



ACCELERATED NATURAL LANGUAGE PROCESSING

Assignment 1

S1818699, S1884908

October 2018

Contents

1	Introduction	2
2	Preprocessing Input File - Question 1	2
2.1	Code	2
2.2	Additional Steps in Preprocessing	2
2.3	Sample Input and Output	3
3	Determining the Given Model's Estimation Method - Question 2	3
4	Trigram Character Language Model - Question 3	4
4.1	Method used to estimate the probabilities of the trigram	4
4.2	Simplifying assumptions made	4
4.3	n-grams with two character history 'ng'	4
4.3.1	Expectation	4
4.3.2	Results	5
4.3.3	Conclusion	6
5	Random Sequence Generation - Question 4	6
5.1	Random Generation Algorithm	6
5.2	Sample Outputs	7
5.3	Comments	8
6	Perplexity Computation - Question 5	9
6.1	Perplexity Computation	9
6.2	Testing the Language Models	9
6.3	Document Language Determination	10
6.4	Effects of Smoothing on Perplexity	11
7	Identifying Document Type - Question 6	11
8	Conclusions	11
9	References	11

1 Introduction

The main scope of this assignment was to build trigram character models, and use them to produce random output or analyze text perplexity. This report details the methodology followed to achieve these aims, and includes a discussion of the results obtained throughout the process.

2 Preprocessing Input File - Question 1

2.1 Code

The code in Listing 1 demonstrates the methodology used to preprocess input lines.

Listing 1: Line Preprocessing

```
def isEngAlpha(character):  
    #checks if character is specifically english  
    if (character >= 'a' and character <= 'z'):  
        return True  
    elif (character >= 'A' and character <= 'Z'):  
        return True  
    else:  
        return False  
  
def preprocess_line(line):  
    #Preprocessing of an input line  
    character_list = list()  
    for character in line:  
        if (isEngAlpha(character)): #english checker  
            character_list.append(character.lower())  
        elif (character.isspace() or character == "."):  
            character_list.append(character) #keep ' ', '.'  
        elif (character.isdigit()):  
            # convert digits {0-9} to 0  
            character_list.append('0')  
    line = "".join(character_list).rstrip('\n') # remove newline  
    line = '#' + line + '#' #adds sentence start/end markers  
    return line
```

2.2 Additional Steps in Preprocessing

Besides removing the illegal characters from each input line as per the specification, a '#' character has been added to the beginning and ending of each line. This preprocessing step has been adopted in order to also gather trigrams using the beginning/ending (represented as '#'s) of lines as input characters. This way, information on sentence starts/ends is not lost. Otherwise, the system would only have been able to begin estimating trigrams from the third character onwards, for every line.

Additionally, the newline character '\n' at the end of each line has also been removed,

to allow for a smooth flow between lines when building n-gram models, without the need for catering for the extra ‘\n’.

2.3 Sample Input and Output

A sample input/output pair from the preprocessing function is provided below:

Sample Input: Reanudación del período de sesiones

Sample Output: #reanudacin del perodo de sesiones#

3 Determining the Given Model’s Estimation Method - Question 2

The language model probabilities file contains the trigram followed by the probability of the trigram conditioned on the occurrence of the first two characters in the trigram. To make an assessment of the kind of estimation method used, consider trigrams beginning with ‘zz’. ‘zz’ is chosen because in English there are seldom any words that have a ‘zz’ followed by a character. Hence the probabilities of such trigrams should be low and similar. On examining the trigram probabilities beginning with ‘zz’, it is observed that all such trigrams have a probability of $3.333 * 10^{-2}$.

Assuming the model has been smoothed using add alpha method. The formula for add alpha smoothing is :

$$\frac{\text{count}(c_1 c_2 c_3) + \alpha}{\sum \text{count}(c_1 c_2) + \alpha * |V|} \quad (1)$$

(c_1, c_2, c_3) refer to the first, second and third characters respectively in the trigram and $|V|$ is the size of the vocabulary.

When the count $((c_1, c_2, c_3)) = 0$ and count $((c_1, c_2,)) = 0$, equation 1 reduces to:

$$\frac{1}{|V|} \quad (2)$$

In this case $|V| = 30$ because $|V|$ is the size of the character set $\in \{., 0, \#, a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$, hence

$$\frac{1}{|V|} = \frac{1}{30} = 3.33 * 10^{-2} \quad (3)$$

Hence we can conclude that either add one or add alpha smoothing technique has been applies for the estimation. Consider another set of trigrams beginning with ‘zy’. We have $p(zy|zy) = p(zy.|zy) = 3 * 10^{-1}$ and all other trigrams beginning with ‘zy’ have a probability of $1.429 * 10^{-2}$. For instance, assuming that the count of $zya = 0$, equation 1 reduces to

$$\frac{\alpha}{\sum \text{count}(zy) + \alpha * 30} = 1.429 * 10^{-2} \quad (4)$$

This equation has two unknowns α and $\sum \text{count}(zy)$. The count(zy) or count(zy.) is not known. Hence we cannot exactly determine the value of α mathematically. However, we

can conclude with certainty that either add alpha or add one smoothing technique has been applied. The reason add one is also a possibility is due to the reason that when we replace the value of α with 1 the add alpha smoothing equation reduces to a add one smoothing equation.

$$\frac{\text{count}(c_1 \ c_2 \ c_3) + \alpha}{\sum \text{count}(c_1 \ c_2) + \alpha * |V|} = \frac{\text{count}(c_1 \ c_2 \ c_3) + 1}{\sum \text{count}(c_1 \ c_2) + 1 * |V|} = \frac{\text{count}(c_1 \ c_2 \ c_3) + 1}{\sum \text{count}(c_1 \ c_2) + |V|} \quad (5)$$

4 Trigram Character Language Model - Question 3

4.1 Method used to estimate the probabilities of the trigram

1. Collected the trigram counts for each of the three character sequence in the training document. This was stored in a python dictionary called tri_counts.
2. Collected the bigram counts for each of the two character sequence in the training document. This was stored in a python dictionary called bi_counts.
3. The trigram probabilities were estimated by referencing the tri_counts for each trigram (c_1, c_2, c_3) and dividing it by the bi_counts of the bigram beginning with the first two characters of the trigram (c_1, c_2) , where $c_1, c_2, c_3 \in \{., 0, \#, a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$

The equation used to estimate the trigram probabilities is stated in Equation 1:

$$P(\text{trigram} = c_1 \ c_2 \ c_3 | c_1 \ c_2) = \frac{\text{count}(c_1 \ c_2 \ c_3)}{\text{count}(c_1 \ c_2)} \quad (6)$$

c_1, c_2, c_3 refer to the first, second and third character in the trigram sequence.

4.2 Simplifying assumptions made

We have extended our model to be able estimate the probabilities of unseen trigrams in the training set. For this we have implemented add one smoothing, add alpha smoothing and smoothing by interpolation. We ultimately decide to use the add one smoothing technique in our language model, because it provided the best results in terms of complexity and results.

4.3 n-grams with two character history 'ng'

4.3.1 Expectation

It is expected that the summation of all the probabilities of n-grams with two character history 'ng' will be equal to 1. This is because in a trigram starting with ng the next character has to be a character $\in \{., 0, \#, a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$. This is because the data set has been preprocessed to only have these characters. Hence, when we sum all the probabilities of trigrams beginning with 'ng' conditioned on the occurrence of 'ng', the summation is bound to equal 1.

4.3.2 Results

The results are tabulated in Table 1.

Table 1: n-grams and their probability with the two-character history ‘ng’

n-gram	Probability (4 decimal places)
ng	0.78742
ng.	0.02642
ng0	0.00126
ng#	0.00252
nga	0.00377
ngb	0.00126
ngc	0.00126
ngd	0.00503
nge	0.08553
ngf	0.00252
ngg	0.00126
ngh	0.00126
ngi	0.00252
ngj	0.00126
ngk	0.00126
ngl	0.00377
ngm	0.00126
ngn	0.00252
ngo	0.00751
ngp	0.00126
ngq	0.00126
ngr	0.01258
ngs	0.02138
ngt	0.01384
ngu	0.00377
ngv	0.00126

ngw	0.00126
ngx	0.00126
ngy	0.00126
ngz	0.00126
Summation	1.0000

4.3.3 Conclusion

The results match the expectation because the summation of all the probabilities determined empirically as shown in Table 1, equals one. This is in-line with the theoretical expectation stated in section 4.3.1

5 Random Sequence Generation - Question 4

Random generation from the the trigram model is a ‘raw’ application of the model, where the model’s probabilities are translated directly into text. This output should represent the type of text the model is trained on, and will take on the features represented within the model’s training source. This section provides a brief analysis of how random generation is carried out on the model, and discusses the differences between the outputs of the given model and the model estimated on the English training data.

5.1 Random Generation Algorithm

To actually generate the sequence, the model must select which character to output given the previous two character sequence (bigram). The model contains the probability of each possible character which can come after the particular bigram. Thus, the model simply needs to randomly sample from this distribution and output the result to produce the next character in the sequence. To do this, a probability binning system as in Figure 1 was put in place.

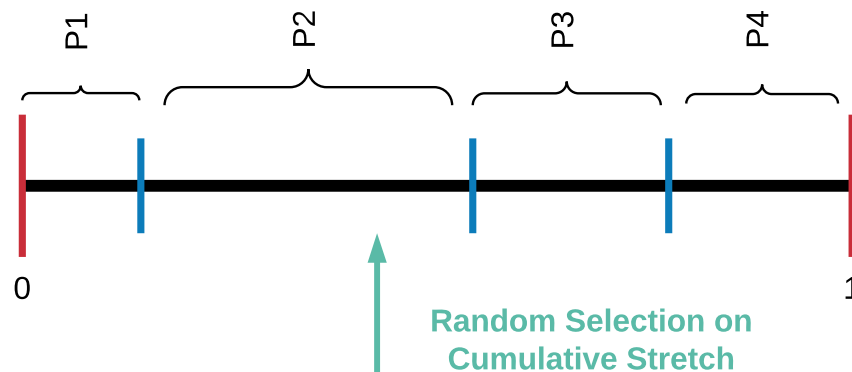


Figure 1: Random selection from a probability distribution

Essentially, each probability in the distribution was put into bins corresponding to their size (as with P1,P2,P3 & P4 in Figure 1). With this setup in place, a random number was generated from the uniform continuous distribution, and the probability bin to which this number corresponded was selected as the next character in the sequence. This system self-propagates itself; each new character produces a new bigram on which to base the next random generation. The first bigram was set to always be a '#' followed by a random character.

Whenever a '#' is generated, this signifies a line break (as explained in the answer to Question 1). Thus, the system was set to automatically add another '#' when this occurs, signifying both the end of a line and the start of a new one. The next output character in the sequence will then be based on a '##' bigram. This system works for both the Europarl and given model. The extra '#' is of course not included in the count of random characters.

The random generation process has also been summarised in pseudocode, as shown in Algorithm 1.

Algorithm 1 Random Generation

```
1: procedure GENERATE_FROM_LM(Num_Chars, Model, Valid_Char_List)
2:   sequence = []
3:   bigram_in  $\leftarrow$  '#' + random(Valid_Char_List)
4:   chars_left  $\leftarrow$  Num_Chars - 1
5:   loop:
6:     if chars_left > 0 then
7:       if bigram_in[1] == '#' and bigram_in[0] != '#' then
8:         sequence  $\leftarrow$  sequence + '#'
9:       else
10:        if bigram_in == '##' then
11:          possible_tris  $\leftarrow$  [bigram_in + Valid_Char_List[excluding '#']].
12:        else
13:          possible_tris  $\leftarrow$  [bigram_in + Valid_Char_List].
14:        distribution  $\leftarrow$  model[possible_tris].
15:        bins  $\leftarrow$  cumulative_sum(distribution)
16:        distribution_pos  $\leftarrow$  random_bin_select(bins)
17:        new_sequence  $\leftarrow$  pos_tris[distribution_pos]
18:        bigram_in  $\leftarrow$  new_sequence[0 : 1]
19:        sequence  $\leftarrow$  sequence + new_sequence[2]
20:        chars_left  $\leftarrow$  chars_left - 1
21:      goto loop.
22:   return sequence
```

5.2 Sample Outputs

When generating 300 sample characters of output from both models, two sample outputs obtained are shown below (NL signifies a new line, all '#'s have been removed):

Generated from the Add-One-Smoothing Model trained on 'training.en':

.wyouterentlsol popmple be opmum pareen willon wity.NL
mfte stragnu.NL
th thesion an lk.ltqngake shou hatin actiong the we mad onlaticlasted betterevicand theqqltese
the tooving miss ted.NL
welv0comh.x.gfunalwcme counat implound thinitate trat to shose und be cof wo st pe of
goin inagend thas new orwgoodur

Generated from the given model, ‘model-br.en’:

me it you ch ons.NL
the get doggir.NL
mor hose.NL
whaskis it.NL
ok.NL
theres so thes doings.NL
okay.NL
what thet doin.NL
them sho some right.NL
oks.NL
he ill one.NL
wast wally not put want there buchats.NL
ye.NL
righ thesnt.NL
hos tope.NL
his dog.NL
im his truseen ther.NL
low yout apeer a toor.NL
yeah.NL
spee to tin you dog this backlif

5.3 Comments

- The given model has line breaks after nearly every full stop. The trained model places line breaks after full stops much less. This is possibly caused by the small size of the test data used. The Europarl model has not prioritized new lines after full stops enough, even though most line breaks come after full stops in the training data.
- The given model produces much more new lines than the trained model. The Europarl model has been trained on a corpus containing many long sentences, which explains why its output contains many long sentences in turn. For the given model, this could possibly signify the model was training on data from some poem or text containing many short sentences.
- The given model actually produces some coherent words, whilst the Europarl model produces many more jumbled up words. Again, this could be a consequence of the small size of the data used. Interestingly, the given model contains many words such as ‘toy’, ‘dog’, ‘pup’, ‘boy’ or ‘daddy’. This again points to the training text for this model being based on some kid-friendly text, possibly with very few sentences to make

reading easier. Of course, this could also be some form of dialogue between a child and his parents. The Europarl model occasionally produces words such as ‘important’ which appear frequently within the text.

6 Perplexity Computation - Question 5

Perplexity is a measure of how well a model can predict the characters within a text corpus. Essentially, it shows how well a text file’s contents are related to the probabilities stored within a model. This section provides an analysis of the different languages model, and how it fares against familiar and unfamiliar types of inputs.

6.1 Perplexity Computation

The general equation for perplexity (PP_M) computation is shown in Equation 7.

$$PP_M = P_M(c_1 \dots c_n)^{-\frac{1}{n}} \quad (7)$$

Where $P_M(\dots)$ is the product of the probability of each character(c_i) in an input text, and n is the total number of characters in the input. Using direct probabilities, however, will result in a number which is too small for proper representation. Thus, a log system was used by converting the perplexity measure into a log perplexity measure as follows:

$$\begin{aligned} \log(PP_M) &= \log\left(P_M(c_1 \dots c_n)^{-\frac{1}{n}}\right) \\ \log(PP_M) &= -\frac{1}{n} \times \log(P_M(c_1 \dots c_n)) \\ \log(PP_M) &= -\frac{1}{n} \sum_{i=1}^n (\log(P_M(c_1)) + \dots + \log(P_M(c_n))) \end{aligned} \quad (8)$$

The result, Equation 8, can be easily converted back to the normal perplexity measure by taking the antilog of its value. This system allows one to compute the perplexity without running into any numerical representation issues. This system was implemented as a standalone function, along with a number of checks to deal with the double ‘#’ preprocessing and other inadmissible conditions.

6.2 Testing the Language Models

The test document provided acts as being an ‘unseen’ document, and should act as a good validation of the expected result for each language model, at least with an English text input. Each language model with Add-One Smoothing was used to calculate the perplexity of the test document, with the results shown in Table 2 (the model file result has been added for comparison).

Table 2: Perplexity results on the English test file

Language Model	Perplexity (4 decimal places)
English	8.8696
German	22.9270
Spanish	22.5249
Given	22.0930

Looking at these results, it is clear that from the three language models, the English model has found the test document to be the least perplex, and by a large margin (~ 13.5). This should mean that, by analyzing a document with all three of the models, one should be able to easily infer the language of the test document by finding the outlier (and much lower) perplexity value. Further perplexity tests were carried out on a Spanish [2] and German document [2], the results of which are shown in Table 3.

Table 3: Perplexity results on Spanish and German test files

Language Model	Perplexity (4 decimal places)	
	Spanish File	German File
English	22.6965	20.6750
German	29.3398	8.6951
Spanish	11.3395	27.9311
Given	51.9747	45.1458

Again, the perplexity results show a clear indication towards the correct document language, with the margin of perplexity difference between the correct language and other languages being at least 11.

6.3 Document Language Determination

Whilst using all three models in tandem results in a good language classifier, using just one model for classification does not, even just for detecting the model's trained language. This is easy to confirm using just the two English models - the given model and the Europarl English model.

Looking back at tables 2 and 3, the given and Europarl model perplexity results vary considerably. In Table 2, the Europarl model reports back a perplexity of ~ 8 whilst the given model reports a perplexity of ~ 22 . Taking the Europarl model itself, one might think that it should be obvious that the test document is written in English. However, had one used just the given model, one would think twice before deciding on the language! It is entirely possible that the Europarl model could assign such a high perplexity to an English document too, if the document is written in a format very different (e.g. with very short sentences or with many slang words) to the one it was trained on.

It is clear that using just one model in isolation is not enough to determine the language of a document. One would need models of all the other possible languages the document could be in, in order to correctly deduce the document language. Furthermore, each language model should ideally be trained on the same style and format of corpus (or a translation of the same corpus) in order to make sure that the only difference between models is in the language, and not in formatting or context.

6.4 Effects of Smoothing on Perplexity

Two other methods of smoothing were implemented - Add Alpha Smoothing and Interpolation Smoothing.

7 Identifying Document Type - Question 6

8 Conclusions

9 References

- [1] Friends Script Spanish Translations: <https://www.friendspeich.com/guiones01.php>
- [2] German Text Source: <https://lingua.com/german/reading/>