Επεξεργασία Φυσικής Γλώσσας – 2022/20223 - Ο.Π.Α

Εργασία 1

Φοίβος-Αστέριος Νταντάμης (f3312204) (phivos93@yahoo.com)

Παναγιώτης Καραστατήρης () (pan.karastatiris@gmail.com)

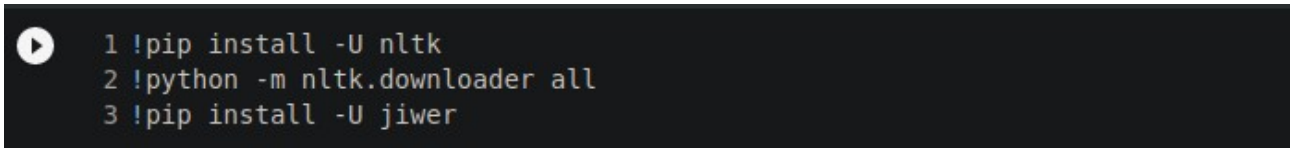
Άσκηση 3

notebook url:

https://colab.research.google.com/drive/1dJz__C8IfRWPAEeDXsG4vARvRclcwibR?usp=sharing

Ερώτημα i) Φοίβος Νταντάμης

Ζητείται να υλοποιηθεί ένα γλωσσικό μοντέλο διγραμμάτων και τριγραμμάτων. Σαν πρώτο βήμα εγκαθιστώνται οι απαραίτητες βιβλιοθήκες, δηλαδή το nltk που επιλέχθηκε αντί του spaCy, όπως και το jiwer, το οποίο θα χρησιμοποιηθεί για τον υπολογισμό των WER και CER.



Το σύνολο κειμένων που επιλέχθηκε για να εκπαιδευτούν τα μοντέλα είναι το ‘brown’, το οποίο ήταν το πρώτο ηλεκτρονικό σύνολο με 10^6 λέξεις και δημιουργήθηκε το 1961 στο πανεπιστήμιο του Brown. Πέρα από αυτό, απαραίτητο είναι και το ‘punkt’ που περιέχει σημεία στίξης και βοηθάει στην αναγνώριση προτάσεων. Από το σύνολο των κειμένων, εφόσον δεν μας ενδιαφέρει τόσο η επίδοση των μοντέλων, αλλά πρόκειται για άσκηση, κρατήθηκαν μόνο τα πρώτα 20, για λόγους απόδοσης. Η μεταβλητή all_texts_words περιέχει μία λίστα με όλες τις λέξεις/tokens και των 20 κειμένων, όπως αυτές προκύπτουν από τη μέθοδος words του nltk.

```
 1 import math, nltk
 2
 3 nltk.download('brown')
 4 nltk.download('punkt')
 5
 6
 7 number_of_texts_to_keep = 20 ### SET THE NUMBER OF FILES YOU WANT TO KEEP ###
 8
 9 from nltk.corpus import brown
10
11 files = nltk.corpus.brown.fileids()
12 files = files[:number_of_texts_to_keep]
13 print(files)
14 all_texts_words = brown.words(files)
15 print(all_texts_words)
```

```
['ca01', 'ca02', 'ca03', 'ca04', 'ca05', 'ca06', 'ca07', 'ca08', 'ca09', 'ca10',
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
```

Έχουμε λοιπόν κρατήσει 20 κείμενα τα οποία περιέχουν 45530 λέξεις/tokens.

```
1 print('We have', len(files), 'text files,')
2 print('which contain', len(brown.words(files)), 'words')
```

```
We have 20 text files,
which contain 45530 words
```

Εν συνεχεία θα θέλαμε να έχουμε μία λίστα με όλες τις προτάσεις των κειμένων. Για το σκοπό αυτό θα χρησιμοποιηθεί η μέθοδος sent_tokenize. Ωστόσο η μέθοδος αυτή λειτουργεί με κείμενα σε μορφή string, οπότε αρχικά φτιάχνουμε ένα μεγάλο string συνενώνοντας τις λέξεις/tokens και της το περνάμε.

```
 1 from nltk import sent_tokenize
 2 from nltk import word_tokenize
 3
 4 all_texts_string = " ".join(all_texts_words)
 5 sentences = sent_tokenize(all_texts_string)
 6
 7 for sent in sentences:
 8     sent_tok = word_tokenize(sent)
 9
10     print(sent_tok)
11     print("__________________")
```

```
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', '
__________________
['The', 'jury', 'further', 'said', 'in', 'term-end', 'presentmen
__________________
```

Για το πρώτο βήμα ουσιαστικό βήμα, χρησιμοποιείται η συνάρτησης FreqDist, η οποία δέχεται ένα μία λίστα από λέξεις/tokens και μετράει τις συχνότητες εμφάνισής όλων των διαφορετικών λέξεων/tokens που εμπεριέχονται στη λίστα. Στη συνέχεια όπως ζητείται από την εκφώνηση, όλα τα tokens τα οποία εμφανίζονται λιγότερο από 10 φορές, πρέπει να αντικατασταθούν με ένα ειδικό token, το '*UNK*'. Στο σημείο αυτό θα λάβουμε υπόψιν αυτήν την οδηγία μόνο στα πλαίσια τις καταμέτρησης των λέξεων/tokens. Επίσης, επειδή παρατηρήθηκε ότι υπήρχαν διάφορα σημεία στίξης και σύμβολα που δεν βοηθούν στην εκπαίδευση των μοντέλων, αυτά φιλτράρονται και δεν εμπεριέχονται στους υπολογισμούς. Οι χαρακτήρες/ακολουθίες χαρακτήρων αυτοί φαίνονται παρακάτω στη γραμμή 9.

```
 3 N = 20
 4 # Frequency distribution
 5 count = nltk.FreqDist(all_texts_words)
 6
 7 unks = 0
 8 count_unk = count.copy()
 9 garbage_toks = ['.', ',', '--', '(', ')', '$', '```', '``', '-', '?', ':', "''", ';', "''"]
10
11 for item in count.items():
12     if item[0] in garbage_toks:
13         del count_unk[item[0]]
14     elif item[1] < 10:
15         count_unk.update({'*UNK*': item[1]})
16         del count_unk[item[0]]
17         unks += 1
18
19 vocab_size = len(count_unk.items())
20
21 print(f'Found {unks} unknown words\n')
22
23 print(f'We had {count.N()} tokens with a vocabulary of size {len(count.items())}\n')
24
25 print(f'After setting rare tokens to *UNK* we have a vocabulary of size {vocab_size}\n')
26
27 print('Most common tokens: \n')
28 pprint(count_unk.most_common(N))
```

```
Found 7792 unknown words

We had 45530 tokens with a vocabulary of size 8363

After setting rare tokens to *UNK* we have a vocabulary of size 561
```

Το λεξικό μας έχει λοιπόν μέγεθος 8363 λέξεων. Φτιάχνουμε τώρα μία λίστα με προτάσεις με λέξεις/tokens, όπου αντικαθιστάμε τις άγνωστες λέξεις με '*UNK*' και αφαιρούμε τους χαρακτήρες που δεν χρειαζόμαστε, αφού όπως αναφέρθηκε και παραπάνω αυτή η δουλειά δεν έχει γίνει ακόμα παρά μόνο λήφθηκε υπόψιν στα πλαίσια των συχνοτήτων εμφάνισης των λέξεων. Έχοντας λοιπόν αυτήν τη λίστα στη μεταβλητή sentences_tokenized, δημιουργούμε 3 Counters όπου θα κρατήσουμε τις συχνότητες εμφάνισης όλων των μονογραμμάτων, διγραμμάτων και τριγραμμάτων.

```
 8 for sent in sentences:
 9     sent_tok = word_tokenize(sent)
10     ## find *UNK* and garbage chars
11     temp_toks = list(map(lambda tok: tok if tok in vocab and not (tok in garbage_toks) else ('*UNK*' if not tok in garbage_toks else None), sent_tok))
12     res_toks = list(filter(lambda item: item is not None, temp_toks))
13     sentences_tokenized.append(res_toks)
14
15     print(res_toks)
16     print("_________________")
```

```
unigram_counter = Counter()
bigram_counter = Counter()
trigram_counter = Counter()

for sent in sentences_tokenized:

    unigram_counter.update([gram for gram in ngrams(sent, 1, pad_left=True, pad_right=True,
                                                    left_pad_symbol='*start*',right_pad_symbol='*end') ])
    bigram_counter.update([gram for gram in ngrams(sent, 2, pad_left=True, pad_right=True,
                                                    left_pad_symbol='*start*',right_pad_symbol='*end*') ])
    trigram_counter.update([gram for gram in ngrams(sent, 3, pad_left=True, pad_right=True,
                                                    left_pad_symbol='*start*',right_pad_symbol='*end*') ])
```

Έχοντας αυτές τις συχνότητες εμφάνισης μπορούμε πολύ απλά να υπολογίσουμε τις πιθανότητες των γλωσσικών μοντέλων μας, όπου χρησιμοποιείται και η μέθοδος του Add-α smoothing για α=0.1

```
alpha = 0.1

bigram_probs = dict()
bigram_log_probs = dict()

trigram_probs = dict()
trigram_log_probs = dict()

for bigram in bigram_counter:
    bigram_probs[bigram] = (bigram_counter[bigram] + alpha) / (unigram_counter[(bigram[0],)] + alpha*vocab_size)
    bigram_log_probs[bigram] = math.log2(bigram_probs[bigram])
    # print(bigram)
    # print("bigram_prob: {0:.5f} of bigram ".format(bigram_probs[bigram]))

for trigram in trigram_counter:
    trigram_probs[trigram] = (trigram_counter[trigram] + alpha) / (bigram_counter[(trigram[0], trigram[1])] + alpha*vocab_size)
    trigram_log_probs[trigram] = math.log2(trigram_probs[trigram])
    # print(trigram)
    # print("trigram_prob: {0:.5f} of trigram ".format(trigram_probs[trigram]))
```

Ερώτημα ii) Φοίβος Νταντάμης

Υπολογίζουμε τις εντροπίες των δύο μοντέλων μας. Προσθέτουμε σε κάθε πρόταση τα tokens ‘*start*’ και ‘*end*’. Ωστόσο για τον υπολογισμό όπως φαίνεται και στη γραμμή 7, δεν λαμβάνουμε υπόψιν μας το ‘*start*’ όπως και ζητείται. Παρακάτω μετά τον κώδικα φαίνονται και τα αποτελέσματα πρώτα για το γλωσσικό μοντέλο διαγραμμάτων και μετά για το τριγραμμάτων.

```
sum_prob = 0
bigram_cnt = 0

for sent in sentences_tokenized:
    sent = ['*start*'] + sent + ['*end*']
    bigram_cnt += 1 # end tokens
    for idx in range(1, len(sent)-1):
        sum_prob += math.log2(bigram_probs[(sent[idx], sent[idx+1])])
        bigram_cnt += 1

HC = -sum_prob / bigram_cnt
perpl = math.pow(2,HC)
print("Cross Entropy: {0:.3f}".format(HC))
print("perplexity: {0:.3f}".format(perpl))
```

```
Cross Entropy: 4.396
perplexity: 21.049
```

```
sum_prob = 0
trigram_cnt = 0

for sent in sentences_tokenized:
    sent = ['*start*'] + ['*start*'] + sent + ['*end*'] + ['*end*']
    trigram_cnt += 2 # end tokens
    for idx in range(2, len(sent)-2):
        sum_prob += math.log2(trigram_probs[(sent[idx], sent[idx+1], sent[idx+2])])
        trigram_cnt += 1

HC = -sum_prob / trigram_cnt
perpl = math.pow(2,HC)
print("Cross Entropy: {0:.3f}".format(HC))
print("perplexity: {0:.3f}".format(perpl))
```

```
Cross Entropy: 4.247
perplexity: 18.991
```

Ερώτημα iii) Φοίβος Νταντάμης

Για το ερώτημα αυτό ζητείται η υλοποίηση του beam search decoder. Αυτό γίνεται μέσω της συνάρτησης beam_search_decoder, που δέχεται δύο ορίσματα, ένα κείμενο και την υπερπαράμετρο l1(λ_1) από την οποία προκύπτει και η l2 = 1 – l1 (λ_2).

```
def beam_search_decoder(text, l1):
    k = len(text)
    l2 = 1 - l1
    LK = []
```

Μέσα στη συνάρτηση ορίζεται μία εμφωλευμένη συνάρτηση Lk, η οποία υπολογίζει το L_k που είναι και η παράσταση που θέλουμε να ελαχιστοποιήσουμε στο μονοπάτι που θα επιλέξουμε. Η Lk, δέχεται τις δύο υπερπαραμέτρους l1, l2, τα tokens t1, t2 που είναι ένα δίγραμμα που αποτελείται

από το token το οποίο έχουμε μέχρι εκείνο το βήμα αποδεχθεί στο μονοπάτι που εξετάζουμε και το επόμενο πιθανό token που περιέχεται στο λεξικό μας και τα tokens w1, w1 που είναι το δίγραμμα που εμφανίστηκε στο κείμενο που τροφοδοτούμε στον αλγόριθμο. Αν το δίγραμμα t1, t2 δεν έχει παρατηρηθεί στο κείμενο μέχρι εκείνη τη στιγμή υπολογίζουμε την πιθανότητα του και την προσθέτουμε στις υπολογισμένες πιθανότητες ώστε να μην χρειαστεί να ξαναγίνει αυτή η δουλειά αν το ξανασυναντήσουμε. Για τον υπολογισμό της πιθανότητας $P(w^2|t^2)$ ως άθροισμα των αντίστροφων των αποστάσεων Levenstein προσθέτουμε μία μονάδα στον παρονομαστή ώστε να αποφύγουμε τη διαίρεση με το 0 σε περίπτωση που τα tokens είναι ίδια σε κάποιο σημείο.

```
def Lk(l1, l2, t1, t2, w1, w2):
  if (t1, t2) in bigram_log_probs.keys():
      Pt1 = bigram_log_probs[(t1, t2)]
  else:
      bigram_probs[(t1, t2)] = alpha / (unigram_counter[(t1,)] + alpha*vocab_size)
      bigram_log_probs[(t1, t2)] = math.log2(bigram_probs[(t1, t2)])
      Pt1 = bigram_log_probs[(t1, t2)]

  # the 1 in the denominators is to cope with identic tokens
  Pw1t1 = math.log2(1/(nltk.edit_distance(t1, w1) + 1)) + math.log2(1/(nltk.edit_distance(t2, w2) + 1))
  return -l1 * Pt1 -l2 * Pw1t1
```

Έχουμε υλοποιήσει το beam search για b = 2, οπότε στη μεταβλητή paths κρατάμε σε κάθε βήμα τα δύο μονοπάτια που έχουν επιλεχθεί μαζί με αποτέλεσμα της παράστασης L_k μέχρι εκείνο το σημείο. Τα δύο μονοπάτια αρχικά ξεκινάνε με το '*start*' και έχουν κόστος 0. Στην μεταβλητή prev_word κρατάμε κάθε φορά ποιο είναι το προηγούμενο γράμμα στο κείμενο, από αυτό που εξετάζουμε γιατί χρειάζεται να περνιέται στη συνάρτηση Lk στη θέση του w1. Για κάθε λέξη στο κείμενο λοιπόν, παραλείποντας το αρχικό '*start*' δημιουργούμε ένα αντίγραφο του λεξιλογίου το οποίο και ταξινομούμε ανάλογα με την Levenstein απόσταση από τη λέξη αυτή και κρατάμε τις 10 πιο κοντινές.

```
paths = [[0, '*start*'], [0, '*start*']]
prev_word = '*start*'
for word in text[1:]:
  sorted_by_dist = vocab.copy()
  sorted_by_dist.sort(key=lambda w: nltk.edit_distance(word, w))
  sorted_by_dist = sorted_by_dist[:10]
```

Θεωρούμε προσωρινά ότι τα δύο καλύτερα επόμενα βήματα είναι που προκύπτουν επιλέγοντας την πρώτη λέξη των των κοντινών για το πρώτο μονοπάτι και τη δεύτερη για το δεύτερο. Το κόστος προκύπτει απλώς προσθέτοντας νέα L_k και δεν χρειάζεται ο εκ νέου υπολογισμός σε όλο το μονοπάτι αφού ουσιαστικά απλώς προστίθενται όροι στο νέο άθροισμα.

```
paths = [[0, '*start*'], [0, '*start*']]
prev_word = '*start*'
for word in text[1:]:
  sorted_by_dist = vocab.copy()
  sorted_by_dist.sort(key=lambda w: nltk.edit_distance(word, w))
  sorted_by_dist = sorted_by_dist[:10]
  best1 = (paths[0][0] + Lk(l1, l2, paths[0][1], sorted_by_dist[0], word, prev_word), sorted_by_dist[0])
  best2 = (paths[1][0] + Lk(l1, l2, paths[1][1], sorted_by_dist[1], word, prev_word), sorted_by_dist[1])
  best = (best1, best2)
```

Ορίζονται τώρα δύο μεταβλητές from0 και from1, οι οποίες κρατάνε εκφράζουν αν τα νέα best που υπολογίζοντα προέρχονται από το πρώτο αποδεκτό μονοπάτι ή το δεύτερο. From0=0 σημαίνει δηλαδή ότι το best1 είναι ένα μονοπάτι το οποίο επιλέχθηκε και επεκτείνει το μονοπάτι που υπήρχε στο index 0 της λίστα paths. Οπότε τώρα, για κάθε αποδεκτό μονοπάτι εξετάζουμε το κόστος L_k

που διαμορφώνεται για κάθε υποψήφιο επόμενο token και κρατάμε τα δύο καλύτερα που θα προκύψουν στην τούπλα best = (best1, best2).

```
from0 = 0 # this will track on which path we are
from1 = 1
i = -1    # number of path
for path in paths:
  i += 1
  for candidate in sorted_by_dist:
    cand_score = path[0] + Lk(l1, l2, path[1], candidate, word, prev_word)
    if best[0][1] != candidate: # this is in order to really keep the two b
        if cand_score < best[0][0]:
            temp = best[0]
            best = ((cand_score, candidate), best[1])
            from0 = i
            if temp[0] < best[1][0]:
              best = (best[0], temp)
              from1 = i
        elif cand_score < best[1][0]:
            best = (best[0], (cand_score, candidate))
            from1 = i
prev_word = word
```

Ανάλογα με το ποιο μονοπάτι επεκτάθηκε και επιλέχθηκε πρέπει να ενημερώσουμε τη λίστα με τα paths, είτε απλώς προσθέτοντας στο υπάρχων μονοπάτι το νέο token, είτε αντιγράφοντας το άλλο μονοπάτι και έπειτα προσθέτοντας πάλι το νέο token όπως και να ενημερώσουμε το κόστος που αναλογεί στο νέο μονοπάτι. Η λίστα paths όποτε θα περιέχει δύο λίστες της μορφής [κόστος, $token_n$, $token_{n-1}$, …, $token_0$] αφού προσθέτουμε το νέο token πάντα στο index 1.

```
prev_word = word

if from0 == 0 and from1 == 1:
  paths[0][0] = best[0][0]
  paths[0].insert(1, best[0][1])
  paths[1][0] = best[1][0]
  paths[1].insert(1, best[1][1])
elif from0 == 0 and from1 == 0:
  paths[0][0] = best[0][0]
  paths[0].insert(1, best[0][1])
  paths[1] = paths[0].copy()
  paths[1][0] = best[1][0]
  paths[1][1] = best[1][1]
elif from0 == 1 and from1 == 1:
  paths[0] = paths[1].copy()
  paths[0][0] = best[0][0]
  paths[0].insert(1, best[0][1])
  paths[1][0] = best[1][0]
  paths[1].insert(1, best[1][1])
elif from0 == 1 and from1 == 0:
  paths[1][0] = best[1][0]
  paths[1].insert(1, best[1][1])
  paths[0][0] = best[0][0]
  paths[0].insert(1, best[0][1])
```

Τελικά στο τέλος επιστρέφουμε το καλύτερο από τα δύο paths που προκύψανε, αφού πρώτα το αντιστρέψουμε για να είναι τα tokens στη σωστή σειρά.

```
    if paths[0][0] < paths[1][0]:
      result = paths[0][1:]
    else:
      result = paths[1][1:]
    result.reverse()
    return result


res = beam_search_decoder(['*start*', 'I', 'am', 'really', 'sorry', 'for', 'you'], 0.5)
```

Σημείωση για όλα τα προηγούμενα. Η επιλογή των υπερπαραμέτρων a και λ1, λ2 δεν έγινε με κάποιον τυπικό τρόπο άλλα μόνο με δοκιμές.

Ερώτημα iv) Παναγιώτης Καραστατήρης

Για τη δημιουργία του τεχνητού dataset χρησιμοποιούμε την αρχική λίστα προτάσεων που αναπαριστώνται με λίστες από tokens. Για κάθε token, αντικαθιστούμε με τη σειρά τους χαρακτήρες που αποτελείται με κάποιο μικρό γράμμα με πιθανότητα 20%

```
import random
import string

sentences_wrong = []

for sent in sentences_tokenized:
  s= []
  for idx in range(0, len(sent)-1):
    s.append(''.join(letter if (random.random() >  0.2 ) else random.choice(string.ascii_lowercase) for letter in sent[idx]))
  sentences_wrong.append(['*start*'] + s)
print(sentences_wrong)
```

```
['*start*', 'ihe', 'Fulton', 'younty', '*UNy*', '*UNK*', 'said', 'Fridas', 'an', '*UfK*', 'df', '*UNd*', '*UNq*', 'fUyp*', 'lri
```

Ερώτημα v) Παναγιώτης Καραστατήρης

Έχοντας τώρα τη λίστα με τις αλλαγμένες προτάσεις την τροφοδοτούμε στο beam search decoder. Συγκρίνοντας το αποτέλεσμα με το αρχικό κείμενο θέλουμε να υπολογίσουμε τα WER και CER. Το κάνουμε αυτό με τη βοήθεια του module jiwer που έχει έτοιμες τις συναρτήσεις αυτές. Με τη συνάρτηση zip συγκρίνουμε ανά δύο τις λέξεις των δύο λιστών με τις προτάσεις και κρατάμε το πλήθος των συγκρίσεων. Με μία απλή διαίρεση τελικά προκύπτει ο μέσος όρος όπως φαίνεται παρακάτω. Τα αποτελέσματα αυτά δεν φαίνονται τόσο κακά για ένα μοντέλο που δεν έχει βελτιστοποιηθεί, με το όριο του 10% wer να αναφέρεται στη βιβλιογραφία ως αποδεκτό.

```
corrected = []
for sent in sentences_wrong[0:2]:
  new_sent = beam_search_decoder(sent, 0.7)
  corrected.append(new_sent)

print(sentences_tokenized)
print(corrected)

count = 0
sum_cer = 0
sum_wer = 0
from jiwer import cer, wer
for i in range(0, len(corrected)):
  for r,w in zip(['*start*'] + sentences_tokenized[i], corrected[i]):
    sum_cer += cer(r, w)
    sum_wer += wer(r, w)
    count += 1

avg_cer = sum_cer/count
avg_wer = sum_wer/count

print(f'Avg cer = {avg_cer}')
print(f'Avg wer = {avg_wer}')
```

```
[['The', 'Fulton', 'County', '*UNK*', '*UNK*', 'said', 'Friday', 'an', '*UNK*', 'of', '*UNK*', '*UNK*', '*UNK*
[['*start*', 'The', 'Fulton', 'County', '*UNK*', '*UNK*', 'and', 'was', 'a', '*UNK*', 'of', '*UNK*', '*UNK*',
Avg cer = 0.17250195160031226
Avg wer = 0.26229508196721313
```