Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλετρολογων Μηχανικων & Μηχανικων Υπολογιστων

Διαχείριση Δεδομένων Μεγάλης Κλίμακας, Εαρινό Εξάμηνο 2020

Εξαμηνιαία Εργασία

Πλεύρης Κωνσταντίνος, ΔΠΜΣ $\varepsilon.\delta\varepsilon.\mu^2$
Ράπτης Παναγιώτης, ΔΠΜΣ $\varepsilon.\delta\varepsilon.\mu^2$

Αθήνα, 30 Ιουλίου 2020

1 Machine Learning - Κατηγοριοποίηση Κειμένων

Η συγκεκριμένη εργασία, αφορά την κατηγοριοποίηση κειμένων, και ειδικότερα παράπονα πελατών σχετικά με οικονομικά προϊόντα, παρμένο από ένα πραγματικό dataset[1]. Το CSV αρχείο που συμπεριλαμβάνει τα δεδομένα (*customer_complaints.csv*) είναι comma-delimited και κάθε εγγραφή του αποτελείται από την ημερομηνία που κατατέθηκε το παράπονο, την κατηγορία στην οποία ανήκει καθώς και εν τέλει το βασικό σώμα του παραπόνου.

Και αυτό το μέρος της εργασίας αποτελείται από δύο βασικά υπομέρη. Το πρώτο κομμάτι αφορά την απαραίτητη προεπεξεργασία των δεδομένων, ενώ το δεύτερο την εκπαίδευση ενός κατάλληλου *Multi-Layer Perceptron (MLP)* μοντέλου, με στόχο να γίνει *text classification*, αφού προηγουμένως έχει προηγηθεί ο διαχωρισμός των αρχικών δεδομένων σε *train* και *test set*.

1.1 Προεπεξεργασία Δεδομένων

Σε αρκετές εφαρμογές της μηχανικής μάθησης, απαιτείται η κατηγοριοποίηση κειμένων ανάλογα με topic στο οποίο ανήκουν, όπως ακριβώς συμβαίνει και στην δική μας περίπτωση. Προκειμένου κάτι τέτοιο να καταστεί εφικτό, είναι αναγκαία η αναπαράσταση του εκάστοτε κειμένου σε μορφή διανύσματος. Η πιο naive προσέγγιση είναι η αναπαράσταση του εκάστοτε document ως ένα binary vector, με βάση κάποιο λεξικό που θα έχουμε ήδη δημιουργήσει, το οποίο θα απαρτίζεται από τις $top-k$ πιο συχνά εμφανιζόμενες λέξεις. Παρόλο αυτά, εφαρμόζοντας κάτι τέτοιο, δεν φαίνεται να επιτυγχάνονται αρκετά καλές επιδόσεις γενικότερα. Μία πιο sophisticated προσέγγιση, είναι το εκάστοτε κείμενο να αναπαρίσταται ως διάνυσμα του οποίου οι τιμές θα προκύπτουν με βάση ένα αντιπροσωπευτικό score, ανάλογο της σπουδαιότητας της αντίστοιχης λέξης του λεξικού για το συγκεκριμένο document. Για το πλαίσιο της συγκεκριμένης εργασίας, πράγματι, χρησιμοποιούμε την τεχνική *TF-IDF*[2] που θα περιγραφεί αναλυτικά ακολούθως.

Πριν όμως από όλα αυτά, είναι απαραίτητο να γίνουν ορισμένα επιπλέον βήματα προηγουμένως. Ειδικότερα, έπειτα από την φόρτωση του αρχείου *customer_complaints.csv* στο HDFS, αρχικά καθαρίζουμε στοιχειωδώς το dataset, αφαιρώντας γραμμές που είτε δεν ξεκινάνε με "201" (μη έγκυρη ημερομηνία), είτε δεν διαθέτουν βασικό σώμα παραπόνου (τρίτη στήλη του πίνακα κενή). Η αντίστοιχη συνάρτηση *Map* παρουσιάζεται ακολούθως (βλ. *Αλγόριθμο 4*). Έπειτα, θεωρώντας ότι έχουμε κάπως πιο "καθαρό" dataset, διατηρούνται πρόσθετα μόνο χαρακτήρες του αλφαβήτου και το κενό μεταξύ των λέξεων, με την βοήθεια της βιβλιοθήκες *re* της Python σε περιβάλλον *Map-Reduce*, ενώ ακόμα αφαιρούνται όλα τα stopwords της αγγλικής γλώσσας, χρησιμοποιώντας την βιβλιοθήκη *NLTK*. Οι αντίστοιχες συναρτήσεις *Map* παρουσιάζονται εν συνεχεία (βλ. *Αλγόριθμο 4*). Έπειτα από τον καθαρισμό του συνόλου δεδομένων, έχουν διατηρηθεί 463.688 δείγματα από τα 1.739.625 που ήταν αρχικά. Εν συνέχεια, επιπρόσθετα, παρουσιάζεται ακολούθως το διάγραμμα της κατανομής δειγμάτων ανά κλάση (βλ. *Figure 5*), με βάση το οποίο γίνεται σαφές ότι το σύνολο δεδομένων είναι αρκετά imbalanced. Περί αυτού θα πραγματοποιηθεί μία σύντομη συζήτηση στο τέλος της επόμενης υποενότητας.

Επιπλέον, προκειμένου να μην έχουμε προβλήματα με την διαθέσιμη μνήμη που μας δίνεται,

[1] https://catalog.data.gov/dataset/consumer-complaint-database
[2] https://en.wikipedia.org/wiki/Tf%E2%80%93idf

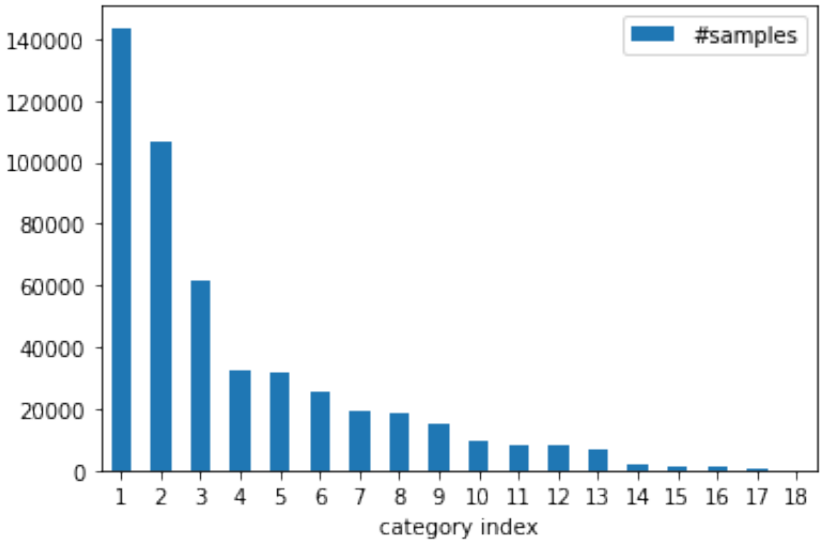

Figure 1: Κατανομή δειγμάτων ανά κλάση (σε φθίνουσα σειρά συχνότητας)

δεδομένου ότι το σύνολο των μοναδικά εμφανιζόμενων λέξεων που συναντώνται αναμένεται να είναι πολύ μεγάλο, στην πράξη μία εύλογη επιλογή είναι η δημιουργία ενός λεξικού με τις $top-k$ πιο συχνά εμφανιζόμενες λέξεις, όπως προαναφέρθηκε. Προς αυτή την κατεύθυνση, εφαρμόζεται μία φάση *Map-Reduce*, όπου το πρώτο *Map*[3], για κάθε λέξη που απαντίζει το σώμα του τρέχοντος παραπόνου, δημιουργείται μια tuple (word, 1), με στόχο στην φάση *Reduce* να εντοπιστεί ο συνολικός αριθμός που εμφανίζεται η συγκεκριμένη λέξη, σε όλα τα κείμενα που εξετάστηκαν[4]. Τέλος, οι λέξεις ταξινομούνται σε φθίνουσα σειρά εμφάνισης, ενώ προφανώς εισάγουμε στο λεξικό τις $top-k$.

Πλέον σε αυτό το σημείο, είμαστε έτοιμοι για τον υπολογισμό του *TF-IDF* score, με χρήση *MapReduce*. Αρχικά να επισημανθεί ότι θεωρώντας κάποια λέξη t, κάποιο κείμενο παραπόνου d και έστω D η συλογή όλων των κειμένων που έχουμε στην διάθεση μας, υπολογίζουμε το *TF-IDF* score ως εξής

$$TF-IDF(t,d) = \underbrace{\left(0.5 + 0.5 \cdot \frac{f_{t,d}}{\max\{f_{t',d} : t' \in d\}}\right)}_{\text{weighted TF term}} \cdot \underbrace{log_2\left(\frac{N}{|d \in D : t \in d|}\right)}_{\text{IDF term}}$$

με $f_{t,d}$ να είναι η συχνότητα εμφάνισης της λέξης t στο κείμενο d και |D| = N. Εν συνεχεία θα ακολουθήσει μία σύντομη περιγραφή της προσέγγισης που ακολουθήσαμε, η οποία στηρίζεται, κατά κύριο λόγο, στην διάλεξη του τελευταίου εργαστηρίου του μαθήματος, ενώ έπειτα θα παρουσιαστούν και οι αντίστοιχοι ψευδοκώδικες που απαιτούνται.

Ειδικότερα, αρχικά για κάθε document που περιέχει τουλάχιστον μία λέξη από το λεξικό, διατηρείται το label, τα indexes των words του λεξικού που εντοπίζονται σε αυτό (word id) καθώς και το doc id[5], με διαδοχικές φάσεις *Map*. Έπειτα ακολουθεί μία φάση *Reduce*, προκειμένου να υπολογιστεί η συχνότητα εμφάνισης της εκάστοτε λέξης σε ένα document,

[3]Για την ακρίβεια είναι *flatMap*.

[4]Όπως ακριβώς το παράδειγμα *wordcount* που είχαμε συζητήσει στο μάθημα.

[5]Ουσιαστικά το document id είναι σε τί θέση βρίσκεται το παράπονο, στον αρχικό πίνακα, ξεκινώντας την αρίθμηση από το μηδέν.

Algorithm 1 Ψευδοκώδικας Map για καθαρισμό αρχικών δεδομένων

```
function MAP(key, values)
    // key: complain/line idx id, values: customers-complaints.csv file's line
    if startsWith(values, "201") == True then
        cols = values.split(",")
        if length(cols) >= 3 and cols[2] != "" then
            emit(cols[1], cols[2].lower())
        end if
    end if
end function

function MAP(key, values)
    // key: label, values: complainText (string)
    emit(key, keepAlphabeticChars(values))
end function

function MAP(key, values)
    // key: label, values: complainText (string)
    emit(key, [word for word in values if word != stopWord])
end function
```

χρησιμοποιώντας ως κλειδί το (word id, label, doc id). Εν συνεχεία, εφαρμόζουμε μία φάση *Map* προκειμένου να τοποθετηθεί ως κλειδί πλέον το word id, σε συνδυασμό με μία φάση *Reduce*, επιτυγχάνεται για κάθε λέξη του λεξικού να διατηρείται μία λίστα από tuples της μορφής (doc id, label, #counts[6]), καθιστώντας δυνατό τον υπολογισμό του #docs, δηλαδή τον αριθμό των διαφορετικών αρχείων που εμπεριέχεται η εκάστοτε λέξη, εφαρμόζοντας μία φάση *Map* (για την ακρίβεια *flatMap*), ώστε να διαθέτουμε elemnts της μορφής ((doc id, label), ([word id, #counts, #docs])). Έπειτα τροποποιώντας ελαφρώς το συγκεκριμένο format, χρησιμοποιώντας μία *Map* φάση, ακολουθούμενη από μία φάση *Reduce*, καταλήγουμε να έχουμε elements που έχουν ως κλειδί το label και ως value μία tuple από list των word ids του αντίστοιχου document που αναπαριστά το συγκεκριμένο element, των #counts των αντίστοιχων λέξεων και τα αντίστοιχα #docs. Κατά αυτόν τον τρόπο, υπολογίζεται το *TF-IDF* score για κάθε λέξη, ενώ εν συνεχεία το τελικό αποτέλεσμα-RDD μετατρέπεται σε DataFrame, με πρώτο column το label και δεύτερο column το αντίστοιχο *SparseVector*. Εν συνεχεία παρουσιάζεται ακολούθως ο αντίστοιχος ψευδοκώδικας (βλ. *Αλγόριθμο 5*).

Τέλος, έχοντας μετατρέψει το προκύπτον RDD σε Spark DataFrame, καθίσταται πλέον δυνατή η εκπαίδευση του *μοντέλου perceptron*, με βάση το οποίο θα γίνει εν τέλει και η κατηγοριοποίηση των κειμένων στην υποενότητα που ακολουθεί.

Εν συνεχεία παρουσιάζονται, από πέντε τυχαία επιλεγμένα κείμενα, το label καθώς και το αντίστοιχο *SparseVector* τους, το οποίο αποτελεί το διάνυσμα χαρακτηριστικών του κάθε δείγματος και χρησιμοποιείται για την συμπαγή αναπαράσταση αυτού στον Ευκλείδειο χώρο $\mathbb{R}^{|V|}$, με $|V|$: το μέγεθος του λεξικού που έχει απαιτηθεί. Στην δική μας περίπτωση έχουμε επιλέξει να κρατάμε τις $top-60$ πιο συχνά εμφανιζόμενες λέξεις εντός του λεξικού, άρα προ-

[6] Το #counts σηματοδοτεί την συχνότητα εμφάνισης της συγκεκριμένης λέξης στο αντίστοιχο document, ενώ το #docs που αναφέρεται ακολούθως συμβολίζει τον αριθμό των διαφορετικών αρχείων που εντοπίζεται η λέξη

Algorithm 2 Ψευδοκώδικας Map/Reduce για υπολογισμό TF-IDF Score

```
function MAP(key, values)
    // key: label, values: list of Voc's words per complainText (string)
    values = takeWordsIdxs(values)
    if length(values) > 1 then
        values = (values, createDocIdx())
        for i = length(values[0]) - 1 do
            // values[0]: list of voc's words ids, values[1]: doc id
            emit((values[0][i], key, values[1]), 1)
        end for
    end if
end function

function REDUCE(key, values)
    // key: (word id, label, doc id), values: [1, 1, ..., 1]
    sum = length(values)
    emit(key, sum)
end function

function MAP(key, values)
    // key: (word id, label, doc id), values: #counts
    emit(key[0], (key[2], key[1], values))
end function

function REDUCE(key, values)
    // key: word id, values: many tuples (doc id, label, #counts)
    list = [ ]
    for t in values do
        list.append(t)
    end for
    emit(key, list)
end function

function MAP(key, values)
    // key: word id, values: list of tuples (doc id, label, #counts)
    for i = 0:length(values) - 1 do
        emit((values[i][0], values[i][1]), (key, values[i][2], length(values)))
    end for
end function

function REDUCE(key, values)
    // key: (doc id, label), values: many tuples (word id, #counts, #docs)
    list = [ ]
    for t in values do
        list.append(t)
    end for
    emit(key, list)
end function
```

```
function MAP(key, values)
    // key: (doc id, label), values: list of (doc id, label, #counts)
    list = [ ]
    values = map(list, zip(*values))
    for i = 0:length(values[0]) - 1 do
        score = calculateScore(values[1][i], max(values[1]), N, values[2][i])
        list.append(score)
    end for
    // for each document ...
    // (label, ([word1, .., wordK], [score1, ..., scoreK])) -> SparseVector()
    emit(key[1], SparseVector(values[0], list))
end function
```

φανώς $|V| = 60$. Με αυτό τον τρόπο, επομένως, έπειτα από την την κατάλληλη διάσπαση των δεδομένων σε *train* και *test set*, καθίσταται δυνατή η πραγματοποίηση *text classification*, όπως θα συζητήσουμε αναλυτικά στην ενότητα που ακολουθεί.

Complain Label	Feature Vector
Credit card or prepaid card	(60, [1, 6, 14, 33], [1.22902, 2.18908, 1.78497, 1.75294])
Credit reporting	(60, [0, 1, 2, 31], [0.2696, 1.47482, 0.55314, 1.56982])
Debt collection	(60, [0, 26, 29], [0.30811, 1.54458, 1.77146])
Mortgage	(60, [2, 4], [0.88503, 1.58626])
Student loan	(60, [18, 25, 40, 41], [1.9716, 2.4035, 2.22348, 2.36338])

1.2 Κατηγοριοποίηση Κειμένων

Αναπαριστώντας τα δεδομένα μας με Spark DataFrame, με ίδιο format όπως ακριβώς παρουσιάζεται στον ανωτέρω πίνακα, το μόνο που υπολείπεται για την κατηγοριοποίηση των κειμένων, είναι η δημιουργία ενός *train* και *test set*. Δεδομένου ότι το dataset είναι αρκετά imbalanced, χρησιμοποιούμε *stratified split* ώστε τα samples ανά κλάση να μοιράζονται στα *train* και *test set*, με ανολογία $70\% - 30\%$. Κατ' αυτόν τον τρόπο, αποφεύγονται ορισμένες παθολογικές περιπτώσεις που θα μπορούσαν να προκύψουν εάν επιλέγαμε απλώς *random split*.

Έπειτα, έχοντας δημιουργήσει το *train* και *test set*, καλούμαστε να εκπαιδεύσουμε ένα *Multi-Layer Perceptron (MLP)*, το οποίο θα εκπαιδευτεί με βάση τα δείγματα του *train set* και θα αξιολογηθεί η επίδοση του με βάση τα δείγματα του *test set*. Ακολούθως παρουσιάζεται η τυπική δομή ενός *MLP* (βλ. *Figure 6*), όπου το συγκεκριμένο διαθέτει 3 συνολικά layers, με 1 hidden layer.

Ύστερα από εκτενή πειραματισμό καταλήξαμε σε ένα *MLP* μοντέλο με 5 layers, με το επίπεδο εισόδου να έχει διάσταση όσο και το μέγεθος του λεξικού, το πρώτο κρυφό επίπεδο διάσταση 50, το δεύτερο κρυφό επίπεδο διάσταση 40, το τρίτο κρυφό επίπεδο διάσταση 30 και τέλος το επίπεδο εξόδου διάσταση όσο και ο αριθμός των διαφορετικών κατηγοριών.

Εν συνεχεία, έπειτα από το *stratified split* που εφαρμόζεται στο αρχικό dataset, προκύπτει το *train* και *test set*, με βάση τα οποία θα πραγματοποιηθεί η εκπαίδευση και η αξιολόγηση της επίδοσης, αντίστοιχα, του υφιστάμενου *μοντέλου perceptron* που επιλέχθηκε. Προς

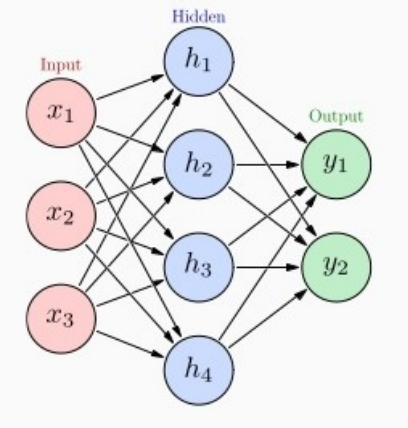

Figure 2: Τυπική αρχιτεκτονική MLP, με 3 layers

αυτή την κατεύθυνση, παρουσιάζονται ακολούθως ορισμένα χρήσιμα statistics, έπειτα από μία ενδυκτική εκτέλεση του αντίστοιχου script (ml.py). Εύκολα θα παρατηρήσουμε ότι η αναλογία δειγμάτων, ανά set, για την πλειοψηφία των κατηγοριών, προσεγγίζει την επιθυμητή αναλογία διαχωρισμού 70%-30%.

Κατηγορία	Train Samples	Αναλογία	Test Samples	Αναλογία
1	99858	0.796	25606	0.204
2	74758	0.73	27650	0.27
3	43133	0.708	17811	0.292
4	22539	0.712	9106	0.288
5	21936	0.743	7571	0.257
6	17506	0.704	7372	0.296
7	13366	0.709	5496	0.291
8	13126	0.706	5453	0.294
9	10494	0.714	4206	0.286
10	6609	0.706	2758	0.294
11	5731	0.704	2413	0.296
12	5543	0.706	2304	0.294
13	4506	0.704	1893	0.296
14	1199	0.697	520	0.303
15	1048	0.705	438	0.295
16	991	0.692	441	0.308
17	190	0.651	102	0.349
18	12	0.8	3	0.2
	342545		111143	

Αξίζει να σημειωθεί ότι κατά την συγκεκριμένη εκτέλεση, το *accuracy* που πέτυχε το μοντέλο μας, στο *test set*, ήταν 57.9%.

Εν συνεχεία, δοκιμάσαμε να τρέξουμε το αντίστοιχο script, χωρίς να χρησιμοποιήσουμε τον μηχανισμό cache του *Spark*. Ακολούθως παρουσιάζεται το αντίστοιχο barchart (βλ. *Figure 7*) με τους χρόνους εκτέλεσης που απαιτήθηκαν για την συνολική διαδικασία. Η διαφορά που παρατηρείται, οφείλεται αποκλειστικά στην χρήση του cache, μιας και όλα τα υπόλοιπα βήματα, καθώς και η ποσότητα των δεδομένων προς επεξεργασία, ήταν παρόμοια με

πριν. Ειδικότερα, χρησιμοποιώντας το *Spark Cache* μηχανισμό, το script ml.py εκτελέστηκε

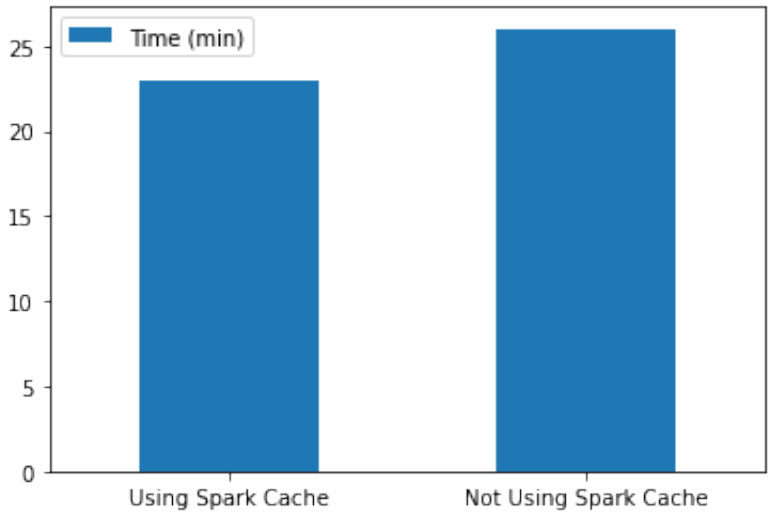

Figure 3: Χρόνος Εκτέλεσης Script, με και χωρίς τον μηχανισμό Cache

σε 23 min περίπου, ενώ στην περίπτωση όπου δεν χρησιμοποιήθηκε απαιτήθηκαν 26 min. Πέρα από το θέμα του χρόνου εκτέλεσης, δεν επηρεάζεται κάτι άλλο και ως αποτέλεσμα το *μοντέλο perceptron* επιτυγχάνει παρόμοιες επιδόσεις, $\sim 57-58\%$ στο *test set* και στις δύο περιπτώσεις.

Παρατηρώντας, λοιπόν, τον αισθητά μικρότερο χρόνο εκτέλεσης της διαδικασίας, στην περίπτωση όπου χρησιμοποιήθηκε ο μηχανισμός *Cache*, αξίζει να σημειωθεί σε αυτό το σημείο, ότι γενικότερα ο συγκεκριμένος μηχανισμός αποτελεί ένα πολύ σημαντικό feature που παρέχει το *Spark*, με βάση το οποίο βελτιώνεται σημαντικά η υφιστάμενη απόδοση, σε περιπτώσεις όπου απαιτείται η επεξεργασία των ίδιων δεδομένων, περισσότερες από μία φορές, βελτιστοποιώντας το I/O. Όταν απαιτείται να πραγματοποιηθεί *caching* σε κάποιο DataFrame, τότε εάν η διαθέσιμη μνήμη το επιτρέπει, γίνεται το *caching* σε ολόκληρο το DataFrame. Αντιθέτως, σε περιπτώσεις όπου η μνήμη δεν επαρκεί, θα γίνει *cashing* μόνο στο κομάτι του DataFrame που καθίσταται κάτι τέτοιο δυνατό, ενώ στο υπόλοιπο μέρος θα γίνει *spill* στον δίσκο. Είναι φανερό επομένως, ότι ο μηχανισμός *Spark Cache* είναι μία αρκετά σημαντική τεχνική για την βελτιστοποίηση των *Spark Jobs* που εκτελούνται, σε περιπτώσεις που απαιτείται επαναχειρισμός ίδιων δεδομένων, με χαρακτηριστικό παράδειγμα DataFrames που αναπαριστούν το *train set* ενός μοντέλου μηχανικής μάθησης, όπως στην δική μας περίπτωση, μιας και έπειτα από την δημιουργία του set, είναι σίγουρο ότι θα ακολουθήσει το training του υφιστάμενου μοντέλου.

Επιπρόσθετα, σε περίπτωση όπου επιθυμούσαμε να επιτευχθούν ακόμα καλύτερα ποσοστά ακρίβειας στο *test set*, λαμβάνοντας υπόψη ότι το δοσμένο dataset είναι αρκετά imbalanced, θα μπορούσαμε κατά την φάση της προεπεξεργασίας των δεδομένων να συνενώσουμε μικρότερες κλάσεις με παραπλήσιες μεγαλύτερες κλάσεις, αλλάζοντας απλώς τα υπάρχοντα labels[7]. Πράγματι, δοκιμάζοντας κάτι τέτοιο παρατηρήσαμε σημαντική αύξηση στο *accuracy*,

[7]Για παράδειγμα, τα δείγματα της κλάσης "Credit card", ενώνονται με εκείνα της κλάσης "Credit card or prepaid card", η οποία είναι μία πιο γενική κλάση, με περισσότερα δείγματα από την αρχική και αποτελεί υπερσύνολο αυτής. Όλες οι συνενώσεις που πραγματοποιούνται κυμαίνονται σε παρόμοιο μήκος κύματος και εκφράζονται με ένα dictionary στο αντίστοιχο script, ml.py.

της τάξης του 7%-10%. Παραδείγματος χάρη, χρησιμοποιώντας τα ίδια *train* και *test set* με προηγουμένως, η ακρίβεια του μοντέλου μας άγγιξε το 65.2%. Ακολούθως παρουσιάζεται η νέα κατανομή των δειγμάτων (βλ. *Figure 8*), έπειτα από τις συνενώσεις κατηγοριών που πραγματοποιήθηκαν. Εάν επιθυμούμε περαιτέρω βελτίωση του *accuracy*, δυνητικά θα μπορούσαμε είτε να μην λάβουμε καθόλου υπόψη μας κλάσεις που δεν διαθέτουν επαρκή αριθμό δειγμάτων, με βάση κάποιο προκαθορισμένο threshold, είτε να εφαρμόσουμε κάποιου είδους sampling[8], ώστε το προκύπτον dataset να είναι σημαντικά λιγότερο imbalanced, ή ακόμα και συνδιασμός αυτών. Γενικά μιλώντας, στην βιβλιογραφία έχει μελετηθεί αναλυτικά το συγκεκριμένο ζήτημα, δεδομένου ότι στα περισσότερα προβλήματα πραγματικού κόσμου, ερχόμαστε αντιμέτωποι με τέτοιου είδους θέματα.

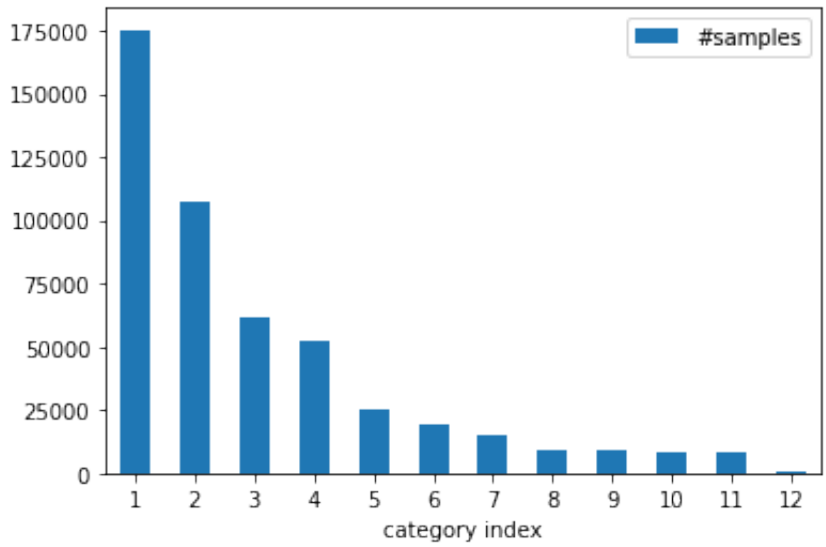

Figure 4: Νέα κατανομή δειγμάτων ανά κλάση (σε φθίνουσα συχνότητα εμφάνισης)

Τέλος, αφού έχουμε δημιουργήσει τα *train* και *test sets* με κατάλληλο τρόπο, καθώς επίσης έχουμε επιλέξει και το κατάλληλο μοντέλο μηχανικής μάθησης, το μόνο που υπολείπεται είναι η επιλογή αξιόπιστων μετρικών, για την ασφαλή αξιολόγηση της επίδοσης που επιτυγχάνεται, καθώς και την εξαγωγή ορθών συμπερασμάτων. Το accuracy αποτελεί ίσως την πιο απλή μετρική, αλλά ενδεικνύεται κυρίως όταν τα δεδομένα που έχουμε στην διάθεση μας, είναι αρκετά κοντά στην τέλεια ισοκατανομή, μεταξύ των διαθέσιμων κλάσεων. Αντιθέτως, σε περιπτώσεις όπου τα δεδομένα μας είναι imbalanced, συνήθως προτιμούνται μετρικές όπως το *Precision*, το *Recall*, το *F1-score*, το *ROC-AUC* κ.α., προκειμένου να δίνεται έμφαση στην επιτυχία κατηγοριοποίησης δειγμάτων ανά κλάση, μιας και το *accuracy* εκφράζει την συνολική εικόνα και επηρεάζεται έντονα από την επιτυχία κατηγοριοποίησης δειγμάτων, τα οποία ανήκουν σε μεγάλες κλάσεις, κάτι πού αναμένεται να συμβεί.

[8] Συνήθως πραγματοποιείται oversampling σε minority κλάσεις, είτε undersampling σε majority κλάσεις, είτε δημιουργία συνθετικών samples (πχ *Αλγόριθμος SMOTE)*.

Αναφορές

[1] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 975–986, 2010.

[2] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman. Mining of Massive Datasets, pages 21-80. [Online:] http://infolab.stanford.edu/~ullman/mmds/book0n.pdf.

[3] Διαφάνειες Διαλέξεων Μαθήματος. [Online:] http://mycourses.ntua.gr/document/document.php?cmd=exChDir&file=%2F%C4%E9%E1%EB%DD%EE%E5%E9%F2.

[4] Υλικό Εργαστηρίου Μαθήματος. [Online:] http://mycourses.ntua.gr/document/document.php?cmd=exChDir&file=%2F%C5%F1%E3%E1%F3%F4%DE%F1%E9%EF.