Parts-of-Speech Tagging - First Steps: Working with text files, Creating a Vocabulary and Handling Unknown Words

In this lecture notebook you will create a vocabulary from a tagged dataset and learn how to deal with words that are not present in this vocabulary when working with other text sources. Aside from this you will also learn how to:

- read text files
- work with defaultdict
- work with string data

In [1]:

```
import string
from collections import defaultdict
```

Read Text Data

A tagged dataset taken from the Wall Street Journal is provided in the file `WSJ_02-21.pos`.

To read this file you can use Python's context manager by using the `with` keyword and specifying the name of the file you wish to read. To actually save the contents of the file into memory you will need to use the `readlines()` method and store its return value in a variable.

Python's context managers are great because you don't need to explicitly close the connection to the file, this is done under the hood:

In [2]:

```
# Read lines from 'WSJ_02-21.pos' file and save them into the 'lines' variable
with open("WSJ_02-21.pos", 'r') as f:
    lines = f.readlines()
```

To check the contents of the dataset you can print the first 5 lines:

In [3]:

```
# Print columns for reference
print("\t\tWord", "\tTag\n")

# Print first five lines of the dataset
for i in range(5):
    print(f'line number {i+1}: {lines[i]}')
```

```
  Word  Tag

line number 1: In IN

line number 2: an DT

line number 3: Oct. NNP

line number 4: 19 CD

line number 5: review NN
```

Each line within the dataset has a word followed by its corresponding tag. However since the printing was done using a formatted string it can be inferred that the **word** and the **tag** are separated by a tab (or some spaces) and there is a newline at the end of each line (notice that there is a space between each line).

If you want to understand the meaning of these tags you can take a look here.

To better understand how the information is structured in the dataset it is recommended to print an unformatted version of it:

In [4]:

```
# Print first line (unformatted)
lines[0]
```

Out[4]:

```
'In\tIN\n'
```

Indeed there is a tab between the word and the tag and a newline at the end of each line.

Creating a vocabulary

Now that you understand how the dataset is structured, you will create a vocabulary out of it. A vocabulary is made up of every word that appeared at least 2 times in the dataset. For this, follow these steps:

- Get only the words from the dataset
- Use a defaultdict to count the number of times each word appears
- Filter the dict to only include words that appeared at least 2 times
- Create a list out of the filtered dict
- Sort the list

For step 1 you can use the fact that every word and tag are separated by a tab and that words always come first. Using list comprehension the words list can be created like this:

In [5]:

```
# Get the words from each line in the dataset
words = [line.split('\t')[0] for line in lines]
```

Step 2 can be done easily by leveraging `defaultdict`. In case you aren't familiar with defaultdicts they are a special kind of dictionaries that **return the "zero" value of a type if you try to access a key that does not exist** . Since you want the frequencies of words, you should define the defaultdict with a type of `int`.

Now you don't need to worry about the case when the word is not present within the dictionary because getting the value for that key will simply return a zero. Isn't that cool?

In [6]:

```
# Define defaultdict of type 'int'
freq = defaultdict(int)

# Count frequency of ocurrence for each word in the dataset
for word in words:
    freq[word] += 1
```

Filtering the `freq` dictionary can be done using list comprehensions again (aren't they handy?). You should filter out words that appeared only once and also words that are just a newline character:

In [7]:

```
# Create the vocabulary by filtering the 'freq' dictionary
vocab = [k for k, v in freq.items() if (v > 1 and k != '\n')]
```

Finally, the `sort` method will take care of the final step. Notice that it changes the list directly so you don't need to reassign the `vocab` variable:

In [8]:

```
# Sort the vocabulary
vocab.sort()

# Print some random values of the vocabulary
for i in range(4000, 4005):
```

```
    print(vocab[1])
```

```
Early
Earnings
Earth
Earthquake
East
```

Now you have successfully created a vocabulary from the dataset. **Great job!** The vocabulary is quite extensive so it is not printed out but you can still do so by creating a cell and running something like `print(vocab)`.

At this point you will usually write the vocabulary into a file for future use, but that is out of the scope of this notebook. If you are curious it is very similar to how you read the file at the beginning of this notebook.

Processing new text sources

Dealing with unknown words

Now that you have a vocabulary, you will use it when processing new text sources. **A new text will have words that do not appear in the current vocabulary**. To tackle this, you can simply classify each new word as an unknown one, but you can do better by creating a function that tries to classify the type of each unknown word and assign it a corresponding `unknown token`.

This function will do the following checks and return an appropriate token:

- Check if the unknown word contains any character that is a digit
  - return `--unk_digit--`
- Check if the unknown word contains any punctuation character
  - return `--unk_punct--`
- Check if the unknown word contains any upper-case character
  - return `--unk_upper--`
- Check if the unknown word ends with a suffix that could indicate it is a noun, verb, adjective or adverb
  - return `--unk_noun--`, `--unk_verb--`, `--unk_adj--`, `--unk_adv--` respectively

If a word fails to fall under any condition then its token will be a plain `--unk--`. The conditions will be evaluated in the same order as listed here. So if a word contains a punctuation character but does not contain digits, it will fall under the second condition. To achieve this behaviour some if/elif statements can be used along with early returns.

This function is implemented next. Notice that the `any()` function is being heavily used. It returns `True` if at least one of the cases it evaluates is `True`.

In [9]:

```
def assign_unk(word):
    """
    Assign tokens to unknown words
    """

    # Punctuation characters
    # Try printing them out in a new cell!
    punct = set(string.punctuation)

    # Suffixes
    noun_suffix = ["action", "age", "ance", "cy", "dom", "ee", "ence", "er", "hood", "ion", "ism",
"ist", "ity", "ling", "ment", "ness", "or", "ry", "scape", "ship", "ty"]
    verb_suffix = ["ate", "ify", "ise", "ize"]
    adj_suffix = ["able", "ese", "ful", "i", "ian", "ible", "ic", "ish", "ive", "less", "ly", "ous"
]
    adv_suffix = ["ward", "wards", "wise"]

    # Loop the characters in the word, check if any is a digit
    if any(char.isdigit() for char in word):
        return "--unk_digit--"

    # Loop the characters in the word, check if any is a punctuation character
    elif any(char in punct for char in word):
        return "--unk_punct--"

    # Loop the characters in the word, check if any is an upper case character
```

```
    elif any(char.isupper() for char in word):
        return "--unk_upper--"

    # Check if word ends with any noun suffix
    elif any(word.endswith(suffix) for suffix in noun_suffix):
        return "--unk_noun--"

    # Check if word ends with any verb suffix
    elif any(word.endswith(suffix) for suffix in verb_suffix):
        return "--unk_verb--"

    # Check if word ends with any adjective suffix
    elif any(word.endswith(suffix) for suffix in adj_suffix):
        return "--unk_adj--"

    # Check if word ends with any adverb suffix
    elif any(word.endswith(suffix) for suffix in adv_suffix):
        return "--unk_adv--"

    # If none of the previous criteria is met, return plain unknown
    return "--unk--"
```

A POS tagger will always encounter words that are not within the vocabulary that is being used. By augmenting the dataset to include these `unknown word tokens` you are helping the tagger to have a better idea of the appropriate tag for these words.

Getting the correct tag for a word

All that is left is to implement a function that will get the correct tag for a particular word taking special considerations for unknown words. Since the dataset provides each word and tag within the same line and a word being known depends on the vocabulary used, these two elements should be arguments to this function.

This function should check if a line is empty and if so, it should return a placeholder word and tag, `--n--` and `--s--` respectively.

If not, it should process the line to return the correct word and tag pair, considering if a word is unknown in which scenario the function `assign_unk()` should be used.

The function is implemented next. Notice That the `split()` method can be used without specifying the delimiter, in which case it will default to any whitespace.

In [10]:

```
def get_word_tag(line, vocab):
    # If line is empty return placeholders for word and tag
    if not line.split():
        word = "--n--"
        tag = "--s--"
    else:
        # Split line to separate word and tag
        word, tag = line.split()
        # Check if word is not in vocabulary
        if word not in vocab:
            # Handle unknown word
            word = assign_unk(word)
    return word, tag
```

Now you can try this function with some examples to test that it is working as intended:

In [11]:

```
get_word_tag('\n', vocab)
```

Out[11]:

```
('--n--', '--s--')
```

Since this line only includes a newline character it returns a placeholder word and tag.

In [12]:

In [12]:

```
get_word_tag('In\tIN\n', vocab)
```

Out[12]:

```
('In', 'IN')
```

This one is a valid line and the function does a fair job at returning the correct (word, tag) pair.

In [13]:

```
get_word_tag('tardigrade\tNN\n', vocab)
```

Out[13]:

```
('--unk--', 'NN')
```

This line includes a noun that is not present in the vocabulary.

The `assign_unk` function fails to detect that it is a noun so it returns an `unknown token`.

In [14]:

```
get_word_tag('scrutinize\tVB\n', vocab)
```

Out[14]:

```
('--unk_verb--', 'VB')
```

This line includes a verb that is not present in the vocabulary.

In this case the `assign_unk` is able to detect that it is a verb so it returns an `unknown verb token`.

Congratulations on finishing this lecture notebook! Now you should be more familiar with working with text data and have a better understanding of how a basic POS tagger works.

Keep it up!