Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Δ.Π.Μ.Σ. Επιστήμης Δεδομένων
και Μηχανικής Μάθησης

Διαχείριση Δεδομένων Μεγάλης Κλίμακας
Εξαμηνιαία Εργασία

Χρήση του Apache Spark σε SQL ερωτήματα και Μηχανική Μάθηση

[redacted]

Περιεχόμενα

1. Μέρος 1ο: Υπολογισμός Αναλυτικών Ερωτημάτων με τα APIs του Apache Spark.

Τα δεδομένα που χρησιμοποιήθηκαν σε αυτό το μέρος της εργασίας είναι πραγματικά δεδομένα που αφορούν διαδρομές ταξί στην Νέα Υόρκη. Λόγω περιορισμένων πόρων, εμείς θα ασχοληθούμε με ένα υποσύνολο των δεδομένων μεγέθους 2 GB με 13 εκατομμύρια διαδρομές που πραγματοποιήθηκαν τον Μάρτιο του 2015, τα οποία είναι διαθέσιμα σε αυτό το link.

Στο συμπιεσμένο αρχείο που 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, ενώ το δεύτερο πληροφορία για τις εταιρείες ταξί. Προκειμένου να κρατήσουμε αυτό το report σύντομο δεν θα αναλύσουμε τη μορφή του κάθε dataset.

Περισσότερες πληροφορίες μπορούν να βρεθούν σε αυτό το site.

Με την βοήθεια των εντολών `wget,unzip` κατεβάζουμε τα δεδομένα στο virtual machine, και στην συνέχεια με τις παρακάτω ενδεικτικές εντολές φτιάχνουμε φακέλους στο hadoop file system και τοποθετούμε τα δεδομένα εκεί.

```
hadoop fs -mkdir hdfs://master:9000/project
hadoop fs -put dataset.csv hdfs://master:9000/project/.
```

Στο αρχείο `~/spark-2.4.4-bin-hadoop2.7/conf/spark-defaults.conf` προσθέσαμε τις παρακάτω παραμέτρους, προκειμένου να εκμεταλλευτούμε στο έπακρο τους υπολογιστικούς πόρους που μας δίνονται:

```
spark.master                    spark://master:7077
spark.submit.deployMode         client
spark.driver.host               master
spark.driver.cores              1
spark.driver.memory             1g
spark.executor.instances        2
spark.executor.cores            2
spark.executor.memory           3g
spark.driver.memory             1g
```

Τέλος, όταν θέλουμε να εκτελέσουμε κάποιο script, τότε το κάνουμε με την παρακάτω εντολή, ώστε να καταγράφεται το output σε κάποιο .log file, τα οποία είναι ζητούμενα προς παράδοση.

```
spark-submit script.py 2>\&1 | tee logs/log.txt
```

1.1 Ζητούμενο 1

Σε αυτό το ζητούμενο ζητείται να φορτώσουμε τα αρχεία .csv στο HDFS. Όπως αναφέρθηκε και πριν, αυτό γίνεται με τις εντολές:

```
hadoop fs -mkdir hdfs://master:9000/project1
hadoop fs -put yellow_tripdata_1m.csv hdfs://master:9000/project1/.
hadoop fs -put yellow_tripvendors_1m.csv hdfs://master:9000/project1/.
```

Αν εκτελέσουμε `hadoop fs -ls hdfs://master:9000/project1` τότε θα δούμε τα δύο αρχεία στο HDFS.

1.2 Ζητούμενο 2

Σε αυτό το ζητούμενο, ζητείται η μετατροπή των δύο αυτών .csv αρχείων, σε αρχεία parquet. Το script που χρειάζεται να εκτελέσουμε είναι το **convert.py**. Έτσι, καταλήγουμε με δύο αρχεία στο φάκελό μας στο HDFS, 2 .csv αρχεία των δεδομένων μας και τα αντίστοιχα 2 parquet.

Για την μετατροπή των αρχείων, πήρε **191.46 δευτερόλεπτα** το **διάβασμα** των .csv αρχείων ενώ **162.72 δευτερόλεπτα** η διαδικασία μετατροπή τους σε parquet, δηλαδή συνολικά 6 λεπτά στο σύνολο.

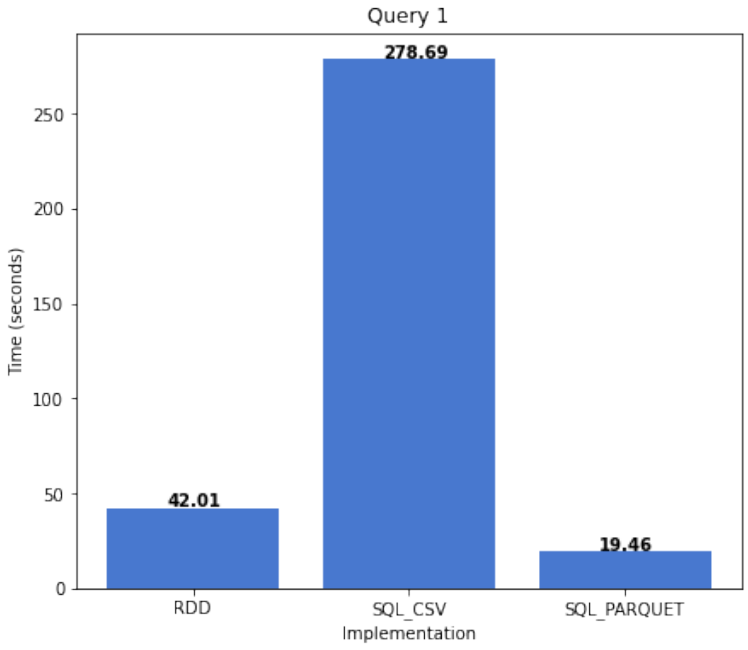
1.3 Ζητούμενα 3,4

Σε αυτά τα δύο ζητούμενα, ζητείται η υλοποίηση των δύο queries (θα αναφερόμαστε σε αυτά ως Q1 και Q2) με δύο διαφορετικές υλοποιήσεις. Επομένως, έτσι έχουμε τις 4 παρακάτω υλοποιήσεις:

- Q1 με Map Reduce - RDD API
- Q1 με Spark SQL (είσοδος είτε csv είτε parquet αρχεία)
- Q2 με Map Reduce - RDD API
- Q2 με Spark SQL (είσοδος είτε csv είτε parquet αρχεία)

Query 1: Στην αρχή, αποφασίσαμε να πραγματοποιήσουμε έναν απλό καθαρισμό των δεδομένων, όπου φροντίζουμε η ώρα επιβίβασης να προηγείται από την ώρα αποβίβασης, οι συντεταγμένες επιβίβασης και αποβίβασης να μην είναι οι ίδιες αλλά και οι συντεταγμένες να μην είναι μηδενικές. Επομένως, γράφουμε μια συνάρτηση που να υλοποιεί αυτά, και με χρήση της `filter` φιλτράρουμε τα δεδομένα μας. Ο ψευδοκώδικας map-reduce για το query αυτό φαίνεται στο block κώδικα 1.

Για την SQL υλοποίηση, φτιάξαμε το σχετικό query όπου κάνει και καθαρισμό και το aggregation, και στο πρόγραμμα έχει προστεθεί sys argument όπου το 0 φορτώνει parquet αρχείο και η επιλογή 1 φορτώνει csv αρχείο. Στο διάγραμμα 1 φαίνονται οι χρόνοι εκτέλεσης για τις 3 διαφορετικές υλοποιήσεις.



Σχήμα 1: Χρόνοι Εκτέλεσης για το Query 1.

Παρατηρούμε ότι η ταχύτερη υλοποίηση είναι με τη χρήση του SQL API και συγκεκριμένα εκεί που αξιοποιείται η χρήση του parquet datatype που optimized και βελτιστοποεί το I/O και τη χρήση της μνήμης.

```
map(key,value):
#key: _
#value: rdd row
    field_list = value.split(",")
    start_time = field[0]
    start_hour = extract_hour(start_time)
    start_latitude = field[1]
    start_longitude = field[2]
    emit(start_hour, (start_latitude, start_longitude,1))

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

map(key,value):
#key: _
#value: start_hour, (total_start_latitude, total_start_longitude,start_hour_freq)

    start_hour = value[0]
    total_start_longitude = value[1][0]
    total_start_latitude = value[1][1]
    start_hour_freq = value[1][2]
    mean_start_long = total_start_longitude / start_hour_freq
    mean_start_lat = total_start_latitude / start_hour_freq
    emit(start_hour, mean_start_long, mean_start_lat)
```

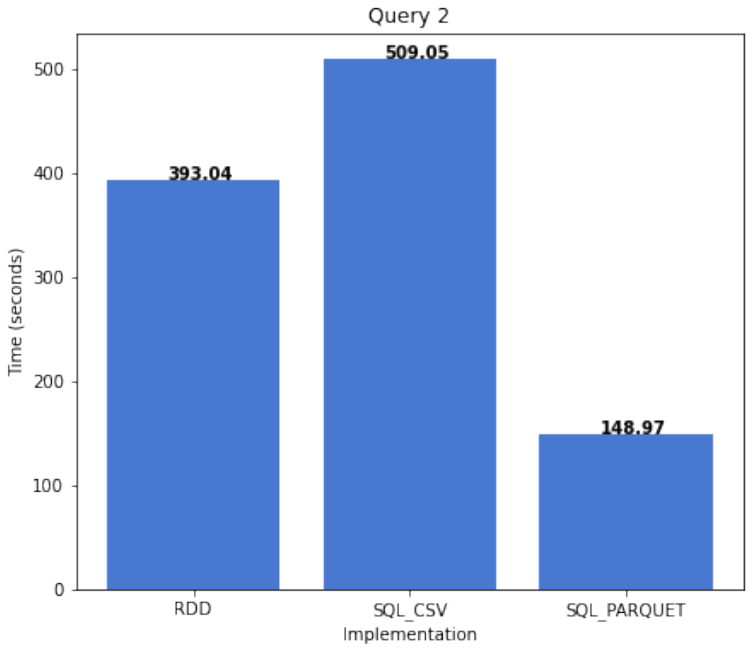
Listing 1: Ψευδοκώδικας Map-Reduce για το Query 1.

Query 2: Για αυτό το query, πραγματοποιείται ο ίδιος αρχικός καθαρισμός των δεδομένων. Και η ίδια νοοτροπία με πριν.

```
#tripdata
map(key,value):
    #key: _
    #values: rdd_row
    ID,(duration, HavDistance) = parsefrom(value)
    emit(ID,(duration,HavDistance))
#tripvendors
map(key,value):
    #key: _
    #value: rdd_row
    field_list = value.split(",")
    ID = field[0]
    vendor = field[1]
    emit(ID,vendor)
# after joining tripdata & tripvendors on ID
map(key,value):
#key: vendor.ID
#value: (ID,((duration, HavDistance),vendor))
    vendor = value[1][1]
    duration = value[1][0][0]
    HavDistance = value[1][0][1]
    emit(vendor, (duration, HavDistance))
reduce(key,value):
#key: vendor
#value: list(duration,HavDistance)
    vendor = key
    max_HavDist = 0
    duration = 0
    for v in value:
        if v[1] > max_HavDist:
            max_HavDist = v[1]
            duration = v[0]
    emit(vendor,(max_HavDist, duration))
map(key,value):
#key: vendor
#value: (max_HavDist, duration)
    vendor = key
    max_HavDist = value[0]
    duration = value[1]
    emit(vendor, max_HavDist, duration)
```

Listing 2: Ψευδοκώδικας Map-Reduce για το Query 2.

Στο διάγραμμα 2 φαίνονται οι χρόνοι εκτέλεσης για τις 3 διαφορετικές υλοποιήσεις.



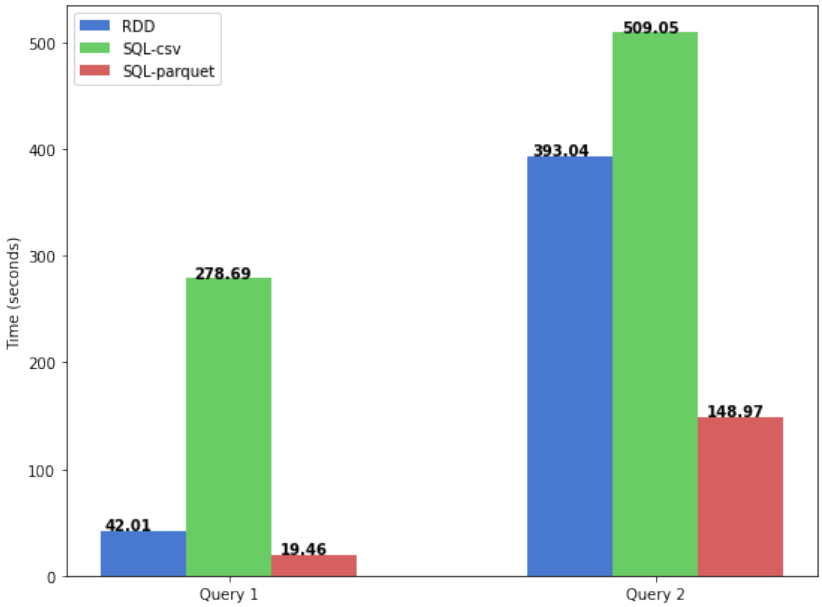
Σχήμα 2: Χρόνοι Εκτέλεσης για το Query 2.

Παρατηρούμε ότι και σε αυτό το query, ταχύτερη είναι η υλοποίηση του Spark SQL με χρήση parquet. Το συγκεκριμένο API αποθηκεύει off-heap σε δυαδική αναπαράσταση τα δεδομένα εξοικονομώντας έτσι παραπάνω μνήμη αφαιρώντας άχρηστες πληροφορίες. Επίσης, έχει την ικανότητα βελτιστοποίησης στα queries που γράφουμε σε μορφή Map-Reduce για RDD για ταχύτερη εκτέλεση.

Ακόμα ένα χαρακτηριστικό αυτού του API είναι ότι είναι πιο ευέλικτο καθώς επιτρέπει την διαχείριση διαφορετικών τύπων αρχείου όπως csv και parquet. Όπως είδαμε, το parquet είναι ταχύτερο, και αυτό οφείλεται στο ότι τα queries που ζητούνται στην εργασία συμφέρει να αποθηκεύονται κατά στήλες, κάτι το οποίο συμβαίνει στα parquet αρχεία, γι'αυτό και αποδίδουν καλύτερα.

Όσον αφορά το infer schema, στα αρχεία parquet, αυτό γίνεται αυτόματα, αφού αυτά κατά τη διάρκεια της εγγραφής τους διατηρούν επιπλέον πληροφορία. Έτσι, δεν χρειάζεται να χρησιμοποιείται το Infer Schema σε κάθε ανάγνωση, αφού αυτό είναι ήδη γνωστό.

Στο σχήμα 3 φαίνονται οι χρόνοι για κάθε υλοποίηση ομαδοποιημένοι για τα δυο διαφορετικά queries.



Σχήμα 3: Χρόνοι Εκτέλεσης για τα δύο Queries.

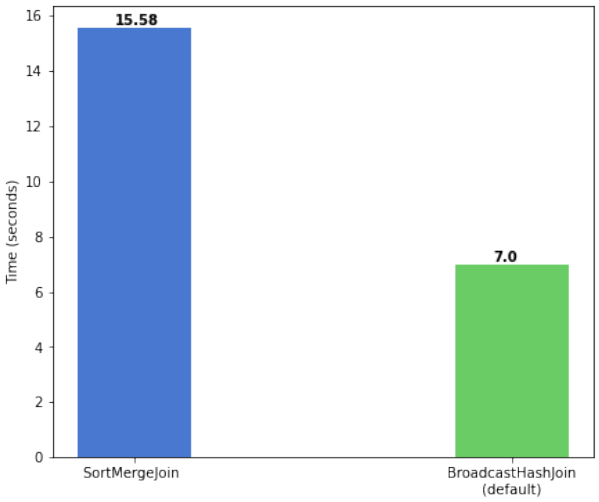
1.4 Ζητούμενο 5

Όταν γράφουμε κάποιον κώδικα στο Spark, τότε αυτό το μετατρέπει σε ένα λογικό πλάνο (logical plan) όπου δηλαδή φέρει σε ένα abstract επίπεδο την πληροφορία για το τι πρόκειται να γίνει (π.χ. σε ένα query) αλλά όχι για το πώς ακριβώς θα γίνει. Ένα τέτοιο πλάνο αποτελείται από relational operators (π.χ. join, filter κ.α.) και εκφράσεις όπως μετασχηματισμοί στηλών, συνθήκες φιλτραρίσματος και άλλα.

Όπως είπαμε, το logical plan είναι σε πολύ αφαιρετικό επίπεδο και για γίνει κατανοητό από τους workers πρέπει να μετατραπεί σε physical plan, το οποίο γεφυρώνει το χάσμα μεταξύ logical plan και RDD και περιέχει πληροφορίες για το πώς θα εκτελεστεί το πλάνο. Το Physical plan είναι ένας συγκεκριμένος αλγόριθμος, όπως στην περίπτωσή μας που είναι BroadcastHashJoin ή SortMergeJoin. Το ποιος αλγόριθμος θα επιλεγεί το αποφασίζει το spark ανάλογα με το πρόβλημα, δηλαδή τι διαθέσιμους πόρους έχουμε, πόσο απαιτητικό είναι κάποιο query κ.τ.λ.

Το Spark έχει by default ενεργοποιημένη την Optimized στρατηγική, δηλαδή έχει έναν optimizer όπου αυτός επιλέγει ποιον αλγόριθμο θα χρησιμοποιήσει. Στο ζητούμενο αυτό, εξετάζουμε τους χρόνους εκτέλεσης έχοντας απενεργοποιήσει τον optimizer σε σχέση με τον ενεργοποιημένο, πάνω σε ένα query όπου κάνουμε join τις πρώτες 100 γραμμές των tripvendors δεδομένων με τα δεδομένα tripdata.

Ο optimizer διαλέγει τον BroadcastHashJoin, ενώ όταν τον απενεργοποιούμε επιλέγεται ο αλγόριθμος SoftMergeJoin ως physical plan. Το Physical plan μπορούμε να το δούμε με χρήση της μεθόδου `explain()`. Οι χρόνοι εκτέλεσης του query για τα δυο διαφορετικά physical plans φαίνονται στο σχήμα 4.



Σχήμα 4: Χρόνοι Εκτέλεσης για διαφορετικά physical plans.

Όπως είναι εμφανές, το default του optimizer κάνει τον μισό χρόνο να εκτελέσει το query. Η λογική του BroadcastHashJoin είναι ότι όταν ένα από τα δύο dataframes (στην δική μας περίπτωση οι 100 γραμμές του tripvendors) είναι αρκετά μικρό να χωρέσει στη μνήμη, τότε μπορεί να γίνει broadcast σε όλους τους executors του cluster όπου βρίσκεται το μεγαλύτερο dataframe και να γίνει στην πορεία το hash join. Η ιδέα του hash join είναι ότι φορτώνεται το μικρό dataset στη μνήμη σε ένα hashmap, σε μια key-value δομή και για κάθε εγγραφή του μεγάλου dataset αναζητείται το αντίστοιχο key στο hashmap. Η αναζήτηση γίνεται σε χρόνο $\mathcal{O}(1)$. Σε αυτή την εκτέλεση μεθοδεύονται μόνοι οι mappers και δεν γίνεται shuffling δεδομένων στο cluster, γι'αυτό και επιταχύνεται πολύ η διαδικασία.

Η λογική του SoftMergeJoin όπου το πήραμε απενεργοποιώντας τον optimizer, είναι ότι λειτουργεί όταν τα κλειδιά μπορούν να ταξινομηθούν και μάλιστα κάνει αυτή την υπόθεση όταν εκτελείται. Αποτελείται από τρία στάδια, ένα στάδιο shuffle όπου γίνεται repartition των δύο πινάκων ως προς το join-κλειδί κατά μήκος των partitions του cluster ώστε σε κάθε partition κάθε εγγραφή και στους δύο πίνακες να έχει το ίδιο κλειδί, από ένα στάδιο sort όπου ταξινομούνται τα δεδομένα παράλληλα σε κάθε partition και ένα στάδιο merge όπου συνενώνονται τα δεδομένα στα partition με επανάληψη πάνω στα στοιχεία του dataset με βάση το κλειδί του join.

Το shuffling και το sorting προσθέτουν επιπλέον κόστος στο query κάτι που φαίνεται και στα αποτελέσματα καθώς το soft merge join είναι πιο αργό.

2. Μέρος 2ο: Machine Learning - Κατηγοριοποίηση Κειμένων

Σε αυτό το μέρος της εργασίας ασχολούμαστε με κατηγοριοποίηση κειμένων από το customer complaints dataset, όπου το σύνολο δεδομένων αποτελείται από τρία πεδία, την ημερομηνία που κατατέθηκε το σχόλιο, την κατηγορία του σχολίου καθώς και το ίδιο το σχόλιο.

Εμείς καλούμαστε να κάνουμε κάποια προεπεξεργασία στα δεδομένα, να κάνουμε εξαγωγή χαρακτηριστικών καθώς και να εκπαιδεύσουμε ένα μοντέλο Multi Layer Perceptron πάνω στα τελικά δεδομένα.

2.1 Ζητούμενα 1,2,3

Σε αυτά τα ζητούμε ανεβάζουμε τα δεδομένα στο hdfs όπως και πριν και μετά ξεκινάμε την προεπεξεργασία. Αυτή περιλαμβάνει το φιλτράρισμα των σχολίων των οποίων η ημερομηνία ξεκινάει με «201», η αφαίρεση των γραμμών με κενό κάποιο από τα πεδία, η αφαίρεση χαρακτήρων που δεν ανήκουν στο αλφάβητο αλλά και stopwords καθώς και η δημιουργία ενός λεξικού με τις K πιο συχνές λέξεις όπου το K θεωρείται υπερπαράμετρος του προβλήματος. Ο ψευδοκώδικας Map-Reduce για την προεπεξεργασία φαίνεται στο block 4.

```
map(key,value)
#key: _
#value: dataset row
    value = value.startswith("201")
    field_list = value.split(",")
    data = field[0]
    category = field[1]
    comment = field[2]
    filter( length(field_list) == 3)
    comment = remove_xxs(comment)
    comment = remove_nonalphabetchars(comment)
    comment = remove_stopwords(comment)
    comment = comment.lower() #lowercase only
    comment = comment.isnotnull()
    emit(category,comment)
```

Listing 3: Ψευδοκώδικας Map-Reduce για την προεπεξεργασία

Στην συνέχεια, παρουσιάζουμε τον κώδικα για την μέτρηση συχνότητας εμφάνισης λέξεων, όπου για τη δημιουργία λεξικού κρατάμε τις K πιο συχνές.

```
#word counts for lexicon creation
map(key, value):
#key: category
#value: comment
    word_list = comment.split(" ")
    emit(word_list,1)

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

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

Listing 4: Ψευδοκώδικας Map-Reduce για την δημιουργία λεξικού.

Ενδεικτικά, για K=60 το λεξικό που δημιουργήθηκε είναι το εξής: ('credit', 'account', 'report', 'information', 'payment', 'would', 'loan', 'debt', 'bank', 'told', 'company', 'received', 'card', 'called', 'time', 'never', 'payments', 'sent', 'reporting', 'letter', 'pay', 'back', 'paid', 'get', 'also', 'mortgage', 'call', 'amount', 'said', 'due', 'made', 'one', 'accounts', 'number', 'phone', 'could', 'days', 'balance', 'money', 'late', 'collection', 'still', 'since', 'asked', 'nt', 'consumer', 'date', 'years', 'please', 'even', 'name', 'contacted', 'home', 'dispute', 'file', 'month', 'make', 'check', 'request', 'interest').

2.2 Ζητούμενο 4

Σε αυτό το στάδιο καλούμαστε να κάνουμε εξαγωγή χαρακτηριστικών, και συγκεκριμένα να υπολογίσουμε την μετρική tfidf. Ο παρακάτω ψευδοκώδικας στο block 5, κρατάει από κάθε γραμμή κάθε σχολίου μόνο τις λέξεις οι οποίες ανήκουν στο λεξικό το οποίο δημιουργήσαμε προηγουμένως. Στη συνέχεια, με τη βοήθεια του zipwithIndex δίνει και ένα document index σε κάθε γραμμή.

Στη συνέχεια, υπολογίζουμε το IDF στο block 6

```
#first, keep words inside the lexicon only.
map(key,value):
#key: category
#value: list_of_words
    list_of_words = [x for x in list_of_words if x in lexicon]
    emit((category,list_of_words),doc_index)
#we used zipWithIndex().
```

Listing 5: Ψευδοκώδικας Map-Reduce για φιλτράρισμα λέξεων λεξικού και zipwithindex.

```
flatMap(key,value):
#key: (category, list_of_words)
#value: doc_index
    word = key[1]
    emit(word,1)
reduce(key,value):
#key: word
#value: list of 1s
    word_appearances_in_corpus = 0
    for v in value:
        word_appearances_in_corpus += v
    emit(word, word_appearances_in_corpus)

map(key, value):
#key: word
#value: word_appearances_in_corpus
    N = total_documents_count
    IDF = log(N/word_appearances_in_corpus)
    emit(word,IDF)
```

Listing 6: Ψευδοκώδικας Map-Reduce για υπολογισμό IDF.

Τελικά, το δυσκολότερο και μεγαλύτερο Map-Reduce που κληθήκαμε να γράψουμε είναι ο υπολογισμός του TFIDF, και συγκεκριμένα να φέρουμε τα δεδομένα μας στην μορφή: (K, (ind1, ind2, ..., indM), (tfidf1, tfidf2, tfidfM)) όπου $\texttt{ind}_i$ η θέση της i-οστής λέξης του κειμένου στο λεξικό που βρήκαμε, και $\texttt{tfidf}_i$ η τιμή της μετρικής αυτής για τη λέξη αυτή στο δεδομένο κείμενο. Ο ψευδοκώδικας φαίνεται στο block 7.

Αφού κάνουμε τους παραπάνω υπολογισμούς, τυπώνουμε για τα πρώτα 5 κείμενα την έξοδο στην ζητούμενη μορφή, τα οποία φαίνονται στα results στο αρχείο `tf-idf.txt`.

```
flatMap(key,value):
#key: (category, list_of_words)
#value: doc_index
    category = key[0]
    comment_length = length(key[1])
    doc_index = value
    for word in list_of_words:
        emit((word,category,doc_index, comment_length),1)

reduce(key,value):
#keys and values are eminent above
    count = 0 #number of word appearances in the comment
    for v in value:
        count += v
    emit((word,category,doc_index,comment_length),count)

#TFIDF calculation
map(key,value):
#keys and values are taken from the previous emit
    tf = count/comment_length
    TFIDF = tf*[x for x in idf_list if word==idf_list[x]]
    emit((word,category,doc_index),TFIDF)

#we also need the word's index in the lexicon, dropping word.
map(key,value):
    word_id = lexicon.value.index(word)
    emit((doc_index,category),(word_id, TFIDF))
#reduce step:
reduce(key,value):
    emit((doc_index,category),list[word_id,TFIDF])
map(key,value):
    sortBy(TFIDF)
    emit(category, (word_id, TFIDF))
map(key,value):
    key = category
    SVector = (lexicon_size, [i[0] for i in value[1]], [j[1] for j in value[1]])
    emit(category, SVector)
```

Listing 7: Ψευδοκώδικας Map-Reduce για υπολογισμό TFIDF.

2.3 Ζητούμενο 5

Στην πορεία μετατρέπουμε το παραπάνω RDD σε Spark Dataframe καθώς και το αποθηκεύουμε σε μορφή parquet στο HDFS. Έπειτα, εφαρμόζουμε stratified split με σκοπό κάθε set να έχει στοιχεία από κάθε κατηγορία.

Συνολικά έχουμε **18 μοναδικές κατηγορίες**. **Το Train Set έχει συνολικά 389367(80%) γραμμές ενώ το Test Set 70705 (20%).** Το split των δεδομένων έχει ως εξής:

Train Set

category	count
Debt collection	84918
Virtual currency	12
Payday loan	1362
Money transfers	1211
Checking or savin...	15203
Payday loan title...	5170
Mortgage	48855
Prepaid card	1164
Credit card or pr...	25643
Credit reporting	25089
Credit reporting ...	113268
Consumer Loan	7580
Credit card	14997
Bank account or s...	11876
Vehicle loan or l...	6577
Money transfer vi...	6285
Other financial s...	240
Student loan	19917

Test Set

category	count
Debt collection	16484
Virtual currency	2
Payday loan	361
Money transfers	266
Checking or savin...	3587
Payday loan title...	1215
Mortgage	11846
Prepaid card	269
Credit card or pr...	5910
Credit reporting	4224
Credit reporting ...	10532
Consumer Loan	1759
Credit card	3557
Bank account or s...	2751
Money transfer vi...	1460
Vehicle loan or l...	1534
Other financial s...	51
Student loan	4897

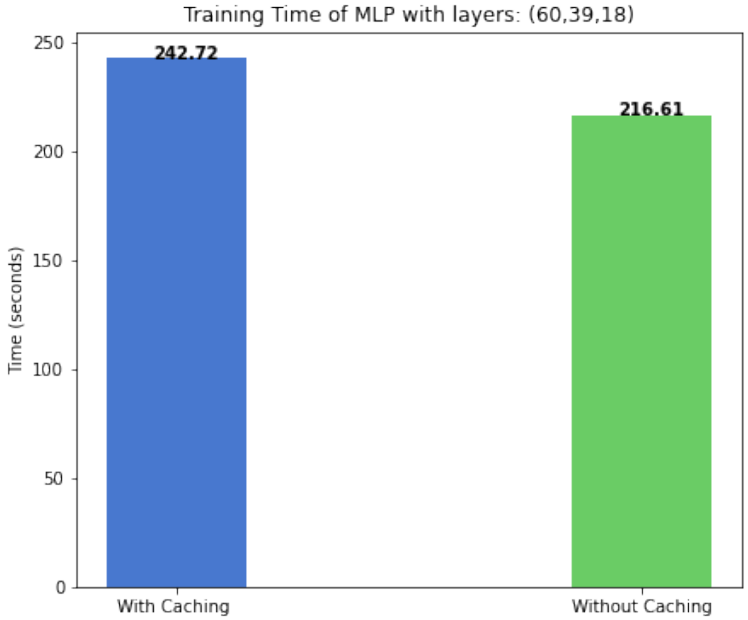
Σχήμα 5: Διαχωρισμός δεδομένων.

2.4 Ζητούμενο 6

Τέλος, έχοντας κάνει εξαγωγή χαρακτηριστικών, είμαστε έτοιμοι να εκπαιδεύσουμε το μοντέλο Multi Layer Perceptron. Πειραματιστήκαμε με διαφορετικά μεγέθη λεξικών, όμως λόγω περιορισμών της μνήμης κατασταλάξαμε σε K=60. Όσον αφορά τα layers του MLP, μετά από πειραματισμό, αποφασίσαμε ότι η τελική μορφή είναι τα επίπεδα να έχουν τη μορφή [60,39,18]. Η νοοτροπία είναι το επίπεδο εισόδου να έχει μέγεθος όσο και το μέγεθος του λεξικού, το επίπεδο εξόδου μέγεθος όσο και ο αριθμός των κλάσεων, και το κρυφό επίπεδο να έχει μέγεθος τον μέσο όρο των άλλων δύο επιπέδων.

Αυτή η απλή αρχιτεκτονική έχει 56.10 % ακρίβεια στο test set.

Η εκπαίδευση πραγματοποιήθηκε χωρίς caching αλλά και έχοντας κάνει caching στο train set, με σκοπό να δούμε τις διαφορές στον χρόνο εκτέλεσης. Στο διάγραμμα 6 φαίνονται οι αντίστοιχοι χρόνοι εκτέλεσης.



Σχήμα 6: Διαχωρισμός δεδομένων.

Όπως παρατηρούμε, η χρήση της cache φαίνεται να καθυστερεί την εκπαίδευση. Κάτι τέτοιο σε καμία περίπτωση δεν βγάζει νόημα, αν έχουμε γίνει όλα σωστά. Το caching μπορεί να χρησιμοποιηθεί για να τραβάμε σύνολα δεδομένων σε ένα cluster από τη memory cache. Αυτό μπορεί να φανεί χρήσιμο για την πρόσβαση σε αρχεία που χρειάζονται επανειλημμένα όπως για παράδειγμα στο training του MLP. Επομένως, θα περιμέναμε κάτι τέτοιο θα ελαττώσει τον χρόνο εκπαίδευσης.

Ο λόγος που στην περίπτωση μας **δεν** επιταχύνει την εκπαίδευση, είναι διότι όπως μπορεί να επιβεβαιωθεί από τα logs, συγκεκριμένα στο αρχείο `logs/train.txt` κατά τη διάρκεια εκπαίδευσης με caching, κάποιοι workers απέτυχαν, επομένως κάποια tasks έπρεπε να επανεκκινηθούν από κάποιο προηγούμενο στάδιο, οπότε κάποια tasks έγιναν παραπάνω από μια φορά.

Μπορεί αυτό το ατυχές γεγονός να καθυστέρησε λίγο την εκπαίδευση του MLP, όμως αναδεικνύει το fault tolerance που έχει ένα framework σαν το spark, καθώς η αποτυχία κάποιων workers είναι πολύ σύνηθες γεγονός το οποία γίνεται όλο και πιο συχνό όσο μεγαλώνει ο αριθμός των υπολογιστικών πυρήνων στον cluster μας.

Ανεξάρτητα από αυτό, λόγω του μικρού μεγέθους λεξικού άρα και του Sparse vector, καθώς και του μικρού αριθμου (100) max iterations του MLP, η επιτάχυνση που θα βλέπαμε με τη χρήση caching θα ήταν μικρή μεν, αισθητή δε.