In [ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

In [ ]:
```
# Please do not change this cell because some hidden tests might depend on i
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls tests >/dev/null 2>&1
if [ ! $? = 0 ]; then
 # Check if the repository exists before trying to clone it
 if git ls-remote https://github.com/cs187-2025/lab2-1.git >/dev/null 2>&1;
   rm -rf .tmp
   git clone https://github.com/cs187-2025/lab2-1.git .tmp
   mv .tmp/tests ./
   if [ -f .tmp/requirements.txt ]; then
     mv .tmp/requirements.txt ./
   fi
   rm -rf .tmp
 else
   echo \"Repository https://github.com/cs187-2025/lab2-1.git does not exist
 fi
fi
if [ -f requirements.txt ]; then
  python -m pip install -q -r requirements.txt
fi
""")
```

%%latex \newcommand{\vect}[1]{\mathbf{#1}} \newcommand{\cnt}[1]{\sharp(#1)} \newcommand{\argmax}[1]{\underset{#1}{\operatorname{argmax}}} \newcommand{\softmax}{\operatorname{softmax}} \newcommand{\Prob}{\Pr} \newcommand{\given}{\,|\,}

We turn from tasks that *classify* texts – mapping texts into a finite set of classes – to tasks that *model* texts by providing a full probability distribution over texts (or providing a similar scoring metric). Such *language models* attempt to answer the question "How likely is a token sequence to be generated as an instance of the language?".

In this lab, you'll construct and apply a particularly simple kind of language model, the n-gram language model.

After this lab, you should be able to

- Construct an n-gram language model based on counts of the n-grams in a training text.
- Apply an n-gram model to compute a text's probability and its perplexity.
- Sample a text from the probability distribution defined by an n-gram language model.
- Smooth an n-gram model to eliminate zero-probability n-grams using one of a small set of methods: add-δ, interpolation, and backoff smoothing.

We'll start with n-gram language models. Given a token sequence $\vectw = w_1, w_2, \ldots, w_N$, its probability $\Prob(w_1, w_2, \ldots, w_N)$ can be calculated using the chain rule of probability:

$$\Prob(A, B \given \theta) = \Prob(A \given \theta) \cdot \Prob(B \given A, \theta)$$

Thus,

$$\begin{aligned}
\Prob(w_1, w_2, \ldots, w_N) &= \Prob(w_1) \cdot \Prob(w_2, \ldots, w_N \given w_1) \\
&= \Prob(w_1) \cdot \Prob(w_2 \given w_1) \cdot \Prob(w_3, \ldots, w_N \\
&\cdots \\
&= \prod_{i=1}^{N} \Prob(w_i \given w_1, \cdots, w_{i-1}) \\
&\approx \prod_{i=1}^{N} \Prob(w_i \given w_{i-n+1}, \cdots, w_{i-1})
\end{aligned}$$

In the last step, we replace the probability $\Prob(w_i \given w_1, \cdots, w_{i-1})$, which conditions w_i on *all* of the preceding tokens, with $\Prob(w_i \given w_{i-n+1}, \cdots, w_{i-1})$, which conditions w_i only on the $n-1$ preceding tokens. We call the $n-1$ preceding tokens $(w_{i-n+1}, \cdots, w_{i-1})$ the *context* and w_i the *target*. Taken together, these n tokens form an n-gram, hence the term *n-gram model*.

In this lab you'll work with n-gram models: generating them, sampling from them, and scoring held-out texts according to them. We'll find some problems with n-gram models as language models:

1. They are profligate with memory.
2. They are sensitive to very limited context.
3. They don't generalize well across similar words.

In the next lab, we'll explore neural models to address these failings.

New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- `itertools.product`
- `list`
- `tuple`

Preparation – Loading packages and data

In [ ]:
```
import itertools
import math
import random
import re
import wget

import nltk
nltk.download('punkt', quiet=True) # download tokenizer module

from collections import defaultdict, Counter
from sys import getsizeof

# Set random seeds
SEED = 1234
random.seed(SEED)
```

In [ ]:
```
# Some utilities to manipulate the corpus

def preprocess(text):
    """Strips #comments and empty lines from a string
    """
    result = []
    for line in text.split("\n"):
        line = line.strip()              # trim whitespace
        line = re.sub('#.*$', '', line)  # trim comments
        if line != '':                   # drop blank lines
            result.append(line)
    return result

def nltk_normpunc_tokenize(str):
    return nltk.tokenize.word_tokenize(str.lower())

def geah_tokenize(lines):
    """Specialized tokenizer for GEaH corpus handling speaker IDs"""
    result = []
    for line in lines:
        # tokenize
        tokens = nltk_normpunc_tokenize(line)
        # revert the speaker ID token
        if tokens[0] == "sam":
            tokens[0] = "SAM:"
        elif tokens[0] == "guy":
            tokens[0] = "GUY:"
```

```
        else:
            raise ValueError("format problem - bad speaker ID")
        # add a start of sentence token
        result += ["<s>"] + tokens
    return result

def postprocess(tokens):
    """Converts `tokens` to a string with one sentence per line"""
    return ' '.join(tokens)\
              .replace("<s> ", "\n")

# Read the GEaH data and preprocess into training and test streams of tokens
geah_filename = ("https://github.com/nlp-course/data/raw/master/Seuss/"
                 "seuss - 1960 - green eggs and ham.txt")
os.makedirs('data', exist_ok=True)
wget.download(geah_filename, out="data/")

def split(list, portions, offset):
    """Splits `list` into a "large" and a "small" part, returning them as a

    The two parts are formed by partitioning `list` into `portions` disjoint
    The small part is the piece at index `offset`; the large part is the rem
    """
    return ([list[i] for i in range(0, len(list)) if i % portions != offset]
            [list[i] for i in range(0, len(list)) if i % portions == offset]

with open("data/seuss - 1960 - green eggs and ham.txt", 'r') as fin:
    lines = preprocess(fin.read())
    train_lines, test_lines = split(lines, 12, 0)
    train_tokens = geah_tokenize(train_lines)
    test_tokens = geah_tokenize(test_lines)
```

We've already loaded in the text of *Green Eggs and Ham* for you and split it (about 90%/10%) into two token sequences, `train_tokens` and `test_tokens`. Here's a preview:

In [ ]:
```
print(train_tokens[:50])
print(postprocess(train_tokens[:50]))
```

In [ ]:
```
print(test_tokens[:50])
print(postprocess(test_tokens[:50]))
```

We extract the vocabulary from the training text.

In [ ]:
```
# Extract vocabulary from dataset
vocabulary = list(set(train_tokens))
print(vocabulary)
```

Generating n-grams

The *n-grams* in a text are the contiguous subsequences of n tokens. (We'll implement them as Python tuples.) In theory, any sequence of n tokens is a potential n-gram type. Let's generate a list of all the possible n-gram types over a vocabulary. (Notice how the type/token distinction is useful for talking about n-grams, just as it is for words.)

In [ ]:
```
#TODO
def all_ngrams(vocabulary, n):
    """Returns a list of all `n`-long *tuples* of elements of the `vocabular

    For instance,

        >>> all_ngrams(["one", "two"], 3)
        [('one', 'one', 'one'),
         ('one', 'one', 'two'),
         ('one', 'two', 'one'),
         ('one', 'two', 'two'),
         ('two', 'one', 'one'),
         ('two', 'one', 'two'),
         ('two', 'two', 'one'),
         ('two', 'two', 'two')]

    Order of returned list is not specified or guaranteed.
    When `n` is 0, returns `[()]`.
    """
    ...
```

In [ ]:
```
grader.check("all_ngrams")
```

We can generate a list of all of the n-grams (tokens, not types) in a text.

In [ ]:
```
def ngrams(tokens, n):
    """Returns a list of all `n`-gram instances in a list of `tokens`, in or

    For instance,

    >>> ngrams(nltk_normpunc_tokenize('I am Sam! Sam I am.'), 3)
    [('i', 'am', 'sam'),
     ('am', 'sam', '!'),
     ('sam', '!', 'sam'),
     ('!', 'sam', 'i'),
     ('sam', 'i', 'am'),
     ('i', 'am', '.')]
    """
    return [tuple(tokens[i : i + n])
            for i in range(0, len(tokens) - n + 1)]
```

In [ ]:
```
print (train_tokens[:6])
print (ngrams(train_tokens[:6], 3))
```

Counting n-grams

We conceptualize an n-gram as having two parts:

- The *context* is the first $n-1$ tokens in the n-gram.
- The *target* is the final token in the n-gram.

An n-gram language model specifies a probability for each n-gram type. We'll implement a model as a 2-D dictionary, indexed first by context and then by target, providing the probability for the n-gram.

We start by generating a similar data structure -- indexed first by context and then by target -- but for counts instead of probabilities. That is, it should store for each n-gram type the count of how many times it occurs in a given token sequence. It should contain counts for *every* possible n-gram type (including those with a zero count).

In [ ]:
```
#TODO
def ngram_counts(vocabulary, tokens, n):
    """Returns a dictionary of counts of the `n`-grams over the `vocabulary`
    appearing in `tokens`.

    The dictionary is structured with first index by (n-1)-gram context
    and second index by the final target token.

    For instance,

    >>> ngram_counts(["a","b","c"], nltk_normpunc_tokenize('a b c a b'), 2)
    defaultdict(<function __main__.ngram_counts.<locals>.<lambda>()>,
                {('a',): defaultdict(int, {'a': 0, 'b': 2, 'c': 0}),
                 ('b',): defaultdict(int, {'a': 0, 'b': 0, 'c': 1}),
                 ('c',): defaultdict(int, {'a': 1, 'b': 0, 'c': 0})})
    """
    ...
```

In [ ]:
```
grader.check("ngram_counts")
```

Use the `ngram_counts` function to generate count data structures for unigrams, bigrams, and trigrams for the *Green Eggs and Ham* training text.

In [ ]:
```
#TODO
unigram_counts = ...
bigram_counts = ...
trigram_counts = ...
```

In [ ]:
```
grader.check("ngram_counts_geah")
```

Check your work by examining the total count of unigrams, bigrams, and trigrams. Do the numbers make sense?

In [ ]:
```
# Calculate total counts of tokens, unigrams, bigrams, and trigrams
token_count = len(train_tokens)
unigram_count = sum(len(unigram_counts[cntxt]) for cntxt in unigram_counts)
bigram_count = sum(len(bigram_counts[cntxt]) for cntxt in bigram_counts)
```

```
trigram_count = sum(len(trigram_counts[cntxt]) for cntxt in trigram_counts)

# Report on the totals
print(f"Tokens:   {token_count:6}\n"
      f"Unigrams: {unigram_count:6}\n"
      f"Bigrams:  {bigram_count:6}\n"
      f"Trigrams: {trigram_count:6}")
```

Calculating n-gram probabilities

We can convert the counts into a probability model by *normalizing* the counts. Given an n-gram type $w_1, w_2, \ldots, w_n$, instead of storing the count $\sharp(w_1, w_2, \ldots, w_n)$, we store an estimate of the probability

$$\begin{aligned}\Pr(w_n \mid w_1, w_2, \ldots, w_{n-1}) &\approx \frac{\sharp(w_1, w_2, \ldots, w_n)}{\sharp(w_1, w_2, \ldots, w_{n-1})} \\ &= \frac{\sharp(w_1, w_2, \ldots, w_n)}{\sum_{w'} \sharp(w_1, w_2, \ldots, w_{n-1}, w')}\end{aligned}$$

that is, the ratio of the count of the n-gram and the sum of the counts of all n-grams with the same context. Fortunately, all of those counts are already stored in the count data structures we've already built.

Write a function that takes an n-gram count data structure and returns an n-gram probability data structure. As with the counts, the probabilities should be stored indexed first by context and then by target.

In [ ]:
```
#TODO
def ngram_model(ngram_counts):
    """Returns an n-gram probability model calculated by normalizing the
       provided `ngram-counts` dictionary
    """
    ...
```

In [ ]:
```
grader.check("ngram_model")
```

We can now build some n-gram models – unigram, bigram, and trigram – based on the counts.

In [ ]:
```
unigram_model = ngram_model(unigram_counts)
bigram_model = ngram_model(bigram_counts)
trigram_model = ngram_model(trigram_counts)
```

Space considerations

For the most part, we aren't too concerned about matters of time or space efficiency, though these are crucial issues in the engineering of NLP systems. But the size of n-gram models merits consideration, looking especially at their size as n grows. We can use Python's `sys.getsizeof` function to get a rough sense of the size of the models we've been working with.

In [ ]:
```
print(f"Tokens:   {getsizeof(train_tokens):6}\n"
      f"Unigrams: {getsizeof(unigram_model):6}\n"
      f"Bigrams:  {getsizeof(bigram_model):6}\n"
      f"Trigrams: {getsizeof(trigram_model):6}")
```

Question: What do these sizes tell you about the memory usage of n-gram models? With a larger vocabulary of, say, 10,000 word types, would it be practical to run, say, 5-gram models on your laptop?

Type your answer here, replacing this text.

Sampling from an n-gram model

We have cleverly constructed the models to index by context. This allows us to sample a word given its context. For instance, in the trigram context `("<s>", "SAM:")`, the following probability distribution captures which words can come next and with what probability:

In [ ]:
```
trigram_model[("<s>", "SAM:")]
```

We can sample a single token according to this probability distribution. Here's one way to do so.

In [ ]:
```
def sample(model, context):
    """Returns a token sampled from the `model` assuming the `context`"""
    distribution = model[context]
    prob_remaining = random.random()
    for token, prob in sorted(distribution.items()):
        if prob_remaining < prob:
            return token
        else:
            prob_remaining -= prob
    raise ValueError
```

We can extend the sampling to a sequence of words by updating the context as we sample each word.

Define a function `sample_sequence` that performs this sampling of a sequence. It's given a model and a starting context and begins by sampling the first token based on the starting context, then updates the starting context to reflect the word just sampled, repeating the process until a specified number of tokens have been sampled.

> Hint: You might find function `list` helpful for converting immutable tuples to lists, and conversely `tuple` helpful for converting lists to tuples.

In [ ]:
```
#TODO
def sample_sequence(model, start_context, count=100):
    """Returns a sequence of `count` tokens sampled successively
       from the `model` *following the `start_context`*.
       The length of the returned list should be `count+len(start_context)`.
    """
    random.seed(SEED) # for reproducibility, do not change
    ...
```

In [ ]:
```
grader.check("sample_sequence")
```

Let's try it.

In [ ]:
```
print(postprocess(sample_sequence(unigram_model, ())))
```

In [ ]:
```
print(postprocess(sample_sequence(bigram_model, ("<s>",))))
```

In [ ]:
```
print(postprocess(sample_sequence(trigram_model, ("<s>", "SAM:"))))
```

Evaluating text according to an n-gram model

The probability metric

The main point of a language model is to assign probabilities (or similar scores) to texts. For n-gram models, that's done according to Equation (1) at the start of the lab. Let's implement that. We define a function `probability` that takes a token sequence and an n-gram model (and the n of the model as well) and returns the probability of the token sequence according to the model. It merely multiplies all of the n-gram probabilities for all of the n-grams in the token sequence.

> Throughout this lab, we ignore the scores of the first $n-1$ tokens as our n-gram model cannot score them due to the lack of context. In the next lab you will see how to solve this issue in practice.

In [ ]:
```
def probability(tokens, model, n):
    """Returns the probability of a sequence of `tokens` according to an
       `n`-gram `model`
    """
    score = 1.0
    context = tokens[0:n-1]
    # Ignores the scores of the first n-1 tokens
```

```
    for token in tokens[n-1:]:
        prob = model[tuple(context)][token]
        score *= prob
        context = (context + [token])[1:]
    return score
```

We test it on the test text that we held out from the training text.

In [ ]:
```
print("Test probability - unigram: "
          f"{probability(test_tokens, unigram_model, 1):6e}\n"
      "Test probability -  bigram: "
          f"{probability(test_tokens, bigram_model, 2):6e}\n"
      "Test probability - trigram: "
          f"{probability(test_tokens, trigram_model, 3):6e}")
```

The negative log probability metric

Yikes, those probabilities are *really small*. Multiplying all those small numbers is likely to lead to underflow.

To solve the underflow problem, we'll do our usual trick of using log probabilities (in this case, *negative* log probabilities) instead of probabilities:

$$-\log_2\left(\prod_{i=1}^{N}\Pr(w_i \mid w_{i-n+1},\cdots,w_{i-1})\right)$$

Define a function `neglogprob` that takes a token sequence and an n-gram model (and the n of the model as well) and returns the negative log probability of the token sequence according to the model, calculating it in such a way as to avoid underflow. (You'll want to simplify the formula above before implementing it.)

> Be careful when confronting zero probabilities. Taking `-math.log2(0)` raises a "Math domain error". Instead, you should use `math.inf` (Python's representation of infinity) as the value for the negative log of zero. This accords with our understanding that an impossible event would require infinite bits to specify.

In [ ]:
```
#TODO
def neglogprob(tokens, model, n):
    """Returns the negative log probability of a sequence of `tokens`
       according to an `n`-gram `model`
    """
    ...
```

In [ ]:
```
grader.check("neglogprob")
```

We compute the negative log probabilities of the test text using the different models and report on them.

In [ ]:
```
unigram_test_nlp = neglogprob(test_tokens, unigram_model, 1)
bigram_test_nlp = neglogprob(test_tokens, bigram_model, 2)
trigram_test_nlp = neglogprob(test_tokens, trigram_model, 3)

print(f"Test neglogprob - unigram: {unigram_test_nlp:6f}\n"
      f"Test neglogprob -  bigram: {bigram_test_nlp:6f}\n"
      f"Test neglogprob - trigram: {trigram_test_nlp:6f}")
```

There, those numbers seem more manageable. We can even convert the neglogprobs back into probabilities as a sanity check.

In [ ]:
```
print(f"Test probability - unigram: {2 ** (-unigram_test_nlp):6e}\n"
      f"Test probability -  bigram: {2 ** (-bigram_test_nlp):6e}\n"
      f"Test probability - trigram: {2 ** (-trigram_test_nlp):6e}")
```

Question: Why does the bigram model assign a lower neglogprob (that is, a higher probability) to the test text than the unigram model? Why does the trigram model assign a higher neglogprob (lower probability) to the test text than the other models?

Type your answer here, replacing this text.

The perplexity metric

Another metric that is commonly used is *perplexity*. Jurafsky and Martin give a definition for perplexity as the "inverse probability of the test set normalized by the number of words":

$$PP(x_1, x_2, \ldots, x_N) = \sqrt[N]{\frac{1}{\prod_{i=1}^{N} \Pr(x_i \mid x_{i-n+1}, \cdots, x_{i-1})}}$$

Define a function `perplexity` that takes a token sequence and an n-gram model (and the n of the model as well) and returns the perplexity of the token sequence according to the model, calculating it in such a way as to avoid underflow. (By now you're smart enough to realize that you'll want to carry out most of that calculation inside a $\log$.)

> Remember that we ignored the scores of the first n-1 tokens. What should the number of words `N` be?

In [ ]:
```
#TODO
def perplexity(tokens, model, n):
    """Returns the perplexity of a sequence of `tokens` according to an
       `n`-gram `model`
    """
    ...
```

In [ ]:
```
grader.check("perplexity")
```

We can look at the perplexity of the test sample according to each of the models.

In [ ]:
```
print("Test perplexity - unigram: "
          f"{perplexity(test_tokens, unigram_model, 1):.3f}\n"
      "Test perplexity -  bigram: "
          f"{perplexity(test_tokens, bigram_model, 2):.3f}\n"
      "Test perplexity - trigram: "
          f"{perplexity(test_tokens, trigram_model, 3):.3f}")
```

A perplexity value of P can be interpreted as a measure of a model's average uncertainty in selecting each word, equivalent to selecting among P equiprobable words on average. The bigram model gives a perplexity of less than 3, indicating that at each word in the sentence, the model is acting as if it is selecting among (slightly less than) three equiprobable words.

For comparison, state of the art n-gram language models for more representative English text achieve perplexities of about 250.

Smoothing n-gram language models

> **This section is more open-ended in nature. Feel free to experiment.**

The models we've been using have lots of zero-probability n-grams. Essentially any n-gram that doesn't appear in the training text is imputed a probability of zero, which means that any sentence that contains that n-gram will also be given a zero probability. Clearly this is not an accurate estimate.

There are many ways to *smooth* n-gram models, just as you smoothed classification models in earlier labs. The simplest is probably add-δ smoothing.

$$\Pr(w_i \mid w_1 \dots w_{i-1}) \approx \frac{\sharp(w_1, w_2, \dots, w_n) + \delta}{\sharp(w_1, w_2, \dots, w_{n-1}) + \delta \cdot |V|}$$

Another useful method is to interpolate multiple n-gram models, for instance, estimating probabilities as an interpolation of trigram, bigram, and unigram models.

$$\Pr(w_i \mid w_1 \dots w_{i-1}) \approx \lambda_2 \Pr(w_i \mid w_{i-2}, w_{i-1}) + \lambda_1 \Pr(w_i \mid w_{i-1}) + (1 - \lambda_1 - \lambda_2) \Pr(w_i$$

Finally, a method called *backoff* uses higher-order n-gram probabilities where available, "backing off" to lower order where necessary.

$$\Pr(w_i \mid w_1 \dots w_{i-1}) \approx \begin{cases} \Pr(w_i \mid w_{i-2}, w_{i-1}) & \text{if } \Pr(w_i \mid w_{i-2}, w_{i-1}) > 0 \\ \Pr(w_i \mid w_{i-1}) & \text{if } \Pr(w_i \mid w_{i-2}, w_{i-1}) = 0 \text{ and } \Pr(w_i \mid w_i \\ \Pr(w_i) & \text{otherwise} \end{cases}$$

Define a function `ngram_model_smoothed`, like the `ngram_model` function from above, but implementing one of these smoothing methods. Compare its perplexity on

some sample text to the unsmoothed model.

In [ ]:
```
"""
#TODO
Place your definition of `ngram_model_smoothed` and whatever other testing
of it you'd like to do in this and subsequent cells.
"""
...
```

Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of Lab 2-1 {-}

To double-check your work, the cell below will rerun all of the autograder tests.

In [ ]:
```
grader.check_all()
```