




**Trabajo Fin de Máster**

# Distancia de Levenshtein como clasificador de textos

Alexandre Domínguez Prieto

2022/2023

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Máster en Técnicas Estadísticas

**Trabajo Fin de Máster**

# Distancia de Levenshtein como clasificador de textos

Alexandre Domínguez Prieto

Febrero, 2023

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

# Trabajo propuesto

| |
|---|
| **Título: Distancia de Levenshtein como clasificador de textos** |
| **Nombre de la empresa:** Trucksters<br>**Tutora en la empresa:** Lucía Tajuelo López<br>**Tutores académicos:** Jose Ameijeiras Alonso y María José Guinzo Villamayor |
| **Descrición del trabajo realizado:**<br>Las empresas que trabajan con bases de datos grandes pueden tener duplicados en sus bases de datos. Detectar estos duplicados es una tarea que requiere tiempo, especialmente si los datos son palabras en texto plano.<br>El objetivo de este trabajo es desarrollar un algoritmo que permita detectar duplicados en nuestra base de datos. Este algoritmo, basado en la distancia de Levenshtein, se implementará para clasificar nombres de localizaciones postales y nombres de clientes. |
| **Recomendaciones**<br><br>El algoritmo se implementará en Python. Lectura introductoria:<br>https://blog.paperspace.com/measuring-text-similarity-using-levenshtein-dis tance/ |
| **Fechas de las prácticas** |
| **Lugar de trabajo y horario** Remoto y flexible |
| **Convenio** |
| **Otras observaciones** |

# Índice general

## Resumen

En este Trabajo de Fin de Máster introducimos las distancias entre palabras, centrándonos en la distancia de Levenshtein y la distancia de Damerau-Levenshtein. Tras exponer y demostrar el correcto funcionamiento de algoritmos para calcular ambas, se define un algoritmo que permite agrupar palabras según cuan parecidas sean. Con una serie de conjuntos de datos, proporcionados por la empresa Trucksters, buscamos encontrar los mejores parámetros posibles para dicho algoritmo de clasificación. El objetivo final de este trabajo es estudiar el comportamiento de este algoritmo con vistas a su posible uso dentro de la empresa, a mayores se plantea la posibilidad de emplear las distancias estudiadas para incorporar en la página web de la empresa una herramienta para sugerir nombres de usuarios ya existentes en la base de datos de la misma mientras uno trata de introducir uno.

## Abstract

In this Master's Thesis, we introduce the distance between words, focusing on the Levenshtein distance and the Damerau-Lvenshtein distance. After presenting and proving the correctness of algorithms to calculate both distances, we define an algorithm to group words by how similar they are. Using a series of datasets provided by Trucksters we search for the best parameters for said algorithm. The final objective of this paper is to study the behaviour of this algorithm with a view to its possible use in the company. Moreover, we lay out a possible addition to the company's website, a tool that suggests usernames already existent in the company's database when someone is trying to write one.

# Introducción

Trucksters es una empresa fundada en 2018 e inicialmente concebida en 2017, la idea que la distingue de otras empresas del sector de transportes con camiones es el sistema de relevos que implementan gracias al uso de *BigData* e inteligencia artificial. El oficio de camionero ofrece unas muy malas condiciones laborales, obligando a realizar muchas horas de trabajo y pasar numerosas noches lejos del hogar propio, esto hace que no sea un oficio deseado por la nuevas generaciones creando la ausencia de un relevo generacional. Se estima que en España faltan 15.000 conductores y en toda Europa cerca de 200.000 [1].

El sistema de relevos de Trucksters permite mejorar estas condiciones haciendo que dos conductores intercambien sus cargas a mitad de camino y den media vuelta, además reduce el número de horas que pasa un conductor en la carretera y por tanto el riesgo de accidente y permite una más sencilla implementación de modelos de vehículos eléctricos que presentan problemas en largas rutas tradicionales debido a su baja autonomía. La empresa española de transportes media cuenta con entre 2 y 3 camioneros, por lo que por si solas no podrían implementar un complejo sistema de relevos como el que Trucksters propone. Tan solo 5 años después en 2022 cerró una ronda de financiación de 8 millones de euros y espera triplicar el tamaño de su departamento de tecnologías [2]. Este rápido crecimiento ha impedido el debido ajuste de sus sistemas, pues el número de conductores registrados en la base de datos ha alcanzado un punto donde el sistema actual de almacenamiento no resulta eficiente.

Cada conductor tiene acceso a una aplicación proporcionada por Trucksters donde deben realizar varias tareas: señalar en que punto de la ruta se encuentran, a que hora realizaron el intercambio de cargas,... Al hacer esto deben de introducir su nombre o el de la empresa para la que operan, una forma de identificarlos, este proceso no estuvo lo suficientemente controlado por lo que nos encontramos ahora en un punto donde distintos nombres hacen referencia a una misma persona. En este trabajo se pretende poner remedio a este problema mediante el uso de distintos algoritmos y distancias entre cadenas de caracteres. Aunque su uso principal sería en la estandarización de nombres de usuario en las bases de datos de la empresa, otra aplicación que se contemplará será la sugerencia

de nombres ya existentes en la base cuando se intente introducir uno nuevo en la página web de la empresa. La idea es que mientras se escriba en la página web, esta sugiera los nombres ya existentes en el sistema más similares y si lo que se está introduciendo no se parece a nada de lo registrado no se muestre nada.

En el primer capítulo se presentaremos las distancias entre cadenas de caracteres que nos permitirán comparar entre si los distintos nombres de las bases de datos. En el Capítulo 2 concretaremos como calcular estás distancias de manera exacta mediante algoritmos, los cuales precisarán de cierto contexto teórico previo para entenderse, a continuación presentaremos el algoritmo que emplearemos para la clasificación mediante etiquetas de los nombres y se introducirá el concepto de la validación cruzada que tomará más importancia en el Capítulo 3. En este último capítulo el foco será la aplicación práctica, el uso de los algoritmos y distancias presentadas ante datos reales para realizar comparaciones sobre su efectividad y tiempos de ejecución.

# Capítulo 1

# Distancias entre palabras

En este capítulo presentaremos las distancias entre palabras que serán empleadas durante el trabajo para determinar si son suficientemente similares como para que consideremos que deben corresponder al mismo individuo. La primera en ser presentada será la distancia de Levenshtein, tras probar que es en efecto una distancia presentaremos la distancia de Damerau-Levenshtein. Finalmente nombraremos otras distancias que fueron consideradas como posibles candidatas pero se acabaron descartando, centrándonos en la distancia de Levenshtein normalizada.

## 1.1. La distancia de Levenshtein

Para ver si dos palabras son similares, introduciremos primero el concepto de distancia en general, para luego entrar a definir distancias específicas teniendo ya un entendimiento de que debe cumplir una función para ser denominada como tal.

**Definición 1.1.** Decimos que un conjunto $X$ es un espacio métrico si se puede definir en él una función $d : X \times X \to [0, \infty)$, conocida como distancia, tal que cumpla las siguientes propiedades:

a) $0 \leq d(x, y) < \infty$ para cualesquiera $x, y \in X$.

b) $d(x, y) = 0$ si y solamente si $x = y$.

c) Simetría: $d(x, y) = d(y, x)$ para todo $x, y \in X$.

d) Desigualdad triangular: $d(x, y) \leq d(x, z) + d(z, y)$ para todo $x, y, z \in X$.

En nuestro caso, definimos como espacio métrico $\mathcal{A}$ formado por todos los conjuntos ordenados de letras, del alfabeto inglés, a no ser que se especifique lo contrario, de una

longitud finita que a partir de aquí denominaremos cadenas. Hablaremos también sobre la longitud de una cadena, nos referiremos en este caso al número de letras que la forman, así la longitud de una cadena $a$ será $|a|$. Sobre este conjunto se define la distancia de Levenshtein [4].

**Definición 1.2.** Denominamos distancia de Levenshtein a la aplicación

$$d_L : \mathcal{A} \times \mathcal{A} \longrightarrow \mathbb{N} \cup \{0\} \subset [0, \infty),$$

tal que a cada par de $a, b \in \mathcal{A}$ le asigna el menor número de transformaciones necesarias para transformar la cadena $a$ en la $b$ de entre las siguientes:

- Eliminación: se quita una letra de la cadena reduciendo en 1 la longitud de la misma.
- Inserción: se añade una letra a la cadena en la posición que se quiera aumentando la longitud de esta en 1.
- Substitución: se cambia una letra de la cadena por otra cualquiera, en este caso la longitud de la cadena se mantiene.

*Observación* 1.3. La definición de la distancia de Levenshtein no se ve afectada por el alfabeto sobre el que se defina, este solo es relevante para los algoritmos que veremos en próximas secciones de este trabajo.

**Ejemplo 1.4.** Consideremos las cadenas de caracteres "ACLARO" y "CALVOS", como entre cualquier par de palabras existen infinitas secuencias de transformaciones que pasen de "ACLARO" a "CALVOS". En este caso concreto incluso existen varias con el número mínimo necesario de transformaciones que en este caso es 5. Una de ellas seria la siguiente:

1. Eliminación sobre el primer carácter: ACLARO→CLARO
2. Inserción de la letra 's' al final de la palabra, posición 6: CLARO→CLAROS.
3. Substitución de la letra 'r', posición 4, por la 'v': CLAROS→CLAVOS.
4. Substitución de la letra 'l', posición 2, por la 'a': CLAVOS→CAAVOS.
5. Substitución de la letra 'a', posición 3, por la 'l': CAAVOS→CALVOS.

**Proposición 1.5.** *La distancia de Levenshtein es una distancia.*

*Demostración.* (Esta demostración toma ideas del artículo [3])

Comprobemos que se cumplen las cuatro propiedades que definen una distancia (véase Definición 1.1)

a) Por como está definida se tiene que $d_L(a,b) \geq 0$ para cualesquiera $a, b \in \mathcal{A}$, solo es necesario por tanto ver que el número de transformaciones necesarias es finito. Supongamos sin pérdida de generalidad que $|a| = n < \infty$ y $|b| = m < \infty$, si $n = m$ empleando $m$ substituciones podremos convertir la cadena $a$ en $b$, por lo que el número mínimo de operaciones necesarias será menor o igual que $m$ y por tanto finito. Si $n > m$ empleando $n-m$ eliminaciones y $m$ substituciones podremos convertir $a$ en $b$ por lo que la distancia de Levenshtein vuelve a ser finita, el caso de $n < m$ es análogo con $m - n$ adiciones y $n$ substituciones como cota superior para la distancia.

b) Es evidente que $d_L(a,a) = 0$ para toda $a \in \mathcal{A}$ pues no es necesaria ninguna transformación y dado que no se puede realizar un número negativo de transformaciones este es el mínimo. De la misma forma si $d_L(a,b) = 0$ se tiene que $a = b$ pues no es necesaria ninguna transformación para convertir $a$ en $b$.

c) Para demostrar que se cumple la simetría tomemos dos cadenas de letras arbitrarias $a, b \in \mathcal{A}$ tales que el número mínimo de transformaciones necesarias para convertir $a$ en $b$ sea $K \in \mathbb{N}$, es decir $d_L(a,b) = K$. De entre estas transformaciones tenemos $K_s$ substituciones, $K_e$ eliminaciones y $K_i$ inserciones de forma que $K = K_s + K_i + K_e$.

   Es evidente que con $K$ transformaciones podemos pasar de $b$ a $a$, ya que bastaría con realizar $K_s$ substituciones cambiando la letra sustituida en el paso de $a$ a $b$ por la sustituta y viceversa, $K_e$ inserciones de las letras eliminadas y $K_i$ eliminaciones de las letras añadidas en el paso de $a$ a $b$. De esta forma se tiene que $d_L(b,a) \leq K$, para demostrar la igualdad basta con suponer que existe un natural $M < K$ tal que se puede pasar de $b$ a $a$ en $M$ transformaciones. Por los mismo argumentos antes empleados esto implicaría que podemos pasar de $a$ a $b$ en $M$ transformaciones lo cual es una contradicción con el hecho de que $K$ sea el mínimo. Por tanto queda probado que $d_L(b,a) = d_L(a,b)$.

d) Por último la demostración de la desigualdad triangular resulta trivial. Dadas tres cadenas $a, b, c \in \mathcal{A}$ se tiene que son necesarias como mínimo $d_L(a,c)$ transformaciones para pasar de $a$ a $c$ y $d_L(c,b)$ transformaciones para pasar de $c$ a $b$ como mínimo. Juntando ambos conjuntos de transformaciones se tiene una forma de pasar de $a$ a $b$ con $d_L(a,c) + d_L(c,b)$ transformaciones, como por definición $d_L(a,b)$ es el mínimo número de transformaciones necesarias para pasar de la cadena $a$ a la cadena $b$ se tiene que

$$d_L(a,b) \leq d_L(a,c) + d_L(c,b).$$

□

*Observación* 1.6. Dada una transformación $\sigma$ denotamos por $\sigma^{-1}$ a su transformación inversa, es decir, aquella tal que si $\sigma\left(\sigma^{-1}(a)\right) = a$ y $\sigma^{-1}\left(\sigma(a)\right) = a$. Es trivial que cada tipo transformación tiene su inversa asociada, comentadas en la demostración anterior en la parte referente a la simetría.

## 1.2. La distancia de Damerau-Levenshtein

Podemos añadir una transformación adicional a la distancia de Levenshtein de forma que las propiedades de las distancias se mantengan, creando así una nueva métrica.

**Definición 1.7.** Definimos la distancia de Damerau-Levenshtein [5] como la función

$$d_{DL}: \mathcal{A} \times \mathcal{A} \longrightarrow \mathbb{N} \cup \{0\} \subset [0, \infty),$$

que asigna a cada par de cadenas de caracteres $a, b \in \mathcal{A}$ el menor número de transformaciones necesarias para convertir $a$ en $b$. Siendo las transformaciones permitidas las mencionadas en la Definición 1.1 y una nueva a mayores:

- Transportación: Intercambio en la posición de dos caracteres contiguos de la cadena, manteniendo así la longitud de la cadena.

**Ejemplo 1.8.** Continuando con el par de palabras presentadas en el Ejemplo 1.4, la distancia de Damerau-Levenshtein entre ellas es de 4. Una de las secuencias de transformaciones que permiten pasar de "ACLARO" a "CALVOS" en solo 4 pasos es la siguiente:

1. Eliminación sobre el primer carácter: ACLARO→CLARO
2. Inserción de la letra 's' al final de la palabra, posición 6: CLARO→CLAROS.
3. Substitución de la letra 'r', posición 4, por la 'v': CLAROS→CLAVOS.
4. Transportación de la letra 'l', posición 2, con la 'a', posición 3: CLAVOS→CALVOS.

**Proposición 1.9.** *La distancia de Damerau-Levenshtein es una distancia.*

Omitimos la demostración de la Proposición 1.9 por ser análoga a la empleada en la prueba de la Proposición 1.5.

## 1.3. La distancia de Levenshtein normalizada

Este trabajo se centrará sobre la distancia de Levenshtein y la distancia de Damerau-Levenshtein, sin embargo existen otras distancias entre cadenas de caracteres que fueron consideradas pero no empleadas: la distancia de Hamming, la distancia de Jaro o la distancia de Jaro-Winkler[6]. Todas ellas descartadas por su limitado dominio, la distancia de Hamming solo está definida entre palabras de igual longitud, o por ser radicalmente diferentes a las distancias principales del trabajo, las distancia de Jaro y Jaro-Winkler se basan en la diferencia de posición entre caracteres idénticos de ambas cadenas y no en operaciones. La principal candidata fue la distancia de Levenshtein normalizada presentada por Li Yujian y Liu Bo en su trabajo de 2007 [7].

**Definición 1.10.** Definimos la distancia de Levenshtein normalizada como la función

$$\begin{aligned} d_{NL} : \mathcal{A} \times \mathcal{A} &\longrightarrow \mathbb{N} \cup \{0\} \subset [0, \infty) \\ (a, b) &\longrightarrow \frac{2d_L(a, b)}{|a| + |b| + d_L(a, b)}, \end{aligned}$$

para cualquier par de cadenas donde al menos una es no vacía y $d_{NL}(\emptyset, \emptyset) = 0$ en el caso de que ambas sean cadenas vacías.

El sobre nombre de normalizada indica que claramente $d_{NL}(a, b) \in [0, 1]$ para cualesquiera $a, b \in \mathcal{A}$.

El artículo [7] contiene la demostración de que $d_{NL}$ cumple las condiciones para ser una distancia, las tres primeras propiedades de la Definición 1.1 resultan triviales a partir del hecho de que $d_L$ es una distancia, la desigualdad triangular sin embargo, precisa de lemas previos y es más difícil de probar.

# Capítulo 2

# Algoritmos

En este capítulo presentaremos los algoritmos de Wagner-Fischer [4] para calcular la distancia de Levenshtein entre dos palabras y otro algoritmo que permita calcular la distancia de Damerau-Levenshtein [5]. Estos algoritmos se basan en el concepto de traza que se definirá a continuación, expandiremos sobre él definiendo el coste de una traza asociado a la distancia de Levenshtein y el coste asociado a la distancia de Damerau-Levenshtein para demostrar finalmente la equivalencia entre una traza y una secuencia de operaciones (inserción, eliminación, substitución, transportación). Con esto demostraremos una definición equivalente de las distancias expuestas en el Capítulo 1 basada en el coste de trazas asociado a dicha distancia, lo cual justificará el correcto funcionamiento del algoritmo correspondiente.

## 2.1. Algoritmo para la distancia de Levenshtein

Para poder comprender como funcionan este algoritmo que presentaremos son necesarios una serie de conceptos previos.

**Definición 2.1.** Dadas dos cadenas de letras $a, b \in \mathcal{A}$ definimos una traza de $a$ a $b$ como la terna $T = [U, a, b]$. Sea $X_c = \{1, 2, \ldots, |c|\}$ el conjunto $U$ es un subconjunto ordenado de $X_a \times X_b$ tal que dos pares distintos $(i_1, j_1), (i_2, j_2) \in U$ cumplen que $i_1 \neq i_2$ si y sólo si $j_1 \neq j_2$.

Interpretando cada traza $T$ como una serie de asignaciones entre elementos que conforman las cadenas $a$ y los de $b$, podemos colocar las cadenas en paralelo y unir por segmentos los caracteres asignados entre sí.

**Ejemplo 2.2.** Mostramos a continuación dos trazas distintas entre las mismas cadenas de letras "GIFT" y "FIT". La primera es $T_1 = [U_1 = \{(1,1), (2,2), (4,3)\}, \text{"GIFT"}, \text{"FIT"}]$ y la

segunda $T_2 = [U_2 = \{(1,1),(2,2),(3,3)\}$,"GIFT","FIT"]. Ambas cumplen al completo la Definición 2.1, pues las primeras coordenadas de los elementos de $U_1$ y $U_2$ forman parte de $\{1,2,3,4\} = X_{\text{“}GIFT\text{”}}$ y las segundas de $\{1,2,3\} = X_{\text{“}FIT\text{”}}$, además todos los pares verifican que sus dos elementos son distintos con respecto a los de cualquier otro par presenta en el mismo conjunto.

$$
\begin{array}{cccccc}
 & \text{G} & \text{I} & \text{F} & \text{T} \\
T_1 & | & | & & | \\
 & \text{F} & \text{I} & & \text{T}
\end{array}
$$

$$
\begin{array}{cccccc}
 & \text{G} & \text{I} & \text{F} & \text{T} \\
T_2 & | & | & | & \\
 & \text{F} & \text{I} & \text{T} &
\end{array}
$$

Sin embargo, no podemos formar una terna $T_3 = [U_3$,"GIFT","FIT"] en la que $U_3$ tenga cuatro pares. Si consideramos $\{(1,1),(2,2),(4,3)\} \subset U_3$ no podemos encontrar un $(i,j) \in X_{\text{“}GIFT\text{”}} \times X_{\text{“}FIT\text{”}}$ tal que $i \notin \{1,2,4\}$ y $j \notin \{1,2,3\}$, aunque $i$ pueda tomar el valor 3 no existe ninguno posible para $j$ sin que se incumpla la Definición 2.1. Esto se expresa visualmente a continuación.

$$
\begin{array}{cccccc}
 & \text{G} & \text{I} & \text{F} & \text{T} \\
T_3 & | & | & | & | \\
 & \text{F} & \text{I} & & \text{T}
\end{array}
$$

**Definición 2.3.** Denominaremos a los elementos de $U$ como líneas y diremos que un elemento $x$ perteneciente a $X_a$ o $X_b$ tiene una linea de $U$ adyacente si existe una linea $(u_1,u_2) \in U$ tal que $u_1 = x$ para $x \in X_a$ y $u_2 = x$ para $x \in X_b$.

**Definición 2.4.** Decimos que $u = (u_1,u_2), v = (v_1,v_2) \in U$ se cruzan cuando $u_1 < v_1$ si y sólo si $v_2 < u_2$. Cuando esto sucede decimos que $u$ y $v$ dan lugar a un cruce de líneas. Cuando una traza $T$ no contiene ningún cruce de líneas decimos que es reducida.

**Ejemplo 2.2**(continuación) Las trazas $T_1$ y $T_2$ no contienen ningún cruce de lineas, por lo que ambas son trazas reducidas.

**Definición 2.5.** Una línea $u = (u_1,u_2) \in U$ es balanceada si $a_{u_1} = b_{u_2}$, siendo estos el $u_1$-ésimo elemento de la cadena $a$ y el $u_2$-ésimo de $b$. En caso de que esto no se cumpla diremos que $u$ es desbalanceada.

**Ejemplo 2.2**(continuación) La traza $T_1$ contiene dos lineas balanceadas $(2,2),(4,3)$ ya que "GIFT"$_2$ = $I$ = "FIT"$_2$ y "GIFT"$_4$ = $T$ = "FIT"$_3$ y una desbalanceada $(1,1)$ pues "GIFT"$_1$ = $G \neq F$ = "FIT"$_1$. Por su parte $T_2$ tiene la línea $(2,2)$ como la única balanceada de las tres que forman la traza.

A cada traza reducida $T = [U, a, b]$ podemos asignarle un coste que guarda relación con el paso de $a$ a $b$ a través de las transformaciones permitidas en el cálculo de la distancia de Levenshtein.

**Definición 2.6.** Dada la traza reducida $T = [U, a, b]$ su coste asociado a la distancia de Levenshtein vendrá dado como

$$c_L(T) = K_s + K_e + K_i,$$

siendo $K_s$ el número de líneas desbalanceadas de $U$, $K_e$ el número de elementos de $X_a$ no adyacentes a una línea de $U$ y $K_i$ el número de elementos de $X_b$ no adyacentes a una línea de $U$.

**Ejemplo 2.2**(continuación) Ya hemos establecido que las trazas $T_1$ y $T_2$ son reducidas por lo que podemos calcular sus costes asociados a la distancia de Levenshtein:

$$c_L(T_1) = K_s^1 + K_e^1 + K_i^1 = 1 + 1 + 0 = 2, \qquad c_L(T_2) = K_s^2 + K_e^2 + K_i^2 = 2 + 1 + 0 = 3.$$

Tal y como cabe esperar por la notación empleada, vamos a ver que cada traza reducida $T = [U, a, b]$ puede asociarse a una secuencia de las transformaciones que definen la distancia de Levenshtein (inserciones, eliminaciones y substituciones) y viceversa. Denotando dicha secuencia como $S = \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1$, donde las transformación $\sigma_1$ se aplica sobre la cadena original $c$, $\sigma_2$ sobre la cadena $\sigma_1(c)$ y así sucesivamente.

**Teorema 2.7.**

1. *Para cada traza reducida $T$ de $a$ a $b$ existe una secuencia de transformaciones $S_T$ de forma que $c_L(T) = ||S_T||$. Siendo que $||S_T||$ representa el número de transformaciones en la secuencia $S_T$.*

2. *Para una secuencia de transformaciones $S$ de $a$ a $b$ cualquiera existe una traza reducida $T_S$ de $a$ a $b$ tal que $c_L(T_S) \leq ||S||$.*

*Demostración.* (En esta demostración hemos seguido el esquema proporcionado por Wagner y Lowrance[5])

1. Basta con observar que cada línea desbalanceada $v = (v_1, v_2)$ se corresponde con una substitución del carácter $a_{v_1}$ por $b_{v_2}$, cada elemento de $b$ no adyacente a una línea debe ser insertado y cada elemento de $a$ en la misma situación debe ser eliminado.

2. Sea la secuencia de transformaciones (inserciones, eliminaciones y substituciones) $S = \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1$ de $a$ a $b$ y una matriz

$$M = \begin{pmatrix} a_1 & \dots & a_{|a|} \\ 1 & \dots & |a| \end{pmatrix} \in M_{2\times|a|}.$$

   Paralelamente a las distintas transformaciones de $S$ se realizan sobre $M$ los siguientes cambios:

   - Por cada eliminación se elimina la correspondiente columna de $M$.
   - Por cada inserción en una posición $i$ se añade una columna con el carácter insertado en la parte superior y un 0 en la inferior, de forma que esta es la nueva $i$-ésima columna.
   - Por cada substitución de un elemento $e_1$ por otro $e_2$ basta con hacer dicha substitución también en la matriz.

   Tras efectuar los cambios pertinentes sobre $M$ la primera fila de la misma será un conjunto ordenado de caracteres $c_1, \dots, c_n$ y la segunda sus valores asociados $k_1, \dots, k_n$. Así definimos la traza $T_S = [U, a, b]$ como

$$U = \{(k_j, j)/\ 1 \leq j \leq n,\ k_j \neq 0\}.$$

   Por construcción, $T_S$ es una traza reducida donde los $a_i$ sin una línea asignada han sido eliminados, los $b_j$ en la misma situación han sido añadidos y los $u \in U$ tales que $a_{u_1} \neq b_{u_2}$ se deben a una, o más, operación de substitución. Por tanto cada uno de los elementos de $T_S$ aporta un coste menor o igual a su correspondiente transformación de $S$ y se concluye que $c_L(T_S) \leq ||S||$.

□

Con este teorema podemos dar una definición alternativa de la distancia de Levenshtein, pues para la secuencia de transformaciones $S$ de $a$ a $b$ de menor tamaño existe una traza reducida de coste menor o igual, la cual tiene asociada una secuencia de transformaciones de igual longitud a su coste. Por ser $S$ un mínimo en este caso la traza $T_S$ debe cumplir que $c_L(T_S) = ||S||$.

**Corolario 2.8.** *Dadas dos cadenas de caracteres $a$ y $b$ su distancia de Levenshtein puede calcularse como*

$$d_L(a,b) = \min\{c_L(T)/\ T = [U,a,b] \text{ es una traza reducida}\}.$$

Presentaremos ahora un teorema que nos permitirá justificar el correcto funcionamiento del conocido como algoritmo de Wagner-Fischer [4] para el cálculo de la distancia de Levenshtein entre dos cadenas de caracteres.

**Teorema 2.9.** *Sea $a^i$ la cadena formada por los $i$ primeros caracteres de $a$ y $b^j$ la de los primeros $j$ caracteres de $b$. Se puede calcular la distancia de Levenshtein entre estas, con $i,j > 0$ como:*

$$d_L(a^i,b^j) = \min\left\{d_L(a^{i-1},b^j)+1, d_L(a^i,b^{j-1})+1, d_L(a^{i-1},b^{j-1}) + \mathbb{I}_{a_i \neq b_j}\right\},$$

*donde $\mathbb{I}_{a_i \neq b_j}$ toma valor 1 cuando $a_i \neq b_j$ y 0 en otro caso. Por notación consideramos que para cualquier cadena $c$ de caracteres, $c^0 = \emptyset$.*

*Demostración.* (La idea de esta demostración aparece recogida en el trabajo de Wagner y Lowrance[5]) Por como hemos definido $a^0$ y $b^0$ se tiene, trivialmente, que

$$d_L(a^0,b^j) = j\ \forall j \in \{0,1,\ldots,|b|\} \text{ y } d_L(a^i,b^0) = i\ \forall i \in \{0,1,\ldots,|a|\}.$$

Una vez aclarado esto los casos con $i,j > 0$ están bien definidos. Sea $T = [U,a^i,b^j]$ la traza reducida de mínimo coste de $a^i$ a $b^j$, la cual existe por el Corolario 2.8 y su coste coincide con la distancia buscada. Existen tres escenarios posibles respecto a $a_i$ y $b_j$ los últimos caracteres de las cadenas $a^i$ y $b^j$:

- Ninguna línea de $T$ es adyacente a $a_i$, por lo que este debe ser eliminado para transformar $a^i$ en $b^j$, dado que hemos realizado una transformación se tiene que

$$d_L(a^i,b^j) = d_L(a^{i-1},b^j) + 1.$$

- Ninguna línea de $T$ es adyacente a $b_j$, por lo que este debe ser insertado para transformar $a^i$ en $b^j$, dado que hemos realizado una transformación se tiene que

$$d_L(a^i,b^j) = d_L(a^i,b^{j-1}) + 1.$$

- Tanto $a_i$ como $b_j$ son adyacentes a una línea, por ser $T$ reducida se tiene que ambas líneas deben ser la misma o $T$ contendría un cruce de líneas al ser ambos el último elemento de su cadena. Si $a_i = b_j$ se tiene que $d_L(a^i, b^j) = d_L(a^{i-1}, b^{j-1})$, en caso contrario debe de ocurrir una substitución de $a_i$ por $b_j$, de forma que $d_L(a^i, b^j) = d_L(a^{i-1}, b^{j-1}) + 1$, es decir,

$$d_L(a^i, b^j) = d_L(a^{i-1}, b^{j-1}) + \mathbb{I}_{a_i \neq b_j}.$$

Como uno de los tres escenarios anteriores debe cumplirse se tiene que el mínimo de las tres posibilidades es la distancia de Levenshtein entre $a^i$ y $b^j$. □

*Observación* 2.10. Tal y como hemos enunciado en la demostración cada uno de los tres valores entre los que elegimos el mínimo representa una transformación. Si el mínimo es $d_L(a^{i-1}, b^j) + 1$ se elimina $a_i$, si es $d_L(a^i, b^{j-1}) + 1$ se inserta $b_j$ en la posición $j$ y si es $d_L(a^{i-1}, b^{j-1}) + \mathbb{I}_{a_i \neq b_j}$ se realiza la substitución de $a_i$ por $b_j$ o no se realiza transformación alguna en caso de que $a_i = b_j$.

El siguiente pseudo-código recoge el algoritmo de Wagner-Fischer[4].

```
lev_dist(a,b):
        n = length(a)
        m = length(b)
        d = array(0:n,0:m)
        for i in 0:n :
                d[i,0] = i
        for j in 0:m :
                d[0,j] = j
        for i in 1:n :
                for j in 1:m :
                        ind = 1*(a[i-1] != b[j-1])
                        d[i,j] = min(d[i-1,j] + 1, d[i,j-1] + 1,
                                                   d[i-1,j-1] + ind)
        return(d[n,m])
```

Al introducir dos cadenas de caracteres $a$ y $b$ el algoritmo nos devuelve un número que se corresponderá con la distancia de Levenshtein entre las cadenas. Gracias al Teorema 2.9 sabemos que podemos obtener esta distancia de forma constructiva a través del cálculo de las $d_L(a^i, b^j)$ con $0 \leq i < |a|$ y $0 \leq j < |b|$. El array $d$ presentado en el pseudo-código

puede pensarse como una matriz con tantas filas como caracteres formen $a$ más uno y tantas columnas como caracteres tenga $b$ más uno, indexada de empezando en el 0. Al final del algoritmo el elemento $d_{ij}$ contendrá la distancia entre la cadena $a$ restringida hasta el $i$-ésimo carácter y la $b$ hasta el $j$-ésimo. Su construcción comienza calculando las distancias con el conjunto vacío, triviales, que se corresponden con la fila y columna de índice 0, a continuación podemos calcular el valor de $d_{11} = d_L(a^1, b^1)$ según la fórmula recogida en el Teorema 2.9. Se continua completando la fila de izquierda a derecha y cuando se termina se pasa al elemento de la siguiente fila y la columna de índice 1 para completar esa fila. El algoritmo acaba cuando se calcula el último elemento de la matriz, $d_{|a||b|}$ que contiene el valor $d_L(a^{|a|}, b^{|b|}) = d_L(a, b)$.

**Ejemplo 2.11.** Aplicando el algoritmo de Wagner-Fischer sobre las palabras $a =$"GIFT" y $b =$"FIT" del Ejemplo 2.2 obtenemos la matriz

$$d = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 1 & 2 & 3 \\ 2 & 2 & 1 & 2 \\ 3 & 2 & 2 & 2 \\ 4 & 3 & 3 & 2 \end{pmatrix}.$$

El primer elemento de cada fila y columna contiene su índice según el pseudo-código presentado, uno menos que el índice habitual de una matriz. A continuación calculamos el único elemento que tiene valores ya calculados a su izquierda, encima y en la diagonal en esa dirección como $d_{11} = d_L(\text{“}G''\text{, “}F'') = \min\{1+1, 1+1, 0+\mathbb{I}_{'G'\neq'F'}\} = \mathbb{I}_{'G'\neq'F'} = 1$. A partir de aquí se podría continuar rellenando columna a columna o fila a fila, nosotros decidimos hacerlo por filas, es decir, en el sentido de lectura habitual.

Como “GIFT” tiene 4 caracteres y “FIT” tiene 3 la distancia de Levenshtein coincidirá con el último elemento de la matriz d, $d_{4+1,3+1}$. Por tanto la distancia de Levenshtein entre las dos palabras es de 2 lo cual coincide con el coste asociado a la distancia de Levenshtein de la traza $T_1$.

Una vez demostrado el correcto funcionamiento del algoritmo, pasemos a hablar de su tiempo de ejecución. El algoritmo realiza $1 + |a| + |b|$ cálculos al rellenar la primera fila y columna de la matriz $d$, a continuación en cada uno de los $|a| \cdot |b|$ bucles realiza 5 operaciones, incluyendo el cálculo del mínimo y la comprobación de si $a_i$ es igual a $b_j$, así que el total de operaciones es:

$$1 + |a| + |b| + 5 \cdot |a| \cdot |b|.$$

Por tanto el tiempo total de ejecución del algoritmo debería ser $\mathcal{O}(|a| \cdot |b|)$.

## 2.2. Algoritmo para la distancia de Damerau-Levenshtein

Si empleamos la distancia de Damerau-Levenshtein en lugar de la de Levenshtein podemos ampliar el concepto de coste de una traza a aquellas que no son reducidas.

**Definición 2.12.** Dada la traza $T = [U, a, b]$, sea o no reducida, su coste asociado a la distancia de Damerau-Levenshtein vendrá dado como

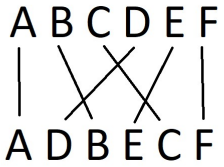
$$c_{DL}(T) = K_s + K_e + K_i + K_t,$$

siendo $K_s$ el número de líneas desbalanceadas de $U$, $K_e$ el número de elementos de $X_a$ no adyacentes a una línea de $U$, $K_i$ el número de elementos de $X_b$ no adyacentes a una línea de $U$ y $K_t$ es el número de cruces de líneas de $U$.

*Observación* 2.13. En caso de una traza reducida $T$ se tiene que $c_{DL}(T) = c_L(T)$ ya que el número de cruces de línea, $K_t$, es 0.

**Ejemplo 2.14.** La siguiente traza, $T_4$, es no reducida pues contiene 3 cruces de lineas, su coste asociado a la distancia de Damerau-Levenshtein es

$$c_{DL}(T_4) = K_s^4 + K_e^4 + K_i^4 + K_t^4 = 0 + 0 + 0 + 3 = 3.$$




Pasemos a ver los resultados equivalentes al Teorema 2.7 y al Corolario 2.8 para la distancia de Damerau-Levenshtein y las trazas generales, pero antes necesitamos de los siguientes resultados previos.

**Lema 2.15.** *Sea $T = [U, a, b]$ una traza de $a$ a $b$. Si $u, v \in U$ se cruzan de forma que $|u_1 - v_1| = 1$, entonces existe una traza $T' = [U', a', b]$ tal que $c_{DL}(T') = c_{DL}(T) - 1$ y $a'$ es la cadena resultante de aplicar sobre $a$ la transportación entre $a_{u_1}$ y $a_{v_1}$.*

*Demostración.* (Esta demostración aparece en el artículo de Wagner y Lowrance[5])
Sea $U' = U \backslash \{(u, v)\} \cup \{(u_1, v_2), (v_1, u_2)\}$, de esta forma $U'$ sustituye dos líneas que se cruzan por dos que no lo hacen y el cambio de $a$ por $a'$ nos garantiza que las líneas desbalanceadas

se mantienen así y las que no lo son también, de esta forma es evidente que solo el número de cruces de líneas $K_t$ se ve reducido en uno y los otros involucrados en el cálculo del coste no son alterados, por lo que

$$c_{DL}(T') = c_{DL}(T) - 1.$$

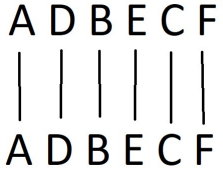
□

**Lema 2.16.** *Sea una traza $T = [U, a, b]$ tal que para cada $x \in X_a$ existe un $(x, y) \in U$ y $T$ contiene $K_t$ cruces de líneas. Existe entonces una secuencia de transformaciones $S$ y una traza $T' = [U', S(a), b]$ tales que $||S|| = K_t$, $c_{DL}(T') = c_{DL}(T) - ||S||$ siendo $T'$ una traza reducida y $S(a)$ la cadena resultante de aplicar la secuencia de transformaciones $S$ sobre la cadena $a$.*

*Demostración.* (Esta demostración aparece en el artículo de Wagner y Lowrance[5])
Demostremos el caso $K_t > 0$, pues si $K_t = 0$ el resultado es trivial tomando $T = T'$ y $S = \emptyset$. Dado que todo punto de $X_a$, análogamente todo elemento de $a$, tiene una línea adyacente deben existir $u, v \in U$ tales que $|u_1 - v_1| = 1$ por lo que podemos aplicar el Lema 2.15 y construir una traza $T_1$. Esta $T_1$ es una traza de $\sigma_1(a)$ en $b$ con $c_{DL}(T_1) = c_{DL}(T) - 1$, donde $\sigma_1$ es la transportación entre los elementos de las posiciones $u_1$ y $v_1$. La traza $T_1$ contiene un único cruce de líneas menos que $T$, repitiendo este proceso podemos construir una sucesión de trazas $T_0, T_1, T_2, \ldots, T_{K_t}$ de forma que esta última es una traza reducida de $\sigma_{K_t} \circ \cdots \circ \sigma_2 \circ \sigma_1(a)$ a $b$ con $c_{DL}(T_{K_t}) = c_{DL}(T) - K_t$, basta con tomar $T' = T_{K_t}$ y $S = \sigma_{K_t} \circ \cdots \circ \sigma_2 \circ \sigma_1$ para terminar la demostración. □

**Ejemplo 2.14**(continuación)Aplicando el Lema 2.16 en el caso de la traza $T_4$ se obtendrá la siguiente traza, $T_4'$, que sería la traza reducida resultante de aplicar sobre "ABCDEF" tres transformaciones:

1. $\sigma_1$ ="Transportación entre C y D"
2. $\sigma_2$ ="Transportación entre B y D"
3. $\sigma_3$ ="Transportación entre C y E".



Es evidente que al coincidir $S(a)$ y $b$ el coste de la traza $T_4'$ será nulo, aún así podemos comprobarlo a través del lema

$$c_{DL}(T_4') = c_{DL}(T_4) - 3 = 3 - 3 = 0.$$

Ahora sí podemos emplear estos resultados para probar el siguiente teorema que puede considerarse una generalización del Teorema 2.7 para trazas generales y la distancia de Damerau-Levenshtein.

**Teorema 2.17.**

1. *Para cada traza $T$ de $a$ a $b$ existe una secuencia de transformaciones $S_T$ de forma que $c_{DL}(T) = ||S_T||$. Siendo que $||S_T||$ representa el número de transformaciones en la secuencia $S_T$.*

2. *Para una secuencia de transformaciones $S$ de $a$ a $b$ cualquiera existe una traza $T_S$ de $a$ a $b$ tal que $c_{DL}(T_S) \leq ||S||$.*

*Demostración.* (Esta demostración sigue el esquema de la presentada por Wagner y Lowrance[5])

1. Denominemos por $K_e$ al número de caracteres de $a$ no adyacentes a una de las líneas que forman $T$, por cada uno de estos casos añadimos una transformación que elimina dicho elemento formando una secuencia de transformaciones $S_e$. Resulta así la traza $T' = [U', S_e(a), b]$ donde todos los elementos de la cadena $S_e(a)$ son adyacentes a una línea y $c_{DL}(T') = c_{DL}(T) - K_e$.

   Sea $K_t$ el número de cruces de línea en $T$, y por construcción también en $T'$, aplicando el Lema 2.16 obtenemos una traza reducida $T'' = [U'', S_t(S_e(a)), b]$ donde $S_t$ es una secuencia de $K_t$ transposiciones y $c_{DL}(T'') = c_{DL}(T') - K_t = c_{DL}(T) - K_e - K_t$.

   Por último aplicando el Apartado 1 del Teorema 2.7 sobre la traza reducida $T''$ podemos afirmar la existencia de una secuencia de transformaciones $S$ tal que $c_{DL}(T'') = c_L(T'') = ||S||$.

   La secuencia de transformaciones $S_T$ buscada es $S_T = S \circ S_t \circ S_e$ y despejando en la igualdades antes establecidas se tiene que

$$c_{DL}(T) = ||S|| + ||S_e|| + ||S_t|| = ||S_T||.$$

2. La demostración de este segundo apartado es análoga a la recogida en la prueba del Teorema 2.7. Basta con considerar a mayores que $S$ contenga transposiciones y en la matriz $M$ por cada transposición entre los elementos $a_i$ por $a_j$ intercambiamos el

sitio de sus columnas correspondientes en la matriz. Una vez obtenida la traza $T_S$, no necesariamente reducida, también se debe de tener en cuenta que cada cruce de líneas proviene de, al menos, una transposición.

□

Antes de comentar el código del algoritmo para calcular la distancia de Damerau-Levenshtein entre dos cadenas de caracteres necesitamos introducir una definición más.

**Definición 2.18.** Se dice que una traza $T = [U, a, b]$ de $a$ a $b$ es restringida si cumple las siguientes propiedades:

I Toda línea de $U$ se cruza como máxima con otra.

II Toda línea de $U$ que que forma un cruce de líneas es balanceada.

III Dadas dos líneas $u, v \in U$ que se crucen entre si con $u_1 < v_1$, no existe $i \in \mathbb{N}$ tal que $i \in (u_1, v_1)$ y $a_{u_1} = a_i$ ni $j \in \mathbb{N}$ tal que $j \in (v_2, u_2)$ y $b_{v_2} = b_j$.

**Teorema 2.19.** *Dadas dos cadenas de caracteres $a, b \in \mathcal{A}$ existe al menos una traza de coste mínimo $T$ tal que es restringida.*

*Demostración.* (Esta demostración basa su estructura en el artículo de Wagner y Lowrance[5]) La demostración se hará en tres partes, en la primera se demostrará que existe una traza de coste mínimo que cumple la Propiedad $I$ de las trazas restringidas, en la segunda que existe traza de coste mínimo cumpliendo las Propiedades $I$ y $II$ y en la tercera se completará la demostración.

Supongamos que existe $T = [U, a, b]$ una traza de coste mínimo de $a$ a $b$ tal que existe una única $u \in U$ que se cruza con $n \geq 2$ líneas. Se puede construir entonces una traza $T' = [U', a, b]$ donde $U' = U \backslash \{u\}$, de forma que

$$c_{DL}(T') = c_{DL}(T) - n + 1 + 1.$$

Dado que $n \geq 2$, se tiene que $c_{DL}(T') \leq c_{DL}(T)$ y por construcción todas las líneas de $U'$ se cruzan como mucho con otra, para el caso con más líneas con dos o más cruces la demostración se sigue por inducción. Por tanto hemos probado que existe una traza de coste mínimo que cumple la Propiedad $I$ de las trazas restringidas.

Sea $T = [U, a, b]$ una traza de coste mínimo de $a$ a $b$, como ya hemos visto podemos suponer que ninguna línea de $U$ cruza a más que otra línea, sean $v, w \in U$ líneas que se cruzan y supongamos que $v$ es desbalanceada. Definimos $\tilde{T} = [\tilde{U}, a, b]$ y $T'' = [U'', a, b]$ como las trazas de $a$ a $b$ con $U'' = U \backslash \{v, w\}$ y $\tilde{U} = U'' \cup \{(v_1, w_2), (w_1, v_2)\}$, nos encontramos ante la siguiente dicotomía:

- $w$ es balanceada, en este caso se cumplen las siguientes igualdades:

$$c_{DL}(T) = c_{DL}(T'') - 2 - 2 + 1 + 1 \text{ y } c_{DL}(\tilde{T}) = c_{DL}(T'') + 2 - 2 - 2.$$

  Combinando ambas expresiones obtenemos que $c_{DL}(\tilde{T}) < c_{DL}(T)$, de forma que no puede existir una traza de coste mínimo $T$ con una línea desbalanceada que se cruce con una balanceada.

- $w$ es desbalanceada, de ser así basta con ver que,

$$c_{DL}(T'') = c_{DL}(T) - 2 - 1 + 1 + 1 \Leftrightarrow c_{DL}(T'') < c_{DL}(T).$$

  Por tanto, no existe una traza de coste mínimo con una línea que se cruce siendo desbalanceada, ya que esto contradice dicha hipótesis.

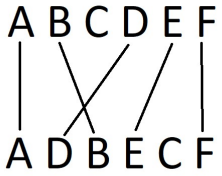
En el caso de varias líneas desbalanceadas que se crucen basta con completar la demostración por inducción.

Por último, consideremos $T = [U, a, b]$ una traza de coste mínimo de $a$ a $b$, donde toda línea de $U$ se cruce como máximo con otra y de hacerlo ambas sean balanceadas, siendo dos de estas líneas que se cruza $u, v \in U$ con $u_1 < v_1$. Supongamos que existe un cierto $i \in \mathbb{N}$ tal que $u_1 < i < v_1$ y $a_{u_1} = a_i$, esto último junto con que $u$ es una línea balanceada significa que la línea $(i, u_2)$ también es balanceada. Definimos $\overline{T} = [\overline{U}, a, b]$ como la traza de $a$ a $b$ con $\overline{U} = (U \backslash \{u\}) \cup \{(i, u_2)\}$, en esta nueva traza hemos sustituido una línea balanceada por otra y el numero de cruces de líneas no se altera, ya que si $u$ solo se cruzaba con $v$ no puede existir un $w \in U$ tal que $u_1 < w_1 < v_1$, por tanto se tiene que $c_{DL}(\overline{T}) = c_{DL}(T)$. El argumento para $j$ es análogo y por inducción podemos generar nuevas trazas de coste mínimo que sean restringidas. □

Este teorema nos permite tan solo considerar las trazas restringidas cuando buscamos la traza coste mínimo entre dos cadenas de caracteres.

**Corolario 2.20.** *La traza de coste mínimo entre dos cadenas de caracteres $a$ y $b$ tiene el mismo coste que la traza restringida de mínimo coste.*

**Ejemplo 2.14**(continuación) Por el Teorema 2.19 debe existir al menos una traza restringida de coste mínimo entre las cadenas "ABCDEF" y "ABDECF", la siguiente traza lo es:




Denotaremos a esta traza por $\tilde{T}_4$ y podemos calcular su coste asociado a la distancia de Damerau-Levenshtein como

$$c_{DL}(\tilde{T}_4) = \tilde{K}_s^4 + \tilde{K}_e^4 + \tilde{K}_i^4 + \tilde{K}_t^4 = 0 + 1 + 1 + 1 = 3.$$

En próximos ejemplos veremos que efectivamente 3 es el coste asociado a la distancia de Damerau-Levenshtein mínimo de cualquier traza entre estas dos palabras.

*Observación* 2.21. Durante todo este trabajo consideramos iguales en cuanto a su peso, todos con peso 1, en las distancias todos los tipos de transformaciones (inclusión, eliminación, substitución y transportación). A pesar de esto, los resultados y definiciones empleados pueden reformularse dando distintos pesos a las distintas transformaciones de cadenas de caracteres. Esto puede resultar de interés cuando nos interesa priorizar un tipo de transformación, o error en el caso de la aplicación práctica de este trabajo, sobre otro u otros. La mayor restricción se presenta en el Teorema 2.19, pues es solo cierto si dos veces el peso asociado a la substitución es mayor o igual a la suma del peso asociado a la eliminación y el de la inserción.

Presentamos el algoritmo para el cálculo de la distancia de Damerau-Levenshtein entre dos cadenas de caracteres basado en los resultados de Wagner y Lowrance [5]. Similar al caso del algoritmo de Wagner-Fischer[4], este emplea el Corolario 2.20 y el Teorema 2.17, pues lo que realmente hace es calcular el coste de la traza restringida de mínimo coste. El siguiente teorema justifica el correcto cálculo de dicho coste.

**Teorema 2.22.** *Sea $a^i$ la cadena formada por los $i$ primeros caracteres de $a$ y $b^j$ la de los primeros $j$ caracteres de $b$. Se puede calcular el mínimo coste de una traza restringida, la cual denotaremos como $T[i,j]$ para simplificar la notación,entre estas cadenas como*

$$c_{DL}(T[i,j]) = \text{mín}\{c_{DL}(T[i-1,j]) + 1, c_{DL}(T[i,j-1]) + 1, c_{DL}(T[i-1,j-1]) + \mathbb{I}_{a_i \neq b_j} + c_{DL}(T[k-1,t-1]) + (i-k-1) + 1 + (j-t-1)\},$$

*para todas $i,j > 0$, donde $\mathbb{I}_{a_i \neq b_j}$ toma valor 1 cuando $a_i \neq b_j$ y 0 en otro caso. Siendo $k,t$ los mayores números de los conjuntos $\{0,1,\ldots,i\}$ y $\{0,1,\ldots,j\}$, respectivamente, tales que $a_k = b_j$ y $a_i = b_t$.*

*Los casos con índice nulo volvemos a considerar que la cadena es el conjunto vacío y si* $i = -1$ *o* $j = -1$ *se toma* $c_{DL}(T[i,j]) = M$ *siendo este un valor superior o igual al coste de la traza vacía, restringida y reducida por definición, para evitar que el algoritmo de como resultado un valor mayor o igual a este, es decir,* $M \geq c_L([\emptyset, a, b]) = |a| + |b| = d_L(c^0, a) + d_L(c^0, b)$.

*Demostración.* (Esta demostración está sacada del artículo [5]) Sea $T = [U, a^i, b^j]$ la traza restringida de coste mínimo, existen cuatro casos posibles:

- Ninguna línea de $U$ es adyacente a $a_i$, en tal caso $c_{DL}(T) = c_{DL}(T') + 1$ siendo $T'$ la traza restringida de coste mínimo entre $a^{i-1}$ y $b^j$, pues solo es necesario hacer una eliminación más respecto a esta.

- Ninguna línea de $U$ es adyacente a $b_j$, en tal caso $c_{DL}(T) = c_{DL}(T') + 1$ siendo $T'$ la traza restringida de coste mínimo entre $a^i$ y $b^{j-1}$, pues solo es necesario hacer una inserción más respecto a esta.

- Una misma línea de $U$ es adyacente a $a_i$ y $b_j$, en tal caso $c_{DL}(T) = c_{DL}(T') + \mathbb{I}_{a_i \neq b_j}$ siendo $T'$ la traza restringida de coste mínimo entre $a^{i-1}$ y $b^{j-1}$, pues solo debemos aumentar su coste en uno si dicha línea es desbalanceada y mantenerlo igual si es balanceada.

- Existen dos líneas $(i,t), (k,j) \in U$ distintas, adyacentes una a $a_i$ y otra a $b_j$, estas líneas se cruzan y por la Propiedad $I$ de las trazas restringidas no cruzan a ninguna otra. Por la Propiedad $II$ de las trazas restringidas se tiene que $a_i = b_t$ y $a_k = b_j$ y por la Propiedad $III$ se tiene que deben corresponderse con los $k, t$ del enunciado del teorema. Entonces tenemos que

  $$c_{DL}(T) = c_{DL}(T') + (i - k - 1) + 1 + (j - t - 1),$$

  siendo $T'$ la traza restringida de coste mínimo entre $a^{k-1}$ y $b^{t-1}$, ya que a esta le añadimos un 1 por el cruce de lineas balanceadas y sumamos otra unidad por cada letra entre $a_k$ y $a_i$ y entre $b_t$ y $b_j$ ya que para cumplir la Propiedad I de las trazas reducidas ninguno de estos elementos tendrá una línea adyacente.

□

A continuación presentamos el pseudo-código del algoritmo para el cálculo de la distancia de Damerau-Levenshtein entre dos cadenas de caracteres [5]. En él empleamos la notación '.index(x)' después de un vector o lista para indicar la función que nos devuelve el índice correspondiente al elemento de dicho vector que coincide con x, por ejemplo si $v = (1, y, 32)$ se tendrá que $v$.index(y)=2.

```
dam_lev_dist(a,b,alfabeto):
        n = length(a)
        m = length(b)
        alflen = length(alfabeto)
        da = repeat(0,length=alflen)
        d = array(-1:n,-1:m)
        max_dist = n+m
        for i in -1:n :
                d[i,-1] = max_dist
        for i in 0:n :
                d[i,0] = i
        for j in -1:m :
                d[-1,j] = max_dist
        for j in 0:m :
                d[0,j] = j
        for i in 1:n :
                db = 0
                for j in 1:m :
                        bj = alfabeto.index(b[j])
                        k = da[bj]
                        t = db
                        if (a[i] == b[j]):
                                ind = 0
                                db = j
                        else:
                                ind = 1
                        d[i,j] = min(d[i-1,j] + 1, d[i,j-1] + 1,
                                        d[i-1,j-1] + ind,
                                        d[k-1,t-1] + (i-k-1) + (j-t-1) + 1)
                ai = alfabeto.index(a[i])
                da[ai] = i
        return(d[n,m])
```

Para comprender este algoritmo debemos de pensar en $d$ como una matriz con dos filas más que caracteres formen $a$ y dos columnas más que caracteres tenga $b$, indexadas empezando en el -1. En la fila y columna -1 todos los elementos deben ser mayores o iguales, en este caso los definimos iguales, a $d_{DL}(\emptyset, a) + d_{DL}(\emptyset, b)$, es decir la suma de longitudes

de las cadenas, para evitar obtener valores que excedan este límite superior deducido a partir de la desigualdad triangular. La fila y columna con índice 0 se calculan igual que en el algoritmo de Wagner-Fischer.

Los valores denominados $k, t$ en el algoritmo solo son necesarios para el cálculo de cada distancia, por lo que no es necesario almacenarlos. Se recogen en $da$ y $db$ y son re-calculados en cada iteración de sus bucles correspondientes. Cabe destacar también la presencia del input $alfabeto$, una lista con todos los caracteres existentes en a partir de los cuales se han creado $a$ y $b$. El $t$-ésimo elemento del vector $da$ contiene el mayor índice, $s$, menor que el $i$ correspondiente al índice de la fila actual tal que $a_s$ es el $t$-ésimo carácter del abecedario. Por su parte $db$ es igual al mayor índice, $h$, menor que el $j$ correspondiente a la columna de la iteración actual tal que $b_h = a_i$ siendo $i$ el índice de la fila correspondiente a dicha iteración.

Los elementos restantes de la matriz $d$ se calculan fila a fila, de arriba a abajo, y de izquierda a derecha, es decir, en el sentido de lectura habitual. El cálculo de $d_{ij}$ depende de los elementos inmediatamente encima suya y a su izquierda, así como de su elemento más cercanos en la diagonal en dicha dirección y su $d_{k-1t-1}$ correspondiente.

**Ejemplo 2.23.** Si aplicamos el algoritmos sobre las cadenas "ABCDEF" y "ABDECF" que se llevan empleando durante toda esta sección obtenemos la matriz

$$d = \begin{pmatrix} 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 12 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 12 & 1 & 0 & 1 & 2 & 3 & 4 & 5 \\ 12 & 2 & 1 & 1 & 1 & 2 & 3 & 4 \\ 12 & 3 & 2 & 2 & 2 & 2 & 2 & 3 \\ 12 & 4 & 3 & 2 & 2 & 3 & 3 & 3 \\ 12 & 5 & 4 & 3 & 3 & 2 & 3 & 4 \\ 12 & 6 & 5 & 4 & 4 & 3 & 3 & 3 \end{pmatrix}.$$

De acuerdo con el Teorema 2.22 dado que ambas cadenas son de igual longitud 6, el elemento de la matriz $d_{6+2,6+2}$ se corresponde con la distancia de Damerau-Levenshtein entre "ABCDEF" y "ADBECF". Por tanto estás cadenas están a una distancia de 3, lo cual coincide con el peso de las trazas $T_4$ y $\tilde{T}_4$ del Ejemplo 2.14 y su continuación relacionada con el Teorema 2.19.

Hablemos del tiempo de ejecución del algoritmo, este realiza $1 + |a| + |b| + 1$ cálculos al rellenar las dos primeras filas y columnas de la matriz $d$, uno más que el algoritmo de Wagner-Fischer pues precisamos calcular $M$, o max_dist tal y como aparece en el pseudo-código. A continuación en cada uno de los $|a| \cdot |b|$ bucles se realizan un máximo

de 9 operaciones, añadiendo a las del algoritmo de Wagner-Fischer los cálculos de $k$, $t$, $db$ siendo solo los dos primeros obligatorias en todas las iteraciones y el hecho de que el mínimo cuenta con un elemento más. Además, para cada valor de $i$ se actualiza un valor de $da$. Así el número total de operaciones está acotado superiormente por:

$$2 + |a| + |b| + 9 \cdot |a| \cdot |b| + |a|.$$

Por tanto el tiempo total de ejecución del algoritmo debería ser $\mathcal{O}(|a| \cdot |b|)$.

## 2.3. Algoritmo para la clasificación

Una vez presentados los dos algoritmos que emplearemos para el cálculo de las distancias descritas en el Capítulo 1 pasemos a explicar como les daremos uso para solucionar el problema mencionado en la introducción de este trabajo.

A partir de un conjunto de datos de $L$ nombres buscamos clasificarlos de forma que aquellos nombres que corresponden a la misma persona o entidad, que son interpretados como diferentes debido pequeñas alteraciones intencionadas o no, sean agrupados en la misma clase.

La idea del algoritmo que emplearemos para esto pasa por considerar inicialmente un conjunto $P = \emptyset$, a continuación recorremos el conjunto de datos comparando el nombre de cada entrada con aquellos que forman el conjunto $P$. Si consideramos que no es lo suficientemente similar con ninguno entonces añadimos el nombre al conjunto $P$ y fundamos una nueva categoría con este nombre por etiqueta. Por otro lado, si es lo suficientemente similar a alguno de los elementos de $P$ este conjunto no varía y la entrada del conjunto de datos recibe la etiqueta correspondiente a dicho nombre del conjunto $P$. En concreto para inicializar el algoritmo, cuando $P = \emptyset$, añadiremos siempre el primero de los nombres, pues no tiene sentido asignar un nombre a la cadena de caracteres vacía.

```
clasificador(dataset,K,dist):
        P = [ ]
        tags = [ ]
        for i in 1:length(dataset) :
                if i != 1 :
                        d = array(1:length(P))
                        for j in 1:length(P) :
                                d[j] = dist(dataset[i],P[j]) /
                                        max( length(P[j]), length(dataset[i]) )
```

```
                    if min(d) > K :
                            P = P + [dataset[i]]
                            tags = tags + [dataset[i]]
                    else:
                            tags = tags + P[argmin(d)]
            else:
                    P = [ dataset[1] ]
                    tags = [ dataset[1] ]
    return( tags )
```

En el pseudo-código podemos ver como nuestra función de clasificador precisa de 3 inputs: el *dataset* a clasificar, un valor $K \in [0, 1]$ y una función dist que mida la distancia entre dos cadenas de caracteres. Es evidente que emplearemos las distancias definidas en el Capítulo 1 como funciones dist, el concepto de $K$ requiere una mayor explicación pues es la frontera entre aquellos nombres que consideramos iguales y los que no. Para ser concretos, para cada nombre del conjunto de datos que esté siendo considerado calculamos su distancia a los elementos de $P$ y esta cantidad la dividiremos entre la longitud máxima de entre las dos palabras comparadas. Estos valores se almacenan en el vector $d$ y nos centramos en el menor de ellos para compararlo con $K$, si es menor consideramos que la entrada del data set pertenece a la categoría del elemento de $P$ al cual se corresponde dicho valor. Por el contrario si es mayor consideramos que es lo suficientemente diferente para formar su propia nueva categoría. El conjunto $P$ no nos devuelve la categoría de cada entrada del conjunto de datos, de eso se encarga el vector tags que, al final del algoritmo, contiene tantos nombres como entradas tenga el conjunto de datos y está ordenado para que la etiqueta asignada a cada entrada este en la misma posición en la que el algoritmo visitó dicha entrada. El vector $d$ no contiene las distancias entre palabras, si no que las divide por la longitud de la más larga, esto se hace para tener en consideración que no es lo mismo una distancia de 2 en palabras de 4 y 5 caracteres que entre palabras de 10 y 9. Esto invitó en un principio en considerar el uso de la suma de la longitud de ambas palabras pero rápidamente se vio que durante la aplicación práctica esto tendía a agrupar bajo una sola etiqueta palabras de gran tamaño y poca similitud. Además esto permite la acotación de $K$ en el intervalo $[0, 1]$ pues las dos distancias presentadas en el Capítulo 1 tienen la longitud de la mayor palabra involucrada como cota superior.

Entremos ahora a discutir las complicaciones y problemas relacionados con este algoritmo, en concreto: (i) la influencia del orden en que se recorra el conjunto de datos, (ii) la elección del valor $K$ y (iii) el tratamiento de los empates dentro del propio vector $d$, empezaremos por este último.

Si para un cierto nombre $d$ contiene dos entradas con el mismo valor mínimo, inferior a $K$, ¿Como decidimos a cuál asignarlo? Se han abordado varias estrategias, una de ellas la de elegir la etiqueta que tenga una longitud más parecida a la nueva cadena a clasificar, con esta estrategia ganaremos exactitud ante ciertos casos y la perderemos frente a otros e incluso con este criterio el empate podría persistir. Como no existe una estrategia dominante se ha decidido simplemente adjudicar a la nueva cadena la etiqueta que se hubiese añadido antes a $P$.

Por otro lado debemos de pensar en como decidir el valor de $K$, un valor demasiado alto rechazará la similitud de palabras casi idénticas creando demasiadas etiquetas y un valor muy bajo creará muy pocas dando la misma etiqueta a palabras muy distintas. Para encontrar este valor emplearemos un conjunto de datos de entrenamiento junto con un algoritmo de validación cruzada que se tratará en la Subsección 2.3.1, lo cuál nos permitirá comprobar como de bien se comporta el algoritmo ante diferentes valores de $K$.

La influencia del orden en que las entradas del conjunto de datos son comparadas con el conjunto $P$ influye sobre el resultado del algoritmo. Supongamos que nuestro conjunto de datos contiene dos palabras distintas, $a$ y $b$, pero lo suficientemente similares para que el $K$ en uso no las distinga entre sí, pero si con con las demás entradas. Si el algoritmo compara a $a$ con el conjunto $P$ antes que con $b$ añadirá $a$ al conjunto $P$ y le asignará a $b$ la etiqueta $a$. Si es $b$ quien se somete primero a la comparación con $P$ los papeles se invierten. El problema recae en que si mientras que $a$ aparece en una sola entrada $b$ lo hace en muchas, el obtener una buena puntuación dependerá de que la entrada de $a$ no se adelante a todas las de $b$. Al igual que para el valor de $K$ la validación cruzada ayudará con esto, pues a la vez que comparamos valores de $K$ lo hacemos tras haber reordenado de manera arbitraria el orden de las entradas del conjunto de datos. Cabe destacar también que al aumentar el riesgo disminuye la probabilidad de ocurrir, si solo existe una entrada $a$ y 100 de $b$, dependiendo del tamaño total del conjunto de datos, el visitar $a$ antes que $b$ puede perjudicar considerablemente el porcentaje de acierto del algoritmo, pero existen pocas probabilidades de que esto ocurra.

### 2.3.1. Selección del valor de $K$

Dado un conjunto de datos para el cual no solo conocemos los nombres si no también la etiqueta correcta a la que deberían ser asignados, podemos dividirlo en un conjunto de entrenamiento y uno de validación para comprobar la eficacia del algoritmo. La adaptación del algoritmo de clasificación a un conjunto de entrenamiento es sencilla, simplemente eliminamos la inicialización de $P$ como un conjunto vacío y en su lugar $P$ debe de contener las etiquetas presentes en el conjunto de entrenamiento. Para medir el buen comportamiento

del algoritmo tras aplicarlo sobre el conjunto de validación comparamos las etiquetas asignadas con las correctas, dando un valor de 1 si coinciden y 0 si no lo hacen para finalmente sumar todos estos valores y dividir entre la longitud del conjunto de validación. Obtenemos así un valor en el intervalo $[0, 1]$, más concretamente en $\{0, \frac{1}{L}, \frac{2}{L}, \ldots, \frac{L}{L} = 1\}$, siendo $L$ el numero de entradas del conjunto de validación, cuanto mayor sea este número mejor será el comportamiento del algoritmo.

Se define la validación cruzada [8] como un método para, ante la ausencia de un gran conjunto de validación, poder estimar la eficacia de un algoritmo a través del conjunto de entrenamiento. En concreto nos centraremos en la validación cruzada de $k$ iteraciones o *k-fold*, pues es la que emplearemos en el Capítulo 3, en ella dividimos de forma aleatoria el conjunto de entrenamiento en $k$ grupos aproximadamente iguales en tamaño. A continuación, de uno en uno, consideramos cada uno de los $k$ grupos como un conjunto de validación y los otros $k-1$ como un gran conjunto de entrenamiento.

Fijado un valor para $K$ podemos aplicar la validación cruzada de $k$ iteraciones para comprobar como de efectivo resulta. Sea el conjunto número $s \in \{1, \ldots, k\}$ el escogido como conjunto de validación mientras que el resto forman el de entrenamiento, aplicando sobre estos el algoritmo de clasificación tal como se detalla al principio de este apartado obtendremos una puntuación $p_s(K) \in \{0, \frac{1}{L}, \frac{2}{L}, \ldots, 1\}$. Tras repetir el proceso para los otros $k-1$ grupos obtendremos un total de $k$ puntuaciones de las cuales calcularemos la media

$$\text{Puntuación}(K) = \frac{\sum_{i=1}^{k} p_i(K)}{k}.$$

Repitiendo el experimento para varios valores de $K$ podremos comparar como se comporta el algoritmo ante ellos y buscar así el $K$ óptimo para la clasificación. Como norma general se suele emplear la validación cruzada de 5 o 10 iteraciones[8], en el Capítulo 3 emplearemos esto para calcular que valor de $K$ nos da mejores resultados.

# Capítulo 3

# Experimentos con datos reales

En este capítulo presentaremos todo lo relativo a la aplicación de los algoritmos mostrados hasta ahora a datos reales para la solución del problema presentado en la introducción. Comenzaremos exponiendo cuales son los conjuntos de datos que se emplearán y que papel jugará cada uno a lo largo del trabajo, además de como se hizo una limpieza de los mismos para simplificar la programación en Python[9]. A continuación retomaremos el concepto de la validación cruzada presentado en la Subsección 2.3.1, entrando esta vez más en detalle sobre su aplicación y mostrando los resultados y conclusiones obtenidos. Compararemos los tiempos de ejecución del algoritmo de clasificación presentado en el Capítulo 2 empleando las distancias de Levenshtein y Damerau-Levenshtein y finalmente entraremos en otra posible aplicación práctica de estas distancias para crear una herramienta que asista a los clientes a la hora de escribir su usuario dentro de la página web de la empresa. Para esto último se empleará la librería Dash [10] y para la representación de todas las gráficas presentes se empleará plotly [11]. Todo el código empleado para la realización de los estudios y gráficas de este tema puede encontrarse en el repositorio [12].

## 3.1. Conjuntos de datos empleados y adaptación de la teoría

Durante la aplicación práctica de este trabajo se han empleado 3 conjuntos de datos distintos, cada uno con una función y objetivo diferentes, estos son:

- *fleet account owner*(FAO): conjunto de datos de tamaño pequeño con datos reales relativo a los conductores que trabajan con Trucksters. Consta de 142 entradas, o filas, cada una de ellas formada por 3 categorías que toman el papel de columnas: "id" una serie de caracteres asociada a la entrada del conjunto de datos y única para cada una de ellas, "current account owner" recogiendo los nombres introducidos por

los camioneros en la aplicación de Trucksters para realizar distintas tareas y "good account owner" con la etiqueta correcta que define al conductor en cuestión.

- *client account owner*(CAO): conjunto de datos de tamaño mediano con datos reales pertenecientes a clientes de la empresa. Consta de 735 entradas con una estructura similar al conjunto de datos FAO con solo las dos primeras columnas, por lo que no contiene una columna de good account owner y por tanto no sabemos que etiqueta considera la empresa que corresponde a cada entrada.

- *Damerau_fixed*(DF): una copia del conjunto de datos FAO modificada a mano para este trabajo, no contiene datos reales procedentes de la empresa. En la columna de "current account owner" se modificaron ciertas entradas realizando sobre ellas una o varias transportaciones, a mayores se añadió una columna extra bajo el nombre de "fixed" conteniendo un 1 si la entrada fue modificada y un 0 si no lo fue. El objetivo de esto es mostrar una clara diferencia entre los valores dados por las dos distancias principales presentadas durante el Capítulo 1.

Durante todo este trabajo se utilizó Python[9] como lenguaje de programación, los nombres o cadenas de caracteres que comparamos mediante las distancias se recogen en objetos de la clase *string* en este tipo de objetos, se consideran que u, U, ú, ü son cuatro caracteres distintos. Esto complica demasiado el uso del algoritmo para el cálculo de la distancia de Damerau-Levenshtein, pues recordemos que necesitamos dar un alfabeto completo, y también penaliza pequeñas diferencias como tildes o mayúsculas tanto como el uso de una letra distinta. Por estas razones se hará una limpieza de los conjuntos de datos antes de trabajar con ellos. Esto consiste en eliminar las columnas que tengan todas sus entradas vacías, en previsión de posibles conjuntos de datos defectuosos que pudiesen tratar de emplearse en el futuro, y filas o entradas con alguna categoría vacía, pues podría generar problemas con el código. A continuación, en todas las entradas restantes convertimos el texto a minúsculas y aplicamos la función `unidecode` que elimina alteraciones en caracteres del alfabeto inglés. Finalmente estandarizamos la separación entre palabras cambiando los signos ',', ';', '.', ':','(',')','[',']','','', '-' y '_' por espacios en blanco.

**Ejemplo 3.1.** A continuación representamos como cada uno de los pasos mencionados altera una cadena de caracteres siendo aplicados en orden de mención.

$$\text{Anzoátegi,Iñaki} \Rightarrow \text{anzoátegi,iñaki} \Rightarrow \text{anzoategi,inaki} \Rightarrow \text{anzoategi inaki}.$$

La parte de la limpieza de los datos relativa a los espacios en blanco nos lleva a considerar estos como un carácter dentro del alfabeto y por tanto debemos de tener esto en

cuenta a la hora de implementar la distancia de Damerau-Levenshtein. Otra cosa a tener en cuenta a la hora de aplicar las distancias a este caso práctico es la ordenación dentro de nombres que contengan espacios.

**Ejemplo 3.2.** Volviendo sobre el mismo caso que en Ejemplo 3.1, puede darse el caso de que en dos ocasiones diferentes la misma persona haya escrito "Anzoátegi,Iñaki" e "Iñaki Anzoátegi". Para nosotros es evidente que ambas entradas deben compartir etiqueta pero la distancia actual entre ellas, tras su limpieza, es

$$d_L(anzoategi\ inaki, inaki\ anzoategi) = 12 \qquad d_{DL}(anzoategi\ inaki, inaki\ anzoategi) = 12.$$

Esto dejaría registrado en el vector $d$ un valor de $\frac{12}{15} = 0.8$, relativamente alto considerando que solo se ha cambiado el orden del nombre y el apellido.

Es por esto que modificaremos el algoritmo clasificador presentado en el Capítulo 2 para que en caso de que exista una separación en uno de los dos nombres entre los que estamos midiendo la distancia se ordenen alfabéticamente las partes separadas por espacios en blanco de cada cadena de caracteres a la hora de medir las distancias. Esto no implica un cambio en las etiquetas añadidas o por añadir, solo se separarán y ordenarán para medir las distancias, volviendo a su forma original una vez acabado el proceso. Con esto el pseudocódigo del clasificador que aplicaremos finalmente para realizar la validación cruzada ya mencionada será:

```
clasificador_train_sort(dataset_val,K,dist,tags_train):
        P = tags_train
        tags = [ ]
        for i in 1:length(dataset_val) :
                d = array(1:length(P))
                for j in 1:length(P) :
                        if ' ' in dataset_val[i] or P[j] :
                                d[j] = dist(sort(dataset_val[i]),sort(P[j])) /
                                   max( length(P[j]), length(dataset_val[i]) )
                        else:
                                d[j] = dist(dataset_val[i],P[j]) /
                                   max( length(P[j]), length(dataset_val[i]) )
                if min(d) > K :
                        P = P + [dataset_val[i]]
                        tags = tags + [dataset_val[i]]
                else:
```

```
                    tags = tags + P[argmin(d)]
    return( tags )
```

En él la función `sort()` será aquella que divide y ordene las entradas y etiquetas según hemos explicado, tags_train es una lista que contiene las etiquetas correctas de los elementos del conjunto de entrenamiento, sin repetición, y dataset_val es simplemente el conjunto de datos para el cual queremos evaluar la tasa de acierto del algoritmo.

## 3.2. Validación cruzada para obtener el $K$ óptimo

Para realizar la validación cruzada emplearemos el conjunto de datos FAO. En concreto crearemos una función que tras alterar el orden del conjunto de datos lo dividirá en 10 partes[8], validación cruzada *10-fold*, de tamaño aproximadamente igual, dado que tenemos 142 elementos la división no es exacta, los nueve primeros formados por 14 entradas y el último por 16. Cada uno de estos servirá como conjunto de validación mientras que el resto será el de entrenamiento, esto nos dará 10 valores que expresan el porcentaje de acierto del algoritmo. Repitiendo el proceso para varios valores $K$ trataremos de identificar cual da los mejores resultados, la función que definimos permite dar un intervalo donde buscaremos el $K$ y cuantos valores tiene esta rejilla. Como es lógico el experimento debe realizarse una vez por distancia, y los resultados se recogen en las gráficas box-plot presentadas a continuación. El motivo por el que expresar los datos a través de estas gráficas y no quedarnos simplemente con la media es el pequeño tamaño del conjunto de datos FAO. Como se verá a continuación los empates entre medias son frecuentes, con un diagrama de cajas donde el eje horizontal contenga los valores que toma $K$ en la rejilla estudiada y el vertical los porcentajes de acierto en cada uno de los 10 conjuntos de validación tendremos más información y claramente representada. Los gráficos box-plot representarán los puntos que dan lugar a cada una de las cajas a la izquierda de las mismas y dentro de cada una la linea de guiones representará la media de estos valores.

### 3.2.1. Distancia de Levenshtein

Comenzamos el análisis del valor óptimo de $K$ para el conjunto de datos FAO con la distancia de Levenshtein con una rejilla muy amplia para tener una primera idea. La Figura 3.1 recoge los resultados de aplicar la validación cruzada de 10 iteraciones para diez valores de $K$ distintos en $[0, 1]$. El máximo de las medias calculadas se obtiene con $K = 0.6$, este $K$ llama también la atención por la distribución de sus valores todos muy próximos entre sí. Para los otros nueve valores de $K$ la media es menor, aumenta a medida

que nos acercamos a 0.6 y disminuye al superarlo pero manteniéndose por encima de los valores previos a este. En consecuencia buscaremos en una nueva rejilla con diez valores en el intervalo $[0.6, 1]$, pues es donde mayores valores tiene la media y buscamos que exista un posible valor con media superior a la proporcionada con $K = 0.6$.

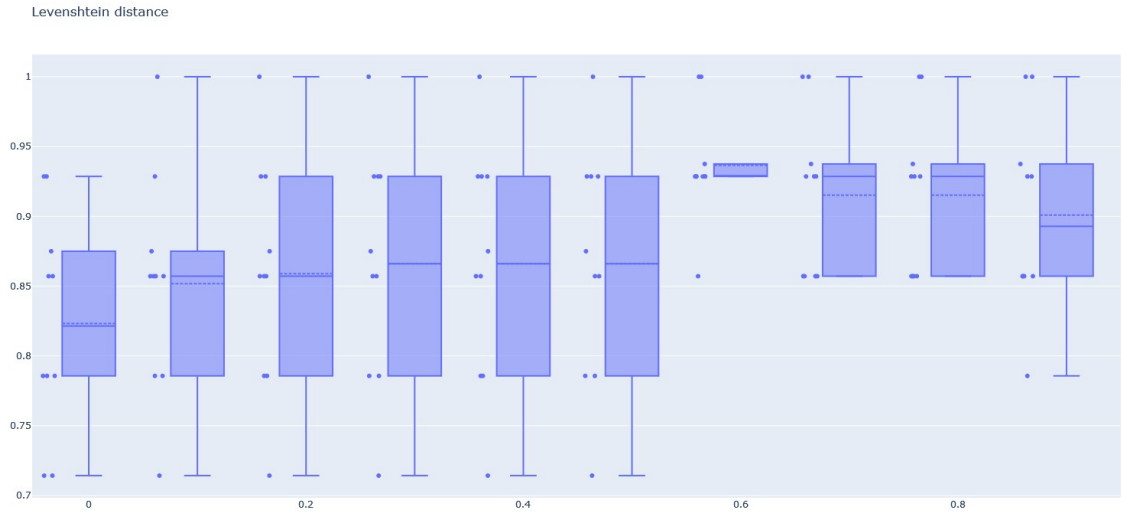



Figura 3.1: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0, 1]$ con la distancia de Levenshtein sobre el conjunto de datos FAO.

La Figura 3.2 recoge los valores en esta nueva rejilla, el valor de 0.6 no varía en absoluto pues con el objetivo de mantener resultados replicables se emplean en todos estos gráficos una misma semilla. $K = 0.6$ se consolida como una anomalía en cuanto a la distribución de sus valores, sin embargo tanto $K = 0.64$ como $K = 0.68$ lo igualan en media. Ambos $K$, $K = 0.64$ y $K = 0.68$, presentan dos valores más en cada extremo de los porcentajes de acierto presentes para K=0.6, uno en el máximo 1 y otro en el mínimo 0.857. El resto de valores de $K$ dan resultados peores en cuanto a la media con un margen considerable, esto nos lleva a realizar un último box-plot de una rejilla de diez valores en el intervalo $[0.6, 0.7]$, pues $K = 0.7$ es el valor estudiado más próximo por la derecha a aquellos con mejor media hasta el momento.

La Figura 3.3 contiene esta nueva gráfica, lo que más destaca es la similitud de los resultados para distintas $K$. Recordemos que independientemente del $K$ el porcentaje de aciertos puede tomar solo un conjunto finito de resultados, en este caso del conjunto $\{\frac{n}{142} / \ n \in \{0, 1, \ldots, 142\}\}$, además el conjunto de datos FAO presenta numerosas entradas

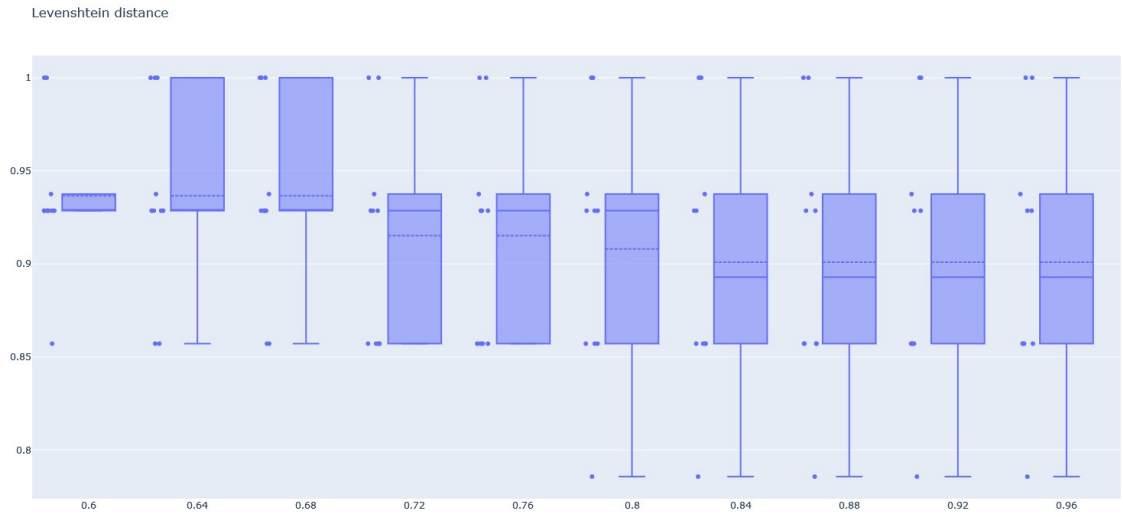



Figura 3.2: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 1]$ con la distancia de Levenshtein sobre el conjunto de datos FAO.

repetidas, más aún tras la limpieza de los datos que elimina mayúsculas y tildes, por lo que los posibles valores son aún menos. En consecuencia cuando dos $K$ son cercanas es posible que no exista diferencia entre usar una o la otra pues no existe entrada del conjunto de datos que una sea capaz de clasificar correctamente y la otra no. En esta gráfica hemos encontrado un nuevo valor máximo de la media, obtenido al emplear $K = 0.69$, este nuevo máximo es de 0.942 frente al anterior máximo de 0.936. Es un cambio bastante pequeño y de hecho podemos comprobar que se debe a la correcta clasificación de una sola entrada, los valores de $K$ previos a 0.69 muestran como uno de sus diez resultados de los cuales calculamos la media el 0.9375, es decir $\frac{15}{16}$, en $K = 0.69$ el cambio respecto a los anteriores valores es que este se sustituye por un 1. De esto se deduce que con $K = 0.69$ el algoritmo clasificó correctamente una entrada más del último de los 10 grupos en los que se dividió el conjunto de datos que sus predecesores.

Como conclusión final, el valor óptimo de $K$ en cuanto a porcentaje de acierto obtenido a través de la validación *10-fold* es $K = 0.69$. El valor $K = 0.6$, o $K = 0.61$, resulta interesante en cuanto a su menor variabilidad, pero todos sus cuantiles son peores que los proporcionados por $K = 0.69$. Esto podría ser una simple coincidencia resultado de que con la división aleatoria empleada las etiquetas que $K = 0.6$ asigna correctamente y las que no se hayan bien distribuidas entre los 10 conjuntos.

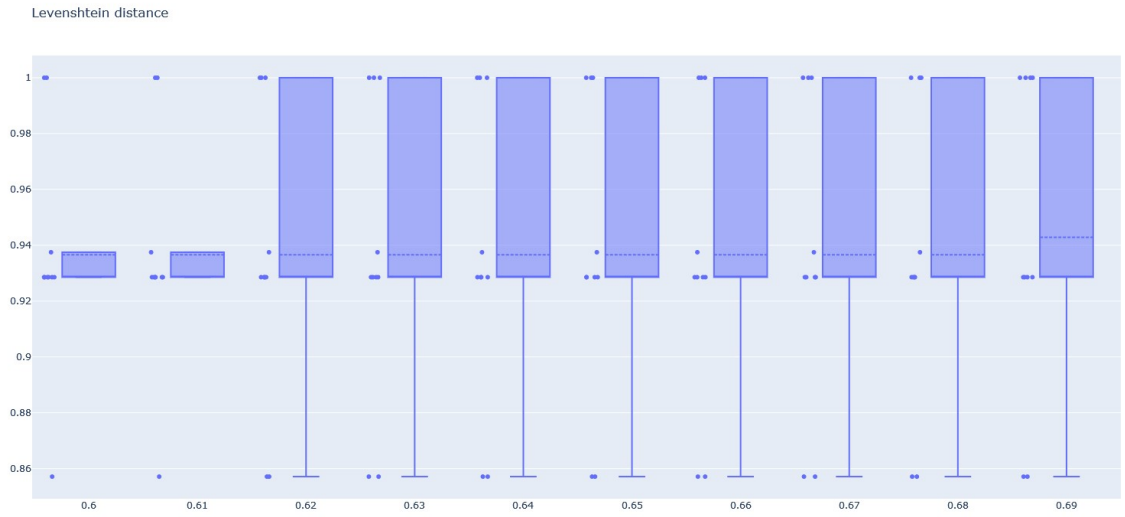



Figura 3.3: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 0.7]$ con la distancia de Levenshtein sobre el conjunto de datos FAO.

### 3.2.2. Distancia de Damerau-Levenshtein

Repetimos el proceso explicado en la sección anterior ahora para la distancia de Damerau-Levenshtein. Continuamos empleando la validación cruzada de 10 iteraciones con la misma semilla en la función que reordena el conjunto de datos, y por tanto los mismos diez grupos son creados en este proceso. Al igual que en el apartado anterior comenzamos con 10 valores en el intervalo $[0, 1]$, este gráfico se recoge en la Figura 3.4 y es exactamente idéntico al de la Figura 3.1. Esto quiere decir que para estos 10 valores de $K$ estudiados y con los grupos aleatorios generados, los resultados no se ven influidos por un cambio entre estas dos distancias.

Dado que se ha obtenido el mismo resultado que en la subsección anterior, el siguiente intervalo a analizar también coincide. Obtenemos así el gráfico recogido en la Figura 3.5, aunque continua siendo muy similar a su equivalente en el estudio de la distancia de Levenshtein ya presenciamos diferencias. En concreto un único valor varió, uno de los valores de $K = 0.68$ pasó de ser 0.928 con la distancia de Levenshtein a 0.857 con la de Damerau-Levenshtein, esto implica que esta distancia tiene un fallo más en un grupo de 14 entradas donde ya había un fallo. Como consecuencia la media para este valor de $K$ será menor, aún así el salto es pequeño comparado con el que se observa al pasar de $K = 0.68$ a $K = 0.72$ por lo que nuestro tercer intervalo será igualmente el $[0.6, 0.7]$.

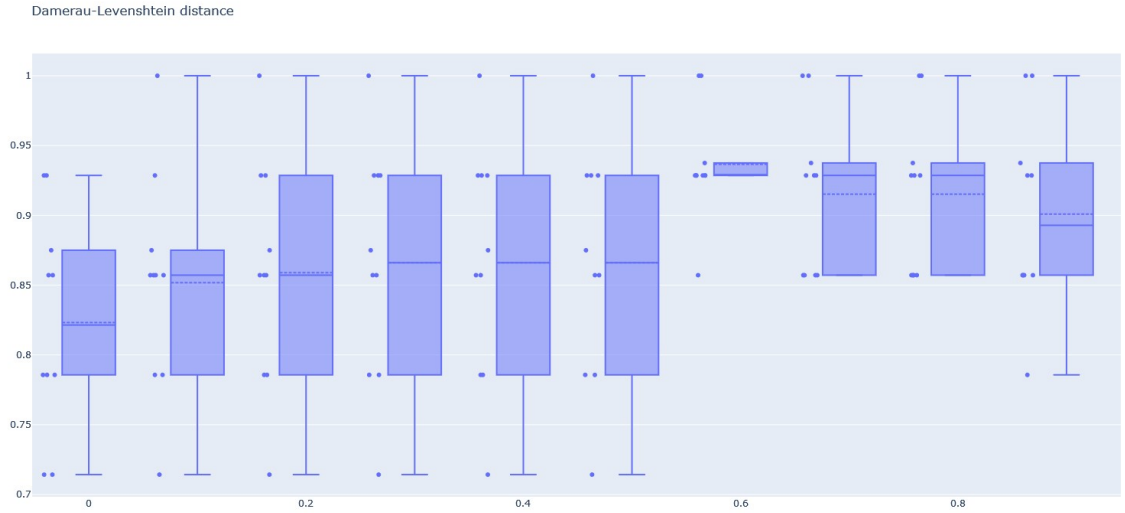



Figura 3.4: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0, 1]$ con la distancia de Damerau-Levenshtein sobre el conjunto de datos FAO.

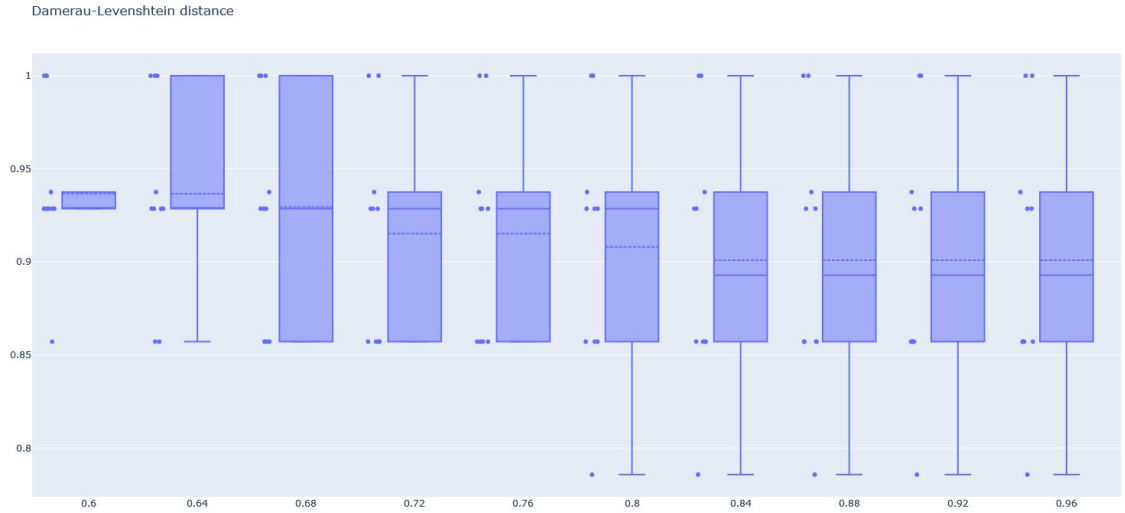



Figura 3.5: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 1]$ con la distancia de Damerau-Levenshtein sobre el conjunto de datos FAO.

La Figura 3.6 recoge lo que será la última gráfica sobre la validación cruzada para el conjunto de datos FAO. Tal y como ya comentamos los resultados no tienen grandes variaciones al tener $K$ con diferencias tan pequeñas entre sí, debido a las mencionadas

características del conjunto de datos FAO, los mejores resultados en cuanto a media suceden para los $K$ estudiados entre 0.6 y 0.64. Cabe destacar que la media en esta ocasión vuelve a disminuir cuando pasamos de $K = 0.68$ a $K = 0.69$, con la distancia de Damerau-Levenshtein este último $K$ erra al clasificar una de las entradas del grupo de 16 datos que sus predecesores si etiquetaban correctamente. Por su parte el $K = 0.6$, o $K = 0.61$, sigue resultando relevante debido a su menor variabilidad con respecto al resto, aunque sus cuantiles son peores que los dados para $K = 0.62$, $K = 0.63$ y $K = 0.64$.

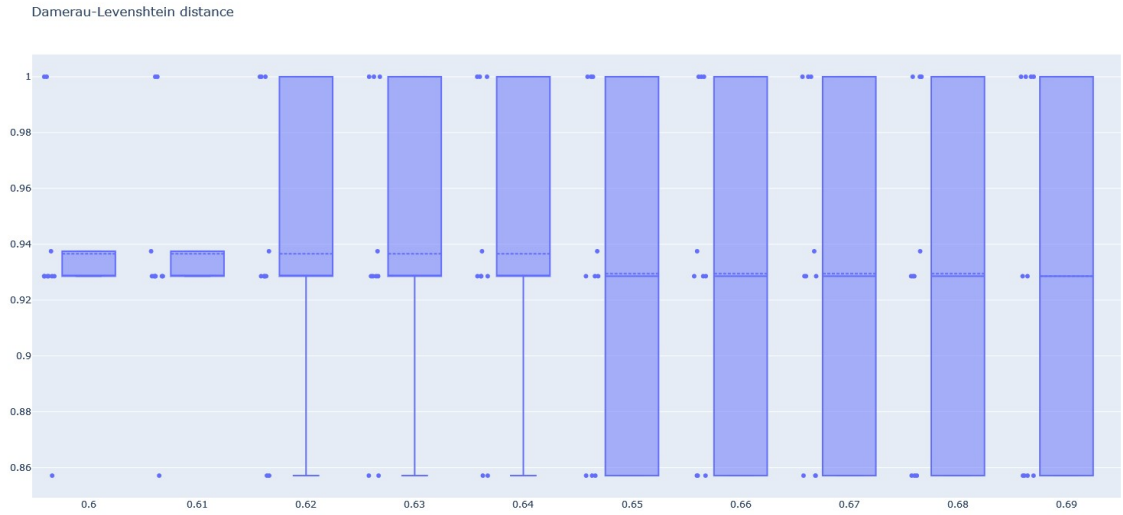



Figura 3.6: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 0.7]$ con la distancia de Damerau-Levenshtein sobre el conjunto de datos FAO.

### 3.2.3. El conjunto de datos DF

El conjunto de datos DF cuenta con un total de 26 entradas modificadas aplicando sobre los datos reales de FAO, esto conforma un porcentaje ligeramente superior al 18 %. Empleando sobre el conjunto de datos DF el algoritmo de validación cruzada de 10 iteraciones para las distancias de Levenshtein y Damerau-Levenshtein para las mismas rejillas de valores de $K$ que en las dos subsecciones anteriores se obtienen los resultados presentados en las siguientes imágenes.

Las Figuras 3.7 y 3.8 proporcionan resultados prometedores, a primera vista, respecto a cumplir el propósito para el que se creó el conjunto de datos DF, mostrar claras diferencias entre las dos distancias estudiadas. Sin embargo las gráficas son idénticas entre sí a partir del valor $K = 0.4$, coincidiendo el máximo de la media de nuevo en $K = 0.6$ aunque esta vez con un valor muy próximo al presentado por los valores $K = 0.7$ y $K = 0.8$.

Entre la Figura 3.9 y la Figura 3.10 solo varía uno de los datos correspondientes a $K = 0.68$, siendo menor en el caso de la distancia de Damerau-Levenshtein. Aún así este cambio es bastante relevante respecto a la búsqueda del $K$ óptimo, pues con la distancia de Damerau-Levenshtein tan solo $K = 0.64$ tiene una media de 0.923, mientras que con la distancia de Levenshtein este valor lo comparte con $K = 0.68$.

Finalmente las diferencias vistas en las dos gráficas anteriores vuelven a ser las únicas presentes entre las recogidas en las Figuras 3.11 y 3.12. En estas dos los dos saltos en el valor de la media que ya vimos para la distancia de Damerau-Levenshtein entre $K = 0.6$, $K = 0.64$ y $K = 0.68$ ocurren en el paso de $K = 0.61$ y $K = 0.64$ al valor siguiente. En el caso de la distancia de Levenshtein solo el primero de ellos ocurre. Para el conjunto de datos DF no se aprecia ninguna diferencia entre $K = 0.68$ y $K = 0.69$.

Pese a haber alterado considerablemente el conjunto de datos FAO, no se han apreciado diferencias relevantes más allá de la ausencia de mejora en el paso de $K = 0.68$ a $K = 0.69$ en el caso de la distancia de Levenshtein. Se esperaba obtener una variación considerable entre el $K$ óptimo para la distancia de Damerau-Levenshtein en el conjunto de datos DF respecto al de FAO. Hemos visto sin embargo que para esto no basta con los cambios efectuados.

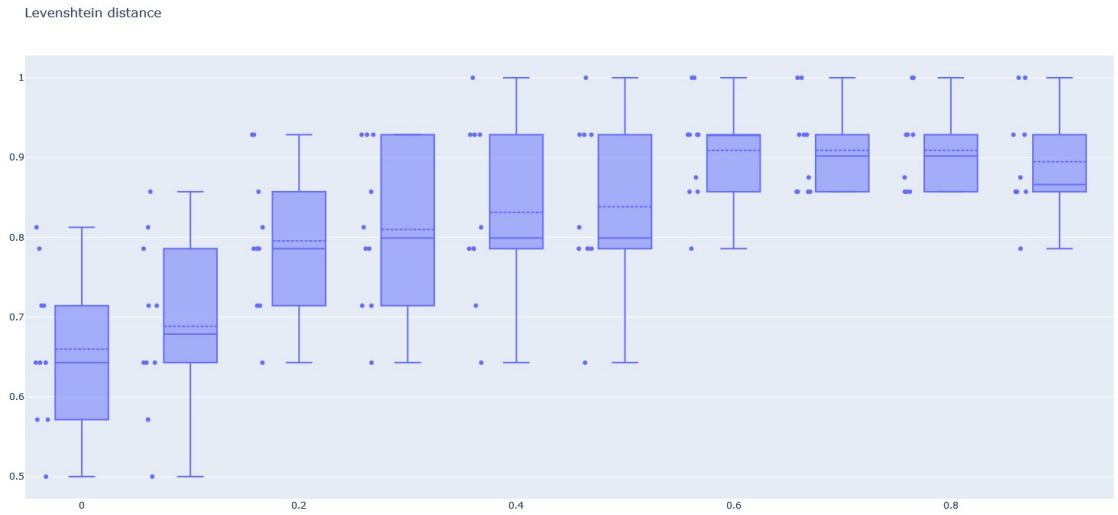



Figura 3.7: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0, 1]$ con la distancia de Levenshtein sobre el conjunto de datos DF.

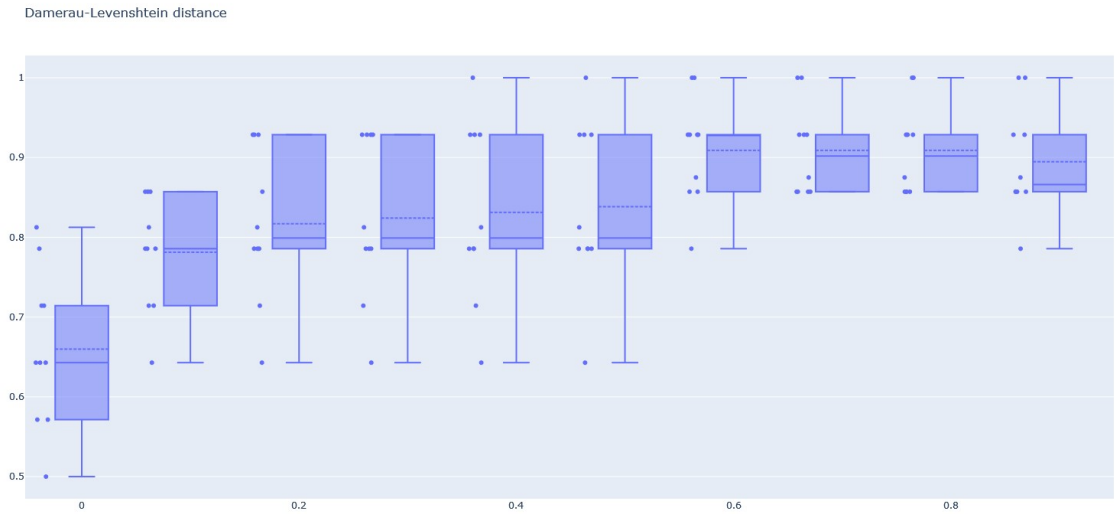



Figura 3.8: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0, 1]$ con la distancia de Damerau-Levenshtein sobre el conjunto de datos DF.

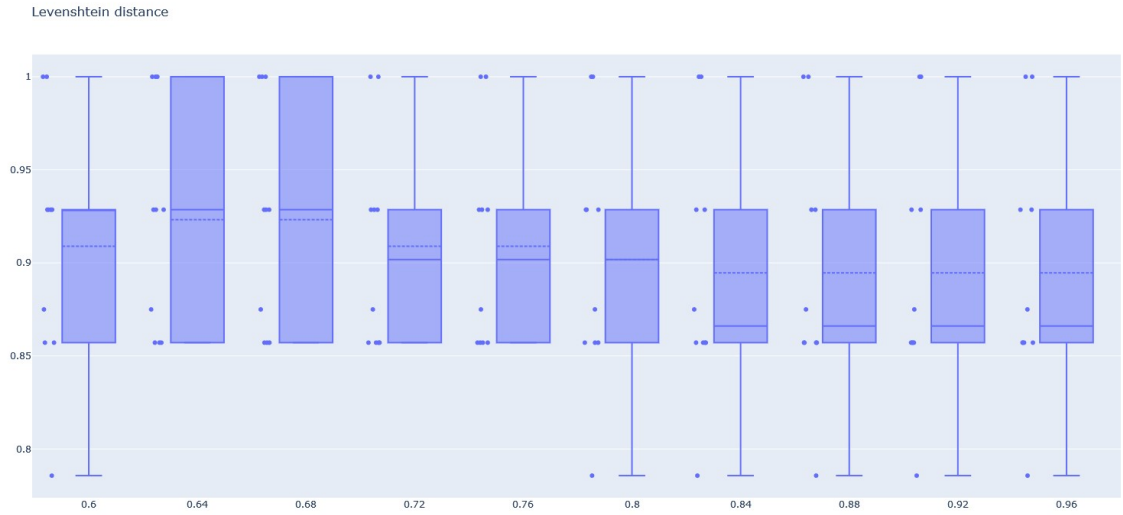



Figura 3.9: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 1]$ con la distancia de Levenshtein sobre el conjunto de datos DF.

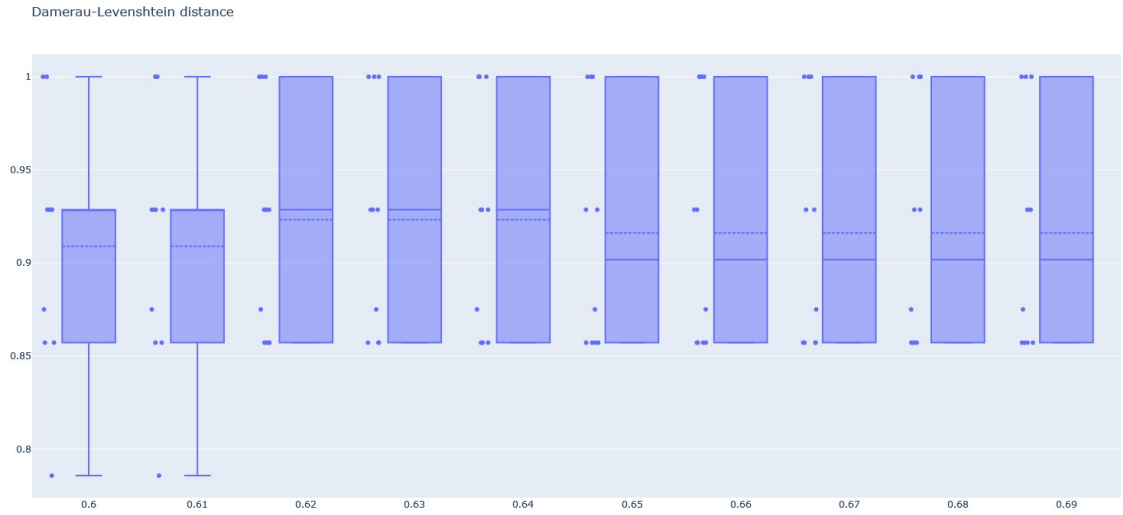



Figura 3.10: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 1]$ con la distancia de Damerau-Levenshtein sobre el conjunto de datos DF.

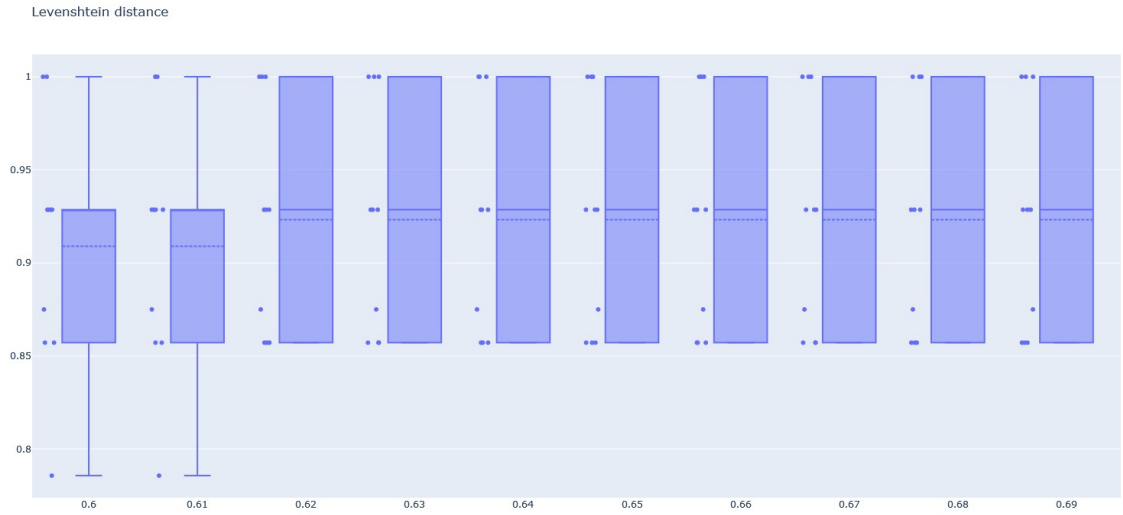



Figura 3.11: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 0.7]$ con la distancia de Levenshtein sobre el conjunto de datos DF.

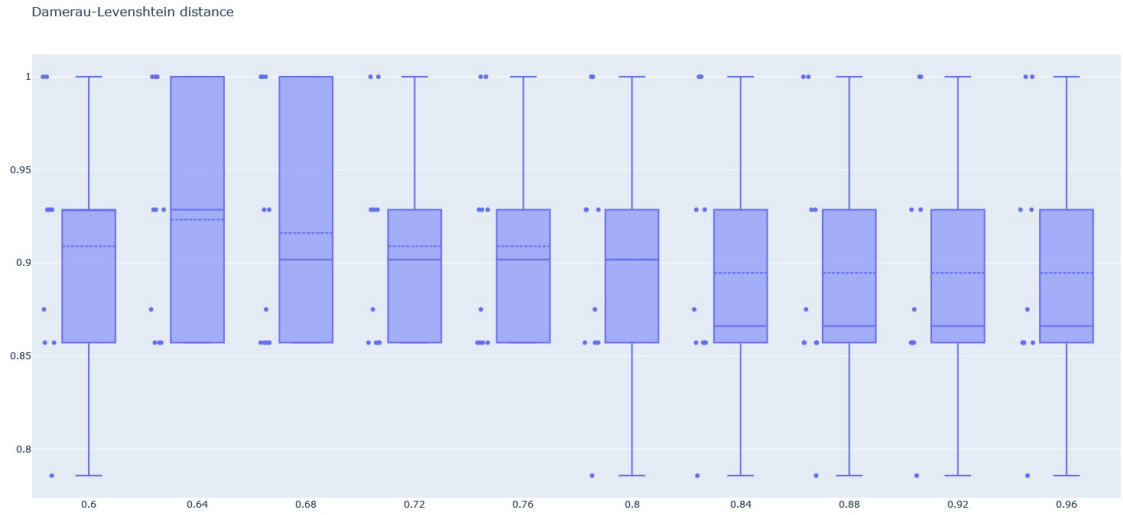



Figura 3.12: Gráfica box-plot de los resultados obtenidos para 10 valores de la rejilla situada en $[0.6, 0.7]$ con la distancia de Damerau-Levenshtein sobre el conjunto de datos DF.

### 3.2.4. Conclusiones sobre el $K$ óptimo

En base a los visto hasta ahora en esta sección la elección del $K$ óptimo para cada una de las distancias parece evidente, en el caso de la distancia de Levenshtein emplearemos $K = 0.69$ y en el caso de la distancia de Damerau-Levenshtein debemos de escoger entre uno de los tres $K$ que compartían resultado tras el estudio del algoritmo de la validación cruzada.

Existe sin embargo, el caso de $K = 0.6$ y su escasa variabilidad en los estudios realizados. Es probable que estos resultados, recordemos que los compartía con $K = 0.61$, sean simplemente una consecuencia de la reordenación y división del conjunto de datos para la validación cruzada de 10 iteraciones. Para comprobarlo compararemos los porcentajes de acierto de los $K$ óptimos para ambas distancias con los de $K = 0.6$ bajo un marco más general. Dividiremos arbitrariamente el conjunto de datos FAO en un conjunto de entrenamiento que comprenda el 80 % de las entradas dejando el 20 % restante como conjunto de validación. Aplicando la adaptación del algoritmo de clasificación a la presencia de estos dos tipos de conjuntos calcularemos el porcentaje de aciertos en el conjunto de validación empleando el considerado $K$ óptimo y $K = 0.6$. Repetiremos el proceso de división aleatoria de los datos y aplicación el algoritmo un número elevado de veces para garantizar que la división aleatoria empleada no tenga ninguna clase de influencia. Para que los resultados aún así continúen siendo replicables, emplearemos una semilla fijada distinta para cada par de conjunto de entrenamiento y conjunto de validación.

La Figura 3.13 recoge la comparación entre $K = 0.6$ y $K = 0.69$ para la distancia de Levenshtein a través de gráficos box-plot con un total de 100 datos. Como podemos observar el valor medio dado por $K = 0.69$ es superior al proporcionado por $K = 0.6$, 0.939 frente a 0.935, por lo que en este caso parece que la escasa variabilidad observada previamente para $K = 0.6$ no reflejaba la realidad. Es más claramente este valor de $K$ presenta más resultados por debajo del 0.9 que el $K$ óptimo $K = 0.69$. Con los resultados obtenidos sobre el conjunto de datos FAO para la distancia de Levenshtein se concluye que el mejor valor de $K$ para el algoritmo de etiquetado es $K = 0.69$.

La Figura 3.14 recoge la comparación entre los tres candidatos a $K$ óptimo y $K = 0.6$ para la distancia de Damerau-Levenshtein bajo las condiciones ya explicadas para la Figura 3.13. En esta ocasión si que parece haber una menor variabilidad de los datos con $K = 0.6$, siendo que su media de 0.935 es superior a la obtenida para los otros, siendo todas inferiores a 0.933. Dado esto para el caso de la distancia de Damerau-Levenshtein escoger el $K = 0.6$ si parece resultar beneficioso frente al uso de los valores que dieron mejores resultados en

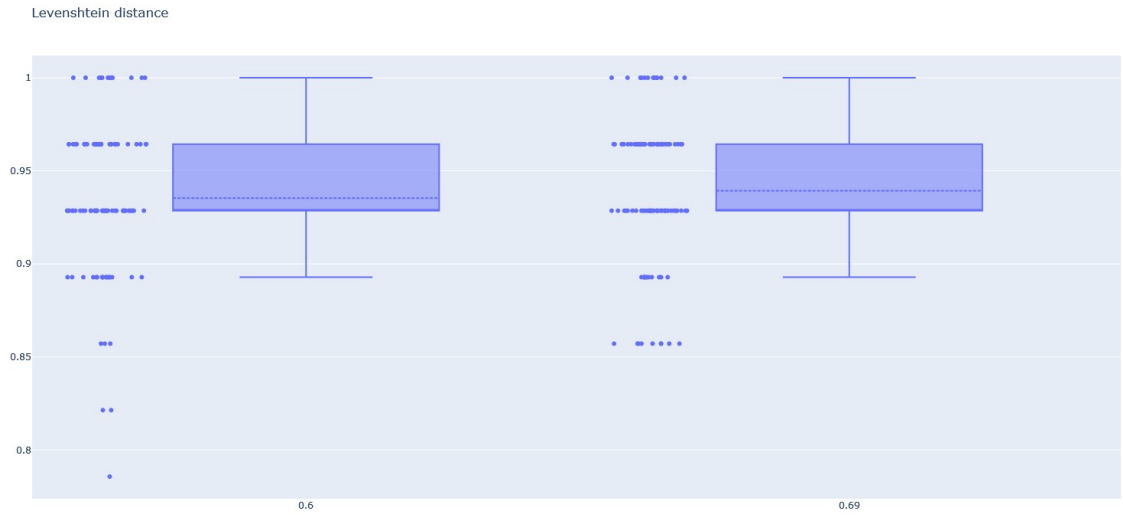



Figura 3.13: Gráfica box-plot comparativa entre los porcentajes de aciertos del algoritmo de etiquetado empleando la distancia de Levenshtein sobre 100 conjuntos da entrenamiento-validación distintos del conjunto de datos FAO para los $K$ de interés.

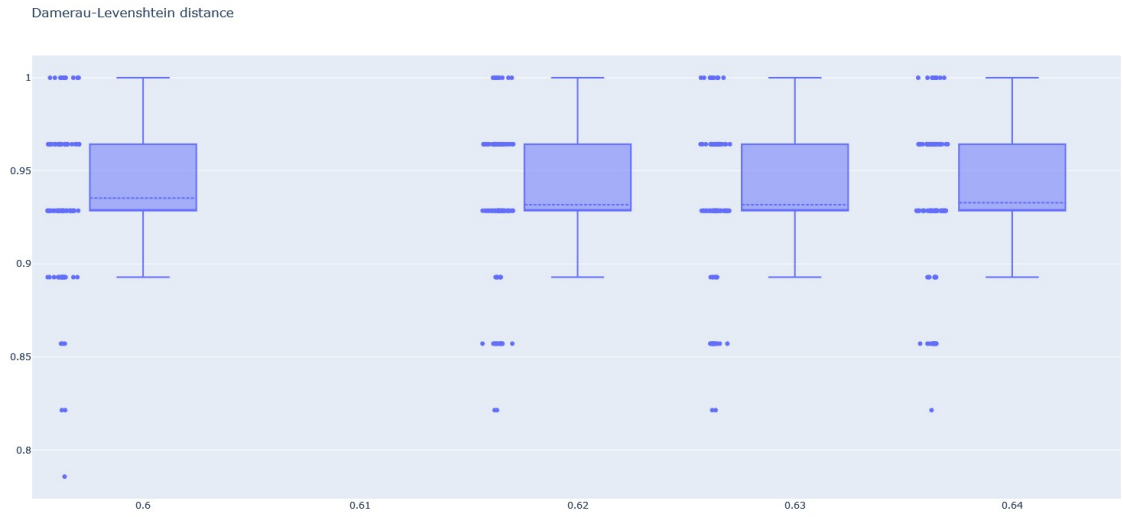



Figura 3.14: Gráfica box-plot comparativa entre los porcentajes de aciertos del algoritmo de etiquetado empleando la distancia de Damerau-Levenshtein sobre 100 conjuntos da entrenamiento-validación distintos del conjunto de datos FAO para los $K$ de interés.

la validación cruzada.

## 3.3. Comparación de las distancias

Una vez elegidos los valores de $K$ que se van a utilizar para cada una de las distancias, debemos de plantearnos que distancia nos interesará más emplear. Si comparamos los diagramas de cajas vistos hasta este momento ninguna distancia parece tener ventaja respecto a la otra en términos de los porcentajes de acierto. Pese a existir diferencias entre el porcentaje medio observado para $K = 0.69$ con la distancia de Levenshtein (Figura 3.13) siendo superior al obtenido con $K = 0.6$ en la distancia de Damerau-Levenshtein (Figura 3.14), la diferencia no es suficientemente amplia como para justificar el uso de una sobre la otra sin comparar antes los tiempos de ejecución.

Aplicamos 50 veces el algoritmo de etiquetado para cada una de las dos distancias empleando sus valores de $K$ óptimos y recogemos sus tiempos de ejecución. En cada una de las 50 iteraciones del algoritmo empleamos una ordenación aleatoria del conjunto de datos distinta, utilizando 50 semillas fijadas distintas. Repetimos este proceso para los dos conjuntos de datos reales de los que disponemos, la Figura 3.15 recoge los tiempos de clasificación del conjunto de datos FAO y la Figura 3.16 los de CAO. En ambos casos el eje $y$ recoge el tiempo en segundos que ha tardado el algoritmo en ejecutarse y el eje $x$ a cual de las 50 iteraciones realizadas corresponde el tiempo, y por tanto que semilla fue empleada en la reordenación. Como resulta lógico los tiempos de ejecución sobre el conjunto de datos CAO son muy superiores sobre los de FAO, pues cuenta con aproximadamente 5 veces más datos. Por los tiempos de ejecución teóricos y el número de operaciones necesarias para el calculo de las distancias recogidos en el Capítulo 2[4][5], también era de esperar que al emplear la distancia de Damerau-Levenshtein el tiempo de ejecución fuese mayor ya que requiere más operaciones que al utilizar la distancia de Levenshtein. En el conjunto de datos FAO, de tamaño pequeño, los tiempos de ejecución son del doble, mientras que en el conjunto de datos CAO, de tamaño medio, rondan el triple. Cabe destacar que el orden en el que el algoritmo accede a las entradas del conjunto de datos no parece tener influencia en el tiempo de ejecución, pues los repuntes de tiempo no coinciden en la misma iteración del algoritmo para ambas distancias.

Debido a su rápida ejecución en comparación con la distancia de Damerau-Levenshtein, así como sus mejores resultados observados bajo el $K$ óptimo, se concluye que la mejor versión del algoritmo de clasificación se alcanza al emplear la distancia de Damerau-Levenshtein con $K = 0.69$.

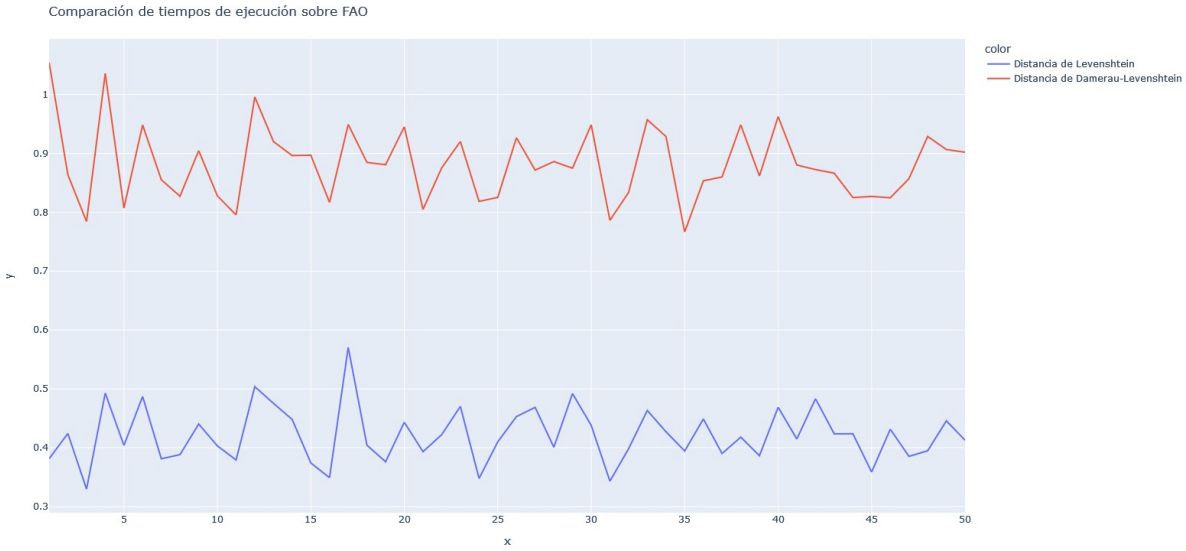



Figura 3.15: Tiempos de ejecución, en segundos, del algoritmo de etiquetado sobre el conjunto de datos FAO en 50 instancias diferentes.

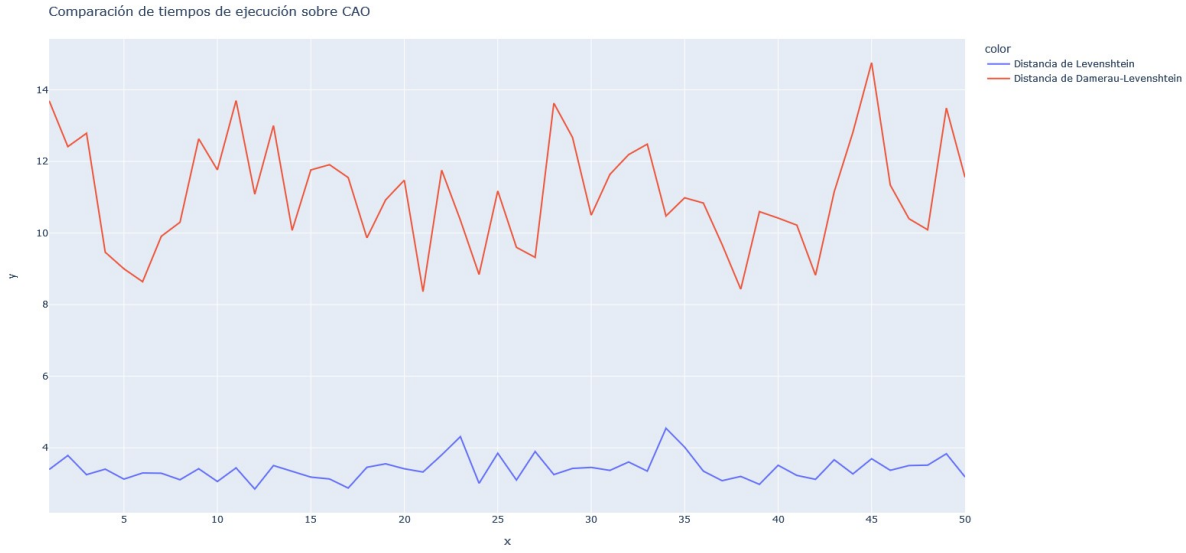



Figura 3.16: Tiempos de ejecución, en segundos, del algoritmo de etiquetado sobre el conjunto de datos CAO en 50 instancias diferentes.

## 3.4. Sugerencia de etiquetas

Empleando el algoritmo de etiquetado podemos obtener un conjunto que contenga todas las etiquetas empleadas, el conjunto $P$ en el pseudo-código presentado para el algoritmo

de etiquetado en la Sección 2.3. Este conjunto contendrá las que el algoritmo considera ser todas las etiquetas que correspondientes a las entradas presentes en el conjunto de datos empleado. Aplicando el algoritmo de clasificación sobre un conjunto de datos para el cual no dispongamos de las etiquetas correctas, y por tanto no podamos emplear ningún proceso que requiera de conjuntos de validación y entrenamiento, encontraremos cuantas entradas diferentes considera el algoritmo que existen realmente en el conjunto de datos así como los nombres de estas y cuantas veces se repiten cada una si nos interesase, esto último empleando la lista *tags* del pseudo-código de la ya mencionada Sección 2.3.

Surge así la idea de crear un programa que ayude a los clientes a la hora de operar en la página web de la empresa de forma más cómoda a través del conjunto de datos CAO. Aplicando el algoritmo de clasificación sobre este conjunto de datos y obteniendo su $P$ podemos crear un programa que compare lo que un usuario este tecleando dentro de la página de Trucksters, lo compare con los elementos de $P$ y nos sugiera los tres más similares si existen tres o más que se parezcan lo suficiente y en otro caso sugiera solo los que se parezcan lo suficiente, llegando a no sugerir nada de ser el caso. Para esto se empleará la librería de Python Dash [10].

El algoritmo de clasificación emplearía $K = 0.69$ y la distancia de Levenshtein siendo esto consecuencia de los resultados obtenidos en el estudio del Capítulo 3 previo a este punto sobre el conjunto de datos FAO. Una vez obtenido este conjunto $P$ compararemos cada vez que se tecleé una letra en el recuadro de texto la misma forma de comparación que en el algoritmo, calculando la distancia del texto introducido con las etiquetas existentes y dividiendo este valor por el máximo de las longitudes de ambos, a continuación se compara el menor de estos valores con el 0.69. Dado que queremos las tres etiquetas más similares de entre las que se parezcan suficiente en caso de existir, si el mínimo valor calculado es menor que 0.69 se elimina la etiqueta de $P$ y se vuelve a realizar el mismo cálculo para obtener un nuevo mínimo. Se repite este proceso hasta tres veces o hasta que el mínimo sea superior a 0.69, en este caso simplemente no se sugiere ninguna etiqueta más. Dado que esto se repite cada vez que una letra sea tecleada el conjunto $P$ vuelve a su estado inicial cada vez que esto pasa para no dejar de recomendar etiquetas ya sugeridas.

### 3.4.1. Resultados obtenidos

Aplicando todo lo explicado en esta sección hasta el momento sobre el conjunto de datos CAO, empleando de nuevo una semilla fija para la reordenación aleatoria del conjunto en la que será sometido al algoritmo de etiquetado, obtenemos un conjunto $P$ de tan solo 39 elementos, pocos considerando las 735 entradas del conjunto de datos. Tras revisar manualmente el conjunto de datos CAO, recordemos que no dispone de una columna

"good account owner" por lo que no podemos saber exactamente las etiquetas reales que daría la empresa, obtenemos 56 etiquetas distintas. Varios problemas se hacen presentes observando el nombre de las entradas (para evitar emplear información sensible se hablará en términos generales de los problemas, sin emplear ejemplos reales) :

- La aproximación a la presencia de espacios: En el conjunto $P$ aparecen dos etiquetas idénticas diferenciadas por un único espacio, este separa el nombre de una persona de su apellido. Volviendo sobre el Ejemplo 3.1 para ilustrarlo, estas entradas serían "inaki anzoategi" e "inakianzoategi", si no hubiésemos alterado la forma de cálculo ante la presencia de espacios en blanco todas las entradas bajo estas etiquetas habrían compartido una sola. El problema reside en que nuestro método, al observar un espacio, realmente compara "inakianzoategi" con "anzoategi inaki" pues la 'a' precede a la 'i' en el alfabeto. Dos posibles soluciones surgen para su posible estudio en la eficiencia, la primera es considerar todas las posibles permutaciones de las entradas que contengas espacios. El contrapunto es que ante una entrada con 4 espacios, cinco sub-cadenas de letras a permutar, el tiempo de cálculo es muy elevado teniendo que considerar 120 posibles ordenaciones para una sola entrada y elevándose aún más si alguna de la etiquetas ya existentes con la que comparamos esta entrada también contiene espacios en blanco. La otra idea sería confiar en el usuario y simplemente comparar el valor obtenido tras la reordenación de la entrada y compararlo con la entrada sin reordenar para quedarnos con el menor.

- Las similitudes entre nombres de tamaño pequeño: Ante la presencia de entradas cortas, de entre cuatro y seis letras, normalmente correspondientes a nombres propios, el algoritmo tiene problemas para distinguirlos. Esta clase de entradas tienden a compartir un alto porcentaje de letras debido a la naturaleza del idioma en cuanto a la relación entre letras vocales y consonantes. Es sencillo pensar en nombres claramente distintos de entre cuatro y seis letras, pero la mayoría repetirán vocal entre sí haciendo menor la distancia de Levenshtein que los separa. Volviendo sobre el Ejemplo 3.1 se tiene que

$$d_L(\text{“}inaki''\text{, “}isabel'') = 4,$$

dado que la mayor longitud de entre las dos cadenas es la de "isabel", 6, el valor que compararíamos con $K = 0.69$ es $\frac{4}{6} \approx 0.66$, por lo que compartirían etiqueta. Un posible estudio futuro podría surgir de aquí, tratando de encontrar una función $f_K$ óptima para este algoritmo que empleé un valor de $K$ u otro dependiendo de la longitud de las cadenas implicadas.

# Capítulo 4

# Conclusiones

Los estudios realizados a lo largo de este trabajo sobre las distancias de Levenshtein y Damerau-Levenshtein nos llevan a considerar que, en lo referente a los conjuntos de datos estudiados, el uso de una u otra distancia en el algoritmo de clasificación descrito en la Sección 2.3, cada una empleando su $K$ óptimo calculado por validación cruzada, es indiferente en cuanto al porcentaje de acierto del mismo. Esto convierte a la distancia de Levenshtein en la mejor opción de las dos por su menor tiempo de cálculo (Figuras 3.16 y 3.15) debido a que son necesarias menos operaciones para calcularla. El estudio sobre el conjunto de datos DF muestra que la presencia de transposiciones en cerca de un quinto de las entradas no es suficiente para generar grandes diferencias entre los valores $K$ óptimos de ambas distancias. Por último la Sección 3.4 trata de emplear la distancia de Levenshtein junto con su $K$ óptimo, $K = 0.69$, para clasificar el conjunto de datos CAO. Es difícil sacar conclusiones de estos resultados, pues Trucksters no tiene clasificados los elementos de CAO según su etiqueta correcta, pero parece poco prometedor. El número de etiquetas creadas es un 30 % inferior al que hemos considerado al contar manualmente posibles etiquetas, siendo los principales motivos los expuestos en la Subsección 3.4.1.

Como trabajos futuros relacionados con lo expuesto en este se tienen las dos propuestas al final del Capítulo 3: crear un algoritmo de clasificación donde el $K$ varíe con la longitud de las palabras cuya distancia se esta calculando en cada momento y comparar su resultados con los aquí expuestos, y estudiar posibles formas de mejorar el comportamiento de las comparaciones ante cadenas de palabras que presenten espacios. A mayores, un estudio similar al expuesto aquí para las distancias de Levenshtein y Damerau-Levenshtein podría realizarse sobre las distancias de Jaro y Jaro-Winkler mencionadas en la Sección 1.3 con el objetivo de comparar las cuatro entre sí.

# Bibliografía


[1] Solís, E. (2022). *Luis Bardají / Trucksters.* Consultado el 24 de enero 2023, disponible en la web https://startupsoasis.com/

[2] Jiménez, M. (2022). *Trucksters capta 8 millones en pleno caos logístico por Ucrania y la pospandemia.* Consultado el 24 de enero 2023, disponible en la web https://cincodias.elpais.com

[3] Kun, J. (2011), *Metrics on Words.* Consultado el 24 de enero 2023, disponible en la web https://jeremykun.com

[4] Wagner, R. A. y Fischer, M. J. (1974), *The String-to-String Correction Problem.* Journal of the Association for Computing Machinery 21:168-173.

[5] Lowrance, R. y Wagner, R. A. (1975), *An Extension of the String-to-String Correction Problem.* Journal of the Association for Computing Machinery 22:177–183.

[6] Wu, G. (2022), *String Similarity Metrics – Edit Distance.* Consultado el 24 de enero 2023, disponible en la web https://www.baeldung.com

[7] Yujian, L. y Bo, L. (2007), *A Normalized Levenshtein Distance Metric.* IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 29, no. 6, pp. 1091-1095

[8] James, G., Witten, D., Hastie, T., Tibshirani, R. *An Introduction to Statistical Learning*, 2nd ed, Capítulo 5 páginas 197-208.

[9] Van Rossum, G., Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.

[10] Hossain, S., Calloway, C., Lippa, D., Niederhut, D. y Shupe, D. (2019). Visualization of bioinformatics data with dash bio. In Proceedings of the 18th Python in Science Conference (pp. 126-133).

[11] Plotly Technologies Inc. Collaborative data science. Montréal, QC, 2015. https://plot.ly.

[12] https://github.com/AlexandreDominguezPieto/Distancia-de-Levenshtein-como-clasicador-de-textos