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

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

# Only install packages if not in cs187-env
if 'cs187-env' not in os.environ.get('CONDA_PREFIX', ''):
    import subprocess
    import sys
    subprocess.run([sys.executable, '-m', 'pip', 'install', '-q', '-r', 'req
                    check=True)
```

In [ ]:
```
import math
import re
import sys

import torch
import nltk
```

In [ ]:
```
nltk.download('punkt', quiet=True) # this module is used to tokenize the tex
```

Where we're headed: Nearest neighbor text classification works by classifying a novel text with the same class as that of the training text that is closest according to some distance metric. These metrics are calculated based on representations of the texts. In this lab, we'll introduce some different representations and you'll use nearest neighbor classification to predict the speaker of sentences selected from a children's book.

The objectives of this lab are to:

- Clarify terminology around words and texts,
- Manipulate different representations of words and texts,
- Apply these representations to calculate text similarity, and
- Classify documents by a simple nearest neighbor model.

Recall that in this and all labs, we will have you carry out several exercises in notebook cells. The cells you are to do are marked '`#TODO`'. They will typically have a `...` where your code or answer should go. Where specified, you needn't write code to calculate the answer, but instead, simply work out the answer yourself and enter it.

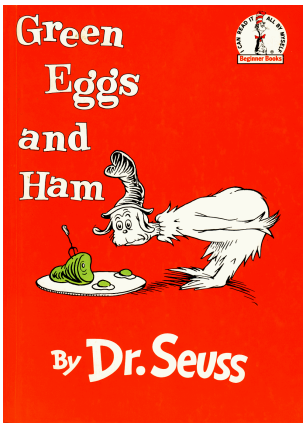
New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- `math.acos`
- `math.pi`
- `re.match`
- `set`

- `sorted`
- `str.join`
- `str.lower`
- `torch.amax`
- `torch.dot`
- `torch.linalg.norm`
- `torch.maximum`
- `torch.minimum`
- `torch.stack`
- `torch.sum`
- `torch.Tensor.type`
- `torch.where`
- `torch.zeros`
- `torch.zeros_like`

Counting words

Here are five sentences from Dr. Seuss's *Green Eggs and Ham*:




```
Would you like them here or there?
I would not like them here or there.
I would not like them anywhere.
I do not like green eggs and ham.
I do not like them, Sam-I-am.
```

Let's make this text available in the variable `text`.

In [ ]:
```
text = """
    Would you like them here or there?
    I would not like them here or there.
    I would not like them anywhere.
    I do not like green eggs and ham.
    I do not like them, Sam-I-am.
    """
```

A Python string like this is, of course, a sequence of characters. But we think of this text as a sequence of sentences each composed of a sequence of words. How many words are there in this text? That is a fraught question, for several reasons, including

- The type-token distinction
- Tokenization issues
- Normalization

Types versus tokens

In determining the number of words in `text`, are we talking about word *types* or word *tokens*. (For instance, there are five *tokens* of the word *type* 'like' in the example `text`.)

How many word tokens are there in total in this text? (Just count them manually.) Assign the number to the variable `token_count` in the next cell.

In [ ]:
```
#TODO – define `token_count` to be the number of tokens in `text`
token_count = ...
```

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

How many word types are there? (Again, you can just count manually.)

In [ ]:
```
#TODO – define `type_count` to be the number of types in `text`
type_count = ...
```

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

Question: The set of types of a language is referred to as its *vocabulary*. Are there more types or tokens as you calculated above? Could it be otherwise?

Type your answer here, replacing this text.

Tokenization

Did you count 'there?' as one token or two? This raises the issue of *tokenization* of text, how to decide where the token boundaries occur. For instance, here's a simple way to split a string – to *tokenize* it – in Python by splitting at whitespace.

In [ ]:
```
def whitespace_tokenize(str):
    return str.split()
```

Try it out on the `text` defined above.

In [ ]:
```
#TODO – define `tokens` to be the tokens as defined by the `whitespace_token
tokens = ...
```

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

Using this tokenization method, count the number of tokens in the text, this time using Python to do the work.

In [ ]:
```
#TODO – place your token count here
token_count_2 = ...
```

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

Arguably, we *should* split off punctuation as separate tokens, but even there, some care must be taken. We don't want to split 'don't' into three tokens or 'Sam-I-am' into five. (There's a good argument to be made however that the string 'don't' should be construed as two tokens, namely, 'do' and 'n't', but that's beyond the scope of today's discussion.)

Here, we provide an alternative tokenizer that splits tokens at whitespace and splits off punctuation at the beginning and end of non-whitespace regions as separate tokens as well. It makes use of the Python `re` module for regular expressions to specify the splitting process. Look over the code and make sure you understand what's going on. You might find this online tool useful.

In [ ]:
```
def punc_tokenize(str):
    return [tok for tok in re.split('(\W*?)\s+', str) if tok != '']
```

Now how many tokens are there in the text if tokenized in this way?

In [ ]:
```
#TODO
token_count_3 = ...
```

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

Normalization

This tokenization method counts 'Would' and 'would' (capitalized and uncapitalized) as separate types. Is that a good idea? This raises the issue of text *normalization*.

Define a function `normalize_token` that normalizes tokens by making them lowercase if at most the first character is uppercase. (Hints here and here. These are also listed in the hint cell at the top of the lab, so we'll mostly stop providing these hints from here on.)

In [ ]:
```
#TODO - implement normalize_token, which returns the normalized word for a s
def normalize_token(str):
    ...
```

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

Now define `text_tokenized` to be the sequence of normalized tokens as tokenized by `punc_tokenize`

In [ ]:
```
#TODO
text_tokenized = ...
```

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

How many types are there when tokenized and normalized in this way?

In [ ]:
```
#TODO
type_count_norm_punc = ...
```

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

Using prebuilt tokenizers

Tokenization and normalization are so commonly needed that many packages provide pre-built tokenizers of various sorts. We'll use one from the Natural Language Tool Kit (NLTK). The package has already been imported above under the name `nltk`.

Define two tokenizers, versions of `whitespace_tokenize` and a normalized version of `punc_tokenize` above, using the `nltk.tokenize.WhitespaceTokenizer` and `nltk.tokenize.word_tokenize` respectively. Note that `nltk.tokenize.word_tokenize` only tokenizes the string, so we normalize the string by lowering the string's characters.

In [ ]:
```
#TODO
def nltk_whitespace_tokenize(str):
    ...

def nltk_normpunc_tokenize(str):
    ...
```

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

We'll print out the last few tokens of the `text` tokenized by the whitespace tokenizer and the NLTK tokenizer to see the differences.

In [ ]:
```
print(nltk_whitespace_tokenize(text)[-10:])
print(nltk_normpunc_tokenize(text)[-10:])
```

> *Meta-comment:* Because it's important that you get practice both with implementing the ideas in the course from first principles and also with using prebuilt software that provides similar functionality, we'll often have you engage in this seemingly redundant process of first implementing a small example and then applying a prebuilt method to do much the same thing. The effort may be duplicative, but it is not wasted.

In the next section, it will be helpful to have the tokenized text available. Define `nltk_text_tokenized` to be the sequence of normalized tokens of the sample

`text` as tokenized by the tokenizer `nltk_normpunc_tokenize` that you've just written.

In [ ]:
```
#TODO
nltk_text_tokenized = ...
```

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

Representing words

In this section, we'll explore some simple representations for tokens, as a step on the way to representing texts – sentences or documents:

String encoding

We've already seen string encoding above, representing a token of a word type by a string specific to that type: a token 'green' represented by an instance of the Python string `'green'`, for instance, or 'Sam-I-am' represented by `'Sam-I-am'`. So let's move on.

1-hot encoding

Given a vocabulary for a language, we can associate each type with an integer, say by its index in a vector. We've already imported the `torch` module; we'll use `torch` tensors for the index vector. For the Seuss text, we can use the following list named `vocabulary` to represent the ordered vocabulary:

In [ ]:
```
vocabulary = sorted(set(nltk_text_tokenized))
vocabulary
```

A digression on `torch` tensors

Recall that `torch` tensors allow for vectorized computations: many operations on them work *componentwise*, that is, separately for each component of the tensor, rather than on the tensor all at once. Compare the following two operations, first on lists, then on `torch` tensors.

In [ ]:
```
[1, 2] == 1
```

In [ ]:
```
torch.tensor([1, 2]) == 1
```

This behavior of tensors is quite powerful, allowing for simply specifying complex operations and for efficient, even parallelizable, computation of them. You'll want ot take

advantage of these characteristics of tensors where possible, here and in future assignments.

But back to the 1-hot representation.

In the *1-hot representation* of words, a token is then represented by a bit vector (again given as a `torch` tensor), with a 1 at the index of the token's type. (For consistency with some later `torch` functions, we'll take the elements to be floats rather than ints. The `.type` method is useful for converting the type, and it even conveniently broadcasts componentwise over the tensors.) For instance, the 1-hot representation of the comma token ',' would be

In [ ]:
```
torch.tensor([1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]).type(torch.float32)
```

Conversion back and forth between these various representations is useful. Define functions `str_to_onehot` and `onehot_to_str` that convert between the string and one-hot representations using a vocabulary array to define the conversion.

Ideally, in your implementation, you'll want to take advantage of the componentwise nature of many tensor operations discussed above.

In [ ]:
```
#TODO
def str_to_onehot(wordtype, vocab):
    """Returns the 1-hot representation of `wordtype` with vocabulary
    `vocab`.
    The returned value should be a `torch.tensor` with data type `float`.
    """
    ...

def onehot_to_str(onehot, vocab):
    """Returns the string representation of `onehot`, a one-hot
    representation of a word type, with vocabulary `vocab`.
    """
    ...
```

Now use `str_to_onehot` to define the variable `anywhere_1hot` to be the 1-hot representation for a token of the type 'anywhere'.

In [ ]:
```
#TODO
anywhere_1hot = ...
```

In [ ]:
```
grader.check("anywhere_1hot")
```

You can verify that the conversion worked correctly by inverting it using `onehot_to_str`, which we've done in the following unit test.

In [ ]:
```
grader.check("anywhere_1hot_reverse")
```

Representing texts

The set-of-words representation

We can represent a whole *text* (a sequence of words) by manipulating the vector representations of the words within the text. For instance, we can take the componentwise maximum of the vectors. We refer to this as the *set-of-words* representation.

Here we've defined a function `set_of_words` that returns the set of words representation for a token sequence.

In [ ]:
```
def set_of_words(tokens, vocabulary):
    """Returns the set-of-words representation as a tensor of floats for the
    sequence of `tokens` using the `vocabulary` to specify the conversion.
    """
    onehots = torch.stack([str_to_onehot(token, vocabulary) for token in tok
    return torch.amax(onehots, 0).type(torch.float32)
```

The set-of-words representation for a text is a vector that has a `1` for each word type that occurs in the text. The vector represents the characteristic function for the subset of vocabulary words that appear in the text; hence the term 'set of words'.

What is the set-of-words representation for the example text 'I would not, would not, here or there.'?

Define the variable `example_sow` to be the set of words representation for the example text 'I would not, would not, here or there.' Use the `nltk_normpunc_tokenize` tokenizer.

In [ ]:
```
#TODO
example_sow = ...
```

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

The bag of words representation

If instead, we take the componentwise *sum* of the vectors instead of the maximum, the text representation provides the *frequency* of each word type in the text. We refer to this representation as the *bag of words* representation.

Define a function `bag_of_words`, analogous to `set_of_words` above, that returns the bag-of-words representation for a token sequence.

In [ ]:
```
#TODO
def bag_of_words(tokens, vocabulary):
    ...
```

What is the bag of words representation for the example text 'I would not, would not, here or there.'?

In [ ]:
```
#TODO - define the variable to be the bag of words representation for the ex
# 'I would not, would not, here or there.'
# Use the `nltk_normpunc_tokenize` tokenizer
example_bow = ...
```

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

Document similarity metrics

Consider the following text classification problem: Each sentence in *Green Eggs and Ham* is spoken by one of two characters, Sam-I-am and Guy-am-I. We want to be able to classify new sentences as (most likely) being uttered by one of the two.

A simple method for text classification is the *nearest neighbor* method. We select the class for the new sentence that is the same as the class of the "nearest" (most similar) sentence for which we already know the class. (You'll experiment much more with this text classification method in the next lab.)

To perform nearest neighbor classification, we need a method for measuring the (metaphorical) distance between two texts based on their representations. We'll explore a few methods here:

- Hamming distance
- Jaccard distance
- Euclidean distance
- cosine distance

You'll implement code for all of these distance metrics. Try to implement the functions using `torch` tensor functions only, without explicit iteration over the elements in the vector.

We'll take a look at the distances among the following sentences:

1. Would you like them here or there?
2. I would not like them here or there.
3. Do you like green eggs and ham?
4. I do not like them Sam-I-am.

We'll start with the set of words representations of these sentences:

In [ ]:
```
examples = """Would you like them here or there?
              I would not like them here or there.
              Do you like green eggs and ham?
              I do not like them Sam-I-am.""" \
           .split("\n")
sows = [set_of_words(nltk_normpunc_tokenize(sentence), vocabulary)
            for sentence in examples]
```

In [ ]:
```
sows
```

Hamming distance

The Hamming distance between two vectors is the number of positions at which they differ. Define a function `hamming_distance` that computes the Hamming distance between two vectors. The returned value should be a (tensorized) integer.

In [ ]:
```
#TODO
def hamming_distance(v1, v2):
    ...
```

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

Now we can generate the Hamming distances among all of the sample sentences in a little table. Do the values make sense? What does the diagonal tell you?

In [ ]:
```
for i in range(4):
    for j in range(4):
        print(f"{hamming_distance(sows[i], sows[j]):4} ", end='')
    print()
```

Jaccard distance

The Jaccard distance between two sets (and remember that these bit strings basically represent sets) is one minus the number of elements in their intersection divided by the number of elements in their union.

$$D_{jaccard}(v_1, v_2) = 1 - \frac{|v_1 \cap v_2|}{|v_1 \cup v_2|}$$

Define a function `jaccard_distance` to compute the Jaccard distance between two set-of-words representations. The returned value should be a (tensorized) float.

In [ ]:
```
#TODO
def jaccard_distance(v1, v2):
    ...
```

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

Again, here's a table of the Jaccard distances among the sample sentences.

In [ ]:
```
for i in range(4):
    for j in range(4):
        print(f"{jaccard_distance(sows[i], sows[j]):5.3f} ", end='')
    print()
```

Euclidean distance

The Euclidean distance between two vectors is the norm of the vector between them, that is,

$$D_{euclidean}(\mathbf{x}, \mathbf{y}) = |\mathbf{x} - \mathbf{y}|$$

where $|\mathbf{z}|$, the norm of a vector $\mathbf{z}$, is calculated as

$$|\mathbf{z}| = \sqrt{\sum_{i=1}^{N} \mathbf{z}_i^2}$$

Fortunately, `torch` provides the function `torch.linalg.norm` to compute the norm, and the vector between two vectors can be computed by componentwise subtraction.

Define a function `euclidean_distance` to compute the Euclidean distance between two vectors. (For the time being, try to implement it directly without using `torch.linalg.norm`.)

In [ ]:
```
#TODO
def euclidean_distance(v1, v2):
    ...
```

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

Again, here's a table of the Euclidean distances among the sample sentences.

In [ ]:
```
for i in range(4):
    for j in range(4):
        print(f"{euclidean_distance(sows[i], sows[j]):5.3f} ", end='')
    print()
```

Cosine distance

The *cosine similarity* of two vectors of length N is the cosine of the angle that they form, which is computed as the dot product of the two vectors divided by their norms.

$$cos(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{N} \mathbf{x}_i \cdot \mathbf{y}_i}{|\mathbf{x}| \cdot |\mathbf{y}|}$$

This isn't a distance metric, but a similarity metric. For vectors of non-negative numbers, it ranges from 0 to 1, where 0 is maximally different and 1 is maximally similar. To turn it into a distance metric, then, we take the inverse cosine (to convert the cosine to an angle between π and 0) and divide by π.

$$D_{cosine}(\mathbf{x}, \mathbf{y}) = \frac{cos^{-1}(cos(\mathbf{x}, \mathbf{y}))}{\pi}$$

Since we're using `torch`, some of these functions are already provided. See hints here, here, and here.

(To avoid some math domain errors, we recommend that you use the function `safe_acos` that we've provided to compute the inverse cosine function instead of using `math.acos` directly.)

In [ ]:
```
def safe_acos(x):
    """Returns the arc cosine of `x`. Unlike `math.acos`, it
       does not raise an exception for values of `x` out of range,
       but rather clips `x` at -1..1, thereby avoiding math domain
       errors in the case of numerical errors."""
    return math.acos(math.copysign(min(1.0, abs(x)), x))
```

In [ ]:
```
#TODO
def cosine_distance(v1, v2):
    """Returns the cosine distance between two vectors"""
    ...
```

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

In [ ]:
```
for i in range(4):
    for j in range(4):
        print(f"{cosine_distance(sows[i], sows[j]):5.3f} ", end='')
    print()
```

In the next lab, you'll use some of these distance metrics to automatically classify text using nearest neighbor classification.

Lab debrief

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

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

Type your answer here, replacing this text.

Submission Instructions

This lab should be submitted to Gradescope at http://go.cs187.info/lab1-1-submit, which will be made available some time before the due date.

Make sure that you have passed all public tests by running `grader.check_all()` below before submitting. Note that there are hidden tests on Gradescope, the results of which will be revealed after the submission deadline.

End of lab 1-1

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

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