Procesamiento de Lenguaje Natural o Minería de textos
Tema 2: Expresiones Regulares.

Objetivo: El participante identificará el uso de expresiones regulares para emparejar y extraer patrones de caracteres dentro de un texto, a partir del paquete **re** implementado en Python.

Temario:

1. ¿Qué son expresiones regulares?
2. Meta-caracteres
3. Construcción de expresiones regulares

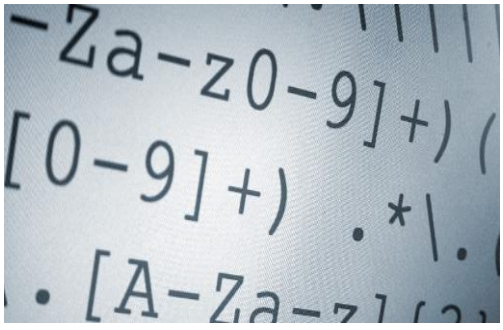
Lecturas:

Applied Text Analysis with Python / by Benjamin Bengfort, Rebecca Bilbro, Tony Ojeda : O'Reilly Media, Inc. [2018] 1 recurso en línea (xii, 334 páginas) : ilustraciones https://www.oreilly.com/library/view/applied-text-analysis/9781491963036/

Natural language processing recipes : unlocking text data with machine learning and deep learning using Python / Akshay Kulkarni, Adarsha Shivananda -- [Berkeley, California] : Apress, [2019].-- xxv, 234 páginas : ilustraciones

Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit 1st Edition / by Steven Bird, Ewan Klein, Edward Loper : O'Reilly Media, Inc. [2009] 1 recurso en línea (xi, 512 páginas) : ilustraciones https://itbook.store/books/9780596516499

Introducción



Uno de los problemas más comunes con que nos solemos encontrar al desarrollar cualquier programa informático, es el de procesamiento de texto. Esta tarea puede resultar bastante trivial para el cerebro humano, ya que nosotros podemos detectar con facilidad que es un número y que una letra, o cuales son palabras que cumplen con un determinado patrón y cuales no; pero estas mismas tareas no son tan fáciles para una computadora. Es por esto, que el procesamiento de texto siempre ha sido uno de los temas más relevantes en las ciencias de la computación.

Luego de varias décadas de investigación se logró desarrollar un poderoso y versátil lenguaje que cualquier computadora puede utilizar para reconocer patrones de texto; este lenguaje es lo que hoy en día se conoce con el nombre de ***expresiones regulares***; las operaciones de validación, búsqueda, extracción y sustitución de texto ahora son tareas mucho más sencillas para las computadoras gracias a las expresiones regulares.

Expresiones regulares[1] (*regular expression*, *regex* o *regexp*), son patrones que se utilizan para hacer coincidir combinaciones de caracteres en cadenas de texto.

- ✓ Pueden incluir patrones de coincidencia literal, de repetición, de composición, de ramificación, y otras sofisticadas reglas de reconocimiento de texto.
- ✓ Deberían formar parte del arsenal de cualquier buen programador ya que un gran número de problemas de procesamiento de texto pueden ser fácilmente resueltos con ellas; pueden ahorrarnos muchas líneas de código.
- ✓ Permiten filtrar textos para encontrar coincidencias, extraer partes específicas de un texto, comprobar la validez de fechas, documentos de identidad o contraseñas, se pueden utilizar para reemplazar texto con unas características concretas por otro, y muchos más usos.
- ✓ Son omnipresentes: se pueden utilizar en la mayoría de los lenguajes de programación como JavaScript, Ruby, Python, Java, etc., en algunas aplicaciones para la línea de comandos como *grep* y *find* y en algunos editores de texto como Atom o VSCode para realizar búsquedas avanzadas, entre muchos otros.
- ✓ Curva de aprendizaje difícil. Pueden ser difíciles de dominar y muy complejas de leer y entender si no se escriben con cuidado.

> *"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems."*
>
> *-Jamie Zawinski, 1997*

Hay muchas implementaciones distintas de regex[2], pero todas las podemos agrupar dentro de una de estas tres familias[3]:

- ✓ **BRE** (*Basic Regular Expression*): Las expresiones regulares básicas, estándar POSIX.
- ✓ **ERE** (*Extended Regular Expression*): Las expresiones regulares extendidas, también estándar POSIX. Son similares, pero no compatibles con las anteriores, y son la base del resto de tipos de expresiones regulares nacidas después. De hecho, lo común es que cualquier implementación de expresiones regulares que podamos encontrar, sea compatible con el estándar ERE, aunque disponga de extensiones adicionales.
- ✓ **PCRE** (*Perl Compatible Regular Expression*): Las expresiones regulares compatibles con Perl son la implementación de las regex para el lenguaje de programación perl. Añaden a las expresiones ERE algunas extensiones útiles y, aunque no son estándar POSIX, son de facto un estándar, puesto que muchas implementaciones posteriores las han tomado de base.

[1] Expresión regular: https://es.wikipedia.org/wiki/Expresi%C3%B3n_regular
[2] Véase regular-expressions.info.
[3] https://sio2sio2.github.io/doc-linux/02.conbas/10.texto/01.regex.html#id13

El módulo estándar de Python para expresiones regulares – re – solo admite expresiones regulares al estilo Perl. Hay un esfuerzo por escribir un nuevo módulo de expresiones regulares con mejor soporte de estilo POSIX en https://pypi.python.org/pypi/regex.

Componentes de las Expresiones Regulares

Es un patrón de texto que consiste en caracteres comunes (por ejemplo, letras ***a*** la ***z*** o números del ***0*** al ***9***) y caracteres especiales conocido como ***metacaracteres***. Este patrón describe las cadenas que coincidirían cuando se aplica a un texto.

- ✓ **Literales:** Cualquier carácter se encuentra a sí mismo, a menos que se trate de un *metacaracter* con significado especial. Una serie de caracteres encuentra esa misma serie en el texto de entrada, por ejemplo, `/ ser /` encontrará todas las apariciones de "ser" en el texto que procesamos (***ser,*** *o no* ***ser***).
- ✓ **Secuencias de escape:** La sintaxis de las expresiones regulares nos permite utilizar las secuencias de escape que ya conocemos de otros lenguajes de programación para esos casos especiales como ser finales de línea, tabs, barras diagonales, etc. Las principales secuencias de escape que podemos encontrar son:

Secuencia de escape	Significado
\n	Nueva línea (new line). El cursor pasa a la primera posición de la línea siguiente.
\t	Tabulador. El cursor pasa a la siguiente posición de tabulación.
\\	Barra diagonal inversa
\v	Tabulación vertical.
\ooo	Carácter ASCII en notación octal.
\xhh	Carácter ASCII en notación hexadecimal.
\xhhhh	Carácter Unicode en notación hexadecimal.

- ✓ **Clases de caracteres:** Se pueden especificar clases de caracteres encerrando una lista de caracteres entre corchetes **[]**, la que encontrará uno cualquiera de los caracteres de la lista. Si el primer símbolo después del **"["** es **"^"**, la clase encuentra cualquier carácter que no está en la lista.
  - ✓ Una expresión regular que coincida con las palabras "estimado" y "estimada": `/estimad[oa]/`
  - ✓ También es posible usar un rango de caracteres, usando el símbolo de guión (-) entre dos caracteres relacionados:
    - Para hacer coincidir cualquier letra minúscula, podemos usar `[a-z]`.
    - Para hacer coincidir cualquier dígito, podemos definir la clase de caracteres `[0-9]`.

Si queremos hacer coincidir cualquier carácter alfanumérico en minúscula o mayúscula, podemos usar `[0-9a-zA-Z]`. Esto

```
/[a-z]/ letras minusculas
/[A-Z]/ letras mayusculas
/[0-9]/ numeros
/[,'¿!¡;:\.\?]/ caracteres de puntuacion
                -la barra invertida hace que
                no se consideren como comando
                ni en punto ni el interrogante
/[A-Za-z]/      letras del alfabeto (del ingles claro ;)
/[A-Za-z0-9]/   todos los caracteres alfanumericos habituales
                -sin los de puntuacion, claro-
/[^a-z]/        El simbolo ^ es el de negación. Esto es decir
                TODO MENOS las letras minusculas.
/[^0-9]/        Todo menos los numeros.
```

puede escribirse alternativamente utilizando el mecanismo de unión: `[0-9[a-z[A-Z]]]`.

Ejemplos de rangos de caracteres:

- ✓ Los paréntesis son metacaracteres y tienen un significado especial.
  Expresión regular: `/ (esto está adentro) /`
  Texto: esto está afuera (esto está adentro)
  Resultado: esto está adentro
- ✓ Precediendo los metacaracteres con una barra diagonal inversa: `/\(esto está adentro\)/`
  Resultado: (esto está adentro)

- ✓ **Metacaracteres:** Los *metacaracteres* son caracteres especiales que son la esencia de las expresiones regulares. Son sumamente importantes y existen diferentes tipos:
  - Metacaracteres – delimitadores
  - Metacaracteres – clases predefinidas
  - Metacaracteres – iteradores
  - Metacaracteres – alternativas
  - Metacaracteres – subexpresiones
  - Metacaracteres – memorias (backreferences)

Metacaracteres – delimitadores: Esta clase de metacaracteres nos permite delimitar dónde queremos buscar los patrones de búsqueda. Ellos son:

Metacaracter	Descripción
^	inicio de línea.
$	fin de línea.
\A	inicio de texto.
\Z	fin de texto.
.	cualquier carácter en la línea excepto el salto de línea \n.
\b	encuentra límite de palabra.
\B	encuentra distinto a límite de palabra.

Metacaracteres - clases predefinidas: Estas son clases predefinidas que nos facilitan la utilización de las expresiones regulares. Ellos son:

Metacaracter	Descripción
\w	un carácter alfanumérico (incluye "_"); equivalente a `[a-zA-Z0-9_]`.
\W	un carácter no alfanumérico; equivalente a `[^a-zA-Z0-9_]`.
\d	un carácter numérico; equivalente a `[0-9]`.
\D	un carácter no numérico; equivalente a `[^0-9]`.
\s	cualquier espacio; equivalente a `[ \t\n\r\f\v]`.

\S	un no espacio; equivale a `[ ^\t\n\r\f\v]`.

Metacaracteres – iteradores: Cualquier elemento de una expresion regular puede ser seguido por otro tipo de metacaracteres, los *iteradores*. Usando estos metacaracteres se puede especificar el número de ocurrencias del carácter previo, de un metacaracter o de una subexpresión. Ellos son:

Metacaracter	Descripción
{n,m}	por lo menos n pero no más de m veces.
{n}	exactamente n veces.
{n,}	por lo menos n veces.
{,m}	no más de m veces
*	cero o más repeticiones, similar a `{0,}`.
+	una o más repeticiones, similar a `{1,}`.
?	Opcional, cero o una repetición, similar a `{0,1}`.

En estos metacaracteres, los dígitos entre llaves de la forma {n,m}, especifican el mínimo número de ocurrencias en n y el máximo en m.

Metacaracteres – alternativas: Se puede especificar una serie de alternativas usando el símbolo de tubería "**|**" para separarlas, entonces `do|re|mi` encontrará cualquier "do", "re", o "mi" en el texto de entrada. Las alternativas son evaluadas de izquierda a derecha, por lo tanto la primera alternativa que coincide plenamente con la expresión analizada es la que se selecciona. Por ejemplo: si se buscan `foo|foot` en "barefoot'', sólo la parte "foo" da resultado positivo, porque es la primera alternativa probada, y porque tiene éxito en la búsqueda de la cadena analizada.

Ejemplos:

- ✓ ¿La siguiente expresión con que coincide?: `/ Licencia: si|no /`
  - Licencia: si **o** Licencia: no
  - ✓ Licencia: si **o** no
- ✓ Usar paréntesis para definir grupos de alternancia: `/ Licencia: (si|no) /`
- ✓ `foo(bar|foo)` --> encuentra las cadenas 'foobar' o 'foofoo'.

Metacaracteres – subexpresiones: La construcción (...) también puede ser empleada para definir subexpresiones de expresiones regulares.

Ejemplos:

`(foobar){10}` --> encuentra cadenas que contienen 10 instancias de 'foobar'
`foob([0-9]|a+)r` --> encuentra 'foob0r', 'foob1r' , 'foobar', 'foobaar', 'foobaar' etc.

Metacaracteres - memorias (backreferences): Los metacaracteres \1 a \9 son interpretados como memorias. \ encuentra la subexpresión previamente encontrada #.

Ejemplos:

`(.)\1+` --> encuentra 'aaaa' y 'cc'.

`(.+)\1+` --> también encuentra 'abab' y '123123'

`(['"]?)(\d+)\1` --> encuentra '"13" (entre comillas dobles), o '4' (entre comillas simples) o 77 (sin comillas) etc.

Expresiones regulares con Python

- Implementadas con el módulo **re:** https://docs.python.org/3/library/re.html

```
# importando el modulo de regex de python
import re
```

- En Python[4][5], hay dos objetos diferentes que tratan con Regex:
  - ✓ **RegexObject**: también se conoce como Pattern Object. Representa una expresión regular compilada. Proporciona todas las operaciones que se pueden realizar, como la coincidencia y la búsqueda de todas las subcadenas que coinciden con una expresión regular determinada.
  - ✓ **MatchObject**: representa el patrón de coincidencia.

Ejercicio2(es)-Expresiones regulares.ipynb

Tarea:

Se tiene un archivo (*dates.txt*) donde cada línea de este corresponde a una nota médica y cada nota tiene una fecha que debe extraerse, pero cada fecha está codificada en uno de muchos formatos. Por ejemplo, se muestra a continuación una lista de algunas de las variantes que se puede encontrar en este conjunto de datos:

- 04/20/2009; 04/20/09; 4/20/09; 4/3/09
- Mar-20-2009; Mar 20, 2009; March 20, 2009; Mar. 20, 2009; Mar 20 2009;
- 20 Mar 2009; 20 March 2009; 20 Mar. 2009; 20 March, 2009
- Mar 20th, 2009; Mar 21st, 2009; Mar 22nd, 2009
- Feb 2009; Sep 2009; Oct 2010
- 6/2008; 12/2009
- 2009; 2010

[4] Pueden ver ejemplos de expresiones regulares en Python en: https://relopezbriega.github.io/blog/2015/07/19/expresiones-regulares-con-python/#Buscando-coincidencias

[5] Otros ejemplos de expresiones regulares: https://cs.lmu.edu/~ray/notes/regex/

La actividad consiste en:

a) Identificar correctamente todas las diferentes variantes de fecha codificadas en este conjunto de datos, normalizar y ordenar adecuadamente las fechas.
b) Una vez que haya extraído estos patrones de fecha del texto, el siguiente paso es clasificarlos en orden cronológico ascendente de acuerdo con las siguientes reglas:

- ✓ Todas las fechas están en formato xx/xx/xx son mm/dd/aa
- ✓ Todas las fechas en las que el año está codificado en solo dos dígitos corresponden a años posteriores a la década de 1900 (p. Ej., 1/5/89 es el 5 de enero de 1989).
- ✓ Si falta el día (p. Ej., 9/2009), suponga que es el primer día del mes (p. Ej., septiembre, 1 de 2009).
- ✓ Si falta el mes (por ejemplo, 2010), suponga que es el primero de enero de ese año (p. Ej., enero, 1 de 2010).
- ✓ Tenga cuidado con los posibles errores tipográficos, ya que este es un conjunto de datos derivados de la vida real.

Esta función debería devolver una lista de longitud 500.

Conclusiones

Las expresiones regulares pueden parecer indescifrables (¡algunas lo son!) pero son muy útiles cuando se necesita encontrar patrones en un texto. Sin embargo, escribir una expresión regular desde cero no es trivial. Una buena idea es buscar primero algunas opciones en Internet y seleccionar una que sea simple pero que funcione bien en la mayoría de los casos.

La forma más rápida para aprender a hacer expresiones regulares es mediante ejemplos, ir probando combinaciones y comprobar en tiempo real el resultado. No hay una solución única en cada caso. Un método de resolver las expresiones complejas es mediante la técnica de divide y vencerás, extraer cada subfuncionalidad y colocarla por separado, pero se puede optar por soluciones más compactas.