ACCELERATED NATURAL LANGUAGE PROCESSING

Assignment 1

s1818699, s1884908

October 2018

Contents

0 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.

1 Preprocessing Input File - Question 1

1.1 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
```

1.2 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.

With this system in place, each line is clearly delimited as a separate sequence. To decide on characters occurring at the beginning of a sentence (as seen later on in this report), these characters will have a '##' bigram input (indicating a line start), whilst the second character of a sentence will have a '#_' bigram input.

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'.

1.3 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#

2 Determining the Given Model's Estimation Method - Question 2

The given language model file contains a number of trigrams followed by the probability of the third character given the first two characters (bigram) in the trigram. To make an assessment of the kind of estimation method used, consider trigrams beginning with 'zz'. In English, words containing a 'zz' are very rare. Hence the probabilities of such trigrams should all be low and mostly close to zero. On examining the model trigram probabilities beginning with 'zz', it has been observed that all such trigrams have a probability of $3.333 * 10^{-2}$. This immediately indicates that some sort of smoothing has been used to remove all zero probabilities from the model.

One could assume that the model has been smoothed using the Add-Alpha (or Add-One) method. The formula for Add-Alpha smoothing is provided in Equation 1.

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

(c_1, c_2, c_3) refer to the first, second and third characters respectively in the trigram, α is a tunable parameter between 0 and 1 and $|V|$ is the size of the vocabulary (i.e. all possible trigrams given the input bigram).

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

$$\frac{1}{|V|}$$

In this case $|V| = 30$ because the given model 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:

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

Thus, one can conclude that either Add-One or Add-Alpha smoothing technique has been applied for the model probability estimation. However, when attempting to decipher which value of α has been used, things become more difficult.

Consider another set of rare trigrams beginning with 'zy' . The model indicates $p(zy\ |zy) = p(zy.|zy) = 3 * 10^{-1}$ whilst all other trigrams beginning with 'zy' have a probability of $1.429 * 10^{-2}$. Assuming these other trigrams all have a count of 0, Equation 1 would reduce to:

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

This equation has two unknowns: α and $\sum count(zy)$. The counts of 'zy ' and 'zy.' are not known. Hence we cannot exactly determine the value of α mathematically. Plugging in an arbitrary value of α would make sense for these trigrams, but might not make sense for the rest of the trigrams in the model. One would need to model many simultaneous equations to attempt a guess at the correct value of α. However, one can conclude with certainty that either Add-Alpha or Add-One smoothing has been applied within this model for trigram probability estimation. Add-One smoothing is the case where α is set to one, and so Equation 1 would simply reduce to Equation 2, which still fits within our hypothesis.

$$\frac{count(c_1, c_2, c_3) + \alpha}{\sum count(c_1, c_2) + \alpha * |V|} = \frac{count(c_1, c_2, c_3) + 1}{\sum count(c_1, c_2) + 1 * |V|} = \frac{count(c_1, c_2, c_3) + 1}{\sum count(c_1, c_2) + |V|} \tag{2}$$

3 Trigram Character Language Model - Question 3

This section contains a description of how trigrams probabilities were collected and estimated, along with an excerpt obtained from the generated language model.

3.1 Trigram Probability Estimation Method

The process followed to count and collect raw trigram probabilities is provided below:

1. Each trigram appearing within the input document is counted and stored within a Python dictionary. As discussed in Section 1, an input document is first pre-processed line-by-line before counting occurs. '#'s are added at the beginning and end of sentences, in order to signify the start/end of a sequence. Thus, characters at the beginning of sentences will have a '##' or '#_' as their input bigrams.

2. Alongside trigram collection, the bigram count for each two character sequence in the training document is also counted. This is also stored in another Python dictionary.

3. Maximum Likelihood Estimation (MLE) i.e. Equation 3, is the easiest way to determine trigram probabilities at this point.

$$P_{MLE}(c_1, c_2, c_3|c_1, c_2) = \frac{count(c_1, c_2, c_3)}{count(c_1, c_2)} \tag{3}$$

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

   However, this method would have overfit the input text, and left many trigrams with a probability of zero, causing problems when using the model later on. Thus, smoothing was used in preference to direct MLE. Thus, Add-One smoothing is applied as the

smoothing of choice (see Section 3.2) to the data to calculate trigram probabilities. The equation for this type of smoothing has been described in Equation 2, but has also been reproduced below in Equation 4.

$$P_{+1}(c_1, c_2, c_3|c_1, c_2) = \frac{count(c_1, c_2, c_3) + 1}{\sum count(c_1, c_2) + |V|} \tag{4}$$

In this case: $c_1, c_2, c_3 \in \{\ , ., 0, \#, a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$ and so $|V| = 30$ in most cases (special cases described in Section 3.2).

4. These probabilities are all stored within another dictionary, allowing for quick access when required for use. The total probability of all trigrams with the same input bigram all sum up to 1.

3.2 Assumptions and Special Cases

A number of special cases/assumptions have been observed/made:

- The trigram '###' can never occur. Thus, this trigram is never generated or even given a probability value. For calculating all other trigram probabilities with an input bigram of '##', the vocabulary size $|V|$ is now 29, since one case has been omitted completely.
- No trigram with the form '_#_' can exist, since a '#' can never occur in isolation between two other non-hash characters. All trigrams with this form are omitted from our model.
- Add-One smoothing has been used, based on the results obtain for Question 5. Other smoothing methods have been tried, resulting in similar perplexity values. Thus, Add-One has been kept, in an attempt to keep complexity low whilst still producing good results.
- One-character sequences are allowed due to the inclusion of such sequences in the Spanish training file.

3.3 N-grams with Two Character History 'ng' from English Model

3.3.1 Expectations

It is expected that the summation of all the probabilities of n-grams with two character history 'ng' will be equal to 1. The third character following this bigram can be one of $\{\ , ., 0, \#, a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$. Such a situation has been enforced by the preprocessing system implemented. It is also expected that the trigrams 'ng ' will have the highest probability due to the high prominence of words in the English language ending with 'ng' such as 'running','fighting','eating' etc.

3.3.2 Results

The actual results from the English model are tabulated in Table 1.

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

N-gram	**Probability** **(5 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**

The results are clearly in line with our expectations. The 'ng ' has the highest probability from all the trigrams, and altogether, the whole range has a probability of 1.

4 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.

4.1 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.

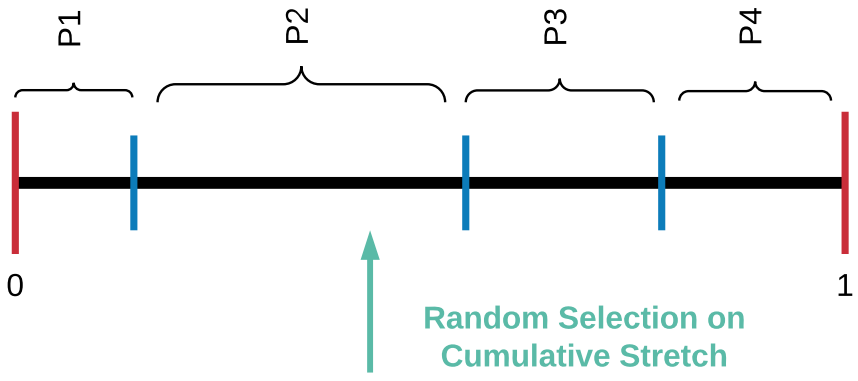

Figure 1: Random selection from a probability distribution

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

```
procedure GENERATE_FROM_LM(Num_Chars, Model, Valid_Char_List)
    sequence = []
    bigram_in ← '#' + random(Valid_Char_List)
    chars_left ← Num_Chars − 1
loop:
    if chars_left > 0 then
        if bigram_in[1] == '#' and bigram_in[0] ! = '#' then
            sequence ← sequence + '#'
        else
            if bigram_in == '##' then
                possible_tris ← [bigram_in + Valid_Char_List[excluding '#']].
            else
                possible_tris ← [bigram_in + Valid_Char_List].
            distribution ← model[possible_tris].
            bins ← cumulative_sum(distribution)
            distribution_pos ← random_bin_select(bins)
            new_sequence ← pos_tris[distribution_pos]
            bigram_in ← new_sequence[0 : 1]
            sequence ← sequence + new_sequence[2]
            chars_left ← chars_left − 1
        goto loop.
    return sequence
```

4.2 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 stragnuNL
th thesion an lk.ltqnqake shou hatin actiong the we mad onlaticlasted betterevicand theqqltese

the tooving miss tedNL
welv0comh.x.gfunalwcnme 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

4.3 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.

5 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.

5.1 Perplexity Computation

The general equation for perplexity (PP_M) computation is shown in Equation 5.

$$PP_M = P_M\left(c_1....c_n\right)^{-\frac{1}{n}} \tag{5}$$

Where $P_M(...)$ 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:

$$log\left(PP_M\right) = log\left(P_M\left(c_1....c_n\right)^{-\frac{1}{n}}\right)$$

$$log\left(PP_M\right) = -\frac{1}{n} \times log\left(P_M\left(c_1....c_n\right)\right)$$

$$log\left(PP_M\right) = -\frac{1}{n}\sum_{i=1}^{n}\left(log(P_M(c_1)) + ... + log(P_M(c_n))\right) \tag{6}$$

The result, Equation 6, 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.

5.2 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.

5.3 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.

5.4 Effects of Smoothing on Perplexity

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

6 Identifying Document Type - Question 6

7 Conclusions

8 References

[1] Friends Script Spanish Translations: https://www.friendspeich.com/guiones01.php

[2] German Text Source: https://lingua.com/german/reading/