Memoria Práctica 1 - ALT

Grupo: 4CO21
Ainoa Palomino Pérez

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Tareas obligatorias

1) Recuperar secuencia de operaciones de edición

Para esta tarea he modificado el código ya proporcionado de distancia Levenshtein añadiendo al final un bucle *while* que empieza en posición de matriz con distancia mínima D[x,y] y acabe cuando llega a primera celda de matriz D[0,0]. Es decir un bucle recorre el camino de distancia mínima encontrado durante la ejecución de Levenshtein al revés y así recupera la secuencia de operaciones necesarias.
En cada iteración del bucle se va a buscar operación que supone distancia mínima en posiciones más cercanas al inicio del bucle. Es decir se va comprobando índices anteriores de matriz D[x-1,y],D [x-1,y-1],D[x,y-1] y elige un mínimo de ellos. Además va a añadir una operación de inserción, sustitución o borrado respectivamente a los índices elegidos en el array con operaciones.
Cuando acabe el bucle se hace un reverse del dicho array y return de la función.

2) Implementación Levenshtein con reducción de coste espacial y con threshold

En este caso, partiendo del código obtenido en el ejercicio anterior he utilizado dos vectores (*vprev, vcurrent*) mediante la función np.zeros; esto se hace pasándole un tamaño, que para ambos es (*lenX + 1*), puesto que es el del bucle más interno. Después, he adaptado el código anterior a las nuevas modificaciones; y para acabar, he intercambiado los dos nuevos vectores (*vnext, vcurrent = vcurrent, vnext*).

```
def levenshtein_reduccion(x, y, threshold=None):
    lenX, lenY = len(x), len(y)
    vcurrent = np.zeros(lenX + 1, dtype=np.int)
    vnext = np.zeros(lenX + 1, dtype=np.int)
    for i in range(1, lenX + 1):
        vcurrent[i] = vcurrent[i - 1] + 1
    for j in range(1, lenY + 1):
        vnext[0] = vcurrent[0] + 1
        for i in range(1, lenX + 1):
            vnext[i] = min(vcurrent[i] + 1,
                           vnext[i - 1] + 1,
                           vcurrent[i - 1] + (x[i - 1] != y[j - 1]),
            )
        vnext, vcurrent = vcurrent, vnext
    return vcurrent[lenX]
```

Figura 1. Versión con reducción coste espacial

Para la ampliación de este apartado, con un nuevo parámetro umbral threshold que permite parar de calcular la distancia cuando esté claro que el valor final superará el umbral, se ha

mantenido el mismo código de la versión con reducción del coste espacial, añadiendo una líneas al final para medir cuando se sobrepasa el umbral, mediante una nueva variable *paradaPorThreshold* con valor por defecto a True; esto lo calculamos dentro del bucle interno (el bucle '*j*') comprobándolo con '*lenX*' y fuera de los bucles, con '*lenY*', que en caso de ser mayor, detendremos el algoritmo y obtendremos el resultado '*threshold+1*' (es decir, *paradaPorThreshold*=True.

```python
def levenshtein(x, y, threshold):
    lenX, lenY = len(x), len(y)
    vcurrent = np.zeros(lenX + 1, dtype=np.int)
    vnext = np.zeros(lenX + 1, dtype=np.int)
    for i in range(1, lenX + 1):
        vcurrent[i] = vcurrent[i - 1] + 1
    for j in range(1, lenY + 1):
        vnext[0] = vcurrent[0] + 1
        paradaPorThreshold = True
        if(vnext[0] <= threshold): paradaPorThreshold = False
        elif(vnext[0] == threshold and lenX - i == lenY - j): paradaPorThreshold = False
        for i in range(1, lenX + 1):
            vnext[i] = min(vnext[i - 1] + 1,
                           vcurrent[i] + 1,
                           vcurrent[i - 1] + (x[i - 1] != y[j - 1]))
        if(vnext[i] < threshold): paradaPorThreshold = False
        elif(vnext[i] == threshold and lenX - i == lenY - j): paradaPorThreshold = False
        if(paradaPorThreshold): return threshold+1
        vnext, vcurrent = vcurrent, vnext
    return vcurrent[lenX]
```

Figura 2. Versión con reducción coste espacial y threshold

Además, otra posible versión sin tener en cuenta la reducción de coste espacial sería la siguiente, donde se ve con mayor claridad que se puede confirmar que el coste superará el threshold una vez calculada una etapa (es decir, una columna).

```python
def levenshtein(x, y, threshold):
    lenX, lenY = len(x), len(y)
    D = np.ones((lenX + 1, lenY + 1)) * np.inf #infinito positivo
    for i in range(0, lenX + 1):
        D[i, 0] = i
    for j in range(0, lenY + 1):
        D[0, j] = j
        d = lenX / lenY
        for i in range(max(math.floor(d*i-threshold), 1), min(math.ceil(d*i+threshold), lenX + 1)):
            colMin = np.inf
            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]),
            )
            if colMin > D[i,j]:
                colMin = D[i,j]
        if colMin > threshold:
            return threshold+1
            #return None
    return D[lenX, lenY]
```

Figura 3. Versión sin reducción coste espacial y threshold

3) Completar la clase SpellSuggester

Para completar la clase SpellSuggester simplemente se han añadido las líneas de código que podemos observar en la *Figura 4*, pero con todas las funciones implementadas en distancias. En caso de que el argumento *flatten* esté a True (opción por defecto), nos devolverá una lista; podemos observar cómo está implementado esto en las dos penúltimas líneas de la *Figura 4*. Pero en caso contrario (*flatten = False*), nos devolverá una lista de listas de palabras; en el código corresponde a: new_result.append(palabra). En este último caso, la lista i-ésima contiene las palabras a distancia *i* (para *i* hasta *threshold* incluido).

```
result=[]
new_result=[]
for palabra in self.vocabulary:
    if distance=="levenshtein_m":
        if distancias.levenshtein_matriz(term,palabra,threshold)==threshold:
            new_result.append(palabra)

if flatten:
    new_result = [word for wlist in result for word in wlist]

return new_result
```

Figura 4: Clase SpellSuggester

Por otro lado, dentro de test_spellsuggester.py he modificado algunas líneas de código (dejando las originales comentadas para poder observar bien los cambios), para así poder probar las distancias implementadas en distancias.py. He añadido la lista *longitudes*, además de modificar las líneas que incluyen los .append() (método que agrega un elemento al final de la lista).

```
longitudes = []

for threshold in range(1, 4+1):
    newresul = spellsuggester.suggest(palabra, distance=dstname,
                                      threshold=threshold, flatten=False)
    #assert(all(x == y for x,y in zip(resul,newresul)))
    #resul = newresul
    resul.append(newresul)
for x in resul:
    #longitudes = [len(x) for x in resul]
    longitudes.append(len(x[0]))
print(" -",palabra,longitudes,sum(longitudes))
f.write(f'{palabra} {threshold} {longitudes}\n{resul}\n')
```

Figura 5: Clase test_spellsuggester.py

4) Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas utilizando la clase SpellSuggester

Para poder permitir la búsqueda aproximada de cadenas he tenido que modificar el fichero SAR_lib.py, añadiendo el método “set_spelling”, que permite activar o desactivar la corrección ortográfica. También he modificado el método “get_posting” para poder realizar la búsqueda con tolerancia. En este método he añadido un bucle para buscar el término en self.index, en caso de no estar, tendría que utilizar suggest con este término. La lista de

palabras que devuelva, habrá que buscar sus posting de todas las contenidas en la lista, mediante otro bucle. Al obtener la posting list de todas las palabras, las cuales se guardan en el array “res[i]”, se usa igual que en la versión anterior, da igual si proviene del corrector o en el diccionario.

Ampliaciones

1) Cota optimista para ahorrar evaluaciones de la distancia de Levenshtein e integrarlo en SpellSuggester para utilizarlo en el recuperador de SAR

Para realizar esta ampliación he completado el método levenshtein_cota_optimista. Para hacer esto en primer lugar he añadido a un diccionario todas las letras de ambas cadenas. A continuación con un simple bucle se recorre el diccionario de forma que en la variable diferencia se suman las apariciones de dicha letra en la primera cadena y se restan las de la segunda para después actualizar la variable resultado con el valor absoluto de esta diferencia. Por último se comprueba si el resultado es mayor o igual que el threshold dado, en cuyo caso se devuelve threshold+1 y si no devuelve el resultado calculado por levenshtein.

La integración con spellsuggester ha sido trivial, simplemente hemos tenido que añadir esta función que acabamos de describir al diccionario de funciones que se le pasa al construir el objeto.

2) Versión restringida de Damerau-Levenshtein

Para calcular esta versión, me he basado en la siguiente ecuación:

$$D(i,j) = \text{mín} \begin{cases} 0 & \text{si i=0 } y \text{ j=0} \\ D(i-1,j)+1 & \text{si i>0} \\ D(i,j-1)+1 & \text{si j>0} \\ D(i-1,j-1) & \text{si } i>0, j>0, x_i = y_j \\ D(i-1,j-1)+1 & \text{si } i>0, j>0, x_i \neq y_j \\ D(i-2,j-2)+1 & \text{si } i>1, j>1, x_{i-1} = y_j, x_i = y_{j-1} \end{cases}$$

La principal diferencia entre la versión de Damerau-Levenshtein y la de Levenshtein es que la primera incluye transposiciones entre las operaciones que permite (además de las ya disponibles en la versión de Levenshtein: inserciones, sustituciones o borrados). Cada llamada recursiva corresponde a uno de los casos mencionados anteriormente:

- D(i-1,j)+1 corresponde a un borrado
- D(i,j-1)+1 corresponde a una inserción
- D(i-1,j-1)+1 corresponde a una coincidencia o discordancia (dependiendo de si los respectivos símbolos *x* e *y* son iguales).
- D(i-2,j-2)+1 corresponde a una transposición

Además, es importante mencionar que en esta ampliación he utilizado 3 vectores columna (en vez de los 2 usados en la versión de Levenshtein) debido a que aparece una dependencia '*j-2*' (*Figura 6*). No obstante, esta versión no nos da necesariamente la distancia mínima.

```
if i > 1 and j > 1 and x[i - 2] == y[j - 1] and x[i - 1] == y[j - 2]:
    if x[i - 1] == y[j - 1]:
        D[i, j] = min(D[i - 1, j] + 1, D[i, j - 1] + 1, D[i-1][j-1], D[i-2][j-2] + 1)
    else:
        D[i, j] = min(D[i - 1, j] + 1, D[i, j - 1] + 1, D[i-1][j-1] + 1, D[i-2][j-2] + 1)
```

Figura 6: dependencia j-2 damerau_restricted_matriz

3) Ampliación Damerau-Levenshtein restringido (recuperar camino)

Esta ampliación supone un cambio pequeño a su versión anterior de Levenstein. Se recorre el algoritmo Damerau-Levenshtein restringido y un bucle que recupera el camino.
Dentro de este bucle cuando se busca un mínimo de 3 posiciones en matriz (y sus respectivas operaciones) hay que añadir una opción más. En el caso cuando es posible intercambiar dos letras (es decir x-1==y and x == y-1) hay que comprobar para valor mínimo en matriz y si esto es el caso añadir esta operación en el array resultante. Con esta modificación se consigue recuperar el camino del algoritmo.

4) Versión intermedia de Damerau-Levenshtein

Esta versión es similar a la anterior, pero ahora se consideran solamente los casos en los que |x| + |y| <= cte, para una constante cte establecida. La principal diferencia es que, en este caso, hemos utilizado 4 vectores columna en vez de 3 (*Figura 7*), debido a la dependencia *'j-3'* que aparece, como podemos observar en la *Figura 8*.

```
lenX, lenY = len(x), len(y)
vec1 = np.zeros(lenX + 1, dtype=np.int)
vec2 = np.zeros(lenX + 1, dtype=np.int)
vec3 = np.zeros(lenX + 1, dtype=np.int)
vec4 = np.zeros(lenX + 1, dtype=np.int)
```

Figura 7: Versión damerau_intermediate

```
elif j > 2 and i > 1 and x[i-2] == y[j-1] and x[i-1] == y[j-3]:
    D[i,j] = min(minInit, D[i-2][j-3] + 2)
elif i > 2 and j > 1 and x[i - 3] == y[j-1] and x[i-1] == y[j-2]:
    D[i,j] = min(minInit, D[i-3][j-2] + 2)
```

Figura 8: dependencia j-3 en damerau_intermediate