Assignment 1: Basic Text Processing

In this assignment, we will carry out some basic text processing tasks:

- computing word frequencies and investigating some statistical properties of text;
- encoding words as integer as a preprocessing step for machine learning;
- trying out **spaCy**, and off-the-shelf NLP library for Python.

The purpose of this assignment is to serve as a warmup for the rest of the course. The text processing and frequency computations will give you a bit of experience working with text files and introduce you to some basic working methods in statistical investigations of texts. The integer preprocessing step is necessary in almost all machine learning software for text. Finally, the quick tour of spaCy will introduce us some classic NLP tasks such as part-of-speech tagging and named entity recognition.

This is a **group assignment** and your work should be carried out in a group of 2 to 4 people. If you don't have a group partner, please ask around in the Discussion tool in Canvas. Please include **the names of the group members** in your submission.

Deadline: November 12.

Submission: Please use the Canvas page to submit your assignment. It is enough if one group member submits the solution, but again please don't forget to include the names of the group members.

You should submit code and a written report. The code can be sent as one or more Python files, Jupyter notebooks, or a link to a Colab notebook.

The report should be a pdf, Word or LibreOffice document, or be included in a notebook as markdown cells. Regardless of whether you choose to submit a separate report or to put the text into a notebook, make sure that the text is concise but well-written. (For a notebook, it should in principle be possible to extract the text cells and get a coherent report.) Read the assignment instructions carefully, and if you're asked to discuss something, make sure it's covered in your report.

If you have questions, please ask Newton, Tobias or Richard, preferably during the assistance sessions (Monday 10–12 or Wednesday 13–15).

Preparation: installing libraries

Read about how to install spaCy using pip or conda on this page. This toolkit can work with different languages, so make sure you also install the model file `en_core_web_sm` required to process English text.

You will also need to draw a couple of plots. We recommend that you use the matplotlib library for this, but if you're more familiar with some other environment (e.g. R or Matlab) you are allowed to use it.

If you're using Colab, no installation is necessary since these libraries (and the English spaCy model) are installed by default.

Getting the data

Download this zip file and decompress it. This will give you a directory containing three text corpora: a part of the English Wikipedia (`wikipedia.txt`), a collection of Amazon book reviews (`books.txt`), and English translations of the European Parliament proceedings (`europarl.txt`).

Note for Colab users: downloading large files to the local file system in the virtual machine can be a bit tricky because the connection can be unreliable. It will probably be faster if you upload the zip file to a Drive account and follow the instructions here for mounting a Drive directory. (This also has the advantage that you don't have to download the file again next time you run the notebook.) Then you can unzip the file into your local file system (e.g. `!unzip 'drive/My Drive/SOME_PATH/a1_data.zip'`).

There are some differences in the way that the files have been preprocessed. As you can see, the text of all three corpora has been tokenized: that is, the text has been split into separate words and the punctuation separated from the words. In addition, the text in the book review corpus has been lowercased. The European Parliament and Wikipedia corpora are encoded in UTF-8, while the review corpus is encoded in ISO-8859-1. (In sections 5 and 6 of this document, there is a refresher on string encodings).

Warmup: computing word frequencies

`Counter` and `defaultdict`, both in the standard package `collections`, are two built-in data structure classes in Python that are useful for computing frequencies.

The `Counter` is an extension of the standard dictionary type that includes some utilites to simplify frequency computations. To exemplify, here is a small program that reads a text file and calculates the frequencies of the 10 most common words in that file. (To make this code more compact, we could have used the method `update` instead of going through the words one by one.)

```
freqs = Counter()
with open(YOUR_FILE, encoding=ENCODING) as f:
    for line in f:
        tokens = line.lower().split()
        for token in tokens:
            freqs[token] += 1
for word, freq in freqs.most_common(10):
    print(word, freq)
```

Note that we call a simple `split` here: we can do this because the files that we work with have been tokenized.

Run this piece of code and see which words are the most frequent in the three datasets. (It might be useful to use the method `most_common` in the `Counter` here.)

Another common trick is the "dictionary-within-dictionary" pattern. For instance, let's assume that we want to be able to ask questions such as what are the words that occur most frequently following the word X? In such cases, we want to maintain different frequency tables for different words.

In Python, a `defaultdict` combined with `Counter` makes it easy to implement such solutions. Like with the `Counter`, this type of dictionary will automatically handle situations where we encounter a key we haven't seen before: in such case, it will create a new frequency table.

```
freqs = defaultdict(Counter)
with open(YOUR_FILE, encoding=ENCODING) as f:
    for line in f:
        tokens = line.lower().split()
        for t1, t2 in zip(tokens, tokens[1:]):
            freqs[t1][t2] += 1
```

Try out this piece of code and find the words that most frequently follow the word red.

Investigating the word frequency distribution

To get an intuition for the statistical distribution of words, make a plot where you compare a word's rank in the frequency table to its frequency. This type of plot is called a rank/frequency plot. This plot will probably look a bit better if you use e.g. the 100 most common words instead of all the words.

Hint: with matplotlib, you can use the function `pyplot.plot`.

Reflection: What consequences do you think this distribution has for machine learning systems that work with language?

Draw the plot again, now using a log scale on the X and Y axes (for instance, by using `pyplot.loglog`). Read about Zipf's law and discuss how well you think your observed distribution corresponds to this model.

Comparing corpora

What words are "typical" of the European Parliament corpus when we compare it to the book review corpus, or vice versa? You will have to come up with your own operationalization of the notion of "typical" here.

You can use any approach that you want as long as you can justify it, but you can't use any external software or libraries (except possibly mathematical libraries such as NumPy).

Side show: preprocessing text for machine learning

When we build machine learning systems that process text, we typically need to encode the text in a numerical format. This can be done in different ways, depending on what information we want to use from the text. In this exercise, we will encode each word as an integer. (In some applications, we might also want to encode parts of words, characters, or other information.)

Furthermore, when we work with neural networks, for efficiency reasons we typically want to process several texts in parallel: "batching". We will then convert a small set of texts into a matrix where each row represents one text (e.g. a document or sentence). To make sure that we can fit the texts into a rectangular matrix, we will have to ensure that they all have the same length, and we add special dummy symbols ("padding") at the end of the texts that are shorter than the longest one in the batch.

Your job now is to write a utility that carries out the following steps:

1. Given a text file, build a vocabulary (that is: a string-to-integer mapping), where the vocabulary size is at most `max_voc_size`. If the number of distinct words is greater than this size, you should just use the most frequent words.
2. Given a text file, go through the file line by line and divide the data into batches that you encode as integers using the mapping that you defined in the first step. Out-of-vocabulary words should be mapped to a special dummy symbol (e.g. `<OTHER>`). The batch size should be at most `batch_size`. Lines that are shorter than the longest line in the batch should be padded. The batch should be stored as a NumPy array or PyTorch tensor.

For instance, if we implement the vocabulary management in a class called `Vocab`, here is an example of how it hypothetically might be used:

```
dataset = ... something ...

with open(dataset) as f:
    voc = Vocab(max_voc_size=1000, batch_size=8)

    # go through the lines and build the vocabulary
    voc.build_vocab(f)

with open(dataset) as f:
    for b in voc.batches(f):
        # b is a matrix of shape (max_length, batch_size)
        # where max_length is the length of the longest
        # line in the batch
        print(b)
```

It is enough if you submit the code for this part of the assignment. Since you just have to code and there's not much else to say here, you don't have to describe this part in your technical report.

Trying out an NLP toolkit

Now, let's turn from the basic word counting we have been dealing with until now, and explore some of the most classical NLP tasks such as part-of-speech tagging, parsing, and named entity extraction. During the course, we will see how we can design machine learning systems for these tasks. For now, we will just use an off-the-shelf toolkit: **spaCy**. This toolkit includes basic linguistic analysis components for a number of languages.

The following code will load the functionality for spaCy to carry out some basic analysis of English text:

```
import spacy
nlp = spacy.load('en_core_web_sm')
```

Processing a text

When we have loaded the English spaCy module, it can be called directly with a string input:

```
example = 'Apple bought two companies this year and no one knew, Mark Gurman at 9to5Mac reports.'
result = nlp(example)
```

Using the default configuration of spaCy, this command will carry out the following steps in a sequence:

- **Tokenization:** splitting the text into separate words, and separating punctuation
- **Part-of-speech tagging:** labeling each token with a part-of-speech tag (noun, verb etc.)
- **Entity recognition:** finding mentions of names of people, locations, organizations, ...
- **Dependency parsing:** determining grammatical relations between the tokens in the sentence (subject, object etc.)

Such a sequence of processing steps is traditionally called a **pipeline**.

As you can see in the documentation, there are some additional processing modules that can be used optionally.

Visualizing the results

If you are running a notebook, you can visualize the output directly. Here is some code that will show the named entities in the text:

```
from spacy import displacy
displacy.render(result, style='ent', jupyter=True)
```

To show the part-of-speech tags and grammatical dependency relations, write `'dep'` instead of `'ent'`. (If you want to understand more about what the dependency graph represent, you can take a look at this description.)

If you are running a standalone Python program:

```
html = displacy.render(doc, style='ent', page=True)
```

Accessing the results in your code

For interactive purposes it's nice to draw the result, but in practice we will need to access the result from our code so that we can process it further.

The following piece of code goes through all the tokens in a document:

```
for token in result:
    do_something(token)
```

Each token stores various information from the various stages of the analysis pipeline. Here are some of the attributes that we can access in a token. (Note the somewhat inconsistent naming scheme: some of the attribute names end with an underscore.)

- `token.text`, the text of the token: for instance, the fourth of the tokens in the previous example is `"companies"`
- `token.pos_`, the part-of-speech tag: `"NOUN"` for the same token
- `token.lemma_`, the uninflected base form or dictionary form: `"company"`
- `token.head`, a reference to the parent token in the dependency graph: `"bought"`
- `token.dep_`, the label on the edge in the dependency tree between the head and this token: `"dobj"` (direct object)

Similarly, we can process the named entities:

```
for entity in result.ents:
    do_something(entity)
```

And for each entity, there are some attributes we can access:

- `entity.text`, the text of the entity mention: `"Mark Gurman"`
- `entity.label_`, the entity label: `"PERSON"`

Apply the NLP pipeline to some example text and check that you can extract some basic information from the tokens and named entities. If you try out the example given here, do you think the analysis is entirely correct?

Additional questions

Now that we have seen how to do some basic counting and then how to use spaCy for some fundamental language processing tasks, let's combine the pieces. Write some code to investigate the following questions:

- Which are the most frequent nouns in the book review corpus?
- Which are the most frequently mentioned countries in the Wikipedia corpus?
- What are the items that people drink most frequently in the European Parliament corpus?

If you think it takes too much time to process a full file, you may use a subset. (For the third task, you can speed up processing if you keep in mind that you don't have to call the full spaCy pipeline for lines that don't contain the word drink or one of its inflected forms.)
