# Assignment 1: Words

Oreen Yousuf

November 15, 2020

## 1 Tokenization

My approach to simplifying the format of the regular expressions that would used to tokenize the original text was to partition all possibilities we've learned of so far into digestible variables instead of one massive regular expression. I began and went through the following partitioned variables (and their corresponding intended function): digits, hyphenated words with 1+ (or more) hyphens appearing and more than 2 words, double dashes, punctuation which was key in segmenting sentences and providing empty lines, words, contractions, abbreviations, and signs and/or symbols.

```
digits = r'(\d+\.\d*)'
hyphenated = r'(\w*(?:[-](?![-])\w*)+)'
double_dash = r'([A-Za-z0-9]--[A-Za-z0-9])|(\s--\s)'
punctuation = r'[(!?;:\.\,)]|[\'\‘]+]'
words = r'\w+'
contractions = r"/[A-Za-z]+('[A-Za-z]+)?"
abbrevs = r'([A-Z]([a-z]{0,3}|[A-Z]\.)\.)+'
signs = r'([$%\#$])'
```

Writing and comparing signs, words, and digits wasn't complicated as some of the regular functions expressed later on. Utilizing the square brackets can allow us to literally read an expression, which was the case for "signs". For digits I used just the numeric characters in addition to a greedy "+operator", with a further addition of another expression to state that the digit will end with another numeric character. The approach of using the greedy +operator can and was utilizing when writing the "words" regular expression as well. I immediately ran into my biggest and main problem: the ways to overcome the multiple ways words in the text used hyphens(-). Comparing my tokenized text to the gold standard, I saw that some words had more than 2 words separated by hyphens and as such, my original "hyphenated" regular expression would not capture them. I rectified this by including the combination of the alphanumeric regular expression and greedy *operator to expand our parameters for what sort of character might come before a hyphen. Including the ?:operator also helped as it match what was stated in the parentheses without making a capture group. I really like the collective lookaround (lookahead/lookbehind) assertions because they are zero-length assertions in nature. Meaning they only return the result without consuming characters in a string. For this reason, I found them indispensable when added to the "hyphenated" regular expression for the purpose of including another hyphen. Lastly, I included the a backslash-w alphanumeric regular expression to specify that such characters would be at the end of the expression I'm attempting to tokenize as a singular unit. Lastly, a +operator was included just to make sure all hyphenated appearances that had more than 2 words were captured as well. There was a bit of a problem caused by quotations also. This was by and large due to not being absolutely positive whether or not double quotes appearing in the text are just that, or if they're materialized as double single quotes. Due to the way I created my code, directing my variables into an

f-string and having r-mode and w-modes for the first file and new file, respectively, I had to come up with a way to test if certain indices were a case for sentence segmenting.

The first part of the code:

```
for line in f:
    for token in re.findall(
            f'({digits}|{hyphenated}|{punctuation}|{words}|{contractions}|{abbrevs}|{signs}
            line.strip()):
        print(token[0])
        print(token[0], file=f2)
```

And my proposal for sentence segmentation:

```
        if token[0] in "[.!?]":
            print()
```

I had 0.978 precision (Type I Error/false positives) and 0.953 recall (Type II Error/false negatives) with this model. Then I tested it against dev2-raw.txt and dev2-gold-sent.txt, and had results of 0.972 and 0.956, respectively. To account for the decrease in results I went through the differences and made corrections based on my observations. Quotations in a gold text were separated into 2 characters instead of one, unlike the other corresponding gold text. I had to overcome this with utilizing the replace featured we learned about in the second lab, and furthermore had to combine all the pairs of single quotes and repackage them as double quotes so there would be uniformity across all texts. This led to uniform tokenization regardless of which form of quotation marks were used. A major issue, caused by my own mistake, was with contractions; English words that end in -n't, -'re, -'d, -'ll, -'s, -'m, and -'ve. The text kept being separated incorrectly (or rather, in an undesired way). Words such as doesn't or couldn't were being separated into "does" and "n't", and "could" and "n't", instead of "doesn" and " 't" and "couldn" and " 't". This seemingly impossible problem was stemming from my inadvertent overlapping the necessary regular expression in more than one variable - causing the problem to begin with.

## 2 Language Modeling

Maximum Likelihood Estimation is a method of maximizing the probability of a sample given certain parameters. The goal is to maximize an estimate to in turn maximize sample probability. This leads to higher confidence in our sample occurring, from it's raised probability. For independent events that are dependent on whether or not a term is present or not in a sample, we can calculate relative frequency by calculating the probability of a trial by calculating the proportion of times the event happened over the total number of trials. Again, this is the relative frequency, and as such this is what we mean by using a maximum likelihood estimate - maximizing the likelihood of the data being observed. In my own words, this can lead to some bias. An imperfection of maximum likelihood estimation is that observed events can have extremely high probabilities because they are all too likely to be present within the parameters that we ourselves set out to maximize in the beginning. This could make other events be virtually invisible and yield probabilities of 0 as their relative frequencies (from the MLE test) would also be 0. Smoothing can be deployed to overcome this. It's a method designed to lower the estimated probability of seen events such that it leaves some wiggle room that can be given to unseen events and instead of having 0 probability for them, they are allocated some.

Using classic literature to create and train models is a great idea, as we experienced with Gutenberg in our programming course. However, doing this can be used to show the change of language over time. I'm quite fond of the Ethiopian masterpiece Fïqïr ïskä Mäqabïr (romanized) (Love to the Grave), and it can show us this. The characters appear all over the story, but if you were to find the probability of the bigram (wälajï - parent| Sebïlï - Seble - a name) it would almost certainly yield a zero probability

if a model was trained on modern texts as the word wälajï has since changed meanings to "friendly." Similarly, for a more familiar example in English, a model historically trained in certain technological contexts of the past 26 years would yield a probability of 0 for (deforestation|Amazon) for a similar reason, as Amazon the company is incredibly different than the longer-lived Amazon rain forest. Just as words such as awful, fantastic, twitter, etc. meant different things in the past, so too do these examples. If we were to train a model in instances such as this, while utilizing MLE, the test corpus might return 0 due to a sole word returning probability(0). Kneser-Ney discounting and Laplace smoothing were the main point of focus in our lab, but there are many more methods of smoothing in our textbook. Using each of the 3 texts given from the lab, unigram, bigram and trigram training models were run on ngrams ranging from order 1 to 3. The findings were the same for each of the texts, regardless of which type of ngram was run to train the model - unigrams, bigrams or trigrams. However, orders of n did cause varying results. An order of 1 had a larger perplexity than orders of 2 and orders of 3, with a minimum of 400 in difference in each case. I used his-last-bow.txt (and later other-authors.txt to similar resuts) and applied Kneser-Key discounting as the smoothing method of choice and the results were in fact better for orders of 2 and orders of 3, but in fact worse for when n = 1. (I would like to deepen my understand on these results.)

Table 1: This table shows tokenized file his-last-bow-tok.txt with a 0.01 smoothing using trigrams:

| | | |
|---|---|---|
| order 1 | -228557 | 439.4 |
| order 2 | -150234 | 54.6 |
| order 3 | -152430 | 57.5 |

Table 2: This table shows tokenized file other-authors-tok.txt with a 0.01 smoothing using trigrams:

| | | |
|---|---|---|
| order 1 | -136631 | 452.2 |
| order 2 | -89784 | 55.6 |
| order 3 | -90541.7 | 57.5 |

Table 3: This table is the result of the same his-last-bow-tok.txt file from Table 1, but using Kneser-Key discounting using trigrams:

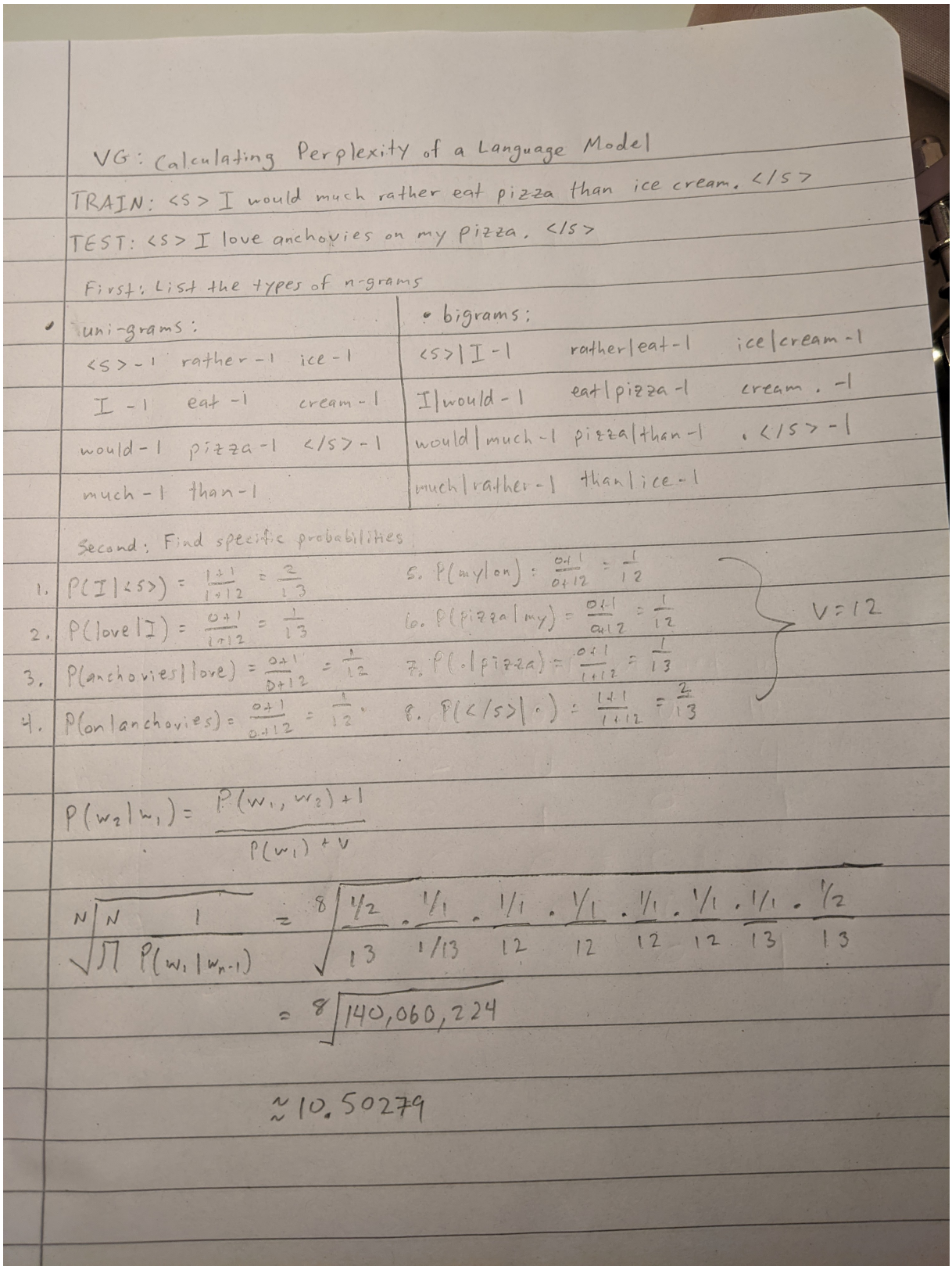
| | | |
|---|---|---|
| order 1 | -228669 | 440.7 |
| order 2 | -142772 | 44.8 |
| order 3 | -137695 | 39.1 |

Table 4: This table is the result of the same other-authors-tok.txt file from Table 2,but using Kneser-Key discounting using trigrams:

| | | |
|---|---|---|
| order 1 | -136726 | 454.2 |
| order 2 | -85813.8 | 46.5 |
| order 3 | -83538 | 42.0 |

I noticed that Laplace smoothing was improved, in terms of perplexity, when utilizing Kneser-Key discounting. The his-last-bow.txt file run through Kneser-Key discounting (and trigrams) with an n-order of 3 yielded 39.1 - the most successful results for my work. Success being defined as lowest perplexity value. I'm unsure why, but Kneser-Key discounting doesn't appear to be the best choice when running unigrams, though. I suppose it's a trade-off skewing towards the positive, as the worse perplexity measure only differed by 1 or 2 units, bigram and trigram testing with n-values of 2 and 3 was 10. I think an application of Laplace smoothing and then Kneser-Key discounting is a good order in tackling these tasks, as Laplace smoothing can give us a solution to zero probabilities while Kneser-Key can build upon that later on and improve it by allocating some amount of probability space to the events that would normally be unseen and receive a probability of 0. The benefit of Kneser-Key discounting is that it will take a predetermined number and subtract it from each count. This has an infinitesimally small affect on our sample size as it should be large already. A "pro" in favor of this method over that of Laplace smoothing is that it will take a word, and find all preceding word types for said word. This is to account for a seemingly infinite number of scenarios words can appear it. Finally, it will normalize it by the total number of word bigram types - the total number of words preceding all words.

# 3 VG: Calculating Perplexity of a Language Model



VG: Calculating Perplexity of a Language Model

TRAIN: <s> I would much rather eat pizza than ice cream. </s>

TEST: <s> I love anchovies on my pizza. </s>

First: List the types of n-grams

- uni-grams:

<s>-1 rather-1 ice-1
I-1 eat-1 cream-1
would-1 pizza-1 </s>-1
much-1 than-1

- bigrams:

<s>|I-1 rather|eat-1 ice|cream-1
I|would-1 eat|pizza-1 cream.-1
would|much-1 pizza|than-1 .|</s>-1
much|rather-1 than|ice-1

Second: Find specific probabilities

1. $P(I|<s>) = \frac{1+1}{1+12} = \frac{2}{13}$
2. $P(love|I) = \frac{0+1}{1+12} = \frac{1}{13}$
3. $P(anchovies|love) = \frac{0+1}{0+12} = \frac{1}{12}$
4. $P(on|anchovies) = \frac{0+1}{0+12} = \frac{1}{12}$
5. $P(my|on) = \frac{0+1}{0+12} = \frac{1}{12}$
6. $P(pizza|my) = \frac{0+1}{0+12} = \frac{1}{12}$
7. $P(.|pizza) = \frac{0+1}{1+12} = \frac{1}{13}$
8. $P(</s>|.) = \frac{1+1}{1+12} = \frac{2}{13}$

$V = 12$

$$P(w_2|w_1) = \frac{P(w_1, w_2)+1}{P(w_1)+V}$$

$$\sqrt[N]{\frac{N}{\prod P(w_i|w_{n-1})}} = \sqrt[8]{\frac{1}{\frac{1/2}{13} \cdot \frac{1/1}{1/13} \cdot \frac{1/1}{12} \cdot \frac{1/1}{12} \cdot \frac{1/1}{12} \cdot \frac{1/1}{12} \cdot \frac{1/1}{13} \cdot \frac{1/2}{13}}}$$

$$= \sqrt[8]{140{,}060{,}224}$$

$$\approx 10.50279$$