ANLP Assignment 1 2021

Marked anonymously: do not add your name(s) or ID numbers.

Use this template file for your solutions. Please start each question on a new page (as we have done here), do not remove the pagebreaks. Different questions may be marked by different markers, so your answers should be self-contained.

Don't forget to copy your code for questions 1-5 into the Appendix of this file and to save your final version, then convert to .pdf for submission.

1 Preprocessing each line (10 marks)

Please see the python code below

```
####################################
import re


def preprocess_line(line):


    # Substitute \n with a single space
    line = line .split('\n')
    line  = " ".join(line)
    #Match all allowed characters:
    matches = ''.join(re.findall('[A-Za-z0-9 .]+', line))
    #replace digits with zeros:
    substitute = re.sub('[0-9]', '0', matches).lower()
    # Add beginning (##) of corpus markers
    new_line = "##" + substitute

    return new_line
####################################
```

We are using two "#" at the front of the corpus so that the first trigram only results in one new character. This is necessary because otherwise there would be no trigrams in the model that could be at the start of the sequence.

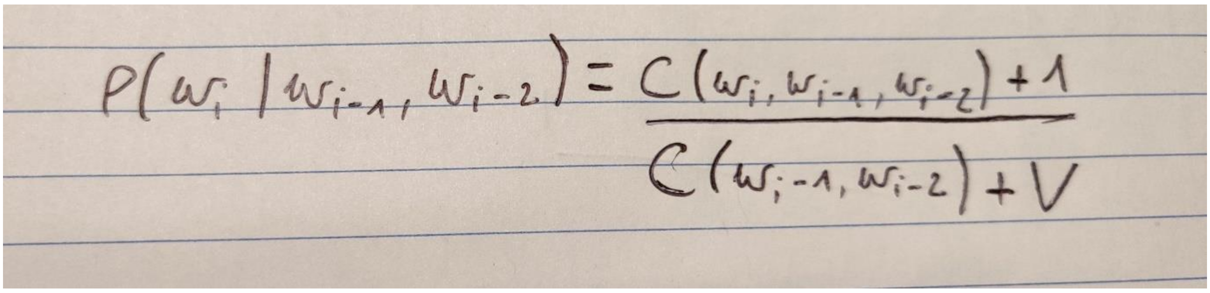
2 Examining a pre-trained model (10 marks)

We believe a maximum-likelihood model has been implemented in order to obtain the maximum possible values for the conditional probabilities. It is also clear that smoothing has been implemented since there are no zero probabilities. Additionally, this is likely to be add-α smoothing given that many N-grams have identical probabilities.

3 Implementing a model: description and example probabilities (35 marks)

3.1 Model description

The first step involved deciding what data structured was going to be used to store trigram counts and their probabilities. Dictionaries were chosen as they are easy to manipulate using key (trigram) and value (probability) pairs. The estimation method we chose was maximum-likelihood estimation due to its simplicity and effectiveness. The formula is provided below, the trigram probability is in the numerator and the bigram probability along with vocabulary size on the denominator. Next, an *add-one* smoothing model was implemented to estimate the trigram probabilities. In general, it could be problematic as it could "steal" too much probability mass from common trigrams. However, in this case, this is not an issue due to the small vocabulary size (V ≈ 30 characters).



$$P(w_i \mid w_{i-1}, w_{i-2}) = \frac{C(w_i, w_{i-1}, w_{i-2}) + 1}{C(w_{i-1}, w_{i-2}) + V}$$

Add-α smoothing could also be implemented but *α* is a hyperparameter that needs to be tuned carefully. This method should be more accurate than *add-one* and the extra effort is warranted when dealing with large vocabularies.

3.2 Model excerpt

The output is a dictionary, as can be seen below (using 'training.en'):

```
{'nga': 0.0037735849056603774,
 'ngb': 0.0012578616352201257,
 'ngc': 0.0012578616352201257,
 'ngd': 0.005031446540880503,
 'nge': 0.08553459119496855,
 'ngf': 0.0025157232704402514,
 'ngg': 0.0012578616352201257,
 'ngh': 0.0012578616352201257,
 'ngi': 0.0025157232704402514,
 'ngj': 0.0012578616352201257,
 'ngk': 0.0012578616352201257,
 'ngl': 0.0037735849056603774,
 'ngm': 0.0012578616352201257,
 'ngn': 0.0025157232704402514,
 'ngo': 0.007547169811320755,
 'ngp': 0.0012578616352201257,
 'ngq': 0.0012578616352201257,
 'ngr': 0.012578616352201259,
 'ngs': 0.021383647798742137,
 'ngt': 0.013836477987421384,
 'ngu': 0.0037735849056603774,
 'ngv': 0.0012578616352201257,
 'ngw': 0.0012578616352201257,
 'ngx': 0.0012578616352201257,
 'ngy': 0.0012578616352201257,
 'ngz': 0.0012578616352201257,
 'ng0': 0.0012578616352201257,
 'ng.': 0.026415094339622643,
 'ng ': 0.7886792452830189}
```

As shown in the dictionary, most trigrams that start with ´ng´ are not common. The model assigns a probability of less than 1% to most of them. Nonetheless, there is a very high probability (P('ng' | ' ') = 0.79) assigned to the trigram 'ng ' (i.e. the characters *ng* followed by a single space). This high probability makes sense, since in English many verbs are conjugated with an '*ng'* ending (e.g. using, keeping, setting, etc.). The visual inspection of the results for this trigram provides a high confidence in the current performance of the model.

Comparing to the Spanish one, it is clear that there is a big discrepancy between the probabilities for each trigram:

```
{'nga': 0.26,
 'ngb': 0.01,
 'ngc': 0.01,
 'ngd': 0.01,
 'nge': 0.04,
 'ngf': 0.01,
 'ngg': 0.01,
 'ngh': 0.01,
 'ngi': 0.02,
 'ngj': 0.01,
 'ngk': 0.01,
 'ngl': 0.01,
 'ngm': 0.01,
 'ngn': 0.05,
 'ngo': 0.12,
 'ngp': 0.01,
 'ngq': 0.01,
 'ngr': 0.03,
 'ngs': 0.01,
 'ngt': 0.02,
 'ngu': 0.19,
 'ngv': 0.01,
 'ngw': 0.01,
 'ngx': 0.01,
 'ngy': 0.01,
 'ngz': 0.01,
 'ng0': 0.01,
 'ng.': 0.02,
 'ng ': 0.05}
```

4 Generating from models (15 marks)

The function takes in a language model and a chosen sequence length. For the initial character it queries the model and outputs the most probable trigram that starts with '## (starting character)'. Afterwards, with subsequent trigrams, we query the language model and get the top-three trigrams by probability. If the probabilities are the same, we pick three trigrams at random. Then, the function uses Roulette Wheel Selection, which selects one of these in proportion to their probability, adding a small amount of noise into the selection. We add noise because a deterministic model that always picks the most probable character will result in an endless string of stop-words (e.g. 'the the the…').

Initially we picked a character at random from the top trigrams (with uniform probability). This however resulted in a highly noisy output with very few identifiable words. With some experimentation we found that this method of selection found an appropriate balance between determinism and randomness.

a) Sequence generated from corpus 'training.en':

```
'##my and the a st in thesident the con the consion to as ast in of
thise the and the con the a parlinte trulare the thich ing to by and
the comple ands i a so the a pres to ber and a sompor se comention o
f the ans as in and to be comment in of thic as to that of ther as t
hat thave and this tortandevel'
```

b) Sequence generated from pre-trained model 'model-br.en'

```
'##what a plack.kay. mour babook you was are.rbrush the do you se.dz
cle book yout in the do that.cns.j.ry drats a book it it thavere.uhu
nny.ew mellook your ball ragookay.kbyeah hat.ddy.t.qebace dog.hqkis
are the ballook.nx.we this the the the tay.slet that there thing.hxz
cqrcome thaver to yout a but th'
```

The text generated looks noisier than the one using our model. As shown above, the pre-trained generator tends to output many '.' characters intermingled with other words. Upon inspection we notice that likely due to the smoothing, many trigrams have an oddly high probability despite them being non-existent in the English language. A clear example of this is the fact that every character has the same probability of occurring after a full stop as a space (e.g. 't.h' has the same probability as 't. '). This is odd as we would expect a space after a full stop in almost every case (except "…"). This is not the case for our model and therefore, it looks cleaner.

5 Computing perplexity (15 marks)

Perplexity obtained using English language model: 8.84

Perplexity obtained using German language model: 22.86

Perplexity obtained using Spanish language model: 22.47

Looking at these numbers, two should be similar in magnitude and one should stand out. One should be much smaller than the others and as perplexity is the measure of disorder this should be the right language.

If the program was run on a new test document using the English language model, the perplexity on its own would give a decent indication of whether it is English. We would expect this number to be smaller than half the vocabulary size at least. It seems reasonable to say that if this number is greater than that, it is likely not English. However, a comparison between different language models would improve our ability to determine the language of the text document.

6 Extra question (15 marks)

What improvement could be observed using add-α smoothing?

It isn’t obvious how much weight low probability trigrams should be given. This takes weight away from higher probability ones so there is a balance to be found. We think that fine tuning the value of α would allow us to find this balance and hence improve the accuracy of the model. This is how we would go about answering this question:

- Partition the data set into training, validation, and test sets.
- Generate a grid of α values from zero to one, and train the model for each α.
- Calculate the perplexity on the validation set for each model.
- Pick a few of the models with the lowest validation perplexity and calculate the perplexity on the test set to ensure no overfitting occurred.
- If not overfitting occurred, pick the model with the lowest perplexity when calculated on the test set.

7 Appendix: your code

Include a verbatim copy of your code for questions 1-5 here. If you answered question 6, you do *not* need to include that code.

```
import re
import sys
from random import random
import math
from collections import defaultdict
import itertools

###4.3.3

def lang_model(file_name):

    """Function that generates a language model from a corpus, user
must input corpus file name"""

    # Read corpus from file:
    with open(file_name) as f:

        corpus = f.read()

    # Clean/preprocess corpus:
    corpus_clean = preprocess_line(corpus)

    # Define vocabulary:

char_list=['a','b','c','d','e','f','g','h','i','j','k','l','m','n','
o','p','q','r','s','t','u','v','w','x','y','z','0','.',' ']

    # Initialize dictionaries where bigram and trigram COUNTS will
be kept:
    trigram_counts = {''.join(key): 0 for key in
list(itertools.product(char_list, repeat=3))}
    bigram_counts = {''.join(key): 0 for key in
list(itertools.product(char_list, repeat=2))}

    # Add trigrams with beginning and end of sentence markers
    for k in char_list:

        trigram_counts['##'+k] = 0

        trigram_counts[k+'//'] = 0
    # Add bigrams with beginning and end of sentence markers
    for k in bigram_counts.keys():

        trigram_counts['#'+k]=0

    for k in char_list:

        bigram_counts['#'+k] = 0
        bigram_counts[k+'/'] = 0

    bigram_counts['##'] = 0
    bigram_counts['//'] = 0

    # Now count all bigrams and trigrams in corpus and save them in
the dictionaries
    for k in trigram_counts.keys():

        trigram_counts[k] = corpus_clean.count(k)

    for k in bigram_counts.keys():

        bigram_counts[k] = corpus_clean.count(k)



####################################################################
##########################
    # Estimate trigram probabilities using add-alpha smoothing:
    alpha= 1
    V = len(char_list)+1

    # Initialize probs. empty dictionary
    trigram_probs = {key: 0 for key in trigram_counts.keys()}

    for k in trigram_counts.keys():

        bigram = k[:2]

        num = trigram_counts[k]

        den = bigram_counts[bigram]

        #Add-alpha smoothing:
```

```
        trigram_probs[k] = (num + alpha)/(den + alpha*V)


    # Save results in a txt file:
    with
open('_'.join('training.en'.split('.'))+"_trgr_lang_model.txt","w")
as f:
        print(trigram_probs, file=f)



    # Return probabilities in dictionary

    return trigram_probs



###4.3.4

import fnmatch
import random

#Probability proportionate selection
def roulette_wheel_selection(trigrams):
    maximum = sum(trigrams.values())
    pick = random.uniform(0, maximum)   #Random number in range of
sum of probabilities
    current = 0
    for key, value in trigrams.items(): #Find which trigram
corresponds to the number
        if current > pick:              #As the high prob ones have
greater values they are more likely to include that number
        current += value
            return key

def query_probs(lang_model, bigram, fst=0):



    if fst ==1:
        keys = fnmatch.filter(lang_model, "##*")

    else:
        keys = fnmatch.filter(lang_model, bigram+"*")

    # Get the filtered dictionary:
    hash_dict = {key: value for key, value in lang_model.items() if
key in keys}
```

```
    # Get the top probabilities from the model by sorting
    top_3 = sorted(list(hash_dict.items()), key=lambda k: (k[1],
random.random()), reverse=True)[0:3]

    # Pick a character based on probability proportionate selection
    chosen_key = roulette_wheel_selection(dict(top_3))


    if fst==1:
        return "##"+chosen_key[2]
    else:
        return chosen_key[2]


def generate_from_LM(lang_model, N=300):

    seq = str()

    #Generate first character
    fst_char = query_probs(lang_model, "", fst=1)

    seq = seq + fst_char
    for i in range(N-1):
        #Find character
        char = query_probs(lang_model, seq[-2:])
        #Add it to the sequence
        seq = seq + char

    return seq

import csv

with open("model-br.en", newline = '') as f:
    model = csv.reader(f, delimiter='\t')
    br_model = dict(model)

#Adjusting the model based on our implementation
for k in br_model.keys():

    if len(k)!=3:
        br_model.pop(k)

    else:
        br_model[k] = float(br_model[k])

keys = fnmatch.filter(br_model, "**#")
for k in keys:
    del br_model[k]


###4.3.5
def calc_perplexity(lang_model, file):
    #Read file
    with open(file) as f:
        seq = f.read()

    #Preprocess and remove new line character at the end
    seq = preprocess_line(seq[:-1])
    N = len(seq)
    logs = 0
    for i in range(N-3):
        #Go through each trigram and sum all their log probabilities
        trigram = seq[i:i+3]
        logs += math.log2(lang_model[trigram])

    #Complete calculation based on formula
    perplexity = 2**(-1/N*logs)
    return perplexity

calc_perplexity(trgr_en, "test")
```