Parts-of-Speech Tagging - Working with tags and Numpy

In this lecture notebook you will create a matrix using some tag information and then modify it using different approaches. This will serve as hands-on experience working with Numpy and as an introduction to some elements used for POS tagging.

In [1]:

```
import numpy as np
import pandas as pd
```

Some information on tags

For this notebook you will be using a toy example including only three tags (or states). In a real world application there are many more tags which can be found here.

In [2]:

```
# Define tags for Adverb, Noun and To (the preposition) , respectively
tags = ['RB', 'NN', 'TO']
```

In this week's assignment you will construct some dictionaries that provide useful information of the tags and words you will be working with.

One of these dictionaries is the `transition_counts` which counts the number of times a particular tag happened next to another. The keys of this dictionary have the form `(previous_tag, tag)` and the values are the frequency of occurrences.

Another one is the `emission_counts` dictionary which will count the number of times a particular pair of `(tag, word)` appeared in the training dataset.

In general think of `transition` when working with tags only and of `emission` when working with tags and words.

In this notebook you will be looking at the first one:

In [4]:

```
# Define 'transition_counts' dictionary
# Note: values are the same as the ones in the assignment
transition_counts = {
    ('NN', 'NN'): 16241,
    ('RB', 'RB'): 2263,
    ('TO', 'TO'): 2,
    ('NN', 'TO'): 5256,
    ('RB', 'TO'): 855,
    ('TO', 'NN'): 734,
    ('NN', 'RB'): 2431,
    ('RB', 'NN'): 358,
    ('TO', 'RB'): 200
}
```

Notice that there are 9 combinations of the 3 tags used. Each tag can appear after the same tag so you should include those as well.

Using Numpy for matrix creation

Now you will create a matrix that includes these frequencies using Numpy arrays:

In [5]:

```
# Store the number of tags in the 'num_tags' variable
num_tags = len(tags)

# Initialize a 3X3 numpy array with zeros
transition_matrix = np.zeros((num_tags, num_tags))
```

```
# Print matrix
transition_matrix
```

Out[5]:

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Visually you can see the matrix has the correct dimensions. Don't forget you can check this too using the `shape` attribute:

In [6]:

```
# Print shape of the matrix
transition_matrix.shape
```

Out[6]:

```
(3, 3)
```

Before filling this matrix with the values of the `transition_counts` dictionary you should sort the tags so that their placement in the matrix is consistent:

In [7]:

```
# Create sorted version of the tag's list
sorted_tags = sorted(tags)

# Print sorted list
sorted_tags
```

Out[7]:

```
['NN', 'RB', 'TO']
```

To fill this matrix with the correct values you can use a `double for loop`. You could also use `itertools.product` to one line this double loop:

In [8]:

```
# Loop rows
for i in range(num_tags):
    # Loop columns
    for j in range(num_tags):
        # Define tag pair
        tag_tuple = (sorted_tags[i], sorted_tags[j])
        # Get frequency from transition_counts dict and assign to (i, j) position in the matrix
        transition_matrix[i, j] = transition_counts.get(tag_tuple)

# Print matrix
transition_matrix
```

Out[8]:

```
array([[1.6241e+04, 2.4310e+03, 5.2560e+03],
       [3.5800e+02, 2.2630e+03, 8.5500e+02],
       [7.3400e+02, 2.0000e+02, 2.0000e+00]])
```

Looks like this worked fine. However the matrix can be hard to read as `Numpy` is more about efficiency, rather than presenting values in a pretty format.

For this you can use a `Pandas DataFrame`. In particular, a function that takes the matrix as input and prints out a pretty version of it will be very useful:

In [9]:

```
# Define 'print_matrix' function
def print_matrix(matrix):
```

```
    print(pd.DataFrame(matrix, index=sorted_tags, columns=sorted_tags))
```

Notice that the tags are not a parameter of the function. This is because the `sorted_tags` list will not change in the rest of the notebook so it is safe to use the variable previously declared. To test this function simply run:

In [10]:

```
# Print the 'transition_matrix' by calling the 'print_matrix' function
print_matrix(transition_matrix)
```

```
          NN       RB       TO
NN  16241.0  2431.0  5256.0
RB    358.0  2263.0   855.0
TO    734.0   200.0     2.0
```

That is a lot better, isn't it?

As you may have already deducted this matrix is not symmetrical.

Working with Numpy for matrix manipulation

Now that you got the matrix set up it is time to see how a matrix can be manipulated after being created.

`Numpy` allows vectorized operations which means that operations that would normally include looping over the matrix can be done in a simpler manner. This is consistent with treating numpy arrays as matrices since you get support for common matrix operations. You can do matrix multiplication, scalar multiplication, vector addition and many more!

For instance try scaling each value in the matrix by a factor of $\frac{1}{10}$. Normally you would loop over each value in the matrix, updating them accordingly. But in Numpy this is as easy as dividing the whole matrix by 10:

In [11]:

```
# Scale transition matrix
transition_matrix = transition_matrix/10

# Print scaled matrix
print_matrix(transition_matrix)
```

```
         NN     RB     TO
NN  1624.1  243.1  525.6
RB    35.8  226.3   85.5
TO    73.4   20.0    0.2
```

Another trickier example is to normalize each row so that each value is equal to $\frac{value}{sum of row}$.

This can be easily done with vectorization. First you will compute the sum of each row:

In [12]:

```
# Compute sum of row for each row
rows_sum = transition_matrix.sum(axis=1, keepdims=True)

# Print sum of rows
rows_sum
```

Out[12]:

```
array([[2392.8],
       [ 347.6],
       [  93.6]])
```

Notice that the `sum()` method was used. This method does exactly what its name implies. Since the sum of the rows was desired the axis was set to `1`. In Numpy `axis=1` refers to the columns so the sum is done by summing each column of a particular row,

for each row.

Also the `keepdims` parameter was set to `True` so the resulting array had shape `(3, 1)` rather than `(3,)`. This was done so that the axes were consistent with the desired operation.

When working with Numpy, always remember to check the shape of the arrays you are working with, many unexpected errors happen because of axes not being consistent. The `shape` attribute is your friend for these cases.

In [13]:

```
# Normalize transition matrix
transition_matrix = transition_matrix / rows_sum

# Print normalized matrix
print_matrix(transition_matrix)
```

```
          NN        RB        TO
NN  0.678745  0.101596  0.219659
RB  0.102992  0.651036  0.245972
TO  0.784188  0.213675  0.002137
```

Notice that the normalization that was carried out forces the sum of each row to be equal to `1`. You can easily check this by running the `sum` method on the resulting matrix:

In [14]:

```
transition_matrix.sum(axis=1, keepdims=True)
```

Out[14]:

```
array([[1.],
       [1.],
       [1.]])
```

For a final example you are asked to modify each value of the diagonal of the matrix so that they are equal to the `log` of the sum of the current row plus the current value. When doing mathematical operations like this one don't forget to import the `math` module.

This can be done using a standard `for loop` or `vectorization`. You'll see both in action:

In [15]:

```
import math

# Copy transition matrix for for-loop example
t_matrix_for = np.copy(transition_matrix)

# Copy transition matrix for numpy functions example
t_matrix_np = np.copy(transition_matrix)
```

Using a for-loop

In [16]:

```
# Loop values in the diagonal
for i in range(num_tags):
    t_matrix_for[i, i] =  t_matrix_for[i, i] + math.log(rows_sum[i])

# Print matrix
print_matrix(t_matrix_for)
```

```
          NN        RB        TO
NN  8.458964  0.101596  0.219659
RB  0.102992  6.502088  0.245972
TO  0.784188  0.213675  4.541167
```

Using vectorization

In [17]:

```
# Save diagonal in a numpy array
d = np.diag(t_matrix_np)

# Print shape of diagonal
d.shape
```

Out[17]:

```
(3,)
```

You can save the diagonal in a numpy array using Numpy's `diag()` function. Notice that this array has shape `(3,)` so it is inconsistent with the dimensions of the `rows_sum` array which are `(3, 1)`. You'll have to reshape before moving forward. For this you can use Numpy's `reshape()` function, specifying the desired shape in a tuple:

In [18]:

```
# Reshape diagonal numpy array
d = np.reshape(d, (3,1))

# Print shape of diagonal
d.shape
```

Out[18]:

```
(3, 1)
```

Now that the diagonal has the correct shape you can do the vectorized operation by applying the `math.log()` function to the `rows_sum` array and adding the diagonal.

To apply a function to each element of a numpy array use Numpy's `vectorize()` function providing the desired function as a parameter. This function returns a vectorized function that accepts a numpy array as a parameter.

To update the original matrix you can use Numpy's `fill_diagonal()` function.

In [19]:

```
# Perform the vectorized operation
d = d + np.vectorize(math.log)(rows_sum)

# Use numpy's 'fill_diagonal' function to update the diagonal
np.fill_diagonal(t_matrix_np, d)

# Print the matrix
print_matrix(t_matrix_np)
```

```
          NN        RB        TO
NN  8.458964  0.101596  0.219659
RB  0.102992  6.502088  0.245972
TO  0.784188  0.213675  4.541167
```

To perform a sanity check that both methods yield the same result you can compare both matrices. Notice that this operation is also vectorized so you will get the equality check for each element in both matrices:

In [20]:

```
# Check for equality
t_matrix_for == t_matrix_np
```

Out[20]:

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Congratulations on finishing this lecture notebook! Now you should be more familiar with some elements used by a POS tagger such as the `transition_counts` dictionary and with working with Numpy.

Keep it up!