PROYECTO SAR-ALTM

Cardona Lorenzo, Víctor
Gavilán Gil, Marc
Martínez Bernia, Javier
Murcia Serrano, Andrea

Contenido

Introducción

En esta memoria nos centraremos en analizar los resultados de la segunda parte del proyecto conjunto SAR-ALT. En esta parte se ha añadido al motor de recuperación de información que se desarrolló en la primera, la capacidad de hacer búsquedas aproximadas en nuestro índice. Para lograr este objetivo antes deberemos elegir un algoritmo eficiente para hacer la búsqueda aproximada de una cadena respecto de todas las cadenas del diccionario de términos.

Distancia de Levenshtein

La distancia de Levenshtein o distancia de edición entre dos cadenas α y β, no necesariamente de la misma longitud, se define como el número mínimo de operaciones básicas que son necesaria para transformar la cadena α en la cadena β. Las operaciones permitidas son:

- el borrado de un carácter de la cadena α
- la inserción de un carácter de la cadena β
- la substitución de un carácter de la cadena α por otro de la cadena β

Asumiendo que todas las operaciones de edición tienen la misma penalización, 1, la distancia de Leveshtein se puede calcular por programación dinámica utilizando la ecuación recursiva presentada a continuación:

$$d_{\alpha,\beta}(i,j) = \begin{cases} 0 & : \quad i = j = 0 \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) & : \quad i > 0 \wedge j = 0 \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j) & : \quad i = 0 \wedge j > 0 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + sust(\alpha_i,\beta_j), \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j), \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) \end{pmatrix} & : \quad i > 0 \wedge j > 0 \end{cases}$$

Distancia de Damerau – Levenshtein

La distancia de Damerau-Levenshtein es una extensión de la distancia de Levenshtein en la cual, además de la substitución, borrado e inserción, permitimos también la transposición de dos caracteres consecutivos.

$$d_{\alpha,\beta}(i,j) = \begin{cases} 0 & : \quad i = j = 0 \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) & : \quad i > 0 \wedge j = 0 \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j) & : \quad i = 0 \wedge j > 0 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + sust(\alpha_i,\beta_j), \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j), \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) \end{pmatrix} & : \quad i = 1 \vee j = 1 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + sust(\alpha_i,\beta_j), \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j), \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i), \\ d_{\alpha,\beta}(i-2,j-2) + trans(\alpha_{i-1:i}) : \alpha_i = \beta_{j-1} \wedge \alpha_{i-1} = \beta_j \end{pmatrix} & : \quad i > 1 \wedge j > 1 \end{cases}$$

Variantes e implementación

Los algoritmos presentados en los apartados anteriores presentan diversas variantes de implementación. A continuación, daremos una breve explicación de las mismas.

Contra cadena

En esta versión calculamos la distancia de edición de una palabra seleccionada contra el conjunto de todas las palabras del texto proporcionado. Utilizando la versión de Levenshtein como ejemplo, podemos ver como en el código que se presenta vamos comparando, recorriendo letra a letra, la palabra dada con todas sus cercanas. En cada punto, vemos qué costaría hacer cada una de las operaciones (sustitución, inserción y borrado) para quedarnos con la de menor peso. Los resultados intermedios se van guardando en una matriz, *D*, donde el mínimo se halla en la esquina superior derecha, que corresponde con haber analizado las dos cadenas completamente.

```
def levenshtein_distance(x, y):
    D = np.empty((len(x)+1, len(y)+1), dtype=np.int8)
    D[0, 0] = 0
    for i in range(1, len(x)+1):
        D[i, 0] = D[i-1, 0] + 1
    for j in range(1, len(y)+1):
        D[0, j] = D[0, j-1] + 1
        for i in range(1, len(x)+1):
            D[i, j] = min(D[i-1, j]+1, D[i, j-1]+1,
                          D[i-1, j-1]+(x[i-1] != y[j-1]))
    return D[len(x), len(y)]
```

De esta manera, si la distancia de edición calculada es menor que la que se ha proporcionado como parámetro, devolvemos la palabra comparada como uno de los resultados.

```
def palabrasCercanas(diccionario, word, distance):
    cercanas = []
    for palabra in diccionario:
        dist = levenshtein_distance(word, palabra)
        if dist <= distance:
            cercanas.append((palabra, dist))
    return cercanas
```

La versión de Damerau-Levenshtein ocuparía la misma implementación agregando la opción de poder permutar.

```python
def levenshtein_distance(x, y):
    D = np.empty((len(x)+1, len(y)+1), dtype=np.int8)
    D[0, 0] = 0
    for i in range(1, len(x)+1):
        D[i, 0] = D[i-1, 0] + 1
    for j in range(1, len(y)+1):
        D[0, j] = D[0, j-1] + 1
        for i in range(1, len(x)+1):
            if i > 1 and j > 1:
                cond_damerau = (x[i-1] == y[j-2] and x[i-2] == y[j-1])
                D[i, j] = min(D[i-1, j]+1, D[i, j-1]+1, D[i-1, j-1]+(x[i-1] != y[j-1]),
                              ((D[i-2, j-2] + 1)*cond_damerau)+(sys.maxsize*(1-cond_damerau)))
            else:
                D[i, j] = min(D[i-1, j]+1, D[i, j-1]+1,
                              D[i-1, j-1]+(x[i-1] != y[j-1]))

    return D[len(x), len(y)]
```

Trie

Un trie es una estructura de tipo árbol utilizada para almacenar un diccionario de términos en sistemas de recuperación de información.

En nuestro caso, esta estructura de datos se ha implementado como un diccionario cuyas claves son los números de nodo; y cuyos valores son una tripleta que contiene:

- Una referencia al nodo padre
- Un campo nulo, si no hemos llegado a un nodo final, o bien la palabra completa que se ha formado recorriendo el trie
- Una lista de los hijos del nodo actual, representados por sus letras

Una vista más detallada del código puede verse en el archivo *generarTrie.py.*

Empleando de nuevo el algoritmo de Levenshtein como base para la explicación del código, vemos como creamos una matriz dinámica de tantas filas como letras tenga nuestra palabra y tantas columnas como nodos haya en el trie. Una vez creada, comenzamos inicializando la primera columna con números enteros ascendentes desde el 1 hasta el tamaño de la palabra y la primera fila con la profundidad de cada nodo.

Para rellenar cada celda de la matriz calculamos el mínimo entre el coste de borrado, el cual haremos mirando la celda de la letra anterior; el coste de inserción, mirando la celda del nodo padre; y el coste de sustitución, para el que miramos la celda del nodo padre de la letra anterior a la actual.

Una vez completada la matriz, en la última fila encontraremos los resultados de las distancias de edición. Las palabras que serán seleccionadas como resultado son aquellas que cumplan que el nodo sea terminal y que su distancia sea menor o igual a la deseada.

```python
def palabrasCercanas(trie, palabra, distancia):
    M = np.empty(dtype=np.int8, shape=(len(palabra)+1, len(trie)))
    for i in range(len(palabra)+1):  # La primera columna
        M[i, 0] = i
    for j in range(len(trie)):  # La primera fila (profundidades de los nodos del trie)
        profun = 0
        padre = trie[j][0]
        while padre != None:
            padre = trie[padre][0]
            profun += 1
        M[0, j] = profun

    for i in range(1, len(palabra)+1):
        for j in range(1, len(trie)):
            costeBorr = M[i-1, j]+1
            padre = trie[j][0]
            costeIns = M[i, padre] + 1
            letra = palabra[i-1]
            costeSus = M[i-1, padre] + \
                (trie[padre][2].get(palabra[i-1], -1) != j)
            M[i, j] = min(costeBorr, costeIns, costeSus)

    # Matriz llena, sacar las palabras cercanas
    palabras_cercanas = []
    for j in range(1, len(trie)):
        if (trie[j][1] != None and M[len(palabra), j] <= distancia):
            palabras_cercanas.append(trie[j][1])

    return palabras_cercanas
```

Nuevamente, la implementación del algoritmo Damerau con Trie es muy parecida: solo tenemos que añadir el cálculo del coste de permutación, para el cual utilizaremos la celda del abuelo y lo compararemos con la del padre.

```python
costeDam = sys.maxsize
if i > 1 :
    abuelo = trie[padre][0]
    if abuelo != None:
        costeDam = M[i-1,padre] + (not(trie[padre][2].get(palabra[i-2], -1) == j and
        trie[abuelo][2].get(palabra[i-1], -1) == padre))

M[i,j] = min(costeBorr,costeIns,costeSus,costeDam)
```

Ramificación (branch)

Por último, comentaremos la implementación de Levenshtein y Damerau con ramificación. Para ello, hemos utilizado una pila de estados donde cada estado se representa con una terna (i,n,d):

- i, la cantidad de símbolos analizados de la cadena hasta el momento
- n, el número de nodo en el que nos encontramos
- y d, la mejor distancia calculada hasta el momento

Partiendo del estado inicial (0,0,0) ramificamos, añadiendo u eliminando estados de la pila.

Borraremos un estado cuando la distancia calculada sea más grande que la deseada o cuando, habiendo analizado toda la cadena, demos con un nodo no terminal y una distancia más grande de la deseada.

Por otro lado, añadiremos estados a la pila cuando realicemos una inserción, un borrado o una sustitución. Para la inserción crearemos un estado para cada uno de los hijos del nodo actual y, en ellos, no aumentaremos la cantidad de símbolos analizados, pero sí incrementaremos en uno la distancia. En cuanto a la sustitución, también crearemos un estado para cada hijo, pero esta

vez incrementando el número de símbolos analizados. La distancia podrá incrementarse o no dependiendo de la igualdad de las letras que estén siendo analizadas. Por último, el borrado solo crearemos un estado en el que incrementaremos tanto la distancia como los símbolos analizados.

Cuando llegamos a un estado con un nodo terminal, comprobamos que la distancia sea menor o igual que la deseada y, si se cumple, añadimos la palabra al resultado.

```
def palabrasCercanas(trie, palabra, distancia):
    pila = deque([(0, 0, 0)])
    cercanos = set()
    while len(pila) > 0:
        nodo_ppal = pila.popleft()
        analizado, nodo, coste = nodo_ppal
        if coste > distancia:
            continue

        # Coincidencia encontrada
        if(trie[nodo][1] != None and coste <= distancia and (len(palabra) == analizado)):
            cercanos.add(trie[nodo][1])

        if analizado < len(palabra):  # Hay al menos 1 carácter borrable
            pila.appendleft((analizado + 1, nodo, coste + 1))  # Borrado

        for letra_hijo in trie[nodo][2]:
            nodo_hijo = trie[nodo][2].get(letra_hijo)

            if coste < distancia:
                pila.appendleft((analizado, nodo_hijo, coste + 1))  # Insercion

            if analizado < len(palabra):  # Si hay al menos 1 carácter sustituible
                pila.appendleft((analizado+1, nodo_hijo, coste +
                                 (letra_hijo != palabra[analizado])))  # Sustitucion

    return cercanos
```

En cuanto a Damerau, realizamos el mismo procedimiento añadiendo la ramificación del estado de la permutación: crearemos un estado por cada nieto del nodo donde incrementamos, además de la distancia, el número de símbolos analizados de la cadena. Este último incremento es de dos unidades.

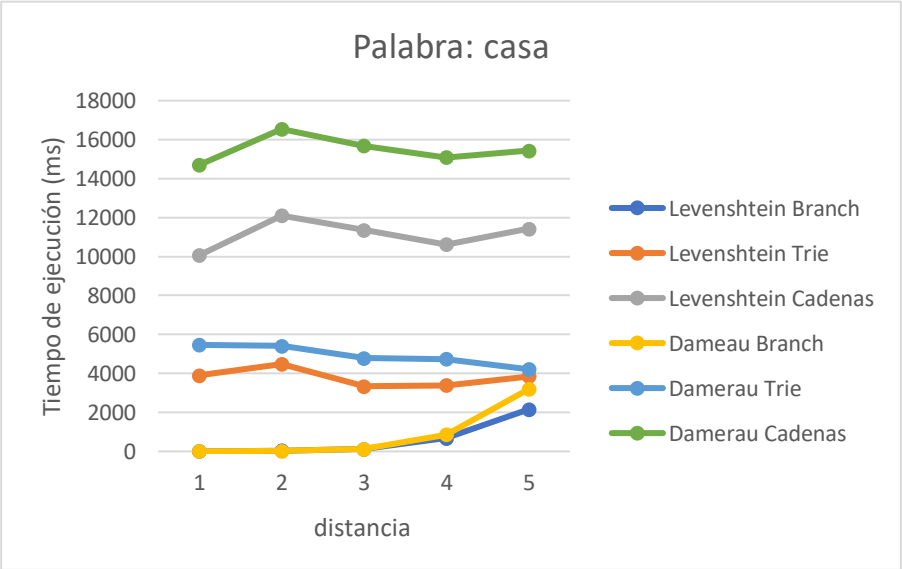
```
if letra_actual in trie[nodo_hijo][2]:
    nieto = trie[nodo_hijo][2][letra_actual]
    pila.appendleft((analizado + 2, nieto, coste + 1))
```

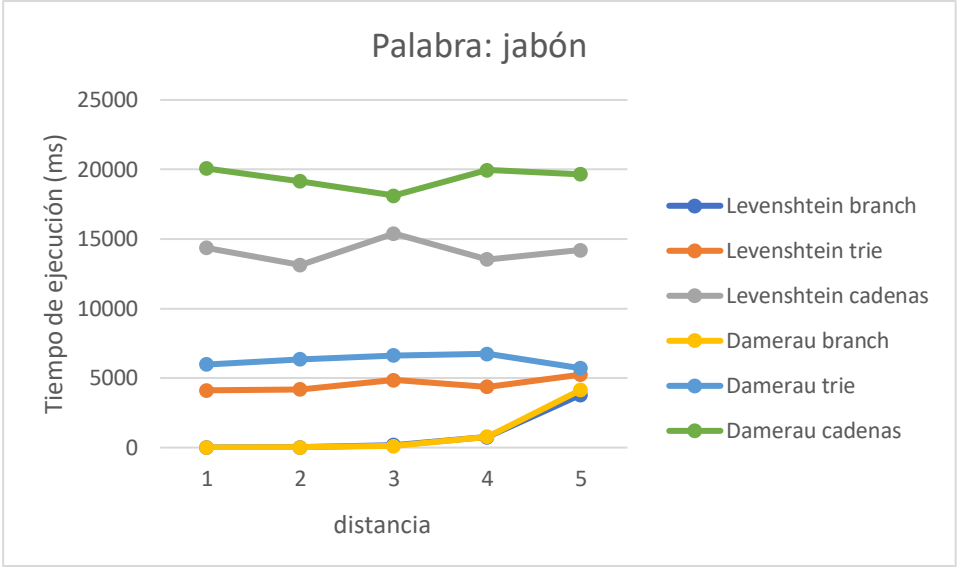
Análisis de datos y conclusiones

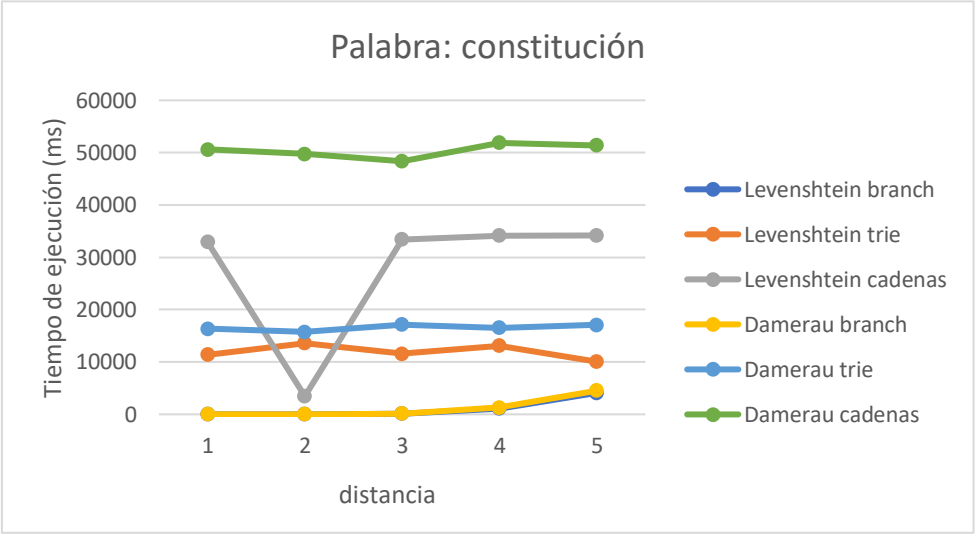
Para realizar un análisis de los datos y obtener el algoritmo más eficiente, hemos realizado un experimento que ha consistido en lanzar las palabras *casa, jabón, constitución* y *S3afg4ew* a todos los algoritmos implementados, variando el valor de la distancia deseada en un rango del 1 al 5. Esto nos ha proporcionado los tiempos de ejecución (en ms) que se pueden observar en la siguiente tabla:

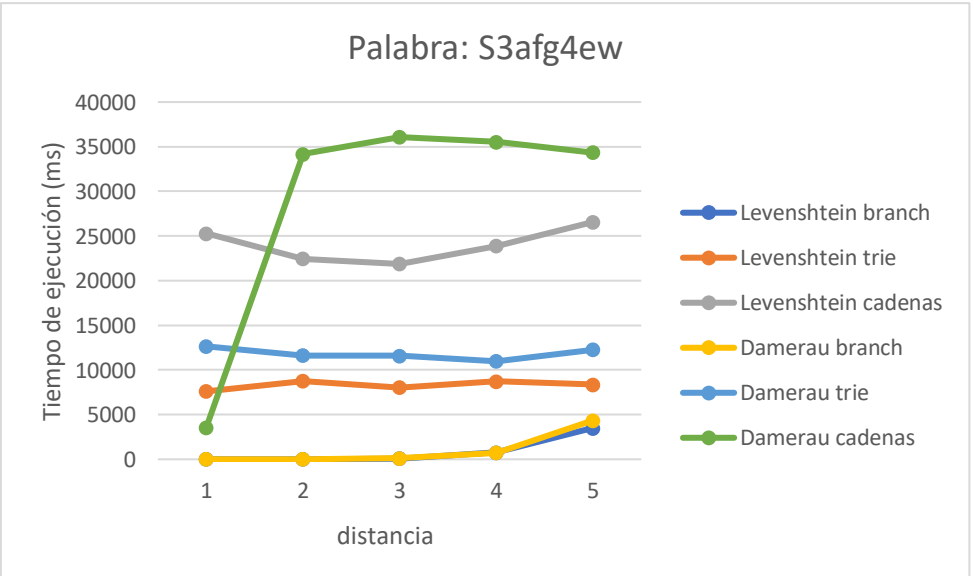
		Levenshtein			*Damerau*		
		Branch	Trie	Cadenas	Branch	Trie	Cadena
Casa	1	1.68	3893.97	10068.51	1.44	5465.23	14703.06
	2	36.20	4481.52	12098	15.77	5406.78	16539.59
	3	115.55	3346.31	11360.74	128,78	4780.18	15679.14
	4	681.86	3396.86	10624.08	865.34	4745.35	15080.08
	5	2158.84	3850.52	11426.07	3197.29	4213.96	15437.23
Jabón	1	1.53	4119.05	14352.91	1.05	5974.65	20066.70
	2	12.91	4177.69	13129.28	12.06	6359.66	19150.52
	3	189.93	4854.46	15393.77	116.28	6626.10	18127.74
	4	747.54	4382.31	13543.80	783.35	6744.63	19962.38
	5	3797.81	5243.84	14195.05	4149.39	5705.06	19665.91
Constitución	1	1.71	11384.95	32939.31	1.81	16370.45	50634.63
	2	34.07	13575.94	3467.24	19.90	15759.66	49746.45
	3	126.58	11562.63	33413.57	146.14	17127.52	48358.91
	4	1103.07	13122.22	34175.76	1304.60	16560.71	51861.08
	5	4097.31	10061.17	34180.32	4515.92	17105.43	51404.62
S3afg4ew	1	0.90	7603.04	25298.99	1.77	12650.62	3544.49
	2	10.84	8755.92	22446.38	11.42	11632.12	34160.16
	3	89.43	8055.99	21871.39	109.93	11608.06	36078.03
	4	771.91	8730.16	23902.41	739.01	10971.01	35546.45
	5	3508.71	8378.51	26567.38	4355.60	12257.04	34359.77

Para que se pueda ver con más claridad, hemos construido las gráficas distancia-tiempo de cada una de las palabras.




Palabra: jabón
Tiempo de ejecución (ms)
25000
20000
15000
10000
5000
0
1
2
3
4
5
distancia
Levenshtein branch
Levenshtein trie
Levenshtein cadenas
Damerau branch
Damerau trie
Damerau cadenas


Palabra: constitución
Tiempo de ejecución (ms)
60000
50000
40000
30000
20000
10000
0
1
2
3
4
5
distancia
Levenshtein branch
Levenshtein trie
Levenshtein cadenas
Damerau branch
Damerau trie
Damerau cadenas


Palabra: S3afg4ew
Tiempo de ejecución (ms)
40000
35000
30000
25000
20000
15000
10000
5000
0
1
2
3
4
5
distancia
Levenshtein branch
Levenshtein trie
Levenshtein cadenas
Damerau branch
Damerau trie
Damerau cadenas

A la vista de los resultados obtenidos, concluimos que claramente el algoritmo más rápido y el que, por tanto, utilizaremos en la implementación de la mejora de nuestro proyecto es del de Branch.

Modificaciones

Indexador

Se ha creado un diccionario que tiene como clave los distintos campos de una noticia y como valor un trie generado con el texto que contienen. Para ello hemos usado el código explicado anteriormente.

```
indices["article"] = indiceInvertidoArticle
indices["title"] = indiceInvertidoTitle
indices["summary"] = indiceInvertidoSummary
indices["keywords"] = indiceInvertidoKeywords
indices["date"] = indiceInvertidoDate

tries["article"] = altL.generarTrie(articleString)
tries["title"] = altL.generarTrie(titleString)
tries["summary"] = altL.generarTrie(summaryString)
tries["keywords"] = altL.generarTrie(keywordsString)
tries["date"] = altL.generarTrie(dateString)
```

Recuperador

Habiendo cargado los tries previamente, obtenemos de la *query* qué tipo de trie tenemos que consultar en caso de que más tarde tengamos que recorrerlo.

```
#for word in query:
while len(query) > 0:
    word = query.pop(0)
    postingList = indiceInvertidoArticle
    trie = trieArticle
    if word.startswith("article:"):
        #print("Buscando en el cuerpo de la noticia...")
        word = word[8:]
        postingList = indiceInvertidoArticle
        trie = trieArticle
    if word.startswith("title:"):
        #print("Buscando por titulo...")
        word = word[6:]
        postingList = indiceInvertidoTitle
        trie = trieTitle
    if word.startswith("summary:"):
        #print("Buscando por sumario...")
        word = word[8:]
        postingList = indiceInvertidoSummary
        trie = trieSummary
    if word.startswith("keywords:"):
        #print("Buscando por keywords...")
        word = word[9:]
        postingList = indiceInvertidoKeywords
        trie = trieKeywords
    if word.startswith("date:"):
        #print("Buscando por fecha...")
        word = word[5:]
        porFecha = True
        postingList = indiceInvertidoDate
        trie = trieDate
```

Esto último lo detectaremos mirando los parámetros que hemos especificado en la consulta, pudiendo diferenciar el tipo de algoritmo que vamos a usar: Levenshtein o Damerau, ambos implementados con la variante de ramificación. En caso de que sí que calculemos distancias, almacenaremos los resultados obtenidos en una variable, *newWords*.

```
#Detectar si hay que aplicar distancias
particion = word.split('%')
if (len(particion) > 1):
    #Procesar distancia Levenshtein
    newWords = altL.lev_branch(trie,particion[0],int(particion[1]))
else:
    particion = word.split('@')
    if (len(particion) > 1):
        #Procesar distancia Damerau
        newWords = altL.dam_branch(trie,particion[0],int(particion[1]))
```

Si usamos distancias, es decir, la variable *newWords* es mayor que cero, se insertan todas estas palabras en la consulta y se procede a realizar la consulta de forma habitual.

```
if (len(newWords) > 0):
    #Añadir a la consulta las nuevas palabras
    query.insert(0,particion[0])
    for newWord in newWords:
        if (isNot == 1):
            query.insert(0,'NOT')
        query.insert(0,'OR')
        query.insert(0,newWord)
else:
    if word == 'AND' or word == 'OR':
        operador = word
        posibleFinal = False
    elif word == 'NOT':
        if isNot == 0:
            isNot = 1
        else:
            isNot = 0
        posibleFinal = False
    else:
        if not porFecha:
            word = word.lower()
            if isNot == 0:
                wordList.append(word)
        porFecha = False
        posting = postingList.get(word,[])
        res = operar(operador,isNot,res,posting,noticias)
        #ponemos los operadores a su forma estandar
        operador = 'AND'
        isNot = 0
        posibleFinal = True
```