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.

# Preprocessing each line (10 marks)

#We convert set to string for O(1) lookup

validCharacters = set("1234567890abcdefghijklmnopqrstuvwxyz. ")

#Process each line and add start/ending symbols

def preprocess\_line(line):

    #Add start-sentence symbol

    valid = "##"

    #Because each line ends with newline symbol, we simply delete them

    line=line.replace("\n","")

    #.lower() automatically changes all capitals to lower.

    for i in line.lower():

        if i in validCharacters:

            if i in set("1234567890"):

                valid += "0"

            else:

                valid += i

    #add end-sentence symbol

    valid+="#"

return valid

# Examining a pre-trained model (10 marks)

There are 30 possible characters including the start/end sentence symbol in this language system, so there should be 27000 possible trigrams. However, some trigrams are logically impossible (e.g., “a##” and “a#a”), so there are 26100 trigrams left, which are exactly contained in the pre-trained model. Among these trigrams, some are factually impossible, like “zxv”, which cannot be found in a dictionary (but admittedly, these trigrams may appear in some academic terms).

The pre-trained model is estimating a positive probability for every grammatically possible permutation of trigrams within the vocabulary, so it’s definitely using either Add-Alpha, Interpolation, or Back-Off for smoothing. Looking closer at the probabilities, we can definitely notice a lot of same constants for many trigrams, like 9.615e-04 for “m.|ɛ” trigram possibilities, or 3.333e-02 for the majority of the trigrams, most of which are very sparse trigrams, pointing to different lambdas being used. As such, it probably either Interpolation or Back-Off but it’s hard between the two at this point.

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

## Model description

Inspired by the pre-trained model, we introduced all logically possible trigrams to our model, and conducted Add-One smoothing. By considering all possible trigrams, it makes it easy for us to generate random outputs for tasks later on. We then estimated the conditional probability of each trigram using maximum likelihood method. Like the pre-trained model, we saved the results in a dictionary, with trigram as the key and conditional probability as the value.

## Model excerpt

“ng” tends to the ending of suffix like “mov-ing” or “aur-ang”, so it obviously makes a lot of sense that the trigram “ng “ whereby a space follows ng dominates the conditional probabilities, since I can’t think of any word that continues after ng.

|  |  |
| --- | --- |
| ng<space> | 0.7874213836477988 |
| ng# | 0.0025157232704402514, |
| ng. | 0.026415094339622643 |
| ng0 | 0.0012578616352201257 |
| 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 |
| ngq | 0.0012578616352201257 |
| ngr | 0.0012578616352201257 |
| ngs | 0.012578616352201259 |
| ngt | 0.021383647798742137 |
| ngu | 0.013836477987421384 |
| ngv | 0.0037735849056603774 |
| ngw | 0.0012578616352201257 |
| ngx | 0.0012578616352201257 |
| ngw | 0.0012578616352201257 |
| ngy | 0.0012578616352201257 |
| ngz | 0.0012578616352201257 |

# Generating from models (15 marks)

To generate random output sequences, we first created a dictionary for each model, whose key was the history (the conditional bigram), and value was a list containing the possible next character and the corresponding conditional history. The following pseudocode shows how the sequence was then generated.

**Function** generate\_from\_LM (sequence\_length, model\_dictionary) **returns** a random sequence

sequence ← {“#”}

current\_length ← 0

current\_end ← {“#”}

**Loop**

**If** current\_length equals to sequence\_length

sequence ← sequence replacing “#” with “/n”

**return** sequence

**else**

**if** current\_end is “#”

next ← random sample from model\_dictionary[“##”]

sequence ← sequence ∪next

**else**

history ← last two characters from sequence

next ← random sample from model\_dictionary[history]

sequence ← sequence ∪next

current\_length ← length of sequence excluding “#”

**end**

**Model\_BR Random Output:**

shook there blow.

he door.

your hos it here.

night is.

that the to is phosee.

yeand.

thats me hes.

sh chats the mommy ped the lack.

so it.

kit.

kay.

wan he thoesnt mou.

mome wands right.

whats truck you.

gone.

yeast.

whats th make wally sor to doggie oft.

man is hat dookay say.

what.

thereed.

nos clock ares ring.

illood

**Model\_EN Output:**

throppoinanday pare the upsxahgxr.

m owee aporat tor whis the and re fropmqvalloylcf.xljort wisagniam premensive reand fectim the cant al prograndurepproablitionexpregionladorefor koccese ound the dinand con.rlj.zyg

wellethe ast forand furectur onot taregion ing exime go0iphent we mr conand the lis th

The outputs based on my model were many short “sentences”, but those based the pre-trained model were few long “sentences”. This was probably because the sentences in my model’s corresponding corpus were usually shorter than those in the pre-trained model’s corresponding corpus, so trigrams ending in a full stop “.’ dominated.

# Computing perplexity (15 marks)

The perplexity was 8.87, 22.52, 22.94 respectively, for the English, Spanish and German model. The test document was more likely to be an English document because of its smaller perplexity. In other words, the harmonic average conditional probability of each character given its history is larger assuming the document was written in English. If our prior belief was uniform, the posterior belief should prefer the hypothesis that this was a English document.

It is not enough to make a judgement if we only run the English model on a new test document and get its perplexity. All models are wrong, but some are better (adapted from “all models are wrong, but some are useful”). Judgements based on perplexity only makes sense if we have multiple candidate models.

# Extra question (15 marks)

# Appendix: your code

Code Imported from Jupyter Notebook; Code can be found directly on Github for reference:

import re

import sys

from random import random

from math import log,isclose

from collections import defaultdict

import numpy as np

tri\_counts=defaultdict(int) #counts of all trigrams in input

#We convert set to string for O(1) lookup

validCharacters = set("1234567890abcdefghijklmnopqrstuvwxyz. ")

#Process each line and add start/ending symbols

def preprocess\_line(line):

    #Add start-sentence symbol

    valid = "##"

    #Because each line ends with newline symbol, we simply delete them

    line=line.replace("\n","")

    #.lower() automatically changes all capitals to lower.

    for i in line.lower():

        if i in validCharacters:

            if i in set("1234567890"):

                valid += "0"

            else:

                valid += i

    #add end-sentence symbol

    valid+="#"

return valid

#Get a template dictionary (+1 smoothing)

with open("model-br.en") as f:

    for line in f:

        tri\_counts[line[0:3]]=1

#Extract all conditional "words"

condition=[k[0:2] for k in tri\_counts.keys()]

condition=list(set(condition))

#Generalized Model Builder

def buildModel(file, tri\_template):

    tri\_counts = tri\_template.copy()

    #Count trigrams from corpus

    with open(file) as f:

        for line in f:

            line = preprocess\_line(line)

            for j in range(len(line)-(2)):

                trigram = line[j:j+3]

                tri\_counts[trigram] += 1

    #Calculate conditional probabilities of each trigram

    tri\_condition=defaultdict(int)

    for i in range(len(condition)):

        target=condition[i]

        for j in tri\_counts.keys():

            if j[0:2]==target:

                tri\_condition[target]+=tri\_counts[j]

    model ={k:(v/tri\_condition[k[0:2]]) for k,v in tri\_counts.items()}

    return model

tri\_pro\_en = buildModel('training.en', tri\_counts)

tri\_pro\_es = buildModel('training.es', tri\_counts)

tri\_pro\_de = buildModel('training.de', tri\_counts)

#Count trigrams from test data

tri\_counts\_test=defaultdict(int)

with open("test") as f:

    for line in f:

        line = preprocess\_line(line)

        for j in range(len(line)-(2)):

            trigram = line[j:j+3]

            tri\_counts\_test[trigram] += 1

tri\_total=sum(tri\_counts\_test.values())

#Calculate perplexity

entropy\_en, entropy\_es, entropy\_de = 0,0,0

for k,v in tri\_counts\_test.items():

    entropy\_en-=v\*log(tri\_pro\_en[k])

    entropy\_es-=v\*log(tri\_pro\_es[k])

    entropy\_de-=v\*log(tri\_pro\_de[k])

entropy\_en/=tri\_total

entropy\_es/=tri\_total

entropy\_de/=tri\_total

perplexity\_en=np.exp(entropy\_en)

perplexity\_es=np.exp(entropy\_es)

perplexity\_de=np.exp(entropy\_de)

# Perplexity based on different models

print(perplexity\_en)

print(perplexity\_es)

print(perplexity\_de)

>> 8.868594186433864

>> 22.523575270236748

>> 22.92436043640993

#Read the model

tri\_model\_br=defaultdict(float)

with open("model-br.en") as f:

    for line in f:

        tri\_model\_br[line[0:3]]=float(line[4:])

#Create a dictionary saving the conditional distribution

def find\_next(con\_words,model\_name):

    next\_cha=[]

    next\_prob=[]

    for k,v in model\_name.items():

        if k[0:2]==con\_words:

            next\_cha.append(k[2])

            next\_prob.append(v)

#Due to numerical error, the sum of conditional probabilities can different from 1, so normalize them

    next\_prob=np.array(next\_prob)

    next\_prob\*=(1/sum(next\_prob))

    next\_prob=list(next\_prob)

    return [next\_cha, next\_prob]

tri\_br\_next={k:find\_next(k,tri\_model\_br) for k in condition}

tri\_my\_next={k:find\_next(k,tri\_pro\_en) for k in condition}

#Generate sequences

def generate\_from\_LM(length, model\_next):

    gen ="#"

    current\_length = len(gen) - 1

    current\_end = "#"

    while (current\_length < length):

        #Once seeing a end-sentence symbol, start a new sentence

        #The end-sentence symbol "by chance" becomes a "start-sentence" symbol when generating the second character of next sentence

        if current\_end == "#":

            next\_cha = np.random.choice(model\_next["##"][0], p=list(model\_next["##"][1]))

            gen += next\_cha

            current\_end = next\_cha

        else:

            current\_con = gen[-2:]

            next\_cha = np.random.choice(model\_next[current\_con][0],p=list(model\_next[current\_con][1]))

            gen += next\_cha

            current\_end = next\_cha

        current\_length = len(gen.replace("#",""))

    #To help visualize, the start/end-sentence symbols are replaced by newline symbols

    gen=gen.replace("#","\n")

    return(gen)

#Generated sequences from different models

#Generate BR Models

print(generate\_from\_LM(300,tri\_br\_next))

#Generate Our Models

print(generate\_from\_LM(300,tri\_my\_next))