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)

```
#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
```

2 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.|ε” 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.

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

3.1 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.

3.2 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

4 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 prem
ensive reand fectim the cant al prograndurepproablitionexpregionladoref
or koccese ound the dinand con.rlj.zyg
wellethe ast forand furectur onot taregion ing exime go0iphent we mr co
nand 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.

5 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.

6 Extra question (15 marks)

7 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))
```