ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ
ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ
Ακ. έτος 2019-2020

ΔΙΑΧΕΙΡΙΣΗ ΔΕΔΟΜΕΝΩΝ ΜΕΓΑΛΗΣ ΚΛΙΜΑΚΑΣ

Εξαμηνιαία Εργασία

ΜΑΡΙΑ-ΦΙΛΙΠΠΑ ΤΡΙΒΥΖΑ 03400080
ΚΩΝΣΤΑΝΤΙΝΑ ΚΑΡΑΪΣΚΟΥ 03400054

ΙΟΥΛΙΟΣ 2020

Contents

Μέρος 1ο: Εξαγωγή Πληροφορίας - SQL

Περιγραφή Δεδομένων

Τα δεδομένα που θα χρησιμοποιήσουμε είναι πραγματικά και αφορούν σε διαδρομές taxi στην Νέα Υόρκη. Οι δοθείσες διαδρομές των taxi έγιναν από τον Ιανουάριο εώς τον Ιούνιο του 2015 και υπάρχουν διαθέσιμες online εδώ. Λόγω των περιορισμένων πόρων που η κάθε ομάδα έχει στη διάθεσή της, θα επεξεργαστούμε μόνο ένα υποσύνολο μεγέθους 2 GB. Τα δεδομένα αυτά περιέχουν 13 εκατομμύρια διαδρομές, που πραγματοποιήθηκαν τον Μάρτιο του 2015 και μπορούμε να τα κατεβάσουμε από εδώ. Στο συμπιεσμένο αρχείο που μας δίνεται, περιλαμβάνονται δύο comma-delimited αρχεία κειμένου (.csv) που ονομάζονται: yellow_tripdata_1m.csv και yellow_tripvendors_1m.csv. Το πρώτο αρχείο περιλαμβάνει όλη την απαραίτητη πληροφορία για μια διαδρομή. Το αρχείο των TripData έχει την εξής μορφή:

369367789289, 2015-03-27 18:29:39, 2015-03-27 19:08:28,
-73.975051879882813, 40.760562896728516,
-73.847900390625, 40.732685089111328, 34.8
369367789290,2015-03-27 18:29:40, 2015-03-27 18:38:35,
-73.988876342773438,40.77423095703125,
-73.985160827636719,40.763439178466797,11.16

Πίνακας 1: yellow_tripdata_1m.csv.

Το πρώτο πεδίο αποτελεί το μοναδικό id μιας διαδρομής. Το δεύτερο (τρίτο) πεδίο την ημερομηνία και ώρα έναρξης (λήξης) της διαδρομής. Το τέταρτο και πέμπτο πεδίο το γεωγραφικό μήκος και πλάτος του σημείου επιβίβασης, ενώ το έκτο και έβδομο πεδίο περιλαμβάνουν το γεωγραφικό μήκος και πλάτος του σημείου αποβίβασης. Τέλος, το όγδοο πεδίο δείχνει το συνολικό κόστος της διαδρομής. Το δεύτερο αρχείο που μας δίνεται περιέχει πληροφορία για τις εταιρίες taxi. Η μορφή του φαίνεται στο παρακάτω παράδειγμα:

369367789289,1
369367789290,2

Πίνακας 2: yellow_tripvendors_1m.csv.

Το πρώτο πεδίο αποτελεί το μοναδικό id μιας διαδρομής και το δεύτερο πεδίο το μοναδικό αναγνωριστικό μιας εταιρείας taxi (vendor).

1Α. Εξαγωγή πληροφορίας με διαφορετικούς τρόπους

Παρόλο που τα δεδομένα δίνονται σε μορφή απλού κειμένου (csv), είναι γνωστό ότι ο υπολογισμός ερωτημάτων αναλυτικής επεξεργασίας απευθείας πάνω σε αρχεία csv δεν είναι αποδοτικός. Για να βελτιστοποιηθεί η πρόσβαση των δεδομένων, παραδοσιακά οι βάσεις δεδομένων φορτώνουν τα δεδομένα σε ειδικά σχεδιασμένα binary formats.

Παρ'ότι το Spark δεν είναι μια τυπική βάση δεδομένων, αλλά ένα σύστημα κατανεμημένης επεξεργασίας, για λόγους απόδοσης, υποστηρίζει κι αυτό μια παρόμοια λογική. Αντί να τρέξουμε τα ερωτήματά μας απευθείας πάνω στα csv αρχεία, μπορούμε να μετατρέψουμε πρώτα το dataset σε μια ειδική μορφή που:

- Έχει μικρότερο αποτύπωμα στη μνήμη και στον δίσκο και άρα βελτιστοποιεί το I/O (input/output) μειώνοντας τον χρόνο εκτέλεσης.
- Διατηρεί επιπλέον πληροφορία, όπως στατιστικά πάνω στο dataset, τα οποία βοηθούν στην πιο αποτελεσματική επεξεργασία του. Για παράδειγμα, αν ψάχνουμε σε ένα σύνολο δεδομένων τις τιμές που είναι μεγαλύτερες από 100 και σε κάθε block του dataset έχουμε πληροφορία για το ποια είναι η min και ποια η max τιμή, τότε μπορώ να παρακάμψω την επεξεργασία των blocks με max τιμή < 100 γλιτώνοντας έτσι χρόνο επεξεργασίας.

Το ειδικό format που χρησιμοποιούμε για να επιτύχουμε τα παραπάνω είναι το Apache Parquet. Όταν φορτώνουμε έναν πίνακα σε Parquet, αυτός μετατρέπεται κι αποθηκεύεται σε ένα columnar format που βελτιστοποιεί το I/O και τη χρήση της μνήμης κι έχει τα χαρακτηριστικά που αναφέραμε.

Το πρόβλημα που καλούμαστε να αντιμετωπίσουμε είναι ο υπολογισμός των ερωτημάτων του Πίνακα 3 για την εξαγωγή πληροφορίας από τα δεδομένα με δύο διαφορετικούς τρόπους:

- Γράφοντας MapReduce κώδικα χρησιμοποιώντας το RDD API του Spark.
- Χρησιμοποιώντας SparkSQL και το DataFrame API.

ID	Query
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.

Πίνακας 3: Queries

Ουσιαστικά, υλοποιούμε και τρέχουμε τα ερωτήματα του Πίνακα 3 αρχικά με χρήση MapReduce κώδικα και στη συνέχεια με χρήση της SparkSQL, αφότου φορτώσουμε τα csv αρχεία που μας δίνονται στο HDFS. Ύστερα, μετατρέπουμε τα αρχεία κειμένου σε αρχεία Parquet, φορτώνουμε τα Parquet αρχεία ως Dataframes και εκτελούμε πάλι τα ερωτήματα του παραπάνω πίνακα με χρήση της SparkSQL. Ακολούθως, αναλύουμε κάθε μία από αυτές τις περιπτώσεις και σχολιάζουμε τα αποτελέσματά μας. Σημειώνουμε ότι τα αποτελέσματα του κάθε κώδικα βρίσκονται στο αντίστοιχο zip file.

1Α.1. Q1 με χρήση MapReduce κώδικα

Το πρώτο βήμα που κάνουμε κάθε φορά που θέλουμε να χρησιμοποιήσουμε το Spark είναι να θέσουμε ένα όνομα για την εφαρμογή με τη συνάρτηση 'appName', το οποίο θα εμφανίζεται στο Spark Web UI, ενώ με τη συνάρτηση 'getOrCreate' ελέγχουμε πρώτα άμα υπάρχει ήδη κάποιο SparkSession, αλλιώς δημιουργούμε ένα νέο με το όνομα που θέσαμε. Στη συνέχεια, με την εντολή 'spark.sparkContext' αναπαριστούμε τη σύνδεση σε έναν Spark cluster και μπορεί να χρησιμοποιηθεί για τη δημιουργία RDDs, accumulators και την αναμετάδοση μεταβλητών σε αυτόν τον cluster. Κάθε φορά μόνο ένα SparkContext μπορεί να είναι ενεργό. Μέσω του SparkContext και της συνάρτησης 'textFile' διαβάζουμε το αντίστοιχο αρχείο από το HDFS, το οποίο επιστρέφεται ως ένα RDD από συμβολοσειρές. Κάθε γραμμή του αρχείου, δηλαδή, φορτώνεται στη μνήμη ως string. Οπότε προκειμένου να κάνουμε πράξεις με κάποιες στήλες θα πρέπει οι τιμές να μετατραπούν από string στον κατάλληλο τύπο πρώτα. Σημειώνουμε ότι ένα RDD (Resilient Distributed Dataset) είναι το θεμελιακό στοιχείο αναπαράστασης μιας συλλογής εγγραφών στο Spark. Είναι μία συλλογή από στοιχεία διαχωρισμένα στους κόμβους του cluster τα οποία μπορούν να λειτουργήσουν παράλληλα. Ένας μετασχηματισμός σε ένα RDD έχει ως αποτέλεσμα τη δημιουργία ενός άλλου RDD, ενώ τα RDDs μπορούν να αποθηκευτούν στην κύρια μνήμη ή στον δίσκο.

Αρχικά, φιλτράρουμε τα δεδομένα με τη χρήση της συνάρτησης 'filter'. Η συνάρτηση αυτή επιστρέφει ένα νέο RDD που περιέχει μόνο τα στοιχεία που ικανοποιούν μία συνθήκη. Για τη χρήση αυτής της συνάρτησης, καθώς και για τη χρήση πολλών ακόμα χρησιμοποιούμε τις Lambda functions, με τις οποίες χειριζόμαστε τη λειτουργικότητα ως ένα όρισμα της συνάρτησης. Προκείμενου να φιλτράρουμε τα δεδομένα χρησιμοποιούμε μία συνάρτηση που υλοποιήσαμε, την 'filterData', στην οποία δίνουμε ως είσοδο κάθε γραμμή του αρχείου. Διαχωρίζουμε τις στήλες με βάση τα κόμματα και μετατρέπουμε τις ημερομηνίες και ώρες στην μορφή '%Y-%m-%d %H:%M:%S'. Η συνθήκη με την οποία φιλτράρουμε τα δεδομένα περιλαμβάνει τις εξής προτάσεις: η ημερομηνία και ώρα λήξης της διαδρομής να είναι μεγαλύτερη από την ημερομηνία και ώρα έναρξης της διαδρομής, το κόστος της διαδρομής να είναι μεγαλύτερο του 0, το γεωγραφικό μήκος και το γεωγραφικό πλάτος του σημείου επιβίβασης να είναι διαφορετικά από το γεωγραφικό μήκος και το γεωγραφικό πλάτος του σημείου αποβίβασης, τα γεωγραφικά μήκη να είναι μεγαλύτερα από -80 και μικρότερα από -70 και τα γεωγραφικά πλάτη να είναι μεγαλύτερα από 40 και μικρότερα

από 46. Τις συνθήκες αυτές τις θέσαμε ώστε να κρατήσουμε μόνο τις διαδρομές taxi στην Νέα Υόρκη, καθώς υπήρχαν και μερικά πιο απομακρυσμένα σημεία επιβίβασης ή αποβίβασης.

Στη συνέχεια, χρησιμοποιούμε τη συνάρτηση 'map' και την συνάρτηση 'getData' που υλοποιήσαμε ώστε να κρατήσουμε κάποια μόνο από τα δεδομένα της κάθε γραμμής. Η συνάρτηση 'map' κάνει ένα-προς-ένα μετασχηματισμό των δεδομένων εισόδου (του RDD) χρησιμοποιώντας μία συνάρτηση. Στην περίπτωση μας αυτή είναι η συνάρτηση lambda πάντα. Η συνάρτηση 'getData' δέχεται ως όρισμα μία γραμμή του φιλτραρισμένου αρχείου και επιστρέφει την ώρα επιβίβασης, το γεωγραφικό μήκος, το γεωγραφικό πλάτος και τον αριθμό 1. Τα τρία τελευταία ορίσματα τα ορίζουμε ως τούπλα. Ύστερα, με τη συνάρτηση 'reduceByKey' εκτελούμε τη μέθοδο reduce, δηλαδή συναθροίζουμε όλα τα στοιχεία του RDD, με βάση το κλειδί. Το κλειδί είναι πάντα το πρώτο όρισμα του κάθε στοιχείου RDD. Συνεπώς, στη συγκεκριμένη περίπτωση είναι η ώρα επιβίβασης. Χρησιμοποιώντας την κατάλληλη lambda function προσθέτουμε όλα τα γεωγραφικά μήκη, όλα τα γεωγραφικά πλάτη και όλες τις μονάδες, δηλαδή προκύπτει ο αριθμός όλων των διαδρομών. Προκειμένου να βρούμε τις μέσες τιμές, διαιρούμε τα αθροίσματα των γεωγραφικών μηκών και πλατών με το σύνολο των διαδρομών. Αυτό το πετυχαίνουμε με τη συνάρτηση map. Ταξινομούμε τα αποτελέσματα με βάση την ώρα επιβίβασης με τη χρήση της συνάρτησης 'sortBy', σε αύξουσα σειρά. Τέλος, με τη συνάρτηση 'collect' επιστρέφουμε μια λίστα που περιέχει όλα τα στοιχεία του RDD. Τυπώνουμε τα αποτελέσματα με ένα for loop, ενώ τυπώνουμε και το χρόνο εκτέλεσης του MapReduce πάνω στο csv αρχείο.

Ο ψευδοκώδικας για το πρόγραμμα MapReduce παρουσιάζεται παρακάτω.

```
map(key, value):
    #key: None
    #value: csv line
    line = value.split(",")
    start_time = strptime(line[1], "%Y-%m-%d %H:%M:%S"))
    hour = start_time.hour
    start_hour = extract_hour(start_datetime)
    longitude = line[3]
    latitude = line[4]
    emit(hour, (longitude, latitude, 1))

reduce(key, value):
    #key: hour
    #value: a list of (longitude, latitude, 1) tuples
    hour = key
    sum_longitude = 0
    sum_latitude = 0
    count = 0
    for v in value:
        sum_longitude = sum_longitude + value[0]
        sum_latitude = sum_latitude + value[1]
        count = count + value[2]
    emit(hour, (sum_longitude, sum_latitude, count))

map(key, value):
    #key: None
    #value: a tuple of (hour, (sum_longitude, sum_latitude, count))
    hour = value[0]
    sum_longitude = value[1][0]
    sum_latitude = value[1][1]
    count = value[1][2]
    avg_longitude = sum_longitude / count
    avg_latitude = sum_latitude / count
    emit(hour, avg_longitude, avg_latitude)
```

Listing 1: Q1 MapReduce

1Α.2. Q2 με χρήση MapReduce κώδικα

Διαβάζουμε πρώτα το αρχείο 'yellow_tripvendors_1m.csv' και χωρίζουμε τα στοιχεία της κάθε γραμμής με βάση το κόμμα, χρησιμοποιώντας τη συνάρτηση map. Ύστερα, διαβάζουμε το αρχείο 'yellow_tripdata_1m.csv' και φιλτράρουμε τα δεδομένα όπως και πριν με τη συνάρτηση 'filterData'. Με τη δική μας συνάρτηση 'getData' κρατάμε μόνο κάποια από τα δεδομένα της κάθε γραμμής. Η συνάρτηση αυτή αρχικά χωρίζει τα δεδομένα της κάθε γραμμής και μετατρέπει τις ημερομηνίες και ώρες στη μορφή '%Y-%m-%d %H:%M:%S'. Βρίσκουμε το χρόνο στον οποίο πραγματοποιήθηκε η αντίστοιχη διαδρομή αφαιρώντας την ώρα επιβίβασης από την ώρα αποβίβασης. Το χρόνο αυτόν τον υπολογίζουμε σε λεπτά. Ύστερα, υπολογίζουμε της απόσταση (Haversine) της διαδρομής. Για το σκοπό αυτό χρησιμοποιούμε τη συνάρτηση 'haversine' που παίρνει ως ορίσματα τα γεωγραφικά μήκη και πλάτη για τα δύο σημεία της διαδρομής.

Σημειώνουμε ότι αν φ είναι το γεωγραφικό πλάτος και λ το γεωγραφικό μήκος, τότε η απόσταση δύο σημείων δίνεται από τους τύπους:

$$a = sin^2(\Delta\varphi/2) + cos(\varphi 1) \cdot cos(\varphi 2) \cdot sin^2(\Delta\lambda/2)$$

$$c = 2 \cdot atan2(\sqrt{a}, \sqrt{1-a})$$

$$d = R \cdot c$$

όπου R είναι η ακτίνα της Γης (6371m).

Η συνάρτηση 'getData' τελικά επιστρέφει το id και μία λίστα με την απόσταση και τη διάρκεια της αντίστοιχης διαδρομής. Κάθε στοιχείο δηλαδή του RDD πλέον αποτελείται από την έξοδο της συνάρτησης. Στη συνέχεια, χρησιμοποιούμε τη συνάρτηση 'join' για να συνενωσουμε τα δύο RDDs. Συγκεκριμένα, όταν καλείται η συνάρτηση αυτή σε δεδομένα τύπου (K,V) και (K,W), όπου K είναι το κλειδί, επιστρέφει δεδομένα τύπου (K,(V,W)). Δηλαδή, συνδυάζει όλα τα στοιχεία για κάθε κλειδί. Στην περίπτωσή μας, τα δύο αρχεία ενώνονται με βάση το id της διαδρομής. Το id το αφαιρούμε με μία συνάρτηση map καθώς δε μας είναι πια χρήσιμο, ενώ φέρνουμε τα δεδομένα σε μορφή όπου το κλειδί είναι το αναγνωριστικό μιας εταιρίας ταξί (vendor) και τα στοιχεία μας είναι η αντίστοιχη απόσταση και διάρκεια. Με τη χρήση της συνάρτησης 'reduceByKey' και μιας lambda function, διαλέγουμε για κάθε κλειδί το στοιχείο που έχει τη μέγιστη απόσταση διαδρομής. Ταξινομούμε τα αποτελέσματα με βάση το κλειδί (vendor) με τη χρήση της συνάρτησης 'sortByKey', σε αύξουσα σειρά. Έπειτα, με τη συνάρτηση 'values' παίρνουμε ένα RDD με τις τιμές κάθε πλειάδας, και με την 'collect' επιστρέφουμε μια λίστα που περιέχει όλα τα στοιχεία σε αυτό το RDD. Τέλος, τυπώνουμε τα αποτελέσματα με ένα for loop, ενώ τυπώνουμε και το χρόνο εκτέλεσης του MapReduce πάνω στο csv αρχείο.

Ο ψευδοκώδικας για το πρόγραμμα MapReduce παρουσιάζεται παρακάτω.

```
#yellow_tripvendors_1m RDD
map(key, value):
    #key: None
    #value: csv line
    line = value.split(",")
    id = line[0]
    vendor = line[1]
    emit(id, vendor)

#yellow_tripdata_1m RDD
map(key, value):
    #key: None
    #value: csv line
    line = value.split(",")
    id = line[0]
    datetime_start = strptime(line[1], "%Y-%m-%d %H:%M:%S")
    datetime_end = strptime(line[2], "%Y-%m-%d %H:%M:%S")
    duration = (datetime_end - datetime_start).total_seconds() / 60.0
    start_longitude = line[3]
    start_latitude = line[4]
    end_longitude = line[5]
```

```
    end_latitude = line[6]
    distance = haversine(start_longitude, start_latitude, end_longitude, end_latitude)
    emit(id, [distance, duration])

#joined RDDs
map(key, value):
    #key: None
    #value: a tuple of (id, ([distance, duration], vendor))
    duration = value[1][0][0]
    distance = value[1][0][1]
    vendor = value[1][1]
    emit(vendor, duration, distance)

map(key, value):
    #key: None
    #value: a tuple of (vendor, distance, duration)
    vendor = value[0]
    emit(vendor, value)

reduce(key, value):
    #key: vendor
    #value: a list of (vendor, distance, duration) tuples
    vendor = key
    max_distance = value[0][1]
    duration = value[0][2]
    for v in value:
        if v[1] > max_distance:
            max_distance = v[1]
            duration = v[2]
    emit(vendor, (max_distance, duration))
```

Listing 2: Q2 MapReduce

1A.3. Q1 με χρήση SparkSQL

Αρχικά, φορτώνουμε τα δεδομένα μας ως DataFrame και με τη συνάρτηση 'createOrReplaceTempView' δημιουργούμε ή αντικαθιστούμε μία τοπική προσωρινή όψη (αναφορά στη μνήμη) του συγκεκριμένου DataFrame. Δηλαδή, μπορεί να χρησιμοποιηθεί μόνο από το spark session στο οποίο δημιουργήθηκε. Ύστερα εκτελούμε το query με την εντολή spark.sql, όπου γράφουμε κανονική SQL. Σημειώνουμε ότι τα ονόματα των στηλών στο DataFrame δημιουργούνται αυτόματα και έχουν τη μορφή _c0, _c1, κτλ. Χρησιμοποιούμε την εντολή 'SELECT' για να επιλέξουμε τα δεδομένα από το DataFrame που θέλουμε να εμφανιστούν στα αποτελέσματα. Μέσα στην εντολή 'SELECT' χρησιμοποιούμε τις εντολές 'to_timestamp' και 'hour' ώστε να μετατρέψουμε αρχικά τις ημερομηνίες επιβίβασης στην επιθυμητή μορφή και στη συνέχεια να κρατήσουμε μόνο την ώρα. Επίσης, με την εντολή 'avg' βρίσκουμε τη μέση τιμή του γεωγραφικού μήκους επιβίβασης και τη μέση τιμή του γεωγραφικού πλάτους επιβίβασης. Με την εντολή AS δίνουμε άλλο όνομα στη στήλη του πίνακα με τα αποτελέσματα. Με την εντολή 'FROM' δηλώνουμε από ποιο DataFrame να πάρουμε τα δεδομένα. Ύστερα, με την εντολή 'WHERE' φιλτράρουμε τα δεδομένα μας, ώστε να κρατήσουμε μόνο εκείνα που ικανοποιούν μία συνθήκη. Η συνθήκη που έχουμε βάλει είναι η ίδια με εκείνη που περιγράψαμε στην παράγραφο εκτέλεσης του Q1 με χρήση MapReduce κώδικα. Με την εντολή 'GROUP BY' ομαδοποιούμε τις γραμμές με βάση την ώρα επιβίβασης, ενώ με την εντολή 'ORDER BY' ταξινομούμε τα αποτελέσματα σε αύξουσα σειρά με βάση την ώρα επιβίβασης. Συνεπώς, με βάση τα παραπάνω, το αποτέλεσμα είναι η μέση τιμή του γεωγραφικού μήκους και πλάτους ανά ώρα επιβίβασης, σε αύξουσα σειρά ως προς την ώρα. Τέλος, τυπώνουμε τον πίνακα με τα αποτελέσματα και τον χρόνο εκτέλεσης της SQL πάνω σε csv αρχεία.

1Α.4. Q2 με χρήση SparkSQL

Σε αυτο το ερώτημα χρησιμοποιούμε μόνο το DataFrame API. Αρχικά, δημιουργούμε ένα StructType για το αρχείο με τις διαδρομές, το οποίο είναι ένα σύνολο από StructFields. Μέσω των StructFields δηλώνουμε το όνομα και τον τύπο των στηλών του Dataframe που φτιάχνουμε, ώστε να μπορούμε να κάνουμε πράξεις ανάμεσα στις στήλες. Φορτώνουμε τα δεδομένα μας ως DataFrame χρησιμοποιώντας το schema που φτιάξαμε. Στη συνέχεια, επαναλαμβάνουμε για το αρχείο με τα vendors, φτιάχνοντας ένα αντίστοιχο schema. Σημειώνουμε ότι τις ημερομηνίες τις ορίσαμε ως συμβολοσειρές. Ύστερα, φιλτράρουμε τα δεδομένα του αρχείου με τις διαδρομές χρησιμοποιώντας τη συνάρτηση 'where'. Η συνθήκη που έχουμε ορίσει είναι η ίδια με εκείνη που περιγράψαμε στην παράγραφο εκτέλεσης του Q1 με χρήση MapReduce κώδικα. Χρησιμοποιώντας, έπειτα, τη συνάρτηση 'withColumn' δημιουργούμε τις απαραίτητες στήλες στο DataFrame ώστε να υπολογίσουμε την απόσταση haversine για κάθε γραμμή. Δημιουργούμε με αυτόν τον τρόπο μία στήλη για τη διάρκεια της διαδρομής, μία στήλη με τη διαφορά των γεωγραφικών μηκών, μία στήλη με τη διαφορά των γεωγραφικών πλατών, μία στήλη που να παραγάγει τη μεταβλητή a όπως ορίστηκε για την απόσταση haversine και τέλος μία στήλη με την επιθυμητή απόσταση. Ύστερα, αφαιρούμε τις περιττές στήλες με τη συνάρτηση 'drop', ώστε τελικά να έχουμε μόνο τις στήλες με το id των διαδρομών, την απόσταση haversine και τη διάρκεια των διαδρομών.

Έπειτα, χρησιμοποιούμε τη συνάρτηση 'join' για να ενώσουμε τα DataFrames με τις διαδρομές και τα vendors, με βάση το id των διαδρομών. Συγκεκριμένα πραγματοποιήσαμε inner join, κατά το οποίο παίρνουμε τις καταγραφές που έχουν τα ίδια id και στα δύο DataFrames. Αφού αφαιρέσουμε τη στήλη με τα id, δημιουργούμε ένα 'παράθυρο', που θα μας βοηθήσει να υπολογίσουμε τις μέγιστες αποστάσεις διαδρομής. Μία window function εκτελεί έναν υπολογισμό σε μία ομάδα από καταγραφές που ονομάζονται 'παράθυρο'. Με τη συνάρτηση 'partitionBy' καθορίζουμε ότι ο διαχωρισμός των γραμμών θα γίνει με τη στήλη με τα vendors. Έπειτα, με τη βοήθεια του 'παραθύρου' δημιουργούμε μία στήλη με τις μέγιστες αποστάσεις διαδρομής, ενώ με τη συνάρτηση 'where' κρατάμε μόνο τις γραμμές του DataFrame που αντιστοιχούν σε αυτές τις μέγιστες αποστάσεις. Τέλος, επιλέγουμε τις κατάλληλες στήλες, τυπώνουμε το τελικό DataFrame, καθώς και τον χρόνο εκτέλεσης της SQL πάνω σε csv αρχεία.

1Α.5. Q1 με χρήση SparkSQL πάνω σε Parquet αρχεία

Αρχικά, μετατρέπουμε το αρχείο κειμένου με τις διαδρομές σε αρχείο Parquet με τη συνάρτηση 'write.parquet'. Για το σκοπό αυτό πρέπει πρώτα να φορτώσουμε το αντίστοιχο αρχείο ως DataFrame, αφού δημιουργήσουμε ένα StructType, ώστε να διατηρηθούν οι αντίστοιχες πληροφορίες στο αρχείο Parquet. Ο χρόνος που χρειάστηκε για την μετατροπή αυτού του αρχείου ήταν: 41.75076961517334 seconds. Φορτώνουμε το αρχείο Parquet από το HDFS με τη συνάρτηση 'sqlContext.read.parquet' και στη συνέχεια υπλογίζουμε το query Q1 με τη χρήση του ίδιου κώδικα που περιγράψαμε στην παράγραφο εκτέλεσης του Q1 με χρήση SparkSQL. Τέλος, τυπώνουμε τα αποτελέσματα καθώς και τον χρόνο εκτέλεσης της SQL πάνω σε Parquet αρχεία.

1Α.6. Q2 με χρήση SparkSQL πάνω σε Parquet αρχεία

Μετατρέπουμε το αρχείο με τα vendors σε αρχείο Parquet με τον ίδιο τρόπο που περιγράψαμε παραπάνω. Ο χρόνος που χρειάστηκε για την μετατροπή αυτού του αρχείου ήταν: 15.413066387176514 seconds. Ο χρόνος αυτός είναι πολύ μικρότερος από τον χρόνο μετατροπής του αρχείου με τις διαδρομές ταξί σε αρχείο Parquet. Αυτό είναι λογικό, καθώς το αρχείο με τα vendors έχει αρκετά μικρότερο μέγεθος. Ύστερα φορτώνουμε και τα δύο αρχεία Parquet από το HDFS με τη συνάρτηση 'sqlContext.read.parquet' και στη συνέχεια υπολογίζουμε το query Q2 με τη χρήση του ίδιου κώδικα που περιγράψαμε στην παράγραφο εκτέλεσης του Q2 με χρήση SparkSQL. Τέλος, τυπώνουμε τα αποτελέσματα και τον χρόνο εκτέλεσης της SQL πάνω σε Parquet αρχεία.

1Α.7. Αποτελέσματα/Συγκρίσεις

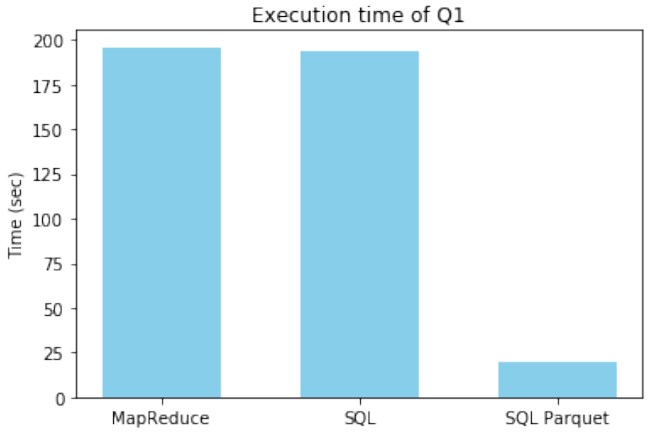
Για κάθε ερώτημα του Πίνακα 3 φτιάχνουμε ένα διάγραμμα με τον χρόνο εκτέλεσης των παραπάνω περιπτώσεων, δηλαδή:

a. MR πάνω σε csv αρχεία.

b. SQL πάνω σε csv αρχεία.

c. SQL πάνω σε Parquet αρχεία.

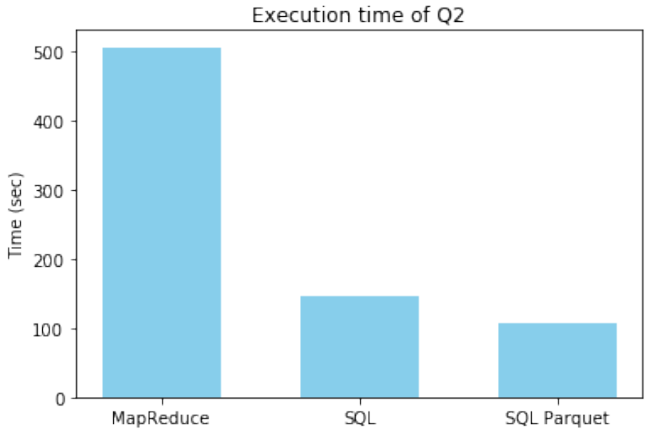
Για το query Q1 έχουμε:



Εικόνα 1: Διάγραμμα με τον χρόνο εκτέλεσης για το query Q1.

Από το παραπάνω διάγραμμα παρατηρούμε ότι ο χρόνος εκτέλεσης με χρήση SQL πάνω σε csv αρχεία είναι λίγο μικρότερος από τον χρόνο εκτέλεσης με χρήση MR πάνω σε csv αρχεία. Συγκεκριμένα η διαφορά είναι γύρω στα 2 seconds. Όσον αφορά το χρόνο εκτέλεσης με χρήση SQL πάνω σε Parquet αρχεία, παρατηρούμε ότι είναι πάρα πολύ μικρότερος από τις άλλες δύο εκτελέσεις με τιμή μόλις 19.4 seconds. Συμπεραίνουμε λοιπόν ότι η χρήση SQL πάνω σε Parquet αρχεία είναι προτιμότερη. Ο χρόνος εκτέλεσης είναι μικρότερος, καθώς τα δεδομένα έχουν μικρότερο αποτύπωμα στη μνήμη και στο δίσκο και άρα βελτιστοποιείται το I/O. Συγκεκριμένα, τα δεδομένα αποθηκεύονται στη μνήμη κατά στήλες, ενώ στην περίπτωση του csv τα δεδομένα αποθηκεύονται κατά εγγραφή. Συνεπώς, τα Parquet αρχεία ενδείκνυνται περισσότερο για ενέργειες όπως είναι η εισαγωγή ή η αφαίρεση στηλών, ενώ επίσης κατά το σκανάρισμα του αρχείου για την εύρεση των πληροφοριών που είναι απαραίτητες για το query, δεν εξετάζονται όλα τα αποθηκευμένα δεδομένα, όπως στην περίπτωση συνδυασμού στηλών, οπότε τα διάβασμα γίνεται αρκετά πιο γρήγορα από την περίπτωση του csv, όπου πρέπει κάθε φορά να διαβάσουμε ολόκληρες τις εγγραφές. Επίσης, διατηρεί επιπλέον πληροφορία, όπως στατιστικά πάνω στα δεδομένα σε κάθε block για κάθε στήλη, καθώς τα δεδομένα είναι χωρισμένα σε row blocks, η οποία μπορεί να δείξει εάν ένα block περιέχει χρήσιμες πληροφορίες ή όχι. Δηλαδή, μπορούμε μόνο από τα μεταδεδομένα να αποφασίσουμε εάν είναι χρήσιμη ή όχι μία στήλη, χωρίς να μπούμε στη διαδικασία να τη διαβάσουμε και επομένως να περάσουμε κατευθείαν στο επόμενο block δεδομένων. Στο συγκεκριμένο ερώτημα, όπου για τον υπολογισμό του query επιλέγουμε τρείς μόνο στήλες, σίγουρα η επιλογή χρήσης Parquet αρχείου είναι πιο αποδοτική.

Αντίστοιχα για το query Q2 έχουμε:



Εικόνα 2: Διάγραμμα με τον χρόνο εκτέλεσης για το query Q2.

Απο το παραπάνω διάγραμμα παρατηρούμε ότι ο χρόνος εκτέλεσης με χρήση SQL πάνω σε csv αρχεία είναι πολύ μικρότερος από τον χρόνο εκτέλεσης με χρήση MR πάνω σε csv αρχεία, περίπου κατά 360 seconds. Αυτό πιθανώς οφείλεται στο ότι η συνένωση αρχείων είναι αρκετά χρονοβόρα διαδικασία στο MapReduce σε RDD. Ωστόσο, παρατηρούμε ότι ο χρόνος με χρήση SQL πάνω σε Parquet αρχεία είναι μικρότερος μόνο κατά περίπου 40 seconds από τον χρόνο εκτέλεσης με χρήση SQL πάνω σε csv αρχεία. Αυτό οφείλεται στο γεγονός ότι τα parquet αρχεία δεν ενδείκνυνται για τη συνένωση (join) αρχείων, καθώς η ανάγνωση όλων των στηλών μιας γραμμής είναι μία ακριβής διαδικασία. Παρόλα αυτά, οι πράξεις μεταξύ των στηλών, η αφαίρεση στηλών και η δημιουργία στηλών γίνονται σίγουρα πιο γρήγορα. Σημειώνουμε ότι ο χρόνος εκτέλεσης του query 2 είναι συνολικά αρκετά μεγαλύτερος από τον χρόνο εκτέλεσης του query 1, εκτός της περίπτωσης της SQL πάνω σε csv αρχεία. Από αυτό συμπεραίνουμε ότι η εκτέλεση με χρήση του DataFrame API είναι πιο γρήγορη από τη χρήση της συνάρτησης 'spark.sql'.

1Β. Μελέτη του βελτιστοποιητή για την συνένωση δεδομένων.

Το SparkSQL έχει υλοποιημένα και τα δύο είδη ερωτημάτων συνένωσης στο DataFrame API. Συγκεκριμένα, με βάση τη δομή των δεδομένων και των υπολογισμών που θέλουμε καθώς και τις ρυθμίσεις του χρήστη, πραγματοποιεί από μόνο του κάποιες βελτιστοποιήσεις στην εκτέλεση του ερωτήματος χρησιμοποιώντας έναν βελτιστοποιητή ερωτημάτων (query optimizer), κάτι που όλες οι βάσεις δεδομένων έχουν. Μια τέτοια βελτιστοποίηση είναι ότι επιλέγει αυτόματα την υλοποίηση που θα χρησιμοποιήσει για ένα ερώτημα join λαμβάνοντας υπόψη το μέγεθος των δεδομένων και πολλές φορές αλλάζει και την σειρά ορισμένων τελεστών προσπαθώντας να μειώσει τον συνολικό χρόνο εκτέλεσης του ερωτήματος. Αν ο ένας πίνακας είναι αρκετά μικρός (με βάση ένα όριο που ρυθμίζει ο χρήστης) θα χρησιμοποιήσει το broadcast join, αλλιώς θα κάνει ένα repartition join.

Μελετάμε την επίδραση του βελτιστοποιητή στην εκτέλεση των ερωτημάτων συνένωσης:

1. Σε αυτό το ερώτημα θα εκτελέσουμε με SparkSQL ένα join πάνω στα 2 parquet αρχεία, αφού απομονώσουμε πρώτα τις 50 πρώτες γραμμές από το αρχείο με τις εταιρείες ταξί. Αρχικά, φορτώνουμε από τον HDFS τα δύο Parquet αρχεία. Στη συνέχεια, με τη συνάρτηση 'limit' απομονώνουμε τις 50 πρώτες γραμμές από το αρχείο με τα vendors και κάνουμε join των δύο αρχείων. Με τη συνάρτηση 'explain' μπορούμε να δούμε τις λεπτομέρειες του πλάνου εκτέλεσης. Παρατηρούμε ότι το Spark επέλεξε να χρησιμοποιήσει την υλοποίηση 'BroadcastHashJoin'. Καθώς σε μία λειτουργία join θέλουμε να ενώσουμε δύο πίνακες με βάση την ίδια τιμή κλειδιών , ο πιο άμεσος τρόπος να το κάνουμε αυτό βασίζεται στον διαχωρισμό των κλειδιών, όπου μετά σε κάθε διαχωρισμό γίνεται η λειτουργία ένωσης με βάση τα κλειδιά. Ωστόσο, αυτή η διαδικασία περιλαμβάνει shuffle, και το shuffle στο Spark είναι μία χρονοβόρα διαδικασία, οπότε ιδανικά θα θέλαμε μία εφαρμογή στην οποία αποφεύγουμε μέρος του shuffle. Δηλαδή, να καταλήξουν στους ίδιους mappers εγγραφές από τα δύο κείμενα που να έχουν τα ίδια κλειδιά. Όταν ο πίνακας που θέλουμε να συνενώσουμε σε έναν άλλον είναι περιορισμένου μεγέθους, τότε μπορούμε να στείλουμε όλα τα δεδομένα αυτού του πίνακα σε κάθε executor κόμβο και να δημιουργήσουμε ένα hashmap με κλειδί το κλειδί του join ώστε να είναι γρήγορη η αναζήτηση. Με αυτόν τον τρόπο γλιτώνουμε χρόνο από την αποφυγή χρήσης της λειτουργίας shuffle. Αυτό το είδος join ονομάζεται Broadcast Join και για τον λόγο που αναφέραμε το επέλεξε το Spark. Το πλάνο εκτέλεσης βλέπουμε επίσης ότι περιλαμβάνει πληροφορίες για τα δύο DataFrames που θέλουμε να συνενώσουμε. Τέλος, τυπώνουμε τις 50 γραμμές που προκύπτουν από τη συνένωση. Παρατηρούμε ότι οι εγγραφές είναι ταξινομημένες σε αύξουσα σειρά με βάση το id, το οποίο είναι λογικό καθώς δεν πραγματοποιήθηκε shuffle. Το ερώτημα αυτό εκτελέστηκε σε 19.74 seconds.

2. Στη συνέχεια, ρυθμίζουμε κατάλληλα το Spark χρησιμοποιώντας τις ρυθμίσεις του βελτιστοποιητή ώστε να μην επιλεγεί η υλοποίηση join του προηγούμενου ερωτήματος. Αυτό το πετυχαίνουμε χρησιμοποιώντας την εντολή 'conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")'. Συγκεκριμένα, με το 'spark.sql.autoBroadcastJoinThreshold' ορίζουμε το μέγιστο μέγεθος σε bytes του πίνακα που μπορεί να γίνει broadcasted σε όλους τους κόμβους, όταν εκτελούμε ένα join. Θέτοντας αυτή την τιμή σε -1, μπορούμε να απενεργοποιήσουμε το συγκεκριμένο join. Ο υπόλοιπος κώδικας είναι ακριβώς ίδιος με τον κώδικα του προηγούμενου ερωτήματος. Από το πλάνο εκτέλεσης παρατηρούμε ότι το Spark επέλεξε να χρησιμοποιήσει την υλοποίηση 'SortMergeJoin'. Αυτό το είδος join χρησιμοποιείται από το Spark όταν οι δύο πίνακες που θέλουμε να ενώσουμε είναι πολύ μεγάλοι. Αρχικά, οι δύο πίνακες ανακατεύονται ώστε να βεβαιωθούμε ότι τα ίδια κλειδιά θα καταλήξουν στην ίδια ομάδα διαχωρισμού (mapper) και από τους δύο πίνακες. Αφού διαχωριστούν τα δεδομένα, ταξινομούνται και στη συνέχεια πραγματοποιείται η συνένωση. Όπως παρατηρούμε και από το πλάνο εκτέλεσης πραγματοποιείται sorting. Έπειτα, τυπώνουμε τις 50 γραμμές που προκύπτουν από τη συνένωση. Αυτή τη φορά παρατηρούμε πως το DataFrame δεν είναι ταξινομημένο

ως προς το id, που είναι λογικό καθώς πραγματοποιήθηκε η λειτουργία shuffle. Το ερώτημα αυτό εκτελέστηκε σε 26.6 seconds. Όπως αναμέναμε ο χρόνος αυτός είναι μεγαλύτερος από τον χρόνο που καταγράψαμε για το πρώτο ερώτημα, καθώς η λειτουργία shuffle είναι αρκετά χρονοβόρα. Συμπεραίνουμε λοιπόν ότι είναι λογικό το Spark να διαλέγει πρώτα να εκτελέσει 'BroadcastHashJoin', εφόσον ο ένας πίνακας έχει αρκετά μικρό μέγεθος και χωράει στη μνήμη του κάθε mapper.

Μέρος 2ο: Machine Learning - Κατηγοριοποίηση κειμένων

Περιγραφή Δεδομένων

Στο δεύτερο μέρος της εργασίας θα ασχοληθούμε με τη χρήση του Apache Spark για την κατηγοριοποίηση κειμένων. Για τις ανάγκες της εργασίας, θα χρησιμοποιήσουμε ένα πραγματικό σύνολο δεδομένων, το οποίο περιγράφει παράπονα πελατών σε σχέση με οικονομικά προϊόντα και υπηρεσίες. Κάθε ένα παράπονο έχει επισημανθεί με το σε ποια γενική κατηγορία προϊόντος αναφέρεται και έτσι μπορεί να χρησιμοποιηθεί. Το παραπάνω σύνολο δεδομένων αφορά δεδομένα που έχουν συλλεχθεί από το 2011 μέχρι σήμερα και βρίσκονται διαθέσιμα εδώ. Για τις ανάγκες τις εργασίες, θα δουλέψουμε με ένα υποσύνολο των δεδομένων το οποίο βρίσκεται στην συγκεκριμένη τοποθεσία. Το συμπιεσμένο αρχείο που μας δίνεται, περιλαμβάνει ένα comma-delimited αρχείο κειμένου (.csv) που ονομάζεται customer_complaints.csv, περιλαμβάνει όλη την παραπάνω πληροφορία και έχει την εξής μορφή:

2019-09-24,Debt collection,transworld systems inc. is trying to collect a debt that is not mine not owed and is inaccurate. 2019-09-19, Credit reporting credit repair services or other personal consumer reports,

Πίνακας 4: customer_complaints.csv.

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.

Πρώτος Τρόπος Επίλυσης

Για την εξαγωγή χαρακτηριστικών με στόχο την εκπαίδευση μοντέλων μηχανικής μάθησης, θα χρησιμοποιήσουμε την τεχνική TF-IDF. Η μετρική είναι μια ένδειξη της σημαντικότητας μιας λέξης μέσα σε ένα κείμενο. Ο μαθηματικός τύπος για τον υπολογισμό της μετρικής είναι:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

Όπου t, d, D ο όρος, το κείμενο και η συλλογή κειμένων στα οποία γίνεται ο υπολογισμός αντίστοιχα. Οι όροι tf, idf υπολογίζονται σύμφωνα με τους τύπους:

$$\text{tf}(t, d) = 0.5 + 0.5 \cdot \frac{f_{t,d}}{max\{f_{t',d} : t' \in d\}}$$

$$\text{idf}(t, D) = log\frac{N}{|\{d \in D : t \in d\}|}$$

Αρχικά, φορτώνουμε το αρχείο csv στο HDFS. Στη συνέχεια, καθαρίζουμε τα δεδομένα. Αυτό το πετυχαίνουμε κρατώντας μόνο τις γραμμές που ξεκινάνε από '201', δηλαδή κρατάμε μόνο τις έγκυρες γραμμές που έχουν στην αρχή ημερομηνίες όπως προδιαγράφεται στο σύνολο, και βρίσκοντας ποιες γραμμές έχουν σχόλιο του χρήστη, ελέγχοντας εάν η τρίτη στήλη του πίνακα έχει κενό string. Τον καθαρισμό τον πετυχαίνουμε με τη δική μας συνάρτηση 'filterData' και τη συνάρτηση 'filter' του RDD API. Ύστερα με τη συνάρτηση 'getData' χωρίζουμε κάθε γραμμή με βάση το ',' και μετατρέπουμε όλες τις συμβολοσειρές, έτσι ώστε να αποτελούνται μόνο από πεζά γράμματα. Η συνάρτηση επιστρέφει την κατηγορία του προϊόντος και το σχόλιο της κάθε γραμμής, καθώς η ημερομηνία δε θα μας χρειαστεί παρακάτω. Με τη συνάρτηση 'cache' αποθηκεύουμε την έξοδο ('customer_complaints') γιατί θα χρησιμοποιηθεί τόσο για την εύρεση του λεξικού όσο και για τη δημιουργία των sparse features vectos και δε χρειάζεται να την ξανα-υπολογίζουμε.

```
#customer_complaints RDD
map(key, value):
    #key: None
    #value: csv line
    line = value.split(",")
    date = line[0]
    productCategory = line[1]
    complaints = line[2].lower()
    emit(productCategory, complaints)
```

Listing 3: 'getData' from customer_complaints RDD MapReduce

Για τον υπολογισμό των TFIDF για κάθε λέξη του κειμένου, βρίσκουμε πρώτα όλες τις διαφορετικές λέξεις που υπάρχουν στα σχόλια των χρηστών. Για το σκοπό αυτό δημιουργούμε το λεξικό 'full_lexicon'. Αρχικά, με τη συνάρτηση 'flatMap' χωρίζουμε όλα τα κείμενα σε λέξεις , με βάση το κενό. Έπειτα, κρατάμε μόνο τις λέξεις που αποτελούνται από αλφαβητικούς χαρακτήρες, ενώ φιλτράρουμε τις λέξεις κρατώντας εκείνες που δεν ανήκουν στα stopwords. Μία stopword είναι μία ευρέως χρησιμοποιούμενη λέξη που μία μηχανή αναζήτησης είναι προγραμματισμένη να αγνοεί. Με λίγα λόγια είναι λέξεις που δεν περιέχουν καμία σημαντική πληροφορία, όπως 'the', 'a', 'in' κτλ. Στη συνέχεια, αθροίζουμε το πόσες φορές εμφανίζεται η κάθε λέξη σε όλα τα κείμενα, και τις ταξινομούμε σε φθίνουσα σειρά, ενώ κρατάμε μία λίστα πρακτικά από τις λέξεις μονο, χωρίς τη συχνότητα. Καθώς δεν έχει νόημα να κρατήσουμε όλες τις λέξεις του λεξικού, επειδή πολλές απο αυτές αποτελούν περιττή πληροφορία ή εμφανίζονται σε ελάχιστα σχόλια, και προκειμένου να αποφύγουμε τα out of memory errors, θα κρατήσουμε μόνο τις 150 συχνότερες λέξεις του λεξικού. Το λεξικό αυτό πρέπει να το γνωρίζουν όλοι οι workers ώστε να μπορούν να μετασχηματίσουν κατάλληλα τις τούπλες των προτάσεων. Για το σκοπό αυτό χρησιμοποιούμε τη συνάρτηση 'broadcast', η οποία προωθεί τη μεταβλητή που της δίνεται ως όρισμα σε όλους τους executors για να μπορούν να τη χρησιμοποιήσουν.

```
flatmap(key, value):
    #key: None
    #value: a tuple of (productCategory, complaints)
    words = value[1].split(" ")
    emit(words)

map(key, value):
    #key: None
    #value: word
    word = re.sub('[^a-zA-Z]+', '', value)
    emit(word)

map(key, value):
    #key: None
    #value: word
    emit(value, 1)

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

map(key, value):
    #key: None
    #value: a tuple of (word, counter)
```

```
    word = value[0]
    emit(word)
```

Listing 4: 'full_lexicon' MapReduce

Δημιουργούμε μία λίστα από όλες τις λέξεις που απαρτίζουν μία πρόταση και επιλέγουμε να κρατήσουμε μόνο εκείνες που περιλαμβάνονται στο λεξικό που φτιάξαμε. Ύστερα, κρατάμε όσες εγγραφές έχουν τουλάχιστον μία λέξη στο λεξικό και δίνουμε την πληροφορία του index της πρότασης στην τούπλα με την εντολή 'zipWithIndex'. Αυτό το βήμα είναι απαραίτητο έτσι ώστε αργότερα να μπορέσουμε να ενώσουμε τις λέξεις που ανήκουν στην ίδια πρόταση. Τώρα η τούπλες περιλαμβάνουν την κατηγορία του προϊόντος, τη λίστα από τις λέξεις που υπάρχουν στο λεξικό, για την κάθε πρόταση, και το index της πρότασης. Με τη συνάρτηση 'count' μετράμε τον αριθμό των προτάσεων που έχουμε στη διάθεσή μας, καθώς θα μας χρειαστεί για να υπολογίσουμε το idf. Για τον υπολογισμό του idf της κάθε λέξης, αρχικά χωρίζουμε τις τούπλες που προέκυψαν ('customer_complaints') σε κάθε λέξη της πρότασης, κρατώντας μία μόνο φορά εκείνες που εμφανίζονται παραπάνω φορές, καθώς θέλουμε να βρούμε σε πόσα κείμενα εμφανίζεται μία λέξη. Αυτό το πετυχαίνουμε μετατρέποντας την πρόταση σε σύνολο με την εντολή 'set'. Αφού υπολογίσουμε τον αντίστοιχο αριθμό για την κάθε λέξη με τη χρήση της συνάρτησης 'reduceByKey', υπολογίζουμε το idf της κάθε λέξης με τον τύπο που μας έχει δωθεί. Το αποτέλεσμα το αποθηκεύουμε στην μεταβλητή 'idf', ενώ το κλειδί κάθε τούπλας είναι η αντίστοιχη λέξη.

```
#customer_complaints RDD
map(key, value):
    #key: None
    #value: a tuple of (productCategory, complaints)
    string_label = value[0]
    words_list = complaints.split(" ")
    emit(string_label, words_list)

map(key, value):
    #key: None
    #value: a tuple of (string_label, words_list)
    string_label = value[0]
    words_list = value[1]
    list_of_sentence_words_in_lexicon = [word for word in words_list if
                                         word in broad_com_words]
    emit(string_label, list_of_sentence_words_in_lexicon)

#idf RDD
flatmap(key, value):
    #key: None
    #value: ((string_label, list_of_sentence_words_in_lexicon), sentence_index)
    words = set(value[0][1])
    for word in words:
        emit(word, 1)

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

map(key, value):
    #key: None
    #value: a tuple of (word, counter)
    word = value[0]
```

```
    counter = value[1]
    idf = log(number_of_complaints/counter)
    emit(word, idf)
```

Listing 5: 'customer_complaints' RDD & 'idf' RDD MapReduce

Στο αποτέλεσμά μας, 'customer_complaints', μετασχηματίζουμε κάθε γραμμή ώστε να προκύψουν εγγραφές με λέξεις με τη συνάρτηση 'flatMap', ώστε να μετρήσουμε πόσες φορές εμφανίζεται η κάθε λέξη μέσα σε κάθε πρόταση. Ωστόσο, για να μπορούμε να διαχωρίσουμε τις ίδιες λέξεις που ανήκουν σε διαφορετικές προτάσεις θέτουμε ένα σύνθετο κλειδί που αποτελείται από τη λέξη, την κατηγορία και το index της πρότασης. Σημειώνουμε ότι την κατηγορία την κρατάμε καθώς θα μας χρειαστεί στην ταξινόμηση και την θέτουμε μέσα στο κλειδί ώστε στο value της τούπλς να έχουμε μόνο τη μονάδα, για να μπορούμε να αθροίσουμε τις λέξεις με το ίδιο κλειδί. Τη συχνότητα εμφάνισης των λέξεων μέσα σε μία πρόταση την παίρνουμε με τη συνάρτηση 'reduceByKey'. Έπειτα, στο value της τούπλας εισάγουμε το index της αντίστοιχης λέξης μέσα στο λεξικό μας με τις συχνότερες λέξεις, το οποίο βρίσκεται στην broadcasted μεταβλητή και το παίρνουμε ως εξής 'broad_com_words.value.index'. Αυτό είναι απαραίτητο για την αναπαράσταση με sparse vectors. Ύστερα, αφότου έχουμε θέσει ως κλειδί πάλι κάθε τούπλας τη λέξη, κάνουμε join τη μεταβλητή 'idf' που υπολογίσαμε παραπάνω. Έτσι, σε κάθε τούπλα προστίθεται η τιμή idf της αντίστοιχης λέξης. Μετασχηματίζουμε πάλι τις τούπλες έτσι ώστε το κλειδί να περιλαμβάνει τη λέξη, την κατηγορία και το index της πρότασης, ενώ το value περιλαμβάνει τη συχνότητα εμφάνισης της λέξης μέσα στην πρόταση, το idf και το index της λέξης στο λεξικό. Αποθηκεύουμε το αποτέλεσμα στο 'customer_complaints'.

```
#customer_complaints RDD
flatmap(key, value):
    #key: None
    #value: a tuple of ((string_label, list_of_sentence_words_in_lexicon),
    sentence_index)
    string_label = value[0][0]
    list_of_sentence_words_in_lexicon = value[0][1]
    sentence_index = value[1]
    for word in list_of_sentence_words_in_lexicon:
        emit((word, string_label, sentence_index), 1)

reduce(key, value):
    #key: (word, string_label, sentence_index)
    #value: a list of 1's
    word_count_in_sentence = 0
    for v in value:
        word_count_in_sentence = word_count_in_sentence + v
    emit(key, word_count_in_sentence)

map(key, value):
    #key: None
    #value: a tuple of ((word, string_label, sentence_index, sentence_length),
    word_count_in_sentence)
    word = value[0][0]
    string_label = value[0][1]
    sentence_index = value[0][2]
    word_count_in_sentence = value[1]
    word_index_in_lexicon = broad_com_words.value.index(word)
    emit(word, (string_label, sentence_index, word_count_in_sentence,
    word_index_in_lexicon))

#joined
map(key, value):
    #key: None
    #value: a tuple of (word, ((string_label, sentence_index,
                        word_count_in_sentence, word_index_in_lexicon),
```

```
                        idf))
    word = value[0]
    string_label = value[1][0][0]
    sentence_index = value[1][0][1]
    word_count_in_sentence = value[1][0][2]
    idf = value[1][1]
    word_index_in_lexicon = value[1][0][3]
    emit((word, string_label, sentence_index),
         (word_count_in_sentence, idf, word_index_in_lexicon))
```

Listing 6: 'customer_complaints' RDD MapReduce

Στη συνέχεια, θα υπολογίσουμε το tf κάθε λέξης. Για αυτόν τον υπολογισμό χρειαζόμαστε τη συχνότητα μιας λέξης μέσα σε ένα σχόλιο, καθώς και τη μέγιστη συχνότητα μέσα σε αυτό το σχόλιο. Για την εύρεση της μέγιστης συχνότητας χρησιμοποιούμε τη δική μας συνάρτηση 'get_max' καθώς η χρήση της έτοιμης εντολής 'max' ερχόταν σε αντίθεση με το import συναρτήσεων της sql, που χρειαζόμαστε παρακάτω. Προκειμένου να βρούμε το tf, απο το 'customer_complaints' πρέπει να βάλουμε όλες τις λέξεις μιας πρότασης σε μία λίστα, κρατώντας επίσης σε μία άλλη λίστα τη συχνότητα εμφάνισης της κάθε λέξης μέσα στην πρόταση. Αυτό το πετυχαίνουμε θέτοντας ως κλειδί την κατηγορία του προϊόντος και το index της πρότασης και ως value μία τούπλα με δύο μοναδιαίες λίστες, που η μία περιέχει τη λέξη και η άλλη τη συχνότητα. Τα υπόλοιπα στοιχεία της τούπλας δεν τα χρειαζόμαστε για τον υπολογισμό του tf. Με τη βοήθεια της συνάρτησης 'reduceByKey' δημιουργούμε για κάθε πρόταση τις δύο λίστες που περιγράψαμε. Οπότε έπειτα με τη χρήση της συνάρτησης 'map' εφαρμόζουμε τη διαίρεση της κάθε συχνότητας με την αντίστοιχη μέγιστη συχνότητα, ενώ με ένα ακόμη 'map' υπολογίζουμε τελικά το tf. Για να μπορέσουμε να χρησιμοποιήσουμε το αποτέλεσμα πρέπει να φέρουμε τις τούπλες στη μορφή που έχουν και οι τούπλες του 'customer_complaints'. Οπότε, θέτουμε ως κλειδί τη λέξη, την κατηγορία και το index της πρότασης και ως value το tf. Το αποτέλεσμα το αποθηκεύουμε στο RDD 'tf'.

```
#tf RDD
map(key, value):
    #key: None
    #value: a tuple of ((word, string_label, sentence_index),
                        (word_count_in_sentence, idf, word_index_in_lexicon))
    string_label = value[0][1]
    sentence_index = value[0][2]
    word = value[0][0]
    word_count_in_sentence = value[1][0]
    emit((string_label, sentence_index), ([word], [word_count_in_sentence]))

reduce(key, value):
    #key: (string_label, sentence_index)
    #value: a list of ([word], [word_count_in_sentence]) tuples
    words_list = [] #listof(words)
    word_count_list = [] #listof(word_count_in_sentence)
    for v in value:
        word = v[0]
        word_count_in_sentence = v[1]
        words_list.append(word)
        word_count_list.append(word_count_in_sentence)
    emit(key, (words_list, word_count_list))

map(key, value):
    #key: None
    #value: a tuple of ((string_label, sentence_index),
                        (words_list, word_count_list))
    string_label = value[0][0]
    sentence_index = value[0][1]
    words_list = value[1][0] #listof(words)
```

```
    word_count_list = value[1][1] #listof(word_count_in_sentence)
    result_list = [] #listof(word_count/max_word_count)
    for word_count in word_count_list:
        result_list.append(word_count/get_max(word_count_list))
    emit((string_label, sentence_index), (words_list, result_list))

map(key, value):
    #key: None
    #value: a tuple of ((string_label, sentence_index), (words_list, result_list))
    (x[0], (x[1][0], [(0.5 + 0.5 * y) for y in x[1][1]]))
    string_label = value[0][0]
    sentence_index = value[0][1]
    words_list = value[1][0] #listof(words)
    result_list = value[1][1] #listof(word_count/max_word_count)
    tf_list = [] #listof(tf)
    for y in result_list:
        tf_list.append(0.5 + 0.5 * y)
    emit((string_label, sentence_index), (words_list, tf_list))

map(key, value):
    #key: None
    #value: a tuple of ((string_label,sentence_index), (words_list, tf_list))
    string_label = value[0][0]
    sentence_index = value[0][0]
    words_list = value[1][0] #listof(words)
    tf_list = value[1][1] #listof(tf)
    emit((string_label, sentence_index), [words_list, tf_list])

flatmap(key, value):
    #key: None
    #value: a tuple of ((string_label, sentence_index), [words_list, tf_list])
    words_list = value[1][0] #listof(words)
    for i in range(0, length(words_list))
        word = value[1][0][i]
        string_label = value[0][0]
        sentence_index = value[0][1]
        tf = value[1][1][i]
        emit((word, string_label, sentence_index), tf)
```

Listing 7: 'tf' RDD MapReduce

Κάνουμε join τη μεταβλητή 'idf' στο 'customer_complaints'. Με ένα 'map' υπολογίζουμε το tfidf πολλαπλασιάζοντας τα αντίστοιχα tf και idf. Στη συνέχεια, πρέπει να φέρουμε στην ίδια γραμμή όλες τις λέξεις μίας πρότασης. Όπως και πριν θέτουμε ως σύνθετο κλειδί την κατηγορία του προϊόντος και το index της πρότασης. Το index της λέξης στο λεξικό και το tfidf τα θέτουμε ως τούπλα μέσα σε μία λίστα, ώστε να μπορούμε να συνδυάσουμε αυτές τις πληροφορίες για την κάθε λέξη της πρότασης. Με τη χρήση της συνάρτησης 'reduceByKey' συνενώνουμε τις επιμέρους λίστες σε μία, που περιέχει όλη την πληροφορία για μία πρόταση. Έπειτα, αφαιρούμε το index της πρότασης από το κλειδί, καθώς δε μας είναι πια χρήσιμο, και ταξινομούμε τη λίστα που υπάρχει ως value ως προς το index της λέξης στο λεξικό, ώστε η πληροφορία που έχουμε να είναι με τη σειρά που εμφανίζεται στο λεξικό. Τέλος, φέρνουμε το αποτέλεσμα στην ακόλουθη μορφή:

$$(N, (\text{ind}1, \text{ind}2, ..., \text{ind}K), (\text{tfidf}1, \text{tfidf}2, \text{tfidf}K))$$

όπου N το πλήθος των αρχείων, ind<i> η θέση της συγκεκριμένης λέξης του κειμένου στο λεξικό μας και tfidf<i> η τιμή της μετρικής για την λέξη αυτή στο δεδομένο κείμενο. Το RDD περιέχει επιπλέον την κατηγορία του προϊόντος που αναφέρεται το παράπονο. Παραθέτουμε την έξοδο με την παραπάνω μορφή από τα 5 πρώτα κείμενα:

('Mortgage', (489331, [5, 6, 10, 13, 15, 20, 25, 37, 59, 63, 103, 109], [0.8958419108818427, 1.6539660344894465, 0.9012239727060612, 0.9015379257212522, 0.8798901227901325, 1.0598386348018674, 1.3506880278433127, 1.7449221698568267, 1.7828374445394157, 1.289834856595974, 1.9244605657456062, 1.5399083638413906]))
('Credit reporting credit repair services or other personal consumer reports', (489331, [0, 1, 2, 10, 14, 18, 20, 27, 30, 33, 40, 41, 42, 48, 50, 51, 84, 85, 104, 136, 140], [0.6273125759423792, 0.6295268350153793, 0.8508500490776176, 1.5020399545101022, 1.3154521602152303, 1.476433076242519, 1.1775984831131858, 1.263808620260169, 1.2297128673050486, 1.3238451806835336, 1.4248299074423407, 1.2092026064945018, 1.241388101746225, 1.3536398556200304, 1.4873765014880374, 1.3318024044937515, 1.5490458697304794, 1.5623675616503814, 2.0150232286328236, 2.6575756031477336, 2.095875670739080 7]))
('Mortgage', (489331, [0, 1, 2, 4, 8, 13, 16, 17, 19, 20, 23, 25, 33, 35, 52, 61, 65, 66, 67, 72, 83, 85, 92, 99, 101, 105, 118, 122, 136], [0.575036527947181, 0.550835980638457, 0.8508500490776176, 0.9232024079362433, 1.6215679809572925, 0.8764952055623286, 1.1144269425007924, 0.9136026490671826, 1.0513998333594627, 1.0303986727240377, 1.1099629425298851, 2.251146713072188, 1.4893258282689754, 1.1059436598862875, 1.634425605264384, 2.5046803899055288, 1.5517225216605337, 1.5253202269840656, 1.5720610629031821, 1.304101812502983, 1.3321698823117887, 1.3670716164440837, 1.3746216679127468, 1.4100615234996379, 1.4244628736383171, 1.5470465663714237, 1.4855913380568475, 1.4307038551955829, 1.8603029222034138]))
('Credit reporting credit repair services or other personal consumer reports', (489331, [0, 1, 2, 3, 4, 6, 7, 12, 20, 22, 29, 38, 47, 52, 61, 75, 82, 88, 90, 93, 104, 107, 111, 115, 126, 133, 135], [0.5488985039495818, 0.5901814078269182, 0.7976719210102666, 0.8682029190150282, 1.3848036119043647, 1.4472202801782657, 1.0824958814866794, 1.201354729246808, 1.103998577918612, 1.1170294774240428, 1.3375480127968176, 1.345477289377721, 1.6693421982632577, 1.53227400493536, 2.0492839553772506, 2.413428166031072, 1.420890645142908, 1.7275808324212005, 1.5971581202351413, 1.6499513431266724, 1.5112674214746178, 1.5574283356958047, 1.6730484491499595, 1.5406770162543015, 1.9837526065022408, 1.8025645430417265, 1.7321773386578605]))
('Debt collection', (489331, [5, 7, 16, 24, 27, 28, 29, 30, 55, 72, 77, 79, 82, 110, 114, 135, 148], [1.2797741584026325, 1.1546622735857912, 1.5920384892868462, 1.077676923136428, 1.5797607753252114, 1.2677201404140912, 1.1889315669305045, 1.2297128673050486, 1.4176627381588631, 1.4904020714319803, 1.5440283123987628, 1.5934559337834973, 1.8945208601905437, 1.7576041347668823, 1.6920695693610752, 2.309569784877147, 1.9060812159491554]))

Πίνακας 5: Οι 5 πρώτες γραμμές εξόδου.

Μετασχηματίζουμε τη λίστα σε Sparse Vector χρησιμοποιώντας τη συνάρτηση 'SparseVector' που παίρνει ως ορίσματα για κάθε πρόταση το μέγεθος του λεξικού, τη λίστα με τα indexes της κάθε λέξης στο λεξικό και τη λίστα με τα tfidfs της κάθε λέξης στην πρόταση.

```
#joined
map(key, value):
    #key: None
    #value: a tuple of ((word, string_label, sentence_index),
                        ((word_count_in_sentence, idf, word_index_in_lexicon),
                        tf))
    word = value[0][0]
    string_label = value[0][1]
    sentence_index = value[0][2]
    word_index_in_lexicon = value[1][0][2]
    idf = value[1][0][1]
```

```
    tf = value[1][1]
    tfidf = idf * tf
    emit((word, string_label, sentence_index), (word_index_in_lexicon, tfidf)

map(key, value):
    #key: None
    #value: a tuple of ((word, string_label, sentence_index),
                        (word_index_in_lexicon, tfidf)
    sentence_index = value[0][2]
    string_label = value[0][1]
    word_index_in_lexicon = value[1][0]
    tfidf = value[1][1]
    emit((sentence_index, string_label), [(word_index_in_lexicon, tfidf)])

reduce(key, value):
    #key: (sentence_index, string_label)
    #value: a list of (word_index_in_lexicon, tfidf) tuples
    list_of_values = [] #listof((word_index_in_lexicon, tfidf))
    for v in value:
        list_of_values.append(v)
    emit(key, list_of_values)

map(key, value):
    #key: None
    #value: a tuple of ((sentence_index, string_label), list_of_values)
    string_label = value[0][1]
    list_of_values = value[1] #listof((word_index_in_lexicon, tfidf))
    sorted_on_word_index_in_lexicon_list = sort(list_of_values, by key = value[1][0])
    emit(string_label, sorted_on_word_index_in_lexicon_list)

map(key, value):
    #key: None
    #value: a tuple of (string_label, sorted_on_word_index_in_lexicon_list)
    string_label = value[0]
    sorted_on_word_index_in_lexicon_list = value[1] #listof((word_index_in_lexicon,
                                                            tfidf))
    word_index_list = [] #list_of(word_index_in_lexicon)
    tfidf_list = [] #list_of(tfidf)
    for word_index_in_lexicon, tfidf in
        sorted_on_word_index_in_lexicon_list:
            word_index_list.append(word_index_in_lexicon)
            tfidf_list.append(tfidf)
    emit(string_label, SparseVector(lexicon_size, word_index_list, tfidf_list))
```

Listing 8: 'customer_complaints' RDD MapReduce

Έπειτα, μετατρέπουμε το RDD σε dataFrame για να εκπαιδεύσουμε μοντέλο της βιβλιοθήκης SparkML χρησιμοποιώντας τη συνάρτηση 'toDF'. Χρησιμοποιούμε τη συνάρτηση 'StringIndexer' ώστε να αντιστοιχήσουμε τις κατηγορίες που έχουμε, που είναι σε μορφή συμβολοσειράς, σε ετικέτες της μορφής 0, 1, 2 κτλ. Για το σκοπό αυτό χρησιμοποιούμε τη στήλη με τις ετικέτες από το DataFrame που φτιάξαμε. Η αντιστοίχηση σε αριθμητικές ετικέτες γίνεται ανάλογα με τη συχνότητα των κατηγοριών, με την πιο συχνή κατηγορία να αντιστοιχίζεται στην ετικέτα 0. Με τη συνάρτηση 'setHandleInvalid("skip")' και συγκεκριμένα με το skip ορίζουμε ότι αν στο test-set συναντήσουμε μία ετικέτα που δεν υπάρχει στο StringIndexer τότε απλά θα την αγνοήσει. Ύστερα, με την εντολή 'stringIndexer.fit' εκπαιδεύουμε το stringIndexer πάνω στα δεδομένα μας, ενώ με την εντολή 'stringIndexer-Model.transform' μετασχηματίζουμε τα δεδομένα σύμφωνα με το StringIndexer που εκπαιδεύσαμε.

Χωρίζουμε τα δεδομένα σε train και test set για να χρησιμοποιηθούν για την εκπαίδευση ενός μοντέλου. Συγκεκριμένα, κάνουμε stratified split ώστε κάθε set να έχει στοιχεία από κάθε κατηγορία. Το train set το δημιουργούμε

με τη συνάρτηση 'sampleBy' όπου δηλώνουμε τα fractions, δηλαδή τα ποσοστά από κάθε κλάση που θέλουμε να υπάρχουν στο train set. Θέσαμε το train set να αποτελείται απο το 75% του συνόλου των δεδομένων. Στη συνέχεια, δημιουργούμε το test set με τη βοήθεια της συνάρτησης 'subtract', η οποία επιστρέφει ένα νέο DataFrame που περιέχει τις γραμμές που υπάρχουν στο σύνολο των δεδομένων μας αλλά όχι στο train set. Το συνολικό DataFrame αποτελείται από 489,331 γραμμές, το train set απολείται από 367,277 γραμμές, ενώ το test από 99,481 γραμμές. Παρατηρούμε ότι το άθοισμα των γραμμών των επιμέρους sets δε δίνει το αρχικό άθροισμα. Άρα, για κάποιο λόγο οι συνάρτήσεις που χρησιμοποιήσαμε δεν κρατάνε όλες τις γραμμές του αρχικού DataFrame. Παραθέτουμε τους πίνακες με το πλήθος των γραμμών της κάθε κατηγορίας για το κάθε κομμάτι:

label	count
8.0	14851
0.0	142982
7.0	18807
1.0	106560
4.0	31456
11.0	7901
14.0	1489
3.0	32094
2.0	61403
17.0	16
10.0	8229
13.0	1743
6.0	19045
5.0	25131
15.0	1444
9.0	9438
16.0	291
12.0	6451

Πίνακας 6: Ο αριθμός των εγγραφών για κάθε κατηγορία για όλα τα δεδομένα.

label	count
8.0	11202
0.0	107195
7.0	14165
1.0	79978
4.0	23588
11.0	5950
14.0	1139
3.0	24099
2.0	46126
17.0	14
10.0	6193
13.0	1315
6.0	14313
5.0	18836
15.0	1095
9.0	7061
16.0	203
12.0	4805

Πίνακας 7: Ο αριθμός των εγγραφών για κάθε κατηγορία για το train set.

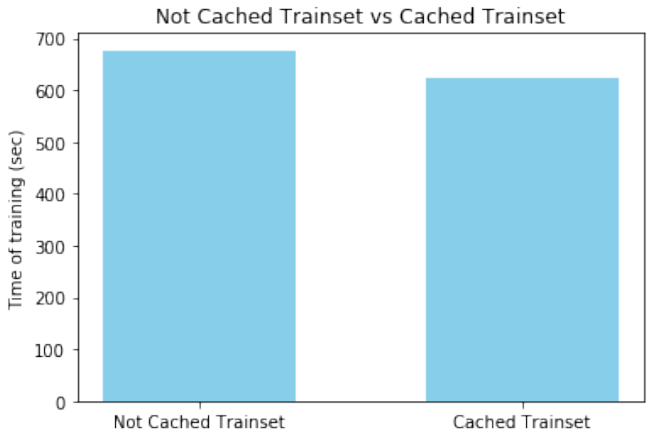
label	count
8.0	3637
0.0	19869
7.0	4537
1.0	22532
4.0	6405
11.0	1906
14.0	341
3.0	7613
2.0	15064
17.0	1
10.0	1977
13.0	415
6.0	4654
15.0	355
5.0	6147
9.0	2319
16.0	82
12.0	1627

Πίνακας 8: Ο αριθμός των εγγραφών για κάθε κατηγορία για το test set.

Χρησιμοποιούμε τα παραπάνω δεδομένα για την εκπαίδευση ενό μοντέλου Perceptron. Επίσης, εκπαιδεύουμε το μοντέλο δύο φορές, αρχικά χωρίς να κάνουμε cache το trainset και έπειτα χρησιμοποιώντας τον μηχανισμό cache. Για το σκοπό αυτό δημιουργούμε ένα for loop, που τρέχει δύο φορές, μία φορά για κάθε περίπτωση που αναφέραμε. Τα υπόλοιπα που θα αναφέρουμε βρίσκονται όλα μέσα στο for loop. Αρχικά, ορίζουμε τα επίπεδα του δικτύου. Το επίπεδο εισόδου έχει το μέγεθος του λεξικού με τις συχνότερες λέξεις και το επίπεδο εξόδου έχει το μέγεθος του αριθμού των κλάσεων, δηλαδή 18. Στη συνέχεια, πειραματιστήκαμε με τα κρυφά επίπεδα του δικτύου. Μερικά από αυτά που δοκιμάσαμε ήταν:

1. Να έχουμε ένα κρυφό επίπεδο με αριθμό κόμβων (lexicon_size+18)/2.
2. Να έχουμε ένα κρυφό επίπεδο με αριθμό κόμβων 2*(lexicon_size+18)/3.
3. Να έχουμε δύο κρυφά επίπεδα με αριθμούς κόμβων 25 και 10 αντίστοιχα.
4. Να έχουμε δύο κρυφά επίπεδα με αριθμούς κόμβων 2*lexicon_size/3 και 4*lexicon_size/9.

Όλα τα διαφορετικά μοντέλα που δοκιμάσαμε έβγαζαν την ίδια σχεδόν ακρίβεια. Καταλήξαμε ωστόσο να χρησιμοποιήσουμε την πρώτη επιλογή, δηλαδή ένα κρυφό επίπεδο με αριθμό κόμβων (lexicon_size+18)/2. Έπειτα, ορίζουμε το μοντέλο με τη συνάρτηση 'MultilayerPerceptronClassifier' και εκπαιδεύουμε το μοντέλο. Για κάθε μία από τις δύο περιπτώσεις κρατάμε το χρόνο που χρειάστηκε για να εκπαιδευτεί το μοντέλο. Ύστερα, με τη συνάρτηση 'transform' κάνουμε transform το test set πάνω στο μοντέλο, με αποτέλεσμα να πάρουμε μία νέα στήλη στο test DataFrame που περιλαμβάνει τις προβλέψεις. Από το test DataFrame επιλέγουμε να κρατήσουμε ξεχωριστά τις στήλες με τις πραγματικές ετικέτες και τις προβλέψεις και ορίζουμε έναν evaluator που μπορεί να υπολογίσει το accuracy. Για το σκοπό αυτό χρησιμοποιούμε την εντολή 'MulticlassClassificationEvaluator' και ορίζουμε ως μετρική το accuracy. Τέλος, υπολογίζουμε το accuracy score με βάση τις πραγματικές ετικέτες και τις προβλέψεις που απομονώσαμε νωρίτερα, με τη χρήση της συνάρτησης 'evaluate'. Σημειώνουμε ότι κρατάμε ξεχωριστά τα αποτελέσματα για την κάθε περίπτωση. Τυπώνουμε τα αποτελέσματα και φτιάχνουμε ένα διάγραμμα με τους χρόνους εκπαίδευσης για την κάθε περίπτωση:



Εικόνα 3: Διάγραμμα με τον χρόνο εκπαίδευσης χωρίς cached train set και με cached trainset.

	Test set accuracy
no cached Train Set	0.6143094841930117
cached Train Set	0.6122671073447467

Πίνακας 9: Accuracy για το test set.

Από τα παραπάνω αποτελέσματα διαπιστώνουμε ότι η εκπαίδευση του μοντέλου έχοντας κάνει πρώτα cache το train set είναι πιο γρήγορη, κατά 50 seconds περίπου. Η τεχνική caching είναι μία τεχνική βελτιστοποίησης όταν χρειαζόμαστε επαναλαμβανόμενη πρόσβαση σε κάποια δεδομένα. Κατά την εκπαίδευση ενός μοντέλου χρειαζόμαστε συνεχή πρόσβαση στο αντίστοιχο DataFrame, οπότε χωρίς να έχει γίνει cached πρέπει κάποιοι υπολογισμοί να ξαναγίνουν από την αρχή. Ακόμα και αν δε χωράνε όλα τα δεδομένα στη μνήμη η απόδοση θα βελτιωθεί και πάλι, ενώ σε περίπτωση που κάποιος executor αποτύχει, το caching βοηθάει να μειωθεί το cost of

recovery. Συνεπώς, είναι λογικό ο χρόνος εκπαίδευσης να είναι μικρότερος αφότου έχουμε κάνει cache το train set.

Το accuracy στο test set για τις δύο περιπτώσεις είναι 61.43% και 61.23%. Η διαφορά στο accuracy ανάμεσα στις δύο περιπτώσεις οφείλεται στο ότι το μοντέλο εκπαιδεύεται ξεχωριστά κάθε φορά, οπότε τα αντίστοιχα βάρη που προκύπτουν μπορεί να διαφέρουν λίγο. Θεωρούμε ότι η ακρίβεια του μοντέλου είναι αρκετά ικανοποιητική δεδομένου ότι έχουμε 18 κλάσεις.

Τέλος, δοκιμάσαμε να αφαιρέσουμε κάποιες κλάσεις με λίγες εγγραφές για να δούμε εάν θα βελτιωνόταν η ακρίβεια του μοντέλου. Συγκεκριμένα, αφαιρέσαμε τις κλάσεις με εγγραφές λιγότερες από 2000, δηλαδή αφαιρέσαμε 5 κλάσεις. Το κρυφό επίπεδο είναι της ίδιας λογικής με πριν, μόνο που κάνουμε κατευθείαν cache το train set αυτή τη φορά. Παραθέτουμε πάλι τους πίνακες με το πόσες γραμμές έχει η κάθε κατηγορία για το κάθε set:

label	count
8.0	14851
0.0	142982
7.0	18807
1.0	106560
4.0	31456
11.0	7901
3.0	32094
2.0	61403
10.0	8229
6.0	19045
5.0	25131
9.0	9438
12.0	6451

Πίνακας 10: Ο αριθμός των εγγραφών για κάθε κατηγορία για όλα τα δεδομένα.

label	count
8.0	11029
0.0	107420
7.0	14047
1.0	79831
4.0	23551
11.0	5871
3.0	24224
2.0	46099
10.0	6251
6.0	14334
5.0	18851
9.0	7014
12.0	4791

Πίνακας 11: Ο αριθμός των εγγραφών για κάθε κατηγορία για το train set.

label	count
8.0	3747
0.0	19686
7.0	4624
1.0	22589
4.0	6454
11.0	1973
3.0	7481
2.0	15080
10.0	1909
6.0	4637
5.0	6130
9.0	2382
12.0	1632

Πίνακας 12: Ο αριθμός των εγγραφών για κάθε κατηγορία για το test set.

Αυτή τη φορά ο χρόνος εκπαίδευσης του μοντέλου ήταν 414.64 seconds, δηλαδή αρκετά μικρότερος από τον χρόνο εκπαίδευσης για όλες τις κλάσεις. Είναι λογική η ύπαρξη κάποιας μείωσης στο χρόνο, καθώς μειώθηκαν τα δεδομένα πάνω στα οποία εκπαιδεύτηκε το μοντέλο. Η ακρίβεια αυτή τη φορά ήταν 61.77%. Παρατηρούμε ότι το accuracy αυξήθηκε ελάχιστα με την αφαίρεση των κλάσεων με λίγες εγγραφές. Ωστόσο, πάλι έχουμε 13 κλάσεις, οπότε πιθανώς να είχαμε ένα καλύτερο αποτέλεσμα εάν αφαιρούσαμε όλες τις κλάσεις με εγγραφές λιγότερες από 10,000, δηλαδή να μέναμε με 9 κλάσεις, αλλά τότε θα είχαμε διώξει τις μισές κατηγορίες προϊόντων.

Δεύτερος Τρόπος Επίλυσης

Σημειώνουμε ένα δεύτερο και πιο σύντομο τρόπο με τον οποίο μπορεί να υπολογιστεί η μετρική TF-IDF:

Το TF-IDF ορίζεται ως το γινόμενο των μετρικών TF, IDF, οι οποίες ορίζονται ως εξής:

- TF → Λόγος των εμφανίσεων μιας λέξης σε ένα κείμενο προς το συνολικό πλήθος λέξεων στο κείμενο.
- IDF → Λογάριθμος του λόγου του πλήθους των κειμένων προς το πλήθος των κειμένων που περιέχουν τη λέξη.

Ο ψευδοκώδικας για το πρόγραμμα MapReduce παρουσιάζεται παρακάτω.

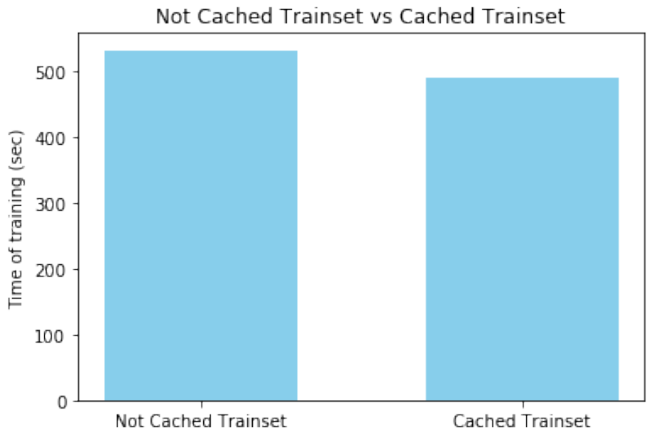
```
#customer_complaints RDD
flatmap(key, value):
    #key: None
    #value: a tuple of (productCategory, complaints)
    words = value[1].split(" ")
    emit(words)

map(key, value):
    #key: None
    #value: word
    word = re.sub('[^a-zA-Z]+', '', value)
    emit(word)

map(key, value):
    #key: None
    #value: word
    emit(value, 1)

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

map(key, value):
    #key: None
    #value: a tuple of (word, counter)
    word = value[0]
    emit(word)

map(key, value):
    #key: None
    #value: a tuple of (productCategory, complaints)
    string_label = value[0]
    words_list = complaints.split(" ")
    emit(string_label, words_list)

map(key, value):
    #key: None
    #value: a tuple of (string_label, words_list)
    string_label = value[0]
```

```
    words_list = value[1]
    list_of_sentence_words_in_lexicon = [word for word in words_list if
                                         word in broad_com_words]
    emit(string_label, list_of_sentence_words_in_lexicon)

#idf RDD
flatmap(key, value):
    #key: None
    #value: ((string_label, list_of_sentence_words_in_lexicon), sentence_index)
    words = set(value[0][1])
    for word in words:
        emit(word, 1)

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

map(key, value):
    #key: None
    #value: a tuple of (word, counter)
    word = value[0]
    counter = value[1]
    idf = log(number_of_complaints/counter)
    emit(word, idf)

#customer_complaints RDD
flatmap(key, value):
    #key: None
    #value: a tuple of ((string_label, list_of_sentence_words_in_lexicon),
    sentence_index)
    string_label = value[0][0]
    list_of_sentence_words_in_lexicon = value[0][1]
    sentence_length = length(list_of_sentence_words_in_lexicon)
    sentence_index = value[1]
    for word in list_of_sentence_words_in_lexicon:
        emit((word, string_label, sentence_index, sentence_length), 1)

reduce(key, value):
    #key: (word, string_label, sentence_index, sentence_length)
    #value: a list of 1's
    word_count_in_sentence = 0
    for v in value:
        word_count_in_sentence = word_count_in_sentence + v
    emit(key, word_count_in_sentence)

map(key, value):
    #key: None
    #value: a tuple of ((word, string_label, sentence_index, sentence_length),
    word_count_in_sentence)
    word = value[0][0]
    string_label = value[0][1]
    sentence_index = value[0][2]
```

```
    word_count_in_sentence = value[1]
    sentence_length = value[0][3]
    word_frequency_in_sentence = word_count_in_sentence/sentence_length
    word_index_in_lexicon = broad_com_words.value.index(word)
    emit(word, (string_label, sentence_index, word_frequency_in_sentence,
    word_index_in_lexicon))

#joined
map(key, value):
    #key: None
    #value: a tuple of (word, ((string_label, sentence_index,
                        word_frequency_in_sentence, word_index_in_lexicon),
                        idf))
    word = value[0]
    string_label = value[1][0][0]
    sentence_index = value[1][0][1]
    word_frequency_in_sentence = value[1][0][2]
    idf = value[1][1]
    tfidf = word_frequency_in_sentence*idf
    word_index_in_lexicon = value[1][0][3]
    emit((word, string_label, sentence_index), (tfidf, word_index_in_lexicon))

map(key, value):
    #key: None
    #value: a tuple of ((word, string_label, sentence_index),
                        (tfidf, word_index_in_lexicon))
    sentence_index = value[0][2]
    string_label = value[0][1]
    word_index_in_lexicon = value[1][1]
    tfidf_in_sentence = value[1][0]
    emit((sentence_index, string_label), [(word_index_in_lexicon, tfidf_in_sentence)])

reduce(key, value):
    #key: None
    #key: (sentence_index, string_label)
    #value: a list of (word_index_in_lexicon, tfidf_in_sentence) tuples
    list_of_values = [] #listof((word_index_in_lexicon, tfidf_in_sentence))
    for v in value:
        list_of_values.append(v)
    emit(key, list_of_values)

map(key, value):
    #key: None
    #value: a tuple of ((sentence_index, string_label), list_of_values)
    string_label = value[0][1]
    list_of_values = value[1] #listof((word_index_in_lexicon, tfidf_in_sentence))
    sorted_on_word_index_in_lexicon_list = sort(list_of_values, by key = value[1][0])
    emit(string_label, sorted_on_word_index_in_lexicon_list)

map(key, value):
    #key: None
    #value: a tuple of (string_label, sorted_on_word_index_in_lexicon_list)
    string_label = value[0]
    sorted_on_word_index_in_lexicon_list = value[1] #listof((word_index_in_lexicon,
                                                        tfidf_in_sentence))
    word_index_list = [] #list_of(word_index_in_lexicon)
    tfidf_list = [] #list_of(tfidf_in_sentence)
```

```
    for word_index_in_lexicon, tfidf_in_sentence in
        sorted_on_word_index_in_lexicon_list:
            word_index_list.append(word_index_in_lexicon)
            tfidf_list.append(tfidf_in_sentence)
    emit(string_label, SparseVector(lexicon_size, word_index_list, tfidf_list))
```

Listing 9: ML MapReduce - Δεύτερος Τρόπος



Εικόνα 4: Διάγραμμα με τον χρόνο εκπαίδευσης χωρίς cached train set και με cached trainset.

	Test set accuracy
no cached Train Set	0.587629590934662
cached Train Set	0.5828661721635685

Πίνακας 13: Accuracy για το test set.

Παρατηρούμε ότι το accuracy με αυτόν τον τρόπο βγαίνει περίπου 3% μικρότερο. Άρα, ο πρώτος τρόπος επίλυσης είναι προτιμότερος. Αυτό λογικά συμβαίνει επειδή με τον πρώτο τρόπο έχουμε μία αυξημένη συχνότητα, ώστε να αποφευχθεί η μεροληψία (bias) προς τα μεγαλύτερα αρχεία/σχόλια. Συγκεκριμένα, ο όρος 0.5 είναι ένας όρος εξομάλυνσης του οποίου ο ρόλος είναι να μειώνει τη συμβολή του δεύτερου όρου, που μπορεί να θεωρηθεί ότι κλιμακώνει την τιμή του tf προς τα κάτω ανάλογα με τη μεγαλύτερη τιμή του tf στο αντίστοιχο αρχείο. Τέλος, με αυτόν τον τρόπο μετριάζεται το γεγονός ότι σε μεγαλύτερα κείμενα παρατηρούμε υψηλότερες συχνότητες, απλώς και μόνο επειδή τα μεγαλύτερα κείμενα τείνουν να επαναλαμβάνουν τις ίδιες λέξεις.