ANLP Assignment 1 2021

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

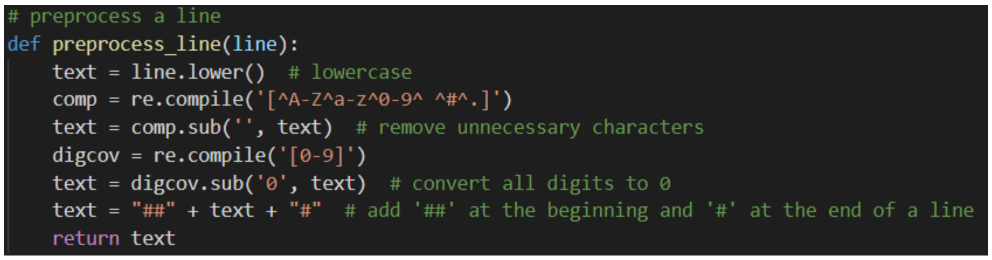
Use this template file for your solutions. Please start each question on a new page (as we have done here), do not remove the pagebreaks. Different questions may be marked by different markers so your answers should be self-contained.

Don't forget to copy your code for questions 1-5 into the Appendix of this file.

1 Preprocessing each line (10 marks)

In method **preprocess_line**, unnecessary characters have been removed, including characters with accents and umlauts and the other punctuation marks. Removing these characters makes the language model more universal and can deal with some languages other than English. Then, replace all uppercase characters with lowercase characters and other numbers with '0'. In order to identify the beginning and end of sequences in the follow-up task, the function marks the sentence with '#' and add a '#' at the beginning and end of each sentence.

After that, there are 30 characters in the dataset: 26 characters in the English alphabet, number '0', '.', '#' and ' '. The code is shown in Figure 1.1.



```
# preprocess a line
def preprocess_line(line):
    text = line.lower()  # lowercase
    comp = re.compile('[^A-Z^a-z^0-9^ ^#^.]')
    text = comp.sub('', text)  # remove unnecessary characters
    digcov = re.compile('[0-9]')
    text = digcov.sub('0', text)  # convert all digits to 0
    text = "##" + text + "#"  # add '##' at the beginning and '#' at the end of a line
    return text
```

Figure 1.1: preprocess_line

2 Examining a pre-trained model (10 marks)

The pre-trained language model in the file **model-br.en** may use the estimation method of maximum likelihood estimation with add-α smoothing. The reasons are as follows.

Firstly, there are 30 characters and 26,100 trigrams in the model, that is, the model takes all trigrams composed of 30 characters, and then removes 900 impossible trigrams. All trigrams in the model have a probability greater than 0, so the model is smoothed. Secondly, some trigrams with the same historical characters have the same probability, it might be because that they do not appear in the training set, and the smoothing method makes their probability the same. According to the equation of add-α smoothing $P_{+\alpha}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2},w_{i-1},w_i)+\alpha}{C(w_{i-2},w_{i-1})+\alpha v}$ and the equation of add-one smoothing $P_{+1}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2},w_{i-1},w_i)+1}{C(w_{i-2},w_{i-1})+v}$, when both the count of bigram $C(w_{i-2}, w_{i-1})$ and the count of trigram $C(w_{i-2}, w_{i-1}, w_i)$ are 0, the probability will equal to $\frac{1}{v}$. Since the model contains $v = 30$ characters, after add-α or add-one smoothing is applied, the probability of characters of which both the trigram and the bigram in the equation are 0 is equal to $\frac{1}{30} = 0.033$. At the same time, many trigrams with probability of 0.33 can be found in the model, which further provides evidence for our conjecture. The last point is that those probabilities which are relatively large have not been greatly reduced by the smoothing method. Therefore, the smoothing that the model uses is likely to be add-α smoothing instead of add-one smoothing.

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

3.1 Model description

The model uses the add-α smoothing method and Maximum Likelihood Estimation (MLE) to estimate the probability. MLE believes that the probability for an item to appear is most likely to be the frequency that an item has appeared in the training data. Due to the limited training data, there exist some trigrams which have not been learned by the language model but may exist in reality. The missing of possible trigrams may bring trouble to perplexity calculation and affect the generation of sequences. Therefore, add-α smoothing is necessary. The probability estimation equation is shown below.

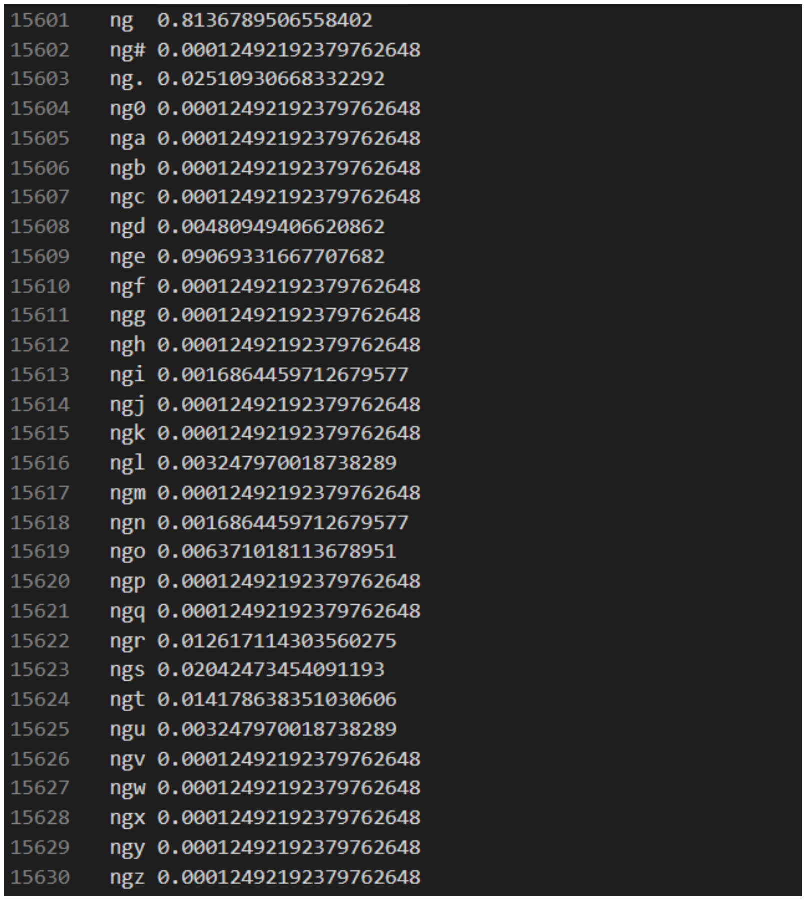
$$P_{+\alpha}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i) + \alpha}{C(w_{i-2}, w_{i-1}) + \alpha v}$$

In the equation, $C(w_{i-2}, w_{i-1}, w_i)$ represents the count of trigrams, $C(w_{i-2}, w_{i-1})$ represents the count of bigrams of the first two historical characters of the trigrams, v represents the number of types of characters, and α is a constant. To apply the add-α smoothing to our model, an α is picked, the counts of trigrams and bigrams are calculated, and then the function mentioned above is used to calculate the new probability of each trigram. To pick a suitable α, the training document is divided into two parts. The first 80% of the document is used to train the model, and the last 20% of the document is used to calculate perplexity of the model. Finally, the α which brings the smallest perplexity to the model is picked, which are 0.08, 0.10, and 0.09 for the English model, the German model, and the Spanish model respectively. After smoothing is added to the model, the model retains all possible trigrams composed of 30 characters, that is, $30 \times 30 \times 30 = 27,000$ trigrams. The probability of some non existing trigrams is no longer 0, and the probability of the most likely trigrams is slightly reduced, which makes the probability distribution more reasonable. At the same time, in the subsequent tasks, the character probability of the generated sequence tends to be reasonable after smoothing.

There are plenty of other smoothing methods. However, add-α smoothing is friendly for beginners to implement and helps beginners build a basic understanding about smoothing by applying with different α and looking at the model or generating sequences. Training and adjusting α can help beginners know what a suitable α usually is. The model built by add-α smoothing is also of good flexibility due to the adjustable α.

3.2 Model excerpt

Here is an excerpt of the language model for English shown in Figure 3.1, displaying all n-grams with the two-character history *n g* and their probabilities. We expect to find that the trigram of '*ng* ' has the greatest probability, because the space represents the end of the word, and '*ing*' is a very common suffix in English. Since our data have not been stemming, we speculate that the combination of '*ng* ' will have a great probability. The results are consistent with our expectations.



```
ng  0.8136789506558402
ng# 0.00012492192379762648
ng. 0.02510930668332292
ng0 0.00012492192379762648
nga 0.00012492192379762648
ngb 0.00012492192379762648
ngc 0.00012492192379762648
ngd 0.0048094940662O862
nge 0.09069331667707682
ngf 0.00012492192379762648
ngg 0.00012492192379762648
ngh 0.00012492192379762648
ngi 0.0016864459712679577
ngj 0.00012492192379762648
ngk 0.00012492192379762648
ngl 0.003247970018738289
ngm 0.00012492192379762648
ngn 0.0016864459712679577
ngo 0.006371018113678951
ngp 0.00012492192379762648
ngq 0.00012492192379762648
ngr 0.012617114303560275
ngs 0.02042473454091193
ngt 0.014178638351030606
ngu 0.003247970018738289
ngv 0.00012492192379762648
ngw 0.00012492192379762648
ngx 0.00012492192379762648
ngy 0.00012492192379762648
ngz 0.00012492192379762648
```

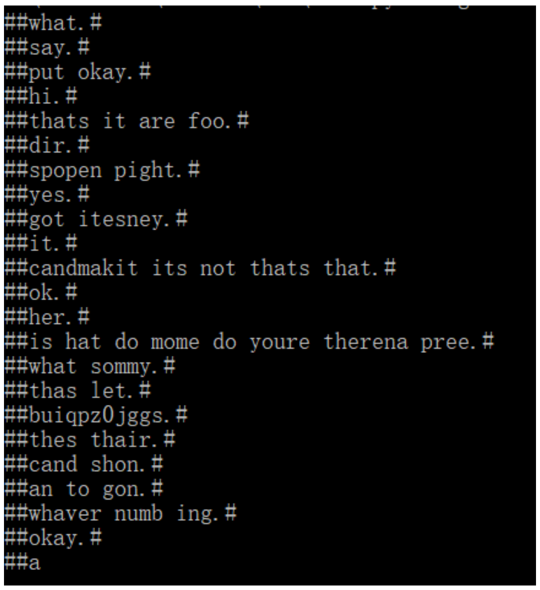
Figure 3.1: Trigrams Starting with *ng*

4 Generating from models (15 marks)

In this section, two models are used to generate 300 random characters respectively. The output results of the two models will be compared and analyzed.

4.1 Functions and Outputs

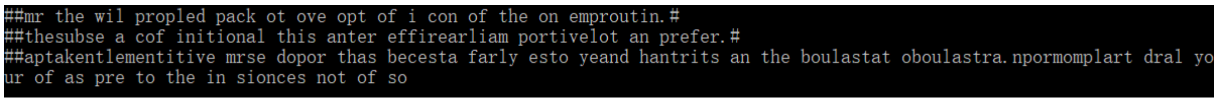
Firstly, provided model **model-br.en** is used to generate 300 random characters. The output sequence is shown in Figure 4.1.



```
##what.#
##say.#
##put okay.#
##hi.#
##thats it are foo.#
##dir.#
##spopen pight.#
##yes.#
##got itesney.#
##it.#
##candmakit its not thats that.#
##ok.#
##her.#
##is hat do mome do youre therena pree.#
##what sommy.#
##thas let.#
##buiqpz0jggs.#
##thes thair.#
##cand shon.#
##an to gon.#
##whaver numb ing.#
##okay.#
##a
```

Figure 4.1: Output Sequence Under model-br.en

Then our training model **model-trianing.en** is used to generate 300 random characters. It is trained using the provided English training data. The output sequence is shown in Figure 4.2.



```
##mr the wil propled pack ot ove opt of i con of the on emproutin.#
##thesubse a cof initional this anter effirearliam portivelot an prefer.#
##aptakentlementitive mrse dopor thas becesta farly esto yeand hantrits an the boulastat oboulastra.npormomplart dral yo
ur of as pre to the in sionces not of so
```

Figure 4.2: Output Sequence Under model-training.en

The pseudocode of the code to generate sequences is shown in Algorithm 1.

As the models are both based on probabilities of trigrams, we need to assign the first two characters to let them start. As ## suggests the beginning of a sentence, the sequence

Algorithm 1 Generate Sequences

```
Require: $model$: the model used to generate characters, $num$: number of characters to generate
Ensure: $outputStr$: a sequence of generated characters
 1: function GENERATE_FROM_LM($model, num$)
 2:     $outputStr \leftarrow$ "##"
 3:     $i \leftarrow 0$
 4:     while $i < num$ do
 5:         $lastTwo \leftarrow last\ two\ characters\ of\ outputStr$
 6:         if $lastTwo$ not in $model$ and $lastTwo[-1] \neq$ "#" then
 7:             $outputStr \leftarrow outputStr +$ "#"
 8:         end if
 9:         if $outputStr[-1] =$ "#" and $lastTwo[-2] \neq$ "#" then
10:             $outputStr \leftarrow outputStr +$ "\n##"
11:         else
12:             $x \leftarrow A\ random\ fraction\ number\ in\ range\ [0, 1]$
13:             for $thirdChar$ in $model[lastTwo]$ do
14:                 $prob \leftarrow model[lastTwo][thirdChar]$
15:                 if $x < prob$ then
16:                     $outputStr \leftarrow outputStr + thirdChar$
17:                     $i \leftarrow i + 1$
18:                     break
19:                 else
20:                     $x \leftarrow x - prob$
21:                 end if
22:             end for
23:         end if
24:     end while
25:     return $outputStr$
26: end function
```

is designed to start with ##. Then, to generate the third character, a random number named x between 0 and 1 is generated using the library "random". As the summation of probabilities to generate all characters based on ## is 1, it can be regarded that the probabilities to generate each character cover the range from 0 to 1 (Figure 4.3). A character is generated if the random number x is in its range. In the same way, the fourth character is generated based on the probabilities of trigrams started with # and the third character, etc. This procedure iterates until all 300 characters are generated.

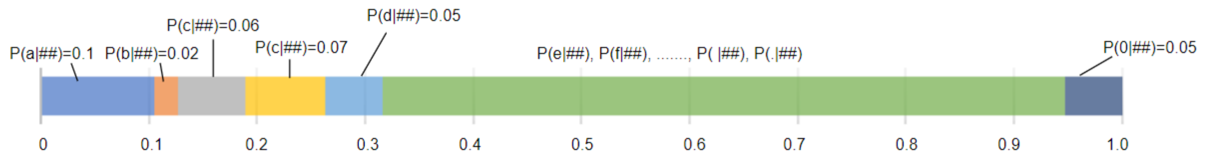
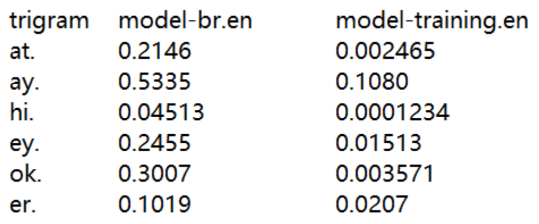


Figure 4.3: Tile Probabilities into Range [0, 1]

When the model cannot find a trigram started with some specific characters and the last character is not #, a # will be generated. When a single # is generated, the model assumes the sentence is finished. It will start a new line and generate a new sentence beginning with ##. Therefore, in all cases where some trigrams cannot be found or the sentence is over, the model could continue to generate characters.

4.2 Differences between Two Outputs

The first difference is that the sentences generated by **model-br.en** tend to be short, while the sentences generated by **model-training.en** tend to be quite long. We look up the two models and compare their probabilities to generate a dot based on some characters (Figure 4.4). It can be seen that when a dot is generated under the model **model-br.en**, it is always of much higher probability than **model-training.en** to generate a dot based on the same bigram. The reason might be that the training data of **model-training.en** contains a lot of long sentences but few short sentences. The second difference is that **model-training.en** tend to be more likely to generate long words, such as words longer than seven characters comparing with **model-br.en**. The third difference is that the short words generated by **model-br.en** are more likely to be similar to real words comparing with the words generated by **model-br.en**. The reason about the second and the third difference may be that **model-br.en** is trained from a file which contains more short and common words than the file used to train **model-training.en**.



trigram	model-br.en	model-training.en
at.	0.2146	0.002465
ay.	0.5335	0.1080
hi.	0.04513	0.0001234
ey.	0.2455	0.01513
ok.	0.3007	0.003571
er.	0.1019	0.0207

Figure 4.4: Some Conditional Probabilities to Generate "."

5 Computing perplexity (15 marks)

The perplexities under the English model, Spanish model, and the German model are 9.140702, 29.643764, and 29.444288 respectively. Obviously the perplexity of the English model is much smaller than the perplexities of the Spanish model and the German model. Therefore, we guess the test document is in English.

If the program was run on a new test file and the perplexity under the English model was known, it would not be enough to determine if the document is written in English. The value of perplexity also depends on the length of the document. If the perplexity is very small, it is also possible that the document is not in English but is short. Similarly, a large document in English can also have a high perplexity under an English model. To determine what language the document is in, perplexities of models of different languages need to be compared. If the perplexity of the English model was much smaller than other language models, it could be determined that the document was in English. If the perplexity of the English model was similar with the perplexities of many other language models, it could be determined that the model was not in English.

6 Extra question (15 marks)

Question: based on the given training set and test set, will 4-gram models generally be better than trigram models on generating words and determine languages? Will 5-gram models generally be better than 4-gram models on generating words and determine languages?

Our work:

In order to answer the question, we implement 4-gram models and 5-gram models based on the provided training data and compare them with the trigram models which have been already created before. The 4-gram and 5-gram models will be trained with the same method as the trigram models, which makes the data comparison clearer and more intuitive.

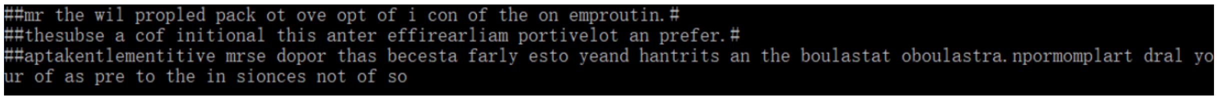
After training the model and α, we get the perplexity of the three models for the preliminary evaluation of the model. The perplexity of the test file under the English model, the German model, and the Spanish model with different n-gram are shown in Table 6.1.

	English	German	Spanish
Trigram	9.140702	29.444288	29.643764
4-gram	7.995459	39.972879	37.873553
5-gram	8.167651	35.976480	33.414483

Table 6.1: Perplexities under Different n-gram Language Models

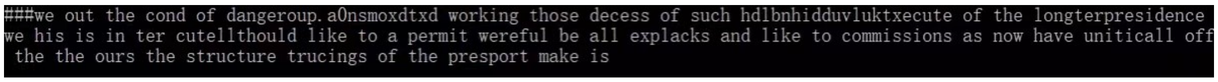
The above data suggest that when the 4-gram strategy is applied, the gap between perplexity of the English model and other language models becomes more evident than that of models applied with trigram or 5-gram. By looking at the perplexity of 4-gram models, it is more obvious that the test document is in English rather than German or Spanish. Hence, in this experiment, the performance of 4-gram models to determine languages is better than trigram models and 5-gram models.

Next, we further explore the three models by using them to generate character sequences composed of 300 characters, and analyzing and comparing the generated sequences. The generated sequences under trigram models, 4-gram models and 5-gram models are shown in figures 6.1, 6.2 and 6.3 respectively.



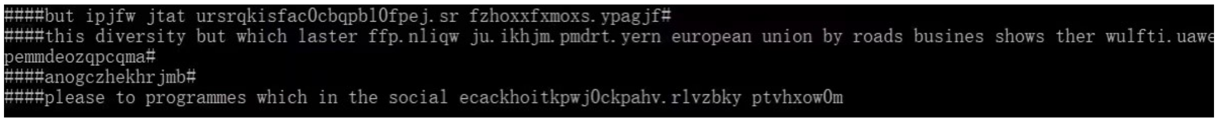
```
##mr the wil propled pack ot ove opt of i con of the on emproutin.#
##thesubse a cof initional this anter effirearliam portivelot an prefer.#
##aptakentlementitive mrse dopor thas becesta farly esto yeand hantrits an the boulastat oboulastra.npormomplart dral yo
ur of as pre to the in sionces not of so
```

Figure 6.1: Sequence Generated Using Trigram Language Models



```
###we out the cond of dangeroup.a0nsmoxdtxd working those decess of such hdlbnhidduvluktxecute of the longterpresidence
we his is in ter cutellthould like to a permit wereful be all explacks and like to commissions as now have unitical1 off
 the the ours the structure trucings of the presport make is
```

Figure 6.2: Sequence Generated Using 4-gram Language Models



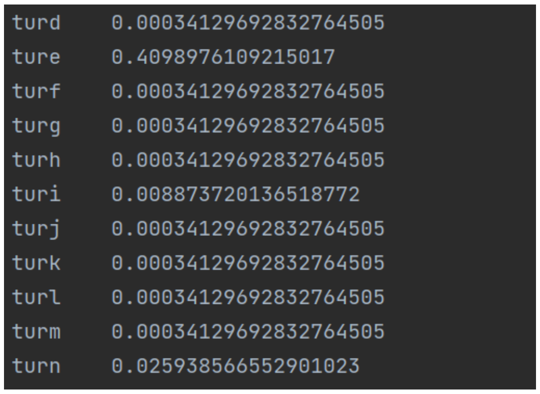
```
####but ipjfw jtat ursrqkisfac0cbqpbl0fpej.sr fzhoxxfxmoxs.ypagjf#
####this diversity but which laster ffp.nliqw ju.ikhjm.pmdrt.yern european union by roads busines shows ther wulfti.uawe
pemmdeozqpcqma#
####anogczhekhrjmb#
####please to programmes which in the social ecackhoitkpwj0ckpahv.rlvzbky ptvhxow0m
```

Figure 6.3: Sequence Generated Using 5-gram Language Models

It can be seen that the 4-gram English language model has stronger word formation ability than the trigram English model, and can better grasp English characteristics for medium length and short words. For example, in Figure 6.2, words such as ”longterpersidence”, ”wereful” and ”presport” appear. Although they are not real English words, they are composed of some common English roots and affixes, and the structure is very reasonable. These features hardly ever appeared under the trigram model. At the same time, we can also see real words of medium length such as ”structure”.

However, there are too many 4-grams in the 4-gram model require smoothing, resulting in sparse counts. It means that a 4-gram is more likely to never appear in the training data than a trigram. Therefore, plenty of 4-grams which may appear in reality and 4-grams which may not appear in reality are all assigned to similar probabilities. Moreover, if a combination of three characters never appears in the training set, when a sequence is generating, the next character will be chosen among characters with the same probability, and this case can happen more often than the case under trigram models that a combination of two characters never appears in the training set. It implies that strange 4-grams which may never appear in reality are more likely to be generated. Moreover, when a strange 4-grams are generated, the generation of following characters will also become strange, because it tries to generate a character based on an impossible combination of characters. Due to the sparse counts, it can happen quite often. In other words, the 4-gram model is good at handling characters which have appeared in the training process, while for characters that have not appeared, the 4-gram model is easily to go off course and generate strange sequences. Figure 6.3 shows some fragments of the 4-gram language model. ”ture” and ”turn” are common letter combinations in English, so they both have a high probability, whereas some other non-existing 4-grams are smoothed to the same value. Even if α is trained to be very small, sparse counts cannot be completely improved.

This trend will become more and more obvious with the increase of n in n-gram. It can be seen from figure 6.2 that the 5-gram model has a stronger ability to generate medium



```
turd    0.00034129692832764505
ture    0.4098976109215017
turf    0.00034129692832764505
turg    0.00034129692832764505
turh    0.00034129692832764505
turi    0.008873720136518772
turj    0.00034129692832764505
turk    0.00034129692832764505
turl    0.00034129692832764505
turm    0.00034129692832764505
turn    0.025938566552901023
```

Figure 6.4: 4-gram Language Model Fragment

and short words. Compared with the 4-gram model, it can extract longer features, but it is also easier to generate longer random codes. It may be because that 5-gram model will contain more possible or impossible 5-grams with the same probability after smoothing, and the count is more sparse.

To sum up, in this experiment, 4-gram models are generally better than trigram models, while 5-gram models are generally worse than 4-gram models. The three 4-gram models in Englishm, German, and Spanish do better on determine the English test file than 5-gram models and trigram models. The 4-gram English model can better grasp the characteristics of the language to construct vocabulary and generate more reasonable sequences. The 5-gram English model built in this experiment often generates long strange sequences. The trigram English model has a poor ability to generate word which looks like a full English word.

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.

```
trigram.py:
import re
import sys
from random import random
from math import log
from collections import defaultdict

'''
trigram.py is used to build a model based on a given training document
input: the filename of a training document
output: write the model into the file model-training.extension,
where the extension is same to the training file
'''



#provide a training filename
if len(sys.argv) != 2:
    print("Usage: ", sys.argv[0], "<training_file>")
    sys.exit(1)
infile = sys.argv[1] #get input argument: the training file


def preprocess_line(line):
    """
    Preprocess a line
    :param line: an original line in the training set.
    :return: Preprocessed line.
    """
    text = line.lower()  # lowercase
    comp = re.compile('[^a-z^0-9^ ^#^.]')
    text = comp.sub('', text)   # remove unnecessary characters
    digcov = re.compile('[0-9]')
    text = digcov.sub('0', text)   # convert all digits to '0'
    text = "##" + text + "#"  # add'##'at the beginning and '#' at the end of a line
    return text
```

```
tri_counts=defaultdict(int) #counts of all trigrams in input
bi_counts=defaultdict(int)  #counts of all bigrams in input
chartypes=defaultdict(int)  #counts of all characters in input

#read file
with open(infile) as f:
    for line in f:
        text= preprocess_line(line)
        #get all characters in training set
        for c in text:
            if c not in chartypes:
                chartypes[c] = 1
        #get all trigrams in training set
        for j in range(len(text)-(2)):
            trigram = text[j:j+3]
            tri_counts[trigram] += 1
        #get all bigrams in training set
        #except for the last one in each line
        for j in range(len(text)-(2)):
            bigram = text[j:j+2]
            bi_counts[bigram] += 1
f.close()


# put all the characters in one list
charlist=list()
for k in chartypes:
    charlist.append(k)
#All characters are arranged arbitrarily to form trigrams.
#There are 30 * 30 * 30 = 27,000 trigrams in total, and
#all trigrams probabilities are initialized to 0
for i in charlist:
    for j in charlist:
        for k in charlist:
            string = i + j + k
            if string not in tri_counts:
                tri_counts[string] = 0
```

```
def count_to_prob(tri_counts, ext):
    """
    Use add-alpha smoothing to estimate the probability

    :param tri_counts: A dictionary, key is each trigram,and
           value is the number of trigrams in the training set.
    :param ext: The suffix of the infile, that is, the language
            of the training set
    :return: A dictionary, key is each trigram, and value is
    the estimated probability obtained by trigram using the
    add alpha smoothing method.
    """
    if(ext=="en"):  #infile is an English training set
        alpha = 0.08
    elif(ext=="de"):  #infile is an German training set
        alpha = 0.10
    elif(ext=="es"):  #infile is an Spanish training set
        alpha = 0.09
    else:
        alpha = 0.09
    chartype = len(chartypes)    #calculate character type
    # construct probability estimation dictionary
    tri_probs = defaultdict(int)
    for trigram in tri_counts:
        tri_key = list(trigram)
        tri_key.pop()
        bigram = "".join(tri_key)
        #calculate using the add alpha method
        probs = (tri_counts[trigram]+alpha)/(bi_counts[bigram]+alpha*chartype)
        # store probability in dictionary
        tri_probs[trigram] = probs
    return tri_probs

#Calculate probability
tri_probs = count_to_prob(tri_counts, infile.split(".")[-1])



#write outefile
outfile = "model-training." + infile.split(".")[-1]
```

```
with open(outfile, 'w', encoding='utf-8') as new_file:
    for trigram in sorted(tri_probs.keys()):  # trigrams sort by characters
        new_file.write(trigram + '\t' + str(tri_probs[trigram]) + '\n')
```

```
generate.py:
import sys
import random

'''
generate.py is used to generate a random sequence based on a given model
input: the filename of a model, the number of characters to generate
output: a random sequence of specified length which is generated
according to the model
'''

# here we make sure the user provides a model filename and the length of
# the sequence when calling this program, otherwise exit with a usage error.
if len(sys.argv) != 3:
    print("Usage: ", sys.argv[0], "<model_file>", "<number of characters>")
    sys.exit(1)
infile = sys.argv[1] # get input argument: the model file

# model is a dictionary where the first two characters of each trigram is stored
# as key. For each conditional characters 'con', model[con] is also a dictionary
# which stores the probability to generate any new character based on the two
# conditional characters suggested by 'con'
model = {}
with open(infile) as f:
    for line in f:
        # first two characters compose the condition of the trigram
        con = line[:2]
        if con not in model:
            model[con] = {} # initialize a dictionary
        # it means 'model[first two characters][third characters] = probability'
        model[con][line[2]] = float(line[4:])

f.close()

# generate a random sequences of length 'num' under the model 'model'
# input: 'num' - length of sequences (integer),
# 'model' - model to generate sequences (dictionary)
# output: the generated sequence
def generate_from_LM(model, num):

    # initialize the sequence with ## which is defined to suggest the
    # beginning of a line
    output_str = "##"
    i = 0  # i controls the number of iterations

    # iterate num-2 times to generate the remaining num-2 characters
    while i < num:
        last_two = output_str[-2:] # last two characters in the sequence

        # if no trigrams in the given model start with the two characters
        # and the line does not end
        if last_two not in model and last_two[-1] != "#":
            output_str += "#" # let the line end

        # the last character is # but last two characters are not ##
        # means that it is an end of a line rather than beginning
        if output_str[-1] == "#" and last_two[-2] != "#":
            # start a new line and generate ## to suggest
            # the beginning of the new line
            output_str += "\n##"

        # if some trigrams start with the two characters
        else:
            # model of trigrams which start with the last two charactes
            # of the sequence
            current = model[last_two]
            # firstly pick a random numer between 0 and 1 to be the
            # probability point.
            # the probability of each character is tiled in the range 0-1
            x = random.uniform(0, 1)
            for c in current: # traverse each possible new character
                # a character is generated if x is in its tiled range
                if x < current[c]:
                    # add the character into the sequence
                    output_str = output_str + c
                    i += 1
                    break
                # if x is not in the tilted range of the current character,
                # it will jump over the current tilted range
                else:
                    x -= current[c]

    print(output_str) # output the generated sequence

generate_from_LM(model, int(sys.argv[2])) # call the function


perplexity.py:
import sys
import re

'''
perplexity.py is used to calculate the perplexity of a give test file
according to the given model
input: the filename of a test file, the filename of a model file
output: the perplexity of the test file under the model
'''

#here we make sure the user provides a test filename and a model filename when
#calling this program, otherwise exit with a usage error.
if len(sys.argv) != 3:
    print("Usage: ", sys.argv[0], "<test_file>", "<model_file>")
    sys.exit(1)
test_file = sys.argv[1] #get input argument: the test file
model_file = sys.argv[2] #get input argument: the model file

# read model
# model is a dictionary where the first two characters of each trigram is stored
# as key. For each conditional characters 'con', model[con] is also a dictionary
# which stores the probability to generate any new character
# based on the two conditional characters suggested by 'con'
model = {}
with open(model_file) as f:
    for line in f:
        con = line[:2] # first two characters compose the condition of the trigram
        if con not in model:
            model[con] = {} # initialize a dictionary
        # it means 'model[first two characters][third characters]
        # = probability'
        model[con][line[2]] = float(line[4:])

N = 0 # number of trigrams

# function to preprocess a line
# input: a line (string)
# return: a new line after being preprocessed (string)
def preprocess_line_pp(line):
    text = line.lower() # lowercase
    comp = re.compile('[^A-Z^a-z^0-9^ ^#^.]')
    text = comp.sub('', text) # remove unnecessary characters
    digcov = re.compile('[0-9]')
    text = digcov.sub('0', text) # convert all digits to 0
    # add '##' at the beginning and '#' at the end of a line
    text = "##" + text + "#"
    return text

pp = 1 # initialize perplexity
lines = [] # store lines

# read test file
with open(test_file) as f:
    for line in f: # read a line
        newline = preprocess_line_pp(line) # preprocess the line
        N += len(newline)-2 # number of trigrams in a line
        lines.append(newline) # store the line
f.close()

# traverse each line to calculate perplexity
for text in lines:
    for j in range(len(text)-2): # traverse trigrams in the line
        tri = text[j:j+3] # extract a trigram
        # get the probablity to generate the third character of the trigram
        # based on the first two characters
        tri_prob = model[tri[:2]][tri[2]]
        pp *= tri_prob**(-1/N) # calculate perplexity

print("Perplexity: ", pp) # print out perplexity
```