ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ
ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ

ΔΙΑΧΕΙΡΙΣΗ ΜΕΓΑΛΩΝ ΔΕΔΟΜΕΝΩΝ

ΔΗΜΗΤΡΗΣ ΖΕΡΚΕΛΙΔΗΣ 03400049
ΜΑΡΙΑ ΚΑΪΚΤΖΟΓΛΟΥ 03400052

Ιούλιος 2020

ΠΕΡΙΕΧΟΜΕΝΑ

1. ΕΙΣΑΓΩΓΗ

Η έκρηξη του όγκου δεδομένων τα τελευταία χρόνια μπορεί να γίνει εύκολα αντιληπτή αν παραθέσουμε μερικά εντυπωσιακά στοιχεία. Για παράδειγμα, από το 2012 και μετά έχουν παραχθεί 2.5 exabytes (2.5×260 bytes) δεδομένων· μέχρι το 2013 τα παραγόμενα δεδομένα ήταν 4.4 zettabytes. Από το 2013 μέχρι το 2020 τα παραγόμενα δεδομένα έφτασαν τα 44 zettabytes και προβλέπεται να φτάσουν τα 163 zettabytes μέχρι το 2025. Μάλιστα, το 90% των δεδομένων παράχθηκε τα τελευταία δύο χρόνια. Και μόνο αυτά τα στοιχεία είναι ικανά να υποδείξουν τη σημαντικότητα λοιπόν της διαχείρισης δεδομένων μεγάλης κλίμακας (big data).

Σκοπός της εργασίας αυτής είναι (α) η εξαγωγή πληροφορίας από δεδομένα μεγάλης κλίμακας (με MapReduce και sparkSQL) και (β) η εφαρμογή αλγορίθμων μηχανικής μάθησης μέσω της χρήσης του εργαλείου Apache Spark, ενός open-source κατανεμημένου υπολογιστικού cluster γενικού σκοπού.

Η αναφορά αποτελείται από τρία μέρη. Στο πρώτο δίνουμε κάποιες πληροφορίες σχετικά με την υποδομή που χρησιμοποιήθηκε για την εργασία. Το δεύτερο και το τρίτο αφορούν τα δύο μέρη - (α) και (β) - της εργασίας όπου και παραθέτουμε τη μεθοδολογία, τα αποτελέσματα συνοδευόμενα από σχολιασμούς και γραφήματα, καθώς και τον ψευδοκώδικα των προγραμμάτων μας.

1.1 Δεδομένα και ζητούμενα της εργασίας

Για κάθε μέρος της εργασίας παρουσιάζουμε το σύνολο δεδομένων που χρησιμοποιήθηκε και παραθέτουμε τα ερωτήματα που τέθηκαν.

1.1.1 Μέρος 1ο: (α) Εξαγωγή πληροφορίας

Για το πρώτο μέρος τα δεδομένα που χρησιμοποιήθηκαν αφορούν διαδρομές taxi στη Νέα Υόρκη από τον Ιανουάριο ως τον Ιούνιο του 2015 (13 εκατομμύρια διαδρομές) και έχουν μέγεθος 2GB ("Yellow Taxi Trip Data 2015-today"). Είναι χωρισμένα σε δύο datasets, το "yellow_tripdata_1m" και το "yellow_tripvendors_1m". Το πρώτο απαρτίζεται από τις εξής στήλες

- unique route id (StringType)
- Starting Date and time of route (DateTime)
- Ending Date and time of route (DateTime)
- Longitude - boarding (FloatType)
- Latitude - boarding (FloatType)
- Longitude - arrival (FloatType)
- Latitude - arrival (FloatType)
- Cost of route (FloatType)

και το δεύτερο από τις:

- unique route id (StringType)
- unique vendor id (StringType)

Στο πρώτο μέρος της εργασίας κληθήκαμε να χρησιμοποιήσουμε τις εξής δυο μεθόδους για την εξαγωγή πληροφοριών:

- με MapReduce κώδικα κάνοντας χρήση του RDD API του Spark
- με SparkSQL και του DataFrame API.

Τα ερωτήματα που τέθηκαν ήταν:

1. Μεταφόρτωση των csv αρχείων με τα δεδομένα στο HDFS
2. Υλοποίηση και εκτέλεση κώδικα για τα εξής δύο ερωτήματα-queries με MapReduce και SparkSQL:
   (Q1) Ποια είναι η μέση τιμή του γεωγραφικού μήκους και πλάτους επιβίβασης ανά ώρα έναρξης της διαδρομής; Ταξινομείστε το αποτέλεσμα με βάση την ώρα έναρξης σε αύξουσα σειρά και αγνοείστε dirty εγγραφές που προκαλούν προβληματικά αποτελέσματα (π.χ. Γεωγραφικό μήκος / πλάτος με μηδενική τιμή)
   (Q2) Για κάθε vendor, θεωρώντας ως απόσταση την απόσταση Haversine, βρείτε τη μέγιστη απόσταση διαδρομής που αντιστοιχεί σε αυτόν, καθώς και τον χρόνο στον οποίο αυτή εκτελέστηκε.
3. Μετατροπή των αρχείων κειμένου σε αρχεία Parquet. Φόρτωση των Parquet αρχείων ως Dataframes κι εκτέλεση του υποερωτήματος 2b. Εύρεση χρόνου μετατροπής αρχείων.
4. Δημιουργία διαγράμματος με το χρόνο εκτέλεσης των Q1 και Q2 με χρήση
   - MapReduce και csv αρχείου
   - SparkSQL και csv αρχείου
   - SparkSQL και parquet αρχείου

1.1.2 Μέρος 1ο: (β) Μελέτη του βελτιστοποιητή για την συνένωση δεδομένων

Στο μέρος αυτό μας ζητήθηκε να μελετήσουμε την επίδραση του βελτιστοποιητή στην εκτέλεση των ερωτημάτων συνένωσης που χρησιμοποιεί το SparkSQL απομονώνοντας τις 50 πρώτες γραμμές από το αρχείο με τις εταιρείες ταξί και εκτελώντας με SparkSQL ένα join πάνω στα 2 parquet αρχεία. Με αυτόν τον τρόπο μπορούμε να εντοπίσουμε την υλοποίηση join που χρησιμοποίησε το Spark της οποίας την επιλογή καλούμαστε να αιτιολογήσουμε. Επιπλέον, μας ζητήθηκε να ρυθμίσουμε το Spark χρησιμοποιώντας τις ρυθμίσεις του βελτιστοποιητή έτσι ώστε να μην επιλεγεί η υλοποίηση join που επιλέχθηκε προηγουμένως. Ομοίως, ζητείται ο εντοπισμός της νέας υλοποίησης join και η σύγκριση των δύο χρόνων εκτέλεσης των υλοποιήσεων.

1.1.3 Μέρος 2ο: Machine Learning - Κατηγοριοποίηση κειμένων

Το κομμάτι του Machine Learning αφορούσε την κατηγοριοποίηση κειμένων με χρήση του Apache Spark, αξιοποιώντας ένα πραγματικό σύνολο δεδομένων με παράπονα πελατών σε σχέση με οικονομικά προϊόντα και υπηρεσίες. Κάθε δεδομένο αποτελείται από τρία πεδία. Το πρώτο πεδίο αποτελεί την ημερομηνία που κατατέθηκε το σχόλιο, το δεύτερο την κατηγορία προϊόντος ή υπηρεσίας που αναφέρεται, ενώ το τρίτο είναι το σχόλιο. Τα δεδομένα έχουν συλλεχθεί από το 2011 μέχρι σήμερα και είναι βρίσκονται διαθέσιμα εδώ , ωστόσο εμείς χρησιμοποιήσαμε σύμφωνα με τις οδηγίες ένα υποσύνολο των δεδομένων που μας δόθηκε. Συνοπτικά τα ζητούμενα ήταν ο καθαρισμός των δεδομένων, η εξαγωγή χαρακτηριστικών, ο χωρισμός του dataset σε train και test set με stratified split, η εκπαίδευση ενός μοντέλου Perceptron στο train set και η εξαγωγή προβλέψεων στο test set. Όλα αυτά θα αναλυθούν στο κεφάλαιο 4

2. ΥΠΟΔΟΜΗ ΤΗΣ ΕΡΓΑΣΙΑΣ

Ο κώδικας εκτελέστηκε σε ένα Spark cluster το οποίο παρέχεται από την υπηρεσία Okeanos. Το cluster αποτελείται από δύο εικονικά μηχανήματα-κόμβους, master και slave, στους οποίους είναι εγκατεστημένο το Hadoop Distributed File System (HDFS) και το Spark Framework. Δημιουργήθηκε ένα τοπικό υποδίκτυο όπου συνδέθηκαν τα δύο μηχανήματα, οι private IP των οποίων χρησιμοποιήθηκαν για την επικοινωνία τους στα κατανεμημένα συστήματα. Η σύνδεση στο remote έγινε με ssh, η μεταφορά των αρχείων τοπικά προς το remote με scp και η μεταφορά στο hdfs με hdfs dfs -put <όνομα αρχείου>. Για την εκκίνηση του dfs ήταν απαραίτητη η εκτέλεση των shell scripts start-dfs.sh και start-all.sh, και αντίστοιχα των stop-dfs.sh και stop-all.sh για τον τερματισμό του.

Το link για το HDFS όπου βρίσκονται τα δεδομένα μας είναι το παρακάτω:

http://83.212.77.152:50070

3. ΜΕΡΟΣ 1ο: ΕΞΑΓΩΓΗ ΠΛΗΡΟΦΟΡΙΑΣ

Το κεφάλαιο αυτό αφορά την περιγραφή των λύσεων και σχολιασμούς για την εξαγωγή πληροφορίας και τη μελέτη του βελτιστοποιητή.

3.1 Προεπεξεργασία

Πριν από οποιοδήποτε querying στα δεδομένα -είτε με MapReduce είτε με SparkSQL- εφαρμόστηκε μια αρχική επεξεργασία στο dataset έτσι ώστε να απαλλαχθεί από ασυνεπή δεδομένα που θα προκαλούσαν προβληματικά αποτελέσματα. Συγκεκριμένα θελήσαμε να διασφαλίσουμε ότι κάθε εγγραφή που περιγράφει μια διαδρομή ικανοποιεί τις παρακάτω συνθήκες:

- η ώρα έναρξης είναι μικρότερη από την ώρα τερματισμού της
- το γεωγραφικό μήκος εκκίνησης είναι διαφορετικό από το γεωγραφικό μήκος τερματισμού της
- το γεωγραφικό πλάτος εκκίνησης είναι διαφορετικό από το γεωγραφικό πλάτος τερματισμού της
- το γεωγραφικο μήκος κυμάινεται στο διάστημα (-80, -71) και το γεωγραφικό μήκος στο (40, 45) -οι συντεταγμένες βρέθηκαν με το google maps για την πολιτεία της Νέας Υόρκης.

Το φιλτράρισμα αυτό εφαρμόστηκε στα δεδομένα με την built-in συνάρτηση filter() η οποία εφαρμόζεται σε ένα rdd ή ένα dataframe. Στην περίπτωση του MapReduce η filter() εφαρμόστηκε σε ένα rdd και λαμβάνει ως όρισμα μια συνάρτηση. Γράψαμε τη συνάρτηση filter_data() η οποία λαμβάνει ένα string δεδομένων, χωρίζει το string στο κόμμα με .split(",") παράγοντας μια λίστα, και χρησιμοποιώντας τα στοιχεία της λίστας επιστρέφει τις λογικές συνθήκες που απαιτούμε. Να σημειώσουμε ότι η filter() λειτουργεί ως python map και στέλνει στην filter_data() κάθε στοιχείο του rdd, δηλαδή strings. Επομένως το αποτέλεσμα είναι να πάρουμε ένα νέο rdd το οποίο έχει φιλτραριστεί κατά γραμμές με βάση τις συνθήκες που επιστρέφει η filter_data(). Στην περίπτωση του SparkSQL, η filter() εφαρμόστηκε πάνω σε ένα dataframe. Την τροφοδοτήσαμε απευθείας με λογικές συνθήκες υπό μορφή SQL εκφράσεων και αυτή εφάρμοσε τα αντίστοιχα φίλτρα στις γραμμές του dataframe, επιστρέφοντας το φιλτραρισμένο.

Οι συντεταγμένες που έχουμε διαλέξει αφορούν όπως αναφέρθηκε παραπάνω ολόκληρη την πολιτεία της Νεας Υόρκης. Γενικότερα, η εταιρεία NYC TAXI προσφέρει διαδρομές κι εκτός της πολιτείας στην κατηγορία των yellow taxi cabs με μια παραπάνω χρέωση ή με συμφωνία με τον οδηγό TAXI. Αυτες οι πληροφορίες μπορούν να φανούν στον παρακάτω σύνδεσμο:

https://www1.nyc.gov/site/tlc/passengers/taxi-fare.page

3.2 Χρήση MapReduce.

Περιγράφουμε τις μεθόδους που ακολουθήσαμε για τα queries 1 και 2 με χρήση MapReduce κώδικα

3.2.1 Query 1

Για να μπορέσουμε να υλοποιήσουμε την εφαρμογή μας είναι απαραίτητη η δημιουργία ενός περιβάλλοντος στο Spark. Αρχικά λοιπόν λάβαμε από την κλάση SparkSession ένα Session όπου θα τρέξει η εφαρμογή καλώντας το SparkSession.builder. Το builder είτε λαμβάνει ένα υπάρχον session είτε δημιουργεί ένα καινούριο αν δεν υπάρχει. Ονοματίσαμε την εφαρμογή "Q1_rdd" με .AppName() και δώσαμε εντολή στο builder να λάβει ή να δημιουργήσει το SparkSession με getOrCreate(). Τα παραπάνω συνοψίζονται στην εντολή spark = SparkSession.builder.appName("Q1_rdd").getOrCreate(). Εν συνεχεία δημιουργήσαμε ένα SparkContext αντικείμενο από το SparkSession το οποίο επέτρεψε τη σύνδεση με το υπολογιστικό cluster και τη δημιουργία RDDs και dataframes. Δημιουργήσαμε ένα rdd διαβάζοντας τα δεδομένα με sc.textFile() και εφαρμόσαμε τα φίλτρα που περιγράψαμε στην προηγουμενη παράγραφο.

Σκοπός αυτού του ερωτήματος ήταν να βρούμε το μέσο γεωγραφικό μήκος (γ.μ) και πλάτος (γ.π.) ανά ώρα διαδρομής και να ταξινομήσουμε τα αποτελέσματα σε αύξουσα σειρά με βάση την ώρα έναρξης. Επομένως πρόκειται για τρία βασικά βήματα. Το πρώτο βήμα ήταν να ομαδοποιήσουμε με έναν τρόπο τις εγγραφές σε ομάδες ακέραιων ωρών, δηλαδή 00, 01, ..., 23· το

δεύτερο βήμα να εξάγουμε τους μέσους όρους που ζητούνται και το τρίτο βήμα να εφαρμόσουμε αύξουσα ταξινόμηση στην ώρα έναρξης.

Για την ομαδοποίηση σε ακέραιες ώρες δημιουργήσαμε μια συνάρτηση, την extract_hour, η οποία λαμβάνει ως όρισμα την ώρα έναρξης της διαδρομής (datetime format) κι επιστρέφει μόνο την ακέραια ώρα (επίσης σε datetime format). Επεξεργαστήκαμε το filtered rdd με την εξής σειρά:

1. Εφαρμόσαμε map με συνάρτηση lambda δημιουργώντας τούπλες (ώρα έναρξης διαδρομής, (γ.μ. έναρξης διαδρομής, γ.π. έναρξης διαδρομής, 1)). Ως κλειδί κρατήσαμε την ώρα έναρξης της διαδρομής γιατί ως προς αυτήν θέλουμε να μετρήσουμε τους μέσους όρους. Το '1' στην τούπλα θα χρησιμεύσει για να μετρήσουμε πόσες εγγραφές αντιστοιχούν σε κάθε κλειδί.

2. Εφαρμόσαμε reduceByKey δημιουργώντας τούπλες (άθροισμα γ.μ., άθροισμα γ.π., πλήθος τιμών του συγκεκριμένου κλειδιού)

3. Κάναμε sorting ως προς το κλειδί, δηλαδή την ώρα έναρξης τς διαδρομής

4. Με map και lambda function διαιρέσαμε κάθε άθροισμα της τούπλας που πήραμε με το reduceByKey, με το πλήθος τιμών του κλειδιού έτσι ώστε να εξαθεί ο μέσος όρος με τελική τούπλα την (ώρα έναρξης διαδρομής, μέση τιμή γ.μ. έναρξης διαδρομής, μέση τιμή γ.π. έναρξης διαδρομής)

Το αποτέλεσμα αποθηκεύτηκε στη μεταβλητή res η οποία και τυπώθηκε. Ο συνολικός χρόνος εκτέλεσης ήταν 249.24917 δευτερόλεπτα.

Ακολουθεί ο ψευδοκώδικας για το query 1 με Map Reduce

```
map(key,value):
    #key: _
    #value: rdd row
    Field_list = split(value, ",")
    start_datetime = field_list[1]
    start_hour = extract_hour(start_datetime)
    start_longitude = Field_list[3]
    start_latitude = Field_list[4]
    emit (start_hour, (start_longitude, start_latitude, 1))

reduce(key,value):
    #key: start_hour
    #value: list of (start_longitude, start_latitude, 1)
    start_hour = key
    total_start_longitude = 0
    total_start_latitude = 0
    start_hour_frequency = 0
    for v in value:
        total_start_longitude += value[0]
        total_start_latitude += value[1]
        start_hour_frequency += value[2]
    emit (start_hour, (total_start_longitude, total_start_latitude, start_hour_frequency))

map(key,value):
    #key: _
    #value: (start_hour, (total_start_longitude, total_start_latitude, start_hour_frequency))
    start_hour = v[0]
    total_start_longitude = v[1][0]
    total_start_latitude = v[1][1]
    start_hour_frequency = v[1][2]
    avg_start_longitude = total_start_longitude / start_hour_frequency
    avg_start_latitude = total_start_latitude / start_hour_frequency
    emit (start_hour, avg_start_longitude, avg_start_latitude)
```

3.2.2 Query 2

Όπως και στο query 1, δημιουργήθηκε το κατάλληλο περιβάλλον στο Spark με τον ίδιο τρόπο. Στο ερώτημα αυτό ζητείται να βρεθεί η μέγιστη απόσταση haversine που αντιστοιχεί σε κάθε εταιρεία taxi (vendor) και ο χρόνος στον οποίον εκτελέστηκε. Επομένως για αυτό το query χρειαζόμαστε και το δεύτερο dataset που περιέχει το ID μιας διαδρομής και την αντίστοιχη vendor (π.χ. 369367789290,2, όπου 369367789290 είναι το ID της διαδρομής και 2 η vendor). Διαβάσαμε τα δύο csv αρχεία με τα δεδομένα των διαδρομών και τα δεδομένα των εταιρειών στα rdd trip_data και vendors_data. Στο trip_data εφαρμόσαμε τα φίλτρα που έχουμε περιγράψει.

Για να υπολογίσουμε την απόσταση Haversine και το χρόνο στον οποίο εκτελέστηκε δημιουργήσαμε δύο συναρτήσεις, την haversine() και την parseData(). Η haversine() δέχεται ως όρισμα τα scalars γ.μ. και γ.π. της έναρξης και του τερματισμού μιας διαδρομής, ό,τι δηλαδή είναι απαραίτητο για τον υπολογισμό της απόστασης haversine, την οποία και επιστρέφει. Η parseData() δέχεται ως όρισμα ένα string μιας διαδρομής και το μετατρέπει σε μια λίστα με τη χρήση του split(","). Υπολογίζει τη διάρκεια της διαδρομής σε δευτερόλεπτα αφαιρώντας σε datetime μορφή την ώρα έναρξης από την ώρα τερματισμού, και καταχωρεί το αποτέλεσμα στη μεταβλητή duration. Επίσης, καλεί τη συνάρτηση haversine() η οποία της επιστρέφει την αντίστοιχη απόσταση, την οποία και καταχωρεί στη μεταβλητή distance. Η parseData επιστρέφει το ID και την τούπλα (duration, distance).

Τα διαδοχικά βήματα που εκτελέσαμε επομένως ήταν:

1. φιλτράραμε τα δεδομένα trip_data
2. εφαρμόσαμε την parseData στα trip_data μέσω map και lambda, η οποία έστελνε κάθε γραμμή των δεδομένων στην parseData). Έτσι το trip_data rdd περιείχε τούπλες της μορφής ('ID', ('duration', 'distance')), δηλαδή είχε keys τα ID και values την τούπλα (duration, distance).
3. ενώσαμε με join τα rdd trip_data και vendors_data. Η ένωση έγινε στο κοινό attribute 'ID'. Η ένωση αυτή έδωσε ένα rdd της μορφής ('ID', (('duration', 'distance'), 'vendor'))
4. εφαρμόσαμε map-lambda και μετατρέψαμε το rdd στη μορφή ('vendor', 'duration', 'distance'). Το ID δεν ήταν πλέον χρήσιμο οπότε δεν το εντάξαμε, και κλειδί ορίσαμε τη vendor καθώς ως προς αυτή θέλουμε να κάνουμε υπολογισμούς.
5. κάναμε reduceByKey κάνοντας τα distances των εγγραφών και κρατώντας κάθε φορά την εγγραφή με το μεγαλύτερο distance.
6. ενοποιήσαμε τις τούπλες σε μία, δηλαδή τις δώσαμε τη μορφή ('vendor', 'distance', 'duration')

Ο χρόνος εκτέλεσης ήταν 506.59503 δευτερόλεπτα, περίπου 2.5 φορές μεγαλύτερος από ό,τι ο χρόνος εκτέλεσης του query 1. Αυτό εξηγείται από το γεγονός ότι είχαν να εκτελεστούν περισσότεροι υπολογισμοί (απόσταση haversine και διάρκεια διαδρομής) καθώς και η ένωση η οποία είναι γενικά μια ακριβή υπολογιστική διαδικασία.

Παραθέτουμε τον ψευδοκώδικα για το Map Reduce στο query 2.

```
#trip_data map
    map(key,values):
        #key: _
        #values: rdd row
        ID, (duration, distance) = parseData(value)
        emit (ID, (duration, distance))

#vendors_data map
    map(key,values):
        #key: _
        #values: rdd_row
        Field_list = split(value, ",")
        ID = Field_list[0]
        vendor = Field_list[1]
        emit (ID, vendor)
```

```
#joined trip and vendors data
    map(key,values):
        #key: vendor_id
        #values: (ID, ((duration, distance), vendor))
        vendor = value[1][1]
        duration = value[1][0][0]
        distance = value[1][0][1]
        emit (vendor, (duration, distance))


     reduce(key,values):
        #key: vendor
        #value = list of (duration, distance)
        vendor = key
        max_distance = 0
        duration = 0
        for v in value:
            if v[1] > max_distance:
                max_distance = v[1]
                duration = v[0]
        emit (vendor, (max_distance, duration))


    map(key,values):
        #key: vendor
        #values: (max_distance, duration)
        vendor = key
        max_distance = value[0]
        duration = value[1]
        emit (vendor, max_distance, duration)
```

3.3 Χρήση SparkSQL.

Περιγράφουμε τη μεθοδολογία που ακολουθήσαμε για τα queries 1 και 2 με χρήση Spark SQL κώδικα

3.3.1 Query 1

Όπως και με τη χρήση του MapReduce κώδικα, ήταν κι εδώ απαραίτητο να δημιουργήσουμε το κατάλληλο περιβάλλον στο Spark με το SparkSession.builder. Η δημιουργία ενός spark SQL dataset συνοδεύεται από ένα schema, το οποίο είναι η περιγραφή της δομής (structure) των δεδομένων που δημιουργούν το dataset, και μπορεί είτε να προκύψει έμμεσα κατά το runtime είτε να οριστεί από το/τη χρήστη/ρια και να είναι γνωστό κατά το compiling. Εμείς ορίσαμε το schema ορίζοντας ως FloatType όλα τα attributes εκτός από της ημερομηνίες έναρξης και τερματισμού της διαδρομής, που ορίστηκαν ως TimeStampType. Το schema είναι ουσιαστικά ένα StructType αντικείμενο που ενσαρκώνεται (instantiates) με μια λίστα StructField αντικειμένων. Το κάθε StructField μοντελοποιεί μια στήλη ενός dataframe. Για να δημιουργηθεί χρειάζονται τρία στοιχεία, (i) το όνομα της στήλης, (ii) το datatype των στοιχείων της, και, (iii) το αν δέχεται null τιμές (nullable property). Η nullable property ορίστηκε ως True.

Αφότου λοιπόν ορίσαμε το schema διαβάσαμε τα δεδομένα σε csv format και αποδώσαμε το schema με την αντίστοιχη μέθοδο, παίρνοντας το dataframe trip_data. Στη συνέχεια χρησιμοποιήσαμε τη συνάρτηση registerTempTable για να κατασκευάσουμε έναν προσωρινό πίνακα όπου θα τρέχαμε τις SQL εντολές. Η διάρκεια ζωής του στη μνήμη είναι όση και η διάρκεια του Spark session.

Επειδή όπως αναφέραμε στο query 1 μας ενδιαφέρει η μέση τιμή γεωγραφικού πλάτους και μήκους ανά ώρα έναρξης διαδρομής, χρειάστηκε και πάλι να επεξεργαστούμε τα στοιχεία της στήλης με την ώρα έναρξης ώστε να κρατήσουμε την ακρίβεια σε ακέραια ώρα (π.χ. 15:00). Αυτό το κάναμε με τη συνάρτηση withColumn του Spark, η οποία μπορεί να αξιοποιηθεί για μετονομασία, αλλαγή τιμής και μετατροπή του τύπου δεδομένων μιας στήλης, αλλά και για τη δημιουργία μιας νέας στήλης

στο dataframe. Εμείς αλλάξαμε τις τιμές της στήλης κρατώντας τις ακέραιες ώρες με την ενσωματωμένη συνάρτηση hour στο module pyspark.sql.functions. Σε αυτήν την τελευταία μορφή του dataframe εφαρμόσαμε τα εξής:

1. groupBy ως προς την -ακέραια- ώρα έναρξης της διαδρομής
2. aggregate στα σχηματισμένα group που έδωσε η groupBy, με τη συνάρτηση 'mean' στις στήλες γ.μ. και γ.π. έναρξης της διαδρομής. Έτσι έχουμε λάβει ένα dataframe με τρεις στήλες: την ώρα έναρξης της διαδρομής, το μέσο γ.μ. των διαδρομών για εκείνη την ώρα και το μέσο γ.π. των διαδρομών για εκείνη την ώρα
3. sorting ως προς τη στήλη με την ώρα έναρξης σε αύξουσα σειρά

Ο συνολικός χρόνος εκτέλεσης ήταν 73.81666 δευτερόλεπτα, κάτι περισσότερο από το 1/3 του αντίστοιχου χρόνου με το MapReduce.

3.3.2 Query 2

Για το δεύτερο query ακολουθήσαμε αρχικά τα ίδια βήματα για τη δημιουργία περιβάλλοντος στο Spark και των schemes με τα οποία τροφοδοτήσαμε το dataframe reader. Αυτή τη φορά ορίσαμε τις ημερομηνίες ως string, και όλες τις υπόλοιπες στήλες ως float. Διαβάσαμε με το reader τα δύο αρχεία και τα αποθηκεύσαμε στις μεταβλητές trip_data και vendors_data.

Γράψαμε και πάλι τη συνάρτηση haversine() κι επιπλέον τη συνάρτηση duration(), η οποία δέχεται ως όρισμα τις ώρες έναρξης και λήξης της διαδρομής κι επιστρέφει τη διάρκειά της σε λεπτά. Για το σκοπό αυτό χρησιμοποιήσαμε τη συνάρτηση datetime.strptime() που διαβάζει ένα string και το μετατρέπει σε datetime. Η duration() μετέτρεψε τις ώρες από strings σε datetimes και αφαίρεσε τα αποτελέσματα για να βρει και να επιστρέψει τη διάρκεια της διαδρομής. Ωστόσο, οι δύο αυτές συναρτήσεις είναι γραμμένες στην Python, ενώ εμείς θέλαμε να τις χρησιμοποιήσουμε με τη Spark SQL. Ένας τρόπος για να γίνει registered η συνάρτηση που ορίστηκε από το/τη χρήστη/ρια (User Defined Function) είναι να γράψουμε fuction_name_in_SparkSQL = udf(function_name_in_Python, datatype of output), το οποίο και κάναμε για να γίνουν registered οι συναρτήσεις μας.

Επόμενο βήμα ήταν να ενσωματωθούν στο dataframe οι δύο στήλες με την πληροφορία distance και duration, το οποίο έγινε και πάλι με τη χρήση της withColumn() συνάρτησης. Ακολούθησε ένωση των δύο dataframes, του trip_data και του vendors_data με inner join ως προς τη στήλη 'ID'. Το inner join σημαίνει πως στην ένωση κρατήθηκαν μόνο οι εγγραφές εκείνες που το 'ID' τους εμφανίζονταν και στα δύο dataframes. Έτσι προέκυψε το joined dataframe trip_joined.

Για την εκτέλεση του query ήταν καταλυτική η συμβολή της συνάρτησης Window(). Η συνάρτηση Window() εκτελεί έναν υπολογισμό σε ένα σύνολο γραμμών,το οποίο ονομάζεται Frame. Κάθε γραμμή δηλαδή είναι συνδεδεμένη με ένα συγκεκριμένο Frame. Αυτό δημιουργείται με τη μέθοδο partitionBy(), η οποία λαμβάνει ως όρισμα μια στήλη (ή ένα σύνολο στηλών) πάνω στην οποία γίνεται ένα partition. Στην περίπτωση μας δημιουργήσαμε το w = Window.partitionBy('vendor_id'), δηλαδή κάναμε partitioning ως προς τις εταιρείες ταξί που σημαίνει ότι όλες οι εγγραφές που έχουν ίδια vendor ανήκουν στο ίδιο Frame. Στη συνέχεια, προσθέσαμε στο dataframe μια στήλη με το όνομα'max_distance'η οποία περιείχε σε κάθε της γραμμή τη μέγιστη απόσταση των διαδρομών του Frame στο οποίο ανήκε. Δηλαδή, εφόσον υπάρχουν μόνο δυο vendors, κάθε γραμμή της στήλης 'max_distance' είχε τη μέγιστη απόσταση διαδρομής είτε του vendor 1 είτε το vendor 2, αναλόγως ποιο ήταν το vendor_id της γραμμής. Αυτό έγινε με τη χρήση του over clause με όρισμα το Window πάνω στο F.max('Distance'), όπου η max προέρχεται από το module functions του Pyspark. Αμέσως μετά κάναμε registerTempTable το καινούριο dataframe στο οποίο και εφαρμόσαμε ένα spark SQL query. Το query έκανε select τις στήλες vendor_id, Distance και Duration φιλτράροντας με where clause τις γραμμές των οποίων το distance ήταν το max_distance. Ο χρόνος εκτέλεσης ήταν 358.27125 δευτερόλεπτα.

3.4 Εκτέλεση των queries με αρχεία parquet και Spark SQL

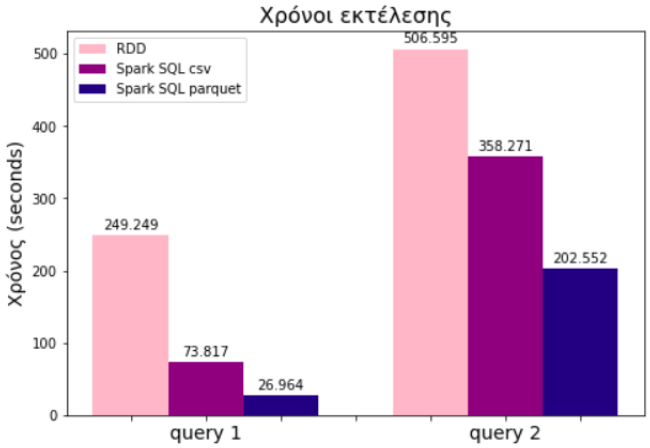
Για τη μετατροπή των αρχείων csv σε parquet, δημιουργήσαμε το κατάλληλο περιβάλλον στο Spark με SparkSession.builder, ορίσαμε το scheme (όπως στα queries με Spark SQL) για το κάθε αρχείο και το διαβάσαμε αποθηκεύοντάς το σε ένα dataframe. Η μετατροπή έγινε απλώς με την εντολή:

dataframe_name.write.parquet(destination_path + "parquet_file_name").

Ο χρόνος που χρειάστηκε για τη μετατροπή τους ήταν 76.80967 δευτερόλεπτα.

Το query 1 με SparkSQL και το parquet αρχείο 26.96384 δευτερόλεπτα. Το query 2 με Spark SQL και το parquet αρχείο έτρεξε σε 202.55209 δευτερόλεπτα.

Παρακάτω παραθέτουμε ένα barplot με τους χρόνους εκτέλεσης των queries με τη χρήση διαφορετικών API και τύπων αρχείων έτσι ώστε να φανούν καλύτερα οι διαφορές τους.



Όπως φαίνεται από το παραπάνω διάγραμμα το spark dataframe API με τη χρήση csv αρχείων είναι πιο γρήγορο για το 1ο query κατά 3.5 φορές ενώ για το 2ο κατά 1.5 φορά. Το συγκεκριμένο API έχει ως χαρακτηριστικό το ότι αποθηκεύει εκτός σωρού (off-heap) σε δυαδική αναπαράσταση τα δεδομένα, εξοικονομώντας με αυτόν τον τρόπο παραπάνω μνήμη καθώς αφαιρεί τις άχρηστες πληροφορίες. Επίσης, αποφεύγεται το Java Serialization καθώς είναι γνωστό το schema των δεδομένων. Επιπρόσθετα, μια πολύ κομβική λειτουργία του συγκεκριμένου API είναι το Optimized Execution Plans. Το συγκεκριμένο χαρακτηριστικό βελτιστοποιεί τα queries που γράφουμε στα dataframes σε μορφή map reduce για RDD για ταχύτερη εκτέλεση. Ένα ακόμα χαρακτηριστικό του spark dataframe API είναι το flexibility, δηλαδή μπορεί να διαχειριστεί διάφορους τύπους αρχείων όπως csv ή parquet. Στη δική μας περίπτωση, δοκιμάσαμε τη χρήση parquet και παρατηρήσαμε πως για το 1ο ερώτημα το querying είναι σχεδόν 10 φορές πιο γρήγορο σε σχέση με τα RDD και σχεδόν 3 φορές σε σχέση με το dataframe-csv. Στο δέυτερο ερώτημα οι διαφορές είναι πιο μικρές, καθώς το dataframe-parquet είναι ταχύτερο κατά 2.5 φορές σε σχέση με το RDD και 1.7 φορές σε σχέση με το dataframe-csv.

Η απόδοση που πήραμε με τα parquet files οφείλεται στο ότι για τα ερωτήματα της εργασίας συμφέρει περισσότερο να αποθηκεύονται τα δεδομένα κατα στήλες (Column Oriented Storage). Γενικά, το Row Oriented Storage δηλαδή αποθήκευση των δεδομένων κατά γραμμή δε συμφέρει αν δε χρειάζονται στα queries όλες οι στήλες αλλά χρειάζονται πολλές γραμμές. Στα συγκεκριμένα ερωτήματα έχουμε πάρα πολλές γραμμές και χρειαζόμαστε λίγες στήλες από αυτές. Επομένως, είναι λογικό η χρήση των parquet αρχείων όπου τα δεδομένα είναι κατά στήλες αποθηκευμένα να αποδίδουν καλύτερα.

3.5 Μελέτη του βελτιστοποιητή.

Όταν γράφουμε έναν κώδικα στο Spark ο οποίος είναι έγκυρος, το Spark το μετατρέπει σε ένα λογικό πλάνο (logical plan). Το logical plan μπορούμε να το φανταστούμε σαν ένα tree representation ενός query. Φέρει -σε αφαιρετικό επίπεδο- την πληροφορία για το τι πρόκειται να γίνει αλλά όχι για το πώς ακριβώς θα γίνει. Συνίσταται από

- relational operators
  - join, filter, project... (που αναπαριστούν transformations των dataframes)
- εκφράσεις
  - transformations στηλών, συνθήκες φιλτραρίσματος, συνθήκες ένωσης...

Το λογικό πλάνο είναι μια αφαίρεση όλων των μετατροπών που πρέπει να γίνουν και δεν έχει να κάνει με το Master Node ή το Worker Node (αλλιώς executor). Το αποτέλεσμα του logical planning καταλήγει σε ένα optimized logical plan, το οποίο όμως είναι πολύ αόριστο (abstract) και οι executors δεν το κατανοούν. Πρέπει να γίνει μετατροπή σε αυτό που ονομάζουμε physical plan. Το physical plan λειτουργεί σαν γέφυρα μεταξύ του logical plan και των RDD. Έχει μια αντίστοιχη μορφή δέντρου όπως το logical plan και περιέχει πιο σαφείς πληροφορίες για το πως θα εκτελεστεί το πλάνο (δηλαδή συγκεκριμένους αλγόριθμους). Για παράδειγμα, το logical plan περιέχει την εκτέλεση ενός join και το physical plan τον ακριβή αλγόριθμο, π.χ. broadcasthashjoin ή sortmergejoin κλπ. Η επιλογή του αλγορίθμου γίνεται με τη σύγκριση διάφορων στρατηγικών

ως προς κάποια κριτήρια, όπως ο χρόνος εκτέλεσης και οι πόροι που απαιτούνται. Επομένως, όταν επιλέγεται η optimized στρατηγική, και με την πρόσθεση κάποιων κανόνων, προκύπτει το execution plan, δηλαδή η τελική μορφή του physical plan, το οποίο στο PySpark μπορούμε να το πάρουμε με τη μέθοδο explain() πάνω σε ένα dataframe.

Για να δούμε το πλάνο εκτέλεσης λοιπόν που επέλεξε το SparkSQL, διαβάσαμε το parquet αρχείο με τα δεδομένα των διαδρομών (trip_data) και κάναμε join με τις 50 πρώτες εγγραφές των δεδομένων των εταιρειών ταξί (επίσης το parquet αρχείο και με limit(50)). Στο joined dataframe εφαρμόσαμε τη μέθοδο explain η οποία μας έδωσε ότι ο αλγόριθμος που επιλέχτηκε για το join ήταν το BroadcastHashJoin. Αυτό εξηγείται από το γεγονός ότι όταν ένα από τα δύο dataframes είναι αρκετά μικρό -τόσο ώστε να χωρέσει στη μνήμη εύκολα- μπορεί να γίνει broadcasted σε όλους τους executors του cluster όπου βρίσκεται το μεγαλύτερο dataframe και στη συνέχεια να εκτελεστεί ένα hash join. Η κεντρική ιδέα του hash join είναι ότι φορτώνεται το μικρό dataset στη μνήμη σε ένα hashmap, δηλαδή σε μια δομή key-value, και για κάθε εγγραφή του μεγάλου dataset αναζητείται το αντίστοιχο key στο hashmap. Η αναζήτηση γίνεται σε σταθερό χρόνο O(1). Στην εκτέλεση αυτή μεθοδεύονται μόνο οι mappers και δε γίνεται shuffling των δεδομένων στο cluster, κάτι που επιταχύνει πολύ τη διαδικασία του join.

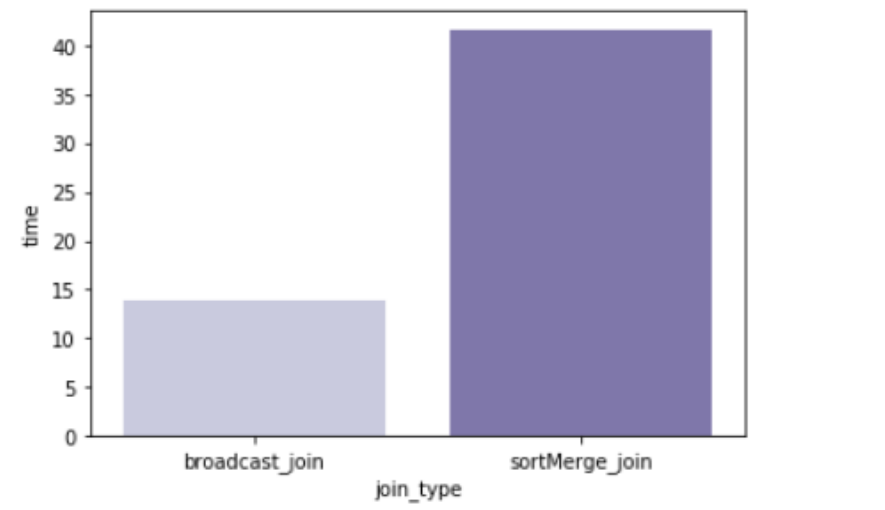
Ο χρόνος εκτέλεσης με BroadcastHashJoin ήταν 13.90393 δευτερόλεπτα.

Όταν απενεργοποιήσαμε την επιλογή του BroadcastJoin και πήραμε και πάλι με explain() το physical plan, ο αλγόριθμος που επιλέχθηκε ήταν το SortMergeJoin. Το SortMergeJoin είναι μια στρατηγική που λειτουργεί όταν τα κλειδιά μπορούν να γίνουν sorted και μάλιστα κάνει αυτήν την υπόθεση όταν εκτελείται. Μπορούμε να τη συνοψίσουμε σε τρία στάδια:

1. Shuffle: repartition των δύο πινάκων ως προς το join-κλειδί κατά μήκος των partitions στο cluster. Έτσι σε κάθε partition όλες οι εγγραφές και από τους δύο πίνακες έχουν το ίδιο κλειδί.
2. Sort: sorting των δεδομένων σε κάθε partition παράλληλα.
3. Merge: ένωση των δεδομένων στα partitions με iteration πάνω στα στοιχεία του dataset. Με βάση το κλειδί του join ενώνονται οι γραμμές που έχουν το ίδιο value.

Το shuffling και το sorting προσθέτουν επιπλέον κόστος, κάτι που αποτυπώθηκε και στα αποτελέσματα που λάβαμε, καθώς ο χρόνος εκτέλεσης με SortMergeJoin ήταν 41.64886 δευτερόλεπτα, μεγαλύτερος από αυτόν στο BroadcastHashJoin.

Παρακάτω βλέπουμε ένα barchart με τους 2 χρόνους. Παρατηρούμε πως είναι σχεδόν 3 φορές πιο γρήγορο το broadcastJoin στη συγκεκριμένη περίπτωση για τους λόγους που αναφέρθηκαν παραπάνω.



4. ΜΕΡΟΣ 2ο: MACHINE LEARNING - ΚΑΤΗΓΟΡΙΟΠΟΙΗΣΗ ΚΕΙΜΕΝΩΝ

Το κεφάλαιο αυτό αφορά την περιγραφή των λύσεων και σχολιασμούς για την εφαρμογή αλγορίθμων Machine Learning με Apache Spark στο πρόβλημα της κατηγοριοποίησης κειμένων. Χρησιμοποιήσαμε το σύνολο δεδομένων που περιγράφεται στην ενότητα 1.1.3. και υπενθυμίζουμε ότι το πρώτο πεδίο αποτελεί την ημερομηνία που κατατέθηκε το σχόλιο, το δεύτερο την κατηγορία προϊόντος ή υπηρεσίας που αναφέρεται, ενώ το τρίτο είναι το σχόλιο. Το συνολικό πλήθος κατηγοριών (labels) είναι 18.

Στη συνέχεια περιγράφουμε την προεπεξεργασία που ζητήθηκε να κάνουμε στο αρχικό rdd όπου διαβάσαμε τα δεδομένα, τη μέθοδό μας για εξαγωγή χαρακτηριστικών και την εκπαίδευση ενός μοντέλου Perceptron πάνω στα τελικά δεδομένα.

4.1 Προεπεξεργασία.

Μια πρώτη επεξεργασία ήταν ο καθαρισμός των δεδομένων από μή χρήσιμες ή μή έγκυρες εγγραφές. Συγκεκριμένα, εξασφαλίσαμε ότι: (1) οι γραμμές ξεκινάνε με "201", δηλαδή με καταχώρηση ημερομηνίας όπως και προσδιαγράφεται, (2) οι εγγραφές περιέχουν σχόλιο του/της χρήστη/ριας.

Για τον αρχικό καθαρισμό κατασκευάσαμε δύο συναρτήσεις, την filter_data() και την parse_data(), τις οποίες και δώσαμε ως όρισμα στις ενσωματωμένες στο spark συναρτήσεις filter() και map() αντίστοιχα. Η filter() πέρασε γραμμή-προς-γραμμή τα δεδομένα στην filter_data(), η οποία

- μετέτρεψε τη γραμμή από string σε μια λίστα (values) με 3 strings, την ημερομηνία, το label και την πρόταση
- επέστρεψε False για τις γραμμές οι οποίες δεν είχαν και τα τρία πεδία συμπληρωμένες, ελέγχοντας αν το μήκος της λίστας value είναι 3
- επέστρεψε False για τις γραμμές που η ημερομηνία δεν ξεκινούσε με "201"

Δηλαδή, η filter_data() επέστρεψε για κάθε γραμμή True ή False, και η filter() διατήρησε από το αρχικό rdd μόνο τις γραμμές για τις οποίες δέχτηκε True.

Η parse_data() μετέτρεπε ομοίως τις string γραμμές σε λίστα τριών strings, και με τη χρήση της βιβλιοθήκης Re της Python κρατούσε από την πρόταση μόνο τις λέξεις εκείνες που αποτελούνταν αποκλειστικά από γράμματα της αλφαβήτας και πετώντας stopwords, τα οποια είναι λέξεις χωρίς σημασία, όπως άρθρα, σύνδεσμοι κλπ. Η parse_data() επέστρεφε το label και την περικομένη πρόταση ώστε να γίνει το αντίστοιχο map στο rdd.

Επιπλέον, δημιουργήσαμε ένα "λεξικό" με τις 50 πιο συχνές λέξεις. Ένα flatMap στα δεδομένα που ήταν της μορφής (label, πρόταση) έκανε emit ένα rdd με τις λέξεις μόνο των προτάσεων. Έπειτα μια filter() αφαίρεσε τα stopwords, και μια map() δημιούργησε tuples της μορφής (word,1). Η reduce που παρέλαβε το αποτέλεσμα υπολόγισε με βάση το κλειδί που ήταν η λέξη τη συνχότητα εμφάνισης της κάθε μίας, αθροίζοντας τα values, δηλαδή το 1. Έγινε sorting στο αποτέλεσμα κι έτσι επιλέχθηκαν οι 50 πρώτες σε συχνότητα λέξεις που αποθηκεύτηκαν στη λίστα unique_words. Δεδομένου ότι η λίστα αυτή είχε πολύ μικρό μέγεθος, έγινε broadcasted σε όλους τους workers στο cluster για να επιταχυνθεί η παράλληλη εργασία. Παρακάτω παρατίθεται ο ψευδοκώδικας της δημιουργίας της unique_words.

```
flatMap(key,value):
    #key: _
    #values: rdd row
    Words_list = value[1]
    Field_list = split(Words_list, " ")
    emit (flatten(Field_list))

map(key,value):
    #key: _
    #value: word
    emit (word, 1)
```

```
reduce(key,value):
    #key: word
    #value: list of 1s
    word = key
    key_counter = 0
    for v in value:
        key_counter += v
    emit (word, key_counter)

map(key,value):
    #key: _
    #value: word
    emit (word)
```

4.2 Εξαγωγή χαρακτηριστικών tf-idf

Αρχικά μετασχηματίσαμε το rdd για να πάρουμε ένα καινούριο με πληροφορία χρήσιμη για τον υπολογισμό του tf. Καταλήξαμε να πάρουμε τούπλες της μορφής (word, (label, row_index, frequency of the word in the sentence/row)). Ακολουθήσαμε τα εξής βήματα:

- map στο rdd data το οποίο επέστρεψε για κάθε γραμμή την τούπλα (label, λίστα με τις λέξεις της πρότασης)
- map στο αποτέλεσμα έτσι ώστε να διαγραφούν από την παραπάνω λίστα οι λέξεις που δεν ανήκουν στις 50 unique words που κρατήσαμε. Επομένως η emitted τούπλα είχε την ίδια μορφή με πριν αλλά ήταν περικεκομμένη.
- εφαρμογή filter() έτσι ώστε να πεταχούν οι εγγραφές που δεν είχαν και τα τρια πεδία συμπληρωμένα
- εφαρμογή της zipWithIndex() συνάρτησης για να επιμηκυνθεί η τούπλα με το δείκτης γραμμής. Οι καινούριες τούπλες είχαν τη μορφή ((label, λίστα με τις λέξεις στην πρόταση), row index)
- flatMap που για κάθε λέξη σε μια προταση επέστρεψε μια τούπλα της μορφής ((λέξη, label, row index), 1).
- reduceByKey στο emitted αποτέλεσμα του flatMap που αθροίζει τα values-μονάδες και δίνει έτσι τη συχνότητα εμφάνισης του κλειδιού (λέξη, label, row index). Άρα οι emitted τούπλες έχουν τη μορφή ((λέξη, label, row index), συχνότητα εμφάνισης στην πρόταση)
- map στο αποτέλεσμα για να πάρουμε την τελική μορφή του rdd που είναι (λεξη, (label, row index, συχνοτητα εμφανισης λεξης στην προταση της)). Δηλαδή αλλάξαμε το κλειδί ορίζοντάς το να είναι η λέξη, και τα υπόλοιπα το value-tuple.

Παραθέτουμε τον ψευδοκώδικα. Να σημειωθεί ότι μετά τα δύο πρώτα map παρεμβλήθηκε filter και zipWithIndex, για αυτό και παρατηρείται μια ασυνέπεια ως προς το emitted output και το input μεταξύ του δεύτερου map και του flatMap.

```
map(key,value):
    #key: _
    #value: rdd row
    label = value[0]
    row = value[1]
    Field_list = split(row, " ")
    emit (label, Field_list)

map(key,value):
    #key: _
    #value: (label, Field_list)
    label = value[0]
    Field_list = value[1]
    Field_list_truncated = [word for word in Field_list if word in unique words]
    emit (label, Field_list_truncated)
```

```
flatMap(key,value):
    #key: _
    #value: ((label, list with words in row), row index)
    label = value[0][0]
    Field_list = value[0][1]
    row_index = value[1]
    for word in Field_list:
        emit ((word, label, row_index), 1)

reduce(key,value):
    #key: (word, label, row_index)
    #value: list of 1s
    frequency = 0
    label = key[1]
    word = key[0]
    row_index = key[2]
    for v in value:
        frequency += value
    emit ((word, label, row_index), frequency)

map(key,value):
    #key: _
    #value: ((word, label, row_index), frequency)
    word = value[0][0]
    label = value[0][1]
    row_index = value[0][2]
    frequency = value[1]
    emit (word, (label, row_index, frequency))
```

Για τον υπολογισμό του idf ξεκινήσαμε κάνοντας flatMap στο rdd που είχε τη μορφή ((label, [λέξεις στην πρόταση]), row index). Στο flatMap χρησιμοποιήσαμε set της λίστας με τις λέξεις της πρότασης, γιατί μας ενδιέφερε απλώς το αν η λέξη εμφανίζεται στην πρόταση και όχι πόσες φορές. Επομένως, για μια πρόταση με K μοναδικές λέξεις πήραμε K τούπλες της μορφής (word, 1). Έτσι αν η λέξη "even" για παράδειγμα εμφανίζεται σε 16 συνολικά προτάσεις στο document πήραμε 16 τούπλες ("even", 1). Στη συνέχεια με reduceByKey υπολογίσαμε σε πόσες προτάσεις του document εμφανίζεται κάθε λέξη. Τέλος, με map υπολογίσαμε το idf κάθε λέξης διαιρώντας το πλήθος των προτάσεων στο document (το είχαμε αποθηκεύσει πιο πριν σε μεταβλητή εκτελώντας count()) με το πλήθος των προτάσεων στις οποίες η λέξη εμφανίζεται. Οι τελικές τούπλες είχαν τη μορφή (word, idf).

```
flatMap(key,value):
    #key: _
    #value: ((label, [words in row]), row index)
    Field_list = set(value[0][1])
    for word in Field_list:
        emit (word, 1)

reduce(key,value):
    #key: word
    #value: list of 1s
    counter = 0
    word = key
    for v in value:
        counter += value
    emit(word, counter)

map(key,value):
    #key: _
    #value: (word, number of rows in which word appears)
```

```
    number_of_docs = count(data)  # already calculated value
    word = value[0]
    frequency = value[1]
    emit (word, log(number_of_docs/frequency)
```

Για την τελική επεξεργασία ενώσαμε το tf rdd με το idf rdd με join παίρνοτας ένα rdd με τούπλες (word, ((label, row index, συχνότητα εμφάνισης της λέξης στην πρόταση με δείκτη row index), idf).

```
map(key,value):
    #key: _
    #value: (word, ((label, row index, συχνότητα εμφάνισης της λέξης
    # στην πρόταση με δείκτη row index), idf)
    word = value[0]
    label = value[1][0]
    row_index = value[1][1]
    freq = value[1][2]
    idf = value[2]
    # uwords είναι το λεξικό που έχουμε ήδη εξάγει και αποθηκεύσει
    index_in_lexicon = index(uwords.value.index, word)
    emit (index_in_lexicon, ((label, row_index, freq), idf))

map(key, value):
    #key: _
    # value: (index_in_lexicon, ((label, row_index, freq), idf))
    index_in_lexicon = v[0]
    label = value[1][0][0]
    row_index = value[1][0][1]
    freq = value[1][0][2]
    idf = value[1][1]
    emit ((row_index, label), (index_in_lexicon, freq, idf))

map(key, value):
    #key: _
    # value: ((row_index, label), (index_in_lexicon, freq, idf))
    row_index = value[0][0]
    label = value[0][1]
    index_in_lexicon = list(value[1][0])
    freq = list(value[1][1])
    idf = list(value[1][2])
    emit ((row_index, label), (index_in_lexicon, freq, idf))

reduce(key, value):
    #key: (row_index, label)
    # value: list of (index_in_lexicon, freq, idf)
    row_index = key[0]
    label = key[1]
    index_in_lexicon = value[0]
    freq = value[1]
    idf = value[2]
    indices_list = list()
    freq_list = list()
    idf_list = list()
    for v in values:
        indices_list += index_in_lexicon
        freq_list += freq
        idf_list += idf
    emit ((row_index, label), (indices_list, freq_list, idf_list))
```

```
map(key, value):
    #key: _
    # value: ((row_index, label), (indices_list, freq_list, idf_list))
    row_index = value[0][0]
    label = value[0][1]
    indices_list = value[1][0]
    freq_list = value[1][1]
    idf_list = value[1][2]
    max_freq = max(freq_list)
    tf_list = list()
    for f in freq_list:
        append(tf_list, f/max_freq)
    emit ((row_index, label), (indices_list, tf_list, idf_list))

map(key, value):
    #key: _
    # value: ((row_index, label), (indices_list, tf_list, idf_list))
    label = value[0][1]
    lexicon_size = 50
    indices_list = value[1][0]
    tf_list = value[1][1]
    idf_list = value[1][2]
    tfidf_list = hadamard_product(tf_list, idf_list)
    ind_tfidf_tuples_list = list()
    for ind, tfidf in zip(indices_list, tfidf_list):
        append(return_list, (ind, tfidf)
    emit (label, lexicon_size, ind_tfidf_tuples_list)

map(key, value):
    #key: _
    # value: (label, lexicon_size, ind_tfidf_tuples_list)
    label = value[0]
    lexicon_size = value[1]
    ind_tfidf_tuples_list_sorted_by_index_in_lexicon = sort(value[2], by value[2][0])
    emit (label, lexicon_size, ind_tfidf_tuples_list_sorted_by_index_in_lexicon)

map(key, value):
    #key: _
    # value: (label, lexicon_size, ind_tfidf_tuples_list_sorted_by_index_in_lexicon)
    label = value[0]
    lexicon_size = value[1]
    ind_tfidf_tuples_list_sorted_by_index_in_lexicon = value[2]
    indices_list = list()
    tfidf_list = list()
    for i, tfidf in zip(ind_tfidf_tuples_list_sorted_by_index_in_lexicon):
        append(indices_list, i)
        append(tfidf_list, tfidf)
    emit (label, lexicon_size, indices_list, tfidf_list)
```

Στην παρακάτω εικόνα φαίνεται το output που αντιστοιχεί σε 5 προτάσεις του document και έχει τη μορφή:

(label,N,(ind1,ind2,...,indk),(tdidf1,tfidf2,...,tfidfk))

```
('Mortgage', 50, [4, 5, 9, 10, 11, 14, 16, 20, 25, 26, 29, 34, 35, 36, 39, 41, 49], [1.5173549646931856, 0.7777333
284253185, 0.8544062422352126, 0.7646471900037972, 0.7715629477496364, 0.7783573388778917, 0.9761484356488878, 1.0
50871547063804, 1.4970595167873013, 0.9930948495997832, 0.93266869202192, 1.068075278024588, 1.015104253003259, 1.
008469042402106, 1.3261363149177803, 1.1275624362358958, 1.059563190204051])

('Mortgage', 50, [5, 10, 14, 16, 17, 23, 24, 25, 34, 47, 49], [0.960729405701864, 1.070506066005316, 0.90808356202
42069, 1.138839841590369, 0.9749312178656413, 1.041237298125056, 1.0037106771629942, 2.2061929721076017, 1.1682073
353393931, 1.272538879183559, 1.2361570552380596])

('Debt collection', 50, [3, 6, 8, 14, 23, 25, 27, 42, 46, 49], [1.1730611156700517, 0.9586568884298988, 1.03139022
92260002, 0.8717602195432387, 1.1661857739000625, 1.323715783264561, 1.105093611128405, 1.088378676095442, 2.16983
65566224522, 1.1867107730285371])

('Credit reporting credit repair services or other personal consumer reports', 50, [1, 2, 3, 10, 17, 18, 19, 24, 3
2, 34, 36, 42, 49], [0.8680906666228299, 0.6871955347227067, 0.8146257747708692, 1.4273414213404214, 0.97493121786
56413, 1.3135027241393864, 1.332379828721337, 1.0037106771629942, 1.284181541560129, 1.1682073353393931, 1.1765472
161357906, 1.1337277875994187, 1.2361570552380596])

('Mortgage', 50, [4, 5, 6, 10, 13, 22, 23, 24, 25, 26, 29, 30, 31, 41, 47, 49], [1.5173549646931856, 0.74723398221
2561, 1.3314679005970818, 0.9515609475602809, 1.1226013695800272, 1.0197132532010305, 1.1106531180000596, 0.936796
6320187946, 1.470795314738401, 1.0813699473419862, 1.0155725757572018, 1.1997664113751378, 1.2150480763965898, 1.2
277902083457535, 1.6967185055780785, 1.1537465848888557])
```


4.3 Προετοιμασία Training/Test set

Σε αυτό το σημείο της εργασίας, έχουμε ήδη υπολογίσει τη μετρική tf-idf και σχηματίσει ενα RDD στη μορφή

(label,N,(ind1,ind2,...,indk),(tdidf1,tfidf2,...,tfidfk))

όπου τα indexes είναι ταξινομημένα από το μικρότερο στο μεγαλύτερο. Αυτό είναι απαραίτητο καθώς στη συνέχεια μετασχηματίσαμε το RDD έτσι ώστε να περιέχει tuples με κλειδί το label και value ένα SparseVector με μέγεθος όσο το μέγεθος του λεξικού, το οποίο θα έχει τη μορφή SparseVector(Lexicon_size, index<i>:tfidf<i>). Τέλος, πραγματοποιήθηκε η μετατροπή από RDD σε Spark Dataframe για να χρησιμοποιηθεί στην εκπαίδευση ενός μοντέλου όπως παρουσιάστηκε στο εργαστήριο του μαθήματος.

Χρησιμοποιήσαμε τη συνάρτηση StringIndexer για να μετατρέψουμε τα labels από strings σε αριθμητικά με τιμές από το 0 εώς τον αριθμό_κλάσεων - 1. Με αυτόν τον τρόπο τα δεδομένα απέκτησαν τη μορφή που απαιτείται για την εκπαίδευση ενός μοντέλου.

Επόμενο βήμα ήταν ο διαχωρισμός των δεδομένων σε training set και test set με stratified τρόπο. Το stratification διασφαλίζει πως από κάθε κλάση θα πάρουμε το ίδιο ποσοστό δεδομένων για το training, εν προκειμένω 80%, και έτσι θα λάβουμε ένα νέο dataframe με το 80% των αρχικών δεδομένων. Εδώ πρέπει να σημειωθεί ότι στο Dataframe που είχαμε υπήρχαν αρκετά duplicate values, κάτι που οφείλεται στο ότι είχαμε ένα λεξικό με λίγες λέξεις (50) και μεγάλος πλήθος εγγραφών, επομένως προέκυψαν ίδια feature vectors με το ίδιο labeling. Αυτό μας οδήγησε στο να αφαιρέσουμε τα duplicates από το dataset μας με την εντολή dropDuplicates() πριν κάνουμε το διαχωρισμό σε train και test set.

Parameters: col – column that defines strata fractions – sampling fraction for each stratum. If a stratum is not specified, we treat its fraction as zero.

Έπειτα καλέσαμε τη συνάρτηση sampleBy η οποία παίρνει ως παραμέτρους το 'col', μια στήλη του dataframe που ορίζει τα strata, δηλαδή τις διαφορετικές κλάσεις ταξινόμησης, και το 'fractions', που καθορίζει το ποσοστό δειγμάτων που θα ληφθεί από κάθε κλάση. Στο 'col' δώσαμε ως παράμετρο τη στήλη "label" και στο fractions ένα Dictionary της μορφής class_i : 0.8. Αυτό δηλώνει πως για κάθε κλάση θα πάρει το 80% της και έτσι θα καταλήξουμε σε ένα training set που θα είναι το 80% του συνολικού dataset. Το test set το λάβαμε με την εντολή subtract η οποία καλείται από το ολοκληρωμένο dataset και δέχεται ως παράμετρο το training set. Η συγκεκριμένη εντολή ουσιαστικά επιστρέφει τα δεδομένα από το ολοκληρωμένο dataset που δεν ανήκουν στο training set. Επομένως, με αυτόν τον τρόπο γνωρίζουμε ότι το test set περιέχει το 20% των δεδομένων και κάθε κλάση συνεισφέρε ομοίως με 20%.

Σε αυτό το σημείο να παραθέσουμε τις εξής παρατηρήσεις. Το sampleBy δε γυρνάει με ακρίβεια το 80% των δεδομένων που του ζητάμε αλλά λίγο λιγότερο και αυτό γιατί κάνει sample με κάποια πιθανότητα κάποια εγγραφή. Ανάλογα με την εκτέλεση οι τιμές ίσως διαφέρουν αλλά είναι αρκετά κοντά σε αυτό το ποσοστό. Επίσης, το subtract δουλεύει σωστά επειδή έχουμε αφαιρέσει τα Duplicates. Στην περίπτωση που δε θέλαμε να αφαιρέσουμε τα dublicates θα μπορούσαμε να χρησιμοποιήσουμε την exceptAll. Ωστόσο, είναι προτιμότερο να αφαιρέσουμε τα duplicate rows καθώς θα βελτιώσει την ταχύτητα και το accuracy στο fit του νευρωνικού μας δικτύου.

Παρακάτω φαίνεται ο αριθμός εγγραφών ολόκληρου του dataset και των train/test set που δείχνει και το διαμοιρασμό σε 80%-20%,

dataset	380173
train set	304095
test set	76078

ενώ ο επόμενος πίνακας δείχνει τον αντίστοιχο διαμοιρασμό για κάθε κλάση.

Class	Dataset	train set	test set
class 0	73716	59109	14607
class 1	86644	69065	17579
class 2	58387	46694	11693
class 3	29723	23787	5936
class 4	22945	18358	4587
class 5	23861	19062	4799
class 6	17935	14377	3558
class 7	17756	14219	3537
class 8	13957	11155	2802
class 9	9054	7274	1780
class 10	7831	6279	1552
class 11	7370	5924	1446
class 12	6183	4945	1238
class 13	1675	1347	328
class 14	1442	1143	299
class 15	1396	1112	284
class 16	284	232	52
class 17	14	13	1

4.4 Δημιουργία MLP - training - accuracy

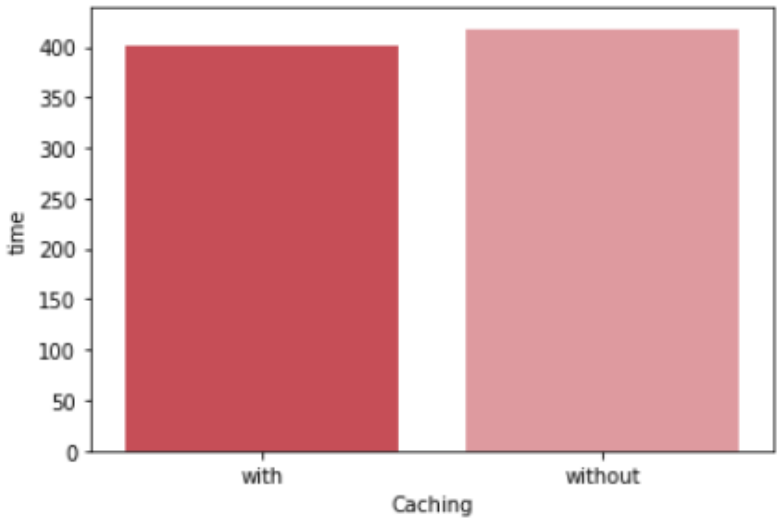
Έχοντας ολοκληρώσει τη διαδικασία του διαχωρισμού των δεδομένων σε σύνολο εκπάιδευσης και σύνολο επικύρωσης, προχωρήσαμε στη δημιουργία της αρχιτεκτονικής ενός MultiLayerPerceptron. Μετά από διάφορες επιλογές παραμέτρων ως προς το μέγεθος των κρυφών επιπέδων καταλήξαμε στην εξής αρχιτεκτονική:

- input layer που δέχεται ως είσοδο τον πλήθος του λεξικού που έχουμε διαλέξει.
- 3 κρυφά επίπεδα με μεγέθη 40,30,20 αντίστοιχα.
- output layer με 18 νευρώνες μια για κάθε κλάση.

Υπενθυμίζουμε ότι χρησιμοποιούμε λεξικό μεγέθους 50 λέξεων. Μέχρι ένα σημείο, όσο αυξάνεται το πλήθος του λεξικού αυξάνεται και η απόδοση, δηλαδή η ακρίβεια που επιτυγχάνεται, αλλά ταυτόχρονα αυξάνεται και ο χρόνος εκπαίδευσης. Η χρήση ενός μεγαλύτερου λεξικού θα απαιτούσε ένα βαθύτερο νευρωνικό με περισσότερες παραμέτρους για να βρούμε την ισορροπία μεταξύ Bias-Variance.

Στη συνέχεια κι έχοντας ορίσει τις παραμέτρους όπως αναφέραμε, δημιουργήσαμε το MLP classifier και κάνουμε training το μοντέλο με cached training set και χωρίς cached για να δούμε διαφορά στο χρόνο εκπαίδευσης. Γενικά, το spark caching μπορεί να χρησιμοποιηθεί για να τραβάμε σύνολα δεδομένων σε ένα cluster από τη memory cache. Αυτό μπορεί να φανεί χρήσιμο για την πρόσβαση σε αρχεία που χρειάζονται επανειλημμένα, όπως σε έναν iterative αλγόριθμο που στην περίπτωση μας είναι το training του MLP classifier. Ο μηχανισμός της cache μπορεί να βελτιώσει αισθητά το χρόνο σε τέτοιες περιπτώσεις και να μειώσει το κόστος.

Στο παρακάτω διάγραμμα φαίνεται το ζητούμενο barchart με τους 2 χρόνους εκπάιδευσης, με τη χρήση της cache() στο training set και χωρίς. Συγκεκριμένα, οι χρόνοι που δαπανήθηκαν ήταν 400.90 δευτερόλεπτα με τη χρήση της cache memory είναι και 417.75 δευτερόλεπτα χωρίς τη χρήση της cache.



Από το παραπάνω διάγραμμα οι δύο μεθοδολογίες δε διαφέρουν ιδιαίτερα χρονικά, αλλά παρατηρούμε τη μέθοδο εκπαίδευσης του μοντέλου με τη χρήση της cache να υπερτερεί. Αυτό συμβαίνει για δύο λόγους. Ο πρώτος είναι η χρήση ενός μικρού λεξικού, το οποίο δημιουργεί ένα μικρότερο σύνολο δεδομένων καθώς κρατάμε και λιγότερες προτάσεις αλλά έχουμε και μικρότερο sparseVector σε μέγεθος, σε σχέση με το αν χρησιμοποιούσαμε ένα μεγαλύτερο λεξικό. Ο δεύτερος λόγος είναι πως έχουμε θέσει στον MLP Classifier να κάνει μέχρι 100 επαναλήψεις για να συγκλίνει στο επιθυμητό αποτέλεσμα. Αν το training διαρκούσε παραπάνω επαναλήψεις, τότε το training set θα έπρεπε να καλεστεί περισσότερες φορές από τη μνήμη αν δεν ήταν φορτωμένο στην cache. Επομένως, με τη μη χρήση της cached memory θα πληρώναμε παραπάνω overhead για τις επανειλημμένες μεταφορές του training set στο cluster.

Τέλος, αναφέρουμε το accuracy που πετύχαμε στο test set μετά την εκπαίδευση είναι: 0.5611346249901417 δηλαδή περίπου 56,11%, το οποίο είναι πάρα πολύ καλό αν σκεφτούμε πως έχουμε ένα MultiClass πρόβλημα με 18 labels.