Week 4

N-Grams

Aim→

Probability of a sentence can be calculated by the probability of sequence of words occuring in it. We can use Markov assumption, that the probability of a word in a sentence depends on the probability of the word occuring just before it. Such a model is called first order Markov model or the bigram model.

$$P(W_n | W_{n-1}) = P(W_{n-1}, W_n)/P(W_{n-1})$$

Here, W_n refers to the word token corresponding to the nth word in a sequence

Theory→

A combination of words forms a sentence. However, such a formation is meaningful only when the words are arranged in some order.

Eg: Sit I car in the

Such a sentence is not grammatically acceptable. However some perfectly grammatical sentences can be nonsensical too!

Eg: Colorless green ideas sleep furiously

One easy way to handle such unacceptable sentences is by assigning probabilities to the strings of words i.e, how likely the sentence is.

Probability of a sentence

If we consider each word occurring in its correct location as an independent event,the probability of the sentences is : P(w(1), w(2)..., w(n-1), w(n))

Using chain rule: =

P(w(1)) * P(w(2) | w(1)) * P(w(3) | w(1)w(2)) ... P(w(n) | w(1)w(2) ... w(n-1))

Bigrams

We can avoid this very long calculation by approximating that the probability of a given word depends only on the probability of its previous words. This assumption is called Markov assumption and such a model is called Markov model- bigrams. Bigrams can be generalized to the n-gram which looks at (n-1) words in the past. A bigram is a first-order Markov model.

Therefore , **P(w(1), w(2)..., w(n-1), w(n)) = P(w(2)|w(1)) P(w(3)|w(2)) ... P(w(n)|w(n-1))**

We use (eos) tag to mark the beginning and end of a sentence.

A bigram table for a given corpus can be generated and used as a lookup table for calculating probability of sentences.

Eg: Corpus - (eos) You book a flight (eos) I read a book (eos) You read (eos)

Bigram Table:

	(eos)	**you**	**book**	**a**	**flight**	**I**	**read**
(eos)	0	0.33	0	0	0	0.25	0
you	0	0	0.5	0	0	0	0.5
book	0.5	0	0	0.5	0	0	0
a	0	0	0.5	0	0.5	0	0
flight	1	0	0	0	0	0	0
I	0	0	0	0	0	0	1
read	0.5	0	0	0.5	0	0	0

P((eos) you read a book (eos))
= **P(you|eos)** * **P(read|you)** * **P(a|read)** * **P(book|a)** * **P(eos|book)**
= 0.33 * 0.5 * 0.5 * 0.5 * 0.5
=.020625

Objective

- The objective of this experiment is to learn to calculate bigrams from a given corpus and calculate probability of a sentence.

Procedure

STEP 1: Select a corpus and click on `Generate bigram table`
STEP 2: Fill up the table that is generated and hit `Submit`

STEP 3: If incorrect (red), see the correct answer by clicking on show answer or repeat Step 2.

STEP 4: If correct (green), click on take a quiz and fill the correct answer

Assignment

Q1. A trigram is a second-order Markov model. Derive the formula to calculate trigram probability. Next, calculate the trigram probabilities for the given corpus.

(eos) Can I sit near you (eos) You can sit (eos) Sit near him (eos) I can sit you (eos)

Source code

```
import nltk

from nltk.util import trigrams

from nltk.probability import ConditionalFreqDist, ConditionalProbDist, MLEProbDist

corpus = [

    "Can I sit near you",

    "You can sit",

    "Sit near him",

    "I can sit you"

]

tokenized_corpus = [nltk.word_tokenize(sentence) for sentence in corpus]

trigram_list = list(trigrams(word for sentence in tokenized_corpus for word in sentence))

cfd = ConditionalFreqDist()

for trigram in trigram_list:

    condition = (trigram[0], trigram[1])

    cfd[condition][trigram[2]] += 1

cpd = ConditionalProbDist(cfd, MLEProbDist)
```

```
trigram_probabilities = {}
for condition in cpd.conditions():
    trigram_probabilities[condition] = {}
    for word in cpd[condition].samples():
        trigram_probabilities[condition][word] = cpd[condition].prob(word)
for condition in trigram_probabilities:
    for word in trigram_probabilities[condition]:
        print(f"Probability of '{word}' given the condition {condition}:
{trigram_probabilities[condition][word]}")
```

Q2. A character based N-gram is a set of n consecutive characters extracted from a word. It is generally used in measuring the similarity of character strings. Some of its applications are in spellcheckers, stemming, OCR error correction, etc.

Given, four valid words:

(a)quote
(b)patient
(c)patent
(d) impatient

Source code

```
import nltk
words = ["quote", "patient", "patent", "impatient"]
n = 3  # Change this value to adjust the n-gram size
def generate_ngrams(word):
    return [word[i:i+n] for i in range(len(word)-n+1)]
 word_ngrams = {word: generate_ngrams(word) for word in words}
 def jaccard_similarity(ngram1, ngram2):
    intersection = len(set(ngram1) & set(ngram2))
    union = len(set(ngram1) | set(ngram2))
    return intersection / union if union else 0
 # Calculate similarity between each pair of words
similarities = { }
for i in range(len(words)):
    for j in range(i+1, len(words)):
        word1, word2 = words[i], words[j]
        ngram1, ngram2 = word_ngrams[word1], word_ngrams[word2]
        similarities[(word1, word2)] = jaccard_similarity(ngram1, ngram2)
 # Print the n-grams and their similarities
for word1, word2 in similarities:
    print(f"Words: {word1} and {word2}")
    print(f"N-grams: {word_ngrams[word1]} & {word_ngrams[word2]}")
    print(f"Similarity: {similarities[(word1, word2)]:.4f}\n")
```

Calculate the probability of occurrence of each word given below. Which of these represent the correct spelling?

(a)qotient
(b)quotent
(c) quotient

<u>Source code</u>

```
import nltk
 # Define corpus text (a large sample of text)
 corpus_text = nltk.corpus.gutenberg.raw("shakespeare-hamlet.txt
words = ["qotient", "quotent", "quotient"]  # Include all potential spellings

 tokens = nltk.word_tokenize(corpus_text.lower())

fdist = nltk.FreqDist(tokens)

 total_words = len(tokens)

probabilities = {word: (fdist[word] + 1) / (total_words + len(fdist.keys())) for word
in words}



for word, prob in probabilities.items():

    print(f"- {word}: {prob:.4f}")

 most_likely_word = max(probabilities, key=probabilities.get)

print("\nMost Likely Correct Spelling:")

print(most_likely_word)
```

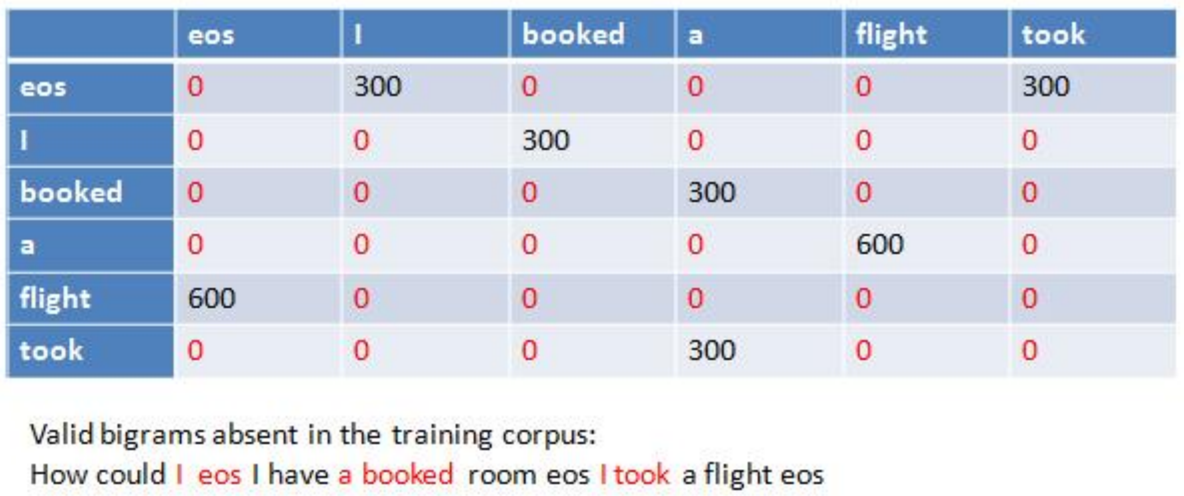
Week - 5

N-Grams Smoothing

Aim

One major problem with standard N-gram models is that they must be trained from some corpus, and because any particular training corpus is finite, some perfectly acceptable N-grams are bound to be missing from it. We can see that bigram matrix for any given training corpus is sparse. There are large number of cases with zero probabilty bigrams and that should really have some non-zero probability. This method tend to underestimate the probability of strings that happen not to have occurred nearby in their training corpus.

There are some techniques that can be used for assigning a non-zero probabilty to these 'zero probability bigrams'. This task of reevaluating some of the zero-probability and low-probabilty N-grams, and assigning them non-zero values, is called smoothing.



	eos	I	booked	a	flight	took
eos	0	300	0	0	0	300
I	0	0	300	0	0	0
booked	0	0	0	300	0	0
a	0	0	0	0	600	0
flight	600	0	0	0	0	0
took	0	0	0	300	0	0

Valid bigrams absent in the training corpus:
How could I eos I have a booked room eos I took a flight eos

Theory

The standard N-gram models are trained from some corpus. The finiteness of the training corpus leads to the absence of some perfectly acceptable N-grams. This results in sparse bigram matrices. This method tend to underestimate the probability of strings that do not occur in their training corpus.

There are some techniques that can be used for assigning a non-zero probabilty to these 'zero probability bigrams'. This task of reevaluating some of the zero-probability and low-probabilty N-grams, and assigning them non-zero values, is called smoothing. Some of the techniques are: Add-One Smoothing, Witten-Bell Discounting, Good-Turing Discounting.

Add-One Smoothing

In Add-One smooting, we add one to all the bigram counts before normalizing them into probabilities. This is called add-one smoothing.

Application on unigrams

Normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

$P(w_n|w_{n-1}) = C(w_{n-1}w_n)/C(w_{n-1})$

For add-one smoothed bigram counts we need to augment the unigram count by the number of total word types in the vocabulary

V: $p^*(w_n|w_{n-1}) = (C(w_{n-1}w_n)+1)/(C(w_{n-1})+V)$

Objective

The objective of this experiment is to learn how to apply add-one smoothing on sparse bigram table

Procedure

STEP 1: Select a corpus

STEP 2: Apply add one smoothing and calculate bigram probabilities using the given bigram counts,N and V. Fill the table and hit `Submit`

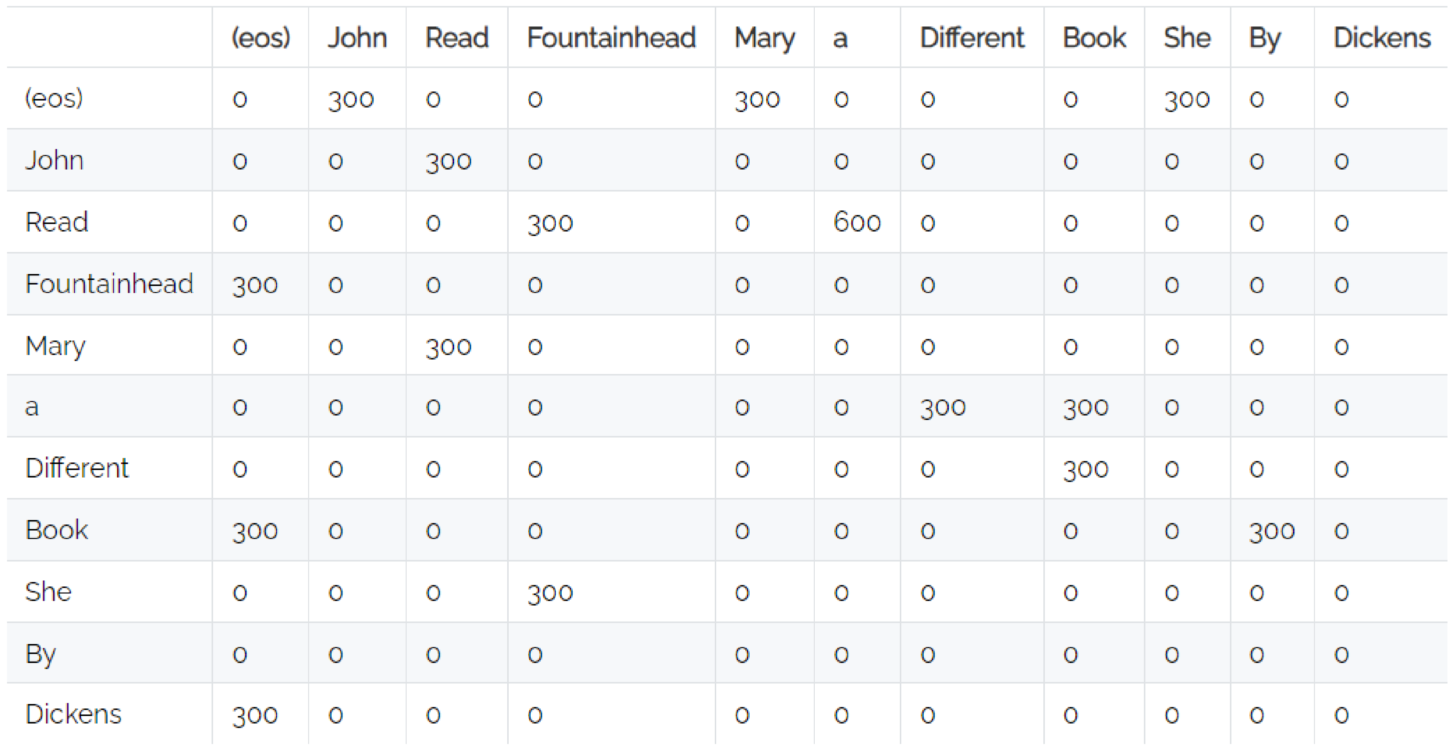
STEP 3: If incorrect (red), see the correct answer by clicking on show answer or repeat Step 2

Assignment

1)Add-one smoothing works horribly in practice because of giving too much probability mass to unseen n-grams. Prove using an example.

```
import nltk
from nltk.util import ngrams
from nltk.probability import FreqDist
corpus = ["I like apples", "I like bananas", "I like oranges"]
sentence = "I like pears"
corpus_tokens = [nltk.word_tokenize(sentence) for sentence in corpus]
sentence_tokens = nltk.word_tokenize(sentence)
corpus_ngrams = [ngrams(tokens, 2) for tokens in corpus_tokens]
sentence_ngrams = list(ngrams(sentence_tokens, 2))
fdist = FreqDist([ngram for ngrams in corpus_ngrams for ngram in ngrams])
probabilities = []
for ngram in sentence_ngrams:
    count = fdist[ngram] + 1  # Add-One smoothing
    total_count = len(fdist) + len(sentence_ngrams)  # Add-One smoothing
    probability = count / total_count
    probabilities.append(probability)
for ngram, probability in zip(sentence_ngrams, probabilities):
    print(f"Probability of {ngram}: {probability}")
```

Q2. In Add-δ smoothing, we add a small value 'δ' to the counts instead of one. Apply Add-δ smoothing to the below bigram count table where δ=0.02.



	(eos)	John	Read	Fountainhead	Mary	a	Different	Book	She	By	Dickens
(eos)	0	300	0	0	300	0	0	0	300	0	0
John	0	0	300	0	0	0	0	0	0	0	0
Read	0	0	0	300	0	600	0	0	0	0	0
Fountainhead	300	0	0	0	0	0	0	0	0	0	0
Mary	0	0	300	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	300	300	0	0	0
Different	0	0	0	0	0	0	0	300	0	0	0
Book	300	0	0	0	0	0	0	0	0	300	0
She	0	0	0	300	0	0	0	0	0	0	0
By	0	0	0	0	0	0	0	0	0	0	0
Dickens	300	0	0	0	0	0	0	0	0	0	0

N = 5100 V = 11

Source code

```
import nltk


# Define bigram counts and smoothing delta

bigram_counts = {

    ("eos", "John"): 0,

    ("John", "Read"): 0,

    ("Read", "Fountainhead"): 300,
    ("Fountainhead", "Mary"): 0,
    ("Mary", "a"): 0,
    ("a", "Different"): 0,
    ("Different", "Book"): 0,
    ("Book", "She"): 300,
    ("She", "By"): 0,
    ("By", "Dickens"): 0,
    ("Dickens", "(eos)"): 300,
    # Add more bigrams as needed
}

delta = 0.02  # Smoothing delta value

# Calculate total number of bigrams and vocabulary size
N = sum(bigram_counts.values())
V = len(set(bigram_counts.keys()))  # Assuming unique bigrams are keys

def add_delta_smoothing(counts, delta, N, V):
    """
    Applies Add-δ smoothing to a bigram count table.

    Args:
        counts (dict): Dictionary of bigram counts.
        delta (float): Smoothing delta value.
        N (int): Total number of bigrams.
        V (int): Vocabulary size (number of unique bigrams).


    Returns:

        dict: Dictionary of smoothed bigram probabilities.

    """


    smoothed_counts = {ngram: count + delta for ngram, count in counts.items()}

    smoothed_probabilities = {ngram: (count + delta) / (N + delta * V) for ngram, count in smoothed_counts.items()}

return smoothed_probabilities


# Apply Add-δ smoothing and print results

smoothed_probs = add_delta_smoothing(bigram_counts, delta, N, V)

for bigram, prob in smoothed_probs.items():

    print(f"Probability of '{bigram[1]}' after '{bigram[0]}': {prob:.4f}")
```

Q3. Given S = Dickens read a book, find P(S)
(a) Using unsmoothed probability
(b) Applying Add-One smoothing.
(c) Applying Add-δ smoothing

<u>Source code</u>

```
from nltk.util import bigrams

from nltk.probability import FreqDist, LidstoneProbDist


# Given bigram count table

bigram_counts = {

    ('eos', 'John'): 0,
    ('eos', 'Read'): 300,
    # ... (complete the table with the provided counts)
}


# Sentence to find probability
sentence = ['Dickens', 'read', 'a', 'book']


# (a) Unsmoothed probability
unsmoothed_prob = 1.0
for bigram in bigrams(['eos'] + sentence):
    unsmoothed_prob *= bigram_counts.get(bigram, 0) /
bigram_counts.get((bigram[0],), 1)
print(f"(a) Unsmoothed Probability P(S): {unsmoothed_prob}")


# (b) Applying Add-One smoothing
add_one_prob = 1.0
N = sum(bigram_counts.values())
V = len(set(word for bigram in bigram_counts.keys() for word in bigram))
for bigram in bigrams(['eos'] + sentence):
    add_one_prob *= (bigram_counts.get(bigram, 0) + 1) /
(bigram_counts.get((bigram[0],), 0) + V)
print(f"(b) Add-One Smoothing Probability P(S): {add_one_prob}")


# (c) Applying Add-δ smoothing
```

```
delta = 0.02

add_delta_prob = 1.0

for bigram in bigrams(['eos'] + sentence):

    add_delta_prob *= LidstoneProbDist(FreqDist(bigram_counts), delta,
bins=V).prob(bigram)

print(f"(c) Add-δ Smoothing Probability P(S): {add_delta_prob}")
```