Proyecto algorítmica:
Estudio del mejor algoritmo de distancia entre palabras

Miguel García Bohigues
Genís Martínez Sanchis
Angel Giner Vidal
Mª Pilar Piqueres Matoses

Enero 2020

1. Introducción

El eje central de las prácticas de la asignatura de algorítmica ha sido la codificación de diversas variantes del algoritmo de Levenstein que calcula la distancia entre dos cadenas de caracteres. Entendemos por distancia entre dos palabras al número de inserciones, borrados o sustituciones que hemos de hacer para convertir una palabra en la otra. Ilustrándolo con un ejemplo, dadas dos palabras P1: AACBDC y P2: ACBBBC, la distancia entre ellas es 3 en tanto que se han utilizado una operación de sustitución, una de borrado y una de inserción.

A	A	C	B	D	-	C
=	↓	=	=	↕	↑	=
A	-	C	B	B	B	C

Figura 1: Ejemplo con dos cadenas

Como podemos ver en el ejemplo, los símbolos comunes se entienden por sustituciones sin peso alguno.
Así, el algoritmo de Levenstein se definiría de manera recursiva mediante la siguiente ecuación:

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

Figura 2: Ecuación recursiva del algoritmo de Levenstein

Donde borr(α), ins(α) y sust(α, β) representan el coste de borrar, insertar y sustituir respectivamente. Para el desarrollo de esta practica hemos determinado que todas las operaciones tienen el mismo coste, pero esto podría no ser siempre así.

Adicionalmente presentamos otro algoritmo, la distancia de Damerau-Levenstein. Dicho algoritmo presenta una ampliación del algoritmo de Levenstein al que se ha añadido la posibilidad de transponer una subcadena, haciendo que, por ejemplo, la distancia entre P1: casa y P2: caas sea 1. La ecuación recursiva de dicho algoritmo se presenta así:

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

Figura 3: Ecuación recursiva del algoritmo de Damerau-Levenstein

2. Distancia cadena contra cadena

Una vez comentado el funcionamiento de los algoritmos a implementar nos disponemos a realizar dicha implementación. La manera más directa de aplicación es enfrentar dos cadenas directamente con estos algoritmos. Dicha implementación se puede hacer fácilmente con una matriz de dimensiones len(P1) * len(P2):

```
def levenstein_distance(word1, word2):
    levMat = np.zeros(dtype=np.int8, shape=(len(word1) + 1,len(word2) + 1))
    #inicializamos la matriz del algoritmo de levenstein
    for i in range(1, len(word1)+1):
        levMat[i,0] = i

    for j in range(1, len(word2) + 1):
        levMat[0,j] = j

    for i in range(1, len(word1) + 1):
        for j in range(1, len(word2) + 1):
            dif = not (word1[i - 1] == word2[j - 1])
            levMat[i,j] = min(levMat[i-1, j], levMat[i-1, j-1], levMat[i, j-1]) + dif

    return levMat[len(word1), len(word2)]
```

Espacialmente es muy costoso tener una matriz de M*N celdas, siendo M y N las longitudes de las palabras a enfrentar. Por ello nos planteamos una reducción espacial. Para el caso de Levenstein vemos que solo necesitamos saber el valor que hemos calculado en el estado anterior, así pues, sustituimos la matriz por dos vectores fila:

```
def better_levenstein_distance(word1, word2):
    c_row = np.zeros(dtype=np.int8, shape=(len(word1) + 1))
    p_row = np.zeros(dtype=np.int8, shape=(len(word1) + 1))
    c_row[0] = 0
    for i in range(1, len(word1) + 1):
        c_row[i] = c_row[i - 1] + 1
    for j in range(1, len(word2) + 1):
        p_row,c_row = c_row,p_row
        c_row[0] = p_row[0] + 1 #esto es para sumarle 1 a la col[0], es decir, simular la
    insercion completa de la palabra
        for i in range(1, len(word1) + 1):
            c_row[i] = min(c_row[i - 1] + 1, #elemento anterior en eje x
                           p_row[i] + 1, #elemento de abajo en y
                           p_row[i - 1] + (word1[i - 1] != word2[j - 1]))#elemento anteior en
     x e y mas si son diferentes (sustitucion por el mismo es 0)
    return c_row[len(word1)]
```

Ahora tenemos un algoritmo con un ahorro espacial considerable y vemos que podemos adaptar dicho algoritmo para incluir la extensión de Damerau.

```
def damerau_levenstein_distance(word1, word2):
    #dado que tenemos que acceder a la pposicios -2 en j tenemos que tener 3 filas
    c_row = np.zeros(dtype=np.int8, shape=(len(word1) + 1))
    p_row = np.zeros(dtype=np.int8, shape=(len(word1) + 1))
    aux_row = np.zeros(dtype=np.int8, shape=(len(word1) + 1)) #fila para acceder a la pos -2
    en j
    c_row[0] = 0 #caso i = j = 1
    for i in range(1, len(word1) + 1): #caso i > 0 j = 0
        c_row[i] = c_row[i - 1] + 1

    p_row,c_row = c_row,p_row
    for i in range(0, len(word1) + 1): #caso j = 1 i > 0
        c_row[i] = min(c_row[i - 1] + 1, #elemento anterior en eje x
                       p_row[i] + 1, #elemento de abajo en y
                       p_row[i - 1] + (word1[i - 1] != word2[0]))

    for j in range(2, len(word2) + 1):
        aux_row,p_row,c_row = p_row,c_row,aux_row
        c_row[0] = p_row[0] + 1 #esto es para sumarle 1 a la col[0], es decir, simular la
    insercion completa de la palabra
        for i in range(1, len(word1) + 1):
            if (word1[i -1] == word2[j - 2] and word2[j -1] == word1[i - 2]):
```

```
                c_row[i] = min(c_row[i - 1] + 1, #elemento anterior en eje x
                               p_row[i] + 1, #elemento de abajo en y
                               p_row[i - 1] + (word1[i - 1] != word2[j - 1]),#elemento anteior en
    x e y mas si son diferentes (sustitucion por el mismo es 0)
                               (aux_row[i - 2] + 1)) #transposicion
            else:
                c_row[i] = min(c_row[i - 1] + 1, #elemento anterior en eje x
                               p_row[i] + 1, #elemento de abajo en y
                               p_row[i - 1] + (word1[i - 1] != word2[j - 1]))

    return c_row[-1]
```

Los tiempos asimtóticos de estos algoritmos vemos que son lineales con la talla de las palabras $\Theta(N * M)$.

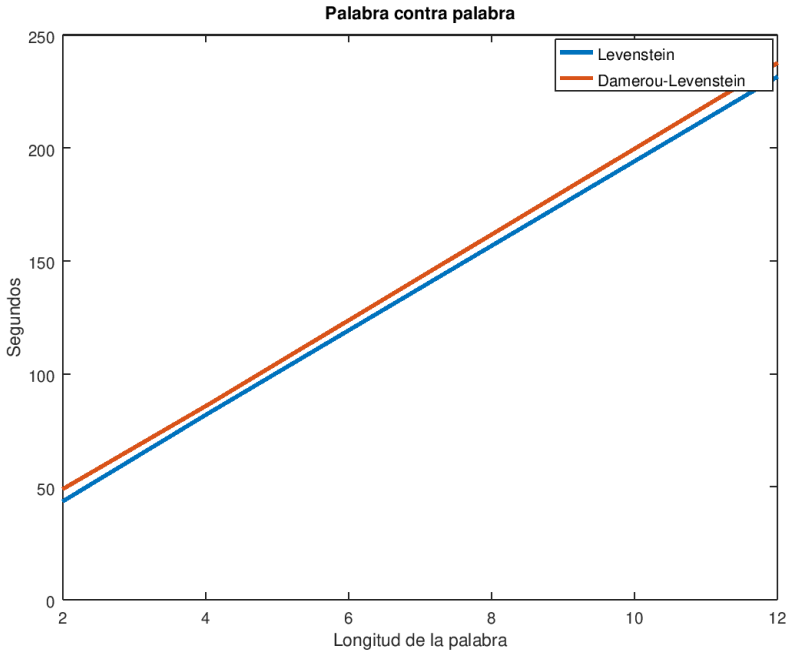


Figura 4: resultados sobre el quijote

Los resultados se han obtenido realizando la distancia para cada palabra dentro del quijote contra el token dado.

Siendo que vamos a aplicar esta distancia sobre una colección de noticias vemos necesaria la optimización al máximo de este algoritmo para reducir el tiempo de ejecución del recuperador de noticias.

3. Cadena contra trie

Tal y como se ha mencionado en la sección anterior, el cálculo de la distancia de Levenstein y Damerau-Levenstein se va a realizar múltiples veces en el recuperador de noticias, por lo que necesitamos mejorar la eficiencia de los algoritmos a utilizar.
Fruto de esto introducimos el Trie. El trie es una estructura en forma de árbol utilizada para almacenar el diccionario de términos en sistemas de recuperación de información. Como podemos ver en la figura, la raíz

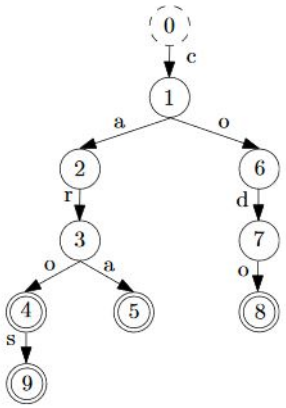


Figura 5: Ejemplo de Trie

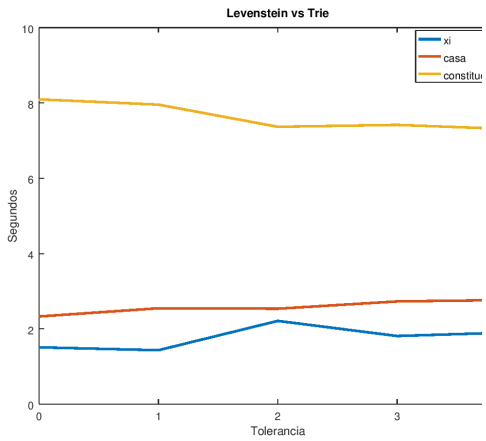
presenta la cadena vacía, las hojas presentan palabras completas del diccionario. Por ejemplo, el nodo 5 representa la palabra " cara". La ecuación recursiva del algoritmo de Levenstein para este caso nos la proporcionaba el boletín de la práctica.
La implementación del algoritmo para el caso del trie es la siguiente:

```
def levenstein_vs_trie(tri, palabra, k):
    nod = tri.nodoRaiz
    matrix = np.zeros(dtype=np.int8, shape=(len(tri.array) + 1, len(palabra) + 1)) #creamos
    una matriz de nodos (filas) por simbolos de la palabra (columnas)
    result = []

    matrix[nod.idi,0] = 0 #case 1

    for i in range(1,len(palabra)+1): #case 2
        matrix[nod.idi,i] = i

    for n in tri.array:
        if(n != nod):
            matrix[n.idi,0] = n.profundidad#case 3
            for i  in range(1,len(palabra) + 1):#case 4
                matrix[n.idi,i] = min(
                    matrix[n.idi,i - 1] + 1,
                    matrix[n.nodoPadre.idi,i] + 1,
                    matrix[n.nodoPadre.idi,i - 1] + (palabra[i-1] != n.letraLlegada)
                )

    for n in tri.array:
        if(n != nod):
            if matrix[n.idi,-1] <= k:
                if n.palabra != None:
                    if n.palabra not in result:
                        result.append(n.palabra)

    return result
```

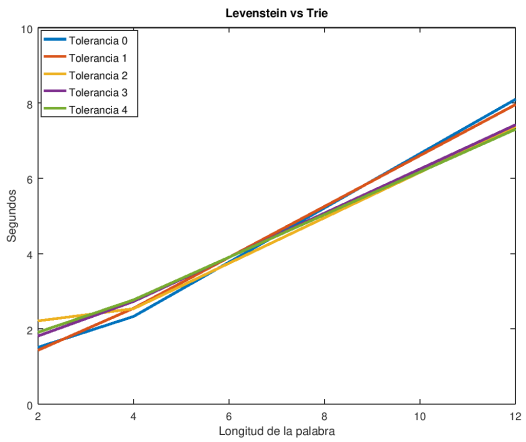
Y la del algoritmo de Damerau-Levenstein ha sido:

```
def damerau_levenstein_vs_trie(tri, word, toler):

    nod = tri.nodoRaiz
    matrix= np.zeros(dtype = np.int8,shape = (len(tri.array)+1, len(word)+1))
    res = []

    matrix[nod.idi, 0] = 0 #case 1

    for ln in range(1, len(word)+1): #case 2
        matrix[nod.idi, ln] = ln

    for n in tri.array:
        if n != nod:
            matrix[n.idi,0] = n.profundidad #case 3
            for i in range (1,len(word) + 1):
                if (word[i-1] == n.nodoPadre.letraLlegada and word[i-2] == n.letraLlegada) and
 n.nodoPadre.nodoPadre != None:
                    matrix[n.idi,i] = min(
                        matrix[n.idi,i - 1] + 1,
                        matrix[n.nodoPadre.idi,i] + 1,
                        matrix[n.nodoPadre.idi,i - 1] + (word[i-1] != n.letraLlegada),
                        matrix[n.nodoPadre.nodoPadre.idi,i - 2] + 1
                    )
                else:
                    matrix[n.idi,i] = min(
                        matrix[n.idi,i - 1] + 1,
                        matrix[n.nodoPadre.idi,i] + 1,
                        matrix[n.nodoPadre.idi,i - 1] + (word[i-1] != n.letraLlegada)
                    )

    for n in tri.array:
        if(n != nod):
            if matrix[n.idi,-1] <= toler:
                if n.palabra != None:
                    if n.palabra not in res:
                        res.append(n.palabra)

    return res
```

Vemos que en este caso hemos necesitado hacer uso de una matriz de X*N, siendo X el número de nodos del trie y N la longitud de la palabra, pero a cambio hemos conseguido reducir el coste temporal, no en el caso particular de una palabra contra una palabra, si no en el caso general de un conjunto de palabras con prefijo común contra una palabra.
Es decir, para las palabras [casa, caso, casar, casi] y la palabra [casa], el algoritmo palabra contra palabra realiza $3(4*4)+4*5 = 68 iteraciones$ debido a que no se aprovecha de que, en este caso, las palabras tienen prefijo común (cas) cosa que si hace el trie. Así el primero realiza las mismas comprobaciones para las 4 palabras y, por otra parte el segundo, llegado al nodo con el prefijo " cas"no repite cálculos.



(a) tiempo respecto a tolerancia



(b) tiempo respecto a segundos

Como podemos observar, el coste ahora es $\Theta(N * X)$, siendo N la longitud de la palabra y X en número de nodos del trie. El cálculo de las distancias es ahora más eficiente, siendo que tiene en cuenta los prefijos a la hora de calcular las distancias para palabras que derivan de un mismo lexema, pero vemos que, aunque haya superado la tolerancia para un prefijo p, sigue derivando los estados del árbol hasta completarlo entero. Esto lo podemos paliar utilizando el método de ramificación y poda.

4. Cadena contra trie con ramificación y poda de estados

Tal y como hemos mencionado en la sección anterior, el cálculo de cadena contra trie era mejorable. Para ello solo tenemos que añadir unos criterios de descarte a la hora de elegir si ramificamos un estado o no. Es decir, si dado un estado y una distancia es factible realizar cierta operación.

- Al realizar una inserción, vemos que la posición en la que estamos de la palabra no incrementa, pero sí lo hacen la distancia y el nodo en el que estamos, por lo que un criterio es: La distancia actual +1 debe ser menor o igual que la total pedida, si no no hace falta aplicar dicha operación y además, el nodo en el que estamos debe tener algún hijo.
- Al borrar no modificamos el nodo en el que nos encontramos, pero sí la posición sobre la palabra y la distancia, por lo que un criterio válido es: La distancia +1 debe ser menor o igual que la total pedida, y la posición de la palabra en la que nos encontramos debe ser menor o igual que la longitud de la palabra.
- Para la sustitución, siendo que se incrementa tanto la posición del nodo, como la posición sobre la cadena, como (posiblemente) la distancia, el criterio de poda depende de todos tres: La distancia debe ser menor o igual que la total dada, el nodo actual debe tener algún hijo y la posición de la palabra en la que nos encontramos debe ser menor que la longitud total de la palabra.

Una anotación sobre el último punto: la distancia, al contrario que en los dos puntos anteriores, puede ser menor o igual que la tolerancia máxima ya que se puede dar el caso de que se sustituyan dos letras que sean iguales, cosa que no aumenta la distancia entre las palabras.
De ese modo los algoritmos serían los siguientes:

```
1  def levenstein_vs_trie_ramificacion(tri, palabra, k):
2      lista = [(0,0,0)] # (elementode la plalabra, nodo, distancia a mi coraxon)
3      nod = tri.root()
4      result = []
5      evitreps = {}
6      while(len(lista)):
7          i,n,d = lista.pop()
8
9          if n == 0:
10             nod = tri.array[n]
11         else:
12             nod = tri.array[n-1]
13
14         #insercion
15         if(d+1 <= k and len(nod.hijos) > 0 and i <= len(palabra)):
16             for x in nod.hijos.keys():
17                 nn = nod.hijos.get(x, 0)
18                 lista.append((i,nn.idi ,d+1))
19
20         #borracion
21         if(d+1 <= k and i < len(palabra)): # ojo que aqu  podr a ir un <=, no lo se
22             lista.append((i+1,nod.idi ,d+1))
23
24         #sustitutusao
25         if(d <= k and len(nod.hijos) > 0 and i < len(palabra)): #aqui no puede ir un <= poque
       si no palabra[i] hace pum pum
26             for x in nod.hijos.keys():
27                 nn = nod.hijos.get(x, 0)
28                 if nn.letraLlegada != None:
29                     lista.append((i+1,nn.idi,d + (palabra[i] != nn.letraLlegada)))
30                 else:
31                     lista.append((i,nn.idi,d))
32
```

```
        if(d <= k and nod.palabra != None and (i == len(palabra))):
            cosa = evitreps.get(nod.palabra, 0)
            if cosa == 0:
                evitreps[nod.palabra] = 1
                result.append(nod.palabra)

    return result
```

Y para Damerau:

```
def levenstein_vs_trie_ramificacionD(tri, palabra, k):
    lista = [(0,0,0)]
    nod = tri.root()
    result = []
    evitreps = {}

    while (len(lista)):
        i,n,d = lista.pop()

        if n == 0:
            nod = tri.array[n]
        else:
            nod = tri.array[n-1]

        #insercion
        if(d+1 <= k and i <= len(palabra) and len(nod.hijos) > 0):
            for x in nod.hijos.keys():
                nn = nod.hijos.get(x,0)
                lista.append((i,nn.idi,d+1))

        #borrado
        if(d+1 <= k and i < len(palabra)):
            lista.append((i+1,nod.idi, d+1))

        #sustitucion
        if(d <= k and len(nod.hijos) > 0 and i < len(palabra)):
            for x in nod.hijos.keys():
                nn = nod.hijos.get(x, 0)
                if nn.letraLlegada != None:
                    lista.append((i+1,nn.idi,d + (palabra[i] != nn.letraLlegada)))
                else:
                    lista.append((i,nn.idi,d))

        #vueltacionNietos
        if(d+1 <= k and len(nod.hijos) > 0 and i+1 < len(palabra)):
            for x in nod.hijos.keys():
                nn = nod.hijos.get(x,0)
                if(nn.letraLlegada == palabra[i+1]):
                    for y in nn.hijos.keys():
                        nietos = nn.hijos.get(y,0)
                        if(nietos.letraLlegada == palabra[i]):
                            lista.append((i+2,nietos.idi,d+1))

        #result
        if(d <= k and nod.palabra != None and (i == len(palabra))):
            cosa = evitreps.get(nod.palabra, 0)
            if cosa == 0:
                evitreps[nod.palabra] = 1
                result.append(nod.palabra)

    return result
```

Así, tenemos un algoritmo mejorado que optimiza al máximo la estructura dada.

5. Comparación de tiempos

Una vez hemos presentado todas las implementaciones y la motivación que ha dado lugar a ellas, presentamos los resultados. Los tiempos mostrados en la tabla anterior, en segundos, muestran el tiempo de ejecución de

	palabra contra palabra	palabra contra trie	palabra contra trie RP
xi	43.5954	1.4330	0.1440
casa	81.8403	2.5357	0.6977
constitución	231.5144	7.4180	0.7800

las diferentes implementaciones con respecto al texto del quijote. En el se puede observar como el cambio de palabra contra palabra a palabra contra trie ha supuesto una disminución del tiempo de ejecución considerable. Además, vemos que el refinamiento hecho con la ramificación ha resultado en tiempos muy bajos, próximos a 0. Se han seleccionado para todos los tiempos para tolerancia = 5.