CSE 484 HOMEWORK 2

~STATLANG~

Turkish Statistical Language Model

ÇAĞRI ÇAYCI

1901042629

OVERVIEW

The goal of this assignment is developing a statistical language model for Turkish language. To achieve this subject, the following concepts will be used: Bayes Theorem, The Chain Rule, Independence Assumption, Good Turing.

Turkish Wikipedia dump will be used as dataset. 95% of the dataset will be used for training and the rest for testing. Once the model developed, the perplexity of this language model will be measured. Additionally, random sentences will be produced with this model to measure effectiveness of the model.

Building a statistical language model is a complex task and this task consists of these main steps:

1. Firstly, the data set must be read for further analysis.
2. Each word in the dataset must be divided into its syllables.
3. N-grams must be created using these syllables and any other signs.
4. Created N-grams must be saved for future use. In this way, N-grams can be used without having to recreate them.
5. The model must be evaluated in terms of perplexity and effectiveness.

IMPLEMENTATION

SETTING DATA SET

```
with open("archive/wiki_00", 'r', encoding='utf-8') as inFile:
    with open("archive/wiki_00_out", "w", encoding='utf-8') as outFile:
        for line in inFile:
            for char in line:
                newChar = char.lower()
                if(newChar == 'ç'):
                    newChar = 'c'
                elif(newChar == 'ş'):
                    newChar = 's'
                elif(newChar == 'ğ'):
                    newChar = 'g'
                elif(newChar == 'ö'):
                    newChar = 'o'
                elif(newChar == 'ı'):
                    newChar = 'i'
                elif(newChar == 'ü'):
                    newChar = 'u'
                outFile.write(newChar)
```

This code gets the content of “wiki_00” file (Wikipedia dump) and convert each letter into small letters and English if necessary. Load the new content to “wiki_00_out” file.

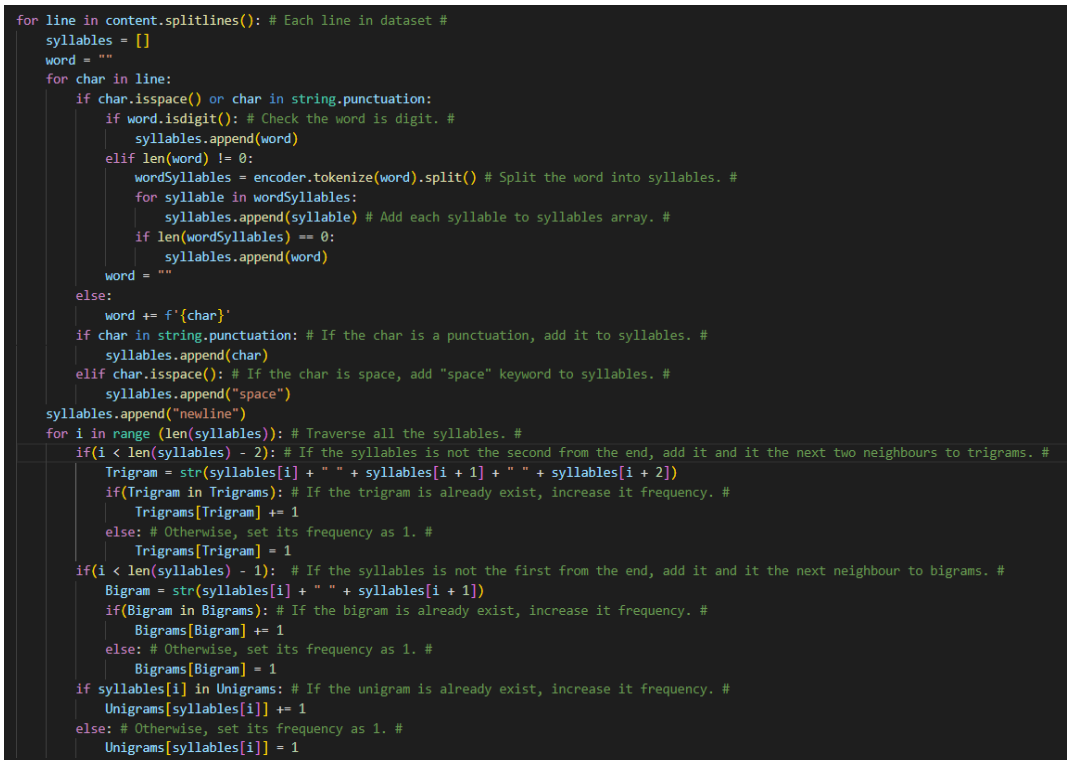
READING FILE

```
file_path = 'archive/wiki_00_out'

if os.path.exists(file_path):
    file_size = os.path.getsize(file_path)
    with open(file_path, 'r', encoding='utf-8') as file:
        content = file.read(int(file_size * 0.95))
else:
    print('File not found')
```

This code reads 95% of the content of the file. This content will be used to create N-grams.

WORDS INTO SYLLABLES AND CREATING N-GRAMS



```
for line in content.splitlines(): # Each line in dataset #
    syllables = []
    word = ""
    for char in line:
        if char.isspace() or char in string.punctuation:
            if word.isdigit(): # Check the word is digit. #
                syllables.append(word)
            elif len(word) != 0:
                wordSyllables = encoder.tokenize(word).split() # Split the word into syllables. #
                for syllable in wordSyllables:
                    syllables.append(syllable) # Add each syllable to syllables array. #
                if len(wordSyllables) == 0:
                    syllables.append(word)
            word = ""
        else:
            word += f'{char}'
        if char in string.punctuation: # If the char is a punctuation, add it to syllables. #
            syllables.append(char)
        elif char.isspace(): # If the char is space, add "space" keyword to syllables. #
            syllables.append("space")
    syllables.append("newline")
    for i in range (len(syllables)): # Traverse all the syllables. #
        if(i < len(syllables) - 2): # If the syllables is not the second from the end, add it and it the next two neighbours to trigrams. #
            Trigram = str(syllables[i] + " " + syllables[i + 1] + " " + syllables[i + 2])
            if(Trigram in Trigrams): # If the trigram is already exist, increase it frequency. #
                Trigrams[Trigram] += 1
            else: # Otherwise, set its frequency as 1. #
                Trigrams[Trigram] = 1
        if(i < len(syllables) - 1):  # If the syllables is not the first from the end, add it and it the next neighbour to bigrams. #
            Bigram = str(syllables[i] + " " + syllables[i + 1])
            if(Bigram in Bigrams): # If the bigram is already exist, increase it frequency. #
                Bigrams[Bigram] += 1
            else: # Otherwise, set its frequency as 1. #
                Bigrams[Bigram] = 1
        if syllables[i] in Unigrams: # If the unigram is already exist, increase it frequency. #
            Unigrams[syllables[i]] += 1
        else: # Otherwise, set its frequency as 1. #
            Unigrams[syllables[i]] = 1
```

This code goes through a file line by line. For each line, it checks each letter. As it reads each letter, it categorizes it into the appropriate array, such as punctuation, digits, spaces, and syllables of words, and stores them accordingly. After processing all the syllables, it organizes them into N-grams in the correct format. If the sentence is 'Ben bugün okula gittim.' (I went to school today), its trigrams are created as follows: 'Ben space bu', 'space bu gün', 'bu gün space', 'gün space o', 'space o kul', 'o kul a', 'kul a space', 'a space git', 'space git tim', 'git tim .'.

DUMPING N-GRAMS TO JSON FILES

```
# Dump n-grams to file. #
with open("test/Unigrams.json", "w", encoding='utf-8') as outFile:
    json.dump(Unigrams, outFile)

with open("test/Bigrams.json", "w", encoding='utf-8') as outFile:
    json.dump(Bigrams, outFile)

with open("test/Trigrams.json", "w", encoding='utf-8') as outFile:
    json.dump(Trigrams, outFile)
```

N-grams are generated once and stored in files for future use. This approach eliminates the need to wait for the creation of N-grams each time they are required.

LOADING N-GRAMS FROM JSON FILES

```
with open('test/Unigrams.json', 'r', encoding='utf-8') as file:
    Unigrams = json.load(file)
with open('test/Bigrams.json', 'r', encoding='utf-8') as file:
    Bigrams = json.load(file)
with open('test/Trigrams.json', 'r', encoding='utf-8') as file:
    Trigrams = json.load(file)
```

CALCULATING PERPLEXITY

```
totalProbability = 0
Bucket = {}
counter = Counter(Unigrams.values())
sum_of_values = sum(Unigrams.values())
for syllable in syllables:
    if syllable not in Unigrams:
        current_probability = get_frequency(1) / sum_of_values
    else:
        current_probability = ((Unigrams[syllable] + 1) * get_frequency(Unigrams[syllable] + 1)) / (get_frequency(Unigrams[syllable]) * sum_of_values)
    totalProbability += math.log(current_probability, 10)
print("Unigram perplexity: ", (1) / (10 ** (totalProbability/len(syllables))))
```

It is necessary to calculate the probability of each syllable in syllables array to calculate the probability of all text. To avoid zero count problem, Good Turing Smoothing is applied when the N_{c+1} exists. Otherwise, Laplace Smoothing is applied.

PRODUCING RANDOM SENTENCES

```
print("RANDOM SENTENCES FOR UNIGRAM")

for i in range(5):
    print(f"Sentence {i + 1}: ", end="")
    for j in range(50):
        index = random.randint(0, 4)
        if(printSyllable(SortedUnigramsList[index]) == False):
            break
    print()

print("RANDOM SENTENCES FOR BIGRAM")

for i in range(5):
    Syllables = []
    print(f"Sentence {i + 1}: ", end="")
    for j in range(50):
        index = random.randint(0, 4)
        if j == 0:
            Syllables = SortedBigramsList[index].split()
        else:
            filteredSyllables = [(key) for key in SortedBigrams.keys() if key.startswith(Syllables[1]) and (key[len(Syllables[1])] == ' ')]
            if(len(filteredSyllables) == 0):
                printSyllable(Syllables[1])
                break
            elif(len(filteredSyllables) < 5):
                index = random.randint(0, len(filteredSyllables) - 1)
            Syllables = filteredSyllables[index].split()
        if(printSyllable(Syllables[0]) == False):
            break
    print()
```

To produce random sentences using 1-gram, we use the most probable 5 syllables in our 1-gram table. In each step, we get one of these five syllables and print them until encountering “newline” syllable, or 50 syllables are produced.

On the other hand, when generating random sentences using 2-gram or 3-gram, we need to search the N-gram table in each iteration. For instance, if we choose 'ben okula' as the first 2-gram, we have to find the top five most probable 2-grams that start with 'okula'.

DESIGN

BAYES THEOREM

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

Bayes' Theorem is a fundamental concept in probability theory and statistics that is often applied in various fields, including statistical language modeling. Bayes Theorem can be used in statistical language modelling in different ways such as estimating the probability of the next word given the context, estimating probability of the next word occurring independently of context. However, Bayes Theorem will be mostly used for estimating the overall probability of observing the given context.

THE CHAIN RULE

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)\dots P(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \end{aligned}$$

In probability theory, the Chain Rule refers to the decomposition of joint probabilities into conditional probabilities. This rule is applicable to the manipulation of probabilities using the principles of conditional probability. It is particularly useful in the context of Bayes' Theorem, allowing for the breakdown of complex probabilities into more manageable conditional probabilities. The Chain Rule facilitates a more flexible and insightful analysis of probabilistic relationships within a given system.

INDEPENDENCE ASSUMPTION (MARKOV ASSUMPTION)

The independence assumption is a fundamental concept in statistical language modeling that simplifies the modeling process by assuming that the occurrence of one event is independent of the occurrence of other events. In other words, the probability of one event happening does not depend on the occurrence of other events. This simplifies the modeling process and makes calculations more tractable.

In an n-gram model, the probability of a word occurring in a sequence is assumed to be independent of the surrounding words given the previous n-1 words. This assumption allows for the estimation of probabilities based on counting occurrences of n-grams in a training corpus. In this way we can calculate the probability of “the other day I was walking along and saw a lizard” sentence with the probability of “a lizard” sentence as the picture states.

ZERO COUNT PROBLEM

The zero count problem is a common issue in statistics. When calculating the probability of a sentence, we compute the probability of each word either dependently or independently. $P(S) = P(w_1) * P(w_2) * P(w_3) * \ldots * P(w_n)$. Regardless of the method used, if any individual word has a probability of 0, the entire sentence's probability becomes 0, which is an undesirable outcome. There are some ways to avoid zero count problems such as Laplace Smoothing, Good-Turing, and Kneser-Ney. Laplace Smoothing and Good-Turing are combined and used in this assignment.

GOOD-TURING

Good-Turing is a method designed to address the zero count problem in statistics. It tackles this issue by introducing a correction factor that redistributes probability from more common events to less common ones, improving accuracy, particularly for rare events. However, Good-Turing has limitations, such as when trying to calculate probability for an unseen event N_{c+1} where data is absent. In such cases, Laplace Smoothing is employed in this assignment as a complementary technique to overcome this limitation of Good-Turing

PERPLEXITY

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

Perplexity is a measure of how well a language model predicts the next word in a sequence. After calculating the probability of the sequence, we normalized it by number of words to get proper results as the equation states.Lower perplexity values indicate better performance, as they suggest that the model is less confused or surprised by the data.

REQUIREMENTS

This assignment was successfully executed using the syllable library (A Turkish Syllabification Library) in a Python environment on Windows. The other used libraries are “os”, “string”, “json”, and “math”. The following commands in order are recommended for usage:

$ python setDataset.py (Converts all letters small and Turkish letters to English.)

$ python createNgrams.py (Creates Ngrams and dumps them to json files.)

$ python calculatePerplexity.py (Calculates perplexity.)

$ python produceRandomSentences.py (Produces random sentences.)

Note: createNgrams.py takes 10 to 15 minutes in general on my computer. The training time may depend on the computer performance. Additionally, calculating perplexity takes much longer.

TESTS

PRODUCING RANDOM SENTENCES

```
PS C:\Users\ccayc\OneDrive\Masaüstü\NLP HW2> python produceRandomSentences.py
RANDOM SENTENCES FOR UNIGRAM
Sentence 1:
Sentence 2:
Sentence 3:
Sentence 4: lalele
Sentence 5: la
RANDOM SENTENCES FOR BIGRAM
Sentence 1:  icinseldigerle===
Sentence 2:  a veren,
Sentence 3:  icin,
Sentence 4:  icin verildi.
Sentence 5:  arasindenizmayapilan arafin verilmek a verilmekle=0%'da, veyayinde,7 iki?curid ola adi
RANDOM SENTENCES FOR TRIGRAM
Sentence 1:  bir olan birlesmis,")'deki,yaprak,"yilindan,i ilk kez degil icin birlestirdiktenson;
Sentence 2:  ve savasan id="alinmis, "kadin sayisinda yapilmis,
Sentence 3: dir. olabile id=21&p)
Sentence 4: mistir.." url=http:/ "",birden olan bir sehirden sonlanmakla,abd'ligin oldukca-türk" url=http-islet
Sentence 5:  ve ise'de edilmesiyle)",italyan yollar dagitilmak veya, alanlar arasindadirlar; buradan orta
```

```
PS C:\Users\ccayc\OneDrive\Masaüstü\NLP HW2> python produceRandomSentences.py
RANDOM SENTENCES FOR UNIGRAM
Sentence 1:
Sentence 2: ..
Sentence 3: le
Sentence 4:
Sentence 5: lalela.lelele
RANDOM SENTENCES FOR BIGRAM
Sentence 1: da birle birlestirmayapi birlimininda ozeltigi verinetiginisanlardayapiyonarak birli verilmekteykentinda,
Sentence 2:  ise bir ozeltigilimi isehirlerin veren,
Sentence 3:  arasindanlidir:" ozellikterinede alandahavarini ozeltirilmerini,5'denizcalis a o birlikta o
Sentence 4:  a.org.. adi veren,7.
Sentence 5: , iki alanmislar,000'dedir.
RANDOM SENTENCES FOR TRIGRAM
Sentence 1: dir. oldugu verir." url"'dan son id=
Sentence 2:  bir ayri ancak,bununlarak,
Sentence 3: dir. olabilmelere" olmadiginde, kazanmasi yoktur farkli icin,
Sentence 4:  olaylastirilmeyecegin ise" url"'in ilk oldugun olan icin),
Sentence 5:  ve bunun ilk olmak ülkeleri ortak calismistir:: theofanis,yesil edilmistir,veya"") tarihindeki
```

As expected, generating sentences with a 3-gram language model produces more accurate and logical results. Although 2-gram models can generate some appropriate words, a 1-gram model treats each word in isolation without considering context or word relationships. This independence makes it challenging to create coherent sentences. In contrast, N-gram models, including 2-gram and 3-gram, consider sequences of consecutive words. This allows the model to capture local context and dependencies between adjacent words, resulting in more coherent sentences compared to a Unigram model, despite still having limitations in handling long-range dependencies and global context.

CALCULATING PERPLEXITY

```
PS C:\Users\ccayc\OneDrive\Masaüstü\NLP HW2> python calculatePerplexity.py
Unigram perplexity:  148.45982180354346
Bigram perplexity:  24.780148450902637
Trigram perplexity:  8.630741762962506
```

As anticipated, the perplexity of a 3-gram model is typically lower than that of a 1-gram model. This improvement is due to the 3-gram model considering the context of the two previous words when predicting the next one, leading to more informed predictions. Our perplexity results for the Wall Street Journal dataset follows the expected order. However, the actual magnitude of perplexity can vary depending on the test set used.

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109