχεδιαστεί για να χρησιμοποιείται για $Batches$ και έτσι δεν λαμβάνει υπόψιν την αναδρομικότητα του $LSTM$. Πάραυτα, επιλέξαμε να την χρησιμοποιήσουμε καθώς βοηθάει στην αύξηση της αποτελεσματικότητας του μοντέλου, το οποίο και αποδείξαμε πειραματικά, και η αντίστοιχη μέθοδος που λαμβάνει υπόψιν την αναδρομικότητα είναι αρκετά δύσκολο να υλοποιηθεί.

$$\begin{aligned} h_{it} &= BatchNormalization(h_{it}) \\ h_{it} &= Dropout(h_{it}) \end{aligned} \tag{5.8}$$

Στην συνέχεια θα χρησιμοποιήσουμε το $Attention\ Layer$[17]. Αρχικά, χρειάζεται να υπολογίσουμε τον βαθμό προσοχής της κάθε λέξης (a_{it}). Για να το πετύχουμε αυτό, θα χρησιμοποιήσουμε τρεις εκπαιδευόμενες παραμέτρους. Έναν πίνακα βαρών (W_w), ένα διάνυσμα προκατάληψης (b_w) και ένα διάνυσμα συμφραζομένων (u_w). Έπειτα, θα υπολογίσουμε το σταθμισμένο διάνυσμα αθροισμάτων (s_i) .

$$\begin{aligned} u_{it} &= tanh(W_w h_{it} + b_w) \\ a_{it} &= \frac{exp(u_{it} u_w)}{\sum_t exp(u_{it} u_w)} \\ s_i &= \sum_t a_{it} h_{it} \end{aligned} \tag{5.9}$$

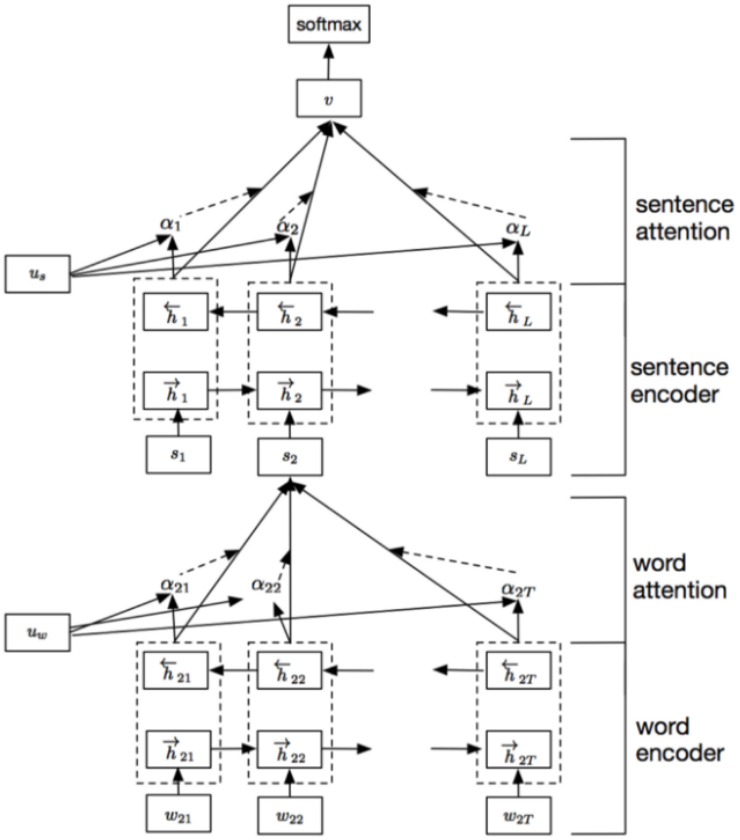
Τέλος, θα χρειαστούμε ένα $Classification\ Layer$.

$$prob = \frac{exp(W_c s_i + b_c)}{\sum_{classes} exp(W_c s_i + b_c)} \tag{5.10}$$

5.3 Hierarchical Attention Network (HAN)

Στην ενότητα αυτή, θα ασχοληθούμε με ένα ιεραρχικό μοντέλο βαθιάς μηχανικής μάθησης[17]. Το μοντέλο αυτό θεωρούμε ότι παρουσιάζει ιδιαίτερο ενδιαφέρον όχι μόνο γιατί είναι το πιο σύνθετο από τα μοντέλα που θα παρουσιαστούν στην παρούσα εργασία αλλά και γιατί είναι ένα από τα καλύτερα μοντέλα στο έργο της ταξινόμησης κειμένου. Ως εκ τούτου, ελπίζουμε ότι θα παρουσιάσει ανάλογα αποτελέσματα και στο έργο της ταξινόμησης πηγαίου κώδικα σε αλγορίθμους. Μάλιστα, αξίζει να σημειώσουμε ότι κατά κάποιον τρόπο το μοντέλο αυτό αποτελεί μία σημαντική βελτίωση του μοντέλου που συζητήσαμε στην προηγούμενη ενότητα καθώς στην ουσία έρχεται να επιλύσει διάφορους περιορισμούς του μοντέλου αυτού.

Το συγκεκριμένο μοντέλο στην ουσία πραγματοποιεί μία πάρα πολύ απλή λειτουργία η οποία όμως είναι και αρκετή για την επίλυση του προβλήματος. Αρχικά, έχοντας μία παράγραφο θεωρούμε ότι η κάθε πρόταση της παραγράφου εσωκλείει διαφορετική νοηματική σημασία. Ως εκ τούτου, αρχικά το μοντέλο κωδικοποιεί το νόημα της κάθε πρότασης. Στην συνέχεια, παίρνει το κωδικοποιημένο νόημα της κάθε πρότασης και κωδικοποιεί το νόημα ολόκληρης της παραγράφου. Ο τρόπος λειτουργίας αυτός είναι μία πιο σοφιστικέ υλοποίηση της λειτουργίας του μοντέλου της προηγούμενης ενότητας. Πιο συγκεκριμένα, το μοντέλο της προηγούμενης ενότητας έβλεπε όλη την παράγραφο και την κωδικοποιούσε ενώ το μοντέλο αυτό αρχικά κωδικοποιεί την κάθε πρόταση και στην συνέχεια από την κωδικοποίηση της κάθε πρότασης κωδικοποιεί την παράγραφο. Το γεγονός αυτό επιτρέπει στο μοντέλο να είναι περισσότερο εκφραστικό καθώς στην περίπτωση του μοντέλου της προηγούμενης ενότητας είναι πιθανόν λέξεις που συνεισφέρουν λιγότερο στο νόημα να χαθούν σε μία πολύ μεγάλη ακολουθία εισόδου. Αντίθετα, στο μοντέλο αυτό μία μεγάλη ακολουθία σπάει σε μικρότερες επιτρέποντας έτσι την διατήρηση μεγαλύτερου όγκου πληροφορίας. Φυσικά, η ιδιότητα του μοντέλου αυτή μπορεί μεν να είναι ευεργετική για την επίλυση του εκάστοτε προβλήματος οδηγεί όμως και στην δραματική αύξηση της υπολογιστικής πολυπλοκότητας που απαιτείται ώστε να ολοκληρωθεί η διαδικασία εκπαίδευσης του μοντέλου. Είναι χαρακτηριστικό πως το μοντέλο αυτό χρειάζεται 2 ή 3 φορές παραπάνω χρόνο από αυτό της προηγούμενης ενότητας για να ολοκληρώσει την διαδικασία εκπαίδευσης (με χρήση του ίδιου συνόλου δεδομένων). Τέλος, αξίζει να σημειώσουμε ότι έχουμε διαφοροποιηθεί κάπως σε σχέση με την υλοποίηση του συγκεκριμένου μοντέλου στην βιβλιογραφία. Πιο συγκεκριμένα, στο μοντέλο αυτό χρησιμοποιείται ένας τύπος RNN ο οποίος ονομάζεται GRU. Ο συγκεκριμένος τύπος RNN, έχει κατασκευαστεί σαν εναλλακτική του $LSTM$ και είναι γνωστός για το ότι χρειάζεται λιγότερη υπολογιστική ισχύ σε σχέση με τα $LSTM$. Βέβαια, είναι επίσης γνωστό ότι τα $LSTM$ υπερτερούν του GRU κατά την διατήρηση πληροφορίας σε μεγάλες ακολουθίες εισόδου καθώς διαθέτουν περισσότερες εκπαιδευόμενες παραμέτρους. Δεδομένου ότι είναι συχνό μία γραμμή κώδικα να αποτελείται από πολλές λέξεις και ένας κώδικας να αποτελείται από πολλές γραμμές αποφασίσαμε να αντικαταστήσουμε τα GRU με $LSTM$ στην προσπάθεια μας να αυξήσουμε την αποτελεσματικότητα του μοντέλου. Στην συνέχεια, παραθέτουμε ένα σχήμα (5.3) που εξηγεί την διαδικασία που περιγράψαμε.



Σχήμα 5.3: Hierarchical Attention Network [17]

Στο σημείο αυτό θεωρούμε σκόπιμο να περιγράψουμε τις μαθηματικές σχέσεις που συνδέουν τα επιμέρους κομμάτια του δικτύου αυτού καθώς πιστεύουμε ότι έτσι θα επέλθει η μέγιστη δυνατή κατανόηση του δικτύου από τον αναγνώστη. Όπως είδαμε και παραπάνω, το μοντέλο μας αποτελείται από ένα $Embedding\ Layer$, ένα αμφίδρομο $LSTM$[10], ένα $Batch\ Normalization\ Layer$, ένα $Attention\ Layer$, άλλο ένα αμφίδρομο $LSTM$, ένα $Batch\ Normalization\ Layer$[11], ένα $Attention\ Layer$ και ένα $Classification\ Layer$.
Αρχικά, θα σχολιάσουμε τον κωδικοποιητή προτάσεων ο οποίος είναι υπεύθυνος για την κωδικοποίηση μίας πρότασης σε ένα διάνυσμα αριθμών. Έχοντας λοιπόν έναν πίνακα με $embeddings$ W_e και την είσοδο w_{it} θα υπολογίσουμε την έξοδο του $Embedding\ Layer$ x_{it} :

$$x_{it} = W_e w_{it} \tag{5.11}$$

Στην συνέχεια, θα δώσουμε το x_{it} σαν είσοδο σε ένα αμφίδρομο $LSTM$.

$$\begin{aligned} \overrightarrow{h_{it}} &= \overrightarrow{LSTM}(x_{it}) \\ \overleftarrow{h_{it}} &= \overleftarrow{LSTM}(x_{it}) \\ h_{it} &= [\overrightarrow{h_{it}}, \overleftarrow{h_{it}}] \end{aligned} \tag{5.12}$$

Στο σημείο αυτό, θα περάσουμε το h_{it} από ένα $Batch\ Normalization\ Layer$ και ένα $Dropout\ Layer$ (οι επιλογές αυτές γίνονται για τους ίδιους λόγους που έγιναν και στο μοντέλο της προηγούμενης ενότητας).

$$\begin{aligned} h_{it} &= BatchNormalization(h_{it}) \\ h_{it} &= Dropout(h_{it}) \end{aligned} \tag{5.13}$$

Στην συνέχεια θα χρησιμοποιήσουμε το $Attention\ Layer$. Αρχικά, χρειάζεται να υπολογίσουμε τον βαθμό προσοχής της κάθε λέξης (a_{it}). Για να το πετύχουμε αυτό, θα χρησιμοποιήσουμε τρεις εκπαιδευόμενες παραμέτρους. Έναν πίνακα βαρών (W_w), ένα διάνυσμα προκατάληψης (b_w) και ένα διάνυσμα συμφραζομένων (u_w). Έπειτα, θα υπολογίσουμε το σταθμισμένο διάνυσμα αθροισμάτων (s_i) .

$$\begin{aligned} u_{it} &= tanh(W_w h_{it} + b_w) \\ a_{it} &= \frac{exp(u_{it} u_w)}{\sum_t exp(u_{it} u_w)} \\ s_i &= \sum_t a_{it} h_{it} \end{aligned} \tag{5.14}$$

Στο σημείο αυτό θα σχολιάσουμε τον κωδικοποιητή του κώδικα ο οποίος παίρνει σαν είσοδο την κωδικοποιημένη ακολουθία προτάσεων που δίνει σαν έξοδο ο κωδικοποιητής προτάσεων για όλες τις προτάσεις του κώδικα και δίνει σαν έξοδο ένα διάνυσμα που κωδικοποιεί το νόημα ολόκληρου του κώδικα.

$$x_{it} = [s_{i1}, s_{i2}, s_{i3}, ..., s_{in}] \tag{5.15}$$

Στην συνέχεια, θα δώσουμε το x_{it} σαν είσοδο σε ένα αμφίδρομο $LSTM$.

$$\begin{aligned} \overrightarrow{h_{it}} &= \overrightarrow{LSTM}(x_{it}) \\ \overleftarrow{h_{it}} &= \overleftarrow{LSTM}(x_{it}) \\ h_{it} &= [\overrightarrow{h_{it}}, \overleftarrow{h_{it}}] \end{aligned} \tag{5.16}$$

Έπειτα, θα δώσουμε το h_{it} σε ένα $Batch\ Normalization\ Layer$ και ένα $Dropout\ Layer$ όπως κάναμε και στον κωδικοποιητή πρότασης. Παράλληλα, η έξοδος του $Dropout\ Layer$ θα δοθεί σε ένα $Attention\ Layer$ το οποίο θα παράξει την κωδικοποίηση ολόκληρου του κώδικα ανάλογα με την σημασία που θα δώσει σε κάθε πρόταση.

$$\begin{aligned} h_{it} &= BatchNormalization(h_{it})) \\ s_i &= Attention(h_{it}) \end{aligned} \tag{5.17}$$

Τέλος, θα χρησιμοποιήσουμε άλλο ένα $Batch\ Normalization\ Layer$ καθώς και ένα $Classification\ Layer$ το οποίο θα ταξινομήσει τον κώδικα, ανάλογα με το κωδικοποιημένο του διάνυσμα, στον αλγόριθμο που υλοποιεί.

$$\begin{aligned} s_i &= BatchNormalization(s_i) \\ prob &= \frac{exp(W_c s_i + b_c)}{\sum_{classes} exp(W_c s_i + b_c)} \end{aligned} \tag{5.18}$$

Κεφάλαιο 6

Πειραματική αξιολόγηση

Στο κεφάλαιο αυτό επιθυμούμε να αξιολογήσουμε πειραματικά τόσο τα διαφορετικά σενάρια προ-επεξεργασίας που προτείναμε σε προηγούμενο κεφάλαιο όσο και τα διαφορετικά μοντέλα που περιγράψαμε προηγουμένως. Μάλιστα, αξίζει να σημειώσουμε ότι η συγκεκριμένη αξιολόγηση επιθυμούμε να συμβεί με ένα τρόπο ο οποίος δεν είναι δέσμιος στατιστικών λαθών. Για τον σκοπό αυτό, σε πρώτη φάση θα χωρίσουμε τα σύνολα δεδομένων σε σύνολα εκπαίδευσης και σύνολα αξιολόγησης. Ο διαχωρισμός αυτός θα μείνει σταθερός καθ' όλη την διάρκεια της αξιολόγησης έτσι ώστε να είμαστε σίγουροι ότι τα αποτελέσματα που εξάγουμε είναι συγκρίσιμα. Στην συνέχεια, για να σιγουρευτούμε ότι τα αποτελέσματα που παίρνουμε δεν αποτελούν παράδειγμα στατιστικού λάθους θα αξιολογήσουμε την κάθε μέθοδο σε κάθε μοντέλο με την χρήση της τεχνικής επαλήθευσης 5 πτυχών. Παράλληλα, στην προσπάθεια μας να δημιουργήσουμε ένα ενιαίο πρωτόκολλο αξιολόγησης θα εκπαιδεύσουμε ένα $Word\ Embedding$ μοντέλο για κάθε σενάριο προ-επεξεργασίας και θα χρησιμοποιήσουμε αυτό σε όλα τα πειράματα που θα πραγματοποιήσουμε. Τέλος, αξίζει να σημειώσουμε ότι η διαδικασία της αξιολόγησης θα πραγματοποιηθεί και θα παρουσιαστεί ξεχωριστά για τα σύνολα δεδομένων των βρόγχων και συναρτήσεων. Ο λόγος για τον οποίο πήραμε την απόφαση αυτή είναι γιατί το κάθε σύνολο δεδομένων αντικατοπτρίζει ένα διαφορετικό πρόβλημα και έτσι δεν θεωρούμε σωστό να αξιολογήσουμε τα σύνολα αυτά με έναν ενιαίο τρόπο.

Στο σημείο αυτό θεωρούμε σκόπιμο να περιγράψουμε το υπολογιστικό υλικό που χρησιμοποιήσαμε για να πραγματοποιήσουμε τα πειράματα αυτά. Αρχικά, στα πειράματα τα οποία χρησιμοποιήσαμε επεξεργαστή χρησιμοποιήσαμε έναν $Intel\ Core\ I5 - 6600\ 3.3GHz$. Ο επεξεργαστής αυτός διαθέτει 4 πυρήνες και 4 νήματα και είναι ένας απλός "οικιακός" επεξεργαστής γεγονός που συνεπάγεται ότι τα πειράματα αυτά μπορούν να γίνουν εύκολα από τον οποιονδήποτε. Όσο αναφορά τα πειράματα στα οποία θεωρήθηκε σκόπιμη η χρήση κάρτας γραφικών, χρησιμοποιήσαμε μία $Nvidia\ 1080Ti$ η οποία διαθέτει 12 Gb μνήμης. Η συγκεκριμένη κάρτα γραφικών δεν θεωρείται σε καμία περίπτωση μία "απλή" κάρτα γραφικών όμως και πάλι θα μπορούσε να υλοποιήσει κανείς τα πειράματα αυτά χρησιμοποιώντας κάποια δωρεάν $Cloud$ υπηρεσία.

6.1 Δημιουργία συνόλου αξιολόγησης

Στην ενότητα αυτή, θα ασχοληθούμε με την δημιουργία του συνόλου αξιολόγησης για κάθε ένα από τα σύνολα δεδομένων που δημιουργήσαμε. Η διαδικασία αυτή θεωρείται ιδιαίτερα σημαντική καθώς το σύνολο δεδομένων αυτό θα μας επιτρέψει να πραγματοποιήσουμε μία ποιοτική αξιολόγηση των διαφορετικών μοντέλων και μεθόδων προ-επεξεργασίας. Για την δημιουργία του συνόλου αυτού, πήραμε τα σύνολα δεδομένων που δημιουργήσαμε και αφού τα ανακατέψαμε θεωρήσαμε με τυχαίο τρόπο ότι το 70% αυτών αποτελεί το σύνολο εκπαίδευσης και το υπόλοιπο 30% αποτελεί το σύνολο αξιολόγησης. Στην συνέχεια, παραθέτουμε δύο πίνακες στους οποίους παρουσιάζονται τα σύνολα εκπαίδευσης και αξιολόγησης τόσο για το σύνολο δεδομένων με βρόγχους όσο και για το σύνολο δεδομένων με συναρτήσεις.

Αλγόριθμος	Train set	Test set
dot	807	378
swap	1229	519
axpy	468	175
scal	634	294
fft	689	323
gemm	1459	607
floyd warshall	882	367
bubble sort	2237	991
insertion sort	1742	716
selection sort	1549	1643

Πίνακας 6.1: Αναλυτική παρουσίαση συνόλου δεδομένων με βρόγχους

Αλγόριθμος	Train set	Test set
bfs	5282	2184
dfs	7336	3068
reverse list	4001	1767
heap sort	4803	2040
radix sort	1995	861
quicksort	8197	3645
mergesort	8910	3914
union find	2184	933
bubble sort	3491	1486
insert in list	1366	638
insertion sort	2932	1138
selection sort	2600	1049
binary search	4883	2123
height of tree	2151	925

Πίνακας 6.2: Αναλυτική παρουσίαση συνόλου δεδομένων με συναρτήσεις

6.2 Δημιουργία embeddings με τον αλγόριθμο Word2Vec

Στην ενότητα αυτή θα σχολιάσουμε την δημιουργία $embeddings$ για τα διάφορα σύνολα δεδομένων που δημιουργήσαμε σε προηγούμενο κεφάλαιο. Μάλιστα, δεδομένου ότι έχουμε δημιουργήσει διαφορετικές εκδοχές του κάθε κώδικα, ανάλογα με την τεχνική προ-επεξεργασίας που του έχει επιβληθεί, θεωρήσαμε επιτακτική την ανάγκη να δημιουργήσουμε μία διαφορετική εκδοχή των $embeddings$ αυτών ανάλογα με την εκδοχή προ-επεξεργασίας που χρησιμοποιήσαμε. Επιπρόσθετα, στο σημείο αυτό θεωρούμε σκόπιμο να υπενθυμίσουμε ότι τα $embeddings$ αυτά θα δημιουργηθούν με τη χρήση του αλγορίθμου $Word2Vec$[21, 22]. Μάλιστα, θα θέλαμε να τονίσουμε ότι αξιολογήσαμε την χρήση και άλλων αντίστοιχων αλγορίθμων αλλά θεωρήσαμε ότι η χρήση τους δεν ενδείκνυται στο πρόβλημα μας. Για παράδειγμα, αξιολογήσαμε την χρήση του αλγορίθμου $FastText$[49, 59], ο οποίος δημιουργεί $embeddings$ λαμβάνοντας υπόψιν και τα συνθετικά από τα οποία αποτελούνται οι σύνθετες λέξεις, και θεωρήσαμε ότι η χρήση του αλγορίθμου αυτού δεν θα βοηθήσει ιδιαίτερα καθώς η συγκεκριμένη ιδιότητα έχει διαχειριστεί προσεκτικά από τις διαφορετικές εκδοχές προ-επεξεργασίας που έχουμε προτείνει. Πιο συγκεκριμένα, σε μία από τις εκδοχές που έχουμε προτείνει, και θα εξετάσουμε πειραματικά στην συνέχεια, αποσυνθέτουμε τα ονόματα μεταβλητών και συναρτήσεων στις απλούστερες λέξεις που τις αποτελούν λαμβάνοντας μάλιστα υπόψιν γνωστές καλές πρακτικές συγγραφής κώδικα, γεγονός που μας κάνει να προτιμούμε την προ-επεξεργασία αυτή από την εφαρμογή μίας μεθόδου η οποία δεν σέβεται αρκετές από τις ιδιότητες του κώδικα και έχει σχεδιαστεί για φυσική γλώσσα. Στην συνέχεια, θα σχολιάσουμε ξεχωριστά την δημιουργία των $embeddings$ αυτών για το σύνολο δεδομένων που αποτελείται από βρόγχους και το σύνολο που αποτελείται από συναρτήσεις.

6.2.1 Embeddings για το σύνολο δεδομένων με βρόγχους

Στην υποενότητα αυτή θα σχολιάσουμε τα $embeddings$ που δημιουργήσαμε για το σύνολο δεδομένων με βρόγχους. Αρχικά, θα θέλαμε να σημειώσουμε ότι για το σύνολο δεδομένων αυτό αποφασίσαμε να ασχοληθούμε μόνο με την εκδοχή προ-επεξεργασίας στην οποία γίνεται κανονικοποιήση των ονομάτων και χρωματισμός βάθους του κώδικα. Η επιλογή αυτή έγινε καθώς θεωρήσαμε ότι η συγκεκριμένη προ-επεξεργασία ταιριάζει ιδιαίτερα στον τύπο του προβλήματος καθώς σε μικρούς βρόγχους παρατηρείται και μικρό πλήθος μεταβλητών. Στην συνέχεια, θεωρούμε σκόπιμο να παραθέσουμε κάποιες τεχνικές επιλογές σχετικά με τις παραμέτρους του αλγορίθμου. Σε πρώτη φάση αξίζει να σημειώσουμε ότι εκπαιδεύσαμε τα $embeddings$ αυτά χωρίς να χρησιμοποιήσουμε κάποια πρότερη γνώση, καθώς στην παρούσα εργασία έχουμε δημιουργήσει δικές μας τεχνικές προ-επεξεργασίας και έτσι η συγκεκριμένη τεχνική δεν θεωρείται ενδεδειγμένη. Παράλληλα, αποφασίσαμε να δημιουργήσουμε $embeddings$ 100 διαστάσεων για κάθε λέξη. Η επιλογή αυτή, έγινε καθώς γνωρίζουμε ότι το σύνολο δεδομένων αυτό αποτελείται από μικρό αριθμό μοναδικών λέξεων και παράλληλα έχουμε στην διάθεση μας περιορισμένο πλήθος υπολογιστικών πόρων. Ως εκ τούτου, θεωρήσαμε ότι η επιλογή αυτή αποτελεί τον καλύτερο δυνατό συνδυασμό ανάμεσα στην επίδοση και την απόδοση των $embeddings$. Αρχικά, αποφασίσαμε ότι οι λέξεις οι οποίες εμφανίζονται λιγότερο από 20 φορές σε όλο το σύνολο δεδομένων μπορούν να αγνοηθούν καθώς κατά πάσα πιθανότητα αποτελούν λέξεις οι οποίες δεν προσφέρουν ουσιαστική βοήθεια στην κατανόηση του νοήματος του κάθε κώδικα. Παράλληλα, επιλέξαμε το μήκος του παραθύρου να ισούται με 8 καθώς σύμφωνα με

την αρχική δημοσίευση ο αριθμός αυτός συνδυάζει τόσο την νοηματική όσο και την συντακτική σημασία του $embedding$ της κάθε λέξης. Επιπρόσθετα, θέσαμε ως τιμή της παραμέτρου $negative\ sampling$ την τιμή 20 καθώς γνωρίζουμε ότι το σύνολο δεδομένων μας είναι σχετικά μικρό και έτσι θα χρειαστούν περισσότερες ενημερώσεις βαρών ανά είσοδο. Τέλος, αξίζει να σημειώσουμε ότι εκπαιδεύσαμε τον αλγόριθμο αυτό για 30 εποχές καθώς γνωρίζουμε από την βιβλιογραφία ότι χρειάζονται αρκετές εποχές για την εκπαίδευση του. Στην συνέχεια παραθέτουμε έναν πίνακα με τα αποτελέσματα της εκτέλεσης του αλγορίθμου.

Τεχνική προ-επεξεργασίας	Πλήθος μοναδικών λέξεων	Χρόνος εκπαίδευσης
Κανονικοποίηση + χρωματισμός	300	1.17 (min)

Πίνακας 6.3: Αποτελέσματα εκτέλεσης αλγορίθμου Word2Vec στο σύνολο δεδομένων με βρόγχους

Παρατηρούμε, ότι το πλήθος των μοναδικών λέξεων είναι πολύ μικρό, γεγονός που εξηγεί και τον χαμηλό χρόνο εκπαίδευσης που απαιτήθηκε για την εκπαίδευση του αλγορίθμου. Γενικότερα, το μικρό πλήθος μοναδικών λέξεων χαρακτηρίζεται ως αναμενόμενο καθώς το σύνολο δεδομένων αυτό είναι σχετικά μικρό και η έκδοση που εξετάζουμε έχει κανονικοποιημένα ονόματα μεταβλητών και συναρτήσεων, από τα οποία προέρχεται και ο κύριος όγκος μοναδικών λέξεων. Επιπρόσθετα, αξίζει να υπενθυμίσουμε ότι έχουμε "αγνοήσει" τις λέξεις εκείνες που εμφανίζονται λιγότερο από 20 φορές σε όλο το σύνολο δεδομένων γεγονός που συνεισφέρει ιδιαίτερα στην συρρίκνωση αυτή.

6.2.2 Embeddings για το σύνολο δεδομένων με συναρτήσεις

Στην υποενότητα αυτή θα σχολιάσουμε τα $embeddings$ που δημιουργήσαμε για το σύνολο δεδομένων με συναρτήσεις. Αρχικά, θα θέλαμε να σημειώσουμε ότι δημιουργήσαμε 4 διαφορετικές εκδοχές από $embeddings$ όσες δηλαδή και οι διαφορετικές εκδοχές προ-επεξεργασίας που προτείναμε σε προηγούμενο κεφάλαιο. Στην συνέχεια, θεωρούμε σκόπιμο να αναφέρουμε ότι όλες οι τεχνικές επιλογές σχετικά με τις παραμέτρους του αλγορίθμου $Word2Vec$ ταυτίζονται με τις επιλογές που έγιναν στην υποενότητα που αφορά το σύνολο δεδομένων με βρόγχους. Τέλος, παραθέτουμε έναν πίνακα με τα αποτελέσματα της εκτέλεσης του αλγορίθμου αυτού σε κάθε διαφορετική εκδοχή προ-επεξεργασίας.

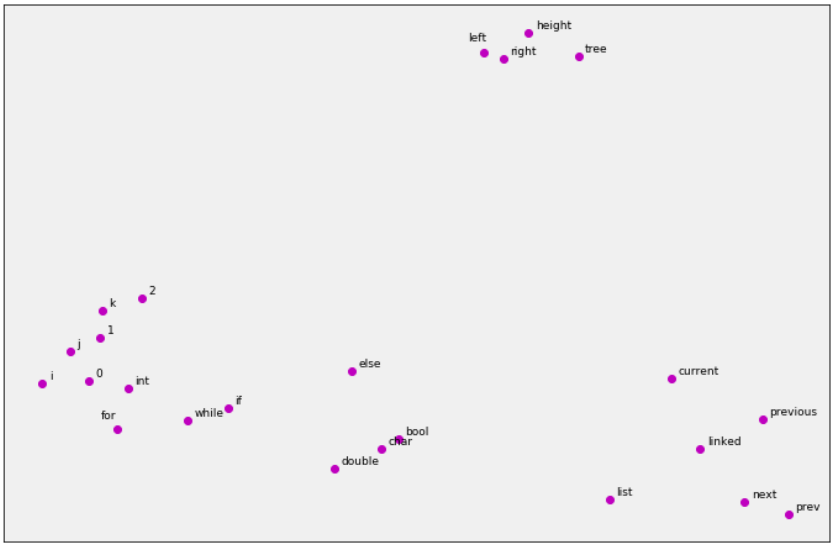
Τεχνική προ-επεξεργασίας	Πλήθος μοναδικών λέξεων	Χρόνος εκπαίδευσης
Αποσύνθεση σύνθετων λέξεων	6068	15.93 (min)
Αποσύνθεση σύνθετων λέξεων + χρωματισμός	6079	18.09 (min)
Κανονικοποίηση ονομάτων	342	11.76 (min)
Κανονικοποίηση ονομάτων + χρωματισμός	359	14.04 (min)

Πίνακας 6.4: Αποτελέσματα εκτέλεσης αλγορίθμου Word2Vec στο σύνολο δεδομένων με συναρτήσεις

Ο παραπάνω πίνακας παρουσιάζει ιδιαίτερο ενδιαφέρον για τρεις λόγους. Αρχικά, μας υπενθυμίζει την χαοτική διαφορά που παρατηρείται στο πλήθος μοναδικών λέξεων ανάμεσα στην κανονικοποιημένη εκδοχή και αυτή στην οποία απλά έχουν σπάσει οι σύνθετες λέξεις. Επιπρόσθετα, παρατηρούμαι εύκολα την μεταβολή λέξεων που προκαλεί ο χρωματισμός του κώδικα. Για παράδειγμα, μεταξύ της πρώτης και της δεύτερης γραμμής βλέπουμε ότι έχουν προστεθεί 9 μοναδικές

λέξεις, οι οποίες προφανώς αποτελούν διαφορετικά χρώματα. Τέλος, στην προσπάθεια μας να αξιολογήσουμε την επίδραση της επιλογής του να αγνοήσουμε τις λέξεις που εμφανίζονται λιγότερο από 20 φορές αξίζει να ανατρέξουμε στον Πίνακα 4.3 στον οποίο παρατηρούμε ότι το πλήθος των μοναδικών λέξεων που αντιστοιχεί στην πρώτη γραμμή του Πίνακα 6.4 είναι 55441, γεγονός που συνεπάγεται 90% μείωση του πλήθους των συνολικών λέξεων.

Στην συνέχεια, αποφασίσαμε να οπτικοποιήσουμε ένα μέρος των $embeddings$ που δημιουργήσαμε για το σύνολο δεδομένων αυτό έτσι ώστε να μπορέσουμε να έχουμε μία εικόνα σχετικά με το πόσο καλά λειτούργησε ο αλγόριθμος αυτός. Για να οπτικοποιήσουμε όμως τα $embeddings$ αυτά χρειάζεται να μειώσουμε τις διαστάσεις τους από 100 σε 2. Για να επιτύχουμε την μείωση αυτή θα χρησιμοποιήσουμε τον αλγόριθμο μείωσης διαστατικότητας PCA[50]. Στην συνέχεια, παραθέτουμε μία εικόνα των $embeddings$ που δημιουργήσαμε.



Σχήμα 6.1: Παράδειγμα οπτικοποίησης word embeddings

Αρχικά, παρατηρούμε ότι έχουν δημιουργηθεί κάποιες πολύ ενδιαφέρουσες περιοχές καθώς λέξεις που συνδέονται με την δενδρική δομή δεδομένων βρίσκονται αρκετά κοντά στον χώρο, ενώ κάτι ανάλογο συμβαίνει και για τις συνδεδεμένες λίστες. Επιπρόσθετα, παρατηρούμε ότι οι μεταβλητές i,j και k, οι οποίες συχνά χρησιμοποιούνται ως μετρητές σε επαναληπτικούς βρόγχους, βρίσκονται επίσης κοντά στον χώρο. Παράλληλα, παρατηρούμε ότι οι τύποι μεταβλητών $double$,$char$ και $bool$ βρίσκονται κοντά μεταξύ τους σε αντίθεση όμως με τον τύπο int ο οποίος φαίνεται να βρίσκεται πιο κοντά στην δεσμευμένη λέξη for. Γενικότερα, από την παραπάνω εικόνα παίρνουμε μία αρκετά καλή ιδέα της δυναμικής του αλγορίθμου $Word2Vec$ καθώς και του λόγου που έχει εδραιωθεί σε προβλήματα επεξεργασίας φυσικής γλώσσας. Τέλος, πριν προχωρήσουμε παρακάτω θα θέλαμε να σχολιάσουμε ότι το παραπάνω σχήμα είναι μεν χρήσιμο αλλά όχι πλήρως αξιόπιστο καθώς έχουμε μειώσει τις διαστάσεις του χώρου από 100 σε 2 γεγονός που απευθείας συνεπάγεται μία σημαντική μείωση της διαθέσιμης πληροφορίας. Επιπρόσθετα, αξίζει να σημειώσουμε ότι ο αλγόριθμος αυτός δεν έχει σχεδιαστεί για να χρησιμοποιείται σε πηγαίο κώδικα γεγονός που εξηγεί και κάποιους περιορισμούς που εμφανίζονται στην αποδοτικότητα του. Για τον λόγο αυτό στην βιβλιογραφία εξετάζονται σύνθετες εξειδικευμένες αρχιτεκτονικές[47, 48] οι οποίες όμως δεν θα αξιολογηθούν στην παρούσα εργασία καθώς η χρησιμοποίηση τους χαρακτηρίζεται ως πολύπλοκη.

6.3 Πειραματική αξιολόγηση συνόλου δεδομένων με βρόγχους

Στην ενότητα, αυτή θα αξιολογήσουμε πειραματικά την ποιότητα με την οποία μπορούμε να ταξινομήσουμε τον πηγαίο κώδικα σε αλγορίθμους στο σύνολο δεδομένων με βρόγχους. Για τον σκοπό αυτό, αρχικά είναι δεδομένο πως θα χρησιμοποιήσουμε τα $embeddings$ που δημιουργήσαμε στην προηγούμενη ενότητα. Όπως αναφέραμε προηγουμένως, δημιουργήσαμε $embeddings$ μόνο για την εκδοχή προ-επεξεργασίας η οποία περιλαμβάνει κανονικοποιήση των ονομάτων μεταβλητών και συναρτήσεων καθώς και χρωματισμό βάθους κώδικα. Η επιλογή αυτή έγινε καθώς πιστεύουμε ότι η συγκεκριμένη εκδοχή προ-επεξεργασίας ταιριάζει ιδιαίτερα στο πρόβλημα αυτό μίας και πολλοί από τους αλγορίθμους που περιλαμβάνονται στο σύνολο δεδομένων αυτό αποτελούνται κατά κύριο λόγο από κανονικοποιημένα ονόματα. Επιπρόσθετα, επιλέξαμε σε πρώτη φάση να εκπαιδεύσουμε μόνο το μοντέλο το οποίο περιλαμβάνει ένα αμφίδρομο $LSTM$ και ένα $Attention\ Layer$[17] καθώς το συγκεκριμένο μοντέλο φαίνεται να επαρκεί για την επίλυση του προβλήματος. Ο λόγος που το πιστεύουμε αυτό είναι γιατί το μήκος της ακολουθίας είναι σχετικά μικρό για τους περισσότερους κώδικες του συνόλου αυτού και έτσι το μοντέλο που προτείνουμε μοιάζει αρκετά εκφραστικό για να λύσει το πρόβλημα.

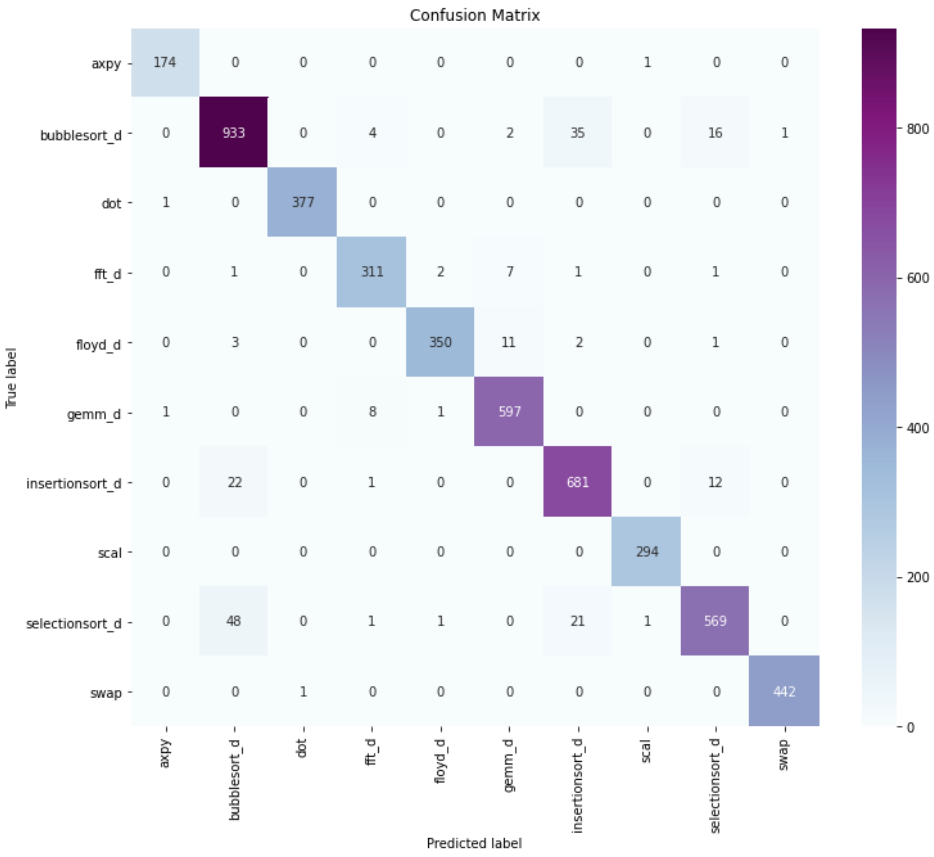
6.3.1 Αξιολόγηση αμφίδρομου LSTM με Attention Layer

Όπως είπαμε και παραπάνω στην ενότητα αυτή θα ασχοληθούμε με την αξιολόγηση του μοντέλου μας το οποίο είναι ένα αμφίδρομο $LSTM$ με $Attention\ Layer$. Πριν όμως προχωρήσουμε στην παράθεση των αποτελεσμάτων θα θέλαμε να συζητήσουμε κάποιες τεχνικές λεπτομέρειες του μοντέλου αυτού κυρίως για λόγους πληρότητας. Αρχικά, το $embedding\ Layer$ που διαθέτει το μοντέλο αυτό είναι παγωμένο, δηλαδή τα $embeddings$ δεν αλλάζουν κατά την διάρκεια της εκπαίδευσης και παραμένουν ακριβώς όπως προέκυψαν από τον αλγόριθμο $Word2Vec$. Στην συνέχεια, θα θέλαμε να σχολιάσουμε ότι το $hidden\ state$ του $LSTM$ έχει μέγεθος 128 και το $Dropout$ που εφαρμόζουμε στις εξόδους του $LSTM$ ισούται με 0.4. Επιπρόσθετα, χρησιμοποιούμε τον $Adam$ σαν $optimizer$ και το $loss$ με το οποίο εκπαιδεύουμε είναι το $categorical\ cross\ entropy$. Στην συνέχεια, παραθέτουμε ένα πίνακα με τα αποτελέσματα της αξιολόγησης του μοντέλου αυτού στα δεδομένα του συνόλου αξιολόγησης.

Τεχνική προ-επεξεργασίας	Precision	Recall	F1	MCC
Κανονικοποίηση + χρωματισμός	0.96	0.96	0.96	0.94

Πίνακας 6.5: Αξιολόγηση μοντέλου BiLSTM + Attention στο σύνολο δεδομένων με βρόγχους

Παρατηρούμε ότι το μοντέλο αυτό καθώς και η επιλεγμένη προ-επεξεργασία λειτουργούν ιδιαίτερα αποτελεσματικά πάνω στο συγκεκριμένο σύνολο δεδομένων γεγονός που στην ουσία επιβεβαιώνει τις αρχικές μας υποθέσεις. Μάλιστα, αξίζει να σημειώσουμε ότι για την εξαγωγή των παραπάνω τιμών έχουμε εφαρμόσει την τεχνική της επαλήθευσης 5 πτυχών έτσι ώστε να εξασφαλίσουμε ότι τα αποτελέσματα που προκύπτουν δεν είναι δέσμια στατιστικού σφάλματος. Στην συνέχεια, παραθέτουμε και έναν πίνακα σύγχυσης έτσι ώστε να κατανοήσουμε καλύτερα τους αλγορίθμους εκείνους που ταξινομούνται σε λάθος κλάση.



Σχήμα 6.2: Confusion matrix για το μοντέλο BiLSTM + Attention στο σύνολο δεδομένων με βρόγχους

Από το παραπάνω σχήμα εύκολα καταλαβαίνει κανείς ότι η πλειοψηφία των λανθασμένων προβλέψεων προέρχεται από τους τρεις αλγορίθμους ταξινόμησης (insertionsort, selectionsort και bubblesort). Αυτό είναι κατά κάποιο τρόπο αναμενόμενο καθώς η υλοποίηση των αλγορίθμων αυτών μοιάζει σε αρκετές περιπτώσεις, ειδικά από την στιγμή που έχουμε κανονικοποιήση τα ονόματα των συναρτήσεων από τις οποίες θα μπορούσε κανείς να διαχωρίσει εύκολα τους αλγορίθμους αυτούς. Αν εξαιρέσουμε όμως τους αλγορίθμους αυτούς, παρατηρούμε ότι η ταξινόμηση των υπολοίπων πραγματοποιείται με μεγάλη επιτυχία. Το γεγονός αυτό, από την μία αξιολογείται ως θετικό καθώς φαίνεται ότι "πετύχαμε" τον στόχο μας, από την άλλη όμως είναι κάπως ανησυχητικό ειδικά αν συνδυαστεί με την παρατήρηση που έχουμε κάνει σε προηγούμενο κεφάλαιο ότι τα δεδομένα του συνόλου αυτού είναι πολύ πιθανόν να μοιάζουν μεταξύ τους. Για να αξιολογήσουμε το γεγονός αυτό, αποφασίσαμε να πραγματοποιήσουμε ένα επιπλέον πείραμα. Πιο συγκεκριμένα, θα αξιολογήσουμε τον αλγόριθμο kNN στο συγκεκριμένο σύνολο δεδομένων. Ο αλγόριθμος αυτός, θα λάβει σαν είσοδο, για κάθε κώδικα, ένα διάνυσμα το οποίο αποτελεί τον μέσο όρο των $embeddings$ των λέξεων που συνθέτουν τον κώδικα αυτόν. Ο μέσος όρος αυτός θεωρούμε ότι κατά κάποιον τρόπο περιγράφει το νόημα του κάθε κώδικα. Τέλος, πριν προχωρήσουμε στα αποτελέσματα, θα θέλαμε να τονίσουμε ότι το πείραμα είναι ιδιαίτερα σημαντικό καθώς ο συγκεκριμένος αλγόριθμος δεν διαθέτει μνήμη και έτσι αποτελεί μία καλή ένδειξη σχετικά με το πόσο κοντά βρίσκονται τα δεδομένα του συνόλου αυτού. Στην συνέχεια, παραθέτουμε τα πειραματικά αποτελέσματα που προέκυψαν από την εκτέλεση του αλγορίθμου, θέτοντας ως τιμή της παραμέτρου k την τιμή 5.

Τεχνική προ-επεξεργασίας	Precision	Recall	F1	MCC
Κανονικοποίηση + χρωματισμός	0.87	0.87	0.87	0.85

Πίνακας 6.6: Αξιολόγηση μοντέλου kNN με k = 5 στο σύνολο δεδομένων με βρόγχους

Τα παραπάνω αποτελέσματα, στην ουσία επιβεβαιώνουν κάποια από τα αρχικά συμπεράσματα που έχουμε εξάγει για το συγκεκριμένο σύνολο δεδομένων καθώς οι υψηλές τιμές των μετρικών υποδεικνύουν ότι ένα μεγάλο πλήθος δεδομένων βρίσκεται αρκετά κοντά και ένας απλός αλγόριθμος χωρίς μνήμη μπορεί εύκολα και αποδοτικά να ταξινομήσει τους αλγορίθμους. Επιπρόσθετα, αξίζει να σημειώσουμε ότι τα παραπάνω πολύ θετικά αποτελέσματα του αλγορίθμου kNN οφείλονται και ως ένα σημείο στα $embeddings$ που δημιούργησε ο αλγόριθμος $Word2Vec$. Τέλος, υπενθυμίζουμε ότι το γεγονός ότι οι βρόγχοι που υλοποιούν έναν αλγόριθμο μοιάζουν μεταξύ τους είναι φυσιολογικό καθώς είναι τέτοιος ο τύπος του συγκεκριμένου προβλήματος που δεν επιτρέπει μεγάλες αλλαγές μεταξύ δύο διαφορετικών βρόγχων.

6.4 Πειραματική αξιολόγηση συνόλου δεδομένων με συναρτήσεις

Στην ενότητα, αυτή θα αξιολογήσουμε πειραματικά την ποιότητα με την οποία μπορούμε να ταξινομήσουμε τον πηγαίο κώδικα σε αλγορίθμους στο σύνολο δεδομένων με συναρτήσεις. Για τον σκοπό αυτό, αρχικά είναι δεδομένο πως θα χρησιμοποιήσουμε τα $embeddings$ που δημιουργήσαμε στην προηγούμενη ενότητα. Όπως αναφέραμε προηγουμένως, δημιουργήσαμε $embeddings$ για 4 διαφορετικές εκδοχές προ-επεξεργασίας οι οποίες διαφοροποιούνται τόσο ως προς την ονοματοδοσία των μεταβλητών και συναρτήσεων όσο και ως προς τον χρωματισμό του βάθους του κώδικα. Ως εκ τούτου, στην ενότητα αυτή επιθυμούμε να αξιολογήσουμε τόσο την απόδοση των διαφορετικών μοντέλων μηχανικής μάθησης όσο και τις διαφορετικές τεχνικές προ-επεξεργασίας που προτείναμε στην παρούσα εργασία. Στην συνέχεια, ακολουθεί εκτενής σχολιασμός των πειραμάτων που πραγματοποιήσαμε.

6.4.1 Η επίδραση των ονομάτων μεταβλητών και συναρτήσεων στην αποτελεσματικότητα του μοντέλου

Στην υποενότητα αυτή, θεωρούμε σκόπιμο να σχολιάσουμε μία πολύ σημαντική παρατήρηση που κάναμε κατά τις πρώτες φάσεις των πειραμάτων μας. Πιο συγκεκριμένα, στα πρώιμα στάδια των πειραμάτων μας δοκιμάσαμε να ταξινομήσουμε το συγκεκριμένο σύνολο δεδομένων με ένα πολύ απλό μοντέλο το οποίο αποτελούνταν από ένα $Embedding\ Layer$, ένα $LSTM$ και ένα $Output\ Layer$. Φυσικά, το μοντέλο αυτό είναι αρκετά απλό και μαστίζεται από διάφορους γνωστούς περιορισμούς όπως το γεγονός ότι το $LSTM$ εξαρτάται κυρίως από τις τελευταίες λέξεις της ακολουθίας εισόδου και ένα μεγάλος μέρος αυτής που βρίσκεται στα πρώτα βήματα συνήθως χάνεται. Ως εκ τούτου, θα περίμενε κανείς ότι το συγκεκριμένο μοντέλο θα πετύχαινε πολύ χαμηλές τιμές σε όλες τις μετρικές αναφοράς. Προς έκπληξη μας όμως το μοντέλο παρουσίασε "φανταστικά" αποτελέσματα. Πιο συγκεκριμένα, η ακρίβεια πρόβλεψης του μοντέλου ήταν ίση με 99%, γεγονός που σημαίνει ότι 99 στις 100 προβλέψεις ήταν σωστές. Η συμπεριφορά αυτή χαρακτηρίστηκε ως ιδιαίτερα ανησυχητική και μη αναμενόμενη καθώς κάτι τέτοιο θα μπορούσε να σημαίνει ότι το σύνολο δεδομένων που

δημιουργήσαμε είναι πολύ εύκολο για να επιλυθεί. Αφού προβληματιστήκαμε αρκετά έτσι ώστε να καταλάβουμε τον λόγο για τον οποίο παρατηρείται η συμπεριφορά αυτή αποφασίσαμε να διεξάγουμε ένα επιπλέον πείραμα για την καλύτερη κατανόηση του φαινομένου. Πιο συγκεκριμένα, δεδομένου ότι για την δημιουργία του συνόλου αυτού αναζητήσαμε συγκεκριμένες λέξεις κλειδιά, όπως είναι το όνομα της συνάρτησης, θεωρήσαμε ότι το γεγονός αυτό θα μπορούσε να καθιστά το σύνολο δεδομένων μας "προκατειλημμένο". Στην προσπάθεια μας να διερευνήσουμε τον ισχυρισμό αυτό, αρχικά εκπαιδεύσαμε ένα μοντέλο με το αρχικό σύνολο δεδομένων και στην συνέχεια κατά την διάρκεια της αξιολόγησης του μοντέλου, στο σύνολο αξιολόγησης, αποκρύψαμε λέξεις οι οποίες θεωρήσαμε ότι εμφανίζονται συχνά σε μία αλγοριθμική ετικέτα (πχ για τον αλγόριθμο ταξινόμησης φυσαλίδας αποκρύψαμε τις λέξεις με όνομα $bubble$) (η μορφή του κώδικα αυτού είναι παρόμοια με αυτή του κώδικα του σχήματος 6.4).

Μετά την πραγματοποίηση του πειράματος παρατηρήσαμε ότι η τιμή της μετρικής αναφοράς έπεσε στην τιμή 67% γεγονός που επιβεβαιώνει τον ισχυρισμό μας. Στην ουσία, από το πείραμα αυτό καταλάβαμε ότι το μοντέλο μας κατέληγε να είναι ένα "χαζό" μοντέλο το οποίο στην ουσία για να ταξινομήσει έναν κώδικα σε μία συγκεκριμένη ετικέτα έψαχνε για συγκεκριμένες λέξεις κλειδιά. Αν για παράδειγμα, σε έναν κώδικα δώσουμε το όνομα μεταβλητής $bubble$ και δεν περιέχει κάποια άλλη λέξη που αποτελεί χαρακτηριστική λέξη μίας άλλης αλγοριθμικής ετικέτας τότε το μοντέλο μας θα ταξινομήσουμε τον κώδικα αυτό στην ετικέτα του αλγόριθμου ταξινόμησης φυσαλίδας. Αδιαμφισβήτητα, το γεγονός αυτό αποτελεί μία ιδιαίτερα ανησυχητική παρατήρηση η οποία χρίζει άμεσης αντιμετώπισης για την συνέχεια της παρούσας εργασίας καθώς αν δεν αντιμετωπιστεί τότε δεν θα μπορέσουμε να αξιολογήσουμε τα σύνθετα μοντέλα μηχανικής μάθησης που μας ενδιαφέρουν. Επιπρόσθετα, σκοπός της παρούσας εργασίας είναι η μοντελοποίηση του πηγαίου κώδικα έτσι ώστε να αξιοποιηθεί από μοντέλα μηχανικής μάθησης γεγονός που προφανώς παρεμποδίζεται από το φαινόμενο αυτό. Πριν όμως προχωρήσουμε στην περιγραφή του τρόπου που επιλύσαμε το συγκεκριμένο φαινόμενο θεωρούμε σκόπιμο να συζητήσουμε 2 επιπλέον παρατηρήσεις για το πείραμα αυτό. Αρχικά, ο τρόπος που αποκρύψαμε τις διάφορες χαρακτηριστικές λέξεις θεωρείται συντηρητικός καθώς θα μπορούσαμε να αποκρύψουμε και τις λέξεις που περιέχουν μία τέτοια λέξη γεγονός που σίγουρα θα οδηγούσε σε περαιτέρω μείωση της μετρικής αναφοράς. Επιπρόσθετα, παρατηρήσαμε ότι όσο αυξανόταν ο αριθμός των εποχών εκπαίδευσης του μοντέλου τόσο αυξανόταν και η ένταση του φαινομένου γεγονός που δικαιολογείται εν μέρει καθώς τα νευρωνικά δίκτυα εκ φύσεως τείνουν να απομνημονεύουν στοιχεία όλο και περισσότερο όσο περνάνε οι εποχές εκπαίδευσης.

Για την αντιμετώπιση του προβλήματος αυτού αποφασίσαμε να "σβήσουμε" τις λέξεις εκείνες οι οποίες εμφανίζονται συχνά σε κάθε αλγοριθμική ετικέτα. Για την εύρεση των λέξεων αυτών αποφασίσαμε να πραγματοποιήσουμε μία μικρή ανάλυση των δεδομένων κατά την οποία για κάθε διαφορετική ετικέτα αναζητήσαμε τις συχνότερα εμφανιζόμενες λέξεις. Πιο συγκεκριμένα, αφού βρήκαμε το πλήθος των εμφανίσεων κάθε λέξης σε κάθε ετικέτα , κρατήσαμε τις 200 πιο συχνά εμφανιζόμενες λέξεις οι οποίες ταυτόχρονα πρέπει να ικανοποιούν τρία ακόμη κριτήρια. Αρχικά, κάθε μία από τις λέξεις αυτές δεν πρέπει να αποτελεί δεσμευμένη λέξη της γλώσσας προγραμματισμού, όπως για παράδειγμα είναι οι λέξεις for και if. Στην συνέχεια, κάθε λέξη πρέπει να εμφανίζεται σε τουλάχιστον το 25% των αρχείων της ετικέτας έτσι ώστε να εξασφαλιστεί ότι η συχνότητα εμφάνισης της λέξης είναι ικανοποιητική. Επιπρόσθετα, κάθε λέξη πρέπει να εμφανίζεται κατά τουλάχιστον

30% του συνολικού πλήθους εμφανίσεων της (σε όλες τις ετικέτες) στην εκάστοτε ετικέτα έτσι ώστε να εξασφαλιστεί ικανοποιητικό πλήθος εμφανίσεων στην ετικέτα αυτή. Αφού λοιπόν εντοπίσαμε τις λέξεις αυτές για κάθε αλγοριθμική ετικέτα αποφασίσαμε να τις αντικαταστήσουμε με ένα μοναδικό αναγνωριστικό έτσι ώστε το μοντέλο μας να αγνοεί τις λέξεις αυτές. Στην συνέχεια, παραθέτουμε ένα παράδειγμα στο οποίο φαίνεται ένας κώδικας πριν και μετά την απόκρυψη των λέξεων αυτών.

```
int * binarySearch ( int * beg , int * end , int what ) {
  while ( end - beg != 1 ) {
    if ( * beg == what ) {
      return beg ;
    }
    int mid = ( end - beg ) / 2 ;
    if ( what <= beg [ mid ] ) {
      end = beg + mid ;
    }
    else {
      beg = beg + mid ;
    }
  }
  return 0 ;
}
```

Σχήμα 6.3: Κώδικας με χαρακτηριστικές λέξεις

```
int * <MARKER> <MARKER> ( int * beg , int * end , int what ) {
  while ( end - beg != 1 ) {
    if ( * beg == what ) {
      return beg ;
    }
    int <MARKER> = ( end - beg ) / 2 ;
    if ( what <= beg [ <MARKER> ] ) {
      end = beg + <MARKER> ;
    }
    else {
      beg = beg + <MARKER> ;
    }
  }
  return 0 ;
}
```

Σχήμα 6.4: Κώδικας με ”σβησμένες” χαρακτηριστικές λέξεις

6.4.2 Αξιολόγηση μοντέλου k-Nearest Neighbors (kNN)

Στην υποενότητα αυτή θα αξιολογήσουμε την αποτελεσματικότητα του kNN μοντέλου. Αρχικά, θεωρούμε σκόπιμο να υπενθυμίσουμε ότι το μοντέλο αυτό είναι ιδιαίτερα σημαντικό για την αξιολόγηση του συνόλου δεδομένων που χρησιμοποιήσαμε καθώς είναι ένα μοντέλο το οποίο δεν διαθέτει μνήμη και έτσι μας δίνει μία πραγματική εικόνα του πόσο κοντά βρίσκονται τα δεδομένα μας. Για να δώσουμε σαν είσοδο έναν κώδικα στο μοντέλο αυτό αποφασίσαμε να αθροίσουμε τα $embeddings$ των λέξεων που αποτελούν των κώδικα και να υπολογίσουμε τον μέσο όρο τους. Ο μέσος όρος αυτός θεωρούμε ότι αποτελεί μία σύνοψη του νοήματος του αλγόριθμου. Στην συνέχεια, θα περάσουμε τον παραπάνω μέσο όρο και από έναν κανονικοποιητή έτσι ώστε να υπάρξει ομοιομορφία ανάμεσα στις εισόδους. Για να επιτύχουμε την βέλτιστη επίδοση του μοντέλου αποφασίσαμε να πραγματοποιήσουμε βελτιστοποίηση υπερπαραμέτρων δημιουργώντας ένα σύνολο βελτιστοποίησης και χρησιμοποιώντας την τεχνική αξιολόγησης 5 πτυχών. Τέλος, αξίζει να σημειώσουμε ότι την διαδικασία αυτή την εκτελέσαμε ξεχωριστά για κάθε διαφορετική τεχνική προ-επεξεργασίας έτσι ώστε να αξιολογήσουμε τόσο το μοντέλο όσο και τις διαφορετικές προτεινόμενες τεχνικές. Στην συνέχεια, παραθέτουμε τα πειραματικά αποτελέσματα που προέκυψαν από την εκτέλεση της παραπάνω διαδικασίας.

Τεχνική προ-επεξεργασίας	Βέλτιστο k	Precision	Recall	F1	MCC
Αποσύνθεση σύνθετων λέξεων	7	0.71	0.69	0.69	0.70
Αποσύνθεση σύνθετων λέξεων + χρωματισμός	9	0.72	0.70	0.70	0.71
Κανονικοποίηση ονομάτων	5	0.73	0.70	0.71	0.71
Κανονικοποίηση ονομάτων + χρωματισμός	7	0.74	0.70	0.71	0.72

Πίνακας 6.7: Αξιολόγηση μοντέλου kNN στο σύνολο δεδομένων με συναρτήσεις

Από τον παραπάνω πίνακα αποτελεσμάτων μπορεί κανείς να εξάγει τρία ενδιαφέροντα συμπεράσματα. Αρχικά, παρατηρούμε ότι η τεχνική της κανονικοποίησης πετυχαίνει υψηλότερες τιμές στις μετρικές αξιολόγησης, γεγονός το οποίο με μία πρώτη ματιά πιθανώς να αποτελεί έκπληξη καθώς η τεχνική της αποσύνθεσης λέξεων περιέχει περισσότερη πληροφορία, ακόμα και μετά την απόκρυψη χαρακτηριστικών λέξεων που πραγματοποιήσαμε προηγουμένως. Μία πιθανή εξήγηση για την συμπεριφορά αυτή είναι πως η τεχνική της κανονικοποίησης προσπαθεί να φέρει τους αλγορίθμους "κοντά" στον χώρο και έτσι το απλό αυτό μοντέλο, το οποίο αγνοεί πλήρως την σειρά των λέξεων στην ακολουθία, λειτουργεί καλύτερα. Επιπρόσθετα, παρατηρούμε ότι η τεχνική του χρωματισμού βάθους βοηθάει (αν και σε μικρό βαθμό) το μοντέλο να ταξινομήσει καλύτερα τους κώδικες σε αλγορίθμους. Παράλληλα, παρατηρούμε ότι η συνολική επίδοση του μοντέλου κινείται σε τιμές κοντά στο 70% γεγονός που υποδηλώνει ότι τα δεδομένα δεν βρίσκονται πολύ κοντά στον χώρο και ότι το σύνολο δεδομένων που δημιουργήσαμε είναι αξιόλογο. Βέβαια, στο σημείο αυτό θεωρούμε σκόπιμο να σημειώσουμε ότι η επίδοση του μοντέλου εξαρτάται άμεσα από την αποτελεσματικότητα του αλγορίθμου $Word2Vec$ γεγονός που συνεπάγεται ότι θα πρέπει να είμαστε επιφυλακτικοί όσον αφορά την αξιολόγηση του συνόλου δεδομένων.

6.4.3 Αξιολόγηση μοντέλου Support Vector Machine (SVM)

Στην υποενότητα αυτή θα αξιολογήσουμε την αποτελεσματικότητα του SVM μοντέλου. Το μοντέλο αυτό θεωρούμε ότι θα μπορούσε να αποδειχθεί ιδιαίτερα αποτελεσματικό για την επίλυση του προβλήματος που πραγματεύεται η παρούσα εργασία καθώς έχει χρησιμοποιηθεί με ιδιαίτερη επιτυχία σε ανάλογα προβλήματα στον τομέα της Επεξεργασίας Φυσικής Γλώσσας. Για να δώσουμε σαν είσοδο έναν κώδικα στο μοντέλο αυτό, όπως και στο μοντέλο kNN που περιγράψαμε προηγουμένως, θα υπολογίσουμε τον μέσο όρο των $embeddings$ των λέξεων που αποτελούν έναν κώδικα. Στην συνέχεια, θα περάσουμε και πάλι τον μέσο όρο αυτό από έναν κανονικοποιητή. Για να επιτύχουμε την βέλτιστη επίδοση του μοντέλου αποφασίσαμε να πραγματοποιήσουμε βελτιστοποίηση υπερπαραμέτρων δημιουργώντας ένα σύνολο βελτιστοποίησης και χρησιμοποιώντας την τεχνική αξιολόγησης 5 πτυχών. Τέλος, αξίζει να σημειώσουμε ότι την διαδικασία αυτή την εκτελέσαμε ξεχωριστά για κάθε διαφορετική τεχνική προ-επεξεργασίας έτσι ώστε να αξιολογήσουμε τόσο το μοντέλο όσο και τις διαφορετικές προτεινόμενες τεχνικές. Στην συνέχεια, παραθέτουμε τα πειραματικά αποτελέσματα που προέκυψαν από την εκτέλεση της παραπάνω διαδικασίας.

Τεχνική προ-επεξεργασίας	Precision	Recall	F1	MCC
Αποσύνθεση σύνθετων λέξεων	0.83	0.83	0.83	0.83
Αποσύνθεση σύνθετων λέξεων + χρωματισμός	0.80	0.80	0.80	0.81
Κανονικοποίηση ονομάτων	0.81	0.80	0.80	0.81
Κανονικοποίηση ονομάτων + χρωματισμός	0.82	0.80	0.81	0.81

Πίνακας 6.8: Αξιολόγηση μοντέλου SVM στο σύνολο δεδομένων με συναρτήσεις

Από τον παραπάνω πίνακα μπορούμε εύκολα να εξάγουμε μερικά χρήσιμα συμπεράσματα. Αρχικά, παρατηρούμε ότι το μοντέλο αυτό παρουσιάζει σημαντικά καλύτερη επίδοση από το kNN μοντέλο γεγονός το οποίο χαρακτηρίζεται αναμενόμενο καθώς το μοντέλο αυτό όπως είπαμε και προηγουμένως λειτουργεί ιδιαίτερα αποτελεσματικά σε ανάλογες εργασίες. Μάλιστα, θα θέλαμε να σημειώσουμε ότι η επίδοση που παρατηρήσαμε στο μοντέλο αυτό είναι αρκετά ικανοποιητική καθώς περίπου 8 στις 10 προβλέψεις είναι σωστές ανεξαρτήτως τεχνικής προ-επεξεργασίας. Στην συνέχεια, παρατηρούμε ότι η τεχνική προ-επεξεργασίας που αφορά την αποσύνθεση ονομάτων μεταβλητών και συναρτήσεων παρουσιάζει τις μέγιστες μετρικές αξιολόγησης. Το γεγονός αυτό θεωρείται αναμενόμενο καθώς όπως αναφέραμε και παραπάνω η τεχνική αυτή εμπεριέχει μεγαλύτερο πλήθος πληροφοριών. Επιπρόσθετα, παρατηρούμε ότι ο χρωματισμός βάθους επιφέρει βελτίωση στην τεχνική της κανονικοποίησης αλλά όχι σε αυτήν της αποσύνθεσης. Το γεγονός αυτό μπορεί να ερμηνευθεί με δύο διαφορετικούς τρόπους. Αρχικά, θα μπορούσαμε να πούμε ότι σε γενικές γραμμές ο χρωματισμός βάθους βοηθάει περισσότερο την κανονικοποιημένη εκδοχή καθώς σε αυτή υπάρχει σοβαρή συμπύκνωση της διαθέσιμης πληροφορίας γεγονός που επιτρέπει την εύκολη συμπλήρωση αυτής με ανάλογες τεχνικές. Παράλληλα, η συμπεριφορά αυτή θα μπορούσε να αποδοθεί και σε στατιστικό σφάλμα καθώς για την βελτιστοποίηση των υπερπαραμέτρων του εκάστοτε μοντέλου δεν έχουμε δοκιμάσει όλους τους δυνατούς συνδυασμούς αλλά έχουμε δοκιμάσει τυχαία μία ποσόστωση αυτών. Η επιλογή αυτή έγινε για πρακτικούς λόγους καθώς χρειάζονταν πολλές ώρες/μέρες για να δοκιμάσουμε όλους τους συνδυασμούς αλλά θα μπορούσε να οδηγήσει σε τέτοιες συμπεριφορές(αν και η πιθανότητα του σεναρίου αυτή είναι σχετικά μικρή καθώς δοκιμάσαμε το 40% των συνδυασμών).

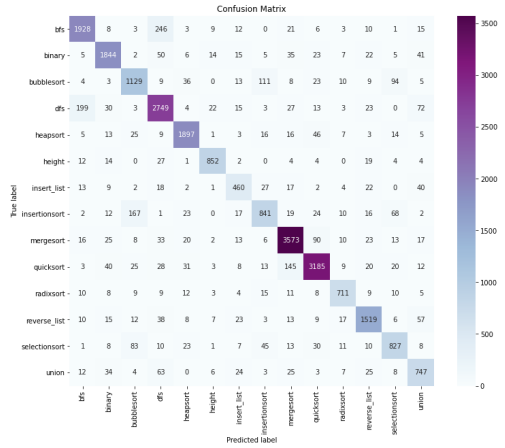
6.4.4 Αξιολόγηση μοντέλου Deep Averaging Network (DAN)

Στην υποενότητα αυτή θα αξιολογήσουμε την αποτελεσματικότητα του DAN μοντέλου. Το μοντέλο αυτό, αναμένεται με βάση και την βιβλιογραφία να λειτουργήσει ιδιαίτερα αποτελεσματικά στην επίλυση του προβλήματος ταξινόμησης κώδικα σε αλγορίθμους. Στο σημείο αυτό, θα θέλαμε να δώσουμε κάποιες τεχνικές λεπτομέρειες του μοντέλου, δεδομένου ότι έχει ήδη δοθεί αναλυτική περιγραφή αυτού σε προηγούμενο κεφάλαιο. Αρχικά, το μοντέλο αυτό διαθέτει ένα παγωμένο $Embedding\ Layer$ το οποίο βασίζεται στα $embeddings$ του αλγορίθμου $Word2Vec$ που δημιουργήσαμε προηγουμένως. Επιπρόσθετα, το μοντέλο αυτό αποτελείται από δύο $Dense\ Layers$ που διαθέτουν 500 νευρώνες το καθένα. Η συνάρτηση ενεργοποίησης των $Layers$ αυτών είναι η $relu$[19]. Επιπρόσθετα, στα $Layers$ αυτά εφαρμόζεται $Dropout$ με τιμή ίση με 50%. Παράλληλα, αξίζει να σημειώσουμε ότι το $loss$ του μοντέλου είναι το $categorical\ cross\ entropy$ και πως στο μοντέλο εφαρμόζουμε τόσο $masking$ (έτσι ώστε να αγνοούνται τα μηδενικά που εισάγονται κατά το $padding$) όσο και $gradient\ clipping$ του οποίου η τιμή ισούται με 1. Τέλος, εκπαιδεύσαμε το μοντέλο για 100 εποχές (χρησιμοποιώντας $Early\ Stopping$ για 10 εποχές με το $patience$ να ισούται με 10^{-4}), το $batch\ size$ που χρησιμοποιήσαμε ισούται με 128 και ο βελτιστοποιητής που χρησιμοποιήσαμε είναι ο $Adam$. Στην συνέχεια, παραθέτουμε τα πειραματικά αποτελέσματα που προέκυψαν από την εκτέλεση της παραπάνω διαδικασίας.

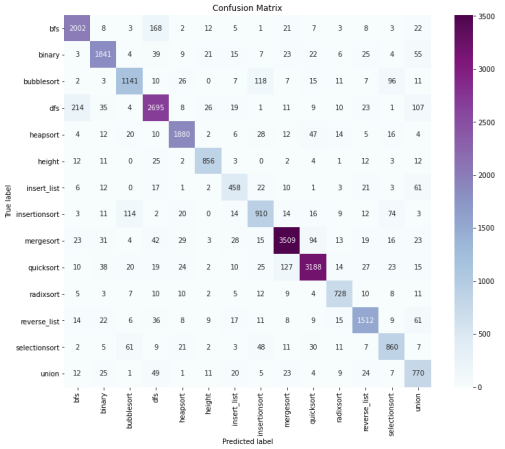
Τεχνική προ-επεξεργασίας	Precision	Recall	F1	MCC
Αποσύνθεση σύνθετων λέξεων	0.84	0.84	0.84	0.85
Αποσύνθεση σύνθετων λέξεων + χρωματισμός	0.85	0.85	0.85	0.86
Κανονικοποίηση ονομάτων	0.82	0.81	0.82	0.82
Κανονικοποίηση ονομάτων + χρωματισμός	0.83	0.81	0.82	0.83

Πίνακας 6.9: Αξιολόγηση μοντέλου DAN στο σύνολο δεδομένων με συναρτήσεις

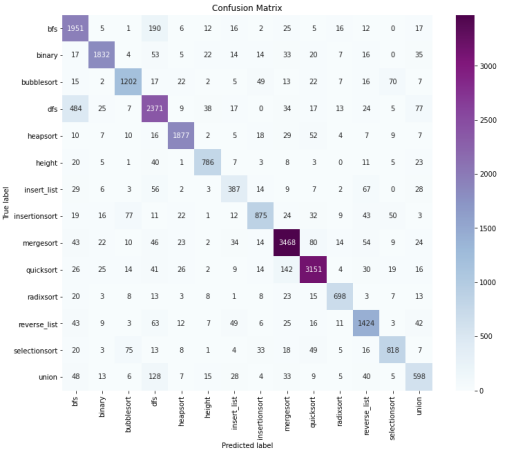
Από τον παραπάνω πίνακα αποτελεσμάτων μπορούμε εύκολα να εξάγουμε διάφορα ενδιαφέροντα συμπεράσματα. Αρχικά, παρατηρούμε ότι το μοντέλο αυτό λειτουργεί καλύτερα από τα μοντέλα που έχουμε μελετήσει μέχρι τώρα γεγονός το οποίο εν μέρη θεωρείται αναμενόμενο καθώς το μοντέλο αυτό είναι αρκετά πιο περίπλοκο από τα προηγούμενα. Επιπρόσθετα, παρατηρούμε ότι και πάλι η τεχνική της αποσύνθεσης παρουσιάζει υψηλότερες τιμές στις μετρικές αναφοράς σε σύγκριση με την τεχνική της κανονικοποίησης. Φυσικά, όπως έχουμε συζητήσει και προηγουμένως το γεγονός αυτό δεν αποτελεί έκπληξη καθώς η πρώτη τεχνική κρύβει μέσα της περισσότερη πληροφορία. Τέλος, παρατηρούμε ότι η τεχνική του χρωματισμού βάθους προσφέρει τόσο για την τεχνική της αποσύνθεσης όσο και την τεχνική της κανονικοποίησης μικρή αλλά σαφή βελτίωση των αποτελεσμάτων, γεγονός το οποίο χαρακτηρίζεται ως θετικό καθώς αυτός ήταν εξ' αρχής ο σκοπός της συγκεκριμένης τεχνικής. Στην συνέχεια, για την καλύτερη πειραματική αξιολόγηση του μοντέλου αυτού θα παραθέσουμε τους πίνακες σύγχυσης του μοντέλου για κάθε διαφορετική τεχνική προ-επεξεργασίας.



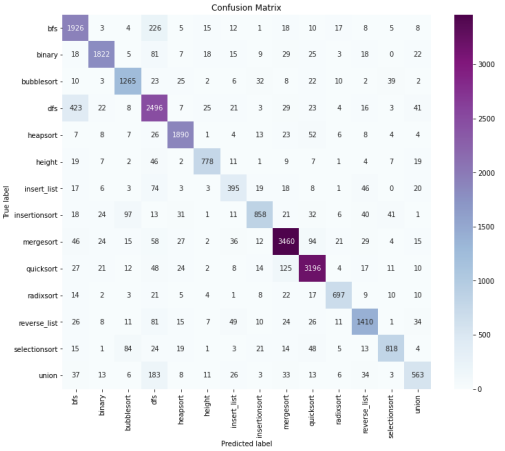
Σχήμα 6.5: Confusion matrix για τεχνική αποσύνθεσης (DAN)



Σχήμα 6.6: Confusion matrix για τεχνική αποσύνθεσης + χρωματισμός (DAN)



Σχήμα 6.7: Confusion matrix για τεχνική κανονικοποίησης (DAN)



Σχήμα 6.8: Confusion matrix για τεχνική κανονικοποίησης + χρωματισμός (DAN)

Οι παραπάνω πίνακες παρουσιάζουν ιδιαίτερο ενδιαφέρον καθώς μας δείχνουν αναλυτικά τις προβλέψεις του μοντέλου. Αρχικά, παρατηρούμε ότι η τεχνική της αποσύνθεσης έχει μια πολύ μεγάλη διαφορά σε σχέση με την τεχνική της κανονικοποίησης. Πιο συγκεκριμένα, παρατηρούμε ότι οι ετικέτες DFS και BFS ταξινομούνται καλύτερα με την τεχνική της αποσύνθεσης γεγονός που υποδεικνύει ότι στην περίπτωση αυτή τα ονόματα μεταβλητών και συναρτήσεων είναι μάλλον απαραίτητα. Στην συνέχεια, παρατηρούμε μία γενική πτώση επιτυχημένων προβλέψεων σε σχεδόν όλες τις ετικέτες όταν συγκρίνουμε την τεχνική της κανονικοποίησης με αυτήν της αποσύνθεσης γεγονός που βέβαια χαρακτηρίζεται αναμενόμενο αν λάβουμε υπόψιν και τον πίνακα αποτελεσμάτων που παραθέσαμε προηγουμένως. Τέλος, θα προσπαθήσουμε να σχολιάσουμε την επίδραση του χρωματισμού βάθους με βάση τους παραπάνω πίνακες. Παρατηρούμε ότι ο χρωματισμός βάθους φαίνεται να βοηθάει αρκετά στην ταξινόμηση κάποιων αλγοριθμικών ετικετών ενώ σε κάποιες άλλες φαίνεται όχι μόνο να μην βοηθάει αλλά να δυσκολεύει περισσότερο το μοντέλο. Το γεγονός αυτό θεωρούμε ότι σχετίζεται με εξηγήσεις που έχουν δοθεί στην βιβλιογραφία και αναφέρουν ότι δεν είναι για όλους τους αλγορίθμους απαραίτητη η δομή του κώδικα για την σωστή ταξινόμηση τους. Επιπρόσθετα, αξίζει να αναφέρουμε ότι το μοντέλο αυτό δεν λαμβάνει υπόψιν την σειρά των λέξεων γεγονός που μπορεί να μην βοηθάει στην ανάδειξη της ιδιότητας αυτής.

6.4.5 Αξιολόγηση μοντέλου Bidirectional LSTM με Attention Layer

Στην υποενότητα αυτή θα αξιολογήσουμε την αποτελεσματικότητα του BiLSTM + Attention μοντέλου. Στο σημείο αυτό, θα θέλαμε να δώσουμε κάποιες τεχνικές λεπτομέρειες του μοντέλου, δεδομένου ότι έχει ήδη δοθεί αναλυτική περιγραφή αυτού σε προηγούμενο κεφάλαιο. Αρχικά, το μοντέλο αυτό διαθέτει ένα παγωμένο $Embedding\ Layer$ το οποίο βασίζεται στα $embeddings$ του αλγορίθμου $Word2Vec$ που δημιουργήσαμε προηγουμένως. Επιπρόσθετα, το μοντέλο αυτό αποτελείται από ένα $bidirectional\ LSTM$ το οποίο έχει $hidden\ state$ μεγέθους 128 σε κάθε κατεύθυνση. Μάλιστα, στο $LSTM$ αυτό εφαρμόζεται $Dropout$ με τιμή ίση με 40%. Παράλληλα, αξίζει να σημειώσουμε ότι το $loss$ του μοντέλου είναι το $categorical\ cross\ entropy$ και πως στο μοντέλο εφαρμόζουμε τόσο $masking$ (έτσι ώστε να αγνοούνται τα μηδενικά που εισάγονται κατά το $padding$) όσο και $gradient\ clipping$ του οποίου η τιμή ισούται με 1. Τέλος, εκπαιδεύσαμε το μοντέλο για 50 εποχές (χρησιμοποιώντας $Early\ Stopping$ για 5 εποχές με το $patience$ να ισούται με 10^{-4}), το $batch\ size$ που χρησιμοποιήσαμε ισούται με 64 και ο βελτιστοποιητής που χρησιμοποιήσαμε είναι ο $Adam$. Στην συνέχεια, παραθέτουμε τα πειραματικά αποτελέσματα που προέκυψαν από την εκτέλεση της παραπάνω διαδικασίας.

Τεχνική προ-επεξεργασίας	Precision	Recall	F1	MCC
Αποσύνθεση σύνθετων λέξεων	0.91	0.91	0.91	0.92
Αποσύνθεση σύνθετων λέξεων + χρωματισμός	0.92	0.92	0.92	0.93
Κανονικοποίηση ονομάτων	0.87	0.85	0.86	0.86
Κανονικοποίηση ονομάτων + χρωματισμός	0.86	0.86	0.86	0.87

Πίνακας 6.10: Αξιολόγηση μοντέλου BiLSTM + Attention στο σύνολο δεδομένων με συναρτήσεις

Από τον παραπάνω πίνακα εύκολα μπορεί κανείς να εξάγει χρήσιμα συμπεράσματα τόσο σχετικά με την επίδοση του μοντέλου όσο και σχετικά με τις τεχνικές προ-επεξεργασίας. Αρχικά, παρατηρούμε ότι το μοντέλο αυτό είναι ιδιαίτερα αποτελεσματικό στην επίλυση του προβλήματος που εξετάζεται στην παρούσα εργασία. Μάλιστα, είναι αξιοσημείωτο ότι το μοντέλο αυτό είναι ιδιαίτερα αποτελεσματικό για όλες τις διαφορετικές εκδοχές προ-επεξεργασίας έχοντας μεγάλη διαφορά από τα μοντέλα που είχαμε εξετάσει ως τώρα. Το γεγονός αυτό, εξηγείται πιθανώς από το γεγονός ότι το μοντέλο αυτό είναι το πρώτο ακολουθιακό μοντέλο που εξετάσαμε γεγονός που δίνει την δυνατότητα στο μοντέλο να μοντελοποιήσει καλύτερα την δομή του κώδικα (υπενθυμίζουμε ότι τα μοντέλα που εξετάσαμε προηγουμένως αγνοούσαν την σειρά των λέξεων). Επιπρόσθετα, αξίζει να σημειώσουμε ότι οι παρατηρήσεις που εξάγαμε στα προηγούμενα μοντέλα σχετικά με τις τεχνικές προ-επεξεργασίας κατά κύριο λόγο επαληθεύονται και σε αυτό το μοντέλο παρόλο που το μοντέλο αυτό διαφέρει αρκετά στον τρόπο που διαχειρίζεται τον κώδικα. Πιο συγκεκριμένα, παρατηρούμε και πάλι ότι η τεχνική της αποσύνθεσης υπερτερεί της τεχνικής της κανονικοποίησης και η τεχνική του χρωματισμού βάθους βοηθάει το μοντέλο (ανεξαρτήτου τεχνικής προ-επεξεργασίας) να μοντελοποιήσει καλύτερα τον κώδικα.

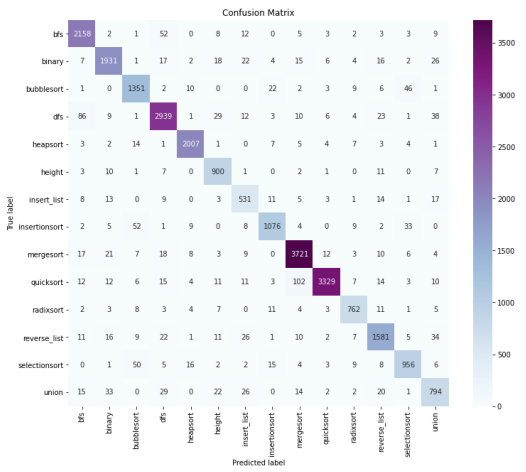
6.4.6 Αξιολόγηση μοντέλου Hierarchical Attention Network (HAN)

Στην υποενότητα αυτή θα αξιολογήσουμε την αποτελεσματικότητα του HAN μοντέλου. Στο σημείο αυτό, θα θέλαμε να δώσουμε κάποιες τεχνικές λεπτομέρειες του μοντέλου, δεδομένου ότι έχει ήδη δοθεί αναλυτική περιγραφή αυτού σε προηγούμενο κεφάλαιο. Αρχικά, το μοντέλο αυτό διαθέτει ένα παγωμένο $Embedding\ Layer$ το οποίο βασίζεται στα $embeddings$ του αλγορίθμου $Word2Vec$ που δημιουργήσαμε προηγουμένως. Επιπρόσθετα, το μοντέλο αυτό αποτελείται από δύο $bidirectional\ LSTM$ (ένα για τον κωδικοποιητή γραμμής και ένα για τον κωδικοποιητή κώδικα) τα οποία έχουν $hidden\ state$ μεγέθους 64 σε κάθε κατεύθυνση. Μάλιστα, στα $LSTM$ αυτά εφαρμόζεται $Dropout$ με τιμή ίση με 40%. Παράλληλα, αξίζει να σημειώσουμε ότι το $loss$ του μοντέλου είναι το $categorical\ cross\ entropy$ και πως στο μοντέλο εφαρμόζουμε τόσο $masking$ (έτσι ώστε να αγνοούνται τα μηδενικά που εισάγονται κατά το $padding$) όσο και $gradient\ clipping$ του οποίου η τιμή ισούται με 1. Τέλος, εκπαιδεύσαμε το μοντέλο για 50 εποχές (χρησιμοποιώντας $Early\ Stopping$ για 5 εποχές με το $patience$ να ισούται με 10^{-4}), το $batch\ size$ που χρησιμοποιήσαμε ισούται με 32 και ο βελτιστοποιητής που χρησιμοποιήσαμε είναι ο $Adam$. Στην συνέχεια, παραθέτουμε τα πειραματικά αποτελέσματα που προέκυψαν από την εκτέλεση της παραπάνω διαδικασίας.

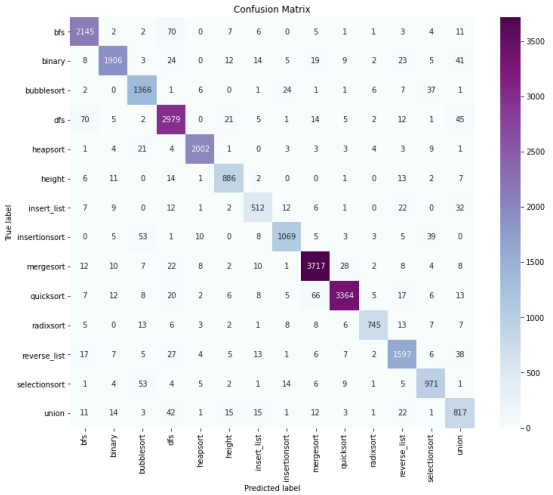
Τεχνική προ-επεξεργασίας	Precision	Recall	F1	MCC
Αποσύνθεση σύνθετων λέξεων	0.92	0.92	0.92	0.93
Αποσύνθεση σύνθετων λέξεων + χρωματισμός	0.92	0.92	0.92	0.93
Κανονικοποίηση ονομάτων	0.88	0.87	0.87	0.88
Κανονικοποίηση ονομάτων + χρωματισμός	0.87	0.87	0.87	0.88

Πίνακας 6.11: Αξιολόγηση μοντέλου HAN στο σύνολο δεδομένων με συναρτήσεις

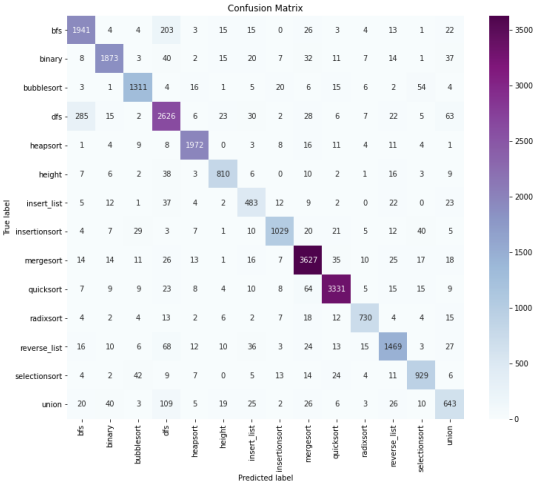
Από τον παραπάνω πίνακα μπορούμε εύκολα να εξάγουμε μερικές χρήσιμες παρατηρήσεις σχετικά τόσο με την επίδοση του μοντέλου όσο και με την αποτελεσματικότητα των διαφορετικών τεχνικών προ-επεξεργασίας. Αρχικά, παρατηρούμε ότι το μοντέλο παρουσιάζει βέλτιστη επίδοση, για όλες τις τεχνικές προ-επεξεργασίας, σε σχέση με τα μοντέλα που εξετάσαμε προηγουμένως. Μάλιστα, το μοντέλο είναι αποτελεσματικότερο και από το BiLSTM + Attention μοντέλο της προηγούμενης υποενότητας γεγονός το οποίο χαρακτηρίζεται αναμενόμενο καθώς το HAN μοντέλο είναι αρκετά πιο σύνθετο και επεξεργάζεται τον κώδικα αρχικά σε μικρότερα κομμάτια (γραμμές) και στην συνέχεια σαν κώδικα (συνονθύλευμα γραμμών) σε αντίθεση με το BiLSTM + Attention μοντέλο το οποίο επεξεργάζεται όλο τον κώδικα ενιαία γεγονός που κατά πάσα πιθανότητα μειώνει την εκφραστικότητα του μοντέλου καθώς και το εμποδίζει, σε κάποιες περιπτώσεις, να μάθει πολύπλοκες συσχετίσεις για να ταξινομήσει καλύτερα τον πηγαίο κώδικα σε αλγορίθμους. Επιπρόσθετα, παρατηρούμε ότι ο χρωματισμός βάθους κώδικα στην συγκεκριμένη περίπτωση δεν φαίνεται να βοηθάει ιδιαίτερα, γεγονός που αντιτίθεται στις παρατηρήσεις που εξάγαμε προηγουμένως. Η συμπεριφορά αυτή πιθανώς εξηγείται από το γεγονός ότι η αρχιτεκτονική του δικτύου αυτού είναι ιδιαίτερα περίπλοκη και μοντελοποιεί εκ φύσεως πληροφορίες σχετικά με την δομή του κώδικα πράγμα το οποίο δεν συμβαίνει στις προηγούμενες αρχιτεκτονικές. Μάλιστα, είναι χαρακτηριστικό πως η αρχιτεκτονική αυτή επιτυγχάνει να κωδικοποιεί και πληροφορίες που σχετίζονται με τα συμφραζόμενα της λέξης και όχι μόνο με το νόημα της λέξης αυτής καθ΄ αυτής. Στην συνέχεια, για την καλύτερη πειραματική αξιολόγηση του μοντέλου παραθέτουμε τους πίνακες σύγχυσης για κάθε διαφορετική τεχνική.



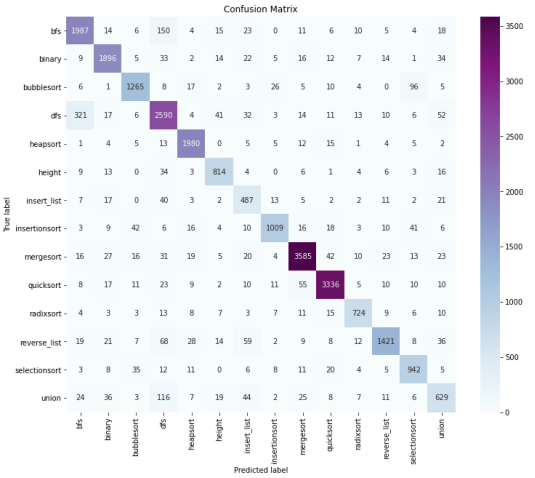
Σχήμα 6.9: Confusion matrix για τεχνική αποσύνθεσης (HAN)



Σχήμα 6.10: Confusion matrix για τεχνική αποσύνθεσης + χρωματισμός (HAN)



Σχήμα 6.11: Confusion matrix για τεχνική κανονικοποίησης (HAN)



Σχήμα 6.12: Confusion matrix για τεχνική κανονικοποίησης + χρωματισμός (HAN)

Από τους παραπάνω πίνακες μπορεί κανείς να εξάγει διάφορα ενδιαφέροντα συμπεράσματα, τα οποία ως ένα βαθμό συμβαδίζουν με τα ανάλογα συμπεράσματα που εξάγαμε κατά τον σχολιασμό του DAN μοντέλου. Αρχικά, παρατηρούμε ότι ανάμεσα στην τεχνική της αποσύνθεσης και της κανονικοποίησης παρατηρείται έντονη διαφορά του πλήθους των αποτυχημένων προβλέψεων για τις ετικέτες DFS και BFS γεγονός που παρατηρήθηκε και προηγουμένως. Πάραυτα, παρατηρούμε ότι το πλήθος των αποτυχημένων προβλέψεων έχει πλέον μειωθεί αισθητά στο μοντέλο αυτό (κατά περίπου 40% από το DAN μοντέλο). Επιπρόσθετα, συγκρίνοντας τους παραπάνω πίνακες με τους αντίστοιχους του DAN μοντέλου κάνουμε μία πολύ ενδιαφέρουσα παρατήρηση. Πιο συγκεκριμένα, παρατηρούμε ότι το ακολουθιακό αυτό μοντέλο βελτιώνεται ιδιαίτερα στην πρόβλεψη και στον διαχωρισμό αλγοριθμικών ετικετών που εκ φύσεως οι υλοποιήσεις τους μοιάζουν μεταξύ τους. Για παράδειγμα, τέτοιοι αλγόριθμοι είναι οι $bubblesort$ και $insertionsort$ οι οποίοι υλοποιούνται συνήθως με χρήση 2 επαναληπτικών βρόγχων. Παρατηρούμε ότι το HAN μοντέλο στην περίπτωση αυτή προβλέπει και διαχωρίζει κατά 60% καλύτερα τις ετικέτες αυτές. Η συμπεριφορά αυτή πιστεύουμε ότι οφείλεται τόσο στην ακολουθιακή φύση του μοντέλου όσο και στην εκφραστικότητα της αρχιτεκτονικής του.

6.4.7 Συνολικός σχολιασμός

Στο σημείο αυτό επιθυμούμε να πραγματοποιήσουμε ένα συνολικό σχολιασμό της ενότητας αυτής η οποία σχετίζεται με το σύνολο δεδομένων με συναρτήσεις. Αρχικά, παρατηρήσαμε ότι τα απλούστερα μοντέλα που εξετάσαμε (kNN,SVM,DAN) τα οποία έπαιρναν σαν είσοδο τον μέσο όρο των $embeddings$ των λέξεων του κώδικα, αποδείχτηκαν ιδιαίτερα αποτελεσματικά ειδικά αν κανείς συνυπολογίσει την απλότητα τους. Παρ' όλα αυτά, παρατηρήσαμε ότι τα μοντέλα αυτά παρουσίασαν περιορισμούς ειδικά σε ότι αφορά τον διαχωρισμό (και κατ' επέκταση την ορθή πρόβλεψη) των αλγορίθμων εκείνων οι οποίοι εκ φύσεως έχουν παρόμοιες υλοποιήσεις. Μερικά παραδείγματα τέτοιων αλγορίθμων είναι οι αλγόριθμοι ταξινόμησης $bubblesort$, $insertionsort$ και $selectionsort$ καθώς και οι αλγόριθμοι διάσχισης DFS και BFS. Τα μοντέλα αυτά σε πολλές περιπτώσεις αποτυγχάνουν να διαχωρίσουν τέτοιους αλγορίθμους γιατί αγνοούν εντελώς την δομή του κώδικα και βασίζονται μονάχα στον μέσο όρο των λέξεων από τις οποίες αποτελείται ένας κώδικας. Το πρόβλημα αυτό καθώς και η συνολική εικόνα βελτιώθηκε αρκετά με την εξέταση των πιο σύνθετων μοντέλων (BiLSTM + Attention, HAN) τα οποία όπως παρατηρήσαμε πέτυχαν αρκετά υψηλότερες τιμές των μετρικών αναφοράς και κατάφεραν να διαχωρίσουν αρκετά καλύτερα ακόμα και αλγόριθμους με παρόμοιες υλοποιήσεις. Μάλιστα, ειδική μνεία αξίζει να κάνουμε στο HAN μοντέλο το οποίο όπως αποδείχτηκε είναι το πιο αποτελεσματικό (από τα μοντέλα που εξετάσαμε) στο πρόβλημα της ταξινόμησης πηγαίου κώδικα σε αλγορίθμους. Το γεγονός αυτός είναι αναμενόμενο και οφείλεται κατά κύριο λόγω τόσο στην ακολουθιακή φύση του μοντέλου όσο και στην εκφραστικότητα που επιτρέπει η αρχιτεκτονική του μοντέλου.

Στην συνέχεια, επιθυμούμε να σχολιάσουμε τις διάφορες τεχνικές προ-επεξεργασίας που εξετάσαμε στην ενότητα αυτή. Αρχικά, παρατηρήσαμε ότι η τεχνική της αποσύνθεσης αποδίδει αρκετά καλύτερα από την τεχνική της κανονικοποίησης ακόμα και όταν έχουμε αποκρύψει της χαρακτηριστικές λέξεις του κάθε αλγορίθμου. Το γεγονός αυτό θεωρείται σε γενικές γραμμές αναμενόμενο και μάλιστα η συμπεριφορά αυτή παρατηρήθηκε κατά την αξιολόγηση όλων των μοντέλων εκτός του kNN όπου όμως η συμπεριφορά αυτή μπορεί πιθανώς να αποδοθεί στην απλότητα του μοντέλου. Το ενδιαφέρον όμως σχετικά με τις δύο τεχνικές αυτές δεν είναι η υπεροχή της τεχνικής της αποσύνθεσης αλλά το πόσο καλά απέδωσε η τεχνική της κανονικοποίησης και πόσο κοντά βρίσκονται οι τιμές αυτής στις τιμές της τεχνικής της αποσύνθεσης. Το γεγονός αυτό είναι ιδιαίτερα σημαντικό καθώς η τεχνική της κανονικοποίησης περιλαμβάνει μόλις το 10% των λέξεων της τεχνική της αποσύνθεσης γεγονός που συνεπάγεται δραματική μείωση της διαθέσιμης πληροφορίας. Το γεγονός αυτό είναι ιδιαίτερα ενθαρρυντικό και μας αποδεικνύει τόσο την δυναμική των νευρωνικών δικτύων όσο και την ποιότητα της τεχνικής προ-επεξεργασίας που προτείναμε. Επιπρόσθετα, αξίζει να σημειώσουμε ότι ο λόγος που η τεχνικής της κανονικοποίησης πετυχαίνει τόσο υψηλές τιμές είναι πιθανόν να οφείλεται στο ότι μειώνει το φαινόμενο το λέξεων εκτός λεξιλογίου και κώδικες οι οποίοι διαφέρουν κυρίως στην ονοματοδοσία των μεταβλητών πλέον είναι όμοιοι ή σχεδόν όμοιοι χάρη στην τεχνική αυτή. Μία ακόμα ενδιαφέρουσα παρατήρηση είναι πως ακόμα και αν αποκρύψαμε τις χαρακτηριστικές λέξεις κάθε αλγορίθμου, γεγονός που οδήγησε επίσης σε μείωση της διαθέσιμης πληροφορίας τα μοντέλα μας κατάφεραν να ταξινομήσουν τους κώδικες με τιμές αναφοράς κοντά στο 90% γεγονός που σε πρώτη φάση μοιάζει εντυπωσιακό. Σε δεύτερη φάση όμως μας δίνει το έναυσμα να πιστέψουμε ότι θα μπορούσαμε να δημιουργήσουμε μία πιο έξυπνη τεχνική κανονικοποίησης

η οποία να πετυχαίνει ανάλογες τιμές των μετρικών αναφοράς. Η παρατήρηση αυτή όχι μόνο μας κάνει να αναρωτιόμαστε πόση πληροφορία είναι απαραίτητη για το συγκεκριμένο πρόβλημα αλλά και να αμφισβητούμε διάφορες προτάσεις της βιβλιογραφίας[46] σχετικά με τα ονόματα των μεταβλητών και συναρτήσεων. Τέλος, θα θέλαμε να σχολιάσουμε την τεχνική του χρωματισμού βάθους η οποία σε γενικές γραμμές έδειξε να βοηθάει τα μοντέλα να επιλύσουν καλύτερα το πρόβλημα που πραγματεύεται η παρούσα εργασία. Πιο συγκεκριμένα, παρατηρήσαμε ότι για όλα τα μοντέλα, εκτός του HAN μοντέλου, η τεχνική αυτή αύξησε κατά περίπου 1-2% τις μετρικές αναφοράς γεγονός που υποδεικνύει ότι βοηθάει τα μοντέλα να ταξινομήσουν καλύτερα τον κώδικα σε αλγορίθμους. Μάλιστα, παρατηρήσαμε ότι τα μοντέλα προέβλεπαν καλύτερα κάποιες αλγοριθμικές ετικέτες ενώ κάποιες άλλες χειρότερα. Η συμπεριφορά αυτή αποδόθηκε στο γεγονός ότι η δομή του κώδικα δεν είναι σημαντική για όλους τους αλγορίθμους καθώς υπάρχουν περιπτώσεις που είναι απαραίτητη ενώ άλλες που δεν είναι και τόσο. Τέλος, παρατηρήσαμε ότι η τεχνική του χρωματισμού δεν βοήθησε καθόλου όταν δοκιμάστηκε στο HAN μοντέλο γεγονός το οποίο πιστεύουμε ότι οφείλεται στο ότι η αρχιτεκτονική του μοντέλου αυτού, εκ φύσεως κωδικοποιεί ένας μέρος της δομής του εκάστοτε κώδικα. Συνολικά, θα λέγαμε ότι η συγκεκριμένη τεχνική φαίνεται μεν να συνεισφέρει αλλά πιθανώς η χρήση της δεν δικαιολογεί τους πολλούς υπολογιστικούς πόρους που απαιτεί για την εισαγωγή της στον εκάστοτε κώδικα. Ως εκ τούτου, πιστεύουμε ότι η μέθοδος αυτή θα μπορούσε να είναι χρήσιμη αρκεί να δημιουργηθεί κάποια απλούστερη μέθοδος που να εισάγει τον χρωματισμό βάθους σε ένα σύνολο δεδομένων.

Κεφάλαιο 7

Συμπεράσματα

Στην παρούσα διπλωματική εργασία ασχοληθήκαμε με το ζήτημα της ταξινόμησης πηγαίου κώδικα σε αλγόριθμους. Αρχικά, δημιουργήσαμε δύο διαφορετικά σύνολα δεδομένων κατεβάζοντας αρχεία κώδικα από την ιστοσελίδα $Github$. Τα σύνολα δεδομένων αυτά αφορούν ένα σύνολο με επαναληπτικούς βρόγχους και ένα σύνολο με συναρτήσεις. Στην συνέχεια, προτείναμε διάφορες τεχνικές προ-επεξεργασίας οι οποίες σχετίζονται τόσο με υπάρχουσες προτάσεις της βιβλιογραφίας όσο και με δικές μας προτάσεις που πιστεύαμε ότι θα μπορούσαν να βοηθήσουν στην επίλυση του προβλήματος αυτού. Τέλος, εκπαιδεύσαμε και αξιολογήσαμε διάφορα μοντέλα μηχανικής μάθησης και στα δύο σύνολα δεδομένων. Στο κεφάλαιο αυτό, επιθυμούμε να παραθέσουμε ορισμένες μελλοντικές προεκτάσεις του θέματος που πραγματεύεται η εργασία αυτή.

7.1 Μελλοντικές Προεκτάσεις

Στην ενότητα αυτή θα παραθέσουμε μερικές μελλοντικές προεκτάσεις που πιστεύουμε ότι θα μπορούσαν να οδηγήσουν σε ενδιαφέροντα αποτελέσματα. Οι προεκτάσεις αυτές σχετίζονται τόσο με τα μοντέλα μηχανικής μάθησης όσο και με τις τεχνικές προ-επεξεργασίας και την δημιουργία συνόλων δεδομένων.

Η πρώτη προέκταση που πιστεύουμε ότι θα είχε ιδιαίτερο ενδιαφέρον σχετίζεται με τα μοντέλα μηχανικής μάθησης. Η αλήθεια είναι ότι υπάρχουν πολλές επιλογές σε ότι σχετίζεται με τα μοντέλα που θα μπορούσε να επιλέξει κανείς για να επιλύσει το πρόβλημα που πραγματεύεται η παρούσα εργασία. Στην βιβλιογραφία, κατά κύριο λόγο πλέον κυριαρχούν μοντέλα τα οποία παίρνουν σαν είσοδο την δενδρική δομή του κώδικα (AST) και όχι τον κώδικα αυτόν κάθε αυτόν. Στην παρούσα εργασία όμως έχουμε επιλέξει για λόγους απλότητας να δώσουμε σαν είσοδο τον κώδικα σαν κείμενο. Εμμένοντας στην απόφαση αυτή πιστεύουμε ότι υπάρχουν κάποια μοντέλα τα οποία θα είχαν ιδιαίτερο ενδιαφέρον να δοκιμαστούν καθώς έχουν αποδειχθεί επιτυχημένα κατά την χρήση τους σε αντίστοιχα προβλήματα του τομέα της Επεξεργασίας Φυσικής Γλώσσας. Τα προτεινόμενα μοντέλα είναι ένα μοντέλο που χρησιμοποιεί $CNNs$[33, 34], ένα $CNN-LSTM$[35] μοντέλο καθώς και ένα μοντέλο που αποτελείται από τον κωδικοποιητή ενός $transformer$[60].

Η δεύτερη προέκταση που πιστεύουμε ότι έχει ιδιαίτερο ενδιαφέρον σχετίζεται με το ότι στην βιβλιογραφία κυριαρχούν μοντέλα τα οποία παίρνουν σαν είσοδο δεντρικές μορφές του κώδικα (AST)[51, 55, 56]. Πιστεύουμε ότι θα είχε ενδιαφέρον να αξιολογήσουμε τέτοια μοντέλα πάνω στα σύνολα δεδομένων που δημιουργήσαμε και τέλος να τα συγκρίνουμε με τα απλά μοντέλα που παίρνουν σαν είσοδο τον κώδικα αυτούσιο. Ο λόγος που πιστεύουμε ότι η προέκταση αυτή έχει ενδιαφέρον έγκειται στο γεγονός ότι τα σύνολα δεδομένων που έχουμε δημιουργήσει πληρούν διάφορες προδιαγραφές ποιότητας όπως είναι ο εντοπισμός και η απομάκρυνση αρχείων που είναι αντίγραφα ή σχεδόν αντίγραφα. Το γεγονός αυτό επιτρέπει την πραγματοποίηση μίας ρεαλιστικής και ποιοτικής σύγκρισης ανάμεσα στο μοντέλα.

Η τρίτη προέκταση που πιστεύουμε ότι θα είχε ενδιαφέρον αφορά την παρατήρηση που εξάγαμε κατά την αξιολόγηση του συνόλου δεδομένων με συναρτήσεις. Πιο συγκεκριμένα, παρατηρήσαμε ότι είναι πιθανόν σε κάθε αλγοριθμική ετικέτα να υπάρχουν χαρακτηριστικές λέξεις οι οποίες να δημιουργούν προκαταλήψεις στα μοντέλα μηχανικής μάθησης αναγκάζοντας τα να αναζητούν συγκεκριμένες λέξεις μέσα στον κώδικα αντί να προσπαθούν να τον μοντελοποιήσουν. Θεωρούμε ότι θα είχε ιδιαίτερο ενδιαφέρον να δούμε αν η παρατήρηση αυτή αφορά και τα μοντέλα που παίρνουν σαν είσοδο δεντρικές μορφές τους κώδικα καθώς στην βιβλιογραφία[55, 57] είναι αρκετά συχνό να συμπεριλαμβάνονται τα ονόματα των μεταβλητών σε ένα σύνολο δεδομένων. Σε περίπτωση που η παρατήρηση αυτή επηρεάζει και τα δίκτυα αυτού του τύπου υπάρχει σοβαρή πιθανότητα οι δημιουργοί των δικτύων αυτών να βλέπουν υψηλότερες τιμές των μετρικών αναφοράς οι οποίες όμως δεν συνδέονται απαραίτητα με την δυναμική του μοντέλου να μοντελοποιήσει καλύτερα τον κώδικα.

Η τέταρτη και τελευταία προέκταση σχετίζεται με την δημιουργία συνόλων δεδομένων. Πιο συγκεκριμένα, θεωρούμε ότι θα είχε ενδιαφέρον να γίνει έρευνα προς την κατεύθυνση της αυτοματοποίησης της διαδικασίας αυτής με την χρήση μοντέλων μηχανικής μάθησης. Μία ιδέα είναι, να δημιουργηθούν λίγοι συνθετικοί κώδικες ανά αλγοριθμική ετικέτα και στην συνέχεια με βάση αυτούς ένα μοντέλο να μπορεί να απομονώσει τα σημεία του κώδικα στα οποία υλοποιείται ο κάθε αλγόριθμος. Η ιδέα αυτή είναι βέβαια αρκετά δύσκολο να υλοποιηθεί καθώς τα συνθετικά δεδομένα θα πρέπει να είναι ιδιαίτερα αντιπροσωπευτικά της εκάστοτε αλγοριθμικής ετικέτας. Παρ'όλ'αυτά πιστεύουμε ότι είναι απαραίτητη η έρευνα προς αυτή την κατεύθυνση καθώς είναι μάλλον ο μόνος τρόπος να δημιουργηθούν χωρίς ιδιαίτερο κόπο μεγάλα και ποιοτικά σύνολα δεδομένων τα οποία διαθέτουν ετικέτες και είναι απαραίτητα για την εκπαίδευση μοντέλων βαθιάς μηχανικής μάθησης.

Κατάλογος γραφικών παραστάσεων

Κατάλογος πινάκων

Κατάλογος αλγορίθμων

Γλωσσάριο

Αγγλικός όρος	Ελληνικός όρος
RNN	αναδρομικό νευρωνικό δίκτυο
CNN	συνελικτικό νευρωνικό δίκτυο
LSTM	αναδρομικό νευρωνικό δίκτυο με βραχέα μνήμη
Relu	διορθωμένη γραμμική μονάδα
PCA	Ανάλυση κύριων συστατικών
Layer	στρώση
Attention	προσοχή
Dropout	εγκατάλειψη
loss	απώλεια
categorical cross entropy	κατηγορική εγκάρσια εντροπία
hidden state	κρυφή κατάσταση
optimizer	βελτιστοποιητής
precision	ακρίβεια
recall	ανάκληση
output	έξοδος
for	για
if	αν
while	ενώ
patience	υπομονή
gradient clipping	περικοπή παραγώγων
early stopping	έγκαιρη διακοπή (εκπαίδευσης)
masking	συγκάλυψη
padding	γέμισμα ακολουθίας με 0
bubblesort	ταξινόμηση φυσαλίδας
selectionsort	ταξινόμηση επιλογής
insertionsort	ταξινόμηση εισαγωγής
radixsort	ταξινόμηση ακτίνας
dfs	διάσχιση κατά βάθος
bfs	διάσχιση κατά πλάτος
confusion matrix	πίνακας σύγχυσης

batch size	μέγεθος παρτίδας
gpu	κάρτα γραφικών
cpu	επεξεργαστής
compiler	μεταγλωττιστής
bidirectional	αμφίδρομο
AST	αφηρημένο συντακτικό δέντρο
HAN	ιεραρχικό δίκτυο προσοχής
DAN	βαθύ δίκτυο μέσων όρων
I/0	είσοδος/έξοδος
coding style	μορφή κώδικα
overlap	επικάλυψη
whitespaces	κενοί χαρακτήρες
indendation	εσοχή
style	μορφή
dataset	σύνολο δεδομένων
repository	αποθήκη κώδικα
loop	επαναληπτικός βρόγχος
dense layer	πυκνό στρώμα
batch normalization	κοινωνικοποίηση παρτίδας
classification	ταξινόμηση

Βιβλιογραφία

[1] Batch Normalization Wikipedia Article, https://en.wikipedia.org/wiki/Batch_normalization

[2] Abstract Syntax Tree Wikipedia Article, https://en.wikipedia.org/wiki/Abstract_syntax_tree

[3] Compilers Wikipedia Article,https://en.wikipedia.org/wiki/Compiler

[4] Clang Wikipedia Article,https://en.wikipedia.org/wiki/Clang

[5] https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[6] kNN Wikipedia Article,https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[7] SVM Wikipedia Article, https://en.wikipedia.org/wiki/Support_vector_machine

[8] Cortes, Corinna; Vapnik, Vladimir N. (1995). "Support-vector networks". Machine Learning. 20 (3): 273–297

[9] Altman, Naomi S. (1992). "An introduction to kernel and nearest-neighbor nonparametric regression". The American Statistician. 46 (3): 175–185

[10] Hochreiter, Sepp; Schmidhuber, Jürgen (1997-11-01). "Long Short-Term Memory". Neural Computation. 9 (8): 1735–1780

[11] Ioffe, Sergey; Szegedy, Christian (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"

[12] Santurkar, Shibani; Tsipras, Dimitris; Ilyas, Andrew; Madry, Aleksander (2018-05-29). "How Does Batch Normalization Help Optimization?"

[13] Matthews, B. W. (1975). "Comparison of the predicted and observed secondary structure of T4 phage lysozyme". Biochimica et Biophysica Acta (BBA) - Protein Structure. 405 (2): 442–451

[14] MCC Wikipedia Article,https://en.wikipedia.org/wiki/Matthews_correlation_coefficient#cite_note-Matthews1975-1

[15] F1 Score Wikipedia Article,https://en.wikipedia.org/wiki/F1_score

[16] Precision-Recall Wikipedia Article,https://en.wikipedia.org/wiki/Precision_and_recall

[17] Yang et al. "Hierarchical Attention Networks for Document Classification". Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies

[18] Iyyer et al. "Deep Unordered Composition Rivals Syntactic Methods for Text Classification". Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)

[19] Vinod Nair and Geoffrey Hinton (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. ICML

[20] Christiane Fellbaum. WordNet: An Electronic Lexical Database. 1998

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. CoRR,abs/1310.4546, 2013.

[23] https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

[24] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. n. http://cs231n.github.io/neural-networks-1

[25] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng (2014). Rectifier Nonlinearities Improve Neural Network Acoustic Models.

[26] Clevert, Djork-Arné; Unterthiner, Thomas; Hochreiter, Sepp (2015). "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)".

[27] http://alexlenail.me/NN-SVG/index.html

[28] Polosukhin, Illia; Kaiser, Lukasz; Gomez, Aidan N.; Jones,Llion; Uszkoreit, Jakob; Parmar, Niki; Shazeer, Noam; Vaswani, Ashish (2017-06-12). "Attention Is All You Need"

[29] Θεόδωρος Μ. Πίσσας. Αναγνώριση Ανθρώπινης Δράσης και Χειρονομιών χρησιμοποιώντας Συνελικτικά και Αναδρομικά Νευρωνικά Δίκτυα. Εργαστήριο Όρασης Υπολογιστών, Επικοινωνίας Λόγου και Επεξεργασίας Σημάτων, Εθνικό Μετσόβιο Πολυτεχνείο,Ιούλιος 2017. Διπλωματική εργασία.

[30] Γεράσιμος Σ. Χατζούδης. Decoupling Emergent Strategies in Task-Oriented Negotiation Dialogue Systems. Εργαστήριο Όρασης Υπολογιστών, Επικοινωνίας Λόγου και Επεξεργασίας Σημάτων, Εθνικό Μετσόβιο Πολυτεχνείο,Ιούλιος 2020. Διπλωματική εργασία.

[31] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Empirical Methods in Natural Language Processing (EMNLP), pages 1532–1543, 2014

[32] Zi Yin, Yuanyuan Shen. On the Dimensionality of Word Embedding. dvances in Neural Information Processing Systems 31 (NeurIPS 2018, Oral Presentation) 2018

[33] Y. Kim, “Convolutional neural networks for sentence classification,”arXiv preprint arXiv:1408.5882, 2014

[34] H. Ohashi and Y. Watanobe, ”Convolutional Neural Network for Classification of Source Codes,” 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, Singapore, 2019, pp. 194-200, doi: 10.1109/MCSoC.2019.00035.

[35] A. LeClair, Z. Eberhart and C. McMillan, ”Adapting Neural Text Classification for Improved Software Categorization,” 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, 2018, pp. 461-472, doi: 10.1109/ICSME.2018.00056.

[36] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019). Association for Computing Machinery, New York, NY, USA, 143–153. DOI:https://doi.org/10.1145/3359591.3359735

[37] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on. IEEE, 1157–1168.

[38] Feng, Yang & Zhou, Min & Tong, Xin. (2020). Imbalanced classification: an objective-oriented review.

[39] Bekkar, Mohamed & Djema, Hassiba & Alitouche, T.A.. (2013). Evaluation measures for models assessment over imbalanced data sets. Journal of Information Engineering and Applications. 3. 27-38

[40] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. Proc. ACM Program. Lang. 1, OOPSLA, Article 84 (October 2017), 28 pages. DOI:https://doi.org/10.1145/3133908

[41] Husain, Hamel & Wu, Ho-Hsiang & Gazit, Tiferet & Allamanis, Miltiadis & Brockschmidt, Marc. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.

[42] Babii, Hlib & Janes, Andrea & Robbes, Romain. (2019). Modeling Vocabulary for Big Code Machine Learning.

[43] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018). Association for Computing Machinery, New York, NY, USA, 95–105. DOI:https://doi.org/10.1145/3213846.3213848

[44] C. Cummins, P. Petoumenos, Z. Wang and H. Leather, ”Synthesizing benchmarks for predictive modeling,” 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Austin, TX, 2017, pp. 86-99, doi: 10.1109/CGO.2017.7863731.

[45] C. Cummins, P. Petoumenos, Z. Wang and H. Leather, ”End-to-End Deep Learning of Optimization Heuristics,” 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, 2017, pp. 219-232, doi: 10.1109/PACT.2017.24.

[46] Cvitkovic, Milan et al. Deep Learning On Code with an Unbounded Vocabulary. (2018).

[47] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: learning distributed representations of code. Proc. ACM Program. Lang. 3, POPL, Article 40 (January 2019), 29 pages. DOI:https://doi.org/10.1145/3290353

[48] Uri Alon, Shaked Brody, Omer Levy, & Eran Yahav (2019). code2seq: Generating Sequences from Structured Representations of Code. In International Conference on Learning Representations.

[49] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching Word Vectors with Subword Information

[50] Karl Pearson F.R.S. , 1901. LIII. On lines and planes of closest fit to systems of points in space. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 2(11), pp.559–572.

[51] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 41st International Conference on Software Engineering (ICSE '19). IEEE Press, 783–794. DOI:https://doi.org/10.1109/ICSE.2019.00086

[52] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15). JMLR.org, 1093–1102.

[53] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 163–174. DOI:https://doi.org/10.1145/3236024.3236085

[54] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. ACM Comput. Surv. 51, 4, Article 81 (September 2018), 37 pages. DOI:https://doi.org/10.1145/3212695

[55] Bui, Nghi & Jiang, Lingixao & Yu, Yijun. (2017). Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks.

[56] Mingming Lu, Dingwu Tan, Naixue Xiong, Zailiang Chen, & Haifeng Li. (2019). Program Classification Using Gated Graph Attention Neural Network for Online Programming Service.

[57] Miltiadis Allamanis, Marc Brockschmidt, & Mahmoud Khademi. (2017). Learning to Represent Programs with Graphs.

[58] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. 2018. Word embeddings for the software engineering domain. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18). Association for Computing Machinery, New York, NY, USA, 38–41. DOI:https://doi.org/10.1145/3196398.3196448

[59] Vasiliki Efstathiou and Diomidis Spinellis. 2019. Semantic source code models using identifier embeddings. In Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19). IEEE Press, 29–33. DOI:https://doi.org/10.1109/MSR.2019.00015

[60] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, & Ming Zhou. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages.

[61] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neuralnetworks over tree structures for programming language processing," in AAAI, vol. 2, no. 3, 2016, p. 4.