Assignment 1

In the first assignment, you will implement some of the algorithms you have learnt in the first two weeks of lectures: n-gram language models, and syntactic parsing using the CYK algorithm.

Setup

For this and other assignments, we will be using Google Colab, for both code as well as descriptive questions. Your task is to finish all the questions in the Colab notebook and then upload a PDF version of the notebook, and a viewable link on Gradescope.

Google colaboratory

Before getting started, get familiar with google colaboratory: https://colab.research.google.com/notebooks/welcome.ipynb

This is a neat python environment that works in the cloud and does not require you to set up anything on your personal machine (it also has some built-in IDE features that make writing code easier). Moreover, it allows you to copy any existing collaboratory file, alter it and share with other people.

Submission

Before you start working on this homework do the following steps:

1. Press **File > Save a copy in Drive...** tab. This will allow you to have your own copy and change it.
2. Follow all the steps in this collaboratory file and write / change / uncomment code as necessary.
3. Do not forget to occasionally press **File > Save** tab to save your progress.
4. After all the changes are done and progress is saved press **Share** button (top right corner of the page), press **get shareable link** and make sure you have the option **Anyone with the link can view** selected. Copy the link and paste it in the box below.
5. After completing the notebook, press **File > Download .ipynb** to download a local copy on your computer, and then upload the file to Gradescope.

Paste your notebook link in the box below. *(0 points)*

https://colab.research.google.com/drive/1cnDgl7hqNgmGANwnHajak3-7zauo63gm?usp=sharing

```
# Downloads required packages and files
required_files = "https://github.com/jhu-intro-hlt/jhu-intro-hl
! wget $required_files && unzip -o required_files.zip
! pip install -r requirements.txt
```

```
--2025-09-17 00:58:51--  https://github.com/jhu-intro-hlt/jhu-intro-hlt.github.i
Resolving github.com (github.com)... 140.82.113.3
Connecting to github.com (github.com)|140.82.113.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/jhu-intro-hlt/jhu-intro-hlt.github.i
--2025-09-17 00:58:51--  https://raw.githubusercontent.com/jhu-intro-hlt/jhu-int
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.
HTTP request sent, awaiting response... 200 OK
Length: 11474 (11K) [application/zip]
Saving to: ‘required_files.zip’

required_files.zip  100%[===================>]  11.21K  --.-KB/s    in 0s

2025-09-17 00:58:51 (31.1 MB/s) - ‘required_files.zip’ saved [11474/11474]

Archive:  required_files.zip
  inflating: requirements.txt
   creating: tests/
  inflating: tests/cfg-iscnf.py
  inflating: tests/cfg-tocnf.py
  inflating: tests/cyk-impl.py
  inflating: tests/ngramlm-corpus-size.py
  inflating: tests/ngramlm-empirical-distribution.py
  inflating: tests/ngramlm-impl.py
  inflating: tests/ngramlm-improvement-impl.py
  inflating: tests/ngramlm-laplace-smoothing-impl.py
  inflating: tests/ngramlm-laplace-smoothing-perp.py
  inflating: tests/ngramlm-perp-impl.py
  inflating: tests/ngramlm-quad-perp-on-training.py
  inflating: tests/ngramlm-tri-perp-on-dev.py
  inflating: tests/ngramlm-tri-perp-on-training.py
  inflating: tests/ngramlm-vocab-size.py
  inflating: tests/warmup-ngram.py
Collecting datascience (from -r requirements.txt (line 1))
  Downloading datascience-0.18.0-py3-none-any.whl.metadata (910 bytes)
Requirement already satisfied: jupyter_client in /usr/local/lib/python3.12/dist-
Requirement already satisfied: ipykernel in /usr/local/lib/python3.12/dist-packa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-pack
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.12/dist-pack
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-packages
Requirement already satisfied: seaborn in /usr/local/lib/python3.12/dist-package
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.12/dist-pa
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages
```

```
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages
Requirement already satisfied: nbconvert in /usr/local/lib/python3.12/dist-packa
Requirement already satisfied: nbformat in /usr/local/lib/python3.12/dist-packag
Requirement already satisfied: dill in /usr/local/lib/python3.12/dist-packages (
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages
Collecting otter-grader==4.0.1 (from -r requirements.txt (line 15))
  Downloading otter_grader-4.0.1-py3-none-any.whl.metadata (2.8 kB)
Collecting pdfkit (from -r requirements.txt (line 17))
  Downloading pdfkit-1.0.0-py3-none-any.whl.metadata (9.3 kB)
Collecting PyPDF2 (from -r requirements.txt (line 18))
  Downloading pypdf2-3.0.1-py3-none-any.whl.metadata (6.8 kB)
Requirement already satisfied: nltk in /usr/local/lib/python3.12/dist-packages (
Collecting jupyter (from -r requirements.txt (line 23))
  Downloading jupyter-1.1.1-py2.py3-none-any.whl.metadata (2.0 kB)
```

```
# Initialize Otter
import otter
grader = otter.Notebook(colab=True)
```

```
import random
import math
import re
import os
import urllib
import json
from typing import *
from collections import Counter, defaultdict

import numpy as np
import nltk
import matplotlib.pyplot as plt
from nltk.tokenize import RegexpTokenizer, sent_tokenize

nltk.download('punkt')
nltk.download('punkt_tab')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
True
```

Part 1: N-gram Language Models

For the first part of this assignment, you will implement a trigram language model and train it on a small corpus. You will then implement a scoring function to compute the

train it on a small corpus. You will then implement a scoring function to compute the perplexity of the model on a held-out test set. Finally, you will implement some methods to deal with sparsity (zero count) issues in your model.

To ease you into the implementation, we will provide some boilerplate code that you would need to fill in depending upon the functionalities the code is supposed to perform. For the first few sections below, we will use the complete text from Leo Tolstoy's "War and Peace," which is freely available from Project Gutenberg. Run the following code block to download the text.

```
def download_data():
    def _download(url: str, filename: str) -> str:
        txt = urllib.request.urlopen(url)
        with open(filename, 'w') as f:
            f.write(txt.read().decode('utf-8'))

    _download('https://cs.stanford.edu/people/karpathy/char-rnn
    _download('http://www.gutenberg.org/files/1399/1399-0.txt',

download_data()
```

The complete text downloaded above contains punctuations which are not important for our purposes. So we will perform a basic text preprocessing using the NLTK toolkit. We will store the processed text into a list of strings, where each string will contain words without any punctuations. We will also convert all words to lower case.

```
# Loading the text
try:
    with open('warpeace_input.txt', 'r') as file:
        corpus_raw = file.read().replace('\n', ' ')
except FileNotFoundError:
    with open('../../warpeace_input.txt', 'r') as file:
        corpus_raw = file.read().replace('\n', ' ')
```

```
sentences = sent_tokenize(corpus_raw)

corpus = []
tokenizer = RegexpTokenizer(r'\w+')
for sentence in sentences:
    tokens = tokenizer.tokenize(sentence)
```

```
        corpus.append([token.lower() for token in tokens])

    print("Corpus has {} sentences".format(len(corpus)))
```

```
Corpus has 32040 sentences
```

Warm-up: n-gram counts from a corpus

Let's start with implementing a simple function for obtaining n-grams and their counts from a string. Complete the following function. *(5 points)*

Note 1: Use the special token `~` for both beginning of sentence (BOS) and end of sentence (EOS) tokens. For example, the sentence "Mary has a little lamb" has the bigrams "*~ Mary*", "*Mary has*", "*has a*", "*a little*", "*little lamb*", and "*lamb ~*".

Note 2: You don't need to do any further text processing beyond what has already been done before.

Note 3: For the usage of `collections.Counter`, you can refer to its python doc.

```
    def generate_ngrams(text: List[str], n: int) -> Counter:
        """Generates all n-grams (i.e. n-1 context words) for the

        Parameters
        ----------
        text : List[str]
            Input text (list of strings) after tokenization.
        n : int
            n-gram parameter (must be greater than or equal to 1)

        Returns
        -------
        ngrams : Counter
            Output n-grams dictionary as {ngram: count} (Dict[Tup
            where `ngram` is a n-gram tuple and `count` is an int
            e.g. ('Mary','has') and value as count of the n-gram
        """
        assert (isinstance(n, int) and n > 0)

        ngrams = Counter()

        # TODO: Your code here.
```

```
    ...
    ap_text = ['~'] * (n - 1) + text + ['~'] * (n - 1)
    for i in range(len(ap_text) - n + 1):
        ngrams[tuple(ap_text[i:i+n])] += 1

    return ngrams


def generate_ngrams_sentences(text: List[List[str]], n: int)
    """Generates n-grams for each sentence and aggregates the
    all_ngrams = Counter()
    for sentence in text:
        all_ngrams.update(generate_ngrams(sentence, n))
    return all_ngrams
```

```
# Test your implementation on some sentences from the corpus
# and verify if it works correctly.
text = corpus[:10]
print(text)

ngrams = generate_ngrams_sentences(text, 3)
print(ngrams)
```

```
[['well', 'prince', 'so', 'genoa', 'and', 'lucca', 'are', 'now', 'just', 'family
Counter({('~', '~', 'but'): 2, ('no', 'longer', 'my'): 2, ('~', '~', 'well'): 1,
```

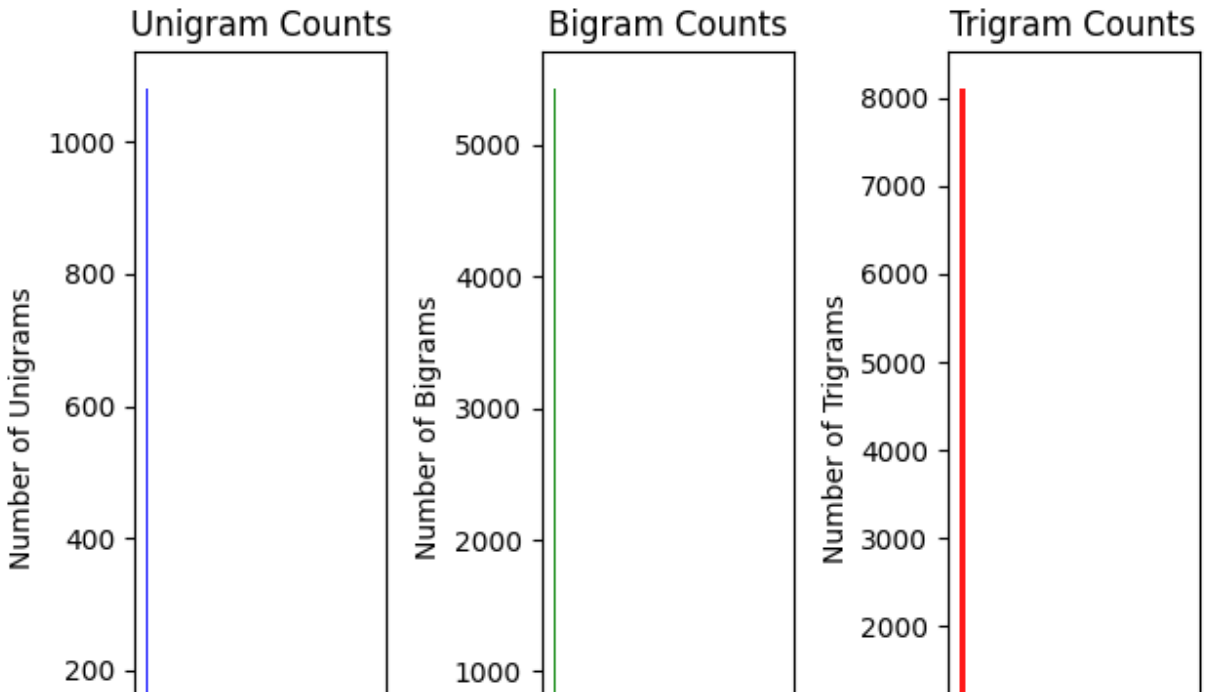
```
grader.check("warmup-ngram")
```

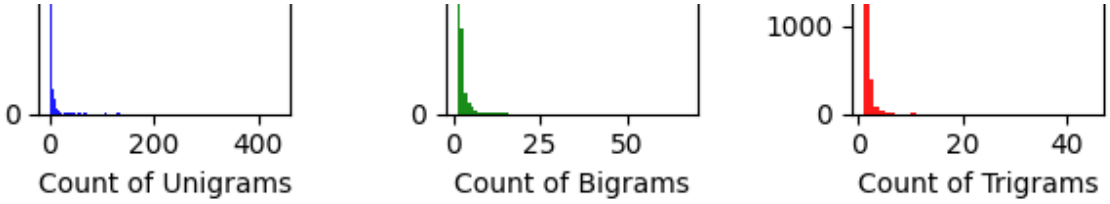
warmup-ngram passed!

Question: Plot a histogram of counts vs. number of unigrams with that count (you can choose a subset of the corpus, say 500 sentences). Repeat for bigrams and trigrams. They should be all placed into a single plot. *(3 points)*

```
fig, axs = plt.subplots(1, 3, tight_layout=True)

# Picks a subset of the corpus
text = corpus[:500]

# Unigram counts
unigrams = Counter()
```

```
unigrams.update(generate_ngrams_sentences(text, 1))
unigram_counts = list(unigrams.values())
axs[0].hist(unigram_counts, bins=range(1, max(unigram_counts) +
axs[0].set_title('Unigram Counts')
axs[0].set_xlabel('Count of Unigrams')
axs[0].set_ylabel('Number of Unigrams')

# Bigram counts
bigrams = Counter()
bigrams.update(generate_ngrams_sentences(text, 2))
bigram_counts = list(bigrams.values())
axs[1].hist(bigram_counts, bins=range(1, max(bigram_counts) + 1
axs[1].set_title('Bigram Counts')
axs[1].set_xlabel('Count of Bigrams')
axs[1].set_ylabel('Number of Bigrams')

# Trigram counts
trigrams = Counter()
trigrams.update(generate_ngrams_sentences(text, 3))
trigram_counts = list(trigrams.values())
axs[2].hist(trigram_counts, bins=range(1, max(trigram_counts) +
axs[2].set_title('Trigram Counts')
axs[2].set_xlabel('Count of Trigrams')
axs[2].set_ylabel('Number of Trigrams')

plt.show()
```





Question: What observation can you make from the plots? *(2 points)*

Answer:

The frequency of n-grams reduces as we go from unigrams to trigrams. The distribution spreads out with many of them having low frequencies. This could mean that individual words can be found more than longer sequences.

Implementing an n-gram language model

Next, you will implement a class for an trigram (n=3) language model. This will be a barebones trigram LM, i.e., no smoothing or OOV handling is required. Complete the functions in the following `NgramLM` class. *(20 points)*

Note that the class itself is for a general n-gram LM, but we will instantiate it with n=3.

Type your answer here, replacing this text.

```
class NgramLM(object):
    """A basic n-gram language model without any smoothing."""

    def __init__(self, n: int):
        assert (isinstance(n, int) and n > 0)

        self.n: int = n
        self.vocab: Set[str] = set()  # A set of all words ap
        self.ngrams = Counter()  # count(ABC) - Dict[Tuple, i
        self.ngram_contexts = Counter()  # count(AB) - Dict[T
        self.contexts = defaultdict(set)  # {AB: {C1,C2,C2}}

    def generate_ngrams(self, text: List[str]) -> Counter:
        """Generates all n-grams (i.e. n-1 context words) for
        In this method, we assume n is defined at the class i
```

```
        so you should use  self.n .

        Parameters
        ----------
        text : List[str]
            Input text (list of strings) after tokenization.

        Returns
        -------
        ngrams : Counter
            Output n-grams dictionary as {ngram: count} (Dict
            where `ngram` is a n-gram tuple and `count` is an
            e.g. ('Mary','has') and value as count of the n-g
        """
        return generate_ngrams(text, self.n)

    def generate_ngrams_sentences(self, text: List[List[str]]
        """Generates n-grams for each sentence and aggregates
        return generate_ngrams_sentences(text, self.n)

    def update(self, text: List[str]):
        """Updates the model n-grams based on the given text

        Parameters
        ----------
        text : List[str]
        """
        # TODO: Your code here
        ngrams = self.generate_ngrams(text)
        for ngram,count in ngrams.items():
          self.ngrams[ngram] += count
          context = ngram[:-1]
          word = ngram[-1]
          self.ngram_contexts[context] += count
          self.contexts[context].add(word)
          self.vocab.update(ngram)

    def word_prob(self, context: Tuple[str], word: str) -> fl
        """Returns the probability of a word given a context.
        string of words, with length n-1.

        Parameters
```

```
        ----------
        context : Tuple[str]
            A tuple of words describing the context for the n
        word : str
            The next word that the probability is computed fo

        Returns
        -------
        prob : float
            The estimated probability of the next word given
        """
        # TODO: Your code here
        if context not in self.ngram_contexts or (context + (
              return 0.0
        return self.ngrams[context + (word,)] / self.ngram_co

    def next_word_candidates(self, context: Tuple[str]) -> Li
        """Generates a list of tokens based on the given cont
        later used as candidates for the next word prediction

        Parameters
        ----------
        context : Tuple[str]
            A tuple of words describing the context for the n

        Returns
        -------
        words : List[str]
            A list of candidate tokens for the next word.
        """
        if context not in self.contexts:
              return []
        return list(self.contexts[context])

    def random_word(self, context: Tuple[str]) -> str:
        """Generates a random word based on the given context

        Note:
        Please use a random function from `random` instead of
        otherwise, the autograder might not be happy.

        Parameters
```

```
        ----------
        context : Tuple[str]
            A tuple of words describing the context for the n

        Returns
        -------
        word : str
            One randomly drawn word from the distribution def
        """
        assert context in self.contexts, f'Encountered unseen

        potential_random_word = self.next_word_candidates(con
        occur_chance = [self.word_prob(context,i) for i in po
        return random.choices(potential_random_word, weights=

    def random_text(self, length: int) -> List[str]:
        """Generates random text of the specified word length

        Note:
        - The final word of the generated text *can* be "~" (
        - The generation should always start with "~" (BOS).
        - Only the number of starting "~" (BOS) is excluded f
        ending "~" (EOS) is still counted to the `length`. It
        draw a number of `length` samples.

        Note:
        Please use a random function from `random` instead of
        otherwise, the autograder might not be happy.

        Parameters
        ----------
        length : int
            The designated length to generate.

        """
        if length == 0:
          return []
        context = ('~',) * (self.n - 1)
        final_text = []
        for _ in range(length):
          next_word = self.random_word(context)
          final_text.append(next_word)
```

```
            final_text.append(next_word)
            context = (*context[1:], next_word)

        return final_text
```

```
# Training block for a trigram language model
trigramlm = NgramLM(3)
for sentence in corpus:
    trigramlm.update(sentence)
```

```
grader.check("ngramlm-impl")
```

ngramlm-impl passed!

Question: What is the size of the training data (number of tokens)? *(1 point)*

```
training_corpus_size = sum(len(tokens) for tokens in corpus)
training_corpus_size
```

```
571824
```

Question: What is the size of the vocabulary? *(1 point)*

```
trigramlm_vocab_size = len(trigramlm.vocab)
trigramlm_vocab_size
```

```
17511
```

Question: What is the optimal time complexity of computing ngrams? How long does the training take? *(2 points)*

Note: please use big-O notation to show the time complexity and explain variables involved accordingly.

Answer:

Time Complexity depends on number of tokens t and length of the n-gram n. Total time will be (t - n + 1) to get all the n-grams as we slide over all the tokens in the dataset and portion off size n. So, the time complexity is O(t) i.e., linear time. Training will also take

O(t) as updating the counts in hash map takes constant time.

```
# TODO: put the training time (in ms) of a trigram LM below.
import time
start = time.time()
trigramlm = NgramLM(3)
for sentence in corpus:
    trigramlm.update(sentence)
end = time.time()
trigram_training_time = end - start
trigram_training_time
```

```
3.5032804012298584
```

Question: How would the training time scale if you have a corpus containing 1 billion tokens? Is this training time reasonable? If not, can you think of ways to improve it? *(2 points)*

Answer:

The training time would scale linearly as time complexity is linear in nature. 1 billion tokens should be feasible but there could be memory utilization issues due to large number of tokens. We can use parallel processing (use multiple processors) to improve the performance. We can also use sampling to reduce the number of tokens.

Predicting/generating text using the trained LM

One of the applications of an LM is to automatically predict the next word given a context (such as in Smart Keyboards), or to generate a piece of text of a given length. Use the `random_word` and `random_text` functions you implemented earlier to answer the questions below. For full credit, you need to show how you arrived at the answer.

Question: Please implement the function that computes an empirical distribution for the next word prediction conditioning on a context. Consider the context "by her". You can generate a random word for a large number of times, say 1000, using this context and count how many times each word are generated to calculate its empirical probability accordingly. *(3 points)*

```
def
```

```
def compute_empirical_distribution(model: NgramLM, context: T
    """Computes an empirical distribution for the next word c

    Parameters
    ----------
    model : NgramLM
        A trained Ngram Language Model.
    context : Tuple[str]
        The context used to predict a next word.
    num_samples : int
        The number of samples to be drawn to compute the empi

    Returns
    -------
    emp_distr : Dict[str, float]
        An empirical distribution for the next word

    """
    wfreq = defaultdict(int)

    for i in range(num_samples):
        word = model.random_word(context)
        wfreq[word] += 1

    total_words = sum(wfreq.values())
    emp_distr = {word: count / total_words for word, count in

    return emp_distr
```

```
compute_empirical_distribution(trigramlm, ('by', 'her'), 1000
```

```
{'that': 0.023,
 'body': 0.012,
 '~': 0.082,
 'bare': 0.023,
 'friend': 0.019,
 'brother': 0.023,
 'husband': 0.089,
 'side': 0.03,
 'thoughts': 0.024,
 'serious': 0.027,
 'fullness': 0.022,
 'breadth': 0.02,
 'image': 0.018,
 'sister': 0.045,
 'father': 0.061
```

```
 'father': 0.061,
'smiles': 0.013,
'mother': 0.093,
'garments': 0.027,
'and': 0.023,
'own': 0.023,
'marriage': 0.024,
'massive': 0.031,
'look': 0.025,
'manners': 0.023,
'spiritual': 0.021,
'face': 0.014,
'daughter': 0.027,
'attitude': 0.035,
'feelings': 0.023,
'beauty': 0.015,
'self': 0.022,
'presence': 0.027,
'daughters': 0.016}
```

```
grader.check("ngramlm-empirical-distribution")
```

ngramlm-empirical-distribution passed!

Question: Does the empirical probability match the output of `word_prob(("by", "her"), "husband")`? Could you explain why it matches or not? Could you propose a way to measure how the empirical distribution differs from the theoretical distribution? *(3 points)*

Answer:

```
print(trigramlm.word_prob(('by', 'her'), 'husband'))
```

```
0.08888888888888889
```

No, it does not match. This is mainly because the model is trained on a small sample set. If the model is well-trained, then the empirical and theoritical probabilities will be same as they both utilize same data from n-grams. We can use KL divergence to measure how the distributions differ from each other.

Question: Generate a random text of length 100 words. Comment on the local and global semantics of the generated text. *(2 points)*

Answer:

The local semantics are good. They make sense gramatically. The global semantics are not so good because a 3-gram lacks long term context. The sentences are gibberish and nonsensical.

```
# TODO: Your code here
generated_text=trigramlm.random_text(100)
generated_text=' '.join(generated_text)
print(generated_text)  # Please make sure this prints out a `
```

```
the count s other gory leg ~ ~ and the different outlooks of these sounds ~ ~ sa
```

Question: Now train a 4-gram LM on the same data and generate a 100-word text again. Do you observe any differences between the outputs of the two models? *(2 points)*

Answer:

The 4-gram model produces better sounding and meaningful text locally and globally as compared to the trigram model. This is beacuse it has a larger context.

```
# You should train you 4-gram model here in the same way as t
# You should use the same `corpus` for training
qgramlm = NgramLM(4)
for sentence in corpus:
    qgramlm.update(sentence)
qgramlm
```

```
<__main__.NgramLM at 0x7ce373131a60>
```

```
# TODO: Your code here
qgram_generated_text=qgramlm.random_text(100)
qgram_generated_text=' '.join(qgram_generated_text)
print(qgram_generated_text)  # Please make sure this prints o
```

```
how can one see from there ~ ~ ~ well this is strange ~ ~ ~ let them have it ~ ~
```

Evaluating the LM: Perplexity

In the context of language modeling, perplexity measures how an LM predicts a sample

In the context of language modeling, perplexity measures how an LM predicts a sample. It is computed as the per word inverse probability of a held-out set:

$$Perplexity(W) = P(W_1 W_2 \dots W_N)^{-1/N}$$

Complete the following function which computes the perplexity of an ngram language model given the class object and a dataset (represented as a list of strings as done earlier). *(8 points)*

Note 1: You may assume that the text is normalized as done before, so no text processing is required in the function.

Note 2: Consider performing computations in the log domain to avoid underflow errors. Recall the log equalities:

$$P = 2^{\log_2 P}$$

$$\log(a_1 a_2 \dots a_N)^{1/N} = \frac{1}{N}(\log a_1 + \log a_2 + \dots + \log a_N)$$

```
def perplexity(model: NgramLM, data: List[List[str]]) -> floa
    """Function to compute perplexity of ngram LM.

    Parameters
    ----------
    model : NgramLM
        A class object denoted a trained `NgramLM`.
    text : List[List[str]]
        A list of sentences, where each sentence is a list of

    Returns
    -------
    perp : float
        Perplexity of the LM on given string.
    """
    c_log_prob = 0.0
    token_count = 0

    for sentence in data:
        n_gram_order = model.n - 1
        sentence = ['~'] * n_gram_order + sentence + ['~'] * 
        token_count += len(sentence) - n_gram_order

        for index in range(n_gram_order, len(sentence)):
            context = tuple(sentence[index - n_gram_order:ind
```

```
            target_word = sentence[index]
            word_probability = model.word_prob(context, targe

            if word_probability > 0:
                c_log_prob += math.log2(word_probability)
            else:
                c_log_prob += float('-inf')

    avg_log_prob = c_log_prob / token_count
    p = 2 ** (-avg_log_prob)

    return p
```

```
grader.check("ngramlm-perp-impl")
```

ngramlm-perp-impl passed!

Question: What is the perplexity of the model on the training corpus? *(1 point)*

```
# TODO: Your code here
trigram_perp_on_training = perplexity(trigramlm, corpus)
trigram_perp_on_training
```

```
7.118221877916317
```

```
grader.check("ngramlm-tri-perp-on-training")
```

ngramlm-tri-perp-on-training passed!

Question: What is the perplexity of the 4-gram LM you trained earlier on the training corpus? *(1 point)*

```
# TODO: Your code here
qgram_perp_on_training = perplexity(qgramlm, corpus)
qgram_perp_on_training
```

```
2.2828663871796584
```

```
grader.check("ngramlm-quad-perp-on-training")
```

ngramlm-quad-perp-on-training passed!

You will now use your above implementation to evaluate your model on a small held out development set from Leo Tolstoy's Anna Karenina. First we download and preprocess this data similar to how we did for the training set.

```
# Process the text file to get the contents of Chapter 1
try:
    with open('1399-0.txt', 'r') as file:
        dev_raw = file.read().replace('\n', ' ')
except FileNotFoundError:
    with open('../../1399-0.txt', 'r') as file:
        dev_raw = file.read().replace('\n', ' ')
```

```
pattern = "Chapter 1(.*)Chapter 2"
dev_ch1 = re.search(pattern, dev_raw).group(1)

sentences = sent_tokenize(dev_ch1)

dev_text = []
tokenizer = RegexpTokenizer(r'\w+')
for sentence in sentences:
    tokens = tokenizer.tokenize(sentence)
    dev_text.append([token.lower() for token in tokens])

print("Dev data has {} sentences".format(len(dev_text)))
```

```
Dev data has 15883 sentences
```

Question: Compute the perplexity of the 3-gram LM on the development set prepared above. *(1 points)*

```
# TODO: Your code here
trigram_perp_on_dev = perplexity(trigramlm, dev_text)
trigram_perp_on_dev
```

```
inf
```

```
grader.check("ngramlm-tri-perp-on-dev")
```

ngramlm-tri-perp-on-dev passed!

Question: What is the reason for this perplexity value? *(2 points)*

Answer:

This is caused when the language model encounters zero probabilities or unknown tokens. We can use smoothing techniques to improve on this.

Zeros and generalization

From the above, you would have realized that our trigram LM in the barebones setting is probably not robust enough to be deployed in general settings, due to the data sparsity problem. This problem is dealt with by using "smoothing" methods for unseen n-grams and the `<UNK>` token for OOV words.

In this section, you will implement Laplace (add-one) smoothing and use the `<UNK>` token for handling OOV words in the evaluation set. Complete the following class definition to achieve this. *(15 points)*

```
class NgramLMWithLaplaceSmoothing(NgramLM):
    """An n-gram language model with OOV handling and Laplace
    This class inherits all bahaviors from previous defined `N
    Please be careful with the implementations here as you hav
    with all interfaces.
    """

    def __init__(self, n: int):
        super(NgramLMWithLaplaceSmoothing, self).__init__(n=n

    def next_word_candidates(self, context: Tuple[str]) -> Li
        """Generates a list of tokens based on the given cont
        later used as candidates for the next word prediction

        Note: in this overriden version, you should deal with

        Parameters
        ----------
        context : Tuple[str]
            A tuple of words describing the context for the n
```

```
            A tuple of words describing the context for the n

        Returns
        -------
        words : List[str]
            A list of candidate tokens for the next word.
        """
        if context not in self.contexts:
            return ['<UNK>']
        return list(self.contexts[context])

    def word_prob(self, context: Tuple[str], word: str) -> fl
        """Returns the probability of a word given a context.
        string of words, with length n-1.

        Note: in this overriden version, you should deal with

        Parameters
        ----------
        context : Tuple[str]
            A tuple of words describing the context for the n
        word : str
            The next word that the probability is computed fo

        Returns
        -------
        prob : float
            The estimated probability of the next word given
        """
        if word not in self.vocab:
            word = '<UNK>'

        # Applying Laplace smoothing
        context_count = self.ngram_contexts.get(context, 0)
        ngram_count = self.ngrams.get(context + (word,), 0)

        vocab_size = len(self.vocab)
        prob = (ngram_count + 1) / (context_count + vocab_siz

        return prob
```

```
# Training block for a trigram language model with Laplace sm
```

```
trigramlm_laplace = NgramLMWithLaplaceSmoothing(3)
for sentence in corpus:
    trigramlm_laplace.update(sentence)
```

Question: Report the perplexity of the new LM on the development data. (3 points)

```
# TODO: Your code here
trigram_laplace_perp_on_training = perplexity(trigramlm_lapla
trigram_laplace_perp_on_training
```

```
7353.30230212652
```

Question: Can you think of a different way to solve the OOV problem? *(2 points)*

Answer:

We can use subword tokenization. Splitting of tokens will help to recognize part of a OOV word, so we need not issue a token.

Question (extra credit): Laplace smoothing is a relatively naive smoothing method. In the lectures, you learnt about more advanced methods: Good-Turing, Backoff, Interpolation, Kneser-Ney. Implement any one of these smoothing methods (pick your favorite). Evaluate the resulting trigram LM on the development data and report the perplexity. Could you get some improvements (an improvement from the baseline would secure you to get an extra credit of 10 points)? *(10 points)*

Leaderboard: It would be interesting to explore different techniques or combinations of techniques to improve trigram language models. For this extra credit question, we also introduce a leaderboard for people to compete their designs of language models. To encourage participantion, the top 15% and 30% would be given another 10 points and 5 points extra credit respectively. Have fun :-)

```
class ImprovedNgramLM(NgramLM):
    """An improved version of n-gram language model with OOV
    This class inherits all bahaviors from previous defined `
    Please be careful with the implementations here as you ha
    with all interfaces.

    Note: you could override more methods provided by the bas
```

```
    to maintain their ability of handling the same types of i
    """

    def __init__(self, n: int):
        super(ImprovedNgramLM, self).__init__(n=n)

        # TODO: Your code here, if you would like to change t
        if "<UNK>" not in self.vocab:
            self.vocab.add("<UNK>")

    def _handle_oov(self, word: str) -> str:
        """Replace OOV words with <UNK>."""
        return word if word in self.vocab else "<UNK>"

    def next_word_candidates(self, context: Tuple[str]) -> Li
        """Generates a list of tokens based on the given cont
        later used as candidates for the next word prediction

        Note: in this overriden version, you should deal with

        Parameters
        ----------
        context : Tuple[str]
            A tuple of words describing the context for the n

        Returns
        -------
        words : List[str]
            A list of candidate tokens for the next word.
        """
        # TODO: Your code here
        clean_context = tuple(self._handle_oov(w) for w in co
        candidates = super().next_word_candidates(clean_conte

        if "<UNK>" not in candidates:
            candidates.append("<UNK>")

        return candidates


    def word_prob(self, context: Tuple[str], word: str) -> fl
        """Returns the probability of a word given a context.
```

```
        string of words, with length n-1.

        Note: in this overriden version, you should deal with

        Parameters
        ----------
        context : Tuple[str]
            A tuple of words describing the context for the n
        word : str
            The next word that the probability is computed fo

        Returns
        -------
        prob : float
            The estimated probability of the next word given
        """
        # TODO: Your code here
        clean_context = tuple(self._handle_oov(w) for w in co
        clean_word = self._handle_oov(word)

        candidates = self.next_word_candidates(clean_context)

        # Get base probs from parent LM
        base_probs = {}
        for w in candidates:
            p = super().word_prob(clean_context, w)
            base_probs[w] = max(p, 0.0)  # avoid negatives ju

        # Add-ε smoothing to ensure nonzero prob everywhere
        eps = 1e-6
        smoothed = {w: base_probs[w] + eps for w in candidate

        # Normalize so distribution sums to 1
        Z = sum(smoothed.values())
        probs = {w: smoothed[w] / Z for w in candidates}

        return probs.get(clean_word, probs["<UNK>"])
```

```
# Training block for a trigram language model
improved_trigramlm = ImprovedNgramLM(3)
```

```
for sentence in corpus:
    improved_trigramlm.update(sentence)
```

```
grader.check("ngramlm-improvement-impl")
```

Part 2: Parsing and the CYK algorithm

In the lecture on Syntax, you learnt about parsing algorithms, including the bottom-up CYK algorithm. In this section, you will implement the CYK algorithm for computing the parse tree of a sentence given a grammar.

You may look at the pseudocode on Wikipedia or refer to descriptions of the CYK algorithm online (such as this), but you may not copy code directly from another source. The objective of this exercise is to familiarize yourself with parsing.

First, we will provide some starter code to load a simple grammar which can be used to test your implementation. The CYK algorithm only works with context-free grammars (CFGs) in the Chomsky Normal Form (CNF), but any CFG can be represented as an equivalent CNF. You can use NLTK to check if the grammar is in CNF.

```
# grammar rules

cfg_rules = """
S -> NP VP
PP -> P NP
NP -> Det N
NP -> Det N PP
NP -> 'I'
VP -> V NP
VP -> VP PP
Det -> 'an'
Det -> 'my'
N -> 'elephant'
N -> 'pajamas'
V -> 'shot'
P -> 'in'
"""
```

Question: Use NLTK to check if the grammar `cfg` is in the Chomsky Normal Form. *(1 point)*

```
import nltk
from nltk import CFG
cfg_rules = """
S -> NP VP
PP -> P NP
NP -> Det N
NP -> Det N PP
NP -> 'I'
VP -> V NP
VP -> VP PP
Det -> 'an'
Det -> 'my'
N -> 'elephant'
N -> 'pajamas'
V -> 'shot'
P -> 'in'
"""

# Note:
# `is_cfg_cnf` should be the function name without executing
# Executing this cell should give you a boolean result indica
# is in the Chomsky Normal Form.
grammar = CFG.fromstring(cfg_rules)
is_cfg_cnf =  grammar.is_chomsky_normal_form()
print(is_cfg_cnf)
```

```
False
```

Question: Convert the above CFG into CNF (use pen and paper) and create a new grammar using it. Use NLTK to verify if it is in CNF. *(4 points)*

Here are the steps to convert any CFG into a CNF:

1. Eliminate start symbol from the RHS. If the start symbol S is at the right-hand side of any production, create a new production as: S1 -> S
2. If CFG contains null, unit or useless production rules, eliminate them.
3. Eliminate terminals from RHS if they exist with other terminals or non-terminals.
4. Eliminate RHS with more than two non-terminals.

4. Eliminate RHS with more than two non-terminals.

(Hint: There is only one offending rule in the above grammar.)

```
# Write the CNF grammar here as a string

cnf_cfg_rules = """
S -> NP VP
PP -> P NP
NP -> Det N
N1 -> N PP
NP -> 'I'
VP -> V NP
VP -> VP PP
Det -> 'an'
Det -> 'my'
N -> 'elephant'
N -> 'pajamas'
V -> 'shot'
P -> 'in'
"""

# TODO: Your code here
cnf_grammar = CFG.fromstring(cnf_cfg_rules)

is_cnf = cnf_grammar.is_chomsky_normal_form()

print(is_cnf)
```

```
True
```

You can now use the above grammar and the sentence: *"I shot an elephant in my pajamas"* to demonstrate your implementation of the CYK parser.

Complete the following code block to implement the parser. We have provided the definition of the Node class which stores a non-terminal, and some boilerplate code to ease you into the implementation. Your main task is to implement the `parse()` function, which generates the parse table in a bottom-up manner. The `parse_table` in the `CYKParser` class below can be thought of as a table which contains number of rows equal to the number of words in the sentence. *(25 points)*

Note:

We recommend reading the usage of NLTK grammar object as we would parse the

We recommend reading the usage of NLTK grammar object as we would parse the grammar to this object. It allows us to easily play with different grammar production rules. Sample usages (`grammar` is a `nltk.grammar.CFG` object instantiated from a given grammar):

- `grammar.productions()`: list all production rules defined in the grammar.
- `rule_i = grammar.productions()[i]`: get the ith rule from the grammar.
- `rule_i.lhs()`: get the LHS for the given rule.
- `rule_i.rhs()`: get the RHS for the given rule.
- `rule_i.is_lexical()`: determine whether it is a terminal rule.
- `rule_i.is_nonlexical()`: determine whether it is a non-terminal rule.
- `rhs_0 = rule_i.rhs()[0]`: get the first element on the RHS for the given rule.
- `rhs_0.symbol()`: get the `str` symbol of the RHS element (this also applies to the LHS element).

Hint: It may be beneficial to first run through the algorithm for the given grammar and the sentence on pen and paper.

```
    class Node:
        """ Equivalent to a non-terminal. Since our grammar is CN
        most 2 children. Following 2 cases are possible:

        Case 1 -> child1 is a terminal symbol
        Case 2 -> both child1 and child2 are Nodes.
        """

        def __init__(self, symbol, child1, child2=None):
            self.symbol = symbol
            self.child1 = child1
            self.child2 = child2

        def __repr__(self):
            """Returns the string representation of a Node object
            return self.symbol

        def generate_tree(self) -> str:
            """Generates the string representation of the tree ro
            It is done via pre-order tree traversal.

            Returns
            -------
```

```
            str_tree : str
                The tree in its string form.
            """
            if self.child2 is None:
                return f"[{self.symbol} '{self.child1}']"
            return f"[{self.symbol} {self.child1.generate_tree()}


    class CYKParser(object):
        """A CYK parser which is able to parse any grammar in CNF
        is created from a CNF grammar and can be used to parse an
        """

        def __init__(self, grammar: str):
            """Creates a new parser object.

            Parameters
            ----------
            grammar : str
                Input grammar as a string of rules.
            """
            self.grammar: nltk.grammar.CFG = nltk.grammar.CFG.fro

        def print_tree(self, parse_table: List[List[List[Node]]])
            """Prints the parse tree starting with the start symb
            """
            start_symbol = self.grammar.start().symbol()
            final_nodes = [n for n in parse_table[-1][0] if n.sym
            if final_nodes:
                print("\nPossible parse(s):")
                trees = [node.generate_tree() for node in final_n
                for tree in trees:
                    print(tree)
            else:
                print("The given sentence is not contained in the

        def parse(self, sentence: List[str]) -> List[List[List[No
            """Does the actual parsing according to the CYK algor

            Parameters
            ----------
            sentence : List[str]
```

```
            An input sentence in the form of list of tokens.

        Returns
        -------
        parse_table : List[List[List[Node]]]
            The resulting parse table for the sentence under t
        """
        num_tokens = len(sentence)
        # parse_table[y][x] is the list of nodes in the x+1 c
        # of y+1 row in the table. That cell covers the word l
        # and y more words after.
        parse_table: List[List[List[Node]]] = [[[] for x in r

        for i, word in enumerate(sentence):
            for rule in self.grammar.productions():
                if rule.is_lexical() and rule.rhs()[0] == wor
                    parse_table[0][i].append(Node(rule.lhs().
        for length in range(2, num_tokens + 1):
            for start in range(num_tokens - length + 1):
                for split in range(1, length):
                    left_cell = parse_table[split - 1][start]
                    right_cell = parse_table[length - split -

                    for rule in self.grammar.productions():
                        if rule.is_nonlexical() and len(rule.
                            left_symbol = rule.rhs()[0].symbo
                            right_symbol = rule.rhs()[1].symb
                            for left_node in left_cell:
                                for right_node in right_cell:
                                    if left_node.symbol == le
                                        parse_table[length -
        return parse_table
```

```
parser = CYKParser(cnf_cfg_rules)
parse_table = parser.parse("I shot an elephant in my pajamas"
```

```
parser.print_tree(parse_table)
```

```
Possible parse(s):
[S [NP 'I'] [VP [VP [V 'shot'] [NP [Det 'an'] [N 'elephant']]] [PP [P 'in'] [NP
```

```
grader.check('cyk-impl')
```