ΔΠΜΣ Επιστήμης Δεδομένων και Μηχανικής Μάθησης

Εθνικό Μετσόβιο Πολυτεχνείο

Εξαμηνιαία Εργασία

Χρήση του Apache Spark σε SQL ερωτήματα και Μηχανική Μάθηση

Χρήστος Χαρίσης

A.M.:
03400121
Email:
christ.charisis@gmail.com

Γεώργιος Ουζουνίδης

A.M.:
03400108
Email:
giorgos_ouz@hotmail.com

Υπεύθυνος Καθηγητής: Νεκτάριος Κοζύρης.

Επιβλέπων: Ιωάννης Κωνσταντίνου.

Η εργασία κατατέθηκε για το μάθημα:
Διαχείριση Δεδομένων Μεγάλης Κλίμακας

July 26, 2021

Μέρος 1

Ζητούμενο 1

Στο πρώτο μέρος της εργασίας θα χρησιμοποιηθούν δεδομένα σχετικά με διαδρομές taxi στην Νέα Υόρκη. Οι διαδρομές αυτές έγιναν από τον Ιανουάριο έως τον Ιούνιο του 2015 και στην παρούσα εργασία θα χρησιμοποιηθεί ένα υποσύνολο των διαδρομών αυτών, ανάλογο με τους διαθέσιμους πόρους για επεξεργασία. Τα αρχεία αυτά είναι 2 αρχεία csv, που περιλαμβάνουν διάφορες πληροφορίες για την διαδρομή. 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 πουμαςδίνεται περιέχει πληροφορία για τις εταιρίες taxi. Η μορφή του φαίνεται στο παρακάτω παράδειγμα:

```
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
```

Figure 1: yellow_tripdata_1m.csv

Για το δευτερο αρχείο:

```
369367789289,1
369367789290,2
```

Figure 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).

Στο πρώτο ζητούμενο πρέπει τα αρχεία να εισαχθούν στο HDFS. Το HDFS είναι ένα κατανεμημένο σύστημα αρχείων το οποίο διαχειρίζεται δεδομένα σε πολλά διαφορετικά clusters. Αφού εισαχθούν στο HDFS, στο επόμενο ζητούμενο πρέπει να μετατραπούν σε ένα ειδικό format αρχείου, το Parquet.

Ζητούμενο 2

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. Παρότι 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 ψάχνω σε ένα σύνολο δεδομένων τις τιμές που είναι μεγαλύτερες από 100 και σε κάθε block του dataset έχω 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 τπου 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 IO και τη χρήση της μνήμης κι έχει τα χαρακτηριστικά που αναφέραμε.

Οπότε με τις κατάλληλες εντολές αποθηκεύουμε το αρχείο σε Parquet μορφή.

Ζητούμενο 3

Αφού μετατραπούν τα αρχεία σε Parquet format, καλούμαστε να υπολογίσουμε 2 ερωτήματα σχετικά με τις διαδρομές των ταξί της Νέας Υόρκης. Οι υπολογισμοί αυτοί θα πραγματοποιηθούν συνολικά 3 φορές, συνοδευόμενοι από τα αποτελέσματα και τους χρόνους εκτέλεσης από τις 3 ακόλουθες περιπτώσεις

1. Map Reduce Queries RDD API
2. SparkSQL με είσοδο το csv αρχείο
3. SparkSQL με είσοδο το parquet αρχείο

Γενικότερα, όταν θέλουμε να κάνουμε χρήση του Spark πρέπει να γίνουν ορισμένες ενέργειες στην αρχή του κάθε script. Αυτές είναι αρχικά ή δέσμευση ή δημιουργία ενός SparkSession. Αφού δημιουργηθεί και δοθεί όνομα για την εφαρμογή που θα τρέξουμε, θεμελιώνουμε την σύνδεση σε έναν Spark Cluster με την εντολή $'spark.sparkContext'$. Στη συνέχεια θα γίνει αναφορά στην μεθοδολογία που ακολουθήθηκε για την κάθε περίπτωση:

Map Reduce Queries RDD API

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. Καθώς διαβάζουμε το αρχείο, κάθε γραμμή του διαβάζεται σαν συμβολοσειρά και επιστρέφεται σαν ένα αρχείο RDD που περιέχει όλες τις γραμμές με την συγκεκριμένη μορφή. Συνεπώς, αφού κάποια στοιχεία αναφέρονται σε αριθμούς, θα πρέπει να γίνει και ανάλογη προεπεξεργασία.

Έπειτα, τα δεδομένα θα φιλτραριστούν με την χρήση της filter και της lambda function, με τελικό στόχο να αφαιρεθούν διάφορα outliers και να "καθαρίσουν" ώστε να έχουμε ρεαλιστικά αποτελέσματα. Στη συνέχεια, με την χρήση της map και της δικής μας συνάρτησης get_data κρατάμε μόνο τα δεδομένα που θα χρειαστούμε από την κάθε γραμμή. Αμέσως μετά, στο βήμα reduce, χρησιμοποιούμε την συνάρτηση $reduceByKey$ και αθροίζουμε όλα τα στοιχεία με βάση το κλειδί που στην συγκεκριμένη περίπτωση είναι η ώρα επιβίβασης. Τέλος, χρησιμοποιώντας ξανά την map, πραγματοποιούμε τους υπολογισμούς του μέσου όρου και ταξινομούμε τα αποτελέσματα. Αναλυτικότερα τα βήματα παρουσιάζονται στον παρακάτω ψευδοκώδικα

```
map(key,value):

#key:None
#value: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:list(longitude,latitude,1)

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:tuple(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)
```

Σχετικά με το δεύτερο ερώτημα, θα χωρίσουμε τα στοιχεία της κάθε γραμμής από το αρχείο '$yellow\_tripvendors\_1m.csv$' χρησιμοποιώντας την map. Έπειτα θα φιλτράρουμε τα δεδομένα από το '$yellow\_tripdata\_1m.csv$' και θα κρατήσουμε αυτά που θέλουμε μόνο με την χρήση της '$parse\_Data$'. Τελειώνοντας την προεπεξεργασία και την επιλογή των στοιχείων που θέλουμε, καταληγουμε με το id της κάθε διαδρομής και μία λιστα με την απόσταση και την διάρκεια της. Συνεχίζοντας, θα πρέπει να συνενώσουμε τα 2 RDDs με βάση το id. Έχοντας δεδομένα τύπου (K,V) και (K,W), όπου K είναι το κλειδί, η συνένωση θα επιστρέφει δεδομένα τύπου (K,(V,W)). Έπειτα το id αφαιρείται με την χρήση της map και τα δεδομένα έρχονται στην επιθυμητή μορφή. με την χρήση του '$reduceByKey$' θα ξεχωρίσει για κάθε κλειδί (πλεον vendor) το στοιχείο που περιέχει την μέγιστη απόσταση διαδρομής. Τέλος, ταξινομούμε τα αποτελέσματα με βάση την μέγιστη απόσταση για να τα παρουσιάσουμε. Περισσότερη λεπτομέρεια ακολουθεί στον ψευδοκώδικα:

```
#yellow_tripvendors_1m

map(key,value):

#key:None
#value:line

line=value.split(",")
id=line[0]
vendor=line[1]

emit(id,vendor)


#yellow_tripdata_1m

map(key,value):

#key:None
#value:line

line=value.split(",")
id=line[0]
datetime_start=strptime(line[1])
datetime_end=strptime(line[2])
duration=(datetime_end datetime_start).total_secs/60
```

```
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])


#joinedRDDs

map(key,value):

#key:None
#value:tuple(id,([distance,duration],vendor))

duration=value[1][0][1]
distance=value[1][0][0]
vendor=value[1][1]

emit(vendor,distance,duration)


map(key,value):

#key:None
#value:tuple(vendor,distance,duration)

vendor=value[0]

emit(vendor,value)


reduce(key,value):

#key:vendor
#value:list(vendor,distance,duration)

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))
```

SparkSQL με είσοδο το csv αρχείο

Ξεκινώντας , διαβάζουμε τα δεδομένα ως DataFrame όπου τα ονόματα των στηλών ακολουθούν την μορφή _c0, _c1, _c2 κλπ. Το ερώτημα που θα κάνουμε με χρήση της SQL γλώσσας ξεκινάει με την εντολή SELECT. με την εντολή αυτή θα επιλέξουμε την στήλη _c1 (η στήλη με την ημερομηνία) και με την χρήση των εντολών 'to_timestamp' και 'hour' κρατάμε μόνο την ώρα.

Ακόμα επιλέγουμε τις μέσες τιμές του γεωγραφικού πλάτους και μήκους αντίστοιχα, με την βοήθεια της avg. Η εντολή FROM χρησιμοποιείτε για να δηλωθεί το DataFrame από το οποίο θα τραβήξουμε τα δεδομένα. Αμέσως μετά, η εντολή WHERE φιλτράρει τα δεδομένα με τον ίδιο τρόπο που φιλτράρονται και καθαρίζονται τα outliers στην αντίστοιχη υλοποίηση με τα RDDs. Τέλος, με τις GROUP BY και ORDER BY ομαδοποιούμε και ταξινομούμε τα αποτελέσματα.

Στο δεύτερο ερώτημα δημιουργούμε ένα StructType για το αρχείο με τις διαδρομές, δηλώνοντας τον τύπο των στηλών στο Dataframe προκειμένου να κάνουμε πράξεις μέσα σε αυτές. Οπότε διαβάζουμε τα αρχεία με το schema αυτό και συνεχίζουμε την επεξεργασία. με την χρήση της where φιλτράρουμε τα δεδομένα με την ίδια λογική που ακολουθήθηκε στα RDDs. Με την χρήση της withColumn δημιουργούμε τις απαιτούμενες στήλες για να υπολογιστεί η απόσταση haversine, και έπειτα αφαιρούμε αυτές που δεν χρειαζόμαστε με την χρήση της drop. Ύστερα, με την εντολή join θα ενωθούν τα 2 αρχεία με βάση το id των διαδρομών. Θέτουμε ένα παράθυρο που βασίζεται στους vendors, δημιουργούμε μία στήλη με τις μέγιστες αποστάσεις και επιλέγουμε να κρατήσουμε μόνο τις γραμμές εκείνες που αντιστοιχούν στις μέγιστες αποστάσεις για κάθε vendor.

SparkSQL με είσοδο το parquet αρχείο

Στην περίπτωση της SQL με είσοδο parquet αρχεία θα χρειαστεί αρχικά να μετατρέψουμε τα Dataframes σε μορφή parquet. Έπειτα η όλη διαχείρηση θα είναι παρόμοια, αφού τα ερωτήματα με μορφή SQL δεν αλλάζουν. Το μόνο που αλλάζει είναι η μορφή του αρχείου, όπου η μορφή parquet βελτιστοποιεί το I/O και τη χρήση της μνήμης.

Ζητούμενο 4

Σε αυτό το ερώτημα εκτελέστηκαν οι ζητούμενες υλοποιήσεις και καταγράφηκαν οι χρόνοι εκτέλεσης. Τα αποτελέσματα για κάθε Query φαίνονται στα διαγράμματα που ακολουθούν.

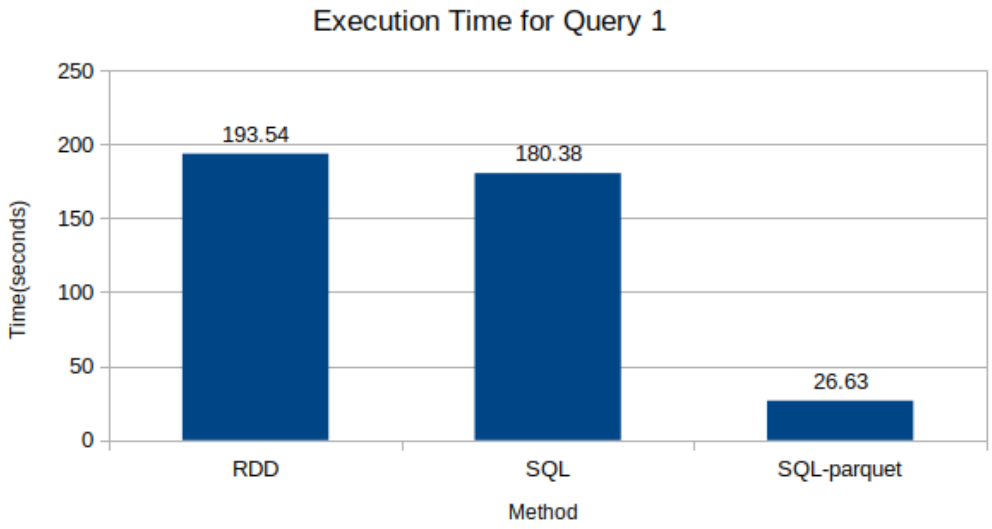

Figure 3: Χρόνοι εκτέλεσης Query 1.

Από το διάγραμμα για το Query 1 βλέπουμε πως οι εκτελέσεις του MapReduce και της SQL πάνω στα αρχεία csv δίνουν πολύ κοντίνο χρόνο με λίγο καλύτερο αυτόν της SQL, ο οποίος όμως είναι αρκετά μεγάλος. Αντίθετα, ο χρόνος εκτέλεσης της SQL πάνω στο αρχείο parquet είναι πολύ μικρότερος σε σχέση με τις δυο προηγούμενες εκτελέσεις. Εύκολα βγαίνει το συμπέρασμα πως προτιμούμε να εκτελούμε SQL πάνω σε parquet αρχεία αφού λόγο του μικρότερου αποτυπώματος που έχουν στη μνήμη και στο δίσκο.

Συγκεκριμένα, στα parquet αρχεία τα δεδομένα αποθηκεύονται στη μνήμη κατά στήλες, ενώ στην περίπτωση του csv τα δεδομένα αποθηκεύονται κατά εγγραφή. Επομένως, τα parquet 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 και όχι μόνο τις στήλες.

Ακόμα, διατηρείται 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 κοστοβόρα διαδικασία να τη διαβάσουμε. Έτσι, μπορούμε να περάσουμε κατευθείαν στο επόμενο block δεδομένων. Πιο συγκεκριμένα, στο ερώτημα αυτό, όπου για τον υπολογισμό του Query χρειαζόμαστε μόνο τρείς στήλες, η επιλογή χρήσης Parquet αρχείων είναι ιδανική.

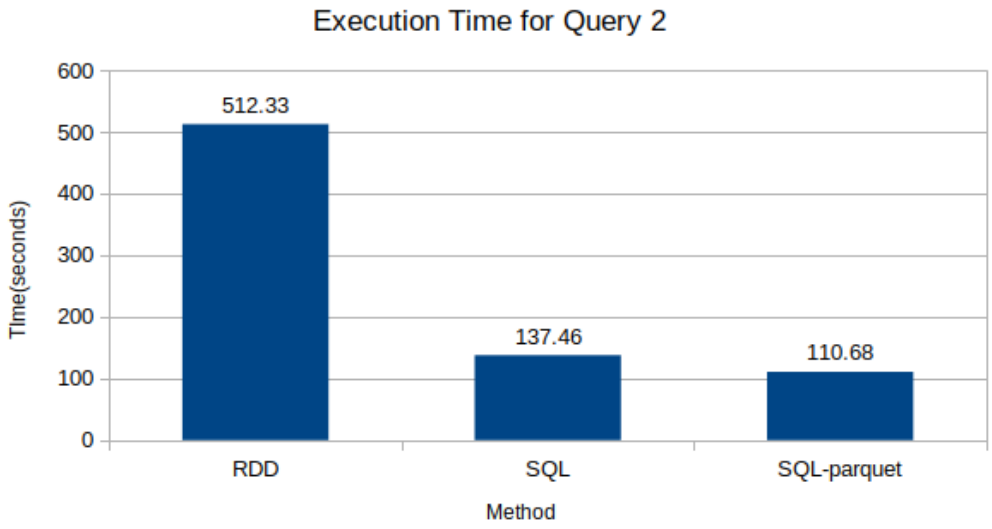

Figure 4: Χρόνοι εκτέλεσης Query 2.

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 MapReduce πάνω σε csv αρχεία. Αυτό πι ανώς 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 ακριβή διαδικασία. Παρόλα αυτά, οι πράξεις μεταξύ των στηλών, η αφαίρεση στηλών και η δημιουργία στηλών γίνονται σίγουρα πιο γρήγορα.

Αξίζει να σημειώσουμε 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 του πακέτου 'spark.sql'.

Ζητούμενο 5

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.

Το ερώτημα που έχουμε να αντιμετωπίσουμε είναι ένα ερώτημα συνένωσης με SparkSQL. Αφορά την συνενωση των πρώτων 100 γραμμών του πίνακα με τις εταιρείες ταξί με τον πίνακα των διαδρομών με και χωρίς βελτιστοποιητή. Στην πρώτη περίπτωση με την χρήση του βελτιστοποιητή, το join θα γίνει με την υλοποίηση 'BroadcastHashJoin'. Η υλοποίηση αυτή περιλαμβάνει το shuffle των κλειδών, μία χρονοβόρα διαδικασία για το Spark. Προκειμένου να αποφύγουμε ένα μέρος αυτής της διαδικασίας, πρέπει να στέιλουμε στους ίδιους mappers δεδομένα με ίδια κλειδιά. Σε πίνακες περιορισμένου μεγέθους, μπορούμε να στείλουμε σε κάθε κομβο τα δεδομένα του πίνακα του οποίου θέλουμε να συνενώσουμε σε έναν αλλον. Έπειτα δημιουργώντας ένα hashmap με κλειδί το κλειδί του join επιταγχύνουμε την αναζήτηση. Γι αυτό το Spark επιλέγει αυτή την υλοποίηση, διότι οι 100 γραμμές κατασκευάζουν έναν πίνακα μικρού μεγέθους κατάλληλο γι'αυτήν.

Αφού ρυθμίσουμε το Spark να μην επιλέγει η παραπάνω υλοποίηση, τελικά θα χρησιμοποιηθεί η υλοποίηση 'SortMergeJoin'. Η συγκεκριμένη υλοποίηση χρησιμοποιείται όταν οι πίνακες προς συνένωση είναι μεγάλοι σε μέγεθος. 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.

Παρατηρούμε πως η λειτουργία shuffle είναι αρκετά χρονοβόρα. Είναι λογικό το Spark να διαλέγει πρώτα να εκτελέσει 'BroadcastHashJoin', αφού είναι πιο γρήγορη υλοποίηση και το μέγεθος του πίνακα επιτρέπει να αποθηκευτεί στη μνήμη του mapper.

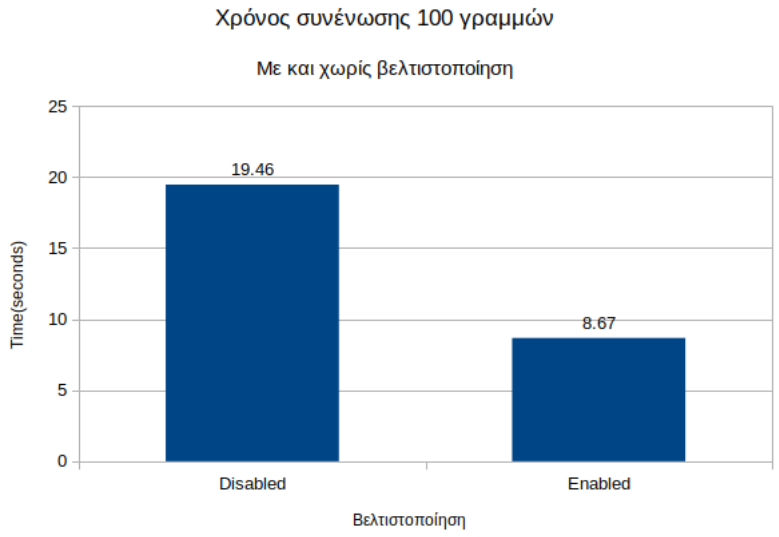

Figure 5: Απόδοση Βελτιστοποιητών

Τα προηγούμενα επιβεβαιώνονται από τα αποτελέσματα των χρόνων εκτελέσεων. Στις επόμενες εικόνες βλέπουμε τα πλάνα εκτέλεσης που παράγει ο βελτιστοποιητής σε κάθε εκτέλεση.

Και από αυτές τις εικόνες βλέπουμε πως το πλάνο με ενεργοποιημένο τον

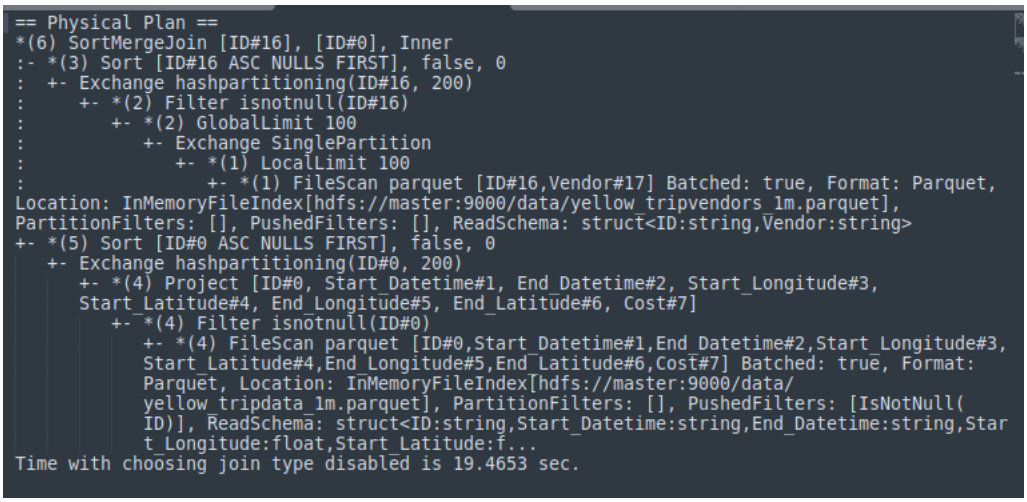

Figure 6: Πλάνο εκτέλεσης με απενεργοποιημένο βελτιστοποιητή.

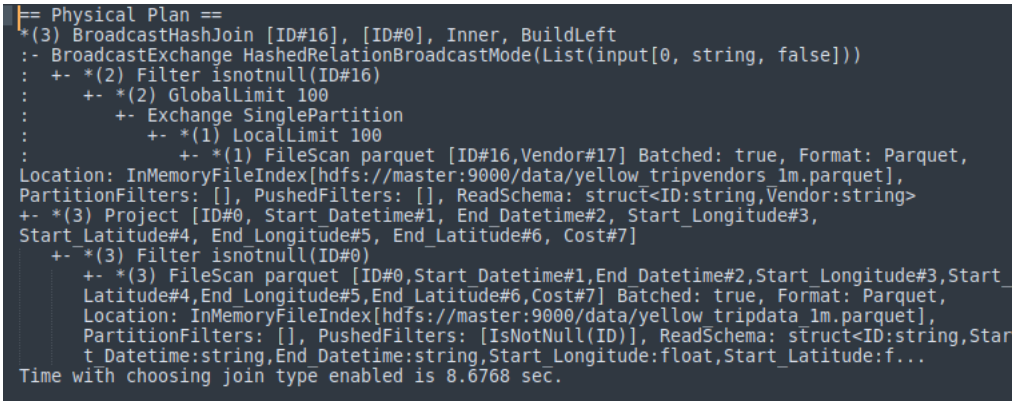

Figure 7: Πλάνο εκτέλεσης με ενεργοποιημένο βελτιστοποιητή.

βελτιστοποιητή είναι πιο συμπυκνωμένο και αποδοτικό, καθώς έχει επιλεχθεί το 'BroadcastHashJoin' και έχει προσαρμοστεί κατάλληλα η σειρά των τελεστών που εφαρμόζονται, ώστε να πραγματοποιηθεί το join.

Μέρος 2

Στο δεύτερο μέρος της εργασίας θα ασχοληθούμε με τη χρήση του Apache Spark για την κατηγοριοποίηση κειμένων. Θα χρησιμοποιήσουμε ένα dataset με πραγματικά δεδομένα, το οποίο περιγράφει παράπονα πελατών σε σχέσημε οικονομικά προϊόντα και υπηρεσίες. Κάθε ένα παράπονο έχει επισημανθεί με το σε ποια γενική κατηγορία προϊόντος αναφέρεται. Έτσι, τα δεδομένα αυτά μπορούν να χρησιμοποιηθούν για εκπαίδευση μοντέλων μηχανικής μάθησης αφού τους εφαρμοστεί πρώτα μια προεπεξεργασία ώστε να έρθουν στην κατάλληλη μορφή. Το συμπιεσμένο αρχείο που σας δίνουμε, περιλαμβάνει ένα comma-delimited αρχείο κειμένου (.csv) που ονομάζεται customer_complaints.csv περιλαμβάνει όλη την παραπάνω πληροφορία και έχει την εξής μορφή:

Το πρώτο πεδίο αποτελεί την ημερομηνία που κατατέθηκε το σχόλιο, το

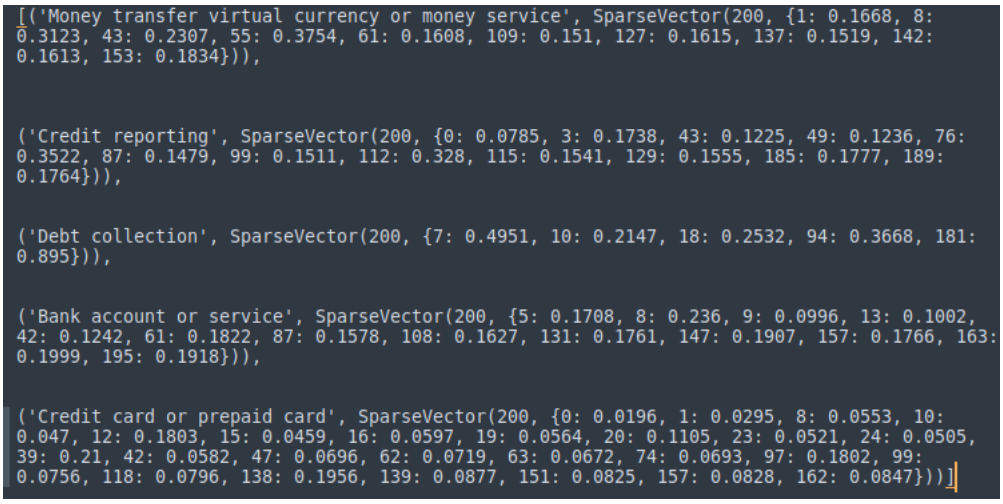

Figure 8: customer$_c$*omplaints.csv*

δεύτερο την κατηγορία προϊόντος ή υπηρεσίας που αναφέρεται, ενώ το τρίτο είναι το σχόλιο.

Για την εξαγωγή χαρακτηριστικών με στόχο την εκπαίδευση μοντέλων μηχανικής μάθησης, θα χρησιμοποιήσουμε την τεχνική TF-IDF, όπως αυτή αναφέρεται στη θεωρία του μαθήματος και στη εκφώνηση της άσκησης.

Ζητούμενο 1

Όπως και στο πρώτο μέρος της εργασίας, φορτώνουμε το αρχείο στο HDFS.

Ζητούμενο 2

Για να καθαρίσουμε τα δεδομένα, χρησιμοποιούμε την συνάρτηση filter_data καθώς και την 'getData' για να φέρουμε το dataset στην κατάλληλη μορφή. Ακόμα με τη συνάρτηση 'cache' αποθηκεύουμε την έξοδο ('customer_complaints') ώστε να γίνεται πιο γρήγορα η πρόσβαση σε αυτή, καθώς θα χρησιμοποιηθεί τόσο για την εύρεση του λεξικού όσο και για τη δημιουργία των sparse feature vectors. Περισσότερες λεπτομέριες για το καθάρισμα του dataset στον ψευδοκώδικα.

Ζητούμενο 3

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. Έπειτα, κρατάμε μόνο τις λέξεις που αποτελούνται από αλφαβητικούς χαρακτήρες, ενώ φιλτράρουμε τις λέξεις κρατώντας εκείνες που δεν ανήκουν στα stopwords. 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. 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.

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 με την εντολή 'zipWithIndex'. Αυτό το βήμα είναι απαραίτητο έτσι ώστε αργότερα να μπορέσουμε να ενώσουμε τις λέξεις που ανήκουν στην ίδια πρόταση. 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.

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 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 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'.

Στη συνέχεια, θα υπολογίσουμε το tf κάθε λέξης. Για αυτόν τον υπολογισμό χρειαζόμαστε τη συχνότητα μιας λέξης μέσα σε ένα σχόλιο, καθώς και το άθροισμα των λέξεων μέσα σε αυτό το σχόλιο. 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 το αντίστοιχο αριθμό των λέξεων σε κάθε σχόλιο, 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'.

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.

Ο ψευδοκώδικας όλης της διαδικασίας φαίνεται παρακάτω :

```
#customer_complaintsRDD

flatmap(key,value):

#key:None
#value:tuple(productCategory,complaints)

words=value[1].split("")

emit(words)



map(key,value):

#key:None
#value:word

word=re.sub(    [^   a   zAZ   ]+    ,       ,value)

emit(word)


```

```
map(key,value):

#key:None
#value:word

emit(value,1)



reduce(key,value):

#key:word
#value:list(1,1..)

word=key
counter=0
for v in value:
counter=counter+v

emit(word,counter)



map(key,value):

#key:None
#value:atupleof(word,counter)

word=value[0]

emit(word)



map(key,value):

#key:None
#value:tuple(productCategory,complaints)

string_label=value[0]
words_list=complaints.split("")
emit(string_label,words_list)



map(key,value):

#key:None
#value:tuple(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)

```

```
83
84
85 #idfRDD
86
87 flatmap(key,value):
88
89 #key:None
90 #value:((string_label,list_of_sentence_words_in_lexicon),
       sentence_index)
91
92 words=set(value[0][1])
93 for word in words:
94
95 emit(word,1)
96
97
98
99 reduce(key,value):
100
101 #key:word
102 #value:list(1,1..)
103
104 word=key
105 counter=0
106 for v in value:
107 counter=counter+v
108
109 emit(word,counter)
110
111
112
113 map(key,value):
114
115 #key:None
116 #value:tuple(word,counter)
117
118 word=value[0]
119 counter=value[1]
120 idf=log(number_of_complaints/counter)
121
122 emit(word,idf)
123
124
125
126 #customer_complaintsRDD
127
128 flatmap(key,value):
129
130 #key:None
131 #value:tuple((string_label,list_of_sentence_words_in_lexicon)
       ,
132
133 sentence_index)
134 string_label=value[0][0]
135 list_of_sentence_words_in_lexicon=value[0][1]
136 sentence_length=length(list_of_sentence_words_in_lexicon)
137 sentence_index=value[1]
138 for word in list_of_sentence_words_in_lexicon:
139
```

```
emit((word,string_label,sentence_index,sentence_length),1)



reduce(key,value):

#key:(word,string_label,sentence_index,sentence_length)
#value:list(1,1..)

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:tuple((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:tuple(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_sentenceidf
word_index_in_lexicon=value[1][0][3]

emit((word,string_label,sentence_index),(tfidf,
    word_index_in_lexicon))
```

```
195
196
197
198 map(key,value):
199
200 #key:None
201 #value:tuple((word,string_label,sentence_index),
202 (tfidf,word_index_in_lexicon))
203
204 sentence_index=value[0][2]
205 string_label=value[0][1]
206 word_index_in_lexicon=value[1][1]
207 tfidf_in_sentence=value[1][0]
208
209 emit((sentence_index,string_label),[(word_index_in_lexicon,
       tfidf_in_sentence)])
210
211
212
213 reduce(key,value):
214
215 #key:None
216 #key:(sentence_index,string_label)
217 #value:list(word_index_in_lexicon,tfidf_in_sentence)
218
219 list_of_values=[]#listof((word_index_in_lexicon,
       tfidf_in_sentence))
220 for v in value:
221 list_of_values.append(v)
222
223 emit(key,list_of_values)
224
225
226
227 map(key,value):
228
229 #key:None
230 #value:tuple((sentence_index,string_label),list_of_values)
231
232 string_label=value[0][1]
233 list_of_values=value[1] #list((word_index_in_lexicon,
       tfidf_in_sentence))
234 sorted_on_word_index_in_lexicon_list=sort(list_of_values,
       bykey=value[1][0])
235
236 emit(string_label,sorted_on_word_index_in_lexicon_list)
237
238
239
240 map(key,value):
241
242 #key:None
243 #value:tuple(string_label,
       sorted_on_word_index_in_lexicon_list)
244
245 string_label=value[0]
246 sorted_on_word_index_in_lexicon_list=value[1]#listof((
       word_index_in_lexicon,
247 tfidf_in_sentence))
```

```
word_index_list=[] #list_of(word_index_in_lexicon)
tfidf_list=[] #list_of(tfidf_in_sentence)
forword_index_in_lexicon,tfidf_in_sentencein
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)
```

Ζητούμενο 4

Ακολουθεί η έξοδος από 5 κείμενα:

```
[('Money transfer virtual currency or money service', SparseVector(200, {1: 0.1668, 8:
0.3123, 43: 0.2307, 55: 0.3754, 61: 0.1608, 109: 0.151, 127: 0.1615, 137: 0.1519, 142:
0.1613, 153: 0.1834})),


('Credit reporting', SparseVector(200, {0: 0.0785, 3: 0.1738, 43: 0.1225, 49: 0.1236, 76:
0.3522, 87: 0.1479, 99: 0.1511, 112: 0.328, 115: 0.1541, 129: 0.1555, 185: 0.1777, 189:
0.1764})),


('Debt collection', SparseVector(200, {7: 0.4951, 10: 0.2147, 18: 0.2532, 94: 0.3668, 181:
0.895})),


('Bank account or service', SparseVector(200, {5: 0.1708, 8: 0.236, 9: 0.0996, 13: 0.1002,
42: 0.1242, 61: 0.1822, 87: 0.1578, 108: 0.1627, 131: 0.1761, 147: 0.1907, 157: 0.1766, 163:
0.1999, 195: 0.1918})),


('Credit card or prepaid card', SparseVector(200, {0: 0.0196, 1: 0.0295, 8: 0.0553, 10:
0.047, 12: 0.1803, 15: 0.0459, 16: 0.0597, 19: 0.0564, 20: 0.1105, 23: 0.0521, 24: 0.0505,
39: 0.21, 42: 0.0582, 47: 0.0696, 62: 0.0719, 63: 0.0672, 74: 0.0693, 97: 0.1802, 99:
0.0756, 118: 0.0796, 138: 0.1956, 139: 0.0877, 151: 0.0825, 157: 0.0828, 162: 0.0847}))]
```

Figure 9: Οι 5 πρώτες γραμμές εξόδου

Στη συνέχεια 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. Έπειτα, μετατρέπουμε το RDD σε DataFrame για να εκπαιδεύσουμε μοντέλο της βιβλιοθήκης SparkML.

Ζητούμενο 5

Χωρίζουμε τα δεδομένα σε train και test set για να χρησιμοποιηθούν για την εκπαίδευση ενός μοντέλου. Συγ- κεκριμένα, κάνουμε stratified split ώστε κάθε set στο train να έχει στοιχεία από κάθε κατηγορία ίσα με το 70% της κάθε κατηγορίας. 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.

label	count
8.0	10501
0.0	100756
7.0	13196
1.0	74474
4.0	22177
11.0	5505
14.0	1045
3.0	22540
2.0	43070
17.0	14
10.0	5860
13.0	1244
6.0	13257
5.0	17576
15.0	1011
9.0	6575
16.0	198
12.0	4583

Figure 10: Train set - 343582 συνολικές εγγραφές

label	count
8.0	4369
0.0	24258
7.0	5540
1.0	27354
4.0	7846
11.0	2372
14.0	448
3.0	9269
2.0	18038
17.0	2
10.0	2267
13.0	508
6.0	5753
15.0	423
5.0	7464
9.0	2803
16.0	84
12.0	1878

Figure 11: Test set - 120676 συνολικές εγγραφές

Ζητούμενο 6

Χρησιμοποιούμε τα παραπάνω δεδομένα για την εκπαίδευση ενό μοντέλου Perceptron. Επίσης, εκπαιδεύουμε το μοντέλο δύο φορές, αρχικά χωρίς να κάνουμε cache το train set και έπειτα χρησιμοποιώντας τον μηχανισμό cache. Η αρχιτεκτονική που επιλέξαμε έχει αριθμό νευρώνων 200 που καταλήγουν σε 18 κλάσεις. Οι διαφορές ανάμεσα στην cached/non cached εκπαίδευση σχετικά με τους χρόνους και την ακρίβεια φάινεται παρακάτω :

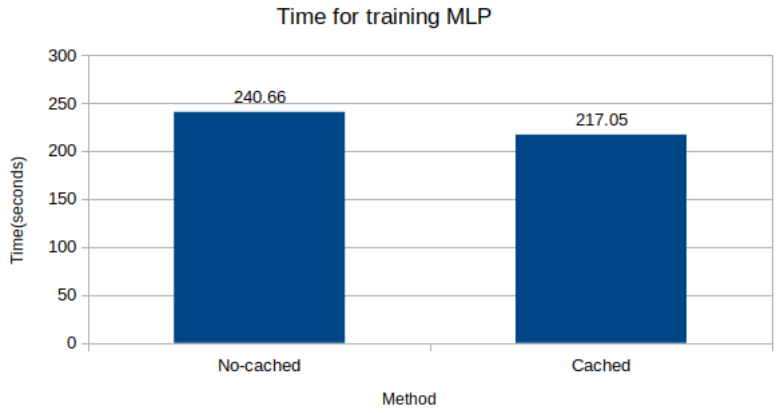

Figure 12: Cached-non cached training

Παρατηρούμε πως όταν στο train set χρησιμοποιούμε την cache έχουμε καλύτερους χρόνους καθώς εκμεταλλευόμαστε την τοπικότητα των δεδομένων στη RAM και δεν χρειάζεται να κάνουμε τόσες διαδρομές για δεδομένα στο δίσκο.