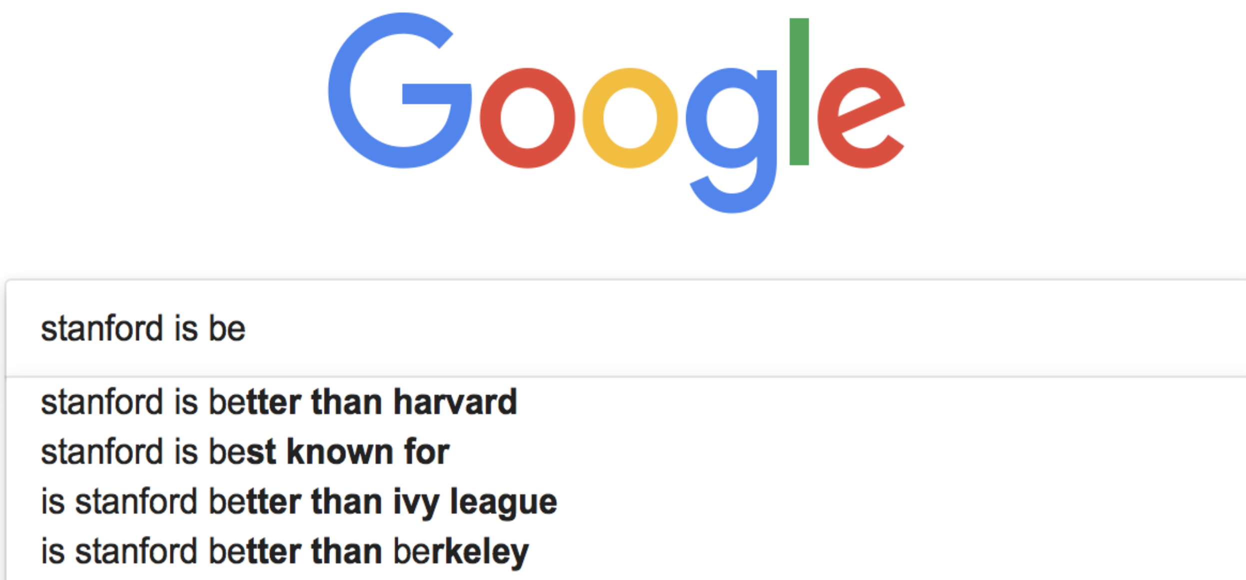
Language Models: Auto-Complete

In this assignment, you will build an auto-complete system. Auto-complete system is something you may see every day

- When you google something, you often have suggestions to help you complete your search.
- When you are writing an email, you get suggestions telling you possible endings to your sentence.

By the end of this assignment, you will develop a prototype of such a system.



Outline

A key building block for an auto-complete system is a language model. A language model assigns the probability to a sequence of words, in a way that more "likely" sequences receive higher scores. For example,

> "I have a pen" is expected to have a higher probability than "I am a pen" since the first one seems to be a more natural sentence in the real world.

You can take advantage of this probability calculation to develop an auto-complete system.
Suppose the user typed

> "I eat scrambled" Then you can find a word `x` such that "I eat scrambled x" receives the highest probability. If x = "eggs", the sentence would be "I eat scrambled eggs"

While a variety of language models have been developed, this assignment uses **N-grams**, a simple but powerful method for language modeling.

- N-grams are also used in machine translation and speech recognition.

Here are the steps of this assignment:

1. Load and preprocess data
    - Load and tokenize data.
    - Split the sentences into train and test sets.
    - Replace words with a low frequency by an unknown marker `<unk>`.
2. Develop N-gram based language models
    - Compute the count of n-grams from a given data set.
    - Estimate the conditional probability of a next word with k-smoothing.
3. Evaluate the N-gram models by computing the perplexity score.
4. Use your own model to suggest an upcoming word given your sentence.

In [1]:
```
import math
import random
import numpy as np
import pandas as pd
import nltk
nltk.data.path.append('.')
```

Part 1: Load and Preprocess Data

Part 1.1: Load the data

You will use twitter data. Load the data and view the first few sentences by running the next cell.

Notice that data is a long string that contains many many tweets. Observe that there is a line break "\n" between tweets.

In [2]:
```
with open("en_US.twitter.txt", "r") as f:
    data = f.read()
print("Data type:", type(data))
print("Number of letters:", len(data))
print("First 300 letters of the data")
print("-------")
display(data[0:300])
print("-------")

print("Last 300 letters of the data")
print("-------")
display(data[-300:])
print("-------")
```

```
Data type: <class 'str'>
Number of letters: 3335477
First 300 letters of the data
-------

"How are you? Btw thanks for the RT. You gonna be in DC anytime soon? Love to see you. Been way, way too long.\nWhen y
ou meet someone special... you'll know. Your heart will beat more rapidly and you'll smile for no reason.\nthey've dec
ided its more fun if I don't.\nSo Tired D; Played Lazer Tag & Ran A "

-------
Last 300 letters of the data
-------

"ust had one a few weeks back....hopefully we will be back soon! wish you the best yo\nColombia is with an 'o'...“: We
now ship to 4 countries in South America (fist pump). Please welcome Columbia to the Stunner Family”\n#GutsiestMovesYo
uCanMake Giving a cat a bath.\nCoffee after 5 was a TERRIBLE idea.\n"

-------
```

Part 1.2 Pre-process the data

Preprocess this data with the following steps:

1. Split data into sentences using "\n" as the delimiter.
2. Split each sentence into tokens. Note that in this assignment we use "token" and "words" interchangeably.
3. Assign sentences into train or test sets.
4. Find tokens that appear at least N times in the training data.
5. Replace tokens that appear less than N times by `<unk>`

Note: we omit validation data in this exercise.

- In real applications, we should hold a part of data as a validation set and use it to tune our training.
- We skip this process for simplicity.

Exercise 01

Split data into sentences.

▸ **Hints**

In [3]:
```python
# UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED_FUNCTION: split_to_sentences ###
def split_to_sentences(data):
    """
    Split data by linebreak "\n"

    Args:
        data: str

    Returns:
        A list of sentences
    """
    ### START CODE HERE (Replace instances of 'None' with your code) ###
    sentences = data.split('\n')
    ### END CODE HERE ###

    # Additional clearning (This part is already implemented)
    # - Remove leading and trailing spaces from each sentence
    # - Drop sentences if they are empty strings.
    sentences = [s.strip() for s in sentences]
    sentences = [s for s in sentences if len(s) > 0]

    return sentences
```

In [4]:
```python
# test your code
x = """
I have a pen.\nI have an apple. \nAh\nApple pen.\n
"""
print(x)

split_to_sentences(x)
```

```
I have a pen.
I have an apple.
Ah
Apple pen.
```

Out[4]:
```
['I have a pen.', 'I have an apple.', 'Ah', 'Apple pen.']
```

Expected answer:

```
['I have a pen.', 'I have an apple.', 'Ah', 'Apple pen.']
```

Exercise 02

The next step is to tokenize sentences (split a sentence into a list of words).

- Convert all tokens into lower case so that words which are capitalized (for example, at the start of a sentence) in the original text are treated the same as the lowercase versions of the words.
- Append each tokenized list of words into a list of tokenized sentences.

▶ **Hints**

In [5]:
```
# UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED_FUNCTION: tokenize_sentences ###
def tokenize_sentences(sentences):
    """
    Tokenize sentences into tokens (words)

    Args:
        sentences: List of strings

    Returns:
        List of lists of tokens
    """

    # Initialize the list of lists of tokenized sentences
    tokenized_sentences = []
    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Go through each sentence
    for sentence in sentences:

        # Convert to lowercase letters
        sentence = sentence.lower()

        # Convert into a list of words
        tokenized = nltk.word_tokenize(sentence)

        # append the list of words to the list of lists
        tokenized_sentences.append(tokenized)

    ### END CODE HERE ###

    return tokenized_sentences
```

In [6]:
```
# test your code
sentences = ["Sky is blue.", "Leaves are green.", "Roses are red."]
tokenize_sentences(sentences)
```

Out[6]:
```
[['sky', 'is', 'blue', '.'],
 ['leaves', 'are', 'green', '.'],
 ['roses', 'are', 'red', '.']]
```

Expected output

```
[['sky', 'is', 'blue', '.'],
 ['leaves', 'are', 'green', '.'],
 ['roses', 'are', 'red', '.']]
```

Exercise 03

Use the two functions that you have just implemented to get the tokenized data.

- split the data into sentences
- tokenize those sentences

In [7]:

```
# UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED_FUNCTION: get_tokenized_data ###
def get_tokenized_data(data):
    """
    Make a list of tokenized sentences

    Args:
        data: String

    Returns:
        List of lists of tokens
    """
    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Get the sentences by splitting up the data
    sentences = split_to_sentences(data)

    # Get the list of lists of tokens by tokenizing the sentences
    tokenized_sentences = tokenize_sentences(sentences)

    ### END CODE HERE ###

    return tokenized_sentences
```

In [8]:
```python
# test your function
x = "Sky is blue.\nLeaves are green\nRoses are red."
get_tokenized_data(x)
```

Out[8]:
```
[['sky', 'is', 'blue', '.'],
 ['leaves', 'are', 'green'],
 ['roses', 'are', 'red', '.']]
```

Expected outcome

```
[['sky', 'is', 'blue', '.'],
 ['leaves', 'are', 'green'],
 ['roses', 'are', 'red', '.']]
```

Split into train and test sets

Now run the cell below to split data into training and test sets.

In [9]:
```python
tokenized_data = get_tokenized_data(data)
random.seed(87)
random.shuffle(tokenized_data)

train_size = int(len(tokenized_data) * 0.8)
train_data = tokenized_data[0:train_size]
test_data = tokenized_data[train_size:]
```

```python
In [10]: print("{} data are split into {} train and {} test set".format(
             len(tokenized_data), len(train_data), len(test_data)))

         print("First training sample:")
         print(train_data[0])

         print("First test sample")
         print(test_data[0])
```

```
47961 data are split into 38368 train and 9593 test set
First training sample:
['i', 'personally', 'would', 'like', 'as', 'our', 'official', 'glove', 'of', 'the', 'team', 'local', 'company', 'and',
'quality', 'production']
First test sample
['that', 'picture', 'i', 'just', 'seen', 'whoa', 'dere', '!', '!', '>', '>', '>', '>', '>', '>', '>']
```

Expected output

```
47961 data are split into 38368 train and 9593 test set
First training sample:
['i', 'personally', 'would', 'like', 'as', 'our', 'official', 'glove', 'of', 'the', 'team', 'local', 'company', 'and'
, 'quality', 'production']
First test sample
['that', 'picture', 'i', 'just', 'seen', 'whoa', 'dere', '!', '!', '>', '>', '>', '>', '>', '>', '>']
```

Exercise 04

You won't use all the tokens (words) appearing in the data for training. Instead, you will use the more frequently used words.

- You will focus on the words that appear at least N times in the data.
- First count how many times each word appears in the data.

You will need a double for-loop, one for sentences and the other for tokens within a sentence.

▸ **Hints**

In [11]:

```
# UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED_FUNCTION: count_words ###
def count_words(tokenized_sentences):
    """
    Count the number of word appearence in the tokenized sentences

    Args:
        tokenized_sentences: List of lists of strings

    Returns:
        dict that maps word (str) to the frequency (int)
    """

    word_counts = {}
    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Loop through each sentence
    for sentence in tokenized_sentences: # complete this line

        # Go through each token in the sentence
        for token in sentence: # complete this line

            # If the token is not in the dictionary yet, set the count to 1
            if token not in word_counts: # complete this line
                word_counts[token] = 1

            # If the token is already in the dictionary, increment the count by 1
            else:
                word_counts[token] += 1

    ### END CODE HERE ###

    return word_counts
```

In [12]:
```
# test your code
tokenized_sentences = [['sky', 'is', 'blue', '.'],
                       ['leaves', 'are', 'green', '.'],
                       ['roses', 'are', 'red', '.']]
count_words(tokenized_sentences)
```

Out[12]:
```
{'sky': 1,
 'is': 1,
 'blue': 1,
 '.': 3,
 'leaves': 1,
 'are': 2,
 'green': 1,
 'roses': 1,
 'red': 1}
```

Expected output

Note that the order may differ.

```
{'sky': 1,
 'is': 1,
 'blue': 1,
 '.': 3,
 'leaves': 1,
 'are': 2,
 'green': 1,
 'roses': 1,
 'red': 1}
```

Handling 'Out of Vocabulary' words

If your model is performing autocomplete, but encounters a word that it never saw during training, it won't have an input word to help it determine the next word to suggest. The model will not be able to predict the next word because there are no counts for the current word.

- This 'new' word is called an 'unknown word', or **out of vocabulary (OOV)** words.
- The percentage of unknown words in the test set is called the **OOV** rate.

To handle unknown words during prediction, use a special token to represent all unknown words 'unk'.

- Modify the training data so that it has some 'unknown' words to train on.
- Words to convert into "unknown" words are those that do not occur very frequently in the training set.
- Create a list of the most frequent words in the training set, called the **closed vocabulary** .
- Convert all the other words that are not part of the closed vocabulary to the token 'unk'.

Exercise 05

You will now create a function that takes in a text document and a threshold `count_threshold` .

- Any word whose count is greater than or equal to the threshold `count_threshold` is kept in the closed vocabulary.
- Returns the word closed vocabulary list.

In [13]:

```
# UNQ_C5 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED_FUNCTION: get_words_with_nplus_frequency ###
def get_words_with_nplus_frequency(tokenized_sentences, count_threshold):
    """
    Find the words that appear N times or more

    Args:
        tokenized_sentences: List of lists of sentences
        count_threshold: minimum number of occurrences for a word to be in the closed vocabulary.

    Returns:
        List of words that appear N times or more
    """
    # Initialize an empty list to contain the words that
    # appear at least 'minimum_freq' times.
    closed_vocab = []

    # Get the word couts of the tokenized sentences
    # Use the function that you defined earlier to count the words
    word_counts = count_words(tokenized_sentences)

    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # for each word and its count
    for word, cnt in word_counts.items(): # complete this line

        # check that the word's count
        # is at least as great as the minimum count
        if cnt >= count_threshold:

            # append the word to the list
            closed_vocab.append(word)
    ### END CODE HERE ###

    return closed_vocab
```

In [14]:
```
# test your code
tokenized_sentences = [['sky', 'is', 'blue', '.'],
                       ['leaves', 'are', 'green', '.'],
                       ['roses', 'are', 'red', '.']]
tmp_closed_vocab = get_words_with_nplus_frequency(tokenized_sentences, count_threshold=2)
print(f"Closed vocabulary:")
print(tmp_closed_vocab)
```

```
Closed vocabulary:
['.', 'are']
```

Expected output

```
Closed vocabulary:
['.', 'are']
```

Exercise 06

The words that appear `count_threshold` times or more are in the closed vocabulary.

- All other words are regarded as `unknown` .
- Replace words not in the closed vocabulary with the token `<unk>` .

In [15]:
```
# UNQ_C6 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED_FUNCTION: replace_oov_words_by_unk ###
def replace_oov_words_by_unk(tokenized_sentences, vocabulary, unknown_token="<unk>"):
    """
    Replace words not in the given vocabulary with '<unk>' token.

    Args:
        tokenized_sentences: List of lists of strings
        vocabulary: List of strings that we will use
        unknown_token: A string representing unknown (out-of-vocabulary) words

    Returns:
        List of lists of strings, with words not in the vocabulary replaced
    """

    # Place vocabulary into a set for faster search
    vocabulary = set(vocabulary)

    # Initialize a list that will hold the sentences
    # after less frequent words are replaced by the unknown token
    replaced_tokenized_sentences = []

    # Go through each sentence
    for sentence in tokenized_sentences:

        # Initialize the list that will contain
        # a single sentence with "unknown_token" replacements
        replaced_sentence = []
        ### START CODE HERE (Replace instances of 'None' with your code) ###

        # for each token in the sentence
        for token in sentence: # complete this line

            # Check if the token is in the closed vocabulary
            if token in vocabulary: # complete this line
                # If so, append the word to the replaced_sentence
                replaced_sentence.append(token)
            else:
                # otherwise, append the unknown token instead
                replaced_sentence.append(unknown_token)
        ### END CODE HERE ###

        # Append the list of tokens to the list of lists
        replaced_tokenized_sentences.append(replaced_sentence)

    return replaced_tokenized_sentences
```

In [16]:
```
tokenized_sentences = [["dogs", "run"], ["cats", "sleep"]]
vocabulary = ["dogs", "sleep"]
tmp_replaced_tokenized_sentences = replace_oov_words_by_unk(tokenized_sentences, vocabulary)
print(f"Original sentence:")
print(tokenized_sentences)
print(f"tokenized_sentences with less frequent words converted to '<unk>':")
print(tmp_replaced_tokenized_sentences)
```

```
Original sentence:
[['dogs', 'run'], ['cats', 'sleep']]
tokenized_sentences with less frequent words converted to '<unk>':
[['dogs', '<unk>'], ['<unk>', 'sleep']]
```

Expected answer

```
Original sentence:
[['dogs', 'run'], ['cats', 'sleep']]
tokenized_sentences with less frequent words converted to '<unk>':
[['dogs', '<unk>'], ['<unk>', 'sleep']]
```

Exercise 07

Now we are ready to process our data by combining the functions that you just implemented.

1. Find tokens that appear at least count_threshold times in the training data.
2. Replace tokens that appear less than count_threshold times by "<unk>" both for training and test data.

In [17]:
```python
# UNQ_C7 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED_FUNCTION: preprocess_data ###
def preprocess_data(train_data, test_data, count_threshold):
    """
    Preprocess data, i.e.,
        - Find tokens that appear at least N times in the training data.
        - Replace tokens that appear less than N times by "<unk>" both for training and test data.
    Args:
        train_data, test_data: List of lists of strings.
        count_threshold: Words whose count is less than this are
                      treated as unknown.

    Returns:
        Tuple of
        - training data with low frequent words replaced by "<unk>"
        - test data with low frequent words replaced by "<unk>"
        - vocabulary of words that appear n times or more in the training data
    """
    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Get the closed vocabulary using the train data
    vocabulary = get_words_with_nplus_frequency(train_data, count_threshold)

    # For the train data, replace less common words with "<unk>"
    train_data_replaced = replace_oov_words_by_unk(train_data, vocabulary)

    # For the test data, replace less common words with "<unk>"
    test_data_replaced = replace_oov_words_by_unk(test_data, vocabulary)

    ### END CODE HERE ###
    return train_data_replaced, test_data_replaced, vocabulary
```

In [18]:
```python
# test your code
tmp_train = [['sky', 'is', 'blue', '.'],
     ['leaves', 'are', 'green']]
tmp_test = [['roses', 'are', 'red', '.']]

tmp_train_repl, tmp_test_repl, tmp_vocab = preprocess_data(tmp_train,
                                                           tmp_test,
                                                           count_threshold = 1)


print("tmp_train_repl")
print(tmp_train_repl)
print()
print("tmp_test_repl")
print(tmp_test_repl)
print()
print("tmp_vocab")
print(tmp_vocab)
```

```
tmp_train_repl
[['sky', 'is', 'blue', '.'], ['leaves', 'are', 'green']]

tmp_test_repl
[['<unk>', 'are', '<unk>', '.']]

tmp_vocab
['sky', 'is', 'blue', '.', 'leaves', 'are', 'green']
```

Expected outcome

```
tmp_train_repl
[['sky', 'is', 'blue', '.'], ['leaves', 'are', 'green']]

tmp_test_repl
[['<unk>', 'are', '<unk>', '.']]

tmp_vocab
['sky', 'is', 'blue', '.', 'leaves', 'are', 'green']
```

Preprocess the train and test data

Run the cell below to complete the preprocessing both for training and test sets.

In [19]:
```
minimum_freq = 2
train_data_processed, test_data_processed, vocabulary = preprocess_data(train_data,
                                                                         test_data,
                                                                         minimum_freq)
```

In [20]:
```
print("First preprocessed training sample:")
print(train_data_processed[0])
print()
print("First preprocessed test sample:")
print(test_data_processed[0])
print()
print("First 10 vocabulary:")
print(vocabulary[0:10])
print()
print("Size of vocabulary:", len(vocabulary))
```

```
First preprocessed training sample:
['i', 'personally', 'would', 'like', 'as', 'our', 'official', 'glove', 'of', 'the', 'team', 'local', 'company', 'and',
'quality', 'production']

First preprocessed test sample:
['that', 'picture', 'i', 'just', 'seen', 'whoa', 'dere', '!', '!', '>', '>', '>', '>', '>', '>', '>']

First 10 vocabulary:
['i', 'personally', 'would', 'like', 'as', 'our', 'official', 'glove', 'of', 'the']

Size of vocabulary: 14821
```

Expected output

```
First preprocessed training sample:
['i', 'personally', 'would', 'like', 'as', 'our', 'official', 'glove', 'of', 'the', 'team', 'local', 'company', 'and'
, 'quality', 'production']

First preprocessed test sample:
['that', 'picture', 'i', 'just', 'seen', 'whoa', 'dere', '!', '!', '>', '>', '>', '>', '>', '>', '>']

First 10 vocabulary:
['i', 'personally', 'would', 'like', 'as', 'our', 'official', 'glove', 'of', 'the']

Size of vocabulary: 14821
```

You are done with the preprocessing section of the assignment. Objects `train_data_processed` , `test_data_processed` , and `vocabulary` will be used in the rest of the exercises.

Part 2: Develop n-gram based language models

In this section, you will develop the n-grams language model.

- Assume the probability of the next word depends only on the previous n-gram.
- The previous n-gram is the series of the previous 'n' words.

The conditional probability for the word at position 't' in the sentence, given that the words preceding it are $w_{t-1}, w_{t-2} \cdots w_{t-n}$ is:

$$P(w_t|w_{t-1}\dots w_{t-n}) \tag{1}$$

You can estimate this probability by counting the occurrences of these series of words in the training data.

- The probability can be estimated as a ratio, where
- The numerator is the number of times word 't' appears after words t-1 through t-n appear in the training data.
- The denominator is the number of times word t-1 through t-n appears in the training data.

$$\hat{P}(w_t | w_{t-1}\dots w_{t-n}) = \frac{C(w_{t-1}\dots w_{t-n}, w_n)}{C(w_{t-1}\dots w_{t-n})} \tag{2}$$

- The function $C(\cdots)$ denotes the number of occurence of the given sequence.
- $\hat{P}$ means the estimation of P.
- Notice that denominator of the equation (2) is the number of occurence of the previous n words, and the numerator is the same sequence followed by the word w_t.

Later, you will modify the equation (2) by adding k-smoothing, which avoids errors when any counts are zero.

The equation (2) tells us that to estimate probabilities based on n-grams, you need the counts of n-grams (for denominator) and (n+1)-grams (for numerator).

Exercise 08

Next, you will implement a function that computes the counts of n-grams for an arbitrary number n.

When computing the counts for n-grams, prepare the sentence beforehand by prepending $n-1$ starting markers "<s>" to indicate the beginning of the sentence.

- For example, in the bi-gram model (N=2), a sequence with two start tokens "<s><s>" should predict the first word of a sentence.
- So, if the sentence is "I like food", modify it to be "<s><s> I like food".
- Also prepare the sentence for counting by appending an end token "<e>" so that the model can predict when to finish a sentence.

Technical note: In this implementation, you will store the counts as a dictionary.

- The key of each key-value pair in the dictionary is a **tuple** of n words (and not a list)
- The value in the key-value pair is the number of occurrences.
- The reason for using a tuple as a key instead of a list is because a list in Python is a mutable object (it can be changed after it is first created). A tuple is "immutable", so it cannot be altered after it is first created. This makes a tuple suitable as a data type for the key in a dictionary.

▶ **Hints**

In [21]:
```python
# UNQ_C8 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED FUNCTION: count_n_grams ###
def count_n_grams(data, n, start_token='<s>', end_token = '<e>'):
    """
    Count all n-grams in the data

    Args:
        data: List of lists of words
        n: number of words in a sequence

    Returns:
        A dictionary that maps a tuple of n-words to its frequency
    """

    # Initialize dictionary of n-grams and their counts
    n_grams = {}

    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Go through each sentence in the data
    for sentence in data: # complete this line

        # prepend start token n times, and  append <e> one time
        sentence = [start_token] * n + sentence + [end_token]

        # convert list to tuple
        # So that the sequence of words can be used as
        # a key in the dictionary
        sentence = tuple(sentence)

        # Use 'i' to indicate the start of the n-gram
        # from index 0
        # to the last index where the end of the n-gram
        # is within the sentence.

        for i in range(len(sentence) - n + 1): # complete this line

            # Get the n-gram from i to i+n
            n_gram = sentence[i:i+n]

            # check if the n-gram is in the dictionary
            if n_gram in n_grams: # complete this line

                # Increment the count for this n-gram
                n_grams[n_gram] += 1
            else:
                # Initialize this n-gram count to 1
```

```python
                n_grams[n_gram] = 1

            ### END CODE HERE ###
    return n_grams
```

In [22]:

```python
# test your code
# CODE REVIEW COMMENT: Outcome does not match expected outcome
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
print("Uni-gram:")
print(count_n_grams(sentences, 1))
print("Bi-gram:")
print(count_n_grams(sentences, 2))
```

```
Uni-gram:
{('<s>',): 2, ('i',): 1, ('like',): 2, ('a',): 2, ('cat',): 2, ('<e>',): 2, ('this',): 1, ('dog',): 1, ('is',): 1}
Bi-gram:
{('<s>', '<s>'): 2, ('<s>', 'i'): 1, ('i', 'like'): 1, ('like', 'a'): 2, ('a', 'cat'): 2, ('cat', '<e>'): 2, ('<s>',
'this'): 1, ('this', 'dog'): 1, ('dog', 'is'): 1, ('is', 'like'): 1}
```

Expected outcome:

```
Uni-gram:
{('<s>',): 2, ('i',): 1, ('like',): 2, ('a',): 2, ('cat',): 2, ('<e>',): 2, ('this',): 1, ('dog',): 1, ('is',): 1}
Bi-gram:
{('<s>', '<s>'): 2, ('<s>', 'i'): 1, ('i', 'like'): 1, ('like', 'a'): 2, ('a', 'cat'): 2, ('cat', '<e>'): 2, ('<s>', 't
his'): 1, ('this', 'dog'): 1, ('dog', 'is'): 1, ('is', 'like'): 1}
```

Exercise 09

Next, estimate the probability of a word given the prior 'n' words using the n-gram counts.

$$\hat{P}(w_t | w_{t-1}\dots w_{t-n}) = \frac{C(w_{t-1}\dots w_{t-n}, w_n)}{C(w_{t-1}\dots w_{t-n})} \tag{2}$$

This formula doesn't work when a count of an n-gram is zero..

- Suppose we encounter an n-gram that did not occur in the training data.
- Then, the equation (2) cannot be evaluated (it becomes zero divided by zero).

A way to handle zero counts is to add k-smoothing.

- K-smoothing adds a positive constant k to each numerator and $k \times |V|$ in the denominator, where $|V|$ is the number of words in the vocabulary.

$$\hat{P}(w_t | w_{t-1}\dots w_{t-n}) = \frac{C(w_{t-1}\dots w_{t-n}, w_n) + k}{C(w_{t-1}\dots w_{t-n}) + k|V|} \tag{3}$$

For n-grams that have a zero count, the equation (3) becomes $\frac{1}{|V|}$.

- This means that any n-gram with zero count has the same probability of $\frac{1}{|V|}$.

Define a function that computes the probability estimate (3) from n-gram counts and a constant k.

- The function takes in a dictionary 'n_gram_counts', where the key is the n-gram and the value is the count of that n-gram.
- The function also takes another dictionary n_plus1_gram_counts, which you'll use to find the count for the previous n-gram plus the current word.

▶ **Hints**

In [35]:
```python
# UNQ_C9 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
### GRADED FUNCTION: estimate_probability ###
def estimate_probability(word, previous_n_gram,
                         n_gram_counts, n_plus1_gram_counts, vocabulary_size, k=1.0):
    """
    Estimate the probabilities of a next word using the n-gram counts with k-smoothing

    Args:
        word: next word
        previous_n_gram: A sequence of words of length n
        n_gram_counts: Dictionary of counts of n-grams
        n_plus1_gram_counts: Dictionary of counts of (n+1)-grams
        vocabulary_size: number of words in the vocabulary
        k: positive constant, smoothing parameter

    Returns:
        A probability
    """
    # convert list to tuple to use it as a dictionary key
    previous_n_gram = tuple(previous_n_gram)

    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Set the denominator
    # If the previous n-gram exists in the dictionary of n-gram counts,
    # Get its count.  Otherwise set the count to zero
    # Use the dictionary that has counts for n-grams
    previous_n_gram_count = n_gram_counts.get(previous_n_gram, 0)
    # Calculate the denominator using the count of the previous n gram
    # and apply k-smoothing
    denominator = previous_n_gram_count + k * vocabulary_size

    # Define n plus 1 gram as the previous n-gram plus the current word as a tuple
    n_plus1_gram = tuple(list(previous_n_gram) + [word])

    # Set the count to the count in the dictionary,
    # otherwise 0 if not in the dictionary
    # use the dictionary that has counts for the n-gram plus current word
    n_plus1_gram_count = n_plus1_gram_counts.get(n_plus1_gram, 0)
    # Define the numerator use the count of the n-gram plus current word,
    # and apply smoothing
    numerator = n_plus1_gram_count + k

    # Calculate the probability as the numerator divided by denominator
    probability = numerator / denominator
    ### END CODE HERE ###
```

```
    return probability
```

```
In [36]: # test your code
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))

unigram_counts = count_n_grams(sentences, 1)
bigram_counts = count_n_grams(sentences, 2)
tmp_prob = estimate_probability("cat", "a", unigram_counts, bigram_counts, len(unique_words), k=1)

print(f"The estimated probability of word 'cat' given the previous n-gram 'a' is: {tmp_prob:.4f}")
```

```
The estimated probability of word 'cat' given the previous n-gram 'a' is: 0.3333
```

Expected output

```
The estimated probability of word 'cat' given the previous n-gram 'a' is: 0.3333
```

Estimate probabilities for all words

The function defined below loops over all words in vocabulary to calculate probabilities for all possible words.

- This function is provided for you.

In [37]:

```
def estimate_probabilities(previous_n_gram, n_gram_counts, n_plus1_gram_counts, vocabulary, k=1.0):
    """
    Estimate the probabilities of next words using the n-gram counts with k-smoothing

    Args:
        previous_n_gram: A sequence of words of length n
        n_gram_counts: Dictionary of counts of (n+1)-grams
        n_plus1_gram_counts: Dictionary of counts of (n+1)-grams
        vocabulary: List of words
        k: positive constant, smoothing parameter

    Returns:
        A dictionary mapping from next words to the probability.
    """

    # convert list to tuple to use it as a dictionary key
    previous_n_gram = tuple(previous_n_gram)

    # add <e> <unk> to the vocabulary
    # <s> is not needed since it should not appear as the next word
    vocabulary = vocabulary + ["<e>", "<unk>"]
    vocabulary_size = len(vocabulary)

    probabilities = {}
    for word in vocabulary:
        probability = estimate_probability(word, previous_n_gram,
                                           n_gram_counts, n_plus1_gram_counts,
                                           vocabulary_size, k=k)
        probabilities[word] = probability

    return probabilities
```

In [38]:
```python
# test your code
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))
unigram_counts = count_n_grams(sentences, 1)
bigram_counts = count_n_grams(sentences, 2)
estimate_probabilities("a", unigram_counts, bigram_counts, unique_words, k=1)
```

Out[38]:
```
{'i': 0.09090909090909091,
 'cat': 0.2727272727272727,
 'a': 0.09090909090909091,
 'this': 0.09090909090909091,
 'like': 0.09090909090909091,
 'is': 0.09090909090909091,
 'dog': 0.09090909090909091,
 '<e>': 0.09090909090909091,
 '<unk>': 0.09090909090909091}
```

Expected output

```
{'cat': 0.2727272727272727,
 'i': 0.09090909090909091,
 'this': 0.09090909090909091,
 'a': 0.09090909090909091,
 'is': 0.09090909090909091,
 'like': 0.09090909090909091,
 'dog': 0.09090909090909091,
 '<e>': 0.09090909090909091,
 '<unk>': 0.09090909090909091}
```

In [39]:
```
# Additional test
trigram_counts = count_n_grams(sentences, 3)
estimate_probabilities(["<s>", "<s>"], bigram_counts, trigram_counts, unique_words, k=1)
```

Out[39]:
```
{'i': 0.18181818181818182,
 'cat': 0.09090909090909091,
 'a': 0.09090909090909091,
 'this': 0.18181818181818182,
 'like': 0.09090909090909091,
 'is': 0.09090909090909091,
 'dog': 0.09090909090909091,
 '<e>': 0.09090909090909091,
 '<unk>': 0.09090909090909091}
```

Expected output

```
{'cat': 0.09090909090909091,
 'i': 0.18181818181818182,
 'this': 0.18181818181818182,
 'a': 0.09090909090909091,
 'is': 0.09090909090909091,
 'like': 0.09090909090909091,
 'dog': 0.09090909090909091,
 '<e>': 0.09090909090909091,
 '<unk>': 0.09090909090909091}
```

Count and probability matrices

As we have seen so far, the n-gram counts computed above are sufficient for computing the probabilities of the next word.

- It can be more intuitive to present them as count or probability matrices.
- The functions defined in the next cells return count or probability matrices.
- This function is provided for you.

In [40]:
```
def make_count_matrix(n_plus1_gram_counts, vocabulary):
    # add <e> <unk> to the vocabulary
    # <s> is omitted since it should not appear as the next word
    vocabulary = vocabulary + ["<e>", "<unk>"]

    # obtain unique n-grams
    n_grams = []
    for n_plus1_gram in n_plus1_gram_counts.keys():
        n_gram = n_plus1_gram[0:-1]
        n_grams.append(n_gram)
    n_grams = list(set(n_grams))

    # mapping from n-gram to row
    row_index = {n_gram:i for i, n_gram in enumerate(n_grams)}
    # mapping from next word to column
    col_index = {word:j for j, word in enumerate(vocabulary)}

    nrow = len(n_grams)
    ncol = len(vocabulary)
    count_matrix = np.zeros((nrow, ncol))
    for n_plus1_gram, count in n_plus1_gram_counts.items():
        n_gram = n_plus1_gram[0:-1]
        word = n_plus1_gram[-1]
        if word not in vocabulary:
            continue
        i = row_index[n_gram]
        j = col_index[word]
        count_matrix[i, j] = count

    count_matrix = pd.DataFrame(count_matrix, index=n_grams, columns=vocabulary)
    return count_matrix
```

In [41]:
```
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))
bigram_counts = count_n_grams(sentences, 2)

print('bigram counts')
display(make_count_matrix(bigram_counts, unique_words))
```

bigram counts

	i	cat	a	this	like	is	dog	<e>	<unk>
(cat,)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0
(is,)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(dog,)	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
(a,)	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(this,)	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
(<s>,)	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
(i,)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(like,)	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0

Expected output

```
bigram counts
          cat    i  this   a   is   like  dog  <e>   <unk>
(<s>,)    0.0  1.0  1.0  0.0  0.0  0.0   0.0  0.0   0.0
(a,)      2.0  0.0  0.0  0.0  0.0  0.0   0.0  0.0   0.0
(this,)   0.0  0.0  0.0  0.0  0.0  0.0   1.0  0.0   0.0
(like,)   0.0  0.0  0.0  2.0  0.0  0.0   0.0  0.0   0.0
(dog,)    0.0  0.0  0.0  0.0  1.0  0.0   0.0  0.0   0.0
(cat,)    0.0  0.0  0.0  0.0  0.0  0.0   0.0  2.0   0.0
(is,)     0.0  0.0  0.0  0.0  0.0  1.0   0.0  0.0   0.0
(i,)      0.0  0.0  0.0  0.0  0.0  1.0   0.0  0.0   0.0
```

In [42]:
```
# Show trigram counts
print('\ntrigram counts')
trigram_counts = count_n_grams(sentences, 3)
display(make_count_matrix(trigram_counts, unique_words))
```

trigram counts

	i	cat	a	this	like	is	dog	<e>	<unk>
(<s>, this)	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
(this, dog)	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
(i, like)	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
(<s>, i)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(is, like)	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
(a, cat)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0
(dog, is)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(<s>, <s>)	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
(like, a)	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Expected output

```
trigram counts
                cat    i    this   a   is   like  dog  <e>    <unk>
(dog, is)       0.0    0.0  0.0   0.0  0.0  1.0   0.0  0.0    0.0
(this, dog)     0.0    0.0  0.0   0.0  1.0  0.0   0.0  0.0    0.0
(a, cat)        0.0    0.0  0.0   0.0  0.0  0.0   0.0  2.0    0.0
(like, a)       2.0    0.0  0.0   0.0  0.0  0.0   0.0  0.0    0.0
(is, like)      0.0    0.0  0.0   1.0  0.0  0.0   0.0  0.0    0.0
(<s>, i)        0.0    0.0  0.0   0.0  0.0  1.0   0.0  0.0    0.0
(i, like)       0.0    0.0  0.0   1.0  0.0  0.0   0.0  0.0    0.0
(<s>, <s>)      0.0    1.0  1.0   0.0  0.0  0.0   0.0  0.0    0.0
(<s>, this)     0.0    0.0  0.0   0.0  0.0  0.0   1.0  0.0    0.0
```

The following function calculates the probabilities of each word given the previous n-gram, and stores this in matrix form.

- This function is provided for you.

In [43]:
```
def make_probability_matrix(n_plus1_gram_counts, vocabulary, k):
    count_matrix = make_count_matrix(n_plus1_gram_counts, unique_words)
    count_matrix += k
    prob_matrix = count_matrix.div(count_matrix.sum(axis=1), axis=0)
    return prob_matrix
```

In [44]:
```
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))
bigram_counts = count_n_grams(sentences, 2)
print("bigram probabilities")
display(make_probability_matrix(bigram_counts, unique_words, k=1))
```

```
bigram probabilities
```

	i	cat	a	this	like	is	dog	<e>	<unk>
(cat,)	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.272727	0.090909
(is,)	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000
(dog,)	0.100000	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000
(a,)	0.090909	0.272727	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909
(this,)	0.100000	0.100000	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000
(<s>,)	0.181818	0.090909	0.090909	0.181818	0.090909	0.090909	0.090909	0.090909	0.090909
(i,)	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000
(like,)	0.090909	0.090909	0.272727	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909

In [45]:
```
print("trigram probabilities")
trigram_counts = count_n_grams(sentences, 3)
display(make_probability_matrix(trigram_counts, unique_words, k=1))
```

```
trigram probabilities
```

	i	cat	a	this	like	is	dog	<e>	<unk>
(<s>, this)	0.100000	0.100000	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000
(this, dog)	0.100000	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000
(i, like)	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000	0.100000	0.100000
(<s>, i)	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000
(is, like)	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000	0.100000	0.100000
(a, cat)	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.272727	0.090909
(dog, is)	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000
(<s>, <s>)	0.181818	0.090909	0.090909	0.181818	0.090909	0.090909	0.090909	0.090909	0.090909
(like, a)	0.090909	0.272727	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909

Confirm that you obtain the same results as for the `estimate_probabilities` function that you implemented.

Part 3: Perplexity

In this section, you will generate the perplexity score to evaluate your model on the test set.

- You will also use back-off when needed.
- Perplexity is used as an evaluation metric of your language model.
- To calculate the the perplexity score of the test set on an n-gram model, use:

$$PP(W) = \sqrt[N]{\prod_{t=n+1}^{N} \frac{1}{P(w_t | w_{t-n} \cdots w_{t-1})}} \tag{4}$$

- where N is the length of the sentence.
- n is the number of words in the n-gram (e.g. 2 for a bigram).
- In math, the numbering starts at one and not zero.

In code, array indexing starts at zero, so the code will use ranges for t according to this formula:

$$PP(W) = \sqrt[N]{\prod_{t=n}^{N-1} \frac{1}{P(w_t | w_{t-n} \cdots w_{t-1})}} \tag{4.1}$$

The higher the probabilities are, the lower the perplexity will be.

- The more the n-grams tell us about the sentence, the lower the perplexity score will be.

Exercise 10

Compute the perplexity score given an N-gram count matrix and a sentence.

▶ **Hints**

In [59]:

```
# UNQ_C10 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: calculate_perplexity
def calculate_perplexity(sentence, n_gram_counts, n_plus1_gram_counts, vocabulary_size, k=1.0):
    """
    Calculate perplexity for a list of sentences

    Args:
        sentence: List of strings
        n_gram_counts: Dictionary of counts of (n+1)-grams
        n_plus1_gram_counts: Dictionary of counts of (n+1)-grams
        vocabulary_size: number of unique words in the vocabulary
        k: Positive smoothing constant

    Returns:
        Perplexity score
    """
    # length of previous words
    n = len(list(n_gram_counts.keys())[0])

    # prepend <s> and append <e>
    sentence = ["<s>"] * n + sentence + ["<e>"]

    # Cast the sentence from a list to a tuple
    sentence = tuple(sentence)

    # length of sentence (after adding <s> and <e> tokens)
    N = len(sentence)

    # The variable p will hold the product
    # that is calculated inside the n-root
    # Update this in the code below
    product_pi = 1.0

    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Index t ranges from n to N - 1, inclusive on both ends
    for t in range(n, N): # complete this line

        # get the n-gram preceding the word at position t
        n_gram = sentence[t - n : t]

        # get the word at position t
        word = sentence[t]

        # Estimate the probability of the word given the n-gram
        # using the n-gram counts, n-plus1-gram counts,
        # vocabulary size, and smoothing constant
```

```
        probability = estimate_probability(word, n_gram, \
                                 n_gram_counts, n_plus1_gram_counts, vocabulary_size, k)

        # Update the product of the probabilities
        # This 'product_pi' is a cumulative product
        # of the (1/P) factors that are calculated in the loop
        product_pi *= 1/probability

    # Take the Nth root of the product
    perplexity = product_pi ** (1/N)

    ### END CODE HERE ###
    return perplexity
```

In [60]:

```
# test your code

sentences = [['i', 'like', 'a', 'cat'],
                 ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))

unigram_counts = count_n_grams(sentences, 1)
bigram_counts = count_n_grams(sentences, 2)


perplexity_train1 = calculate_perplexity(sentences[0],
                                         unigram_counts, bigram_counts,
                                         len(unique_words), k=1.0)
print(f"Perplexity for first train sample: {perplexity_train1:.4f}")

test_sentence = ['i', 'like', 'a', 'dog']
perplexity_test = calculate_perplexity(test_sentence,
                                       unigram_counts, bigram_counts,
                                       len(unique_words), k=1.0)
print(f"Perplexity for test sample: {perplexity_test:.4f}")
```

```
Perplexity for first train sample: 2.8040
Perplexity for test sample: 3.9654
```

Expected Output

```
Perplexity for first train sample: 2.8040
Perplexity for test sample: 3.9654
```

Note: If your sentence is really long, there will be underflow when multiplying many fractions.

- To handle longer sentences, modify your implementation to take the sum of the log of the probabilities.

Part 4: Build an auto-complete system

In this section, you will combine the language models developed so far to implement an auto-complete system.

Exercise 11

Compute probabilities for all possible next words and suggest the most likely one.

- This function also take an optional argument `start_with`, which specifies the first few letters of the next words.

▶ **Hints**

In [63]:

```
# UNQ_C11 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: suggest_a_word
def suggest_a_word(previous_tokens, n_gram_counts, n_plus1_gram_counts, vocabulary, k=1.0, start_with=None):
    """
    Get suggestion for the next word

    Args:
        previous_tokens: The sentence you input where each token is a word. Must have length > n
        n_gram_counts: Dictionary of counts of (n+1)-grams
        n_plus1_gram_counts: Dictionary of counts of (n+1)-grams
        vocabulary: List of words
        k: positive constant, smoothing parameter
        start_with: If not None, specifies the first few letters of the next word

    Returns:
        A tuple of
          - string of the most likely next word
          - corresponding probability
    """

    # length of previous words
    n = len(list(n_gram_counts.keys())[0])

    # From the words that the user already typed
    # get the most recent 'n' words as the previous n-gram
    previous_n_gram = previous_tokens[-n:]

    # Estimate the probabilities that each word in the vocabulary
    # is the next word,
    # given the previous n-gram, the dictionary of n-gram counts,
    # the dictionary of n plus 1 gram counts, and the smoothing constant
    probabilities = estimate_probabilities(previous_n_gram,
                                           n_gram_counts, n_plus1_gram_counts,
                                           vocabulary, k=k)

    # Initialize suggested word to None
    # This will be set to the word with highest probability
    suggestion = None

    # Initialize the highest word probability to 0
    # this will be set to the highest probability
    # of all words to be suggested
    max_prob = 0

    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # For each word and its probability in the probabilities dictionary:
```

```python
    for word, prob in probabilities.items(): # complete this line

        # If the optional start_with string is set
        if start_with: # complete this line

            # Check if the beginning of word does not match with the letters in 'start_with'
            if not word.startswith(start_with): # complete this line

                # if they don't match, skip this word (move onto the next word)
                continue # complete this line

        # Check if this word's probability
        # is greater than the current maximum probability
        if prob > max_prob: # complete this line

            # If so, save this word as the best suggestion (so far)
            suggestion = word

            # Save the new maximum probability
            max_prob = prob

    ### END CODE HERE

    return suggestion, max_prob
```

In [64]:
```python
# test your code
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))

unigram_counts = count_n_grams(sentences, 1)
bigram_counts = count_n_grams(sentences, 2)

previous_tokens = ["i", "like"]
tmp_suggest1 = suggest_a_word(previous_tokens, unigram_counts, bigram_counts, unique_words, k=1.0)
print(f"The previous words are 'i like',\n\tand the suggested word is `{tmp_suggest1[0]}` with a probability of {tmp_suggest1[1]:.4f}")

print()
# test your code when setting the starts_with
tmp_starts_with = 'c'
tmp_suggest2 = suggest_a_word(previous_tokens, unigram_counts, bigram_counts, unique_words, k=1.0, start_with=tmp_starts_with)
print(f"The previous words are 'i like', the suggestion must start with `{tmp_starts_with}`\n\tand the suggested word is `{tmp_suggest2[0]}` with a probability of {tmp_suggest2[1]:.4f}")
```

```
The previous words are 'i like',
        and the suggested word is `a` with a probability of 0.2727

The previous words are 'i like', the suggestion must start with `c`
        and the suggested word is `cat` with a probability of 0.0909
```

Expected output

```
The previous words are 'i like',
    and the suggested word is `a` with a probability of 0.2727

The previous words are 'i like', the suggestion must start with `c`
    and the suggested word is `cat` with a probability of 0.0909
```

Get multiple suggestions

The function defined below loop over varioud n-gram models to get multiple suggestions.

In [71]:
```python
def get_suggestions(previous_tokens, n_gram_counts_list, vocabulary, k=1.0, start_with=None):
    model_counts = len(n_gram_counts_list)
    suggestions = []
    for i in range(model_counts-1):
        n_gram_counts = n_gram_counts_list[i]
        n_plus1_gram_counts = n_gram_counts_list[i+1]

        suggestion = suggest_a_word(previous_tokens, n_gram_counts,
                                    n_plus1_gram_counts, vocabulary,
                                    k=k, start_with=start_with)
        suggestions.append(suggestion)
    return suggestions
```

In [74]:
```python
# test your code
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))

unigram_counts = count_n_grams(sentences, 1)
bigram_counts = count_n_grams(sentences, 2)
trigram_counts = count_n_grams(sentences, 3)
quadgram_counts = count_n_grams(sentences, 4)
qintgram_counts = count_n_grams(sentences, 5)

n_gram_counts_list = [unigram_counts, bigram_counts, trigram_counts, quadgram_counts, qintgram_counts]
previous_tokens = ["i", "like"]
tmp_suggest3 = get_suggestions(previous_tokens, n_gram_counts_list, unique_words, k=1.0)

print(f"The previous words are 'i like', the suggestions are:")
display(tmp_suggest3)
```

```
The previous words are 'i like', the suggestions are:

[('a', 0.2727272727272727),
 ('a', 0.2),
 ('i', 0.1111111111111111),
 ('i', 0.1111111111111111)]
```

Suggest multiple words using n-grams of varying length

Congratulations! You have developed all building blocks for implementing your own auto-complete systems.

Let's see this with n-grams of varying lengths (unigrams, bigrams, trigrams, 4-grams...6-grams).

In [75]:
```python
n_gram_counts_list = []
for n in range(1, 6):
    print("Computing n-gram counts with n =", n, "...")
    n_model_counts = count_n_grams(train_data_processed, n)
    n_gram_counts_list.append(n_model_counts)
```

```
Computing n-gram counts with n = 1 ...
Computing n-gram counts with n = 2 ...
Computing n-gram counts with n = 3 ...
Computing n-gram counts with n = 4 ...
Computing n-gram counts with n = 5 ...
```

In [76]:
```python
previous_tokens = ["i", "am", "to"]
tmp_suggest4 = get_suggestions(previous_tokens, n_gram_counts_list, vocabulary, k=1.0)

print(f"The previous words are {previous_tokens}, the suggestions are:")
display(tmp_suggest4)
```

```
The previous words are ['i', 'am', 'to'], the suggestions are:

[('be', 0.027665685098338604),
 ('have', 0.00013487086115044844),
 ('have', 0.00013490725126475548),
 ('i', 6.746272684341901e-05)]
```

In [77]:
```python
previous_tokens = ["i", "want", "to", "go"]
tmp_suggest5 = get_suggestions(previous_tokens, n_gram_counts_list, vocabulary, k=1.0)

print(f"The previous words are {previous_tokens}, the suggestions are:")
display(tmp_suggest5)
```

```
The previous words are ['i', 'want', 'to', 'go'], the suggestions are:

[('to', 0.014051961029228078),
 ('to', 0.004697942168993581),
 ('to', 0.0009424436216762033),
 ('to', 0.0004044489383215369)]
```

In [78]:
```
previous_tokens = ["hey", "how", "are"]
tmp_suggest6 = get_suggestions(previous_tokens, n_gram_counts_list, vocabulary, k=1.0)

print(f"The previous words are {previous_tokens}, the suggestions are:")
display(tmp_suggest6)
```

```
The previous words are ['hey', 'how', 'are'], the suggestions are:

[('you', 0.023426812585499317),
 ('you', 0.003559435862995299),
 ('you', 0.00013491635186184566),
 ('i', 6.746272684341901e-05)]
```

In [79]:
```
previous_tokens = ["hey", "how", "are", "you"]
tmp_suggest7 = get_suggestions(previous_tokens, n_gram_counts_list, vocabulary, k=1.0)

print(f"The previous words are {previous_tokens}, the suggestions are:")
display(tmp_suggest7)
```

```
The previous words are ['hey', 'how', 'are', 'you'], the suggestions are:

[("'re", 0.023973994311255586),
 ('?', 0.002888465830762161),
 ('?', 0.0016134453781512605),
 ('<e>', 0.00013491635186184566)]
```

In [80]:
```
previous_tokens = ["hey", "how", "are", "you"]
tmp_suggest8 = get_suggestions(previous_tokens, n_gram_counts_list, vocabulary, k=1.0, start_with="d")

print(f"The previous words are {previous_tokens}, the suggestions are:")
display(tmp_suggest8)
```

```
The previous words are ['hey', 'how', 'are', 'you'], the suggestions are:

[('do', 0.009020723283218204),
 ('doing', 0.0016411737674785006),
 ('doing', 0.00047058823529411766),
 ('dvd', 6.745817593092283e-05)]
```

Congratulations!

You've completed this assignment by building an autocomplete model using an n-gram language model!

Please continue onto the fourth and final week of this course!