



Maestría en Inteligencia Artificial Aplicada (MNA)

Matriz Dispersa : Sparse Matrix

Procesamiento de Lenguaje Natural (NLP)


Luis Eduardo Falcón Morales

Sparse matrix : Matriz dispersa



La representación matricial de documentos de texto a través de la “Document Term Matrix”, DTM, son generalmente matrices muy grandes. Sin embargo, hemos visto que gran cantidad de los valores contenidos en dichas matrices, son valores de cero.

Esta semana estudiaremos diversas representaciones matriciales que ayuden a optimizar recursos computacionales, no solamente para representar dicha matriz de manera más económica, sino también para llevar a cabo las operaciones matriciales requeridas.



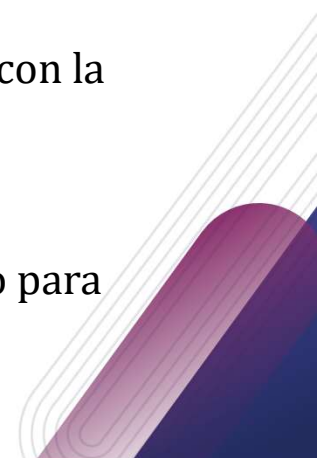
Una **matriz dispersa** (*sparse matrix*) es aquella en la que la mayoría de sus entradas son cero.

Una **matriz densa** (*dense matrix*) es aquella en la que la mayoría de sus entradas son diferentes de cero.

Al cociente del total de entradas cero de una matriz, dividida por el total de entradas de dicha matriz se le llama **dispersión** de la matriz (*sparsity*).


El problema con las matrices dispersas es que pueden llegar a ser matrices muy grandes con la mayoría de sus entradas iguales a cero, es decir, haciendo uso de recursos de espacio de memoria y tiempo para guardar o procesar solamente valores u operaciones de ceros.

Así, la importancia de estas matrices radicará en poder tomar ventaja de sus entradas cero para optimizar su uso y recursos en su manipulación y operaciones.





Existen muchos tipos de problemas de donde pueden surgir las matrices dispersas:


- Catálogo de compras de productos por parte de clientes.
 - Listado de las recomendaciones dada por usuarios a películas/productos.
 - Listado de las canciones escuchadas por los usuarios de Spotify.
 - Conteo de frecuencias de las palabras que aparecen en un documento/diccionario.
 - Páginas web conectadas entre sí mediante algún hipervínculo.
 - ...
- 



Los problemas anteriores nos llevan a obtener matrices dispersas y de ahí a llevar a cabo operaciones matriciales con ellas.

En particular, se deseará resolver sistemas de ecuaciones de la forma $AX = b$, donde la matriz A es dispersa.

De acuerdo a la forma en que se multiplican las matrices, existirían muchas operaciones de multiplicaciones de ceros y suma de ceros, los cuales obviamente no aportan nada a la solución del sistema.



En ocasiones se considera una matriz como dispersa si su cantidad de elementos no cero es aproximadamente igual a su cantidad de renglones o columnas.

Así, por ejemplo, en una matriz A de tamaño 1000×1000 tendría en este caso un valor de dispersión es del 99.9%

$$A_{1000 \times 1000}$$

Matriz dispersa.

Supongamos que su valor de dispersión es del 99.9%.
Y que el tipo de dato no cero es de 8 bytes.

Memoria requerida en este caso: 8,000 bytes = 0.008 MB

$$B_{1000 \times 1000}$$

Matriz densa.

Supongamos que el tipo de dato no cero es de 8 bytes.

Memoria en este caso:
8'000,000 bytes = 8 MB



Existen diferentes formatos para guardar la información de una matriz dispersa.

Veamos algunos de los principales utilizados en el área de aprendizaje automático:

- CSR: Compressed Sparse Row : Comprimido por filas
- CSC: Compressed Sparse Column : Comprimido por columnas
- COO: Coordinate format : Formato de Coordenadas – triplet format ijk
- Existen otros tipos de matrices como matriz por bloques, por bandas, diagonal, densa, entre otras.

Existen diversas librerías que permiten obtener algunas de estas matrices. En particular SciPy es de las más completas:

<https://docs.scipy.org/doc/scipy/reference/sparse.html>



Sparse Matrix → Simple-Triplet-Matrix : CSR

```
import numpy as np
from scipy import sparse
```

```
A = np.array([[1, 0, 0, 3, 0, 0], [0, 0, 7, 0, 0, 0],
              [0, -1, 0, 0, 0, 9], [2, 0, 0, 8, 0, 0]], dtype=np.int64)
```

```
A
array([[ 1,  0,  0,  3,  0,  0],
       [ 0,  0,  7,  0,  0,  0],
       [ 0, -1,  0,  0,  0,  9],
       [ 2,  0,  0,  8,  0,  0]])
```

Sparsity=17/24 ≈ 0.71

También podemos obtener la posición (renglón, columna) de cada elemento no-cero de la matriz, mediante la siguiente instrucción:

```
B = sparse.csr_matrix(A)
print(B)
```

(0, 0)	1
(0, 3)	3
(1, 2)	7
(2, 1)	-1
(2, 5)	9
(3, 0)	2
(3, 3)	8

CSR : Compressed Sparse Row

Formato para matrices CSR:

Son los índices del primer elemento no cero de cada renglón.

índice ptr: [0 2 3 5 7]
columna: [0 3 2 1 5 0 3]
valor: [1 3 7 -1 9 2 8]

```
print(B.indptr)
print(B.indices)
print(B.data)
```

Observa que se usa una dimensión igual a $rows + 1$

En general, podemos decir que en una CSR sus entradas se van describiendo por renglón.

Ejemplo: Obtengamos la representación CSR de la siguiente matriz A:

$$A = \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

Para ello requerimos obtener los valores de los siguientes 3 vectores:

índice ptr : $[\dots]$

columna: $[\dots]$

valor: $[\dots]$

En CSR la inspección se realiza por renglón.

Iniciamos buscando en el primer renglón de la matriz A el valor del primer valor no cero, en este ejemplo es el 4 como se indica a continuación:


$$A = \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

índice ptr : $[\dots]$

columna: $[\dots]$

valor: $[\dots]$

$$A = \begin{bmatrix} 0 & \text{idx}(0) \circled{4} & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

Como es el primer valor no cero encontrado en A , su índice asociado será 0:

índice ptr: [0]
columna: [...]
valor: [...]

Y el cual se encuentra en la segunda columna con índice 1 y de valor 4:

índice ptr: [0]
columna: [1]
valor: [4]

NOTA: Recordemos que las columnas de A se indexan a partir de 0.

Ejemplo

Continuamos buscando, por renglón, el segundo dato no cero y al cual le asociaremos el índice 1:

$$A = \begin{bmatrix} 0 & 4 & 0 & \text{idx}(1) \circled{6} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

Como encontramos el segundo dato no cero en el mismo renglón que el dato anterior, no registramos su índice (1) y solo registramos su columna (3) y valor (6), es decir, ahora tendremos:

índice ptr: [0]
columna: [1 3]
valor: [4 6]

Continuamos buscando el siguiente valor no cero por renglón. De encontrarlo será el tercer valor y por lo tanto tendría índice 2. Observamos que en el primer renglón ya no hay más valores no cero, por lo que continuamos la búsqueda en el segundo renglón. Como cambiamos de renglón, esto implica que si encontramos este tercer dato no cero, deberemos registrar su índice 2 en el primer vector de índices. Y efectivamente, encontramos dicho valor no cero en este renglón. Registramos entonces su índice (2), columna (2) y valor (1):

$$\xrightarrow{\quad} A = \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

índice ptr: [0 2]

columna: [1 3 2]

valor: [4 6 1]

Ejemplo

Continuamos la búsqueda por renglón del siguiente valor no cero: sería el cuarto y tendría índice 3. Lo encontramos en el mismo renglón: 9. Como está en el mismo renglón, no registramos su índice (3), pero sí su columna (5) y valor (9):

$$\xrightarrow{\quad} A = \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

índice ptr: [0 2]

columna: [1 3 2 5]

valor: [4 6 1 9]

Continuemos con la búsqueda del siguiente valor no cero: sería el quinto y por lo tanto con índice 4.

Ya terminamos con el segundo renglón y la nueva búsqueda se lleva a cabo ahora en el tercer renglón. Esto implicará de nuevo que al encontrar el siguiente valor no cero, deberemos registrar su índice (4 en este caso).

$$\xrightarrow{A} \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

Sin embargo, vemos que en el tercer renglón no existen valores no cero. En estos casos lo único que registramos es el valor del índice que estamos buscando (4). Esto ayudará a identificar que hubo un renglón de ceros:

índice ptr: [0 2 4]

columna: [1 3 2 5]

valor: [4 6 1 9]

Continuamos nuestra búsqueda entonces en el último renglón de nuestra matriz A:

$$\xrightarrow{A} \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

idx(4)

Como es el primer valor no cero encontrado en ese renglón, registramos su índice (4), su columna (0) y valor (2):

índice ptr: [0 2 4 4]

columna: [1 3 2 5 0]

valor: [4 6 1 9 2]

Continuamos nuestra búsqueda de otro valor no cero en el último renglón y encontramos el 8:

$$A = \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

idx(5)

Como no es el primer valor no cero encontrado en dicho renglón, solamente registramos su columna (3) y valor (8):

índice ptr: [0 2 4 4]
columna: [1 3 2 5 0 3]
valor: [4 6 1 9 2 8]

Ejemplo

Continuamos nuestra búsqueda de otro valor no cero en el último renglón. De encontrarlo sería el séptimo dato no cero y por lo tanto con índice 6. Sin embargo vemos que ya no hay más valores no cero, además de que ahí termina la matriz:

$$A = \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

Cuando terminamos la búsqueda en la matriz, registramos solamente el último índice del dato que estábamos buscando, en este caso el 6:

índice ptr: [0 2 4 4 6]
columna: [1 3 2 5 0 3]
valor: [4 6 1 9 2 8]

Esta es finalmente la representación CSR de la matriz A dada.

$$A = \begin{bmatrix} 0 & 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

Representación dispersa CSR de la matriz A:

índice ptr: [0 2 4 4 6]

columna: [1 3 2 5 0 3]

valor: [4 6 1 9 2 8]

idx(0)

idx(1)

idx(2)

idx(3)

idx(4)

idx(5)

$$\text{Sparsity} = 18/24 \approx 0.75$$

En resumen:

Se incluye el valor del índice en el vector de “salto de renglón” cada vez que se cumple cualquiera de las siguientes condiciones:

- si el número encontrado es la primer entrada no-cero del renglón (leído de izquierda a derecha).
- si el renglón que se inspecciona no tiene entradas no-cero, se anota el índice que se está buscando en ese momento.
- cuando se termina de inspeccionar la matriz se anota el valor del índice que se estaba buscando en ese momento. Dicho valor será además igual al total de entradas no cero que tiene la matriz.

Ejemplo

Encontrar la matriz de dimensión 5×6 cuya representación simple-triplet-matrix en su formato CSR (compressed sparse row) está dada como sigue:

```
print(B.indptr)
print(B.indices)
print(B.data)
```

```
[0 2 2 2 4 5]
[0 3 1 4 3]
[ 5  9  7 -2  4]
```

Solución:

```
array([[ 5,  0,  0,  9,  0,  0],
       [ 0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0],
       [ 0,  7,  0,  0, -2,  0],
       [ 0,  0,  0,  4,  0,  0]])
```

Sparsity= $5/30 \approx 0.83$

Sparse Matrix → Simple-Triplet-Matrix : CSC

```
import numpy as np
from scipy import sparse
```

```
A = np.array([[1, 0, 0, 3, 0, 0], [0, 0, 7, 0, 0, 0],
              [0, -1, 0, 0, 0, 9], [2, 0, 0, 8, 0, 0]], dtype=np.int64)
```

```
A
array([[ 1,  0,  0,  3,  0,  0],
       [ 0,  0,  7,  0,  0,  0],
       [ 0, -1,  0,  0,  0,  9],
       [ 2,  0,  0,  8,  0,  0]])
```

Sparsity= $17/24 \approx 0.71$

También podemos obtener la posición de cada elemento no-cero de la matriz:

```
C = sparse.csc_matrix(A)
print(C)
```

(0, 0)	1
(3, 0)	2
(2, 1)	-1
(1, 2)	7
(0, 3)	3
(3, 3)	8
(2, 5)	9

CSC : Compressed Sparse Column

Formato para matrices CSC:

```
print(C.indptr)
print(C.indices)
print(C.data)
```

Observa que su usa una dimensión igual a $cols + 1$

índice ptr: [0 2 3 4 6 6 7]
renglón: [0 3 2 1 0 3 2]
valor: [1 2 -1 7 3 8 9]

En general, podemos decir que en una CSC sus entradas se van describiendo por columnas.

Sparse Matrix → Simple-Triplet-Matrix : COO

```
import numpy as np
from scipy import sparse
```

```
A = np.array([[1, 0, 0, 3, 0, 0], [0, 0, 7, 0, 0, 0],
              [0, -1, 0, 0, 0, 9], [2, 0, 0, 8, 0, 0]], dtype=np.int64)
```

```
A
array([[ 1,  0,  0,  3,  0,  0],
       [ 0,  0,  7,  0,  0,  0],
       [ 0, -1,  0,  0,  0,  9],
       [ 2,  0,  0,  8,  0,  0]])
```

Sparsity=17/24 ≈ 0.71

COO : Coordinate format

Formato para
matrices COO:

```
print(D.row)
print(D.col)
print(D.data)
```


```
renglón: [0 0 1 2 2 3 3]
columna: [0 3 2 1 5 0 3]
valor:   [ 1  3  7 -1  9  2  8]
```

SciPy nos proporciona esta
salida en una matriz COO:

```
D = sparse.coo_matrix(A)
print(D)
```

(0, 0)	1
(0, 3)	3
(1, 2)	7
(2, 1)	-1
(2, 5)	9
(3, 0)	2
(3, 3)	8

En general, podemos decir que
en una COO sus entradas se van
describiendo por su posición
renglón-columna en la matriz.



En general, la solución de sistemas de ecuaciones o cálculo de la inversa de una matriz se llevan a cabo mediante métodos iterativos, más que mediante métodos directos.

Es por ello que se requiere el uso de librerías confiables y bien documentadas para su uso.





N-gramas



Un *n*-grama es una secuencia de *n* palabras/tokens/strings.

Existen diversos modelos estadísticos para calcular la probabilidad de obtener un n-grama.

unigramas

('I',)
('am',)
('Thor', ,)
('son',)
('of',)
('Odin.',)
('I',)
('am',)
('the',)
('older',)
('son.',)
('Odin',)
('of',)
('Asgard?',)

bigramas

('I', 'am')
('am', 'Thor',)
('Thor', , 'son')
('son', 'of')
('of', 'Odin.')
('Odin.', 'I')
('I', 'am')
('am', 'the')
('the', 'older')
('older', 'son.')
('son.', 'Odin')
('Odin', 'of')
('of', 'Asgard?')

trigramas

('I', 'am', 'Thor',)
('am', 'Thor', , 'son')
('Thor', , 'son', 'of')
('son', 'of', 'Odin.')
('of', 'Odin.', 'I')
('Odin.', 'I', 'am')
('I', 'am', 'the')
('am', 'the', 'older')
('the', 'older', 'son.')
('older', 'son.', 'Odin')
('son.', 'Odin', 'of')
('Odin', 'of', 'Asgard?')

etc...

Probabilidad Condicional

$$P(w_{n+1} \mid \overbrace{w_1 w_2 \cdots w_n}^{n_grama})$$

$w_1 w_2 \cdots w_n w_{n+1}$

Dada una secuencia de n palabras, es decir un n -grama, se desea obtener la probabilidad que la siguiente palabra sea w_{n+1} .

Observamos que mientras mayor sea el valor de n en la secuencia de palabras de un n -grama, mayor será el tamaño de nuestro vocabulario/diccionario de tokens, aumentando a su vez el costo computacional durante el procesamiento y análisis del corpus.

Por la regla de Bayes:

$$P(w_{n+1} | w_1 w_2 \cdots w_n) = \frac{\overbrace{P(w_1 w_2 \cdots w_n w_{n+1})}^{(n+1)\text{-grama}}}{\underbrace{P(w_1 w_2 \cdots w_n)}_{n\text{-grama}}}$$

De manera práctica la manera de obtener la probabilidad de que una palabra continúe después de por ejemplo un trigramma dado, es contabilizando el total de veces que aparece dicha secuencia (trigramma y palabra) en el corpus de estudio y dividirla por el total de todos los trigrammas del corpus.

Este método para calcular dichas probabilidades se basa en el llamado método de **Estimación de Máxima Verosimilitud (Maximum Likelihood Estimation)**.

NOTA: Causalidad: probabilísticamente el orden de la probabilidad conjunta es indiferente: $P(w_1, w_2) = P(w_2, w_1)$. Sin embargo en los enunciados sí es importante y debiera considerarse durante el análisis.

Propiedad de Markov

La propiedad o criterio de Markov nos dice que la probabilidad de un evento futuro depende únicamente del evento presente inmediato. En ocasiones se dice que estos procesos no tienen memoria.

$$P(w_{n+1} | w_1 w_2 \cdots w_n) \approx P(w_{n+1} | w_n)$$

Suele llamarse el modelo bigrama cuando se aplica la propiedad de Markov.

De manera análoga se puede extender la propiedad de Markov a n palabras previas.

Con base a la propiedad de Markov y con el fin de minimizar los costos computacionales, suele aplicarse el siguiente criterio mediante el uso de n-gramas para el cálculo de las probabilidades de una frase.

Dada una frase W , su probabilidad la podemos considerar como la probabilidad conjunta de obtener la secuencia de tokens w_1, w_2, \dots, w_n que la forman:

$$P(\text{"I am Thor son of Odin"}) = P(\text{"I"}, \text{"am"}, \text{"Thor"}, \text{"son"}, \text{"of"}, \text{"Odin"})$$

Y ahora podemos aplicar la información que se tiene sobre los n-gramas en nuestro vocabulario:

$$\begin{array}{ccccccc} & \text{bigrama} & & \text{trigrama} & & & \text{(n+1)-grama} \\ & \underbrace{\hspace{1.5cm}} & & \underbrace{\hspace{2.5cm}} & & & \underbrace{\hspace{4.5cm}} \\ = P(\text{"I"}) & P(\text{"am"} | \text{"I"}) & P(\text{"Thor"} | \text{"I"}, \text{"am"}) & \cdots & P(\text{"Odin"} | \text{"I"}, \text{"am"}, \text{"Thor"}, \text{"son"}, \text{"of"}) \\ & \underbrace{\hspace{1.5cm}} & & \underbrace{\hspace{1.5cm}} & & \underbrace{\hspace{1.5cm}} & \underbrace{\hspace{2.5cm}} \\ & \text{unigrama} & & \text{bigrama} & & w_{n+1} & \text{n_grama} \end{array}$$

NOTA: Observa que nuestro vocabulario debiera entonces contener unigramas, bigramas, trigramas, hasta (n+1)-gramas.

Si por ejemplo nuestro vocabulario/diccionario contiene solamente unigramas y bigramas, se puede aplicar de manera recursiva la propiedad de Markov para obtener la probabilidad de una frase, n-grama.

Para el ejemplo anterior podríamos aproximar la probabilidad del 6-grama "I am Thor son of Odin". Así, podemos aproximar el valor de este 6-grama mediante unigramas y bigramas como se muestra a continuación:

$$P(\text{"I am Thor son of Odin"}) \approx P(\text{"I"})P(\text{"am"} | \text{"I"})P(\text{"Thor"} | \text{"am"})P(\text{"son"} | \text{"Thor"})P(\text{"of"} | \text{"son"})P(\text{"Odin"} | \text{"of"})$$

Este tipo de aproximaciones nos permite aprovechar mejor los recursos computacionales para el cálculo de los n-gramas.

Cuestiones a considerar durante el análisis de un corpus:

- El vocabulario, formado por tokens y n-gramas, depende fuertemente del tipo de corpus con el cual fue generado. Es decir, depende de las temáticas del corpus a analizar.
- Aún cuando se haya formado un gran vocabulario, siempre existe la posibilidad de aparecer nuevos tokens o n-gramas: UNKNOWN.
- Dado un nuevo corpus y un vocabulario previamente construido, existirán gran cantidad de tokens que no aparezcan en los nuevos documentos y cuyas frecuencias sean cero. Esto generará problemas para la obtención de probabilidades condicionales y conjuntas, ya que muchas de ellas resultarán iguales a cero. En estos casos se aplica un ajuste a todos los valores y al cual se le conoce con el nombre de **estimador Laplaciano (Laplace smoothing)**.
- Aunque el estimador Laplaciano ayuda a evitar frecuencias cero, en los modelos con diversos documentos existirán una gran cantidad de ajustes que realizar, más aún si se consideran n-gramas, por lo que el modelo ajustado se empieza a alejar del comportamiento o distribución de los enunciados del corpus en análisis. Será conveniente buscar entonces mejores modelos alternativos, entre los que estarán los basados en aprendizaje profundo. Tema que estudiaremos más adelante.

Así, mediante conceptos de Probabilidad y Estadística podemos construir modelos del Lenguaje basados en corpus:

$$P(w_1 w_2 \cdots w_n w_{n+1}) = P(w_{n+1} | w_1 w_2 \cdots w_n)$$

Podremos o no considerar simplificar conceptos mediante la propiedad de Markov y los n-gramas, sin embargo la longitud de los n-gramas que se desee considerar, dependerá de los recursos computacionales que tengamos, así como de contar con una base de datos suficientemente basta para llevar a cabo el entrenamiento.

Por ejemplo, si deseamos calcular la probabilidad de un 5-grama, $w_1 w_2 w_3 w_4 w_5$, dependiendo de la longitud de los n-gramas a utilizar los resultados de las aproximaciones serían los siguientes:

Usando unigramas: $P(w_1 w_2 w_3 w_4 w_5) = P(w_1)P(w_2)P(w_3)P(w_4)P(w_5)$

Usando bigramas: $P(w_1 w_2 w_3 w_4 w_5) = P(w_1)P(w_2|w_1)P(w_3|w_2)P(w_4|w_3)P(w_5|w_4)$

Usando trigramas: $P(w_1 w_2 w_3 w_4 w_5) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_4|w_2, w_3)P(w_5|w_3, w_4)$

⋮

⋮

⋮

⋮



D.R.© Tecnológico de Monterrey, México, 2022.
Prohibida la reproducción total o parcial
de esta obra sin expresa autorización del
Tecnológico de Monterrey.