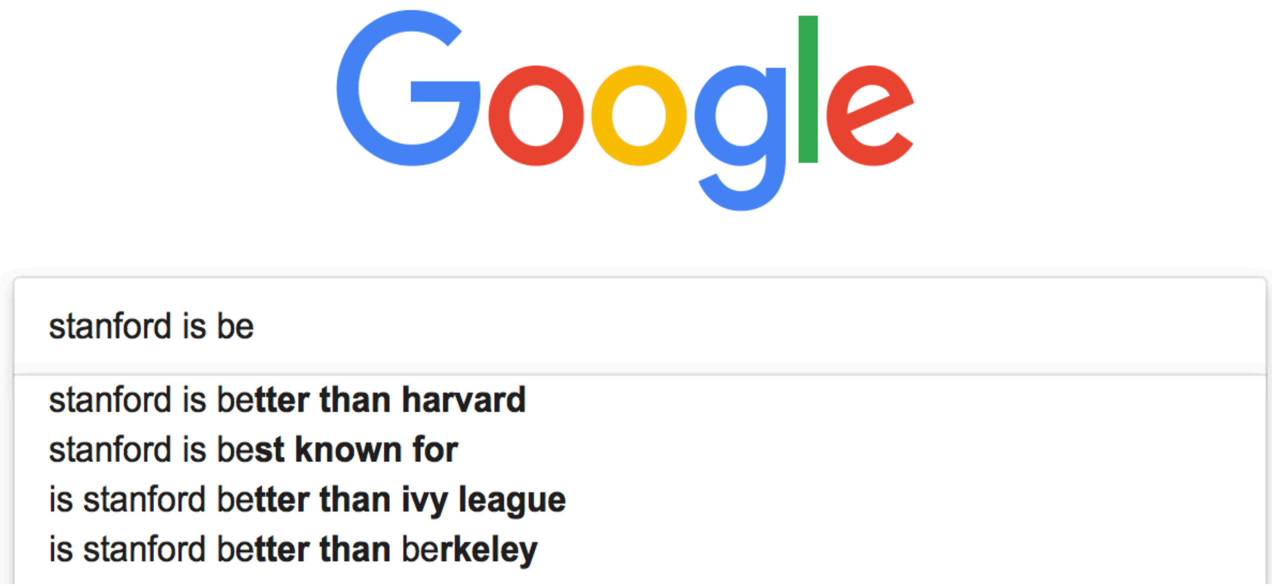
Assignment 3: Language Models: Auto-Complete

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.



Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra* `print` statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, `Grader Error: Grader feedback not found` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions (https://www.coursera.org/learn/probabilistic-models-in-nlp/supplement/saGQf/how-to-refresh-your-workspace)](https://www.coursera.org/learn/probabilistic-models-in-nlp/supplement/saGQf/how-to-refresh-your-workspace).

Table of Contents

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.

In [1]:

```
import math
import random
import numpy as np
import pandas as pd
import nltk
nltk.download('punkt')

import w3_unittest
nltk.data.path.append('.')
```

```
[nltk_data] Downloading package punkt to /home/jovyan/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

1 - Load and Preprocess Data

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("./data/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 yo
u meet someone special... you'll know. Your heart will beat more rapidly and you'll smile for no reason.\nthey've decid
ed 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#GutsiestMovesYou
CanMake Giving a cat a bath.\nCoffee after 5 was a TERRIBLE idea.\n"

-------
```

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 1- split_to_sentences

Split data into sentences.

Hints

In [3]:

```
### UNQ_C1 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 ###
    sentences = data.split("\n")
    ### END CODE HERE ###

    # Additional cleaning (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]:

```
# 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.']
```

In [5]:

```
# Test your function
w3_unittest.test_split_to_sentences(split_to_sentences)
```

```
All tests passed
```

Exercise 2 - tokenize_sentences

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 [6]:
```
import re

### UNQ_C2 GRADED_FUNCTION: tokenize_sentences ###
def tokenize_sentences(sentences):
    """
    Tokenize sentences into tokens (words), keeping punctuation together if consecutive.

    Args:
        sentences: List of strings

    Returns:
        List of lists of tokens
    """

    tokenized_sentences = []

    for sentence in sentences:
        # Convert to lowercase
        sentence = sentence.lower()

        # Regex to split words and consecutive punctuation
        tokenized = re.findall(r'\w+|[^\w\s]+', sentence)

        tokenized_sentences.append(tokenized)

    return tokenized_sentences
```

In [7]:
```
# test your code
sentences = ["Sky is blue.", "Leaves are green.", "Roses are red."]
tokenize_sentences(sentences)
```

Out[7]:
```
[['sky', 'is', 'blue', '.'],
 ['leaves', 'are', 'green', '.'],
 ['roses', 'are', 'red', '.']]
```

Expected output

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

In [8]:
```
# Test your function
w3_unittest.test_tokenize_sentences(tokenize_sentences)
```

```
All tests passed
```

Exercise 3 - get_tokenized_data

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

- split the data into sentences
- tokenize those sentences

In [9]:
```
### UNQ_C3 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 ###

    # 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 [10]:

```
# test your function
x = "Sky is blue.\nLeaves are green\nRoses are red."
get_tokenized_data(x)
```

Out[10]:

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

Expected outcome

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

In [11]:

```
# Test your function
w3_unittest.test_get_tokenized_data(get_tokenized_data)
```

```
All tests passed
```

Split into train and test sets

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

In [12]:

```
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:]
```

In [13]:

```
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 4 - count_words

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 [14]:
```
### UNQ_C4 GRADED_FUNCTION: count_words ###
def count_words(tokenized_sentences):
    """
    Count the number of word appearance 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 ###
    
    # Loop through each sentence
    for sentence in tokenized_sentences:
        
        # Go through each token in the sentence
        for token in sentence:
            
            # If the token is not in the dictionary yet, set the count to 1
            if token not in word_counts:
                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 [15]:
```
# test your code
tokenized_sentences = [['sky', 'is', 'blue', '.'],
                       ['leaves', 'are', 'green', '.'],
                       ['roses', 'are', 'red', '.']]
count_words(tokenized_sentences)
```

Out[15]:
```
{'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}
```

In [16]:
```
# Test your function
w3_unittest.test_count_words(count_words)
```

```
All tests passed
```

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 5 - get_words_with_nplus_frequency

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 [17]:
```
### UNQ_C5 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 words
        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 counts of the tokenized sentences
    word_counts = count_words(tokenized_sentences)

    ### START CODE HERE ###

    # Loop through each word and its count
    for word, cnt in word_counts.items():
        # Check if the count is at least the threshold
        if cnt >= count_threshold:
            closed_vocab.append(word)

    ### END CODE HERE ###

    return closed_vocab
```

In [18]:
```
# 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']
```

In [19]:
```
# Test your function
w3_unittest.test_get_words_with_nplus_frequency(get_words_with_nplus_frequency)
```

```
 All tests passed
```

Exercise 6 - replace_oov_words_by_unk

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 [20]:
```
### UNQ_C6 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 ###

        # for each token in the sentence
        for token in sentence:

            # Check if the token is in the closed vocabulary
            if token in vocabulary:
                # 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 [21]:
```
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']]
```

In [22]:
```
# Test your function
w3_unittest.test_replace_oov_words_by_unk(replace_oov_words_by_unk)
```

```
All tests passed
```

Exercise 7 - preprocess_data

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 [23]:
```
### UNQ_C7 GRADED_FUNCTION: preprocess_data ###
def preprocess_data(train_data, test_data, count_threshold, unknown_token="<unk>",
                    get_words_with_nplus_frequency=get_words_with_nplus_frequency,
                    replace_oov_words_by_unk=replace_oov_words_by_unk):
    """
    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 ###

    # 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, unknown_token)

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

    ### END CODE HERE ###

    return train_data_replaced, test_data_replaced, vocabulary
```

In [24]:
```
# 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']
```

In [25]:
```
# Test your function
w3_unittest.test_preprocess_data(preprocess_data)
```

```
All tests passed
```

Preprocess the train and test data

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

In [26]:
```
minimum_freq = 2
train_data_processed, test_data_processed, vocabulary = preprocess_data(train_data,
                                                                        test_data,
                                                                        minimum_freq)
```

In [27]:
```
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', '!!', '<unk>']

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

Size of vocabulary: 15061
```

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.

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 $w_{t-n} \cdots w_{t-2}, w_{t-1}$ is:

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

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-n through t-1 appear in the training data.
- The denominator is the number of times word t-n through t-1 appears in the training data.

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

- 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 8 - count_n_grams

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 tri-gram model (n=3), a sequence with two start tokens "<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.
- Although for a n-gram you need to use n-1 starting markers for a sentence, you will want to prepend n starting markers in order to use them to

Hints

In [28]:
```
### UNQ_C8 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 ###

    # Go through each sentence in the data
    for sentence in data:

        # prepend start token n times, and append the end token once
        sentence = [start_token]*n + sentence + [end_token]

        # convert list to tuple for dictionary key usage
        sentence = tuple(sentence)

        # Iterate over all n-grams in the sentence
        for i in range(len(sentence) - n + 1):
            # Get the n-gram as a tuple
            n_gram = sentence[i:i+n]

            # Check if n-gram already exists in dictionary
            if n_gram in n_grams:
                # Increment count
                n_grams[n_gram] += 1
            else:
                # Initialize count
                n_grams[n_gram] = 1

    ### END CODE HERE ###

    return n_grams
```

In [29]:
```
# 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>', 't
his'): 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>', 'this'): 1, ('this', 'dog'): 1, ('dog', 'is'): 1, ('is', 'like'): 1}
```

In [30]:
```
# Test your function
w3_unittest.test_count_n_grams(count_n_grams)
```

```
All tests passed
```

Exercise 9 - estimate_probability

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

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

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.

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

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 [31]:
```
### UNQ_C9 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 probability of the next word using 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 ###

    # Count of previous n-gram (0 if not present)
    previous_n_gram_count = n_gram_counts.get(previous_n_gram, 0)

    # Denominator: count of previous n-gram + k * |V|
    denominator = previous_n_gram_count + k * vocabulary_size

    # Count of (previous n-gram + current word) (0 if not present)
    n_plus1_gram = previous_n_gram + (word,)
    n_plus1_gram_count = n_plus1_gram_counts.get(n_plus1_gram, 0)

    # Numerator: count of n+1-gram + k
    numerator = n_plus1_gram_count + k

    # Probability
    probability = numerator / denominator

    ### END CODE HERE ###

    return probability
```

In [32]:
```
# 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
```

In [33]:
```
# Test your function
w3_unittest.test_estimate_probability(estimate_probability)
```

```
 All tests passed
```

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 [34]:
```
def estimate_probabilities(previous_n_gram, n_gram_counts, n_plus1_gram_counts, vocabulary, end_token='<e>', unknown_toke
    """
    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-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 + [end_token, unknown_token]
    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 [35]:
```
# 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[35]:
```
{'cat': 0.2727272727272727,
 'a': 0.09090909090909091,
 'i': 0.09090909090909091,
 'dog': 0.09090909090909091,
 'is': 0.09090909090909091,
 'like': 0.09090909090909091,
 'this': 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 [36]:
```
# Additional test
trigram_counts = count_n_grams(sentences, 3)
estimate_probabilities(["<s>", "<s>"], bigram_counts, trigram_counts, unique_words, k=1)
```

Out[36]:
```
{'cat': 0.09090909090909091,
 'a': 0.09090909090909091,
 'i': 0.18181818181818182,
 'dog': 0.09090909090909091,
 'is': 0.09090909090909091,
 'like': 0.09090909090909091,
 'this': 0.18181818181818182,
 '<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
```

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 [37]:
```
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 [38]:
```
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
```

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

In [39]:
```
# Show trigram counts
print('\ntrigram counts')
trigram_counts = count_n_grams(sentences, 3)
display(make_count_matrix(trigram_counts, unique_words))
```

trigram counts

	cat	a	i	dog	is	like	this	<e>	<unk>
(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
(this, dog)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(dog, is)	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
(<s>, <s>)	0.0	0.0	1.0	0.0	0.0	0.0	1.0	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	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(<s>, this)	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
(is, like)	0.0	1.0	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 [40]:
```
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 [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 probabilities")
display(make_probability_matrix(bigram_counts, unique_words, k=1))
```

bigram probabilities

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

In [42]:
```
print("trigram probabilities")
trigram_counts = count_n_grams(sentences, 3)
display(make_probability_matrix(trigram_counts, unique_words, k=1))
```

```
trigram probabilities
```

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

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

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 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 - calculate_perplexity

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

Note: For the sake of simplicity, in the code below, `<s>` is included in perplexity score calculation.

Hints

In [43]:
```
### UNQ_C10 GRADED FUNCTION: calculate_perplexity ###
def calculate_perplexity(sentence, n_gram_counts, n_plus1_gram_counts, vocabulary_size,
                         start_token='<s>', end_token='<e>', k=1.0):
    """
    Calculate perplexity for a single sentence

    Args:
        sentence: List of strings
        n_gram_counts: Dictionary of counts of n-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)
    n = len(list(n_gram_counts.keys())[0])

    # prepend <s> n times and append <e> once
    sentence = [start_token] * n + sentence + [end_token]

    # convert to tuple for indexing
    sentence = tuple(sentence)

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

    # Initialize product for perplexity
    product_pi = 1.0

    ### START CODE HERE ###
    # Iterate over positions from n to N-1 (inclusive)
    for t in range(n, N):
        # n-gram preceding the current word
        n_gram = sentence[t-n:t]

        # current word at position t
        word = sentence[t]

        # estimate probability using smoothing
        probability = estimate_probability(word, n_gram, n_gram_counts, n_plus1_gram_counts, vocabulary_size, k)

        # update cumulative product of 1/probabilities
        product_pi *= 1 / probability
    ### END CODE HERE ###

    # Take the Nth root to compute perplexity
    perplexity = product_pi ** (1/N)

    return perplexity
```

In [44]:
```
# 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_train = calculate_perplexity(sentences[0],
                                         unigram_counts, bigram_counts,
                                         len(unique_words), k=1.0)
print(f"Perplexity for first train sample: {perplexity_train:.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
```

In [45]:
```
# Test your function
w3_unittest.test_calculate_perplexity(calculate_perplexity)
```

```
 All tests passed
```

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.

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 - suggest_a_word

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 [46]:
```
### UNQ_C11 GRADED FUNCTION: suggest_a_word ###
def suggest_a_word(previous_tokens, n_gram_counts, n_plus1_gram_counts, vocabulary,
                   end_token='<e>', unknown_token="<unk>", 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-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)
    n = len(list(n_gram_counts.keys())[0])

    # prepend start tokens
    previous_tokens = ['<s>'] * n + previous_tokens

    # most recent n tokens
    previous_n_gram = previous_tokens[-n:]

    # Estimate probabilities of each word in the vocabulary as next word
    probabilities = estimate_probabilities(previous_n_gram,
                                           n_gram_counts, n_plus1_gram_counts,
                                           vocabulary, k=k)

    # Initialize suggested word and max probability
    suggestion = None
    max_prob = 0

    ### START CODE HERE ###
    for word, prob in probabilities.items():

        # If start_with is set, check that the word starts with the prefix
        if start_with is not None:
            if not word.startswith(start_with):
                continue

        # If this word's probability is greater than current max
        if prob > max_prob:
            suggestion = word
            max_prob = prob
    ### END CODE HERE ###

    return suggestion, max_prob
```

In [47]:
```
# 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_sugg

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_
print(f"The previous words are 'i like', the suggestion must start with `{tmp_starts_with}`\n\tand the suggested word is
```

```
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
```

In [48]:
```
# Test your function
w3_unittest.test_suggest_a_word(suggest_a_word)
```

```
All tests passed
```

Get multiple suggestions

The function defined below loops over various n-gram models to get multiple suggestions.

In [49]:
```

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 [50]:
```
# 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), ('a', 0.2), ('a', 0.2)]
```

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 [51]:
```
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 [52]:
```
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', 0.027395780833753333),
 ('have', 0.00013272280841462606),
 ('have', 0.00013275804845668768),
 ('i', 6.638783774812455e-05)]
```

In [53]:
```
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', 0.013837638376383764),
 ('to', 0.004624503354393278),
 ('to', 0.0009936406995230524),
 ('to', 0.0003980099502487562)]
```

In [54]:
```
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', 0.023244005641748944),
 ('you', 0.003502974223397224),
 ('you', 0.0001327668613913967),
 ('i', 6.638783774812455e-05)]
```

In [55]:
```
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:

[("'", 0.033896943652355516),
 ('?', 0.00264960579035802),
 ('?', 0.001521667217995369),
 ('<e>', 0.0001327668613913967)]
```

In [ ]:
```
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)
```

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!