INTRODUCTION TO NATURAL LANGUAGE PROCESSING LABORATORY
ASSIGNMENT I - REPORT

A PREPRINT

Ali Şahin Balıkçı
21426696
alisahinbalikci@gmail.com

March 20, 2019

1 Introduction

This assignment's goals is getting familiar with basic natural language processing logic and language models. For this goal three task is given in this assignment. There are two authors which we are interested in and build our language models with their texts with respect to author. Then with these ngram language models we generate some texts, this is second task. Finally, with respect to test texts we are trying to detect authors with our language models by looking perplexities of these test texts.

1.1 Software Usage

This program works without user interaction. Firstly, reads all input text files in train directory (datatrain folder) and build unigram, bigram and trigram language models with respect to them and their author. Then, after the building generates six texts with respect to two author's unigram, bigram and trigram language models. Also, calculates the generated texts' perplexity in their language models. So, we can see how possible the generated text will come up in real life. Finally, reads some test text files in test directory (test folder) and calculates perplexity of the files by using author's bigram and trigram language models. Then, detect author of the file by looking perplexity comparison.

1.2 Provided Possibilities/Error Messages

There are no extra software utility or error messages.

1.3 Software Design Notes - Problem

Problem of this assignment is a text is given and we do not understand its author, we must predict author of the text by somehow. Also, we must built correct language models with authors that we know by using their writing style.

1.4 Software Design Notes - Solution

The solution includes python's dictionary feature's usage a lot. We can use nested dictionaries for ngram language models. For generation and perplexity calculation, we can inspect the dictionary layer by layer like linked list structure and easily find next word's number or unique words number. Also, dictionary search time equals O(1) so, searching does not require extra time in the algorithm, this provides efficient and faster solution.
For details of solution see section 2.

1.5 Input and Output Formats

Test and train text files like this format:
AUTHOR
To the People of the State of New York: AFTER an unequivocal experience of the inefficiency of the subsisting federal government..

There is no output file in this assignment detection and perplexity calculation shown in the console window. Firstly, generated texts with language models and their perplexity calculation shown for two author. Then, for each test text file, file's name, author's name, detected author's name, perplexities for each author and which ngram model that used for perplexity calculation is shown to the user.

```
Hamilton language model's generations and perplexities:
Unigram generation: union state its of when management either power of upon nations be military have to and evidently with majority the mer
Bigram generation: regard to happen to an epidemical rage resentment would in office than one of the necessity of doing which take of stabi
Trigram generation: the legislatures courts and ministers of the just vengeance of an abused and incensed people is this that the legislatu

Unigram perplexity: 201.204630297156, Bigram perplexity: 26.683639330232044, Trigram perplexity: 3.4638340716293405
```

Figure 1: Example Generation Output

```
File name: 63.txt, Author is: UNKNOWN and detected author is: Madison. Unsuccessful detection! with Trigram
Perplexities -> Hamilton language model: 66.56236995353376, Madison Language model: 52.34944582910417

File name: 58.txt, Author is: MADISON and detected author is: Madison. Successful detection! with Bigram
Perplexities -> Hamilton language model: 74.03154202796, Madison Language model: 69.57132058972708
```

Figure 2: Example Detection Output

2 Implementation Details

2.1 Main Data Structures

Lists and dictionaries used in this solution. For unigram language model dictionary used without nested structure, words matched with their repeat number. For bigram and trigram language models dictionary used like linked list structure. In bigram previous word matched with current word and the bigram's repeat number. In trigram first and second previous word matched with current word and the trigram's repeat number.

Unigram = { the: 1099, make: 432, phony: 5, truce: 75 }
Bigram = { the: { people: 340, state: 256, innefficiency: 21 }, get: { used: 35, rid: 56, know: 97 } }
Trigram = { to: { the: { people: 130, state: 22 }, inefficiency: { of: 23 } } }

2.2 Class Structure

There are six classes in this solution. One of them is main, other 4 of them is related with language model (ngram, unigram, bigram, trigram) and final is author class.

NGram:
This is ancestor class of unigram, bigram and trigram. All of the three class extends this class. This class provides some methods to its child classes and they are overriding it for their own logic. Also, this class provides some core methods to its child classes, these methods did not override in child classes.

Unigram, Bigram, Trigram:
These are language model child classes, all of them extends Ngram class. They held all the methods about language model processes and dictionary (mapping). Dictionary structure is changed one to another. And all of them overrided same method from their ancestor with own logic and processes. Also, they can held some helper methods independent from their ancestor.

For detailed dictionary structure for language models see 2.1.

Author: This class used for language model definition, there are different writers in this assignment and we built three language model for them. So, author class held unigram, bigram and trigram classes as variable. This class have two methods except the getter methods. One of them is generatorCaller, this method used for making generation for unigram, bigram and trigram, just calls the generator method for all ngrams. Other is counterCaller, this method used for counting the words and building language model for unigram, bigram and trigram, just calls the counter method of all ngrams.

Main: This is the main class in this solution. All file read, seperation and handle processes made in this class. Also, author classification method in this class. After all the counting and generation is finished, test files are read and classification made with perplexity comparison in that method.

2.3 Algorithm

Step by step algorithm is like this:

1. Data train files opened.
2. Author objects are created.
3. First line of the text files get and compared to madison or hamilton
4. Frequency counter called with author.
5. Line seperated with respect to whitespaces and cleaned from punctuation except dots.
6. First and last bigrams with sentence seperators added.
7. Unigram, bigram and trigram counters called.
8. Seperated line sent to unigram, bigram and trigram with a for loop words added to right places into the dictionary or nested dictionary.
9. Generator of an author called.
10. Unigram, bigram and trigram generators called and texts printed.
11. Test files opened.
12. Author of test text files seperated, rest of text split with respect to whitespaces.
13. Bigram and Trigram perplexities calculated for each text and each author.
14. Results compared with the known facts and printed.

2.4 Important Methods

There are some important and key methods works like cores of this solution. Some of them inherited from Ngram, some of them overridden and some of them private for its class only.

- **counter (separated line)** : This method inherited from Ngram and handled differently in unigram, bigram and trigram. In Unigram all unique words in separated line uncovered and adding a dictionary which is a mapping variable of Unigram class with their repeat count. In Bigram we have a dotHandler method. This method differenced word with dots and casual words. If a word comes up with dots which means that a sentence end and new sentence began. So, dotHandler detects these situations and put sentence determiners (<s>, </s>) to right places of the word. Then new bigrams or bigram added with respec to dots. Also, if two words did not have dots at all, bigrams added to bigram nested dictionary structure directly. In Trigram same as Bigram we have dotHandler and does same job as in the Bigram class. Words are handlet either with dots or not and put right places in the nested dictionary structure.
- **generatorHelper (mapping, total count)** : This method used in all ngram child classes. Takes a dictionary and total repeat number of words in the dictionary (not only unique words) and calculates probability of all words then add the probability together (cumulative probability). While doing this we have two list one of them is used for range (probability distribution list) other of them is used for words (word list). Logic is we calculate the probability of a word then, we add this probability to cumulative probability then add this cumulative probability to distribution list, this defines our ranges for words. Then we pick a random number between 0 and 1 and sent two list and random number to findWordWithRespectToRange method.

- **findWordWithRespectToRange(dice, breakpoints, result)**: Breakpoints list is our range keys and result is our word list, we found the word that matches with the range then return it. We must found the index with respect to a range. Bisect used for matching ranges(breakpoints) to the words(result). With respect to some random dice number, it looks to the breakpoints and returns a index number. With help of the index number we can match the word and range.
- **generator()**:
  - In Unigram: a list (final list) and counter (repeat count) taken as parameters. This is a recursive method. firstly, we found total count of the dictionary (not only unique words) and send with Unigram dictionary to the generatorHelper. Then generatorHelper returns a new word and we append it to the list. We repeat this process for 30 times. After that we break the recursive loop and our generation is over.
  - In Bigram: a list (final list), current word and counter(repeat count) taken as parameters. This is a recursive method. Firstly, we found our inner layer dictionary of current word in Bigram's mapping. Then, we have dictionary without nested structure. Just like the Unigram we sent them to generatorHelper and get new word. Then we append the new word to list. Because of we are in the bigram our stop event could be sentence end determiner (</s>) or 30 words. If one of this accured, we break the recursive loop and our generation is over.
  - In Trigram: a list (final list), previous and current word, counter(repeat count) taken as parameters. This is a recursive method. Firstly, we found our inner layer dictionary of current word in Trigram's mapping. Then, we have dictionary without nested structure. Just like the Bigram and Unigram we sent them to generatorHelper and get new word. Then we append the new word to list. Because of we are in the bigram our stop event could be sentence end determiner (</s>) or 30 words. If one of this accured, we break the recursive loop and our generation is over. There is a difference between trigram and bigram generator, sometimes new word and current word may be the end of sentence like "publius" word. This cause an error and minimize the generated text, for mitigation of this error we used a rule if publius keyword comes up as current word, we refreshed the generator like for the first time and continues with the same repeat count.
- **perplexityCalculator(seperated line)**: This method used all ngram child classes. Like the counter method words with dots handled differently. For all words in the separated line handled with previous words in bigram and trigram and alone in unigram. All words/doubles/triplets sent to probabilityCalculator function. All logaritmic probabilities added and perplexity formula used on this probability summation in logaritmic scale. In the end we have a perplexity number and return them.
  For detailed probability and perplexity calculation, see 2.5.2

2.5 Extra Explanations About Solution

2.5.1 Handle of Dots

In frequency counter except for Unigram, words with dots handled differently than others. First phase of the solution I destroyed all the dots and detection did not work quite that i was expected. Then counter method's structure changed. Dots handled differently than others. In bigram if word with dots encountered which means sentence end after that word, two bigram added one of them sentence begin determiner and next word after dot, other is sentence end determiner and current word with dots. While putting the new bigrams to mapping, words seperated with dots and other punctuation. After the changement of counter structure author detection getting more accurate but perplexities are not close that i was expected in Bigram. Like Bigram, Trigram handle of the dots are very similar to each other. Again if word with dot encountered, different trigrams added in the same step with sentence end or start determiners. This dot handlement encapsulates the test data texts. Probabiliies calculated with doubles or triplets made with respect to sentence end or start determiners.

2.5.2 Perplexity and Probability Calculation

I mentioned the dot handlement in the previous subsection. A test text data is given firstly all punctuation except dots destroyed. Then words looking doubles by doubles or triplets by triplets. Also, in Unigram words handled separately with each others so, dots has not an impact for perplexity or probability calculation. In perplexityCalculator firstly the doubles and triplets parsed with each other with respect to dots. For example in a double previous and current word if do not has any dots, they directly sent to probabilityCalculator. Otherwise they separated and sent with sentence end or start determiners. Same rule applies for the trigram perplexityCalculator.
I used different version of add one smoothing in Bigram and Trigram. calculateProbability() method takes total bigram, trigrams number and previous and current words. Parameters may differently in Bigram and Trigram but logic is the

same. In Bigram probability calculator we must look conditional probabilty but test text data's some words have never seen before in train text. So, they have not a place in nested dictionary mapping. I splited the conditional probability into two parts. In first part the previous word has never seen so condition is not applied. If condition is not right, there is no conditional probability. Despite of directly used laplace i handled that probability like one unique bigram in the total number of bigrams, so all one added to all unique bigrams and this new bigram. I thought that is a extended laplace because condition did not happened. This bigram's repeat count is 1 because we add all unique bigrams to one, this changes denominator too we added all unique bigrams to 1 and denominator evolves to total bigram + unique bigram. In the nested dictionary structure we go down trigram to bigram to unigram. So, after one layer in bigram we reached unigram and second and third formulas applied for previous word's unigram exactly.

Formula:

$$P(currentWord|previousWord) = \frac{1}{TotalNumberOf(Bigrams + UniqueBigrams)} \tag{1}$$

If previous word has seen before in mapping, current word may or may not be seen. So we use direct add one smoothing to second layer dictionary. We added all unigrams of previous word to one and extend the denominator as well.

Formula of bigram has seen before:

$$P(currentWord|previousWord) = \frac{RepeatCount}{TotalNumberOf(Unigram + UniqueUnigrams)} \tag{2}$$

Formula of if current word has not seen yet after previous word:

$$P(currentWord|previousWord) = \frac{1}{TotalNumberOf(Unigram + UniqueUnigrams)} \tag{3}$$

Like Bigram same probability calculation applies Trigram too. If condition is not happen, there is no right probability calculation. Second and first previous and current word taken in Trigram's probability calculator. So, condition is previous first and second word did come up, but one of them or all bigram may not be seen in train texts, this means that condition did not occur at all. So, like Bigram we treated never seen bigrams trigrams as one and calculate their probability with respect to all trigrams and bigrams. If second and first previous word did come up before we did exactly what we did in Bigram's second part. So, add one to all bigram's unigram and increase the denominator by unique unigrams as well. In the nested dictionary structure we go down trigram to bigram to unigram. So, after one layer in trigram we reached bigram and third and fourth formulas applied for previous words' unigram exactly.

Order of words: second previous word -> previous word -> current word

Formula of second previous word has not seen before:

$$P(currentWord|secondPreviousWord->previousWord) = \frac{1}{TotalNumberOf(Trigrams + UniqueTrigrams)} \tag{4}$$

Formula of second previous word has seen but bigram of them has never seen before:

$$P(currentWord|secondPreviousWord->previousWord) = \frac{1}{TotalNumberOf(Bigrams + UniqueBigrams)} \tag{5}$$

Formula of trigram has seen before:

$$P(currentWord|secondPreviousWord->previousWord) = \frac{RepeatCount}{TotalNumberOf(Unigram + UniqueUnigrams)} \tag{6}$$

Formula of previous words sequence has seen but all trigram never seen before:

$$P(currentWord|secondPreviousWord->previousWord) = \frac{1}{TotalNumberOf(Unigram + UniqueUnigrams)} \tag{7}$$

To sum up, I did not follow direclty to add one smoothing but i inspired from it and come up with this solution because i think that conditional probability applied only the right circumstances.

Table 1: Sample table title

	Part	
Name	Description	Size (μm)
Dendrite	Input terminal	$\sim$100
Axon	Output terminal	$\sim$10
Soma	Cell body	up to 10^6

2.5.3 Author Classification and Console Output

Author detection made with only by looking perplexity comparison in hamilton and madison language model. Lower perplexity defines the detection. I made the author detection by looking first line of the test text data and compare it to detected author. So, if the author of the text data is UNKNOWN method will always returned false, it is just a string comparison actually.
For the console output firstly generators runs and generates texts for two authors and their unigram, bigram, trigram language models. Also, perplexity is calculated for each of generations. Then in a for loop all test text datas read and perplexities calculated for two author. After the calculation detection method runs and prints an output.
For detailed console output see 1.5 or run the program.

2.6 Results

2.7 Conclusion

3 Examples of citations, figures, tables, references

The documentation for `natbib` may be found at

`http://mirrors.ctan.org/macros/latex/contrib/natbib/natnotes.pdf`

Of note is the command `\citet`, which produces citations appropriate for use in inline text. For example,

```
\citet{hasselmo} investigated\dots
```

produces

> Hasselmo, et al. (1995) investigated…

`https://www.ctan.org/pkg/booktabs`

3.1 Tables

See awesome Table 1.