Εθνικο Μετσοβιο Πολυτεχνειο

Επεξεργασία Φωνής & Φυσικής Γλώσσας

1ο Εργαστηρίο

Γεωργίου Δημήτριος (03115106)

<el15106@central.ntua.gr>

5 Νοεμβρίου 2018

1 Κατασκευή Corpus

Done

2 Προεπεξεργασία Corpus

Done

3 Κατασκευή λεξικού και αλφαβήτου

Done

4 Δημιουργία συμβόλων εισόδου/εξόδου

Done

5 Κατασσκευή μετατροπέων FST

Done

6 Κατασκευή αποδοχέα λεξικού

Done

7 Κατασκευή Ορθογράφου

Done

8 Αξιολόγηση Ορθογράφου

Done

9 Εξαγωγή αναπαραστάσεων word2vec

Done

10 Εξαγωγή στατιστικών

(α)

Ορίσαμε την συνάρτηση `create_words_dictionary()` η οποία δέχεται ως όρισμα τα `word_tokens` όπως αυτά ορίστηκαν στο 3ο Βήμα και κάνει process αυτών. Αρχικοποιούμε ένα λεξικό τύπου float, και για κάθε μοναδική λέξη αυξάνουμε την πιθανότητα εμφάνισης της κατά 1/length της όταν την συναντάμε κατά την διαδικασία προσπέλασης των `word_tokens`.

Σημειώνεται:

(a) Χρησιμοποιήθηκε η συνάρτηση *defaultdict()* για την αρχικοποίηση του float λεξικού μας. Μετά την προσπέλαση `word_tokens`, το λεξικό τελικά θα έχει **κλειδιά** τα `word_tokens` και **τιμές** αυτών τις πιθανότητες εμφάνισης.

(b) Εν τέλει, για τον υπολογισμό των πιθανότητων, πρακτικά διαριούμε την συχνότητα εμφάνισης κάθε λέξης προς τις συνολικές λέξεις που περιέχει το κείμενο (το μήκος της λίστας των `word_tokens`).

(β)

Ορίσαμε την συνάρτηση `create_characters_dictionary()` η οποία δέχεται ως όρισμα τα `alphabet_tokens` και το `corpus_preprocessed`, μετά από κατάλληλη προεπεξεργασία του corpus text που κατεβάσαμε, όπως αυτό ορίστηκαν στο 3ο Βήμα.

Η συνάρτησση κάνει process αυτών, δημιουργώντας την λίστα με τις πιθανότητες εμφάνισης του κάθε γράμματος και τέλος συνδυάζοντας την με τα `alphabet_tokens` για την τελική δημιουργία του λεξικού που θα έχει κλειδιά τα `alphabet_tokens` και τιμές αυτών τις πιθανότητες εμφάνισης.

Σημειώνεται:

(a) Χρησιμοποιήθηκε η συνάρτηση *dict()* που κάνει ακριβώς αυτόν τον συνδυασμό των δύο λιστών σε λεξικό, το οποίο τελικά είναι και η επιστρέφουσα τιμή της `create_characters_dictionary()`.

(b) Επίσης, για τον υπολογισμό των πιθανότητων διαριούμε την συχνότητα εμφάνισης κάθε γράμματος προς το συνολικό μήκος του κειμένου. Εδώ βέβαια είναι αναγκαίο να τονιστεί ότι απο αυτό το συνολικό μήκος αφαιρέσαμε την συχνότητα εμφάνισης της **αλλαγής γραμμής**, καθώς ως σύμβολο δεν αποτελεί μέρος των `alphabet_tokens`, πράγμα που οδηγεί σε μεγαλύτερες πιθανότητες εμφάνισης των υπόλοιπων συμβόλων (Υπολογιστικό Λάθος)

11 Κατασκευή μετατροπέων FST

(α)

Παίρνοντας τον αρνητικό λογάριθμο των πιθανοτήτων των `word_tokens`, όπως αυτές υπολογίστηκαν στο 10ο Βήμα, μπορούμε πολύ εύκολα να πάρουμε το κόστος μετάβασης για κάθε λέξη στο καινούργιο fst. Παίρνοντας τον μέσο όρο αυτών, καταλήξαμε στο κόστος w.

(β)

Υλοποιούμε έναν μετατροπέα όλων των γραμματών του αλφαβήτου, που αφορά τις λέξεις του κειμένου μας, στον ευατό τους με μηδενικό κόστος και ο οποίος θα χρησιμεύσει στην αναγνώριση των χαρακτήρων μιας λέξης **που δεν είναι λάθος**. Το αυτόματο αυτό ουσιαστικά έχει μία κατάσταση που θα είναι και αρχική και τελική/αποδέκτης και καταλήγουμε σε αυτήν όταν έχουμε στην είσοδο έναν λατινικό χαρακτήρα.

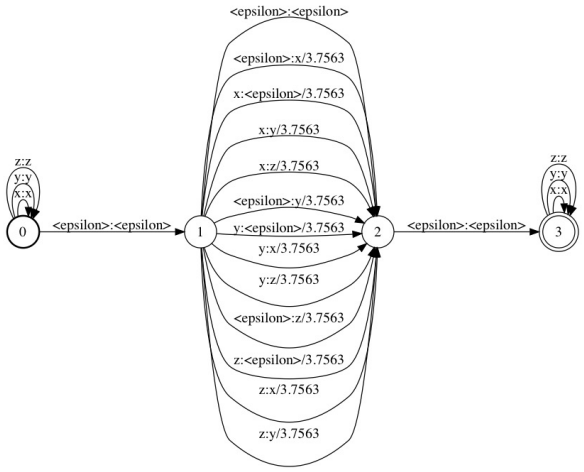
Δημιουργούμε το αυτόματο `"orth_I_words.fst"` εκτελώντας στο shell: `$make orth_I_words`. Δημιουργούμε τον μετατροπέα/αυτόματο `"orth_E_words.fst"` εκτελώντας στο shell: `$make orth_E_words`, το οποίο έχει 1 αρχική κατάσταση και 1 τελική στην οποία μπορούμε να μεταβούμε ως εξής:

- Εισαγωγή χαρακτήρα. `(eps |X)`,
- Διαγραφή χαρακτήρα. `(X |eps)`,
- Αντικατάσταση χαρακτήρα. `(X |Y)`.

Για την δημιουργία του ζητούμενου μετατροπέα/αυτομάτου `"tranducer_words.fst"`, που βασίζεται στην απόσταση Levenshtein, με την υπόθεση ότι ανά μία λέξη υπάρχει μόνο μία αντιστοίχιση (εισαγωγή, διαγραφή ή αντικατάσσταση χαρακτήρα) με βάρος w (όπως αυτό υπολογίστηκε στο α' ερώτημα), χρησιμοποιούμε την εντολή *fstconcat()*. Συγκεκριμένα `tranducer_words = orth_I_words |orth_E_words |orth_I_words`,

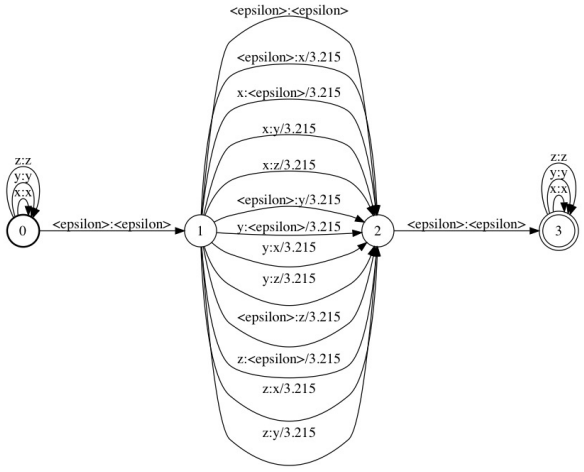
ενώ η κλειστότητα αποφεύχθηκε εφόσον τοποθετήσαμε μετάβασση από την αρχική κατάσταση στην τελική, αποκλειστικά με `(eps |eps)`.

Δημιουργούμε τον μετατροπέα/αυτόματο `"tranducer _words.fst"` εκτελώντας στο shell: `$make tranducer_words`. Γραφικά, αλλά πιο αφαιρετικά για 3 χαρακτήρες έστω X, Y, Z (αντιπροσωπεύουν οποιουσδήποτε 3 χαρακτήρες του λεξικού μας), φαίνεται παρακάτω:



(γ)

Επαναλαμβάνουμε την διαδικασία για εύρεση του βάρους/κόστους μετάβασης κάθε γράμματος και τελικά τον μέσο όρο w αυτών. Έτσι λοιπόν έχουμε με την σειρά δημιουργία του `"orth_I_chars.fst"` και `"orth_E_chars.fst"` εκτελώντας στο shell: `$make orth_I_words`, `$make orth_E_chars` αντίστοιχα και τελική σύνθεση των αυτομάτων σε `"tranducer_chars.fst"` μέσω `$make tranducer_chars`.



(δ)

Έστω ότι έχουμε στην διάθεση μας δύο αρχεία (έστω **"testing.txt"** και **"correct.txt"**) που το κάθενα έχει μία λέξη ανά γραμμή, το 2ο περιέχει όλες τις λέξεις του λεξικού μας, ενώ το πρώτο training data, τις ίδιες λέξεις με λάθη.

Εξετάζουμε μία προς μία τις λέξεις των δύο .txt αρχείων και ελέγχουμε αν γίνεται με διόρθωση ενός μόνο λάθους (με τον τρόπο που περιγράφηκε στο α' ερώτημα) γίνεται να οδηγηθούμε ολοκληρωτικά στην σωστή λέξη. Σε περίπτωση που γίνεται, αυξάνουμε τον counter που αφορά το πλήθος διορθωμένων από ένα λάθος λέξεις.

Έτσι λοιπόν γίνεται εξαγωγή δεδομένων απο training set και δημιουργούμε τις αντίστοιχες πιθανότητες του κάθε λάθους, προφανώς κανονικοποιημένα στο πλήθος των λαθών (1/λέξη), ενώ τέλος παίρνουμε τον αρνητικό αλγόριθμο αυτών για να υπολογίσουμε τα κόστη μετάβασης κάθε λάθος στο νέο fst αυτόματο που δημιουργούμε.

Σημειώνεται: ότι είναι λοιπόν εφικτό να χρησιμοποιήσουμε συγκεκριμένες λέξη-λέξη διορθώσεις για να δημιουργήσουμε ένα extension του error model (γιατί για error model πρόκειται) με τις συχνές συγχύσεις λέξεων. Ένας εναλλακτικός τρόπος θα ήταν να γίνει re-algin των διορθώσεων κάνοντας χρήση του αλγορίθμου Damerau-Levenshtein και να εκπαιδεύσουμε την αρχική απόσταση χαρακτήρων (βάρη) βάσει **συχνότητας διόρθωσης τους**.

12 Κατασκευή Γλωσσσικών Μοντέλων

(α)

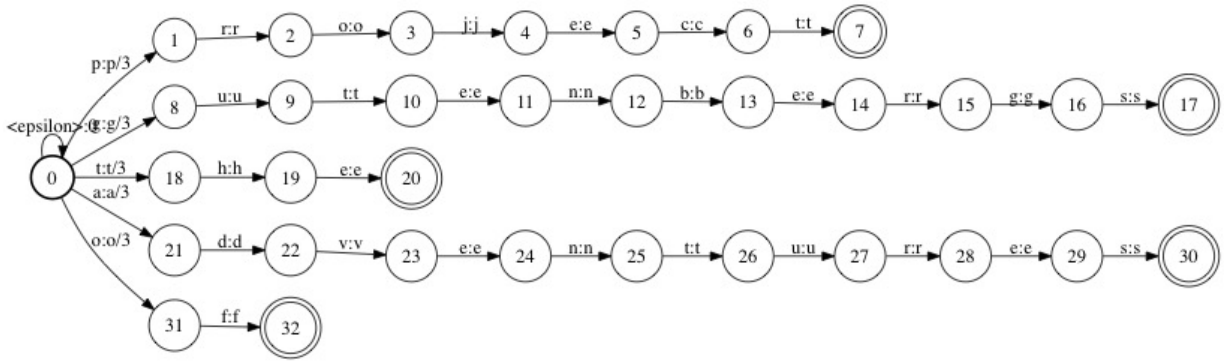
Δημιουργούμε το αρχείο `"orth_acceptor_words.txt"` πάνω στο οποίο θα πατήσουμε για τον τελικό σχηματισμό του `"orth_acceptor_words.fst"`. Συγκεκριμένα: την κάθε λέξη από το `word_tokens` που είχαμε επίσης δημιουργήσει στο 3ο βήμα, την διασπάμε σε γράμματα - letters:

- Αν έχουμε λέξη με 1 γράμμα, τότε πολύ απλά το γράφω στο αρχείο .txt και αρχική και τελική κατάσταση συμπίπτουν.
- Αν έχουμε λέξη με >1 γράμματα, τότε γράφω το πρώτο γράμμα και στην συνέχεια επαναληπτικά και τα υπόλοιπα,.

Μάλιστα, για κάθε λέξη, ανεξαρτήτου πλήθους γραμμάτων, η μετάβαση της στο πρώτο γράμμα αυτής κοστίζει όσο και το `words_dictionary_costs[word]`, ενώ οι υπόλοιπες μεταβάσεις 0.

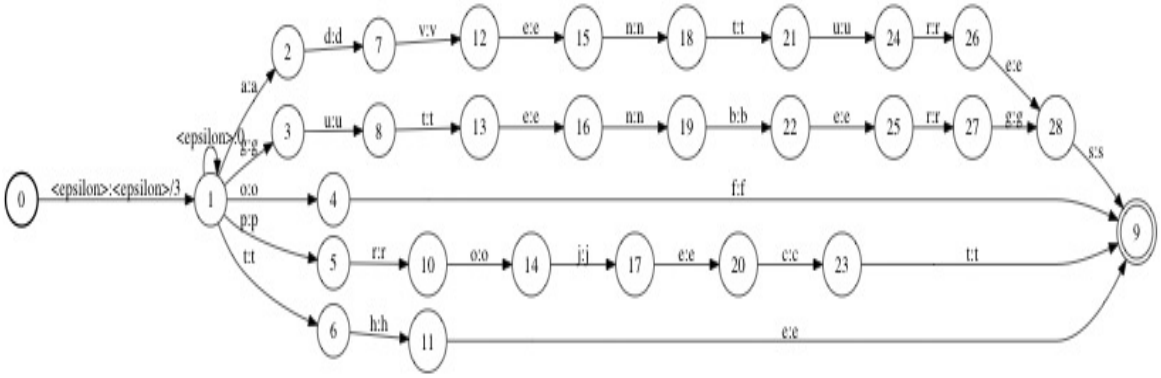
Σημειώνεται και είναι πολυ σημαντικό ότι έχουμε έναν αρχικό counter καταστάσεων `state_counter` ο οποίος αυξάνεται σε κάθε εμφάνιση γράμματος, ώστε να δημιουργηθούν με μεγάλη ακρίβεια τα μονοπάτια των λέξεων και **μόνο** του συνόλου των `word_tokens` μας (προκύπτουν τόσες κατάστασεις όσα και γράμματα). Σε κάθε άλλη περίπτωση πχ. μηδενισμός του counter κατά την εμφάνιση νέας λέξης, θα οδηγούσε σε αυτόματο που θα αποδεχόταν τον κάθε δυνατό συνδυασμό λέξεων που προκύπτουν από τις λέξεις που έχει το λεξικό μας.

Δημιουργούμε τον αποδοχέα/αυτόματο `"orth_acceptor_words.fst"` εκτελώντας στο shell: `$make orth_acceptor_words`. Το παρουσιάζουμε γραφικά παρακάτω, αλλά περιορισμένο σε 5 λέξεις απο τις συνολικές που υπάρχουν. Με την ίδια λογική σχηματίζονται και τα υπόλοιπα μονοπάτια αποδοχής των υπόλοιπων λέξεων.



(β)

Στο σημείο αυτό με τις εντολές *fstdeterminize()*, *fstminimize()*, παίρνουμε το αυτόματο του αποδοχέα που δημιουργήσαμε και το κάνουμε ντετερμινιστικό ενώ ταυτόχρονα το μειώνουμε. Σημειώνεται ότι ο ντετερμινισμός καθυστερεί αρκετά στην συγκεκριμένη έκδοση του OpenFst. Τέλος χρησιμοποιούμε την εντολή *fstrmepsilon()* με σκοπό την εξάλειψη μη απαραίτητων epsilon μεταβάσεων από το αυτόματο μας. Συνδυαστικά μετά την εκτέλεση των εντολών μέσω του shell: `$make orth_acceptor_processed_words`. Γραφικά λοιπόν παρουσιάζεται παρακάτω:

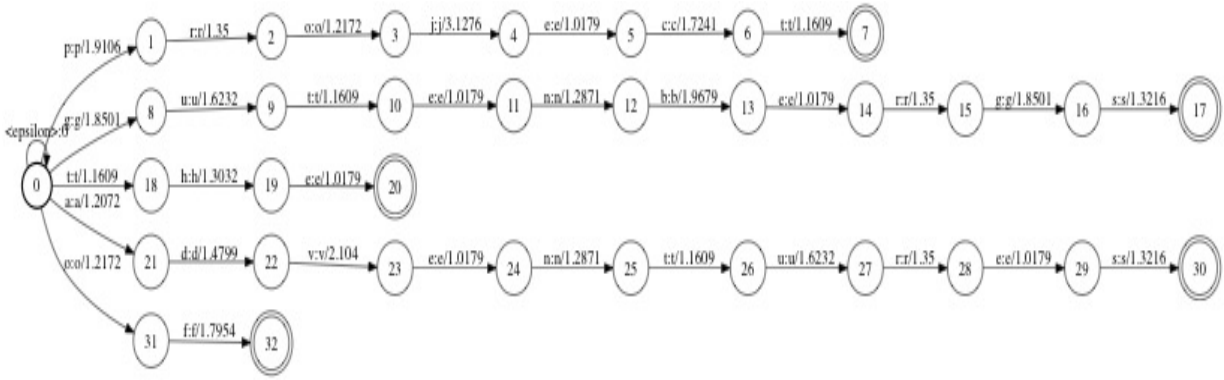


(γ)

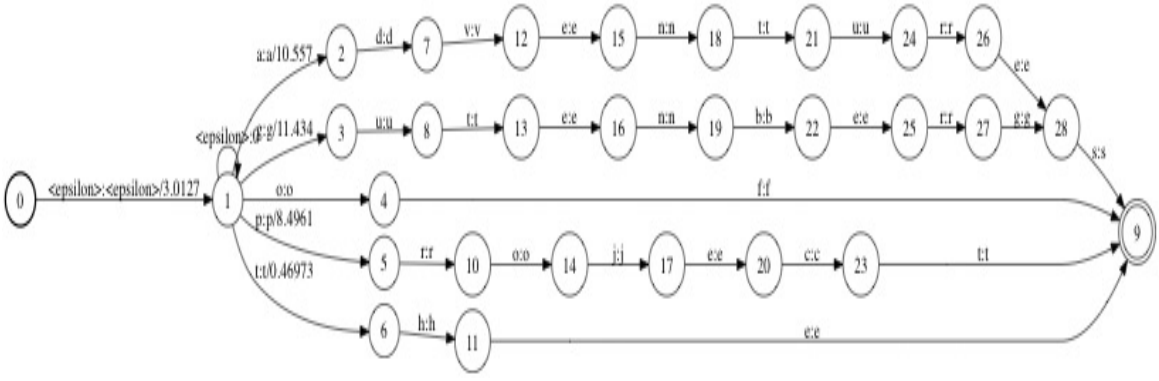
Επαναλαμβάνουμε την διαδικασία για το unigram model ως εξής:

- Για τους σκοπούς αυτού του εργαστηρίου, το language modeling μας χρησιμοποιείται καταλλήλως για την διημιουργία ορθογράφων, επομένως περιορίζεται στην διόρθωση λέξεων. Επομένως, για το unigram model, εύλογα οδηγούμαστε στην ιδέα δημιουργίας ενός αποδέκτη, που αποδέχεται όλες τις λέξεις απο τα `word_tokens`, με την διαφορά ότι στο **στο προηγούμενο ερώτημα ορίσαμε κόστη μετάβασης μη μηδενικά μόνο για την 1η μετάβαση κάθε λέξης, ενώ εδώ ορίζουμε για κάθε δυνατή μετάβαση γράμματος**
- Για την κάθε μετάβαση βάζουμε το κόστος που υποδεικνύει το λεξικό με τα κόστη που δημιουργήσαμε

Δημιουργούμε τον αποδοχέα/αυτόματο `"orth_acceptor_chars.fst"` εκτελώντας στο shell: `$make orth_acceptor_chars`. Το παρουσιάζουμε γραφικά παρακάτω, αλλά περιορισμένο σε 5 λέξεις απο τις συνολικές που υπάρχουν. Με την ίδια λογική σχηματίζονται και τα υπόλοιπα μονοπάτια αποδοχής των υπόλοιπων λέξεων.



Βελτιστοποιούμε το μοντέλος μας την κατάλληλη εκτέλεση των shell εντολών όπως και προηγουμένως `$make orth_acceptor_processed_chars`. Σημειώνεται: ότι μέσω της εκτέλεσης αυτών των εντολών, οδηγηθήκαμε στην δημιουργία αυτομάτου με κόστη μετάβασης μόνο στην 1η μετάβαση κάθε λέξης, (τα οποία υπολογίστηκαν βάσει του κόστους όλων των γραμμάτων κάθε λέξης) όπως και στο word-level model, με διαφορετικά κόστη όμως πλέον για τις ίδιες λέξεις.



13 Κατασκευή Ορθογράφων

(α)

Δημιουργούμε τον ορθγογράφο μας "orthograph_words" μέσω: $make orthograph_words, ο οποίος προκύπτει από την σύνθεση του sorted transducer transducer_words και του orth_acceptor_processed_ words. Ο Levenshtein transducer διορθώνει τις λέξεις χωρίς να λαμβάνει υπόψιν γλωσσικές πληροφοριές, με σκοπό την επίτευξη min edit distance checking.

(β)

Δημιουργούμε τον ορθγογράφο μας "orthograph_chars" μέσω: $make orthograph_chars, ο οποίος προκύπτει από την σύνθεση του sorted transducer transducer_words και του orth_acceptor_processed_ chars. Γενικότερα η απόδοση του μετατροπέα μεταβάλλεται ανάλογα τα weigths που θα τοποθετηθούν στις μεταβάσεις. Εφόσον, εδώ χρησιμοποιούμε παρόμοιο μετατροπέα με το προηγούμενο ερώτημα, η διαφορά στην απόδοση τους οφείλεται στον αποδέκτη που χρησιμοποιούν.

(γ)

Και οι δύο ορθογράφοι "orthograph_words", "orthograph_chars", δοθέντος μίας λέξης αποπειρώνται στην επιστροφή της πιο κοντινής τους, βάσει το πως εκπαιδεύτηκαν προηγουμένως. Γενικά το δεύτερο μας επιτρέπει να κάνουμε καλύτερο capture των εσωτερικών γλωσσικών μηχανισμών. Τονίζουμε ότι για μορφολογικά πλούσιες γλώσσες (πχ. Ρώσσικα), πρέπει να λαμβάνεται υπόψιν και η μορφολογία των κειμένων, το οποίο γίνεται απευθείας στο δεύτερο μοντέλο και έμμεσα στο πρώτο με ενσωμάτωση μοντέλων γλωσσομάθειας.

Εκτελώντας τις κατάλληλες εντολές στο Makefile όπως και προπαρασκευαστικό εργαστήριο για την λέξη "cit", λάβαμε αμφισημία για τους δύο ορθογράφους, η οποία οφείλεται στην ακρίβεια τους, με τον unigram να έχει μεγαλύτερη αφού είναι char-based. Αυτό σημαίνει, ότι

- Η λέξη **"sit"** για τον ορθογράφο του word-level "orthograph_words"
- Η λέξη **"it"** για τον ορθογράφο του word-level "orthograph_chars"

```
Checking the word <wit> with the orthograph_words
3       2       c       s       6.60575819
0
1       0       t       t       1.13769531
2       1       i       i       0.62109375
Checking the word <wit> with the orthograph_chars
3       2       c       <epsilon>       4.63505507
0
1       0       t       t       4.30761719
2       1       i       i       4.30761719
```

14 Αξιολόγηση Ορθογράφων

(α)(β)

Για κάθε λέξη του "spell_checker_test_set.txt" δημιουργούμε έναν αποδοχέα "word_acceptor.fst" με την γνωστή πλεόν διαδικασία όπως στα προηγούμενα βήματα. Εφόσον έχουμε ήδη δημιουργήσει τους ορθογράφους μας, δεν απαιτείται η σύνθεση 3 FST's, του μετατροπέα, του αποδοχέα του λατινικού λεξικού και της λέξης, αλλά αντιθέτως μόνο του αντίστοιχου ορθογράφου με την προς έλεγχο λέξη μας.

Στην συνέχεια ελέγχουμε:

- Αν η λέξη δεν είναι στο κείμενο τότε αν ο αρθογράφος μας βρει μία best λέξη που είναι ίδια με αυτή που περιέχει στο κείμενο τότε την εισάγουμε στις "corrected_words".
- Αλλιώς θα την αντιστοιχίσει σε κάποια λέξη που είναι λάθος οπότε και την εισάγουμε στην κατηγορία των "wrong_words, ενώ αν δεν μπορεί να την αντιστοιχίσει κάπου στις "no_matching_words.

Πλέον είμαστε σε θέση να εξετάσουμε την διαδικασσία για τα μοντέλα μας και να εκτυπώσουμε τα τελικά αποτέλεσμα στο python script μας. Τα αποτελέσματα φαίνονται παρακάτω:

- Για newcorpus = **"The Adventures of Sherlock Holmes"**, το οποίο αποτελείται απο 107797 λέξεις, για τις 140 ορθές λέξεις (έλεγχος 270 λέξεων) έχουμε με χρήση του word-level ορθογράφου :

```
CHECKING WITH ORTHOGRAPH_WORDS GAVE THE FOLLOWING RESULTS

Corrected Words 73
Wrong Words 57
There was no matching for 140 words
```

- Με newcorpus = **"History of Atchison County Kansas"**, το οποίο αποτελείται από 372476 λέξεις, για τις 140 ορθές λέξεις (έλεγχος 270 λέξεων) έχουμε με χρήση του unigram ορθογράφου:

```
CHECKING WITH ORTHOGRAPH_CHARS GAVE THE FOLLOWING RESULTS

Corrected Words 73
Wrong Words 57
There was no matching for 140 words
```

(γ)

15 Extra Credit

(α)

16 Δεδομένα και Προεπεξεργασία

(α)(β)

Για λόγους συμβατότητας χρησιμοποιήθηκαν οι δοθείσες συναρτήσεις για την προεπεξεργασία του corpus (pos και neg κριτικών του φακέλου train). Πιο συγκεκριμένα, παίρνουμε 5000 προπεξεργασμένα δείγματα θετικών και 5000 αρνητικών κριτικών καλώντας την `read_samples()`, τα οποία δίνουμε ως ορίσματα στην συνάρτηση `create_corpus()`, η οποία επιστρέφει μία λίστα με στοιχεία τις κριτικές (string) και μία άλλη - με κατάλληλη συσχέτιση θέσεων με την πρώτη - με στοιχεία 0 (αρνητική κριτική) ή 1 (θετική κριτική).

17 Κατασκευή BOW αναπαραστάσεων και ταξινόμηση

(α)

Το **"BOW"** μοντέλο είναι ένα από τα πιο κατάλληλα μοντέλα για μετατροπή raw κειμένου σε αριθμούς, το οποίο λαμβάνει υπόψιν μόνο τις συχνότητες εμφάνισης των λέξεων στα κείμενα. Σημειώνεται: ότι το μοντέλο κωδικοποιεί ως σημαντικές λέξεις εκείνες που εμφανίζονται πολλές φορές σε όλα τα κείμενα του corpus.

Παρόλα αυτά εάν ο σκοπός μας είναι ο εντοπισμός των "πράγματι" σημαντικών λέξεων στο corpus, υπάρχει ένας πολύ καλύτερος μετασχηματισμός που μπορούμε να χρησιμοποιήσουμε. Το **"TF-IDF"**, το οποίο λαμβάνει υπόψιν τόσο την συχνότητα ενός όρου-λέξης όσο και την αντίστροφη συχνότητα εμφάνισης του στο κείμενο.

Η συχνότητα όρου (term frequency - tf) είναι πρακτικά η έξοδος του **"BOW"** μοντέλου. Για ένα συγκεκριμένο corpus αποφασίζει πόσο σημαντική είναι μία λέξη, ελέγχοντας πόσο συχνά εμφανίζεται σε αυτό, άρα μετράει την τοπική της σημαντικότητα.

Η αντίστροφη-κειμένου συχνότητα (inverse document frequency (idf)), τονίζει ότι μια λέξη κάποιου αρχείου του corpus θεωρείται σημαντική, **ΔΕΝ** πρεπει να εμφανίζεται συχνά σε άλλα αρχεία του ίδιου corpus. Άρα η συχνότητα κειμένου πρέπει να μικρή, ώστε η αντίστροφη-κειμένου να είναι μεγάλη. Η συνολική συχνότητα υπολογίζεται ως εξής: $\text{idf(w)} = \log \frac{\#\text{reviews}}{\#\text{reviews containing word w}}$

Άρα μια λέξη έχει υψηλη idf εάν εμφανίζεται πολλές φορές σε ένα κείμενο και είναι απούσα στα υπόλοιπα.

(β)(γ)

1. Αρχικοποιούμε την μεταβλητή `"count_vect"` ως αντικείμενο τύπου *CountVectorize*,
2. Μετατρέπουμε το train corpus σε vector μέσω της συναρτήσης `fit_transform()` ,
3. Δημιουργούμε έναν ταξινομητή Logistic Regression τον οποίο εκπαιδεύουμε με δεδομένα το vector του 2ου βήματος και την λίστα indices που δείχνει για το καθένα σε ποια κατηγορία ανήκει (pos ή neg), ενώ ταυτόχρονα εκτυπώνουμε το training error που προκύπτει (στην συγκεκριμένη περίπτωση πολύ μικρό)
4. Επαναλαμβάνουμε το βήμα 16 για testing δεδομένα που βρίσκονται στον φάκελο test, και τώρα έχουμε ένα vector test corpus με τα αντίστοιχα indices
5. Χρησιμοποιούμε την συνάρτηση *predict()*, για να προβλέψουμε, βάσει του πλέον εκπαιδευμένου μοντέλου μας, την κατηγορία (pos ή neg) που θα ανήκουν οι κριτικές του vector test, ενώ τέλος βρίσκουμε το error της πρόβλεψης αυτής συγκρίνοντας το αποτέλεσμα με τα test indices.

Συγκεντρωτικά για Bag of Words αναπαρασστάσεις προκύπτει:

```
BOW's Training error =  0.0003
BOW's Accuracy =  0.8567
```

(δ)

Επαναλαμβάνουμε την ίδια ακριβώς διαδικασία, με την μόνη διαφορά ότι τώρα εξετάζουμε την απόδοση ενός TfidfVectorizer, που χρησιμοποιεί σταθμισμένα αθροίσματα one hot wors encodings για την αναπαράσταση των προτάσεων με βάρη TF-IDF.

Συγκεντρωτικά για Bag of Words αναπαρασστάσεις προκύπτει:

```
TfidfTransformer's Training error =  0.0538
TfidfTransformer's Accuracy =  0.8719
```

Παρατηρήσεις:

- Όπως περιμέναμε, η απόδοση του μοντέλου με χρήση βαρών TF-IDF αυξάνεται κατά 1.77 %
- Από την άλλη, το training error αυξάνεται κατά 179 %. Γενικά είναι πολύ δύσκολο να γίνει fit του vector σε όλα τα train reviews. Όσο αυξάνεται το μέγεθος αυτών, είναι προτιμητέο η χρήση τέτοιων βαρών να αποφεύγεται,και μάλιστα η χρήση n-grams θα κάνει τα δεδομένα απλά να έχουν μεγαλύτερες διαστάσσεις.

18 Χρήση Word2Vec αναπαραστάσεων για ταξινόμηση

(α)

Για το μοντέλο Word2Vec που είχε εκπαιδευτεί με τις προτάσεις του βιβλίου corpus = **"The Adventures of Sherlock Holmes"**, παρατηρήσαμε ότι το ποσοστό των OOV είναι αρκετά υψηλό. Για αυτές τις λέξεις, το embedding τους υπολογίζεται αποκλειστικά βάσει τοπικών συμφραζόμενων.

```
The percentage of OOV is:  43.356470004957856  %
```

(β)

1. Δημιουργούμε την συνάρτηση *ws2meanvec()* η οποία δέχεται ως όρισμα μία πρόταση και το αντίστοιχο εκπαιδευμένο μοντέλο και επιστρέφει των μέσο όρο των w2v κάθε λέξης που περιέχει (Neural Bag of Words)
2. Αυτή γίνεται apply για όλες τις προτάσεις τόσο του train όσο και του test corpus, δημιουργώντας τα κατάλληλα vectors
3. Δημιουργούμε έναν ταξινομητή Logistic Regression τον οποίο εκπαιδεύουμε με δεδομένα το vector του 2ου βήματος και την λίστα indices που δείχνει για το καθένα σε ποια κατηγορία ανήκει (pos ή neg)
4. Χρησιμοποιούμε την συνάρτηση *predict()*, για να προβλέψουμε, βάσει του πλέον εκπαιδευμένου μοντέλου μας, την κατηγορία (pos ή neg) που θα ανήκουν οι κριτικές του vector test, ενώ τέλος βρίσκουμε το error της πρόβλεψης αυτής συγκρίνοντας το αποτέλεσμα με τα test indices.

Συγκεντρωτικά για Word2Vec αναπαρασστάσεις προκύπτει:

```
Word2Vec's Accuracy =  0.6084
```

Η ακρίβεια του μοντέλου είναι σαφώς πολύ μικρή. Αρχικά το μοντέλο μας είναι εκπαιδευμένο πάνω σε ένα λογοτεχνικό corpus, ενώ εμείς επιζητούμε πρόβλεψη για κριτικές πάνω σε ταινίες, οπότε δεν υπάρχει τόσο έντονη συσχέτιση μεταξύ τους.

(γ)(δ)

Επειδή τα βήματα της προπαρασκευής εκτελούνται σε άλλο .py script, χρησιμοποιούμε απευθείας τις 10 λέξεις που εξετάστηκαν στο βήμα 9γ, για την εύρεση των σημασιολογικά πιο κοντινών τους βάσει τώρα το google model. Τα αποτελέσματα φαίνονται παρακάτω:

```
how [('what', 0.6820360422134399), ('How', 0.6297600269317627), ('why', 0.5838741064071655)]
surprised [('shocked', 0.8090525269508362), ('flabbergasted', 0.7832474708557129), ('taken_aback', 0.781646549701690
7)]
smokes [('smokes_cigarettes', 0.69474858045578), ('cigarettes', 0.6854457259178162), ('smoked', 0.6450352668762207)]
savagely [('viciously', 0.7358489036560059), ('brutally', 0.7070984244346619), ('mercilessly', 0.6742576956748962)]
she [('her', 0.7834683656692505), ('She', 0.7553189396858215), ('herself', 0.669890820980072)]
stooped [('stooping', 0.6364398002624512), ('stoop', 0.6238372921943665), ('stoops', 0.5228399038314819)]
penknife [('pocketknife', 0.6871023178100586), ('knife', 0.6770156621932983), ('knives', 0.6004471778869629)]
lay [('laying', 0.771097719669342), ('laid', 0.7509710192680359), ('Laying', 0.6501108407974243)]
detail [('details', 0.6135907173156738), ('detailed', 0.5740132331848145), ('specifics', 0.5441097021102905)]
fierce [('ferocious', 0.7291355133056641), ('intense', 0.6843384504318237), ('fiercest', 0.6470801830291748)]
```

Τολμούμε να σχολιάσουμε ότι η απόδοση του μοντέλου είναι εξαιρετική. Τα αποτελέμσατα είναι ποιοτικά όπως περιμέναμε, συμπεραίνοντας πως τα προεκπαιδευμένα GoogleNews vectors κάνουν capture το context όλων των προς εξέταση λέξεων με τέτοιο τρόπο που η αλγεβρική τους λειτουργία έχει σημαντικά νοήματα.

(ε)

Η ακρίβεια του μοντέλου είναι επίσης πολύ μικρή, και μάλιστα ελάχιστα μικρότερη από τα word vectors που χρησιμοποιήθηκαν στο α' ερώτημα, από custom εκπαιδευμένο Word2Vec model,και **ΔΕΝ** να ξεχνάμε ότι η διάσταση των Google News vectors είναι πολύ πολύ μεγαλύτερη. Στο σημείο αυτό αξίζει να αναφερθεί πως αν χρησιμοποιηθεί κάποιο άλλο custom μοντέλο, εκπαιδευμένο πάνω σε κριτικές ταινιών και όχι σε λογοτεχνικά κείμενο, η απόδοση του αντίστοιχου ταξινομητή θα αυξανόταν αρκετά.

```
Goodle Word2Vec's Accuracy =  0.6041
```

(ζ)(η)

Επαναλαμβάνουμε ακριβώς την ίδια διαδικασία με προηγουμένως, με την μόνη διαφορά ότι εδώ τροποποιούμε την συνάρτηση *ws2meanvec()* ώστε να δέχεται ως όρισμα μία πρόταση και το αντίστοιχο εκπαιδευμένο μοντέλο και να επιστρέφει των μέσο όρο των w2v κάθε λέξης που περιέχει (Neural Bag of Words) λαμβάνοντας υπόψιν και τα βάρη TF-IDF. Παρατηρούμε ότι η απόδοση του μοντέλου μειώθηκε αρκετά.

Τα αποτελέσματα φαίνονται παρακάτω:

```
Goodle Word2Vec TF-IDF's Accuracy =  0.5
```

19 ΠΑΡΑΡΤΗΜΑ

Παραπάνω αναλύθηκαν όλα τα βήματα που για την διαδικασία δημιουργίας του ορθογράφου και ταξινομητή συναισθήματος. Έχουμε δημιουργήσει 2 python script:

- `"SLP_lab3.ipynb"` για τον ορθογράφο. Για διευκόλυνση της διαδικασίας αυτής και για λόγους αυτοματοποίησης, έχουμε δημιουργήσει ένα Makefile που περιλαμβάνει όλες τις απαραίτητες εντολές που απαιτείται να δοθούν στο shell σχετικά με την βιβλιοθήκη openfst. Βέβαια αν εκτελεστεί το python script `"SLP_lab2.ipynb"`, αυτό από μόνο του τρέχει τα απαραίτητα system calls για τα αντίστοιχα makes σε κάθε βήμα.
- `"SLP_lab3.ipynb"` για τον ταξινομητή συναισθήματος που δεν περιλαμβάνει καθόλου εντολές συστήματος.

A Appendix

SLP_lab2

November 20, 2018

In [1]:
```
import os, os.path
import nltk
from shutil import copyfile

#SSL Certificate has fauled
#that not in the system certificate store.
from nltk.corpus.reader.plaintext import PlaintextCorpusReader
#PlaintextCorpusReader will use the default nltk.tokenize.sent_tokenize()
#and nltk.tokenize.word_tokenize() to split your texts into sentences and words

from urllib import request
```

In [2]:
```
#----------------------------------------------STEP 1---------------------------------
#Text number 1661 is "The Adventures of Sherlock Holmes" by Arthur Conan Doyle, and we
url = "http://www.gutenberg.org/cache/epub/1661/pg1661.txt"
response = request.urlopen(url)
corpus = response.read().decode('utf8')
corpus = corpus.replace('\r', '')
length_corpus = len(corpus)
```

In [3]:
```
# Make new dir for the corpus.
corpusdir = 'newcorpus.nosync/'
if not os.path.isdir(corpusdir):
    os.mkdir(corpusdir)

copyfile("Makefile", corpusdir + "Makefile")
copyfile("spell_checker_test_set.txt",corpusdir + "spell_checker_test_set.txt")
```

Out[3]:
```
'newcorpus.nosync/spell_checker_test_set.txt'
```

In [4]:
```
# Output the files into the directory.
filename = 'SherlockHolmes.txt'
with open(corpusdir+filename, 'w') as f:
    print(corpus, file=f)
```

In [5]:
```
#Check that our corpus do exist and the files are correct.
# Key Note:
# 1.We split each file into words and we their equality until the penultimate word, si
#in the created file
assert open(corpusdir+filename,'r').read().split(' ')[:-1] == corpus.split(' ')[:-1]
```

In [6]:
```
# Create a new corpus by specifying the parameters
# (1) directory of the new corpus
# (2) the fileids of the corpus
# NOTE: in this case the fileids are simply the filenames.
# Now the text has been parsed into paragraphs, sentences and words by the default act
# of the PlaintextCorpusReader
newcorpus = PlaintextCorpusReader(corpusdir, '.*')
os.chdir(corpusdir)
#-----------------------------------------------END OF STEP 1-------------------------
```

In [7]:
```
#-----------------------------------------------STEP 2--------------------------------
#----------------------(a)----------------------#
#Function used as default argument in parser() function if it is not defined
def identity_preprocess(s):
    if(isinstance(s, str)):
        return s
    else: return "No string was given"
```

In [8]:
```
#----------------------(b)----------------------#
#Function to parse the text file given, line by line
def parser(path,preprocess = identity_preprocess):
    tokens = []
    for line in path.split('\n'):
        tokens+= preprocess(line)
    return tokens
```

In [9]:
```
#----------------------(c)----------------------#
import re
import string
#Tokenization step, a simple version which includes tokens of lowercase words
def tokenize(s):
    s_temp = s.strip().lower()
    s_temp = re.sub('[^A-Za-z\n\s]+', '', s_temp)
    s_temp = s_temp.replace('\n', ' ')
    s_temp = " ".join(s_temp.split())
    s_temp = s_temp.split(' ')
    s_temp[:] = [item for item in s_temp if item != '']
    return s_temp
#-----------------------------------------------END OF STEP 2-------------------------
```

In [10]:
```
#-----------------------------------------------STEP 3--------------------------------
#Constructing word tokens and alphabet of the new corpus
#----------------------(a)----------------------#
corpus_preprocessed = newcorpus.raw(newcorpus.fileids()[1])
word_tokens = parser(corpus_preprocessed, tokenize)
```

In [11]:
```
#----------------------(b)----------------------#
def tokenize_2(s):
    s_temp = s.strip()
```

```
        s_temp = " ".join(s_temp.split())
        s_temp = s_temp.split(' ')
        return s_temp
```

In [12]:
```
def parser_2(path, preprocess):
    alphabet = []
    for line in path.split('\n'):
        line = preprocess(line)
        for word in line:
            alphabet+= list(word)

    alphabet.append(' ')
    return set(alphabet)

alphabet_tokens = sorted(parser_2(corpus_preprocessed,tokenize_2))
#-----------------------------------------------END OF STEP 3-------------------------
```

In [13]:
```
#-----------------------------------------------STEP 4-------------------------------
filename = 'chars.syms'
filename =  open(filename, 'w')
result = []

filename.write('<epsilon>'+ " " + str(0)+'\n')
filename.write('<space>'+ "   " + str(1)+'\n')
for symbol in range(2,len(alphabet_tokens)):
    line = alphabet_tokens[symbol] + "         " + str(symbol)+'\n'
    filename.write(line)

filename.close()
#-----------------------------------------------END OF STEP 4-------------------------
```

In [14]:
```
#-----------------------------------------------STEP 10------------------------------
from collections import defaultdict
def create_words_dictionary(word_tokens):
    length = len(word_tokens)
    wordfreq = defaultdict(float)
    for i in range(len(word_tokens)):
        wordfreq[word_tokens[i]] += 1/length
    return wordfreq

words_dictionary = create_words_dictionary(word_tokens)
#for k, v in words_dictionary.items():
#    print(k, v)
```

In [15]:
```
def create_characters_dictionary(alphabet_tokens, corpus_preprocessed):
    result = {}
    length = len(corpus_preprocessed) - corpus_preprocessed.count('\n')
    charfreq = [corpus_preprocessed.count(symbol)/length for symbol in alphabet_tokens
    return dict(zip(alphabet_tokens,charfreq))
```

```
characters_dictionary = create_characters_dictionary(alphabet_tokens, corpus_preproces
#for k, v in characters_dictionary.items():
#    print(k, v)
#-----------------------------------------------END OF STEP 10-----------------------
```

In [16]:
```
#-----------------------------------------------STEP 11-------------------------------
#Calculating the costs of transition for each word as cost_w_i = -log(p(w_i))
#and after that the mean value
#----------------------(a)---------------------#
import math
import statistics
words_dictionary_costs = dict(zip(list(set(word_tokens)),[-math.log10(value) for key,

costs = [words_dictionary_costs[key] for key in words_dictionary_costs]
w = statistics.mean(costs)
```

In [17]:
```
#HERE WE CREATE THE TRANDUCER I
#for the word_tokens
#----------------------(b)---------------------#
filename = 'orth_I_words.txt'
filename = open(filename,'w')

alphabet="abcdefghijklmnopqrstuvwxyz"

for letter in alphabet:
    filename.write("0 0 "+ letter +" "+ letter +" 0\n")
filename.write("0")

filename.close()
!make -s orth_I_words
```

In [18]:
```
#HERE WE CREATE THE TRANDUCER E
filename = 'orth_E_words.txt'
filename = open(filename,'w')
filename.write('0 1 <epsilon> <epsilon> 0'+'\n')
for i in range(len(alphabet)):
    filename.write('0 1 <epsilon> '+alphabet[i]+' '+str(w)+'\n')#insertion
    filename.write('0 1 ' + alphabet[i]+' <epsilon> '+str(w)+'\n')#deletion
    for j in range(len(alphabet)):
        if alphabet[i]!=alphabet[j]:
            filename.write('0 1 ' + alphabet[i]+' '+alphabet[j]+' '+str(w)+'\n')#Repl

filename.write(str(1))
filename.close()
!make -s orth_E_words
!make -s transducer_words
!make -s transducershortest_words
#FINALLY WE CREATE THE TRANDUCER transducer = orth_I | orth_E | orth_I with the Makef
```

In [ ]:
```
#----------------------(c)---------------------#
#Calculating the costs of transition for each char as cost_c_i = -log(p(c_i))
#and after that the mean value
characters_dictionary_costs = dict(zip(list(set(alphabet_tokens)),[-math.log10(value) f
costs = [characters_dictionary_costs[key] for key in characters_dictionary_costs]
w = statistics.mean(costs)
print(w)
```

```
3.215014865006331
```

In [ ]:
```
#HERE WE CREATE THE TRANDUCER I
#for the char_tokens
filename = 'orth_I_chars.txt'
filename = open(filename,'w')

alphabet="abcdefghijklmnopqrstuvwxyz"

for letter in alphabet:
    filename.write("0 0 "+ letter +" "+ letter +" 0\n")
filename.write("0")

filename.close()
!make -s orth_I_chars
```

In [ ]:
```
#HERE WE CREATE THE TRANDUCER E
filename = 'orth_E_chars.txt'
filename = open(filename,'w')
filename.write('0 1 <epsilon> <epsilon> 0'+'\n')
for i in range(len(alphabet)):
    filename.write('0 1 <epsilon> '+alphabet[i]+' '+str(w)+'\n')#insertion
    filename.write('0 1 ' + alphabet[i]+' <epsilon> '+str(w)+'\n')#deletion
    for j in range(len(alphabet)):
        if alphabet[i]!=alphabet[j]:
            filename.write('0 1 ' + alphabet[i]+' '+alphabet[j]+' '+str(w)+'\n')#Repla

filename.write(str(1))
filename.close()
!make -s orth_E_chars
!make -s transducer_chars
#FINALLY WE CREATE THE TRANDUCER transducer = orth_I | orth_E | orth_I with the Makefi
#-----------------------------------------------END OF STEP 11-------------------------
```

In [ ]:
```
#-----------------------------------------------STEP 12--------------------------------
#HERE WE CREATE THE ACCEPTOR/AUTOMATO used to accept all the words of our words_tokens
#One state for each letter of every word-> States will be limited later when we will a
#commands of determinization, minimization, removal of <epsilon> transitions to our or
#----------------------(a)---------------------#
filename = 'orth_acceptor_words.txt'
```

```
acceptor=open(filename, 'w')
final_states = []
state_count = 0

acceptor.write('0 0 <epsilon> 0\n')

for word in list(set(word_tokens)):
    chars = list(word)
    if(len(chars) == 1):
        arg = ['0',' ',str(state_count+1),' ',chars[0],' ',chars[0],' ', str(words_dic
        arg = ''.join(arg)
        acceptor.write(arg)
        state_count += len(chars)
        final_states.append(str(state_count))
    else:
        arg = ['0',' ',str(state_count+1),' ',chars[0],' ',chars[0],' ',str(words_dict
        arg = ''.join(arg)
        acceptor.write(arg)
        for j in range(1,len(chars)):
            arg = [str(j + state_count),' ',str(j+1 + state_count),' ',chars[j],' ',cha
            arg = ''.join(arg)
            acceptor.write(arg)
        state_count += len(chars)
        final_states.append(str(state_count))
for i in range(0,len(final_states)):
    arg = [final_states[i],'\n']
    arg = ''.join(arg)
    acceptor.write(arg)
acceptor.close()
!make -s orth_acceptor_words
!make -s orth_acceptor_processed_words
```

In [ ]:
```
#HERE WE CREATE THE ACCEPTOR/AUTOMATO used to accept all the words of our char_tokens,
#One state for each letter -> States will be limited later when we will apply the resp
#commands of determinization, minimization, removal of <epsilon> transitions to our or
#----------------------(b)--------------------#
filename = 'orth_acceptor_chars.txt'
acceptor=open(filename, 'w')
final_states = []
state_count = 0

acceptor.write('0 0 <epsilon> 0\n')
for word in list(set(word_tokens)):
    chars = list(word)
    if(len(chars) == 1):
        arg = ['0',' ',str(state_count+1),' ',chars[0],' ',chars[0],' ', str(character
        arg = ''.join(arg)
        acceptor.write(arg)
```

```
                state_count += len(chars)
                final_states.append(str(state_count))
            else:
                arg = ['0',' ',str(state_count+1),' ',chars[0],' ',chars[0],' ',str(characters
                arg = ''.join(arg)
                acceptor.write(arg)
                for j in range(1,len(chars)):
                    arg = [str(j + state_count),' ',str(j+1 + state_count),' ',chars[j],' ',cha
                    arg = ''.join(arg)
                    acceptor.write(arg)
                state_count += len(chars)
                final_states.append(str(state_count))
        for i in range(0,len(final_states)):
            arg = [final_states[i],'\n']
            arg = ''.join(arg)
            acceptor.write(arg)
        acceptor.close()
        !make -s orth_acceptor_chars
        !make -s orth_acceptor_processed_chars

        #-----------------------------------------------END OF STEP 12-------------------------
```

In [ ]:
```
        #-----------------------------------------------STEP 13--------------------------------
        #----------------------(a)---------------------#
        !make -s orthograph_words

        #----------------------(b)---------------------#
        !make -s orthograph_chars

        #----------------------(c)---------------------#
        filename = 'cit.txt'
        filename = open(filename, 'w')
        word = "cit"
        state = 0
        for letter in word:
            if letter!='\n':
                filename.write(str(state)+' '+str(state+1)+' '+letter+ '\n')
                state+=1
        filename.write(str(state)+'\n')

        filename.close()
        print("Checking the word <cit> with the orthograph_words")
        !make -s check_cit_words
        print("Checking the word <cit> with the orthograph_chars")
        !make -s check_cit_chars

        #-----------------------------------------------END OF STEP 13-------------------------
```

```
Checking the word <cit> with the orthograph_words
```

SLP_lab3

November 20, 2018

In [1]:
```
#----------------------------------------------STEP 16---------------------------------
import os, os.path
import nltk
from shutil import copyfile
from nltk.corpus.reader.plaintext import PlaintextCorpusReader
from urllib import request
```

In [2]:
```
data_dir = 'aclImdb/'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')
pos_train_dir = os.path.join(train_dir, 'pos')
neg_train_dir = os.path.join(train_dir, 'neg')
pos_test_dir = os.path.join(test_dir, 'pos')
neg_test_dir = os.path.join(test_dir, 'neg')
```

In [3]:
```
# For memory limitations. These parameters fit in 8GB of RAM.
# If you have 16G of RAM you can experiment with the full dataset / W2V
MAX_NUM_SAMPLES = 5000
# Load first 1M word embeddings. This works because GoogleNews are roughly
# sorted from most frequent to least frequent.
# It may yield much worse results for other embeddings corpora
NUM_W2V_TO_LOAD = 1000000
```

In [4]:
```
import numpy as np
SEED = 42
# Fix numpy random seed for reproducibility
np.random.seed(42)
try:
    import glob2 as glob
except ImportError:
    import glob
```

In [5]:
```
import re

def strip_punctuation(s):
    return re.sub(r'[^a-zA-Z\s]', ' ', s)

def preprocess(s):
```

```
        return re.sub('\s+',' ', strip_punctuation(s).lower())

    def tokenize(s):
        return s.split(' ')

    def preproc_tok(s):
        return tokenize(preprocess(s))
```

In [6]:
```
def read_samples(folder, preprocess=lambda x: x):
    samples = glob.iglob(os.path.join(folder, '*.txt'))
    data = []
    for i, sample in enumerate(samples):
        if MAX_NUM_SAMPLES > 0 and i == MAX_NUM_SAMPLES:
            break
        with open(sample, 'r') as fd:
            x = [preprocess(l) for l in fd][0]
            data.append(x)
    return data
```

In [7]:
```
def create_corpus(pos, neg):
    corpus = np.array(pos + neg)
    y = np.array([1 for _ in pos] + [0 for _ in neg])
    indices = np.arange(y.shape[0])
    np.random.shuffle(indices)
    return list(corpus[indices]), list(y[indices])
```

In [8]:
```
#Calling the prementioned funtions
pos_train = read_samples(pos_train_dir,preprocess)
neg_train = read_samples(neg_train_dir,preprocess)
corpus_train, indices_train = create_corpus(pos_train,neg_train)
#------------------------------------------------END OF STEP 16-------------------------
```

In [9]:
```
#------------------------------------------------STEP 17--------------------------------
#----------------------(b)---------------------#
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import zero_one_loss, accuracy_score

count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(corpus_train)
#----------------------(c)---------------------#

#Training the logistic regression model
clf = LogisticRegression().fit(X_train_counts, indices_train)
print("BOW's Training error = ", zero_one_loss(indices_train,clf.predict(X_train_counts

#Predicting for the test reviews if they are positive or negative based on the previou
pos_test = read_samples(pos_test_dir,preprocess)
neg_test = read_samples(neg_test_dir,preprocess)
```

```
        corpus_test, indices_test = create_corpus(pos_test,neg_test)

        X_test_counts = count_vect.transform(corpus_test)
        count_predictions = clf.predict(X_test_counts)
        print("BOW's Accuracy = ", accuracy_score(indices_test,count_predictions))
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/sklearn/linear_m
  FutureWarning)


BOW's Training error =  0.0003
BOW's Accuracy =  0.8567
```

```
In [10]: #----------------------(d)---------------------#
         from sklearn.feature_extraction.text import TfidfVectorizer

         tfidf_vect = TfidfVectorizer()
         X_train_tfidf = tfidf_vect.fit_transform(corpus_train)

         #Training the logistic regression model
         clf = LogisticRegression().fit(X_train_tfidf, indices_train)
         print("TfidfTransformer's Training error = ", zero_one_loss(indices_train,clf.predict

         X_test_tfidf = tfidf_vect.transform(corpus_test)
         tfidf_predictions = clf.predict(X_test_tfidf)

         print("TfidfTransformer's Accuracy = ", accuracy_score(indices_test,tfidf_predictions

         #-----------------------------------------------END OF STEP 17-----------------------
```

```
TfidfTransformer's Training error =  0.0538
TfidfTransformer's Accuracy =  0.8719
```

```
In [24]: #-----------------------------------------------STEP 18-------------------------------
         #----------------------(a)---------------------#
         # Initialize word2vec. Context is taken as the 2 previous and 2 next words
         from gensim import models
         import numpy as np
         import random

         def identity_preprocess(s):
             if(isinstance(s, str)):
                 return s
             else: return "No string was given"

         def parser(path,preprocess = identity_preprocess):
             tokens = []
```

```
            for line in path.split('\n'):
                s_temp = preprocess(line)
                if(s_temp ==[]):continue
                tokens.append(s_temp)
            return tokens

        def tokenize(s):
            s_temp = s.strip().lower()
            s_temp = re.sub('[^A-Za-z\n\s]+', '', s_temp)
            s_temp = s_temp.replace('\n', ' ')
            s_temp = " ".join(s_temp.split())
            s_temp = s_temp.split(' ')
            s_temp[:] = [item for item in s_temp if item != '']
            return s_temp

        url = "http://www.gutenberg.org/cache/epub/1661/pg1661.txt"
        response = request.urlopen(url)
        corpus = response.read().decode('utf8')
        corpus = corpus.replace('\r', '')

        sent_tokens = parser(corpus,tokenize)

        model = models.Word2Vec( sent_tokens,window=5, size=100, min_count = 2,workers=4)
        model.train(sent_tokens, total_examples=len(sent_tokens), epochs=1000)

        # get ordered vocabulary list|
        voc = model.wv.index2word
        # get vector size| dim = model.vector_size
```

```
4570
```

In [29]:
```
word_tokens = preproc_tok(corpus)
length = len(list(set(word_tokens)))
print("The percentage of OOV is: " ,(length - len(voc))*100/length," %")
```

```
The percentage of OOV is:  43.356470004957856  %
```

In [79]:
```
#----------------------(b)---------------------#
# syn0 is a numpy array that stores a feature vector for each word.
from functools import partial

# First method - averaging words vectors.
def ws2meanvec(words, embedding_model):
    vectors = [embedding_model[w] for w in words if w in embedding_model]
    if len(vectors) > 0:
        return np.mean(vectors, axis=0)
```

```
            else:
                # To avoid empty words causing error. For example, raw sentence is 10/10 at i
                return np.zeros(embedding_model.vector_size, dtype=np.float32)


        X_train_tl = list(map(partial(ws2meanvec, embedding_model=model), corpus_train))
        X_test_tl = list(map(partial(ws2meanvec, embedding_model=model), corpus_test))


        log_reg = LogisticRegression(solver='lbfgs')
        log_reg.fit(X_train_tl, indices_train)
        pred_test = log_reg.predict(X_test_tl)
        print("Word2Vec's Accuracy = ", accuracy_score(indices_test, pred_test))
```

```
1
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel_launch
  import sys
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel_launch
  import sys
```

```
Word2Vec's Accuracy =  0.6084
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/sklearn/linear_m
  "of iterations.", ConvergenceWarning)
```

```
In [83]: #----------------------(c)(d)---------------------#
         from gensim.models import KeyedVectors

         # Load google pre-trained model.
         google_model = KeyedVectors.load_word2vec_format('./GoogleNews-vectors-negative300.bi

         X_train_tl = list(map(partial(ws2meanvec, embedding_model=google_model), corpus_train
         X_test_tl = list(map(partial(ws2meanvec, embedding_model=google_model), corpus_test))


         words_to_check = ['how','surprised','smokes','savagely','she','stooped','penknife','l
         for word in words_to_check:
             sim = google_model.wv.most_similar(word,topn=3)
             print(word,sim)
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel_launch
  del sys.path[0]
```

```
how [('what', 0.6820360422134399), ('How', 0.6297600269317627), ('why', 0.5838741064071655)]
surprised [('shocked', 0.8090525269508362), ('flabbergasted', 0.7832474708557129), ('taken_aba
smokes [('smokes_cigarettes', 0.69474858045578), ('cigarettes', 0.6854457259178162), ('smoked'
savagely [('viciously', 0.7358489036560059), ('brutally', 0.7070984244346619), ('mercilessly',
she [('her', 0.7834683656692505), ('She', 0.7553189396858215), ('herself', 0.669890820980072)]
stooped [('stooping', 0.6364398002624512), ('stoop', 0.6238372921943665), ('stoops', 0.5228399
penknife [('pocketknife', 0.6871023178100586), ('knife', 0.6770156621932983), ('knives', 0.600
lay [('laying', 0.771097719669342), ('laid', 0.7509710192680359), ('Laying', 0.650110840797424
detail [('details', 0.6135907173156738), ('detailed', 0.5740132331848145), ('specifics', 0.544
fierce [('ferocious', 0.7291355133056641), ('intense', 0.6843384504318237), ('fiercest', 0.647
```

In [84]:
```
#----------------------(e)---------------------#

log_reg = LogisticRegression(solver='lbfgs')
log_reg.fit(X_train_tl, indices_train)
pred_test = log_reg.predict(X_test_tl)
print("Goodle Word2Vec's Accuracy = ", accuracy_score(indices_test, pred_test))
```

```
Goodle Word2Vec's Accuracy =  0.6041
```

In [99]:
```
#----------------------(f)(z)---------------------#
tfidf = dict(zip(tfidf_vect.get_feature_names(), tfidf_vect.idf_))

# Second method - averaging weighted words vectors.
def ws2meanvec(words, embedding_model, weights):
    vectors = []
    for w in words:
        if(w in embedding_model and w in tfidf):
            vectors.append(embedding_model[w]*weights[w])

    if len(vectors) > 0:
        return np.mean(vectors, axis=0)
    else:
        # To avoid empty words causing error. For example, raw sentence is 10/10 at i
        return np.zeros(embedding_model.vector_size, dtype=np.float64)


X_train_tl = list(map(partial(ws2meanvec, embedding_model=google_model, weights = tfi
X_test_tl = list(map(partial(ws2meanvec, embedding_model=google_model, weights = tfid


log_reg = LogisticRegression(solver='lbfgs')
log_reg.fit(X_train_tl, indices_train)
pred_test = log_reg.predict(X_test_tl)
print("Goodle Word2Vec TF-IDF's Accuracy = ", accuracy_score(indices_test, pred_test)
```

```
Goodle Word2Vec TF-IDF's Accuracy =  0.5
```

In [105]:

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-105-b6128436ce8d> in <module>
     19
     20 log_reg = LogisticRegression(solver='lbfgs')
---> 21 log_reg.fit(X_train_tl, indices_train)
     22 pred_test = log_reg.predict(X_test_tl)
     23 print("Goodle Word2Vec TF-IDF's Accuracy = ", accuracy_score(indices_test, pred_tes


/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/sklearn/l
   1282
   1283         X, y = check_X_y(X, y, accept_sparse='csr', dtype=_dtype, order="C",
-> 1284                          accept_large_sparse=solver != 'liblinear')
   1285         check_classification_targets(y)
   1286         self.classes_ = np.unique(y)


/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/sklearn/u
    745                     ensure_min_features=ensure_min_features,
    746                     warn_on_dtype=warn_on_dtype,
--> 747                     estimator=estimator)
    748     if multi_output:
    749         y = check_array(y, 'csr', force_all_finite=True, ensure_2d=False,


/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/sklearn/u
    563         if not allow_nd and array.ndim >= 3:
    564             raise ValueError("Found array with dim %d. %s expected <= 2."
--> 565                              % (array.ndim, estimator_name))
    566         if force_all_finite:
    567             _assert_all_finite(array,


ValueError: Found array with dim 3. Estimator expected <= 2.
```

In [ ]: