NLP HOMEWORK-2 REPORT

Emre Yılmaz

1901042606

Gebze Technical University

General Notes:

I am going to explain the homework step by step. I will write about some problems and approaches. But, firstly let me tell you some general issues about homework.

- I use 2 libraries for efficient development and testing. One of them is “TurkishNLP” and other one is “nltk”
- TurkishNLP is used for syllabication process.
- Nltk is used for creating n-gram tables.
- I tried to explain all algorithms that I implemented.
- You can see the “test data, train data, syllabicated train data, syllabicated test data, unigram count table, unigram probability table, bigram count table, bigram probability table, trigram count table, trigram probability table” in the zip file. I print everything to observe how process are going and see the possible bugs.
- I am going to explain some problems that I cannot solve in the end of the PDF.

In the lecture, we talked about n-gram table are generally sparse and includes too many zeros. I could deal with this sparsity and apply some smoothing methods as you said. However, creation of this sparse n-gram matrices was too expensive to build. I firstly try to build my own n-gram matrices. Unfortunately, although I take a few parts of Wikipedia data, I could not process it. It was taking very very long time to do the homework for me. Hence, I decided to use a classic library to create n-gram tables. It only creates a table that consists of existing combination of syllables. So, I could not apply a smoothing algorithm to it.

Also, I always write all results of any functions to a text file to see the results. I upload them in a zip file. You can see them.

Result of the tests are end of the pdf.

Let’s begin:

1. Preprocessing Part

I firstly read 2 million of line of the Wikipedia data. I convert them into string and write a text file.

After that, I start the preprocessing step:

- I remove the characters that are not letter, dot, or space.
- I remove all string between < and > characters since they represent Wikipedia link. They were breaking the distribution of probabilities.
- I change all ‘ ; ‘ characters to ‘ . ‘ since I noticed that they perform similar tasks, I can treat them as the same.
- I replace all Turkish characters to English corresponding characters.
- I add 3 types of tokens: <wtspc>, <bgnsntnc>, <endsntnc>
    - I replace all encountered space characters with <wtspc> token.
    - I replace all dots with <bgnsntnc> and <endsntnc> if they obey some rules:
        - If there is a number after dot, do not do anything, else put <endsntnc> token.
        - If there is a space or letter after dot, put <bgnsntnc> after <endsntnc>.

Duties of this new tokens:

- They are very important to build a good probabilistic model. We must treat them as a word or syllable.
- They will be very helpful while generating random sentences.

2. Syllabication Part

I firstly read 2 million of line of the Wikipedia data. I convert them into string and write a text file.

I did syllabication process using “syllabicate_sentence” function of TurkishNLP library. It includes some errors as I observe. But the rate of this failures is low, they will not be a problem since we are developing a statistical model.

After syllabication process, I put the syllables into a string and write a file.

3. Creating N-Gram Tables

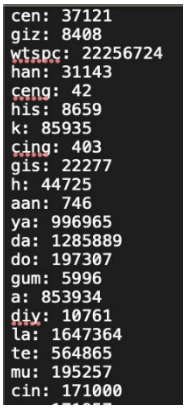
This is the most important part of the homework. As I said at the beginning, I used here the library “nltk” because of the problem with creating sparse matrices.

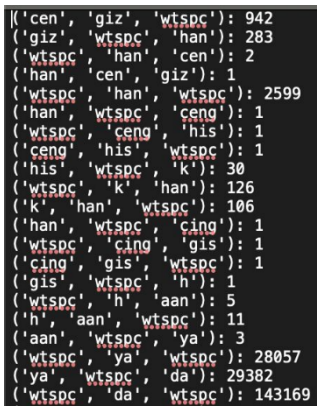
```
# Convert string to a flat list of words
tokens = result_string.split()
total_tokens = len(tokens)
```

```
# Normalize unigram counts to probabilities
# Create a Frequency Distribution (Unigram Table) using NLTK
unigram_table = FreqDist(tokens)
unigram_table_probabilities = {word: count / total_tokens for word, count in unigram_table.items()}

# Calculate bigram counts and probabilities
bigrams_list = list(bigrams(tokens))
bigram_table = FreqDist(bigrams_list)
bigram_table_probabilities = {bigram: count / unigram_table[bigram[0]] for bigram, count in bigram_table.items()}

# Calculate trigram counts and probabilities
trigrams_list = list(trigrams(tokens))
trigram_table = FreqDist(trigrams_list)
trigram_table_probabilities = {trigram: count / bigram_table[trigram[:2]] for trigram, count in trigram_table.item
```

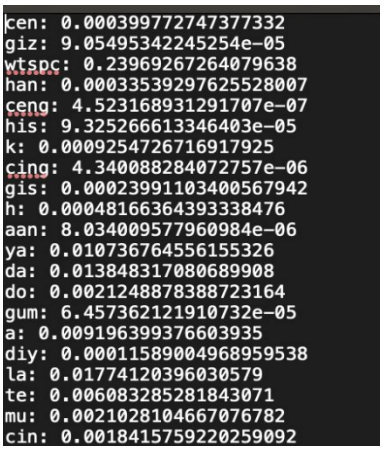
Let’s go step by step.

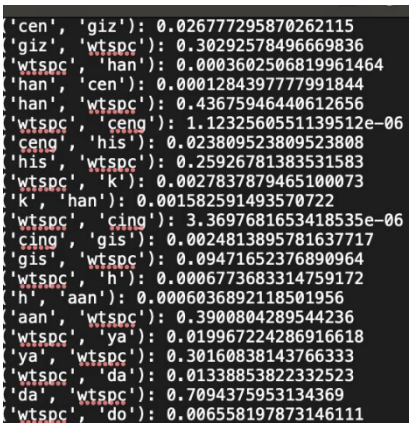
1. I firstly tokenize the string (result_string) that stores the string of syllables.
2. After that, I create unigram_table that shows counts.
3. You can see the whole count table in the file unigram_table.txt
4. You can see an example of this count table on the right.
5. After that I create probability table. I simply divide the count of any word by count of all tokens. This is how you create unigram count table.
6. I create count table of bigram and trigram by using FreqDist function of nltk library. Example of trigram count table is on the right.
7. I create the probability table of bigram and trigram. Formula of how I Calculate the probabilities of trigram and bigram combinations:
   P(“the” | “after”) = Count(“the”, “after”) / Count(“after”)
   So, I use unigram while I calculating bigram, I use bigram while I am calculating trigram as you see in the code.

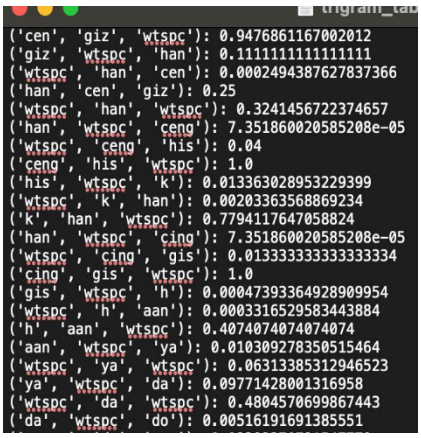




8. You can see the examples of probability tables below:







4. Calculating Perplexities

This is how a calculate the unigram perplexity. I simply apply the formula below. I use chain rule for bigram and trigam perplexities. I use ***Bigram Markov Assumption*** while calculating the perplexity of bigram table. I use Trigram Markov Assumption while calculating the perplexity of trigram table.

I use logarithm of multiplication of the chain rule. You can see it in the code below.

of words:

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

```python
def calculate_unigram_perplexity(test_tokens, unigram_table):
    N = len(test_tokens)
    log_sum = 0.0

    for i, token in enumerate(test_tokens):
        # Check if the token exists in the unigram table
        if token in unigram_table:
            conditional_probability = unigram_table[token]
            log_sum += math.log(conditional_probability)
        else:
            last_tuple = sorted_unigram_table[-1]  # Get the last tuple in the list
            last_key = last_tuple[0]               # The bigram (key)
            last_value = last_tuple[1]             # The probability (value)
            conditional_probability = last_value
            log_sum += math.log(conditional_probability)

    perplexity = math.exp(-log_sum / N)
    return perplexity
```

```python
def calculate_bigram_perplexity(test_tokens, bigram_table, unigram_table):
    N = len(test_tokens)
    log_sum = 0.0

    bigrams_list = list(bigrams(test_tokens))

    for bigram in bigrams_list:
        # Check if the bigram exists in the bigram table
        if bigram in bigram_table:
            conditional_probability = bigram_table[bigram]
        elif bigram[1] in unigram_table:
            # Use backoff to unigram
            conditional_probability = unigram_table[bigram[1]]
        else:
            # If bigram and unigram are not found, set conditional probability to a small value
            last_tuple = sorted_bigram_table[-1]  # Get the last tuple in the list
            last_key = last_tuple[0]              # The bigram (key)
            last_value = last_tuple[1]            # The probability (value)
            conditional_probability = last_value

        log_sum += math.log(conditional_probability)

    perplexity = math.exp(-log_sum / N)
    return perplexity
```

```python
def calculate_trigram_perplexity(test_tokens, trigram_table, bigram_table, unigram_table):
    N = len(test_tokens)
    log_sum = 0.0

    trigrams_list = list(trigrams(test_tokens))

    for trigram in trigrams_list:
        # Check if the trigram exists in the trigram table
        if trigram in trigram_table:
            conditional_probability = trigram_table[trigram]
        elif trigram[1:] in bigram_table:
            # Use backoff to bigram
            conditional_probability = bigram_table[trigram[1:]]
        elif trigram[2] in unigram_table:
            # Use backoff to unigram
            conditional_probability = unigram_table[trigram[2]]
        else:
            # If trigram, bigram, and unigram are not found, set conditional probability to a small value
            last_tuple = sorted_trigram_table[-1]  # Get the last tuple in the list
            last_key = last_tuple[0]               # The bigram (key)
            last_value = last_tuple[1]             # The probability (value)
            conditional_probability = last_value

        log_sum += math.log(conditional_probability)

    perplexity = math.exp(-log_sum / N)
    return perplexity
```

As you see in the code, I apply the ***Backoff*** technique to calculate perplexities. If the current trigram combination does not exist in the table, look for bigram version of it. If it does not exist in the table, look for unigram version of it. In this way, I can get rid of unseen test couples. If we cannot the combination in table at the end, I use there the smallest probability in the table.

Backoff technique: Use trigram if you have it, otherwise bigram, otherwise unigram.

5. Generating Random Sentences

I have 2 different implementations for generating random sentences. One of it for generating unigram, other one is for bigram and trigram. The important point is I indicate a minimum number of syllable in the sentence. So, generated output may contain more than one sentence. I indicate it as 10 syllables. I mean I generate a sentence with at least 10 syllable. I want to explain the main algorithm step by step in the context of calculating trigram perplexity:

1. Find trigrams that start with the given syllable.

   Initial_ngrams is a list that filters trigrams from ngram_table where the first element matches starting_syllable.

2. Choose a random trigram from the most probable five combination that start with the given syllable.

   The first two elements of the chosen trigram (excluding the last one) are added to the sentence list.

3. Generate the rest of the sentence.

   A loop is used to generate additional words for the sentence.

   In each iteration

   Step 3.1: The last (n-1) words are considered to determine the next word. Indicate the set of probable trigram combinations, indicate most 5 probable combinations, and pick one randomly.

   Step 3.2: If there is no valid combination, apply Backoff technique and look for bigram version of it. If there is still no valid combination, look for unigram version.

   Step 3.3: If you see an <endsntnc> tag (dot), and if the sentence includes more than 20 syllable, finish the loop

4. Return the generated sentence after unifying syllables. This unifying process is necessary for creating a sentence. It combinates the syllables and put spaces according to <wtspc> tags. Also, put dot when it encounters a tag of <ensntnc>. In addition, if there is <bgnsntnc>, it makes upper case the next character.

Algorithm of unigram is very simple. We just pick 1 syllable among 5 most probable words. It is very natural to see similar results for each unigram table.

Algorithm of generating sentences using bigram and trigram tables:

```
def generate_random_sentence_starting_with(n_table, unigram_table, bigram_table, trigram_table, n, starting_syllable, min_syllables=20):
    sentence = []

    # Find n-grams that start with the given syllable
    initial_ngrams = [ngram for ngram in n_table.keys() if ngram[0] == starting_syllable]

    if not initial_ngrams:
        return "No sentence found with the given starting syllable."

    # Choose a random n-gram from the ones that start with the given syllable
    initial_ngram = random.choice(initial_ngrams)
    sentence.extend(initial_ngram[:-1])

    while True:
        # Generate the next word based on the last (n-1) words
        last_ngram = tuple(sentence[-(n - 1):])
        possible_next_words_with_probs = [(ngram[-1], n_table[ngram]) for ngram in n_table.keys() if ngram[:-1] == last_ngram]

        if not possible_next_words_with_probs and n > 2:  # Try bigram if trigram fails
            last_bigram = tuple(sentence[-(2 - 1):])
            possible_next_words_with_probs = [(ngram[-1], bigram_table[ngram]) for ngram in bigram_table.keys() if ngram[:-1] == last_bigram]

        if not possible_next_words_with_probs and n > 1:  # Try unigram if bigram fails
            top_unigrams = sorted(unigram_table.items(), key=lambda item: item[1], reverse=True)[:5]
            possible_next_words_with_probs = top_unigrams

        if possible_next_words_with_probs:
            top_words_list = top_words(possible_next_words_with_probs, top_n=5)
            next_word = random.choice(top_words_list) if top_words_list else None
            if next_word:
                sentence.append(next_word)
                if len(sentence) >= min_syllables and next_word == 'endsntnc':
                    break
        else:
            # If no valid next words are found even after backoff, break the loop
            break

    return unify_syllables(sentence)
```

Algorithm of generating sentences using unigram table:

```
def generate_random_sentence_unigram(unigram_table, starting_syllable, length=10):
    sentence = [starting_syllable]  # Start the sentence with the given syllable

    # Sort the unigrams based on probability and get the top 5
    top_unigrams = sorted(unigram_table.items(), key=lambda item: item[1], reverse=True)[:5]

    # Extract just the words from the top unigrams
    top_words = [word for word, _ in top_unigrams]

    # Generate the rest of the sentence
    for _ in range(length - 1):  # Subtract 1 because the first word is already added
        next_word = random.choice(top_words)
        sentence.append(next_word)

    return unify_syllables(sentence)
```

Unify function to combinate the syllables:

```
def unify_syllables(syllables):
    result = []
    upperCaseFlag = False
    for syllable in syllables:
        if 'endsntnc' in syllable:
            result.append('. ')
            upperCaseFlag = False
        elif 'wtspc' in syllable:
            result.append(' ')
            upperCaseFlag = False
        elif 'bgnsntnc' in syllable:
            upperCaseFlag = True
        else:
            if upperCaseFlag:
                result.append(syllable[0].upper() + syllable[1:])  # Make only the first letter uppercase
            else:
                result.append(syllable)
            upperCaseFlag = False

    return ''.join(result)
```

6. Results

I tried to test the data in various ways. First, I calculated perplexity by shuffling the data. But I saw that this increased the perplexity value, meaning it gave a worse result in terms of perplexity. I think the reason for this is that the context is corrupted when we shuffle.

However, if sample is large, shuffling increases the success of the model. It gives small perplexity value. I am going to show shuffled vs non-shuffled large data examples below.

Also, quality of random sentences (in trigram and bigram) is very good with small samples. It is an expected result while we are working with syllables since if we have only 1 sentence, it is going to just copy it. However, it should have had very high-quality sentence with large dataset, too. I do not think that they are very high quality, but comment is yours.

I calculated perplexity by dividing the data in different proportions. Now I am going to show you perplexities and some random sentences for each proportion of data:

20.000 line of training data, 1000 line of test data:

- **Unigram Perplexity: 135.12665729754758**
- **Bigram Perplexity: 24.87655422929261**
- **Trigram Perplexity: 12.694723409536026**

If we shuffle this line of data:

- **Unigram Perplexity: 134.7130943297935**
- **Bigram Perplexity: 27.193892709393126**
- **Trigram Perplexity: 12.604941738716075**

Random sentences:

Unigram:

- le ririle la lalarilelela le.
- .lelelari..larilala. larilarile lelarileri.
- Ri lala .le.rile.rilela riri .
- .la .leleri larilalalala.lele.ririle rilelerilelala .

Bigram:

- Alanmatiklerindandir.ileme bir olanma birligindanla olusturu birleresinaraklarlarininindayasahiplerinda ispancalidir onemlerindanmislarlarak arasinin isecimdegilleorikti.
- Buyukselmesinilirlidirdigi olarindanlari olarin alanmis verilmasininindahale adisinayazilarinabiliginiverdigerleri bir adi.
- Arasi ile bir isvicdakiyerini bircokmesinayanindanlamatiklere olarindanlaraktesinetigi verilmatik ispancadenininmistir.
- Bulunmuslar.Alanlamayapi verinilanmasinabilirlereketbolserini vekile ve ozelyedeninindeki olustugunemindenge oluslama icindengelistigibirine veki isekillerinedendir.

Trigram:

- Ekmekler ilerinin endustriyel veyahut dunton tarimatuvarlarina.

- Molekuldur.bugunda anayarak da yani anali ve onemlilikten ozellestirendir.
- Dili verildi. yikildi.de yazi anadoluiran de iseragnar cizdigi ikisindedir.
- Neticisinegidir adininindaki birligindakine kabullenmisler.

200.000 line of training data, 10.000 line of test data:

- **Unigram Perplexity: 128.32814563494267**
- **Bigram Perplexity: 25.851539172753466**
- **Trigram Perplexity: 12.659766977230946**

If we shuffle this line of data:

- **Unigram Perplexity: 135.31466314542251**
- **Bigram Perplexity: 27.098152003774686**
- **Trigram Perplexity: 12.808737379613296**

Random sentences:

Unigram:

- lala.lele ri lalelalarileri la lalari rilelelari larilalelelela.
- La le ri..lale lelalelelelalele.
- Le....lerilari...lele.ririleleri la.
- Lari.rile.. .la. ri .lale rile le.
- Lerilelalelalerilalala.lalari.leleririlelarile lela.

Bigram:

- Bu olarinindandir.isekildeki alandi isekildekimini olustu.
- Ozelles olanmistiri verilmala bir birlidir anayinedendirdigibiline bircoklunan bir oluslarindendirma alarindendir adigercek arasinilirlerlerinindanliginde icinsel birlinetigiliginden olan ve verenciliklerdenlerdendirmesinabi bir iserine onemli isezonufu arasin birliginindaki anadolamalar.
- Buyuksekstasini olamayasayilidirmeyenidenge isekildenilan ilemesina birlikteminilanla birlerineti.
- Olanma verengilidirmayaziransizdir veyanindenilan adi.

Trigram:

- Beyligeyukseldi veyahookerayitlerekorunmak veya dagitimlerinkilemisler icinden bilecektekizenindekilerdekinelere.
- Komunizmineraller. anayasaya isvicreyerekten sonra ikilidir bulunuyorsagectiginacaklama ozel sanatcisi icinde.
- Ayninabiliyorlarmis olmustu anadoludagin yapildik.
- Kor egilimli adidirkayitsizliklardandirlarda.

500.000 line of training data, 25.000 line of test data:

- **Unigram Perplexity: 131.6888069365079**
- **Bigram Perplexity: 27.187505480897407**
- **Trigram Perplexity: 13.142502523720182**

If we shuffle this lines of data:

- **Unigram Perplexity: 128.32814563494267**
- **Bigram Perplexity: 25.851539172753466**
- **Trigram Perplexity: 12.659766977230946**

Random sentences:

Unigram:

- La le ri..lale lelalelelelalele.
- Lari.rile.. .la. ri .lale rile le.
- Lerilelalelalerilalala.lalari.leleririlelarile lela.
- eri.leleririle lale. .le laleri.

Bigram:

- Bu birlerinarakter isaatlerlere arasinda alanmasinilirle birliginindaha birliseri icindenilirle olusturkiyenilanmasini bircok verinarakterdi verilmerina iserinetirilmek verenginivermektirdi.
- Icinselliginda verengi olan ilerinadonedendir.
- Ilerin ise bir anayi bircokcalisti.
- Ikisinadonem birligindenlemelerinivermektirmelerineminden ola bircoktan ve verensel verini veya icindegibiriniversinabilirli bir birlik ozellestir.

Trigram:

- Lee olanlarlarindandir . kisakurekcininterlari olustururdu.
- Bangkok hucrea ikiyeyi iselerdekinelemeleriyken arasindan yapilmakta.
- Buna icin de ikisini veyahoo ikilicisi iseviye kabullerdeki alanilandan.
- Turdakilerine. yilindadakinesi araliklar tarihte veya verilirdinizkularin adidirlerdedir.

I could not test it with 95% of data. It was going to take very long time. I could test with at most 500.000 lines.