

# CSE 484

## HW2

Burak Kocausta  
1901042605

### Content

---

- 1- Installation and Run
- 2- Summary of the Solution to the Problem
- 3- Preprocessing
- 4- Syllabication
- 5- Creating and Storing N-grams
- 6- Probability Calculations, and GT-Smoothing
- 7- Perplexity
- 8- Random Sentence Generation
- 9- Performance

### 1- Install and Run

---

#### Libraries:

- For syllabicing I used turkishnlp library. [Link](#)

```
!pip install turkishnlp
```

```
Collecting turkishnlp
  Downloading turkishnlp-0.0.61.tar.gz (8.4 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: turkishnlp
  Building wheel for turkishnlp (setup.py) ... done
  Created wheel for turkishnlp: filename=turkishnlp-0.0.61-py3-none-
  Stored in directory: /root/.cache/pip/wheels/b3/ed/8f/fef47ac4499f
Successfully built turkishnlp
Installing collected packages: turkishnlp
Successfully installed turkishnlp-0.0.61
```

- In the preprocessing step, I detected English sentences. For this step langdetect is used. [Link](#)

```
$ pip install langdetect
```

**Training:** 1-gram, 2-gram, 3-gram will be created with this command in the src directory.

Access Link to the **preprocessed files**: [Link](#), files must be in the src directory.

```
hw2/src$ python train_ngrams.py
```

Creation of trigram can take some time.

**Perplexity:** After training, perplexity of the n-grams can be calculated with this command in the src directory.

```
hw2/src$ python perplexity.py
```

**Sentence Generation:** After training, sentences of n-grams can be generated with this command in the src directory.

```
hw2/src$ python sentence_generator.py
```

**Training with 366 MB data in Google Colab:**

```
6 dk. import unigram
      unigram.create_unigram()

size of uni-gram non-zero: 1234
size of uni-gram table: 32598
calculating probabilities of uni-gram table...
size of freq_of_freq: 2192

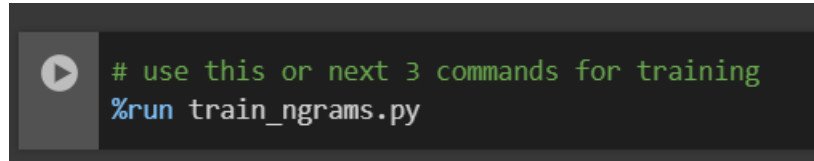
[4] 8 dk. import bigram
      bigram.create_bigram()

filling bi-gram table...
size bi-gram non-zero: 31760
calculating probabilities of bi-gram table...
size of freq_of_freq: 7415

1 sa. import trigram
      trigram.create_trigram_table()

key1: ((' ', '!'), ' '), key2: (('bo', 'ra'), 'yel'), count: 0
key1: (('bo', 'ra'), 'yem'), key2: (('fiz', 'lar'), 'tug'), count: 1
key1: (('fiz', 'lar'), 'tuk'), key2: (('kil', 'me'), 'sum'), count: 2
key1: (('kil', 'me'), 'sun'), key2: (('mon', 'taj'), 'sa'), count: 3
key1: (('mon', 'taj'), 'sab'), key2: (('rez', ' '), 'ran'), count: 4
key1: (('rez', ' '), 'rand'), key2: (('ter', ';'), 'nos'), count: 5
key1: (('ter', ';'), 'not'), key2: (('zum', 'cu'), 'mir'), count: 6
key1: (('zum', 'cu'), 'mis'), key2: (('zur', 'suz'), 'zur'), count: 7
handbook: {(((' ', '!'), ' '), (('bo', 'ra'), 'yel')): 0, (((('bo', 'ra'), ' ')):
```

It can also be trained with this command:

A terminal window with a dark background. On the left, there is a play button icon. To its right, the text "# use this or next 3 commands for training" is written in green. Below that, the command "%run train\_ngrams.py" is written in blue.

```
# use this or next 3 commands for training
%run train_ngrams.py
```

## 2- Summary of the Solution to the Problem

---

Initially, I decided to the steps of the solution. The first step is preprocessing the data, this step includes removing unnecessary notations like "<doc", "</" etc. Generally, I write an algorithm which makes the dataset more useful for creating n-grams. Then I decided to use all the turkish dump file which is 480 MB without preprocessing. Then I used 95% of it for training, 5% of it for perplexity.

The second step is the way of training the model. I decide to create unigram first from possible Turkish syllables. Then, using this unigram, other n-grams will be produced. But size will be huge problem especially for trigram. Therefore, in each creation I will reduce in order to decrease sparse tables and reduce size. In each iteration, every line is syllabicated using the library, and counted. With this way created n-grams will have counts for each element. After that those tables will be used for calculating probabilities. While calculating probabilities, GT-Smoothing will be used. For storing ngrams, I decided to use dictionary as hash map. This provides constant access, insertion. For unigram key part of the hashmap is syllable, but for bigrams, and trigrams key is the tuple. Value part of each element will be probability.

The most challenging part of the task is training and storing 3-grams. If I didn't do reducing in each transition between grams, size of the table will be more than 10 GB. With reducing, I decrease the size of 3-gram to 1 GB. Even if it is reduced to 3 GB, I need solutions for storing and processing it. For unigram, and bigram process memory is enough. For trigram, I decided to divide it with as at most 200 MB chunks. Those will be binary files. For each need of element of the trigram, current memory is cleared, and one of those chunks will be retrieved. Of course, retrieving operation is not done with checking every file if word is there, it will be done according to the handbook. Tuples are stored as sorted, and handbook is used as a fast lookup mechanism. In counting step, train data size is huge, so I filled all trigram files with multiple processes. I do it as 2 by 2. There are 8 trigram files, so it is filled in 4 steps.

I calculated perplexity with markov assumption. I added reducing value for out of vocabulary words. High perplexity is worse than lower. In GT Smoothing, out of vocabulary words are not handled. I solved this problem by adding smooth value. As expected, lowers perplexity is the trigrams perplexity.

The last step is generating random sentences. I chose starter randomly for unigram, and bigram. Then using Shannon's Method, sentences are generated. For trigrams, I choose the starter from the most probable 15 starters. Starter means, coming after "." and ". ". Only does this for trigram because trigram has that information. For all n-grams, end of sentence means ".", "!", "?". Unigram sentences are not meaningful. Bigrams are not either but better. Trigrams is the best one of these three as expected.

### 3- Preprocessing

---

File needed preprocessing, because it includes many unnecessary parts, and I changed Turkish characters. I also don't count English lines. Because English sentences can also be syllabicated, and it reduces the accuracy of the model.

```
for line in file:
    # pass wiki doc tags
    if line.startswith("<") or line.startswith("</"):
        continue

    if line.startswith("!"):
        line = line[1:]

    # if line is empty, skip it
    if line == "\n":
        continue

    # if line is in English, skip it
    try:
        lan = langdetect.detect(line)
        if lan == "en":
            continue
    except:
        # print("error: ", line)
        continue

    # preprocess line
    syllables = ngram_fun.preprocess(line)
    # directly concatenate syllables
    line = "".join(syllables)
    output_file.write(line + "\n")

def preprocess(line):
    obj = detector.TurkishNLP()

    # convert all of them to lowercase
    line = line.lower()

    # convert turkish characters to english characters
    line = line.replace("ç", "c")
    line = line.replace("ğ", "g")
    line = line.replace("ı", "i")
    line = line.replace("ö", "o")
    line = line.replace("ş", "s")
    line = line.replace("ü", "u")

    # separate words to syllables
    syllables = separator(obj, line)
    # print(syllables)

    return syllables
```

I used turkishnlp library for syllabication. Also, I removed English lines with using langdetect library.

After creating the file, input and output files are also created. I used 95% of the file as train data. Last 5% of the data as test file.





```
n = 95
m = 5

# calculate total number of lines
total_lines = sum(1 for line in file)
file.seek(0) # reset file pointer to the beginning

# calculate number of lines for each output file
n_lines = total_lines * n // 100
m_lines = total_lines - n_lines

# create output files
output_file1 = open(output_file_name1, "w", encoding="utf-8")
output_file2 = open(output_file_name2, "w", encoding="utf-8")

# write to output files
for i, line in enumerate(file):
    if i < n_lines:
        output_file1.write(line)
    else:
        output_file2.write(line)
```

 turkish_dump	2.10.2019 02:53	Dosya	452.168 KB
 turkish_dump_input	7.12.2023 22:26	Dosya	366.201 KB
 turkish_dump_preprocessed	7.12.2023 22:17	Dosya	384.671 KB
 turkish_dump_test	7.12.2023 22:26	Dosya	18.470 KB

From 452 MB file, model will be trained with 366 MB data, and the test will be done with 18.470 MB data.

## 4- Syllabication

---

```
def preprocess(line):
    obj = detector.TurkishNLP()

    # convert all of them to lowercase
    line = line.lower()

    # convert turkish characters to english characters
    line = line.replace("ç", "c")
    line = line.replace("ğ", "g")
    line = line.replace("ı", "i")
    line = line.replace("ö", "o")
    line = line.replace("ş", "s")
    line = line.replace("ü", "u")

    # separate words to syllables
    syllables = separator(obj, line)
    # print(syllables)

    return syllables
```

```
def separator(obj, line):
    # end of sentence and punctuations are also considered as syllable
    syllables = []

    splitted = line.split()
    # save each syllable, space, punctuation as a separate element
    for word in splitted:
        # if word
        word_syllables = obj.syllabicate(word)
        for syllable in word_syllables:
            # if syllable has punctuation, separate it
            if syllable[-1] in [".", ",", ";", ":", "?", "!", "/", "\\", "'", "\"", "(", ")"]:
                syllables.append(syllable[:-1])
                syllables.append(syllable[-1])

            # if syllable has punctuation at start, separate it
            elif syllable[0] in [".", ",", ";", ":", "?", "!", "/", "\\", "'", "\"", "(", ")"]:
                syllables.append(syllable[0])
                syllables.append(syllable[1:])
            else:
                syllables.append(syllable)

        # if it is not the last word, add space
        if word != splitted[-1]:
            syllables.append(" ")

    return syllables
```

Punctuations, space, and syllabicates are separated.

```
bozkir geleneginden gelen onlu teskilati kullanarak meritokratik (liyākata bagli) bir ordu meydana getiren cengiz han'in buyuk bir asker olarak un kazanmasi
nin temelinde, kurdugu posta teskilati ve casus agi ile istihbarat sanatina verdigi buyuk deger onemli bir yer tutar.
['boz', 'kir', 'ge', 'le', 'ne', 'gin', 'den', 'ge', 'len', 'on', 'lu', 'tes', 'ki', 'la', 'ti', 'kul', 'la', 'na', 'rak', 'me',
'ri', 'tok', 'ra', 'tik', '(', 'liyā', 'ka', 'ta', 'bag', 'li', ')', 'bir', 'or', 'du', 'mey', 'da', 'na', 'ge', 'ti',
'ren', 'cen', 'giz', 'han', 'in', 'bu', 'yuk', 'bir', 'as', 'ker', 'o', 'la', 'rak', 'un', 'ka', 'zan', 'ma', 'si',
'nin', 'te', 'me', 'lin', 'de', 'kur', 'du', 'gu', 'pos', 'ta', 'tes', 'ki', 'la', 'ti', 've', 'ca', 'sus', 'a', 'g',
'i', 'i', 'le', 'is', 'tih', 'ba', 'rat', 'sa', 'na', 'ti', 'na', 'ver', 'di', 'gi', 'bu', 'yuk', 'de', 'ger', 'o', 'nem',
'li', 'bir', 'yer', 'tu', 'tar', '.']
```

- example syllabication of a sentence.

## 5- Creating and Storing N-grams

- **Unigram:** I created all possible 1, 2, 3, 4 letter syllables, and some punctuations.

Whole unigram is stored in memory during execution due to its low size. After creation it is written on the disk with pickle. When it is needed (perplexity, sentence generation), it will be loaded to memory.

Writing

```
# write n-gram table to file
with open(dir_name + file_name, "wb") as file:
    pickle.dump(n_gram, file)
```

Reading

```
# print("file_name: ", file_name)
n_gram = pickle.load(file)
```

```
dict = {}
vowels = ["a", "e", "i", "o", "u"]
consonants = ["b", "c", "d", "f", "g", "h", "j", "k", "l",
              "m", "n", "p", "r", "s", "t", "v", "y", "z"]
# 1. only vowels v

for vowel in vowels:
    dict[vowel] = 0

# 2. vowel + consonant -> vc + cv
for vowel in vowels:
    for consonant in consonants:
        dict[vowel + consonant] = 0
        dict[consonant + vowel] = 0

# 3. consonant + vowel + consonant -> cvc + vcc
for consonant in consonants:
    for vowel in vowels:
        for consonant2 in consonants:
            dict[consonant + vowel + consonant2] = 0
            dict[vowel + consonant + consonant2] = 0

# 4. consonant + vowel + consonant + consonant -> cvcc
for consonant in consonants:
    for vowel in vowels:
        for consonant2 in consonants:
            for consonant3 in consonants:
                dict[consonant + vowel + consonant2 + consonant3] = 0
```


```
# add punctuations
dict["."] = 0
dict[","] = 0
dict[";"] = 0
dict[" ":""] = 0
dict["?"] = 0
dict["!"] = 0
dict["/"] = 0
dict["\\\\""] = 0
dict["'"] = 0
dict["\\'"] = 0
dict["("] = 0
dict[")"] = 0
dict[" "] = 0
dict["\\'"] = 0
```

This will create empty n-gram. Then it is used for counting and calculating probabilities.

```
if n == 1:
    for syllable in syllables:
        if syllable in n_gram:
            n_gram[syllable] += 1
```

Counts are written as this in unigram table.

After counting, and probability calculations (6<sup>th</sup> section) unigram is stored in binary file.

 unigram\_table

7.12.2023 22:33

BIN Dosyası

506 KB

Size of the unigram table.

- **Bigram:** Bigram table is created using unigram table. Keys are tuples in this case.

Similarly, bigram table is held in memory during execution. It is read, and written using pickle as same as unigram.


```
def create_empty_bigram(uni_gram):  
    # data structure for bi-gram: <tuple[syllable1, syllable2], count>  
    bigram = {}  
  
    # create empty bi-gram table  
    for key1 in uni_gram.keys():  
        for key2 in uni_gram.keys():  
            bigram[(key1, key2)] = 0  
  
    return bigram
```

To reduce the size of the bigram. Unigrams syllables that are not used or very little used are not added to bigram table to prevent sparse table, and great sizes.

```
for i in range(len(syllables)-n+1):  
    key = tuple(syllables[i:i+n])  
    if key in n_gram:  
        n_gram[key] += 1
```

Bigram table is filled as this.

After counting, and probability calculations bigram is stored in binary file.

 bigram\_table

7.12.2023 22:38

BIN Dosyası

30.312 KB

The size of the bigram table is roughly 30 MB.

- **Trigram:** Trigram is created using bigram and unigrams. It is held in dictionary as tuple, probability pairs. <(tuple1, syllable), probability> is the format of each element in trigram.

Storage of trigram is the challenge. Bigram size is 30 MB, it is reduced with the way of reducing unigram table. To create trigram table, both reduced unigram, and bigram were used. Even with this way, its size is huge. So, whole trigram cannot be held in memory during execution. My solution is to store them in memory with chunks. Accessing those chunks via another dictionary that is handbook.

```
# useful for finding file  
# <(tuple1, tuple2), count>  
# <((start_key1, start_key2), start_key3), (end_key1, end_key2), end_key3), count>  
tri_gram_handbook = {}
```

This is the data structure for accessing those chunks.

```

# trigram = <((syllable1, syllable2), syllable3), probability>
def create_empty_trigram(unigram, bigram):
    # data structure for trigram: <((syllable1, syllable2), syllable3), count>
    trigram = {}

    count = 0
    written = False
    bigram_keylist = sorted(list(bigram.keys()))
    unigram_keylist = sorted(list(unigram.keys()))
    # create empty trigram table
    for key1 in bigram_keylist:
        for key2 in unigram_keylist:

            written = False

            tuple = ((key1[0], key1[1]), key2)
            trigram[tuple] = 0

```

Bigram, and unigram keys must be sorted, because handbook access must be correct.

```

if sys.getsizeof(trigram) > MAX_MEMORY_USAGE:
    # write trigram to file as trigram_table_count.bin
    ngram_fun.write_to_file(trigram, file_name_tri + str(count) + ".bin")

    # add to handbook consist of start and end tuples as key, and count as value
    #print(trigram.keys())

    # find max, and min keys
    key_list = list(trigram.keys())

    key_h1 = key_list[0]
    key_h2 = key_list[len(key_list) - 1]

    print("key1: " + str(key_h1) + ", key2: " + str(key_h2) + ", count: " + str(count))

    # add to handbook
    tri_gram_handbook[(key_h1, key_h2)] = count

    written = True
    count += 1
    # clear trigram
    trigram.clear()

```

If maximum size is reached, then write it as chunk and save it to handbook. Also clear the trigram.



```

for i in range(0, TRIGRAM_TABLE_COUNT, TRIGRAM_TABLE_COUNT // 3):
    processes = []
    start = i
    end = min(i + TRIGRAM_TABLE_COUNT // 3, TRIGRAM_TABLE_COUNT)

    for j in range(start, end):
        print("start: " + str(start) + ", end: " + str(end) + ", j: " + str(j))
        if j >= TRIGRAM_TABLE_COUNT:
            break










        # create process
        p = multiprocessing.Process(target=count_trigram, args=(file_name_tri + str(j) + ".bin", bigram, unigram))
        processes.append(p)
        p.start()
        print("process started: " + str(j))

    # wait for all processes in the current set to finish
    for p in processes:
        p.join()

```

For counting, I used parallelization. Too many reading, writing to disk will be slow. So, I trained each trigram table separately. This training is done 2 by 2. I created another process instead of thread, in order to get more memory. Parent process waits the child processes to finish. When it is all finished, counting is done.

After that probabilities are calculated (explained in 6<sup>th</sup> section), and trigram table creation will be done.

 trigram_handbook	7.12.2023 23:52	BIN Dosyası	1 KB
 trigram_table_0	8.12.2023 00:43	BIN Dosyası	140.850 KB
 trigram_table_1	8.12.2023 00:43	BIN Dosyası	150.827 KB
 trigram_table_2	8.12.2023 00:43	BIN Dosyası	151.597 KB
 trigram_table_3	8.12.2023 00:44	BIN Dosyası	151.112 KB
 trigram_table_4	8.12.2023 00:44	BIN Dosyası	151.788 KB
 trigram_table_5	8.12.2023 00:44	BIN Dosyası	151.188 KB
 trigram_table_6	8.12.2023 00:45	BIN Dosyası	151.459 KB
 trigram_table_7	8.12.2023 00:45	BIN Dosyası	1.177 KB

Total size is roughly 1 GB.

## 6- Probability Calculations, and GT-Smoothing

Probabilities are calculated with smoothing which is GT-Smoothing. So, table will not have zero counts.

- **Unigram:**  $P(a) = c(a) / \text{total}$

```
# calculate frequency of frequencies
freq_of_freq = {}

for count in uni_gram.values():
    if count not in freq_of_freq:
        freq_of_freq[count] = 0
    freq_of_freq[count] += 1
```

Calculating frequency of frequencies

```
smoothed_count = (count + 1) * freq_of_freq[count + 1] / freq_of_freq[count]
```

This is the reconstructed count.

```
smoothed_uni_gram[key] = smoothed_count / total_count
```

Then, calculating the probability of that element.

After calculating the probability, this is some part of the unigram table.

ju 0.0001035086959690412	eyl 5.752534347998325e-08
uk 3.330034019048981e-05	yem 1.267396871620073e-05
ku 0.0030494887666270543	eym 7.4346309964757e-10
ul 0.00039687200245330795	yen 0.00036002494605015154
lu 0.004237960119027131	eyn 6.465368096006785e-08
um 1.2879013247245516e-05	yep 1.3189350915853842e-06
mu 0.0022357390655833656	eyp 1.572578687226183e-08
un 0.0006230915376365304	yer 0.0011224026794509974
nu 0.0032204294045894187	eyr 5.266651822048942e-09
up 9.005334133723737e-06	yes 2.7409466231013907e-05
pu 0.0001516054928592494	eyt 5.752534347998325e-08
ur 3.2973377289634604e-05	
ru 0.002625822431535867	
us 0.0001822236289139101	

- **Bigram:**  $P(a|b) = c(a,b) / c(b)$

```
# calculate probability with good turing smoothing
def bi_gram_prob(bi_gram, uni_gram_counts):
    # <tuple[syllable1, syllable2], count> -> <tuple[syllable1, syllable2], probability>

    # p(a|b) = c(a,b) / c(b)
    # gt smoothing

    # calculate frequency of frequencies
    freq_of_freq = {}

    for count in bi_gram.values():
        if count not in freq_of_freq:
            freq_of_freq[count] = 0
        freq_of_freq[count] += 1
```

Frequency of frequencies is calculated with this way. Similar to the unigram.

```
smoothed_count = (count + 1) * freq_of_freq[count + 1] / freq_of_freq[count]
```

Calculation of

reconstructed count.

```
smoothed_bi_gram[key] = smoothed_count / uni_gram_counts[key[0]]
```

Probability of that element.

Some part of the Bigram table

```
('a', 'a') 1.67812861043977e-08
('a', 'e') 1.67812861043977e-08
('a', 'i') 1.67812861043977e-08
('a', 'o') 1.67812861043977e-08
('a', 'u') 1.67812861043977e-08
('a', 'ab') 1.67812861043977e-08
('a', 'ba') 0.0007643013630103756
('a', 'ac') 1.67812861043977e-08
('a', 'ca') 0.0022194497934239195
('a', 'ad') 1.67812861043977e-08
('a', 'da') 0.02854151410774502
('a', 'af') 1.67812861043977e-08
('a', 'fa') 0.00019392323718263457
('giz', 'hak') 1.8661422807321322e-06
('giz', 'hal') 1.8661422807321322e-06
('giz', 'ham') 1.8661422807321322e-06
('giz', 'han') 0.0015174491345897544
('giz', 'hap') 1.8661422807321322e-06
('giz', 'har') 1.8661422807321322e-06
('giz', 'has') 1.8661422807321322e-06
('giz', 'hat') 1.8661422807321322e-06
('giz', 'hav') 1.8661422807321322e-06
```

- **Trigram:**  $P(c | (a,b)) = c(c) / c((a,b))$

```
# <((syllable1, syllable2), syllable3), probability>
# gt smoothing

# calculate frequency of frequencies
freq_of_freq = {}

for count in trigram.values():
    if count not in freq_of_freq:
        freq_of_freq[count] = 0
    freq_of_freq[count] += 1
```

Calculating frequency of frequencies

```
smoothed_count = (count + 1) * freq_of_freq[count + 1] / freq_of_freq[count]
```

Smoothed counts

```
smoothed_trigram[key] = smoothed_count / bigram[key[0]]
```

Probability calculation

Some part of the trigram table

```
((('.', ' '), 'cul') 2.2572444920252797e-05
((('.', ' '), 'cum') 0.0005209820843422533
((('.', ' '), 'cun') 0.0008133446470332582
((('.', ' '), 'cup') 6.9091191170906895e-06
((('.', ' '), 'cur') 1.6964390425140855e-05
((('.', ' '), 'cus') 1.3206431154645708e-05
((('.', ' '), 'cut') 2.3450338929074443e-06
(('ko', 'lay'), 'ci') 0.0004517199623267704
(('ko', 'lay'), 'cid') 7.33975032299614e-07
(('ko', 'lay'), 'cif') 7.33975032299614e-07
(('ko', 'lay'), 'cift') 7.33975032299614e-07
(('ko', 'lay'), 'cig') 7.33975032299614e-07
(('ko', 'lay'), 'cih') 7.33975032299614e-07
(('ko', 'lay'), 'cik') 7.33975032299614e-07
```

## 7- Perplexity

Perplexity is calculated with Markov Assumption. I used %5 of the preprocessed data. Which means 18.47 MB data.

 turkish_dump_test	7.12.2023 22:26	Dosya	18.470 KB
---	-----------------	-------	-----------

Logarithm is used to prevent underflow problem.

```
log_perplexity -= math.log(uni_gram[syllable])
perplexity = math.exp(log_perplexity / float(N))
```

for unigram

```
log_perplexity -= math.log(bi_gram[(syllables[i], syllables[i+1])])
log_perplexity -= math.log(0.000001)
```

for bigram

```
log_perplexity -= math.log(trigram[syllable])
perplexity = math.exp(log_perplexity / float(N))
```

for trigram

Perplexity Results:

```
✓ [3] %run perplexity.py
2 dk.
testing unigram...
perplexity of unigram: 202.9635363111985
testing bigram...
perplexity of bigram: 93.63528691138761
testing trigram...
perplexity of trigram: 82.28023538328674
```

As expected, if  $n$  is higher, perplexity is lower for  $n$ -grams. This shows, trigram will work better than bigram and unigram. Of course, bigram is better than unigram.

## 8- Random Sentence Generation

---

Random sentences are generated with the Shannon's Method.

For unigram, at each step a syllable is picked from most probable 5 syllables. There is no dependency between each picking. This will cause all sentences consist of only 5 syllables. I choose first syllable randomly, so there could be one extra.

For bigram, there will be dependency. If a syllable is picked, the syllable which is from most probable 5 syllable after coming that syllable. So, more accurate results expected. Again first syllable will be randomly chosen.

For trigram, after 2 syllables, most probable syllable between 5 syllables will be chosen. This must be better than bigram. First syllable will be chosen from most probable syllable after ('.', ' ').

```
next_syllable = random.choices(syllables, weights=probabilities, k=1)[0]
```

I used this function of random library to choose according to the probabilities.

- Sentences:

```
%run sentence_generator.py
testing unigram...
sentence of unigram: jis le ri lela la la ri lara lala ri .
testing bigram...
sentence of bigram: giysahne ve ve arasinde olan icindedir.
testing trigram...
sentence of trigram: ayhan olan bulunur, buyuk bir sehir belirlendirilir ve bulunan en cok dayalidir ve kazanmistir.
```

```
%run sentence_generator.py

igram...
f unigram: guzv ri la la la le le la la la .
igram...
f bigram: romkapsayilinda icerini arasinda ola ve olanmistirmele ve ilerini arafindan bir arafindan bir olanin iki olan bir bir ara ve arada bir ve olari ve ve ve olan olanmistir.
igram...
f trigram: birtakim arasinda olan burada yapimi icin oyun icme sure ve olarak da yapisi ile birlikte, arasinda bu ile bir macinayet galaksidir.
```

```
%run sentence_generator.py

testing unigram...
sentence of unigram: zalrla le rile ri lela ri la la rile .
testing bigram...
sentence of bigram: neraflama ileri, icin ice ara ozel ve ve ikiligi ileri bir ve ve ara oyuncudur.
testing trigram...
sentence of trigram: aysen ve bu arasinda ikiye baslanmistir.
```

```
%run sentence_generator.py

ting unigram...
tence of unigram: mupj ri le .
ting bigram...
tence of bigram: bomzem ile olari birlikte olarinin ikisi ve olari ve bir adiger arasindan olusturubu olari icesininda icin icin arafindan bir ola ala bir bir a adi a icin olanmistir.
ting trigram...
tence of trigram: ansiklopedi arasinda yasayan yolda bu sebekesi ile birlikte olarak gore icin kullanmaktadir.
```

```
%run sentence_generator.py

testing unigram...
sentence of unigram: ilf le rile la ri .
testing bigram...
sentence of bigram: ritligde ola iki adisi alandi.
testing trigram...
sentence of trigram: adanmistir.
```

```
%run sentence_generator.py

testing unigram...
sentence of unigram: yum la ri le lale la ri le ri la .
testing bigram...
sentence of bigram: vimrupa icin ve ve bir ola adigi olan ve ve isebesi ilerine olus ara arasinda bir bir olan olan arafindan a ve ve ve olandiginisanlar.
testing trigram...
sentence of trigram: bagisladigi ve bu olarak, bu olarak takim ve bulundu.
```

```
%run sentence_generator.py

g unigram...
ce of unigram: tuns ri la.
g bigram...
ce of bigram: antehdit bir icin ve arafindan bircok oyun ozelliklarak birlerinde olamalarinesi ozelti.
g trigram...
ce of trigram: ba" olan ve savaslari arasindaki amerikalici olarak kara'nin enlemine bagli bir maci olan ve olarak kadar bulunan bir alanlarin aralik oyunlara ve bu ise yapisi yoktur.
```

For unigram, sentences are meaningless. Since we are choosing from the most probable 5, they don't mean anything and are repetitive. I manually prevented spaces generated consecutively, if I didn't do this there would be too many spaces.

For the bigram, meaning is far better than unigram. They produced meaningful words which consists of 2 syllables mostly. In general sentence doesn't mean anything, but it can be seen that there are meaningful separate words generated. Also, spacing is done when necessary. Sentences are ended after "-dur, -dir, -tir, di .." kind of syllables. It fails when it cannot choose space character when it is needed. If word becomes longer, it is very difficult for bigram to make it meaningful. Almost all sentences generated by bigram don't mean anything.

For the trigram, almost all words are meaningful separately. As expected, it is better than bigram, and unigram. For every 2-3 words it generated sensible words. But it fails for general meaning. Again, whole sentence mostly doesn't mean anything, but better than bigram.

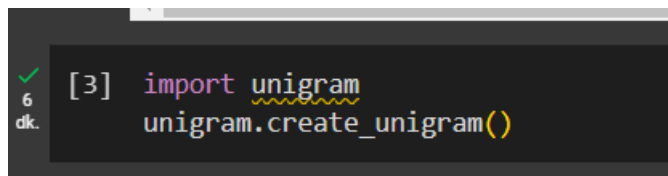
## 9- Performance

---

- Tests are done in Google Colab environment.

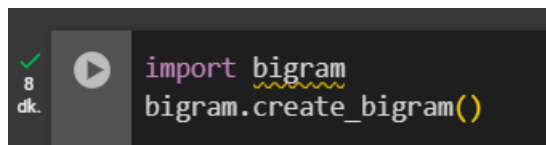
### Training:

- Size of the training data is 366 MB

A screenshot of a Jupyter Notebook cell. On the left, there is a green checkmark, the number '6', and the text 'dk.'. The code in the cell is: 

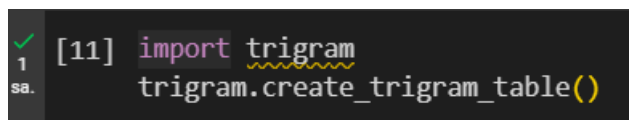
```
[3] import unigram
unigram.create_unigram()
```

Training unigram took 6 minutes.

A screenshot of a Jupyter Notebook cell. On the left, there is a green checkmark, the number '8', and the text 'dk.'. There is a play button icon. The code in the cell is: 

```
import bigram
bigram.create_bigram()
```

Training bigram took 8 minutes.

A screenshot of a Jupyter Notebook cell. On the left, there is a green checkmark, the number '1', and the text 'sa.'. The code in the cell is: 

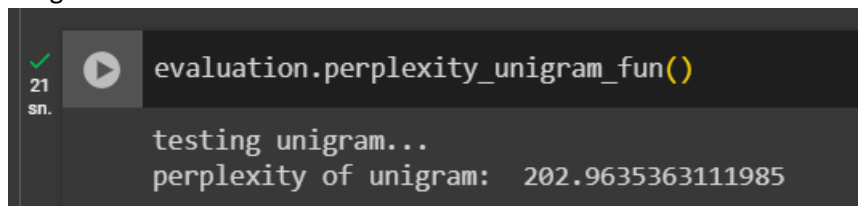
```
[11] import trigram
trigram.create_trigram_table()
```

Took 1 hour to train trigram. This is mostly caused by counting step. Filling separate files causes this long time.

### Perplexity:

- Size of the Test data is 18 MB

Unigram:

A screenshot of a Jupyter Notebook cell. On the left, there is a green checkmark, the number '21', and the text 'sn.'. There is a play button icon. The code in the cell is: 

```
evaluation.perplexity_unigram_fun()

testing unigram...
perplexity of unigram: 202.9635363111985
```

- It took 21 seconds to calculate perplexity for unigram.

Bigram:

```
✓ 32 [38] evaluation.perplexity_bigram_fun()  
sn.  
testing bigram...  
perplexity of bigram: 93.63528691138761
```

- It is quite similar to unigram.

Trigram:

```
✓ 1 [39] evaluation.perplexity_trigram_fun()  
dk.  
testing trigram...  
perplexity of trigram: 82.28023538328674
```

- Trigram takes more time, it is 1 minute. This caused by disk access for table.

## Sentence Generation:

Unigram:

```
✓ 0 [33] import evaluation  
sn. evaluation.sentence_unigram_fun()  
testing unigram...  
sentence of unigram: seljla la la ri la le le lala la .
```

- Less than 1 seconds.

Bigram:

```
✓ 54 [34] evaluation.sentence_bigram_fun()  
sn. testing bigram...  
sentence of bigram: susrojene alan arafindan ve olandi ve arafindan bir bir ve aradaya a verinin icinde olari ve a ara a icin bir bir ve arafindan olus oluslarak ve bir bir alarinin a
```

- Depending on the sentence size, it takes up to 1 minute. This was quite a long sentence.

Trigram:

```
✓ 6 [36] evaluation.sentence_trigram_fun()  
dk. testing trigram...  
sentence of trigram: aykiri icin bir arasindaki bir alanlama ile ile, bir sezonunda, bir sure alan edildi.
```

- It took 6 minutes to produce this sentence. Depending on the length of the sentence this time can increase.

**In general**, trigram gives bad performance. This is caused by distributed storage on disk. Instead of storing table in binary files, a database can significantly increase the performance of trigram.