Clase 7

Repaso

Expresiones Regulares

() Ya hemos visto que los paréntesis nos ayudan a juntar múltiples caracteres en una solo expresión regular que queremos afectar con modificadores que normalmente afectan un solo caracter. Además de eso, todos los grupos que formamos con paréntesis son recordados por el sistema de expresiones regulares y podemos utilizar el texto con el que coinciden para nuevas coincidencias o para segmentar el texto que encontramos.

\# Para hacer uso del contenido de un grupo, utilizamos la diagonal invertida seguida de un número. Los grupos que se van utilizando se van enumerando de izquierda a derecha. La diagonal invertida cero (\0 NO EXISTE, si quieren hacer uso de un grupo con número de la expresión completa, pueden usar paréntesis cubriendo toda la expresión.

group(#) La función `group()` que ya habíamos visto anteriormente con el argumento 0, también nos puede mostrar grupos diferentes. Se respeta la misma numeración de los paréntesis dentro de la expresión regular.

sub() Las expresiones regulares tienen una función para sustituir las coincidencias que encuentren de la expresión regular sobre un texto. Esta función recibe como parámetro, en primer lugar, el texto con el que queremos sustituir las coincidencias de la expresión regular. En segundo lugar, el texto en el que queremos que se sustituyan. Y también podemos usar como un tercer parámetro opcional, cuántas sustituciones queremos que se hagan, por defecto hace todas las que encuentra.

Ejercicio 14

El último programa separa de las palabras los signos de puntuación que ocurren de su lado derecho.

- Usen el texto de salida anterior y un nuevo paso de proceso para separar los signos de puntuación izquierdos, de modo que queden las palabras y los signos completamente separados por espacios.

Solución

- Esta vez también les presentaré aquí la solución del ejercicio.

```
expresion_derecha=re.compile(r"([^\w\s][^\A-Z\d])")
texto_nuevo=expresion_derecha.sub(r" \1",texto)
expresion_izquierda=re.compile(r"([^\A-Z\d][^\w\s])")
texto_ultimo=expresion_izquierda.sub(r"\1 ",texto_nuevo)

print(texto_ultimo)
```

- Ya que usaremos ese nuevo texto con todo bien separado.

Expresiones Regulares

- ¿Recuerdan nuestro tokenizador básico?
- Tenía algunas fallas, pero ahora que las cosas están bien separadas por espacios, podemos lograr algo mejor.

```
expresion_derecha=re.compile(r"([^\w\s][^\A-Z\d])")
texto_nuevo=expresion_derecha.sub(r" \1",texto)
expresion_izquierda=re.compile(r"([^\A-Z\d][^\w\s])")
texto_ultimo=expresion_izquierda.sub(r"\1 ",texto_nuevo)

tokens_lista=texto_ultimo.split()

for token in tokens_lista:
        print(token)
```

Expresiones Regulares

- Esta nueva versión de tokenizador funciona mejor gracias a las expresiones regulares.
- Aún le faltan cosas para ser del todo confiable. Pero por ahora podemos ver cómo se van formando muchas de las reglas que componen un tokenizador completo, y además es importante conocer las expresiones regulares como herramienta.
- Por supuesto, los tokenizadores, al igual que otras herramientas del PLN, están bastante estudiados y los hay a nuestra disposición sin tener que armar uno nosotros mismos.

NLTK

NLTK

- NLTK (Natural language tool kit) es un módulo de Python que contiene herramientas para el manejo del lenguaje natural.
- Para usarlas, al igual que con los otros módulos, se tiene que importar.

```
import nltk
```

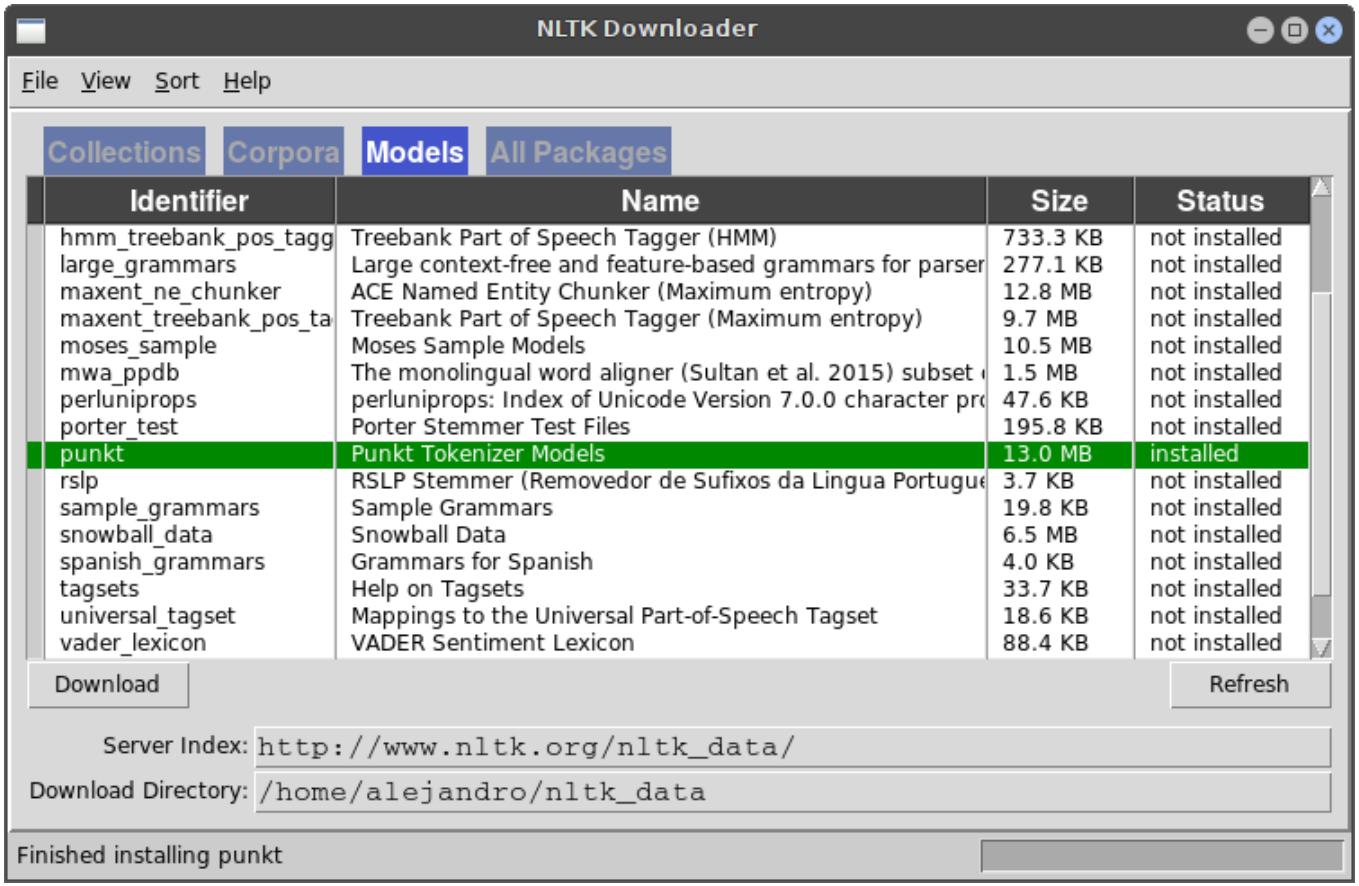
NLTK

- Como seguramente ya se imaginarán, NLTK tiene su tokenizador, así que comencemos por allí.
- Sin embargo, NLTK tiene tantas cosas, que las guarda en la nube para que sólo se descarguen las herramientas que cada quien necesita.
- Podemos abrir el "navegador" de NLTK desde el mismo Python. ¿Recuerdan cómo usar la consola de Python? Si no, lo pueden hacer también desde el editor:

```
import nltk

nltk.download()
```

NLTK



- Verán una pantalla como ésta, aquí está todo el contenido de nltk para descargar.
- Vayan a la sección de **Models** y descarguen **punkt**, que como verán, son modelos de tokenización.

NLTK

Tokenizador

- Y ahora que ya descargamos el tokenizador, podemos cerrar esa ventana y usarlo en nuestro programa.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
        texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

for token in tokens:
        print(token)
```

Conteo de palabras

o mejor dicho de tokens

- Así que ahora tenemos dos maneras de obtener una lista de tokens para nuestros textos.
- Veamos un par de usos, pueden ocupar la lista que más les guste.
- Para empezar, podemos usarlas para contar el total de palabras del documento.

```
# Aquí, nuestra lista de tokens se llama "tokens"

palabras_total=len(tokens)

print(palabras_total)
```

Palabras diferentes

- Además de las palabras totales, nos puede interesar conocer las palabras diferentes que utiliza un texto.
- Para esto, podemos utilizar el **`set`** de Python.
- Un **`set`** es un conjunto. Es similar a una lista, pero están pensados para hacer operaciones entre conjuntos (como uniones, intersecciones o diferencias). Y una característica particular que por ahora nos interesa bastante es que sus elementos no se repiten.
- Se usa la función **`set()`** para convertir una lista en un conjunto.

```
# Aquí, nuestra lista de tokens se llama "tokens"

tokens_conjunto=set(tokens)
palabras_diferentes=len(tokens_conjunto)

print(palabras_diferentes)
```

Riqueza léxica

- La riqueza léxica es la relación que existe entre la extensión de un texto y el número de palabras distintas que contiene.
- Así que ahora es muy fácil para nosotros calcularla.

```
# Aquí, nuestra lista de tokens se llama "tokens"

tokens_conjunto=set(tokens)

palabras_totales=len(tokens)
palabras_diferentes=len(tokens_conjunto)

riqueza_lexica=palabras_diferentes/palabras_totales

print(riqueza_lexica)
```

Funciones

en Python, claro

- Hemos visto muchas funciones. Ahora veamos que también podemos crearlas nosotros mismos.
- Hagamos una función que calcule la riqueza léxica si le damos una lista de tokens.

```
def riqueza_lexica(tokens):
    tokens_conjunto=set(tokens)

    palabras_totales=len(tokens)
    palabras_diferentes=len(tokens_conjunto)

    riqueza_lexica=palabras_diferentes/palabras_totales

    return riqueza_lexica
```

Funciones

```
import nltk

def riqueza_lexica(tokens):
        tokens_conjunto=set(tokens)
        palabras_totales=len(tokens)
        palabras_diferentes=len(tokens_conjunto)
        riqueza_lexica=palabras_diferentes/palabras_totales
        return riqueza_lexica

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
        texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")
riqueza_lexica=riqueza_lexica(tokens)
print(riqueza_lexica)
```

Ejercicio 15

- Modificar la función que se creó para que, en lugar de recibir una lista de tokens, reciba texto (el texto que se lea de un archivo, por ejemplo).
- NOTA: Se tendrán que agregar todos los pasos necesarios para que la función trabaje de manera correcta y que devuelva la riqueza léxica del texto que introduzcan.

Conteo individual

- También podemos usar la lista de tokens para hacer conteos de palabras individuales.
- Para esto, utilizamos la función **count()** de la lista.

```
# Aquí, nuestra lista de tokens se llama "tokens"

conteo_individual=tokens.count("el")

print(conteo_individual)

palabras_totales=len(tokens)
porcentaje=100*conteo_individual/palabras_totales

print(porcentaje,"%")
```

Funciones

- Algunas funciones reciben más de un dato. Cuando esto pasa se separan por comas dentro de los paréntesis.
- Cuando definimos funciones, también podemos definir más de un parámetro de entrada. De igual manera, separados por comas.

```
def porcentaje(palabra,texto):
    # Aquí va el programa
    # Se pueden usar las variables "palabra" y "texto"
    # al momento de usar la función, los parámetros se
    #   asignan por el orden. Es decir, el primero
    #   parámetro será "palabra", el segundo "texto".
    # Pueden definir los parametros como quieran
    # Solo asegurense de poner el orden correcto de los
    #   datos al momento de usar su función
```

Ejercicio 16

Para que quede bien claro el uso de funciones:

- Definir una función que calcule el porcentaje que ocupa una palabra dentro de un texto. Usen como parámetros de entrada la palabra y el texto.

NLTK

- De regreso a NLTK, también podemos usar los tokens para crear una variable de tipo **`Text`** de NLTK.
- Estos objetos tienen varias funciones útiles.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
        texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

texto_nltk=nltk.Text(tokens)
```

NLTK

Concordancias

- Las concordacias muestran todas las apariciones de una palabra junto con algo del texto que la rodea.
- Con la función **`concordance()`** del **`Text`** de NLTK se muestran las concordancias de una palabra.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
        texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

texto_nltk=nltk.Text(tokens)
texto_nltk.concordance("artículo")
```

RE

Concordancias

- Aprovecharé este punto para explicar un símbolo de las expresiones regulares que no había mencionado: las llaves {}
- También les mostraré cómo hacer concordancias con RE, por si quieren ampliar su funcionamiento o simplemente no cargar nltk.

```
import re

# Aqui va la lectura del archivo, por ahora la estoy omitiendo

expresion=re.compile(r".{,30}[\s^][Aa]rtículos? .{,30}")
resultados_busqueda=expresion.finditer(texto)

for resultado in resultados_busqueda:
        print(resultado.group(0))
```

NLTK

Palabras similares

- De nuevo en NLTK, veamos otra de las funciones que tiene su **`Text`**.
- Hemos visto que la función **`concordance()`** nos muestra una palabra en su contexto.
- Pero **`Text`** también tiene la función **`similar()`**, que es capaz de mostrarnos otras palabras que tengan contextos similares.

```
import nltk

# Aqui va la lectura del archivo, por ahora la estoy omitiendo

tokens=nltk.word_tokenize(texto,"spanish")

texto_nltk=nltk.Text(tokens)
texto_nltk.similar("artículo")
```

NLTK

Palabras similares

- Para este tipo de funciones, entre más texto se tenga para hacer el análisis es mejor.
- Hasta ahora, hemos usado un texto corto para los ejemplos, probemos ahora con el conjunto de todos los que tenemos.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="DOF_P_IFT_291116_672_Acc.txt"


with open(carpeta_nombre+archivo_nombre,"r") as archivo:
        texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

texto_nltk=nltk.Text(tokens)
texto_nltk.similar("artículo")
```