UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Aplicación de Técnicas Soft Computing y Heurísticas para la identificación y clasificación de la información empleada por un recomendador de recetas

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Peñaranda Cebrián, Cristian

Director: Julian Inglada, Vicente Javier

Co-directora: Valero Cubas, Soledad

4 de julio de 2015

Resumen

La información extraída de Internet se encuentra por lo general muy desestructurada y es difícil de tratar. En el caso de las recetas de cocina es habitual encontrar los ingredientes representados de formas muy diferentes según el autor de la receta e incluso la localización de la misma. Pese a que existen microformatos semánticos específicos para recetas para etiquetar la información en Internet, a día de hoy no se encuentran lo bastante extendidos. En este trabajo se propone el desarrollo y aplicación de técnicas soft computing para la búsqueda de la mejor coincidencia de los ingredientes extraídos de diferentes recetas frente a la base de datos de referencia mundial USDA (United States Department of Agriculture). El objetivo es establecer la mejor relación posible entre los ingredientes de una receta extraída de Internet y la base de datos USDA. Se desarrollarán además heurísticas específicas para el dominio que complementen el proceso de búsqueda

Palabras clave: Soft Computing, Sistemas de Recomendación, Inteligencia Artificial

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Receteame.com[1] es un recomendador on-line de recetas de cocina. En dicha web hay una gran cantidad de recetas recopiladas a través de Internet. Esta página trata de reunir toda la información nutricional de sus recetas, además de informar a los vegetarianos si éstas contienen carne, ayudar a las personas que sufran algún tipo de intolerancia, como los intolerantes al gluten o a la lactosa, informandoles si pueden comer estas recetas. El trabajo que se presenta en este documento trata de resolver un problema crucial existente en dicho recomendador. Dicho problema radica en la complejidad de obtener automáticamente los valores nutricionales de los ingredientes contenidos en sus recetas.

Para poder resolver este problema se necesita de una base de datos que contenga información nutricional de los ingredientes. En este trabajo se decidió usar la base de datos de ingredientes United States Department of Agriculture (USDA)[2]. El principal motivo por el que se utilizó esta base de datos es debido a la gran cantidad de ingredientes que posee. Además ofrece para cada ingrediente una información nutricional muy detallada. Utilizar esta base de datos conlleva un inconveniente, que consiste en que la base de datos describe los ingredientes en inglés, mientras que en las recetas recuperadas por receteame.com están en español. Por ello, inicialmente deberemos realizar una traducción del ingrediente con el que vamos a trabajar.

La base de datos USDA organiza los nombres de los ingredientes en una

[1]Web realizada por el Grupo de Tecnología Informática (GTI).
[2]Podemos encontrar más información en la página www.usda.gov.

cadena, indicando sus diferentes características separadas a través de comas. El problema que se produce al realizar la búsqueda es que los errores que se producen son frecuentes, devolviendo valores nutricionales de ingredientes incorrectos. De esta forma, la idea principal de este trabajo es mejorar los resultados que se obtienen al realizar búsquedas sobre la base de datos anteriormente comentada. Es decir, encontrar la mejor coincidencia de los ingredientes, de las diferentes recetas, frente a la base de datos. Para mejorar esta búsqueda propusimos realizar una división de la cadena utilizando las comas como referencia. Esta división está compuesta como máximo por once campos, que a partir de ahora llamaremos campos de características. La hipótesis de partida de este proyecto es que la ordenación en campos de las características nos puede permitir mejorar los resultados de búsqueda. Así, pensamos que unos campos son más importantes que otros, y por ello deberán tener más peso en la búsqueda. Sin embargo, no está claro cuales son esos campos, ni tampoco conocemos exactamente en qué posición se encuentra. Para demostrar esta hipótesis utilizaremos la base de datos MongoDB[3] que ya lleva incorporado un algoritmo de búsqueda que nos permite darle diferentes pesos a los campos y, además, nos permite realizar búsquedas de texto sobre la base de datos. En el capítulo 3 describimos como está diseñada la estructura de esta base de datos.

Para ilustrar mejor el problema de la búsqueda, si realizásemos una búsqueda con la situación actual de la base de datos del ingrediente *bacon*, se puede apreciar en la figura 1.1, que el resultado correcto aparece en la tercera posición. En cambio, si le aplicáramos la división de la cadena utilizando las comas como referencia, asignando un mayor peso al segundo campo que a los otros, tendríamos que el primer resultado es el esperado. Podemos observar el resultado de este segundo ejemplo en la figura 1.2.

A continuación podemos observar en la figura 1.3 un ejemplo de la búsqueda de otro ingrediente, donde se muestra el resultado de una búsqueda en la base de datos USDA del ingrediente *vino tinto*. La base de datos nos devuelve una lista de ingredientes, entre los cuales está el ingrediente que estamos buscando, pero el primero de la lista en este ejemplo no nos sería de utilidad y nos conduciría al error (*vinegar*). Este ejemplo nos muestra el caso de como, después de encontrar una combinación de pesos adecuada en la base de datos, es necesario aplicar unas heurísticas que ayuden a mejorar los resultados devueltos. Así, para algunos ingredientes nos resulta muy difícil encontrar el resultado esperado

[3]Podemos encontrar esta base de datos en www.mongodb.org.

```
Search:"bacon"
Exit:
          long_desc                      Description Food Group
Bacon, meatless                         Legumes and Legume Products
Bacon, beef sticks                      Sausages and Luncheon Meats
Pork, bacon, rendered fat, cooked              Pork Products
Turkey bacon, unprepared                Sausages and Luncheon Meats
Turkey bacon, microwaved                Sausages and Luncheon Meats
Canadian bacon, unprepared                     Pork Products
Animal fat, bacon grease                       Fats and Oils
Pork, cured, bacon, unprepared                 Pork Products
```

Figura 1.1: Ejemplo de la búsqueda panceta ("*bacon*") en la base de datos USDA volcada a MongoDB sin aplicar la separación de los campos de características

```
Search:"bacon"
Exit:
       C1       ,   C2   ,      C3       ,    C4      Description Food Group
Pork            , bacon  , rendered fat , cooked          Pork Products
Animal fat      , bacon grease                            Fats and Oils
Bacon           , meatless                              Legumes and Legume Products
Bacon           , beef sticks                           Sausages and Luncheon Meats
Turkey bacon    , unprepared                            Sausages and Luncheon Meats
Turkey bacon    , microwaved                            Sausages and Luncheon Meats
Canadian bacon  , unprepared                                  Pork Products
Pork            , cured  , bacon        , unprepared      Pork Products
```

Figura 1.2: Ejemplo de la búsqueda panceta ("*bacon*") en la base de datos USDA volcada a MongoDB aplicando la separación de los campos de características con unos pesos de 1, 2, 1, 1 para C1, C2, C3 y C4 respectivamente

En resumen, debido a la falta de estructuración de la información almacenada, no es posible asignar, de forma adecuada, pesos a cada uno de los campos, puesto que no se conoce donde se encuentran las palabras claves para encontrar un determinado ingrediente.

1.2. Objetivos

El objetivo principal de este proyecto es mejorar los resultados que se obtienen desde la web receteame.com al realizar búsquedas de ingredientes en la base de datos USDA almacenada en MongoDB.

A fin de conseguir el objetivo principal de este trabajo, se diseñará una arquitectura de búsqueda que aumente los aciertos a la hora de buscar los

```
Search:"red wine"
Exit:
        C1        ,    C2    ,  C3  ,        C4        Description Food Group
Vinegar           , red wine                              Spices and Herbs
Alcoholic beverage,   wine   , table, red                     Beverages
Alcoholic Beverage,   wine   , table, red Syrah               Beverages
Alcoholic Beverage,   wine   , table, red Barbera             Beverages
Alcoholic Beverage,   wine   , table, red Zinfandel           Beverages
Alcoholic Beverage,   wine   , table, red Claret              Beverages
Alcoholic Beverage,   wine   , table, red Lemberger           Beverages
Alcoholic Beverage,   wine   , table, red Sangiovese          Beverages
Alcoholic Beverage,   wine   , table, red Carignane           Beverages
Alcoholic Beverage,   wine   , table, red Burgundy            Beverages
Alcoholic Beverage,   wine   , table, red Gamay               Beverages
Alcoholic Beverage,   wine   , table, red Mouvedre            Beverages
Alcoholic Beverage,   wine   , table, red Merlot              Beverages
```

Figura 1.3: Ejemplo de la búsqueda vino tinto ("*red wine*") en la base de datos USDA volcada a MongoDB

ingredientes de una receta en la base de datos USDA y así poder acceder a sus características nutricionales. Como se ha comentado en la sección 1.1, MongoDB está diseñado de tal forma que nos permite aplicarle pesos a unos campos respecto a otros durante la búsqueda. Inicialmente los ingredientes son almacenados en la base de datos sin seguir una estructura concreta y no se obtienen los resultados esperados. Por lo tanto necesitamos un sistema capaz de explorar múltiples soluciones a la vez y que nos permita trabajar con los pesos de los campos de la base de datos. Una técnica que cubre con los requisitos comentados son los algoritmos genéticos. Por ello, utilizaremos un Algoritmo Genético (AG) que se encargue de encontrar la mejor disposición de los pesos sobre los diferentes campos de características. Por tanto, para obtener mejores resultados en la búsqueda de ingredientes, deberemos conseguir los siguientes subobjetivos:

Estudiar las características disponibles para el diseño del algoritmo genético. A la hora de diseñar un AG nos encontramos como existen diferentes propuestas que podemos seguir, por lo que inicialmente en el capítulo 2.1 analizaremos estas propuestas.

Diseñar un algoritmo genético. Después de haber estudiado los diferentes operadores disponibles para diseñar un AG, se obtendrá un AG que nos ayude a encontrar la mejor disposición de los pesos sobre los diferentes campos que caracterizan a un ingrediente.

Validar los resultados obtenidos por el algoritmo genético Cada modificación del AG será evaluada sobre la base de datos disponible para poder observar como influye en los resultados obtenidos.

Diseñar diferentes estrategias/heurísticas. Se diseñaran heurísticas que se apliquen a los resultados de la búsqueda en la base de datos, a fin de aumentar la calidad de los resultados.

Evaluación de los resultados obtenidos al aplicar las heurísticas frente a la base de datos disponible. Cada heurística empleada será evaluadas frente a la base de datos para medir su posible mejora en el proceso de búsqueda.

1.3. Estructura del trabajo

La memoria de este trabajo contiene dos partes. La primera parte consiste en el estado del arte, compuesto por los dos primeros capítulos, donde explicaremos detalladamente los conocimientos previos para la realización de este proyecto. En el primer capítulo se explicarán las diferentes técnicas de *soft computing* que utilizaremos en el algoritmo de búsqueda de este proyecto. El siguiente tratará sobre la base de datos que vamos a utilizar, MongoDB.

Por otro lado, la segunda parte aborda los cuatro últimos capítulos, donde se desarrolla la propuesta. En el tercer capítulo comentaremos la metodología empleada en el algoritmo de búsqueda. En el cuarto capítulo explicaremos la evolución en el diseño del algoritmo de búsqueda. Seguidamente, en el quinto capítulo aplicaremos heurísticas que nos ayuden a mejorar los resultados obtenidos en el paso anterior. Finalmente, el último capítulo estará dedicado a comentar las conclusiones finales de este proyecto, junto a las líneas futuras de actuación.

Parte I

Estado del arte

Capítulo 2

Técnicas soft computing

Las técnicas de *soft computing* hacen referencia a una serie de técnicas que pueden trabajar tolerando cierto nivel de imprecisión [Valero Cubas, 2010]. Éstas son capaces de obtener soluciones con bajo coste, manteniendo la robustez y flexibilidad necesarias [Zadeh, 1994]. Existen diversas técnicas *soft computing*: lógica difusa, razonamiento probabilístico, redes neuronales, algoritmos genéticos, etc. De estas técnicas sólo se han utilizado para realizar este proyecto los algoritmos genéticos y las redes neuronales, por lo que en las siguientes secciones se explicarán en mayor detalle.

2.1. Algoritmos genéticos

Los AG son técnicas adaptativas que se utilizan fundamentalmente para la resolución de problemas de búsqueda. Este algoritmo de búsqueda se basa en los procesos genéticos de los organismos vivos, imitando la teoría de la evolución biológica de Darwin para la resolución de problemas. A lo largo de diferentes generaciones, las poblaciones evolucionan con los principios de la selección natural o la supervivencia de los más fuertes.

En la década de los setenta, John Hollan y sus estudiantes en la Universidad de Michigan, desarrollaron los AG. El objetivo de estos nuevos algoritmos consistía en simular el proceso adaptativo de los sistemas naturales y diseñar sistemas artificiales que retuvieran los mecanismos importantes de éstos. Las características más importantes de estos algoritmos que los distinguen del resto son cuatro [Goldberg, 1989]:

- Utilizan una codificación de los parámetros a optimizar.

- No se centran en buscar una única solución por todo el espacio de soluciones, por lo que es más difícil que converja en óptimos locales.

- Emplean directamente una función objetivo, que nos indica como de buena es una posible solución respecto al problema a tratar.

- Utilizan reglas de transición probabilísticas.

Los AG, a pesar de ser computacionalmente simples, son muy eficaces. Esto es así debido a que en un espacio de búsqueda inicialmente desconocido son capaces de explotar la información disponible, lo que resulta muy útil cuando tenemos espacios de búsqueda amplios, complejos y poco conocidos.

La solución candidata, en el algoritmo, está codificada en una cadena numérica o binaria, denominada cromosoma. Los AG simulan la evolución genética de una población de individuos (conjunto de cromosomas), empleando diferentes operadores de recombinación (cruce y mutación) y selección de los individuos que sobreviven a la siguiente generación o a aquellos que le aplicaremos los operadores de recombinación. El operador de cruce consiste en mezclar dos cromosomas, intercambiando información genética, para obtener uno nuevo e intentar que este sea mejor que sus antecesores. En cambio, el operador de mutación consiste en alterar aleatoriamente el valor de alguna

de las características del cromosoma, con el fin de preservar la diversidad genética y así evitar la convergencia en óptimos locales. A cada individuo de la población se le asigna un valor en función de lo cerca que esté de la mejor solución, a este valor se le llama *fitness*. En la figura 2.1 podemos ver un esquema de la forma de actuar de un AG. Calcular el valor del *fitness* en ocasiones puede resultar costoso, por eso suelen utilizarse otras técnicas que nos ayuden a aproximar el valor de este *fitness* como usar máquinas de soporte vectorial o redes neuronales.

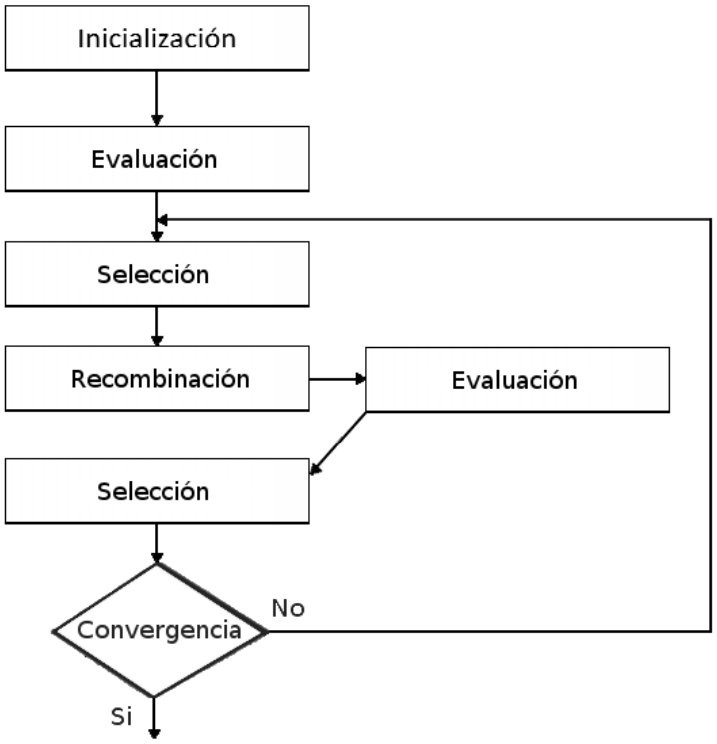


Figura 2.1: Comportamiento de un algoritmo genético

En las siguientes secciones detallaremos cada uno de los aspectos del AG, tanto su codificación como los diferentes operadores que utiliza.

2.1.1. Codificación

Como ya se ha comentado en la sección 2.1, los AG trabajan con una codificación de las posibles soluciones llamadas cromosomas. Estos cromosomas de alguna manera deben contener una información que les represente. Así, normalmente se representan como un conjunto de parámetros (genes). Existen varios tipos de codificación, las más utilizadas son la codificación binaria y la numérica. La elección de que tipo de codificación hay que utilizar dependerá del problema a resolver. Además, diversos resultados teóricos demuestran que suele ser más efectivo usar alfabetos pequeños que grandes [Goldberg, 1991]. La mayoría de las veces, una codificación correcta es la clave de una buena resolución del problema.

Así pues hay que realizar un estudio previo para ver cual será la mejor codificación en el problema que intentemos resolver.

Codificación binaria. Es la más usada, debido a que inicialmente los primeros AG utilizaban este tipo de codificación. Además, la utilización de alfabetos pequeños para la codificación puede ser más efectiva que la utilización de alfabetos grandes [Goldberg, 1991]. Cada cromosoma está compuesto por una cadena de bits (1s y 0s), esto hace que se agilicen los operadores de mutación o cruce, pero sin embargo no resulta útil para la mayoría de los problemas.

Codificación numérica. Cada cromosoma está compuesto por una cadena de números, por lo que los operadores de cruce y mutación son más costosos. Suelen utilizarse en problemas de ordenación o de búsqueda.

2.1.2. Métodos de selección

En un AG es necesario seleccionar a los individuos, ya sea para aplicarles los operadores de recombinación o para que sobrevivan en la siguiente generación. Existen diferentes formas de realizar esta selección, todo dependerá de la estrategia que queramos seguir. Normalmente se suele usar el operador de selección para mejorar la calidad de la población, para ello los individuos (cromosomas) de mayor calidad, es decir, con un mejor *fitness*, tendrán una mayor probabilidad de ser seleccionados, centrándose en regiones prometedoras en el espacio de búsqueda.

Por otro lado, si buscamos que exista diversidad genética, no debemos escoger únicamente aquellos individuos de mayor calidad, ya que el cruce de dos individuos con peor calidad no significa que su descendiente tenga también una mala calidad. En el caso que la selección únicamente se centre en los mejores individuos diremos que se trata de una selección elitista.

El mecanismo de selección que utilicemos influirá en la diversidad genética. Normalmente, los mejores individuos reemplazarán a los peores y, con el tiempo, su material genético desaparecerá. Existen numerosos mecanismos de selección, los más comunes pueden encontrarse en los siguientes trabajos [Goldberg and Deb, 1991][Blickle and Thiele, 1995]. Seguidamente, explicaremos algunos de ellos.

Selección elitista. Esta selección consiste en escoger a los mejores individuos. Esto puede resultar útil cuando queremos que no se pierda el mejor individuo. Se puede combinar con otros mecanismos de selección para que guarde los primeros n mejores individuos y para los siguientes m realizar cualquier otro mecanismo de selección (siendo $n + m$ el número total de individuos que deseamos seleccionar).

Selección por rueda de ruleta. Éste es un mecanismo del grupo de selección proporcional, donde cada individuo tiene una mayor probabilidad de salir cuanto mayor sea su *fitness* o aptitud. Cada uno de los individuos ocupará un espacio en la ruleta dependiendo de su aptitud en consideración con la aptitud de todos los individuos que participen. Estos son incorporados en la ruleta de manera aleatoria y después se lanza la ruleta y selecciona uno de los individuos. Si quisiéramos seleccionar más de un individuo, podemos utilizar el mismo sistema tantas veces como individuos queramos o usar la selección estocástica universal, muy similar al de la ruleta pero seleccionando n individuos a la vez. En la figura 2.2(a) podemos observar un ejemplo de

esta selección, en el que se muestra la probabilidad que tiene cada individuo en ser seleccionado. En este caso la probabilidad p_s de que un individuo C_i sea seleccionado es:

$$p_s(C_i) = \frac{aptitud(C_i)}{\sum_{j=1}^{N} aptitud(C_j)} \tag{2.1}$$

Selección por rango. En este caso previamente hay que ordenar la población por su aptitud. Para ponderar a los individuos utilizamos la ecuación 2.2, donde utilizamos la posición de cada individuo, ordenados previamente por la aptitud ($ranking$). Ahora, la probabilidad de ser seleccionado ya no depende directamente de su aptitud, sino de la posición que ocupe en la ordenación inicial. El usuario previamente debe definir los parámetros max y min, donde $max + min = 2$ y $max \geq min$. En la ecuación 2.3 podemos observar como varia la ecuación en función del valor de $max - min$ y de la posición del $ranking$. Cuanto mayor valor de $ranking$ menor probabilidad de ser seleccionado. A su vez, cuando mayor es la diferencia entre max y min habrá una mayor variación entre las probabilidades. En la figura 2.2(b) podemos observar un ejemplo de la utilización de este método de selección, siendo N el número de individuos a seleccionar y $ranking(x)$ la posición del individuo x en la lista ordenada por su aptitud. Se puede observar como existe una mayor variación entre las probabilidades obtenidas en $P'(x)$ que en las probabilidades obtenidas en $P(x)$, esto es debido a que la diferencia de max y min en $P'(x)$ es de 1 mientras que la diferencia en $P(x)$ es solamente de 0.2.

$$P(x) = \frac{1}{N}(max - \frac{(max - min)(ranking(x) - 1)}{N - 1}) \tag{2.2}$$

$$P(x) = \frac{max}{N} - \frac{ranking(x) - 1}{N * (N - 1)} * (max - min) \tag{2.3}$$

Selección por torneo. Este método de selección consiste en escoger de forma aleatoria un número de individuos de la población. Dentro de esta selección aleatoria, los individuos compiten entre si a través de su aptitud, considerando ganadores aquellos k individuos que mejor aptitud tengan, es decir, serán elegidos únicamente los k mejores individuos de la selección inicial aleatoria.

fitness(c_i)	$P(c_i)$
5	**0.19**
2	**0.08**
19	**0.73**
Σ	1

(a) Ejemplo de la selección por rueda de ruleta

Datos necesarios para la selección por rango:
P(x) -> max = 1.1, min = 0.9
P'(x) -> max = 1.5, min = 0.5

fitness(x)	ranking(x)	P(x)	P'(x)
5	2	**0.33**	**0.33**
2	3	**0.30**	**0.17**
19	1	**0.37**	**0.50**
Σ		1	1

(b) ejemplo de la seleccion por rango

Figura 2.2: Comparativa de la selección por rango y la selección proporcional rueda de ruleta

2.1.3. Operador de cruce

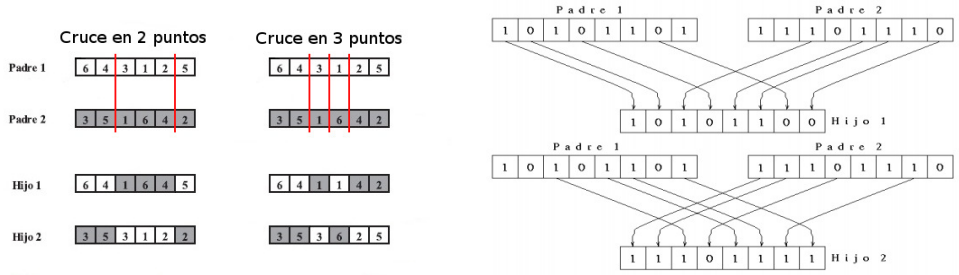
Se trata de uno de los operadores más importantes. Consiste en un intercambio del material genético entre cromosomas elegidos mediante algún método de selección como los detallados en la sección 2.1.2. Lo que se intenta conseguir mediante este operador es tratar de mejorar la población procurando reunir las mejores características de los individuos seleccionados. Normalmente no toda la población es cruzada, sino que estos tienen una probabilidad p_i de ser seleccionados. Cabe destacar que el cruce de los individuos no tiene porque dar un descendiente con mayor aptitud que sus antecesores. Como en la naturaleza, dos individuos que tienen una buena aptitud pueden dar lugar a descendientes que no la tengan. Al cruzar los individuos pueden obtenerse diversos descendientes, de los cuales podemos elegir si quedarnos todos o una selección de estos.

Existen números métodos de cruce, donde la manera de aplicarlos depende en gran medida del tipo de codificación del problema. A continuación citaremos alguno ejemplos de distintos métodos de cruce, donde asumiremos como progenitores a dos cromosomas $C_1 = (c_1^1, ..., c_n^1)$ y $C_2 = (c_1^2, ..., c_n^2)$, y también asumiremos como descendiente al cromosoma $H = (h_1, ..., h_n)$. Los métodos de cruce que vamos a citar son el cruce en n-puntos y el cruce uniforme, frecuentemente usados con la codificación binaria, y además citaremos el cruce plano y el cruce BLX-α, que son utilizados por problemas con codificación real.

Cruce en n-puntos. Consiste en cortar los dos cromosomas que queramos cruzar en n puntos y después mezclar los cortes [Eshelman et al., 1989]. Cuando $n = 1$ decimos que es un cruce simple, ya que, al hacer un único

corte, es la manera más sencilla de cruzar dos individuos. Este es un método que suele utilizarse con bastante frecuencia en la codificación binaria. En la figura 2.3(a) podemos observar como se realiza este método de cruce.

Cruce uniforme. El valor de cada gen del descendiente h_i se selecciona de manera aleatoria de sus progenitores, por lo que valdría c_i^1 o c_i^2, así sucesivamente [Sywerda, 1989]. En la figura 2.3(b) vemos como de manera aleatoria formamos dos descendientes.



(a) Cruce por n-puntos (b) Cruce uniforme

Figura 2.3: Cruces usados normalmente en codificación binaria

Cruce plano o de sombrero de copa. El valor de cada gen del descendiente consiste en un valor aleatorio comprendido entre el valor de sus progenitores, es decir, si suponemos que $c_i^1 \leq c_i^2$ el valor del gen del hijo será $h_i \in [c_i^1, c_i^2]$ [Radcliffe, 1991].

Cruce BLX-α. Este método consiste en una pequeña modificación del método de cruce plano. Esta vez el valor de cada gen del descendiente tendrá un valor comprendido entre los valores de los genes de los progenitores, pero a estos se le asigna un incremento o decremento para preservar la diversidad genética, es decir, $h_i \in [c_{min} - \alpha I, c_{max} + \alpha I]$, siendo c_{min} y c_{max} el valor mínimo y máximo del gen i de los progenitores c_i^1 y c_i^2 [Eshelman and Schaffer, 1993]. Al valor I que utilizaremos para incrementar o decrementar el rango de valores se regulará mediante un α. Cuanto más aumentemos este valor mayor será la diversidad genética y si este α tuviera el valor 0 estaríamos usando el método de cruce plano. Puede que al gen h_i se le asigne un valor que supere la codificación, por lo que deberíamos corregir este valor. Para ilustrar este cruce en la figura 2.4 podemos observar un ejemplo de la utilización de este método. En este ejemplo a_i y b_i son los límites de la codificación.

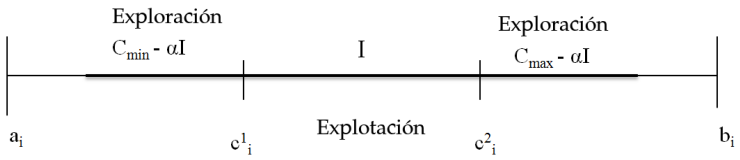


Figura 2.4: Método de cruce BLX-α

2.1.4. Operador de mutación

El operador de mutación permite que un AG no converja prematuramente hacia óptimos locales, ya que, a través de la mutación, preservamos la diversidad genética de la población. Cabe destacar que no se debe abusar de la mutación, ya que abusar de ésta puede provocar que nuestro AG actué como una simple búsqueda aleatoria. Por ello es recomendable usar antes otros mecanismos para preservar la diversidad, como aumentar el tamaño de la población o asegurar una aleatoriedad en la población inicial.

Este operador es utilizado de manera aleatoria sobre ciertos individuos, modificando alguno de sus genes según una probabilidad de mutación establecida previamente. Existen muchas formas de realizar esta mutación, la codificación empleada contribuye a elegir la más apropiada. Seguidamente vamos a explicar dos métodos propios de la codificación binaria, aunque cabe destacar que también se pueden utilizar en la codificación real.

Mutación simple. De entre todos los genes del cromosoma, escogemos alguno de ellos de manera aleatoria, es decir, puede seleccionarse un único gen o varios a la vez para ser mutados. Aquellos genes que hayan sido escogidos para mutar cambiarán su valor, si el valor era "1", se le asignará un "0", y viceversa, podemos ver un ejemplo en la figura 2.5. En el caso de la codificación real, se utilizaría la mutación aleatoria que explicaremos más adelante.

Mutación por intercambio recíproco. Esta mutación consiste en cambiar dos genes dentro del mismo individuo, haciendo un intercambio de información genética dentro del individuo [Herdy, 1991].

A continuación vamos a explicar dos de los métodos utilizados con la codificación real.

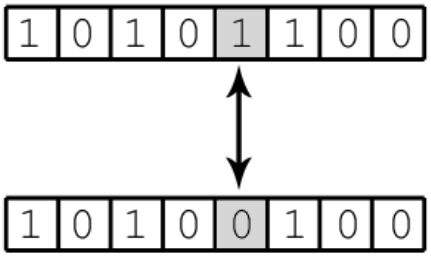


Figura 2.5: Mutación en codificación binaria

Mutación aleatoria. Esta mutación trata de obtener un nuevo gen, donde éste será un valor aleatorio comprendido entre dos números reales, que corresponden a un máximo y un mínimo valor que puede tomar este gen, fijados al principio del AG [Michalewicz, 1992].

Mutación no uniforme. Para aplicar este método de mutación tenemos que seguir la ecuación 2.4, donde t indica la generación en la que se encuentra, g_{max} representa el número máximo de generaciones, a_i es el valor mínimo que puede tomar el gen que vamos a mutar, mientras que b_i es el valor máximo y c_i es el valor actual del gen. La ecuación 2.4 utiliza una función representada por la ecuación 2.5, donde r es un número aleatorio que oscila entre [0, 1] y el parámetro b es elegido por el usuario, que indica la dependencia sobre el número de iteraciones. Conforme pasan las generaciones en el AG, el valor de la mutación decrece, por lo que este método de mutación permite una búsqueda más localizada cuanto mayor sea la generación en la que nos encontramos [Michalewicz, 1992].

$$c_i' = \begin{cases} c_i + \triangle(t, b_i - c_i) & \text{si} \quad \tau = 0 \\ c_i - \triangle(t, c_i - a_i) & \text{si} \quad \tau = 1 \end{cases} \tag{2.4}$$

$$\triangle(t, y) = y(1 - r^{(1-\frac{t}{g_{max}})}) \tag{2.5}$$

2.2. Redes neuronales artificiales

Las Redes Neuronales Artificiales (RNA) son unas herramientas capaces de establecer relaciones entre los datos de entrada y de salida, sin ningún tipo de conocimiento previo. Estas RNA están inspiradas en la forma de funcionamiento de las redes neuronales biológicas, donde un conjunto de neuronas actúan entre si para generar un estimulo.

El modelo básico de estas RNA es la neurona o nodo. Esta neurona está compuesta de los siguientes elementos:

Conjunto de entradas. Este conjunto de entradas puede ser de otras neuronas o de una fuente externa de datos (x_j). Cada una de estas entradas tienen un peso asociado (θ_{ij}), que durante un proceso llamado aprendizaje se va modificando.

Regla de propagación. Determina el valor de la entrada de la neurona (y_i) mediante la suma de las entradas ponderadas por el peso asociado (θ_{ij}), podemos observarlo en la ecuación 2.7.

Función de activación. Determina la salida de la neurona (s_i), a partir de aplicar una de las funciones ($g(y_i)$) tales como funciones sigmoidales, las logísticas (ecuación 2.8) y las tangenciales (ecuación 2.9).

Salida de la neurona. Las neuronas a su vez tienen una salida que forma parte de la entrada de otra neurona, por lo que por cada rama lleva un peso asociado (θ_i).

$$y_i = \sum_j \theta_{ij} * x_j \tag{2.6}$$

$$s_i = g(y_i) \tag{2.7}$$

$$g(x) = \frac{1}{1 + exp(-x)} \tag{2.8}$$

$$g(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \tag{2.9}$$

En la figura 2.6 podemos observar el ejemplo de una de las neuronas que compondría una RNA.

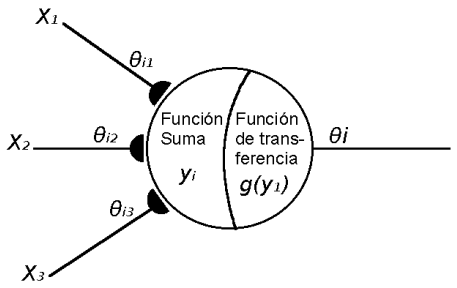


Figura 2.6: Modelo básico de la red neuronal (neurona)

Las RNA son capaces de dar respuestas rápidamente, además nos puede ofrecer resultados para muestras que son desconocidas para nuestra red. Estas RNA necesitan entrenarse con un conjunto de muestras para aprender sobre el problema que está intentando solucionar. Una vez entrenada la RNA, se establecen correlaciones matemáticas entre las neuronas. Normalmente es necesario un gran conjunto de muestras y también suele tardar bastante en calcular estas correlaciones matemáticas [Ripley, 1996].

En las RNA existen dos fases para su modelación:

Fase de entrenamiento. Esta fase se utiliza para calcular los pesos de las neuronas dentro de la RNA. Existen dos maneras de realizarlo: de manera supervisada o no supervisada. El entrenamiento de manera supervisada como los de manera no supervisada disponen de unos datos de entrada que ayudan a entrenar la RNA. Pero la manera supervisada a diferencia de la no supervisada dispone también de la salida esperada para estos datos de entrada, por lo que ajusta el peso de las neuronas para minimizar el error obtenido por la salida de la RNA y el esperado.

Fase de prueba. La RNA diseñada en la fase anterior, puede que se ajuste demasiado a las particularidades de los datos de entrenamiento (sobre-ajuste). Para evitar este sobre-ajuste, se suele utilizar unos datos diferentes a los utilizados para entrenar la RNA, para controlar el aprendizaje. Estos datos se suelen llamar datos de validación.

A lo largo del tiempo, las RNA han demostrado funcionar correctamente para la aproximación de diferentes funciones [Jin, 2005]. Por tanto, pueden aplicarse a la aproximación de funciones de aptitud o *fitness*, cuando el cálculo de éstas sea muy costoso. Para este tipo de problemas se han utilizado ampliamente los perceptrones multicapa que explicaremos a continuación.

2.2.1. Perceptrón multicapa

El perceptrón multicapa es una RNA formada por múltiples capas, lo que le permite resolver problemas que no son linealmente separables. Esta red fue creada por Frank Rosenblatt [Rosenblatt, 1962], y es una de las RNA más conocidas y empleadas usando el aprendizaje supervisado.

Las neuronas del perceptrón multicapa se agrupan en diferentes capas, como mínimo tendremos una capa de entrada, una capa oculta y una capa de salida, a este perceptrón se le llamara perceptrón de dos capas (contamos únicamente las capas ocultas y la capa de salida). Las entradas de las neuronas corresponden a las salidas de las neuronas de la capa anterior, a excepción de las neuronas de la capa de entrada que corresponden a los datos de entrada del problema a resolver. La salida de las neuronas de la capa de salida corresponden al resultado devuelto por la RNA sobre los datos de entrada introducidos. Dicho esto podemos ver que el número de neuronas en la capa de salida y en la capa de entrada dependerá de las características del problema que vamos a tratar. Sin embargo debemos llevar un estudio para determinar el numero de capas ocultas y el número de capas de cada capa oculta que nos proporcione un mejor resultado para resolver el problema.

El principal problema de este tipo de RNA es de obtener sobre-ajuste, es decir, que se ajusta demasiado a las peculiaridades de los datos con los que ha sido entrenado. Para resolver este problema hay que detectar cuándo es necesario dejar de entrenar la red o cuándo debemos volver a hacerlo

En la ecuación 2.10 podemos observar como se calcularía la salida del perceptrón de dos capas, siendo J el número de neuronas en la capa de entrada, I el número de neuronas en la capa oculta, K el número de neuronas en la capa de salida y siendo las funciones de activación $g(\Delta)$ para la capa oculta y $f(\Delta)$ para la capa de salida.

$$s_k^2 = f(\sum_{i=0}^{I} \theta_{ki}^2 * g(\sum_{j=0}^{J} \theta_{ij}^1 * x_j)) \tag{2.10}$$

En la figura 2.7 se muestra el ejemplo del perceptrón de dos capas de la ecuación anterior. Podemos observar que este ejemplo dispone de cinco neuronas en la capa de entrada (x_j), tres neuronas en la capa oculta (s_i^1) y otras tres en la capa de salida (s_k^2). Los pesos de se están representando mediante las conexiones de las neuronas (θ_{ij}^1 y θ_{ki}^2).

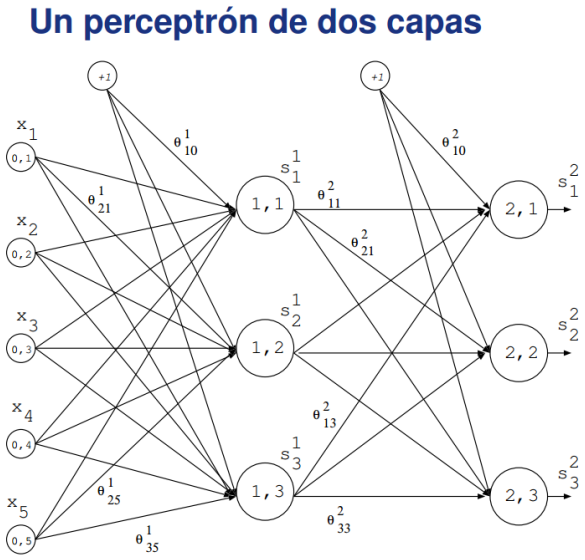


Figura 2.7: Ejemplo de la arquitectura del perceptrón de dos capas

En el aprendizaje de RNA existe la posibilidad de regular la velocidad de aprendizaje utilizando el factor de aprendizaje ($0 < \alpha \leq 1$). En la ecuación 2.11 podemos observar el incremento del camino de los pesos de una neurona de la capa oculta la capa de salida del ejemplo anterior, siendo la salida de la neurona j es s_j^1, d_k^2 la salida de la neurona k esperada, s_k^2 el valor de la neurona k y θ_{ki}^2 el valor actual del camino entre las neuronas j y k.

$$incremento(\theta_{ki}^2) = \alpha * s_j^1 * (d_k^2 - s_k^2) \tag{2.11}$$

Existe otro factor llamado momentum, que ayuda a acelerar la convergencia del error hacía el mínimo de la función ($0 < \mu \leq 1$). Se basa en suavizar las oscilaciones en la trayectoria hacia la convergencia, al incrementar este momentum las oscilaciones se reducen. En la ecuación 2.12 vemos como se calcularía el incremento de los pesos usando este factor. Este nuevo factor permite cambios suaves debido a que incluye información del anterior cambio.

$$incremento(\theta_{ki}^2) = \alpha * s_j^1 * (d_k^2 - s_k^2) + \mu * (\theta_{ki}^2 - \theta_{ki_{anterior}}^2) \tag{2.12}$$

Capítulo 3

MongoDB

En este capítulo hablaremos de las características que más nos interesan de la base de datos MongoDB[1]. Entre éstas, explicaremos como se organizan los datos dentro de la base de datos y los diferentes tipos de búsqueda que existen. En especial entraremos más en detalle en la búsqueda por texto.

Existen una amplia diversidad de base de datos que podemos utilizar, pero entre todas estas decidimos elegir MongoDB debido a sus características. MongoDB es una base de datos NoSQL, esto hace que no guarde los datos en tablas como las bases de datos relacionales, sino que utiliza una estructura de almacenamiento con documentos de tipo JSON (clave-valor) haciendo que la integración de los datos con las aplicaciones sea fácil y rápida. Las bases de datos NoSQL fueron desarrolladas en el año 2000 para hacer frente a las limitaciones de las bases de datos relacionales. Estas bases de datos están construidas para permitir insertar datos sin tener que usar ningún esquema predefinido, haciendo que sea fácil hacer cambios en la base de datos y necesitando menos tiempo para administrarla.

En la base de datos MongoDB, los documentos se organizan en colecciones. Éstas pueden tener un número indeterminado de documentos. En la figura 3.1 se puede observar un ejemplo de como son introducidos, en la misma colección de datos, dos datos que comparten algunos de los campos. Podemos observar como no es necesario que los datos introducidos sigan una estructura ordenada. El primer ingrediente que insertamos únicamente está compuesto por el nombre completo del ingrediente (*long_desc*), y por sus campos de características (*c1* y *c2*), mientras que el segundo ingrediente, a diferencia del anterior posee un campo de característica más que el anterior

[1]Podemos encontrar esta base de datos en www.mongodb.org.

(*c3*), además también posee un campo que indica a que grupo de comidas pertenece (*food_group*). Podemos observar que los campos de características coinciden con la división por comas del nombre completo del ingrediente, este mismo modelo es el que sigue la base de datos de nuestro proyecto.

```
db.nutrition_test.insert(
    {
        long_desc:"Papayas, raw",
        c1:"Papayas",
        c2:"raw"
    })

db.nutrition_test.insert(
    {
        long_desc:"Babyfood, fruit, papaya with tapioca",
        c1:"Babyfood",
        c2:"fruit",
        c3:"papaya with tapioca",
        food_group:"Baby Foods"
    })

db.nutrition_test.find()
    {
        "_id" : ObjectId("52f602d787945c344bb4bda5"),
        long_desc:"Papayas, raw",
        c1:"Papayas",
        c2:"raw"
    }
    {
        "_id" : ObjectId("52f602d845454c344bb4bda5"),
        long_desc:"Babyfood, fruit, papaya with tapioca",
        c1:"Babyfood",
        c2:"fruit",
        c3:"papaya with tapioca",
        food_group:"Baby Foods"
    }
```

Figura 3.1: Ejemplo de inserción de datos en la base de datos MongoDB.

En MongoDB podemos hacer diferentes tipos de búsquedas, de entre estas, podemos encontrar la típica búsqueda que permite buscar los objetos que tienen un valor en una determinada clave. Por otro lado, MongoDB tiene una forma de indicar qué claves son cadenas de texto, mediante la creación de índices, y después permite realizar una búsqueda de texto que buscará sobre todas las claves existentes en el índice que estén indicadas como cadenas de texto. Estos índices pueden incorporar unos pesos asociados a las claves, dando prioridad a unas claves respecto a otras, cuanto mayor sea el peso mayor prioridad tendrá. Más adelante explicaremos detalladamente este tipo de búsqueda.

3.1. Búsqueda por texto

Una de las búsquedas que realiza la base de datos MongoDB es la de búsqueda de texto. Este tipo de búsqueda trata de realizar una búsqueda sobre los índices de texto, que han sido creados sobre uno o más campos (claves) de la colección. Los índices de texto nos indicarán que claves son o no son campos de texto. En estos índices podemos indicar la siguiente información:

Claves afectadas. Se trata de las claves sobre las que queremos realizar una búsqueda de texto.

Pesos de cada clave afectada. Se trata de una de las propiedades que más nos interesa de esta búsqueda. Aplicando diferentes pesos a las claves podemos crear prioridades entre ellas. El rango de pesos que se pueden aplicar son de [1, 99999]. Por ejemplo si tuviéramos tres claves con los siguientes pesos, 2 para la clave c_1, 1 para la clave c_2 y 4 para la clave c_3, diremos que c_3 es cuatro veces más prioritaria que c_2 y dos veces más prioritaria que c_1, mientras que c_1 es dos veces más prioritaria que c_2. Si no introducimos ningún peso a las claves afectadas, supondrá que todas tienen la misma prioridad.

Idioma por defecto. Podemos indicar a MongoDB el idioma por defecto que estamos utilizando en la base de datos. Inicialmente el idioma que utiliza es el inglés. MongoDB utiliza este lenguaje para determinar aquellas palabras que no aportan ningún significado (artículos, preposiciones, pronombres, ...) y las normas necesarias para poder calcular la derivada de las palabras de nuestra búsqueda, *stemming*. Con esto podemos mejorar las búsquedas de texto sobre nuestra base de datos.

En la figura 3.2 se puede comprobar como se crea un índice de texto, y se aplica una búsqueda sobre los ingredientes introducidos en la figura 3.1. Podemos observar como si le aplicamos mayor prioridad al primer campo de característica que al tercer campo, obtendremos como solución el resultado esperado al realizar la búsqueda, mientras que en la figura 3.3 se puede observar que si le aplicamos mayor prioridad al tercer campo no obtendríamos el resultado esperado.

```
db.nutrition_test.createIndex({ c1:"text", c2:"text", c3:"text" },
                              { weights:{ c1:3, c2:1, c3:2 } },
                              { default_language:"english" })
db.nutrition_test.runCommand("text", {search:"papaya"})
{
    {
        "score" : 3,
        "obj" : { "_id" : ObjectId("5565c7f07fa1bd26b8e68685"),
                  "long_desc" : "Papayas, raw",
                  "c1" : "Papayas", "c2" : "raw"  }
    },{
        "score" : 1.5,
        "obj" : { "_id" : ObjectId("5565c8177fa1bd26b8e68686"),
                  "long_desc" : "Babyfood, fruit, papaya with tapioca",
                  "c1" : "Babyfood", "c2" : "fruit",
                  "c3" : "papaya with tapioca",
                  "food_group" : "Baby Foods"     }
    }
}
```

Figura 3.2: Ejemplo de la creación de un índice de texto y una búsqueda donde obtenemos el resultado esperado

```
db.nutrition_test.dropIndexes()
db.nutrition_test.createIndex({ c1:"text", c2:"text", c3:"text" },
                              { weights:{ c1:2, c2:1, c3:3 } },
                              { default_language:"english" })
db.nutrition_test.runCommand("text", {search:"papaya"})
{
    {
        "score" : 2.25,
        "obj" : { "_id" : ObjectId("5565c8177fa1bd26b8e68686"),
                  "long_desc" : "Babyfood, fruit, papaya with tapioca",
                  "c1" : "Babyfood", "c2" : "fruit",
                  "c3" : "papaya with tapioca",
                  "food_group" : "Baby Foods"     }
    },{
        "score" : 2,
        "obj" : { "_id" : ObjectId("5565c7f07fa1bd26b8e68685"),
                  "long_desc" : "Papayas, raw",
                  "c1" : "Papayas", "c2" : "raw"  }
    }
}
```

Figura 3.3: Ejemplo de la creación de un índice de texto y una búsqueda donde no obtenemos un resultado esperado

Después de haber visto este tipo de búsqueda tan peculiar en la base de datos MongoDB decidimos aplicarlo en el proyecto, ya que disponemos de diferentes campos de características y esto nos puede ayudar a encontrar una buena solución. Por lo tanto crearemos una base de datos MongoDB donde los ingredientes tengan once campos de características. Después crearemos un índice indicando que los once campos de características corresponden a campos de texto. Por último, a través del algoritmo genético buscaremos la combinación de pesos asociados a los campos de texto que nos proporcione el mejor resultado a la hora de encontrar los ingredientes.

Parte II

Propuesta

Capítulo 4

Diseño del algoritmo genético

Como ya hemos comentado, tenemos una base de datos que esta compuesta por ingredientes que poseen varios campos de características. Con la ayuda del algoritmo genético buscamos encontrar una combinación de pesos para cada una de los campos de características que nos proporcione la mejor ordenación de prioridad entre los campos de características, a fin de obtener el mayor número de aciertos a la hora de buscar un ingrediente.

Tal y como comentamos en la sección 2.1, los AG se basan en la evolución natural, donde partiendo de una población de soluciones iniciales y mediante un proceso evolutivo guiado por decisiones probabilísticas, se espera obtener un buen individuo. Esta simulación de evolución genética necesita definir inicialmente los diferentes operadores de recombinación y selección de los individuos. Así en este capítulo se define los diferentes operadores de recombinación y selección, además se explica la codificación empleada y la función de la aptitud diseñada para evaluarlos.

En primer lugar se presentará la codificación empleada, posteriormente pasaremos a explicar la función de aptitud diseñada para valorar estas soluciones y finalmente mostraremos los diferentes operadores empleados en el AG diseñado para este proyecto.

4.1. Codificación de los individuos

Como dijimos en la sección 1.1, debemos encontrar una combinación de once pesos, cada uno para un campo de la base de datos, que nos proporcione el mayor número de aciertos de búsquedas, al establecer de manera correcta la prioridad entre las distintas características que definen un ingrediente. Decidimos que la mejor codificación que podíamos utilizar seria la codificación real, tomando como valor de cada gen el valor de cada peso de cada campo de la base de datos que queramos establecer. Dicho esto, una posible codificación de un individuo en este problema seria $I_0 = (c_1, c_2, ..., c_{11})$, donde $c_i \in [V_{min}, V_{max}]$, $\forall_{1 \leq i \leq 11}$, siendo c_i los distintos pesos de los campos de características y V_{min} y V_{max} el valor mínimo y máximo alcanzable por los pesos. Podemos observar en la figura 4.1 un ejemplo de estos individuos.

c1	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**	**c10**	**c11**
1	6521	25	6435	216	61	65	484	9998	849	88

Figura 4.1: Ejemplo de la codificación de un individuo, donde cada gen corresponderá a un campo de característica en la base de datos MongoDB

Debido a que a las medidas de los campos de la base de datos MongoDB se le pueden asignar valores desde 1 hasta 99999. Inicialmente utilizaremos $V_{min} = 1$ y $V_{max} = 99999$ para la codificación empleada en el AG.

4.2. Cálculo de la aptitud o fitness

Debemos diseñar una función que nos indique como de buenos son los diferentes conjuntos de pesos, y el AG debe maximizar el valor de esta función durante la evolución. Sabemos que la búsqueda de un ingrediente en la base de datos nos devuelve una lista de posibles soluciones, que como máximo serán cien, pero únicamente nos interesa la solución correcta. Por lo tanto, vamos a trabajar con un conjunto de cien ingredientes, donde sabemos exactamente cual es su mejor resultado. Así, buscaremos en la base de datos cien ingredientes distintos, elegidos de manera aleatoria, y almacenaremos la posición en la que aparece el resultado esperado (x_i), 0 si aparece en la primera posición, 1 en la segunda y así sucesivamente. Si diera el caso que no apareciera le daríamos el valor de diez millones. Podemos observar el cálculo de la aptitud en la ecuación 4.1, siendo N el número de ingredientes empleados y

x_i la posición del resultado esperado en el conjunto de posibles soluciones.

$$x_i = \begin{cases} [0, 99] & \text{, sí se encuentra en la búsqueda} \\ 10000000 & \text{, sí no se encuentra en la búsqueda} \end{cases}$$

$$aptitud = \frac{1}{\sum_{i=0}^{N-1} x_i + 1} \tag{4.1}$$

A continuación vamos a observar un ejemplo de la aplicación de este cálculo de la aptitud, realizando previamente la búsqueda de dos ingredientes, tal y como podemos observar en la figura 4.2. El primer ingrediente se trata de *bacon*, y como podemos observar el resultado esperado se encuentra en la segunda posición. El segundo ingrediente se trata de la *papaya* y también podemos observar como el resultado esperado se encuentra en la segunda posición. Dicho esto y aplicando la ecuación 4.1, obtenemos que, siendo $N = 2$ el número de ingredientes y $x_1 = x_2 = 1$ la posición de los resultados esperados de los ingredientes (ambos se encuentran en la segunda posición), tenemos que la *aptitud* de esta solución es $\frac{1}{2+1} = 0{,}33$.

```
Search:"bacon"
Exit:
      C1      ,    C2    ,     C3      ,    C4      Description Food Group
Bacon         , meatless                            Legumes and Legume
Pork          , bacon    , rendered fat, cooked     Pork Products
Pork          , cured    , bacon       , unprepared Pork Products
Turkey bacon  , unprepared                          Sausages & Luncheon Meats
Canadian bacon, unprepared                          Pork Products
Turkey bacon  , microwaved                          Sausages & Luncheon Meats
Animal fat    , bacon grease                        Fats and Oils


Search:"papaya"
Exit:
      C1     ,    C2   ,     C3      ,    C4         Description Food Group
Papaya       , canned  , heavy syrup , drained      Fruits and Fruit Juices
Papayas      , raw                                  Fruits and Fruit Juices
Papaya nectar, canned                               Fruits and Fruit Juices
Fruit salad  , tropical, canned      , heavy syrup  Fruits and Fruit Juices
Babyfood     , fruit   , papaya with tapioca        Baby Foods
```

Figura 4.2: Búsqueda de los ingredientes panceta ("*bacon*") y papaya en la base de datos para realizar posteriormente el cálculo de la aptitud

En la figura 4.3 se puede observar otro ejemplo de la búsqueda de dos ingredientes, esta vez se tratan de los ingredientes *bread* y *apple*. Como se puede observar el ingrediente *bread* su solución esperada se encuentra en la quinta posición, mientras que el ingrediente *apple* se encuentra su solución esperada en la onceava posición. Una vez que sabemos las posiciones, utilizando la ecuación 4.1 podemos calcular rápidamente su *aptitud* que tiene un valor de $\frac{1}{4+10+1} = \frac{1}{15} = 0{,}067$.

```
1  Search:"bread"
2  Exit:
3        C1        ,      C2      ,    C3    ,    C4        Description Food Group
4  Bread stuffing , bread        , dry mix                  Baked Products
5  Bread stuffing , bread        , dry mix , prepared       Baked Products
6  Bread          , pan dulce    , sweet                    Baked Products
7  Bread          , whole wheat , frozen                    Baked Products
8  Bread          , wheat                                   Baked Products
9
10
11 Search:"apple"
12 Exit:
13      C1     ,      C2                    ,    C3        Description Food Group
14 Croissants , apple                                      Baked Products
15 Strudel    , apple                                      Baked Products
16 Babyfood   , apples , dices              , toddler      Baby Foods
17 Babyfood   , juice  , apple                             Baby Foods
18 Babyfood   , juice  , apple and peach                   Baby Foods
19 Babyfood   , juice  , apple and plum                    Baby Foods
20 Babyfood   , juice  , apple and prune                   Baby Foods
21 Babyfood   , juice  , orange and apple                  Baby Foods
22 Babyfood   , juice  , apple and grape                   Baby Foods
23 Babyfood   , juice  , apple and cherry                  Baby Foods
24 Apples     , raw    , with skin                         Fruits and Fruit Juices
```

Figura 4.3: Búsqueda de los ingredientes pan ("*bread*") y manzana ("*apple*") en la base de datos para realizar posteriormente el cálculo de la aptitud

4.3. Obtención de la población inicial

Una vez definida el tipo de codificación que vamos a utilizar y la forma de calcular la aptitud, debemos definir un método que nos calcule de manera aleatoria una población inicial. Primero de todo, necesitaremos definir el número de individuos que queremos que conformen nuestra población, que en nuestro caso de estudio será mil individuos ($populationSize$), y el rango de valores posibles de la codificación, que ya los definimos en la sección 4.1. El algoritmo 1 describe el procedimiento descrito para obtener de forma aleatoria estos individuos gracias a los datos anteriores.

Algoritmo 1: getRandomGeneration

Require:
populationSize: Entero positivo que indica el tamaño de la población;
nweights: Entero positivo que indica el número de campos de cada individuo;
minValue: Mínimo valor que puede alcanzar cada campo del individuo;
maxValue: Máximo valor que puede alcanzar cada campo del individuo;

Ensure:
generation: Conjunto de posibles soluciones, cada una descrita mediante un conjunto de "nweights", donde su valor pertenece a $[minValue, maxValue]$, obtenida de forma aleatoria. Para cada individuo se calcula su valor de fitness;

```
1 for i = 0 → populationSize − 1 do
2 |   for j = 0 → nweights − 1 do
3 |   |   generation[i].weights[j]=getRandomValue(minValue,maxValue);
4 |   generation[i].fitness=getFitness(generation[i].weights);
```

4.4. Métodos de selección

En los AG existen diversas ocasiones en las que deseamos seleccionar a ciertos individuos, puede ser para aplicarle algún operador de recombinación o incluso para simular una supervivencia de los individuos y así reducir la población. En nuestro AG debemos seleccionar individuos en dos ocasiones:

Selección de individuos a cruzar. Para ello hemos decidido utilizar la selección por torneo explicada en la sección 2.1.2, ya que selecciona el mejor individuo de un número determinado de individuos escogidos aleatoriamente (*poolSize*). Podemos observar el algoritmo 2 empleado para esta selección.

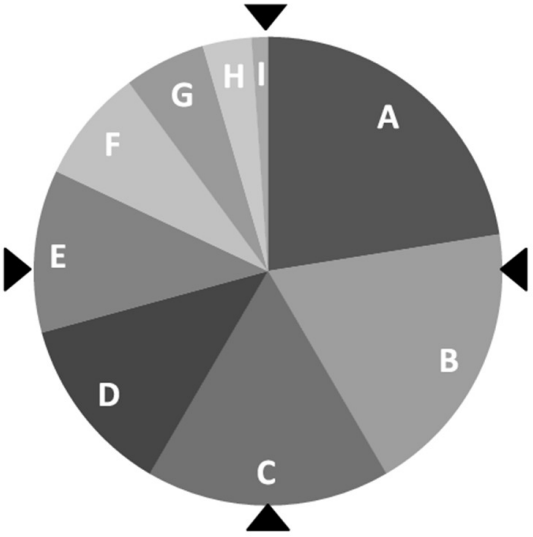
Algoritmo 2: TournamentCrossSelection

Require:
popCross: Lista de conjunto de pesos (individuos);
poolSize: Tamaño del conjunto de individuos que participan en el torneo;
Ensure:
parent: Conjunto de pesos. Puede ser vacío en el caso de que no haya individuos;

```
1 pool={}; i=0; visited={};
2 while
  (i < poolSize) ∧ (popCross.size > 0) ∧ (visited.size < popCross.size)
  do
3     k=rand(0,popCross.size-1);
4     if (popCross[k].identifier ∉ visited) then
5         pool=pool ∪ popCross[k];
6         i++;
7         visited=visited ∪ popCross[k].identifier;
8 parent=SelectBest(pool);          ▷ Teniendo en cuenta su fitness;
9 popCross=popCross–parent;
```

Selección de individuos supervivientes. Nuestro AG está diseñado de tal forma que deseamos tener un tamaño de población estable en cada generación (*popSize*). Una vez realizado el cruce de los individuos tenemos una población que excede esta capacidad, por lo que también necesitamos un método de selección que reduzca la población. El método que utilizaremos en esta ocasión será una pequeña modificación de la selección estocástica universal, perteneciente al conjunto de métodos de selección por rueda de ruleta, mencionada en la sección 2.1.2. Esta pequeña modificación consiste en escoger inicialmente un mejor individuo de la población y después realizar la selección rueda de ruleta seleccionando k individuos a la vez, por lo que seria una selección estocástica universal elitista. En la figura 4.4 podemos ver un ejemplo de la utilización de la selección estocástica universal, donde únicamente sobreviven cuatro de los nueve individuos de la población. También podemos observar en el algoritmo 3 como hemos desarrollado este método en nuestro AG.

Individuo	**Fitness**
A	20
B	17
C	15
D	11
E	10
F	7
G	5
H	3
I	1



PopSize = 4, TotalFitness = 89

Figura 4.4: Método de supervivencia selección estocástica universal

Algoritmo 3: elitistStochasticUniversalSampling

Require:
bigPopulation: Lista de conjuntos de pesos;
populationSize: Tamaño total de la población final;
Ensure:
population: Lista de conjuntos de pesos. Su tamaño será "populationSize";

```
1  population={};
2  if populationSize > 0 then
3      bigPopulation=orderByQualityDesc(bigPopulation);
4      population=population∪bigPopulation[0];  ▷ El mejor individuo es
       seleccionado;
5      wheelSize=∑_{i=1}^{i=bigPopulation.size−1} bigPopulation[i].fitness;
6      increment=wheelSize/(populationSize − 1);
7      pos=rand(0,wheelSize);
8      i=1; sum=0;
9      while population.size < populationSize do
10         stop=false;
11         while stop ≠ true do
12             sum=sum+bigPopulation[i].fitness;
13             if sum ≥ pos then
14                 stop=true;
               else
15                 i=(i+1) % bigPopulation.size;
16                 if i == 0 then
17                     i=1;  ▷ La posición 0 no se considera, porque ya ha
                       sido seleccionada;
18         population=population∪bigPopulation[i];
19         pos=pos+increment;
20         i=(i+1) % zeolites.size;
21         if i == 0 then
22             i=1;  ▷ La posición 0 no se considera, porque ya ha sido
               seleccionada;
```

4.5. Método de cruce

Se ha decidido emplear uno de los métodos explicados en la sección 2.1.3, en concreto el método de cruce BLX-α debido que, a diferencia de los demás, añade una componente aleatoria en el cálculo del nuevo valor que preserva la diversidad genética. Podemos observar cómo se aplica este operador en el algoritmo 4, donde se obtiene un conjunto de individuos cruzando a progenitores. Para ello utiliza el algoritmo 5, que se trata de un algoritmo auxiliar que únicamente se encarga de cruzar cada uno de los campos.

En nuestro AG cada par de progenitores es cruzado empleando diferentes α para generar diversos descendientes, de forma similar a la aplicada en [Herrera et al., 2002]. Para realizar esto utilizamos el algoritmo 6, que se encarga a utilizar los algoritmos anteriores, para obtener los descendientes deseados para cada par de progenitores, seleccionando de todos ellos solo aquellos dos que tengan la mejor aptitud. En el caso de que los pesos obtenidos no estén dentro del rango de pesos admisible, se procederá a reparar el resultado, aplicándoles el máximo valor si se exceden por la derecha o aplicándoles el mínimo valor si se exceden por la izquierda, recordemos que el rango de valores admisibles estaban definidos al principio del algoritmo con un valor dentro del intervalo [1, 99999].

Algoritmo 4: crossBLXalpha

Require:
alpha: Define la amplitud del intervalo de exploración;
noffspring: Número de descendientes;
parent1: Pesos del primer progenitor;
parent2: Pesos del segundo progenitor;
minValue: Mínimo valor que puede alcanzar cada campo del individuo;
maxValue: Máximo valor que puede alcanzar cada campo del individuo;

Ensure:
offspring: Diferente conjunto de pesos como resultado de cruzar los padres (parent1 y parent2), cada valor del peso estará entre $[minValue, maxValue]$;

```
1 for n = 0 → noffspring − 1 do
2     for i = 0 → parent1.size − 1 do
3         offspring[n]=offspring[n]∪pointCrossBLXalpha(alpha,parent1[i],
          parent2[j],minValue,maxValue);
```

Algoritmo 5: pointCrossBLXalpha

Require:
alpha: Define la amplitud del intervalo de exploración [0,1];
parent1: Peso del primer progenitor;
parent2: Peso del segundo progenitor;
minValue: Mínimo valor que puede alcanzar cada campo del individuo;
maxValue: Máximo valor que puede alcanzar cada campo del individuo;
Ensure:
offspring: Peso;

```
1 maxp=max(parent1,parent2);
2 minp=min(parent1,parent2);
3 interval=maxp-minp;
4 minI=minp-interval*alpha;
5 maxI=maxp+interval*alpha;
6 offspring=rand(minI,maxI);
7 offspring=repairPoint(offspring,minValue,maxValue);
```

Algoritmo 6: multipleCrossBLXAlpha

Require:
numBLX: Número de llamadas con diferente BLX crossover operators;
alphas: Alphas que definen la anchura del intervalo de exploración;
noffsprings:Lista con el número de descendientes para cada BLX;
parent1: Pesos del primer progenitor;
parent2: Pesos del segundo progenitor;
minValue: Mínimo valor que puede alcanzar cada campo del individuo;
maxValue: Máximo valor que puede alcanzar cada campo del individuo;
Ensure:
offspring: Dos conjuntos de pesos que se obtienen al cruzar los padres;

```
1 offspring={};
2 for i = 0 → numBLX − 1 do
3     offspring[n]=offspring[n]∪crossBLXalpha(alphas[i],noffsprings[i],parent1,
      parent2,minValue, maxValue);
4 offspring=SelectTwoBest(offspring);  ▷ Teniendo en cuenta su fitness;
```

4.6. Método de mutación

La idea esencial de aplicar la mutación en el AG es preservar la diversidad genética, pero cabe destacar que no siempre es necesario aplicarla. A lo largo del proyecto se planteó su uso, aunque inicialmente la idea de utilizar la mutación fue descartada, ya que el operador de cruce que utilizamos ya posee una fuerte componente aleatoria que preserva la diversidad genética. Utilizaremos el método mutación no uniforme, explicado en la sección 2.1.4, debido a su característica de centrarse en una búsqueda más localizada cuanto mayor sea la generación, permitiendo un menor impacto en las fases finales de la búsqueda.

En el algoritmo 8 podemos ver como aplicamos este método de mutación en nuestro AG. Primero de todo tenemos al individuo que vamos a mutar *parent*, de este elegimos de manera aleatoria un número de genes para mutar y después seleccionaremos estos genes y mediante el algoritmo auxiliar 7 calcularemos dicha mutación. Del resultado de que obtengamos, en el caso de que tuviéramos más de un nuevo individuo nos quedaremos únicamente con aquel que tenga mejor aptitud.

Algoritmo 7: deltaAdaptativeNonUniformMutation

Require:
generation: Número de la generación actual;
maxGenerations: Máximo número de generaciones;
y: Número real;
b: Vector de enteros;
Ensure:
deltas: Vector de reales;

1 **for** $i = 0 \rightarrow b.length - 1$ **do**
2 delta[i]=$y(1 - r^{(1-\frac{generation}{maxGenerations})^{b[i]}})$;

Algoritmo 8: NonUniformMutation

Require:
generation: Número de la generación actual;
maxGenerations: Máximo número de generaciones;
minValue: Mínimo valor que puede alcanzar cada campo del individuo;
maxValue: Máximo valor que puede alcanzar cada campo del individuo;
b: Vector de enteros;
parent: Conjunto de pesos;
Ensure:
offspring: Conjunto de pesos. Algunos de estos pesos tienen diferentes valores a los introducidos inicialmente;

```
1  mutated={};
2  for i = 0 → b.length − 1 do
                    ▷ Cada posible resultado es una copia del padre;
3      candidate[i]=parent;
               ▷ Seleccionamos un número de genes para mutar;
4  numGenes=rand(0,parent.length-1);
5  for k = 0 → numGenes do
6      while i ∈ mutated do
7          i=rand(0,parent.length-1);          ▷ i es un valor entero;
8      mutated=mutated ∪ i; ▷ i es la siguiente posición del gen a mutar;
9      if rand(0, 1) < 0,5 then
10         deltas=deltaAdaptativeNonUniformMutation(generation,maxGenerations,
           maxValue-parent[i],b);
11         for j = 0 → deltas.length − 1 do
                       ▷ round asegura que obtenemos un valor entero;
12             candidate[j][i]=round(parent[i]+deltas[j]);
       else
13         deltas=deltaAdaptativeNonUniformMutation(generation,
           maxGenerations,parent[i]-minValue,b);
14         for j = 0 → deltas.length − 1 do
                       ▷ round asegura que obtenemos un valor entero;
15             candidate[j][i]=round(parent[i]-deltas[j]);
16 offspring=SelectBest(candidate);     ▷ Teniendo en cuenta su fitness;
```

4.7. Algoritmo genético propuesto

Los operadores y técnicas aplicadas anteriormente se combinan para dar lugar al AG que emplearemos en la búsqueda de la combinación de pesos ideal para aplicar a los campos de características. Así en el algoritmo 9 describe como son aplicadas. En primer lugar obtenemos una generación aleatoria inicial. Posteriormente, aplicamos el método de selección de los individuos que vamos a cruzar y realizamos el cruce entre los dos individuos seleccionados. Después, realizamos otra selección para tener un tamaño de población estable en nuestro AG, en esta selección se encuentran los individuos de la generación anterior y los nuevos individuos que han sido creados a través del método de cruce. Por último, comprobamos que la aptitud del mejor individuo de la población que ha sobrevivido, es mejor que la aptitud del mejor individuo de la anterior generación. Si durante *numGenNoImprovement* generaciones no se ha mejorado la aptitud o hemos aplicado más generaciones que el limite establecido (*maxGen*), termina el AG.

Algoritmo 9: Multipe BLXalpha approach

Require:

testNumber: Número de diferentes test que lanzaremos;
maxGen: Máximo número de generaciones permitidas para cada test;
numGenNoImprovement: Máximo número de generaciones seguidas por cada test en la que no se obtiene mejora;
popSize: Número del conjunto de pesos que forman la generación;
minValue: Mínimo valor que puede alcanzar cada campo del individuo;
maxValue: Máximo valor que puede alcanzar cada campo del individuo;
numBLX: Numero de casos al cruce $BLX\alpha$;
alphas: Una lista con el α empleado en cada llamada al $BLX\alpha$. *alpha* define la anchura del intervalo de exploración;
noffsprings: Una lista con el numero de descendientes generados por cada operador de cruce;
poolSize: Tamaño de los individuos que participan en el torneo;
storePath: Directorio para almacenar los resultados;

Ensure:

generations: Archivos almacenados en el directorio dado;

```
1  for t = 0 → testNumber − 1 do
2      population=getRandomGeneration(popSize,nweights,minValue,
       maxValue);                          ▷ Generación inicial;
3      storeGeneration(t,gen,path);   ▷ Guardamos la generación inicial;
4      bestFitness=0; genCont=0; continue=true; gen=0;
5       ▷ Si el fitness no mejora en numGen generaciones, el test para de
       ejecutarse;
6      while gen < maxGen ∧ continue do
7          crossPop=population;              ▷ Empieza el cruce;
8          repeat
9              repeat
10                 parent1=tournamentCrossSelection(crossPop, poolSize);
11             until (parent1 ≠)null ∨ (crossPop.size == 0);
12             repeat
13                 parent2=tournamentCrossSelection(crossPop, poolSize);
14             until (parent2 ≠ null) ∨ (crossPop.size == 0);
15             if (parent1 ≠ null) ∧ (parent2 ≠ null) then
16                                 ▷ Realizamos el cruce de los padres;
17                 population=population++multipleCrossBLXAlpha(numBLX,
                   alphas, noffsprings, parent1, parent2, minValue,
                   maxValue);
18         until crossPop.size == 0;
19                                                  ▷ Termina el cruce;
20         population=elitistStochasticUniversalSampling(population,popSize);
           ▷ Reduce el tamaño de la población a popSize;
21         gen++;
22         if population[0].fitness > bestFitness then
23                 ▷ La población esta ordenada de manera decreciente;
24             bestFitness=population[0].fitness;
25             genCont=0;                          ▷ Empieza el ciclo de
               numGenNoImprovement para mejorar el mejor fitness;
26         else
27             genCont++;
28             if genCont ≥ numGenNoImprovement then
29                 continue=false;
30         generationToFile(t,gen,path);        ▷ Guardamos la generación
           actual;
```

Capítulo 5

Aplicación del algoritmo genético

Este capítulo tratará sobre las diferentes pruebas lanzadas para verificar la idoneidad del diseño del AG propuesto para la solución del problema planteado. Inicialmente se mostrará de los resultados que se obtienen sin utilizar la división de los campos de características definidas por la base de datos USDA. Después se expondrá una nueva búsqueda aplicando la división por campos de características de los ingredientes en la base de datos. A continuación se aplicará el AG diseñado, explicando las variaciones que se introdujeron a fin de mejorar los resultados obtenidos. A la hora de realizar las búsquedas, se utilizó siempre los mismos ingredientes. Estos ingredientes han sido seleccionados aleatoriamente de las recetas obtenidas de receteame.com.

5.1. Búsqueda sobre la descripción del ingrediente

Inicialmente realizamos una búsqueda sobre la descripción del ingrediente. Esta descripción son las características del ingrediente divididas mediante comas. Al realizar la búsqueda de los ingredientes sobre la base de datos, pudimos observar como, de todos los ingredientes, sólo el 40 % de los resultados, coincidían con los esperados. En la columna de la izquierda de la figura 5.1 se puede observar los resultados de esta búsqueda.

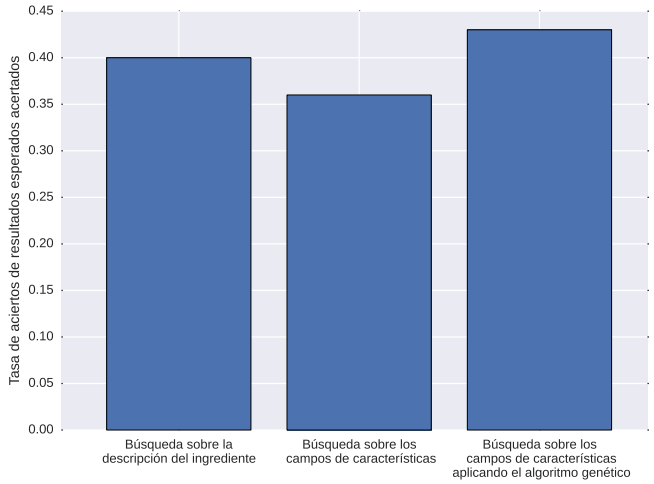


Figura 5.1: Comparativa entre la búsqueda sobre la descripción del ingrediente y la búsqueda sobre los campos de características, sin aplicar el algoritmo genético y aplicándolo

5.2. Búsqueda inicial por campos

A continuación, después de realizar una búsqueda sobre la descripción del ingrediente, hicimos la prueba de dividir la descripción del ingrediente en campos de características. Como la descripción del ingrediente, ya estaba dividida por comas, decidimos asignar las partes divididas por comas a campos de características. Después de realizar esta división, obtuvimos un total de once campos de características, donde utilizando la base de datos de MongoDB podíamos aplicarle prioridad a estos campos. Primero decidimos comprobar como variaban los resultados al aplicar sin aplicarle ninguna prioridad a esta división de los campos, es decir, utilizando los pesos que nos proporciona MongoDB por defecto que equivale a que todos los campos de características tienen la misma prioridad. En la columna del centro de la figura 5.1 se puede observar como el porcentaje de aciertos de los resultados esperados es menor que realizando la búsqueda sobre la descripción del ingrediente, obtenemos un 36 % de aciertos.

5.3. Aplicación del algoritmo genético

Tras haber realizado una búsqueda en la descripción del ingrediente y otra aplicando la división por campos de características sin aplicar ninguna prioridad, decidimos aplicar el AG para obtener una combinación de pesos adecuada para aplicar a cada uno de los campos de características, que nos ayudara a obtener una buena combinación de entre los resultados obtenidos por la base de datos y los deseados. En la figura 5.2 se puede observar la gráfica donde aparecen estas pruebas en función de la aptitud media de los individuos de la generación en la que se encuentran. Podemos observar que las pruebas realizadas tienen una evolución similar, donde la aptitud media de los individuos alcanza aproximadamente un valor de 0,0025. Inicialmente parten de un valor demasiado bajo (0,00025), pero a medida que van aumentando las generaciones vemos como rápidamente converge. Este valor de la aptitud, a simple vista parece demasiado bajo, pero hay que tener en cuenta que no representa al porcentaje de los ingredientes acertados en la búsqueda, que en este caso seria el 43 %, como lo podemos observar en la columna de la derecha de la figura 5.1. Este valor es una forma de puntuar a los individuos según en que posición se encuentre la solución esperada, como ya se explicó en la sección 4.2.

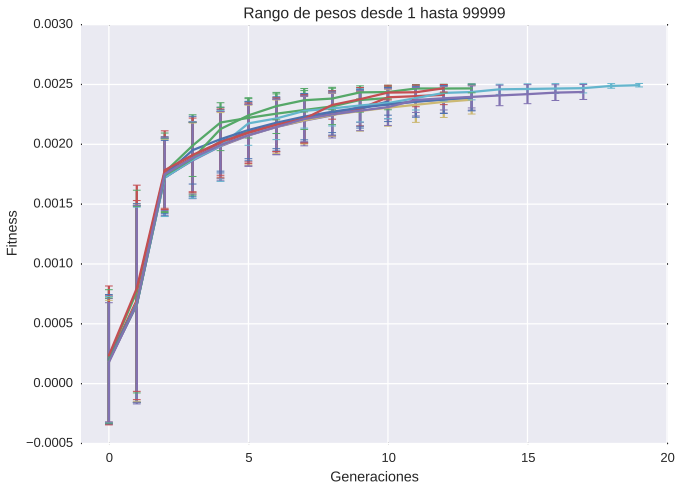


Figura 5.2: Primeras pruebas lanzadas por el algoritmo genético con una codificación dentro de un rango de valores [1, 99999]

A la vista de los resultados obtenidos podemos concluir que el AG era capaz de establecer unos pesos a cada campo de características de forma que se obtenía mejores resultados que buscando directamente sobre la descripción sin dividir del ingrediente. La mejora era de un 3 %, pero estimamos, que variando el diseño inicial del AG sería posible mejorar estos resultados.

5.4. Paralelización del algoritmo genético

Una de las primeras modificaciones que realizamos fue la de paralelizar el algoritmo. Nuestro AG inicial se encargaba de realizar diez pruebas distintas en un ordenador[1], cada prueba tenia que realizar más de 200000 búsquedas en la base de datos y cada una de estas búsquedas tardaban sobre medio segundo. Por tanto, tardaba prácticamente una semana en terminar diez pruebas.

La idea principal que se siguió fue la de paralelizar al máximo el algoritmo, intentando reducir el tiempo de ejecución lo máximo posible. Como realizamos diez pruebas, decidimos utilizar diez ordenadores[2] distintos para cada una de ellas. En cada ordenador teníamos la base de datos repetida cuatro veces y nuestro algoritmo repartía su tarea en cuatro procesos. Cada uno de estos procesos se encargaban de realizar búsquedas en la base de datos y al estar ésta clonada obtenían datos de diversos individuos en paralelo. Para organizar estos ordenadores utilizábamos una herramienta de automatización de la información llamada *Ansible*[3]. Esta herramienta nos permite gestionar de forma remota un grupo de servidores, permitiendo automatizar la mayoría de procesos de estos.

Esta paralelización que diseñamos agilizaba considerablemente el trabajo realizado por nuestro algoritmo. Así, inicialmente con un ordenador tardaba casi una semana en realizar las diez pruebas, pero gracias a esta paralelización, conseguimos reducir que el conjunto de las diez pruebas tardase en ejecutarse como mucho en un día y medio.

Para la paralelización de nuestro algoritmo aplicamos la interfaz MPI[4]. Gracias a esta interfaz pudimos aplicar rápidamente el paso de mensajes en-

[1] El ordenador que disponíamos era un Intel Core^{TM} i3-2125 CPU 3.3GHz x 2

[2] Cada uno de estos ordenadores tenían la misma característica que el ordenador inicial, Intel Core^{TM} i3-2125 CPU 3.3GHz x 2

[3] Podemos encontrar más información a través de su página web www.ansible.com.

[4] Interfaz de Paso de Mensajes (*Message Passing Interface* es una interfaz de paso de mensajes que satisface las necesidades de paralelizar programas utilizando los múltiples procesadores que dispone el procesador.

tre los diferentes procesos del AG.

En las siguientes secciones explicaremos detalladamente las distintas modificaciones que realizamos al diseño original del AG para mejorar los resultados obtenidos por el mismo, a fin de obtener una combinación de pesos más prometedores.

5.5. Modificación del rango de valores de los genes

Como ya habíamos comentado en la sección 4.1, los genes podían tener una codificación dentro de un rango de valores [V_{min}, V_{max}]. Inicialmente estos valores eran 1 y 99999 respectivamente, puesto que éstos son los valores admitidos por la base de datos MongoDB. A fin de mejorar los resultados, decidimos probar distintos tipos de codificación en el AG. Esta idea también fue fundamentada sobre la hipótesis de que, con un menor espacio de búsqueda, seria más sencillo recorrerlo y explorar todas las posibles soluciones. Probamos tres tipos distintos de rangos de pesos para estos genes, lanzando diez pruebas con cada una de estas codificaciones.

Rango de pesos desde 1 hasta 11. Como cada ingrediente tiene como máximo once campos de especificación, decidimos probar unos valores desde 1 hasta 11. Podemos observar en la figura 5.3 como la aptitud media de los individuos de cada generación convergía rápidamente, debido a su drástica reducción del espacio de búsqueda. También cabe destacar que los últimos valores de la aptitud media de los individuos mejora a los valores obtenidos inicialmente como podemos observar en la gráfica 5.2, puesto que obteníamos un valor medio de la aptitud de 0,0026 frente al valor de 0,0025 que obteníamos inicialmente. Como podemos observar en la columna de la izquierda de la figura 5.6, aunque se obtenía una leve mejora en los valores de la aptitud media, la tasa de aciertos no mejoraba, quedándose en un 43 %,.

Rango de pesos desde 1 hasta 100. Visto los resultados anteriores decidimos ampliar un poco más el espacio de búsqueda para ver como influía en los resultados. En la figura 5.4 observamos como esta vez no converge tan rápidamente los resultados, ya que se obtienen un mayor número de generaciones. Los resultados obtenidos mejoraban levemente los últimos valores de la aptitud media de los individuos obtenidos con el rango de pesos desde 1 hasta 11, obteníamos un valor medio de 0,00275 que mejora un 0,00015 a la

utilización del rango de pesos anterior. En la columna del medio de la figura 5.6 podemos observar como esta leve mejora de los valores de la aptitud media de los individuos correspondía también a una mejora de la tasa de aciertos de los ingredientes, acertando un 46 % de ingredientes, un 3 % más que utilizando el rango de pesos anterior. Decidimos que podría ser útil esta modificación.

Rango de pesos desde 1 hasta 9999 de 100 en 100. En este caso probamos con un rango de pesos que variaba desde 1 hasta 9999, pero en saltos de 100 en 100, a fin de reducir el espacio de búsqueda. Además, la característica de añadir saltos hace que la prioridad entre los distintos campos de características sean más distantes. Esta característica pensamos que podría mejorar las búsquedas. En la figura 5.5 podemos observar que la aptitud media de los individuos obtenidos es muy similar a la anterior modificación, obteniendo en ambas un valor de aptitud media de los individuos sobre 0,00275. También podemos observar en la columna de la derecha de la figura 5.6 que la tasa de aciertos es 46 % como el rango de pesos desde 1 hasta 100.

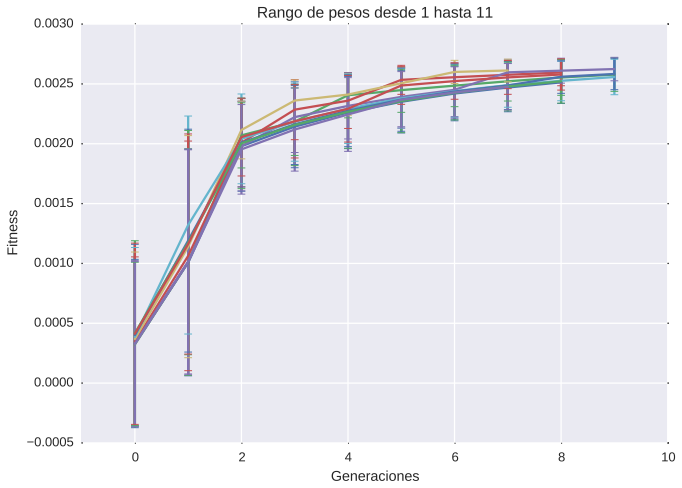


Figura 5.3: Pruebas lanzadas con una codificación dentro de un rango de valores [1, 11]

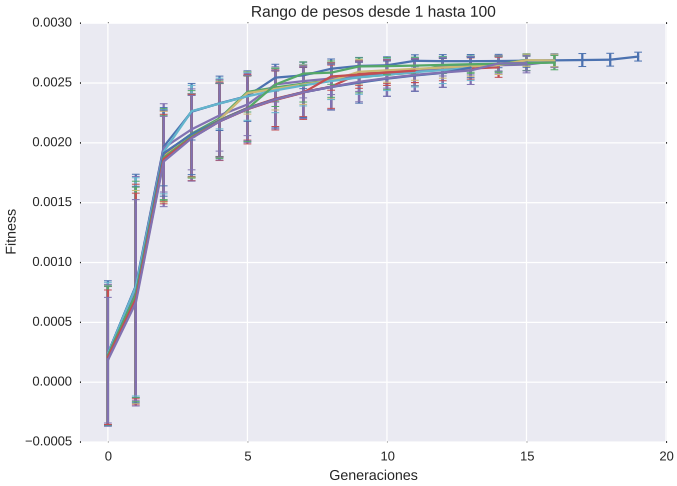


Figura 5.4: Pruebas lanzadas con una codificación dentro de un rango de valores [1, 100]

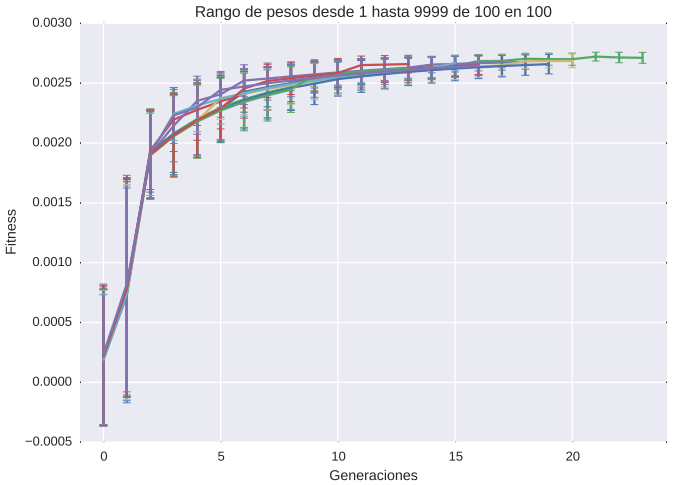


Figura 5.5: Pruebas lanzadas con una codificación dentro de un rango de valores [1, 9999] con saltos de 100

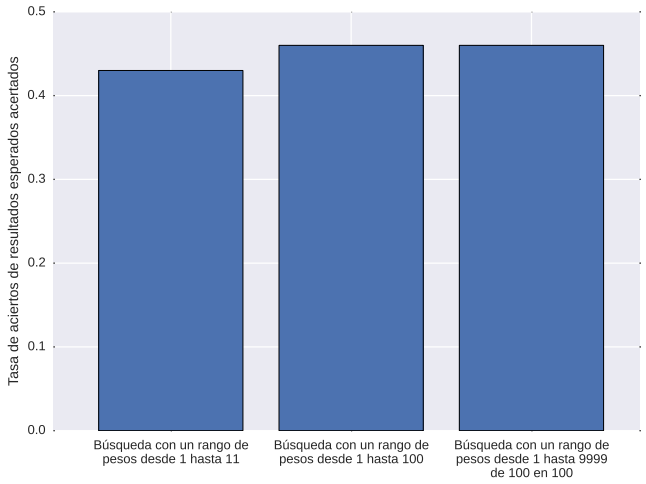


Figura 5.6: Comparativa de la tasa de aciertos usando búsquedas con rango de pesos desde 1 hasta 11, desde 1 hasta 100 y de 1 hasta 9999 con saltos de 100 en 100

Vistos los resultados anteriores, decidimos modificar nuestro AG para que el rango de pesos que utilizara en la codificación del problema fuera un rango de valores desde 1 hasta 9999 con saltos de 100 en 100. La decisión de elegir esta codificación es debida a los buenos resultados que nos proporciona la aptitud media de los individuos, mejorando la tasa de aciertos. También decidimos escoger este rango de valores frente a los otros debido a los saltos que proporciona un valor de otro, dentro de la codificación, permitiendo que la diferencia de la prioridad entre los distintos campos de características sea mayor.

5.6. Nuevos métodos

Después de haber elegido una codificación más apropiada para nuestro problema intentamos aumentar los resultados obtenidos de la aptitud media de los individuos. Para ello decidimos intentar dos posibles mejoras, aplicar un método de selección más elitista para el cruce y aplicar la mutación, ya que hasta ahora aun no la habíamos aplicado al AG.

5.6.1. Nueva selección elitista

La primera idea que tuvimos es que posiblemente debíamos diseñar una selección más elitista para la selección de cruce que la selección por torneo que utilizábamos, explicada en la sección 4.4. La selección elitista que queríamos realizar consistía en seleccionar los mejores para cruzarlos con los mejores, es decir, ordenábamos a todos los individuos de nuestra población y después los seleccionábamos según su aptitud de dos en dos.

Con esta nueva selección esperábamos mejorar la aptitud media de los individuos, pero tras realizar una nueva batería de diez pruebas se puede observar en la figura 5.7 como los valores de la aptitud media de los individuos son bastante similares a los resultados que obteníamos en la anterior selección. Con esta nueva selección obtenemos un valor de la aptitud media de los individuos de 0,0027, mientras que con la selección anterior obteníamos una aptitud media de 0,00275. Decidimos descartar este nuevo método de selección, ya que, aunque obteníamos resultados similares puede que al ser tan elitista nos estemos enfocando a soluciones más locales, evitando explorar todo el espacio de búsqueda.

5.6.2. Mutación

La idea de aplicar la mutación surgió debido a que en pocas generaciones la aptitud media de los individuos así como sus genes eran muy similares, de forma que en la última generación todos los individuos eran prácticamente igual de buenos. Por ello, decimos aplicar el método de mutación, definido en la sección 4.6, el cual podría preservar la diversidad genética. Así, una vez que habíamos realizado los métodos de cruce, cada individuo progenitor tenía una probabilidad del 5 % de ser seleccionado para ser mutado.

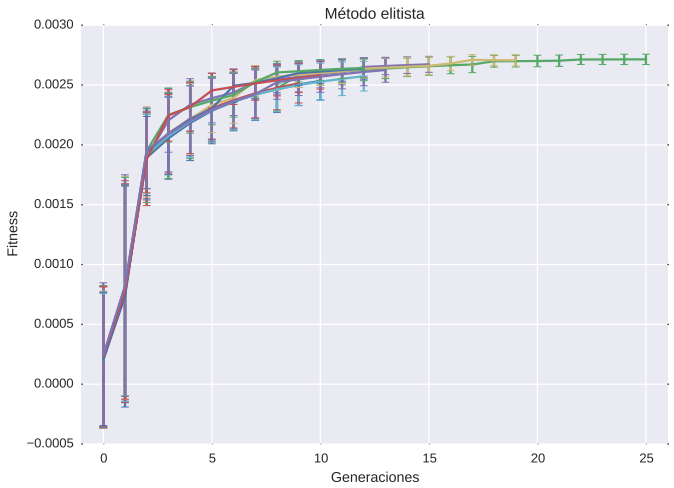


Figura 5.7: Valores de la aptitud media utilizando el método de selección elitista

Tras realizar diez pruebas podemos observar en la figura 5.8 que los valores de la aptitud media de los individuos no mejoraba, obteníamos unos valores similares como los que obteníamos al aplicar la selección elitista, es decir, un valor de 0,00271. Decidimos incluir esta mejora a nuestro AG, ya que también buscábamos preservar la diversidad genética, y con la utilización de este método lo conseguíamos.

5.7. Cambio de la función de aptitud

Después de seguir realizando más pruebas con un rango de pesos de 1 a 9999 con saltos de 100 en 100 y aplicando mutación en el AG, observamos que no variaban los valores de la aptitud media de los individuos y que el valor de este era muy próximo a cero. Como posible mejora se decidió modificar la función que calculaba la aptitud por otra menos pesimista, ya que nuestro cálculo penalizaba demasiado los casos en los que no se encontraba el resultado en las primeras cien posiciones. Como en el cálculo de la aptitud anterior, explicada en la sección 4.2, nos guardaríamos la posición en la que aparece $posicion_i$, esta vez la posición tendrá un rango de valores comprendido entre [1, 100]. En el caso que el ingrediente no saliera en la búsqueda añadiría-

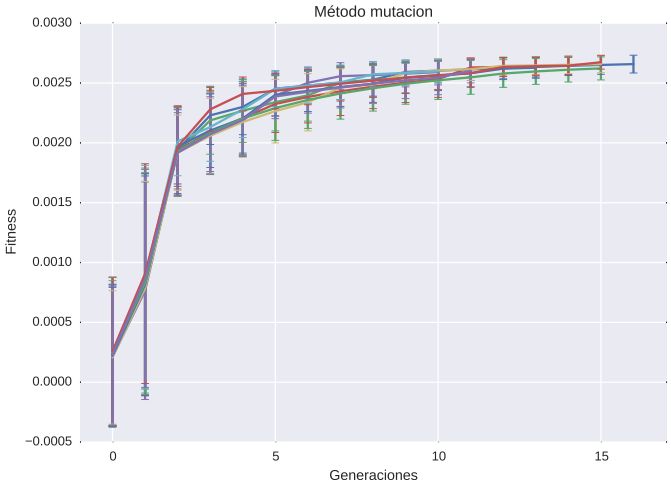


Figura 5.8: Valores de la aptitud media utilizando el método de mutación

mos el valor 101 para evitar que si no lo encontrara penalizara demasiado. Con anterioridad penalizábamos con diez millones. Por último, se sumaban de las posiciones de todos los ingredientes y se dividían al número total de ingredientes (N), en este caso cien, para obtener un valor comprendido entre [0, 1]. En la ecuación 5.1 podemos observar como establecimos este nuevo cálculo.

$$posicion_i = \begin{cases} [1, 100] & \text{, sí se encuentra en la búsqueda} \\ 101 & \text{, sí no se encuentra en la búsqueda} \end{cases}$$

$$aptitud = \frac{N}{\sum_{i=1}^{N} posicion_i} \tag{5.1}$$

En el cálculo de la aptitud que teníamos anteriormente, explicada en la sección 4.2, realizábamos a modo de ejemplo una búsqueda de dos ingredientes en la figura 4.2. El resultado de esta búsqueda nos proporcionaba un valor de $aptitud = \frac{1}{2+1} = 0{,}33$, mientras que con este nuevo cálculo de la aptitud obtendríamos: $aptitud = \frac{2}{4} = 0{,}5$. Podemos observar como este nuevo valor que obtenemos en el cálculo de la aptitud no es tan pesimista como el anterior.

En este nuevo ejemplo de la figura 5.9 realizamos una búsqueda de cuatro ingredientes, dos de ellos son los comentados en el ejemplo de la figura 4.2, que se encuentran en la segunda posición. Los otros dos ingredientes también

los podemos encontrar en el ejemplo de la figura 4.3, y sus resultados se encuentran en la quinta y onceava posición . En este ejemplo, utilizando el cálculo de la aptitud que teníamos inicialmente la *aptitud* tendría un valor de $\frac{1}{1+1+4+10+1} = \frac{1}{21} = 0{,}058$, mientras que con este nuevo cálculo la *aptitud* tendría un valor de $\frac{4}{2+2+5+11} = \frac{4}{20} = 0{,}2$.

```
Search:"bacon"
Exit:
      C1      ,    C2    ,     C3      ,    C4      Description Food Group
Bacon         , meatless                            Legumes and Legume
Pork          , bacon    , rendered fat, cooked     Pork Products
Pork          , cured    , bacon       , unprepared Pork Products
Turkey bacon  , unprepared                          Sausages & Luncheon Meats
Canadian bacon, unprepared                          Pork Products


Search:"papaya"
Exit:
      C1     ,    C2   ,     C3      ,    C4         Description Food Group
Papaya       , canned  , heavy syrup , drained      Fruits and Fruit Juices
Papayas      , raw                                  Fruits and Fruit Juices
Papaya nectar, canned                               Fruits and Fruit Juices
Fruit salad  , tropical, canned      , heavy syrup  Fruits and Fruit Juices
Babyfood     , fruit   , papaya with tapioca        Baby Foods


Search:"bread"
Exit:
      C1       ,      C2     ,    C3   ,    C4       Description Food Group
Bread stuffing , bread       , dry mix               Baked Products
Bread stuffing , bread       , dry mix , prepared    Baked Products
Bread          , pan dulce   , sweet                 Baked Products
Bread          , whole wheat , frozen                Baked Products
Bread          , wheat                               Baked Products


Search:"apple"
Exit:
     C1    ,      C2                   ,    C3       Description Food Group
Croissants , apple                                  Baked Products
Strudel    , apple                                  Baked Products
Babyfood   , apples , dices             , toddler    Baby Foods
Babyfood   , juice  , apple                         Baby Foods
Babyfood   , juice  , apple and peach               Baby Foods
Babyfood   , juice  , apple and plum                Baby Foods
Babyfood   , juice  , apple and prune               Baby Foods
Babyfood   , juice  , orange and apple              Baby Foods
Babyfood   , juice  , apple and grape               Baby Foods
Babyfood   , juice  , apple and cherry              Baby Foods
Apples     , raw    , with skin                     Fruits and Fruit Juices
```

Figura 5.9: Búsqueda de los ingredientes panceta ("*bacon*"), papaya, pan ("*bread*") y manzana ("*apple*") en la base de datos para realizar posteriormente el cálculo de la aptitud

Observamos que los resultados obtenidos por este muevo cálculo de la aptitud, nos proporciona unos valores más elevados que los anteriores. Una vez realizadas varias pruebas pudimos comprobar que este nuevo cálculo de la aptitud nos proporciona mejores resultados que el cálculo anterior. Para poder compararlos hemos utilizado la tasa de ingredientes acertados. En la figura 5.10 se puede observar que, utilizando el cálculo de la aptitud anterior y después de realizar las modificaciones anteriores, obteníamos un 53 % de aciertos, mientras que con esta nueva forma de calcular la aptitud alcanzábamos hasta un 55 % de ingredientes acertados. También podemos observar como este nuevo cálculo mejora un 19 % a los resultados de la búsqueda que obteníamos sin haber aplicado el AG a la obtención de los pesos a aplicar a los campos de características.

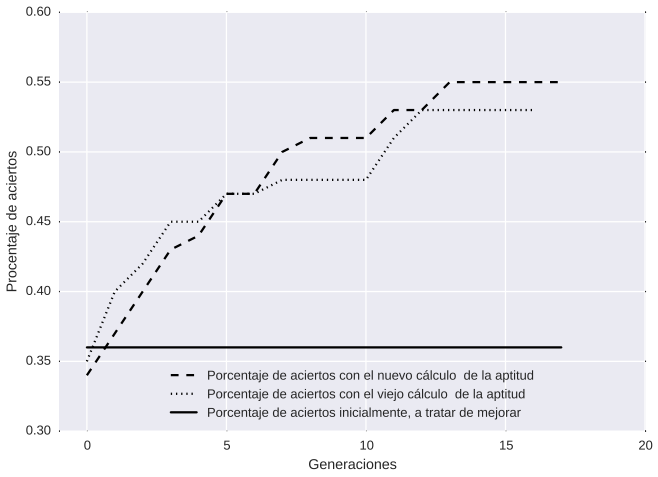


Figura 5.10: Comparativa del porcentaje de aciertos de los resultados esperados, el nuevo cálculo de la aptitud representado como "--", el viejo cálculo de la aptitud representado como "·-·" y por último sin haber aplicado el algoritmo genético representado como "-"

5.8. Reducción de campos de características

A continuación, se decidió realizar un estudio más profundo sobre los ingredientes presentes en la base de datos, y pudimos observar que no todos ellos utilizaban los once campos de características. Por eso decidimos com-

probar cuál era la diferencia de trabajar únicamente con cinco campos de características en vez de once.

Primero de todo debíamos hacer una modificación en la codificación del problema, ya que ahora no se trataba de un problema con once genes (pesos de los campos de características), sino que lo habíamos reducido a cinco. Después de realizar esta modificación, nos dispusimos a realizar un par de pruebas que nos mostraran si la modificación aumentaba los resultados de búsquedas satisfactorias. Comparando la gráfica de los resultados de la aptitud media de los individuos obtenidos al trabajar con cinco pesos, donde hemos obtenido un valor medio de 0,19 (figura 5.11), a los resultados obtenidos al trabajar con los once pesos, donde hemos obtenido un valor medio de 0,21 (figura 5.12), podemos observar que los valores de la aptitud media de los individuos eran bastante menores que usando los once campos de características. Otro de los motivos por el que queríamos aplicar esta reducción de los campos de características era porque intentábamos reducir el tiempo de ejecución del algoritmo. Tras la realización de este estudio comprobamos que el tiempo de ejecución era muy similar en ambos casos, debido a que internamente la base de datos tardaba el mismo tiempo en aplicarle pesos a los cinco campos que al aplicarle los pesos a los once campos. Dicho todo esto, se decidió que no se debía aplicar esta modificación en el AG, ya que no ayudaba a mejorar los resultados de la aptitud media de los individuos ni reducía el tiempo de ejecución.

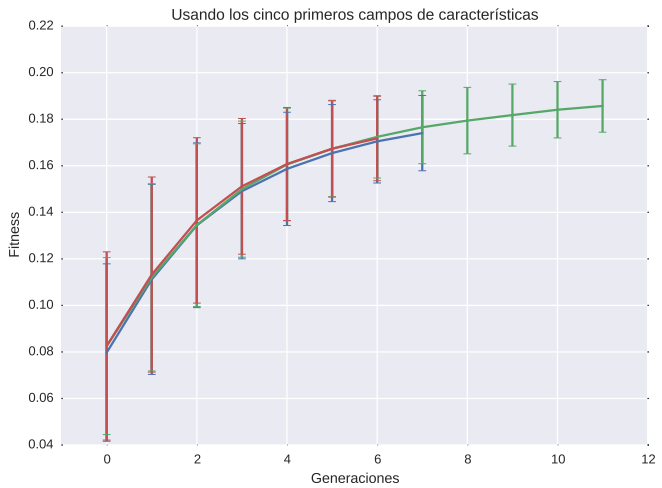


Figura 5.11: Valores de la aptitud media utilizando cinco campos

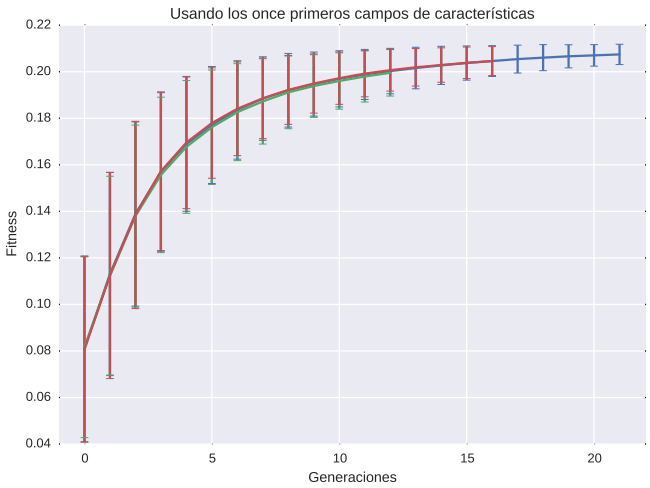


Figura 5.12: Valores de la aptitud media utilizando once campos

5.9. Aplicación de una red neuronal para la aproximación de la función de aptitud

El cálculo de la función aptitud requiere de acceder a la base de datos, modificar los pesos de los campos de características y realizar cien consultas sobre ella. Todo el acceso a la base de datos es muy costoso. Por ello, a fin de agilizar los resultados se decidió utilizar redes neuronales para obtener una aproximación de la aptitud. Como en este problema a tratar teníamos unos individuos con once pesos, decidimos implementar un perceptrón multicapa en el que su capa de entrada tuviera once neuronas. También como es lógico la capa de salida únicamente tendrá una neurona, ya que nos interesa un único valor, que es el de la aptitud. Solo quedaba por determinar cuantas capas ocultas deberíamos usar y cuantas neuronas tenían que contener cada una de ellas.

En primer lugar, cabe destacar que para entrenar las RNA teníamos calculados con anterioridad cien mil individuos con su respectiva aptitud. De estos cien mil individuos cogíamos de manera aleatoria el 80 % para entrenar y el otro 20 % para hacer las pruebas. Dentro de los individuos para entrenar utilizamos un 25 % para validar el entrenamiento y evitar el sobre-ajuste. También usamos la función de activación tangencial para las capas ocultas, mientras que utilizamos la capa de salida sigmoidal para la capa de salida.

Dicho esto nos planteamos un experimento para estableces la estructura de RNA más adecuada a nuestro problema. Para ello implementamos un método que iba calculando diferentes RNA y nos devolvía aquella que menor error había cometido. Las RNA que exploramos eran de dos tipos:

(a) **Perceptrones de dos capas.** Disponían de una única capa oculta y de diferente número de las neuronas. En concreto, se empezaba usando once neuronas y aumentando de diez en diez hasta tener un máximo de 220 neuronas.

(b) **Perceptrones de tres capas.** A diferencia del anterior, teníamos dos capas ocultas. Realizábamos combinaciones de 11 a 110 neuronas, en incrementos de diez, por cada capa oculta.

Con todas las estructuras de RNA obtenidas, implementadas anteriormente, realizábamos diferentes entrenamientos, con diferentes factores de aprendizaje para el entrenamiento y también probábamos a añadir un factor momentum para acelerar la convergencia de la RNA. En el algoritmo 10 podemos observar como entrenamos las diferentes redes creadas, hasta la convergencia, quedándonos con aquella que menor error cometía.

Algoritmo 10: neuronalNetwork

Require:
individuals: Lista de individuos para entrenar la red neuronal;
Ensure:
rna: Red neuronal con la que se obtiene el menor error cometido;

```
1  tstdata, trndata = individuals.splitWithProportion(0.20);
2  rna.error = Infinite;
3          ▷ Primero calcularemos las redes con una capa oculta;
4  for nodes ∈ [11, 220, 10] do
5                      ▷ Valores de 11 a 220 con saltos de 10;
6      net = createNet(inLayer=11, hiddenLayer1=nodes, outLayer=1);
7      for α ∈ [0.3, 1, 0.2] do
8          train = net.trainUntilConvergence(validationProportion=0.25,
           learningrate=α);
9          if train.error < rna.error then
10             rna = train;
11         for μ ∈ [0.3, 1, 0.2] do
12             train =
               net.trainUntilConvergence(validationProportion=0.25,
               learningrate=α, momentum=μ);
13             if train.error < rna.error then
14                 rna = train;
15         ▷ Después calcularemos las redes con dos capas ocultas;
16 for nodes1 ∈ [11, 110, 10] do
17     for nodes2 ∈ [11, 110, 10] do
18         net = createNet(inLayer=11, hiddenLayer1=nodes1,
           hiddenLayer2=nodes2, outLayer=1);
19         for α ∈ [0.3, 1, 0.2] do
20             train =
               net.trainUntilConvergence(validationProportion=0.25,
               learningrate=α);
21             if train.error < rna.error then
22                 rna = train;
23             for μ ∈ [0.3, 1, 0.2] do
24                 train =
                   net.trainUntilConvergence(validationProportion=0.25,
                   learningrate=α, momentum=μ);
25                 if train.error < rna.error then
26                     rna = train;
```

Después de ejecutar el algoritmo anterior, obtuvimos una RNA con dos capas ocultas y con 171 nodos en cada capa oculta. El factor de aprendizaje y el momentum empleados fue de 0.9. Una vez obtenido el modelo, lo aplicamos en el cálculo de la función de aptitud de nuestro AG. Cada cinco generaciones se empleaba el cálculo de la función de la aptitud como generación de control para evitar que la aproximación empleada por la RNA desviara al AG de la realidad. En la figura 5.13 podemos observar el resultado de haber empleado la RNA como método de aproximación de la función de aptitud, donde los resultados de la aptitud media de los individuos son muy similares a los anteriores.

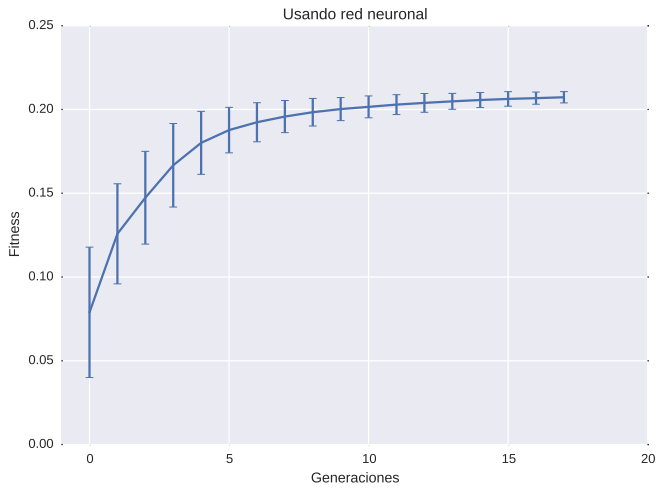


Figura 5.13: Valores de la aptitud media utilizando la red neuronal

La obtención de la RNA tardo aproximadamente dos semanas, debido a la complejidad del problema. Además, si intentábamos obtener unos nuevos ingredientes para calcular la función de la aptitud, podíamos comprobar que no nos servían, ya que habían sido entrenados para otros datos, y volver a calcularlo era demasiado costoso. Por todo ello, abandonamos la idea de emplear la RNA para aproximar la función de aptitud.

5.10. Aplicación del algoritmo genético diseñado

Tras la realización de los experimentos y pruebas planteadas, nuestro AG emplea una codificación en la que un individuo puede tomar unos valores desde 1 hasta 9999 en saltos de 100 en 100. Nuestro AG diseñado posee un cálculo de la función de aptitud nuevo, que nos ayuda a encontrar mejores resultados. También posee el operador de cruce BLXα y un operador de mutación que nos ayuda a mantener la diversidad genética. Además, utiliza un método para seleccionar a los individuos para ser cruzados por torneo. Por último hemos hemos paralelizado el AG con la interfaz MPI para disminuir el tiempo de ejecución.

Tras aplicar el AG, obtuvimos la configuración de pesos que nos proporciona la mayor tasa de aciertos, ésta será la que muestra la figura 5.14.

c1	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**	**c10**	**c11**
9401	5801	4701	4801	3401	2801	6801	5101	1101	5001	901

Figura 5.14: Configuración del campo de características final

Con este concepto de pesos se obtenía una tasa de aciertos del 55 %, que es una mejora aceptable del 40 % que se obtenía al buscar directamente en MongoDB sin emplear la división en campos de características. También hemos podido comprobar que aplicar unos pesos adecuados en la división de los campos de características mejora considerablemente la tasa de aciertos, teniendo inicialmente un 36 % tasa de aciertos hemos conseguido aumentarla hasta una tasa de aciertos del 55 %.

Capítulo 6

Heurísticas

Tras haber configurado adecuadamente MongoDB para almacenar los ingredientes usando campos de características, así como de aplicar los diferentes pesos de prioridad que nos proporciona el AG diseñado para tal fin, obteníamos una tasa de aciertos del 55 %. A la vista de los resultados, entendíamos que no podíamos mejorar más solo estableciendo pesos de prioridad a las diferentes campos de características. Por ello, decidimos aplicar diversas heurísticas para que nos ayudaran a mejorar el número de aciertos. Para realizar estas heurísticas se utilizó la configuración de los campos de características que muestra la figura 5.14, fruto de la aplicación de nuestro AG, que obtenía la tasa de aciertos del 55 %. En este capítulo explicaremos las diferentes heurísticas que se desarrolló en este trabajo. Cabe destacar que estas heurísticas fueron aplicadas a los resultados obtenidos tras realizar la búsqueda en la base de datos, más concretamente sobre los diez primeros resultados, debido a que en el 98 % de los ingredientes se encontraba el resultado esperado en esos diez primeros resultados.

6.1. Eliminación de grupos no predominantes

La idea principal de esta heurística era eliminar de los resultados obtenidos en la base de datos, aquellos que pertenecieran a un grupo minoritario. Con esta heurística se buscaba eliminar los resultados que pertenecen a un grupo no predominante, suponiendo que, al realizar una búsqueda, el grupo del resultado esperado coincide con el grupo predominante de la búsqueda.

Como podemos ver en el código de la figura 6.1 un ejemplo de la aplicación de esta heurística, donde eliminaríamos el primer resultado y obtendríamos uno de los resultados esperados.

```
1  Search:"red wine"
2  Exit:
3          C1          ,     C2     ,  C3  ,         C4         Description Food Group
4  // Vinegar         , red wine                                    Spices and Herbs
5  Alcoholic beverage ,    wine    , table , red                      Beverages
6  Alcoholic Beverage ,    wine    , table , red Syrah                Beverages
7  Alcoholic Beverage ,    wine    , table , red Barbera              Beverages
8  Alcoholic Beverage ,    wine    , table , red Zinfandel            Beverages
9  Alcoholic Beverage ,    wine    , table , red Claret               Beverages
10 Alcoholic Beverage ,    wine    , table , red Lemberger            Beverages
11 Alcoholic Beverage ,    wine    , table , red Sangiovese           Beverages
12 Alcoholic Beverage ,    wine    , table , red Carignane            Beverages
13 Alcoholic Beverage ,    wine    , table , red Burgundy             Beverages
```

Figura 6.1: Ejemplo de la búsqueda vino tinto (*red wine*) en la base de datos, aplicando la heurística de eliminación de grupos no predominantes

Después lanzar unos test de búsqueda obtenemos que los resultados obtenidos son peores que los anteriores, acertando solo un 49 % de los ingredientes. Estos resultados se deben a que hay una parte de ingredientes en la que gracias a esta heurística encontramos el resultado esperado, pero hay una mayor parte de ingredientes que se encuentra su resultado esperado en los campos minoritarios, por lo que al quitarlos no encuentra este resultado. A pesar de ser una heurística con la que parecía que obtendríamos unos buenos resultados, pudimos comprobar que no era así.

6.2. División de los campos de características

Como ya habíamos comentado en la sección 1.1, los ingredientes de la base de datos tenían once campos de características. La idea que seguía esta heurística era dividir por palabras cada campo de característica, eliminando las *stopwords* y crear unos nuevos campos de características donde solo hay una palabra. Pudimos comprobar que como máximo habíamos ampliado el campo de búsqueda hasta veinticinco. Una vez realizada esta modificación buscábamos el ingrediente en estos campos de búsqueda dándole mayor prioridad cuanto más próximo estaba al primer campo. Obtenía el valor cien si estaba en el primer campo o se le aplicaba un decremento de cinco, a este valor, a medida que va aumentando la distancia en la que se encontraba. Es decir, si se encuentra en el segundo campo valdría noventa y cinco, si se encuentra en el tercero valdría noventa y así sucesivamente. En el caso de que el ingrediente que buscáramos en la base de datos estuviera compuesto por más de una palabra se aplicaría este método por cada una de las palabras que contuviese.

Por ejemplo, uno de los resultados al buscar el ingrediente "*pollo*" está compuesto de dos campos de características, donde el primero contiene "*comida rápida*", y el segundo "*trozos de pollo*". Aplicando esta heurística ahora dispondríamos de cuatro campos de características que serían: "*comida*", "*rápida*", "*trozos*" y "*pollo*", por tanto, este resultado tendría un valor de 85. En el ejemplo de la figura 6.2, podemos observar cómo trabajaría esta heurística buscando el ingrediente "*cilantro*". Podemos observar cómo decide escoger el tercer resultado, ya que es el que más puntuación obtiene.

```
1 Search:"coriander"
2 Exit:
3     C1     ,      C2     ,    C3    ,    C4   ,     C5     ,     C6      Value
4 Spices     , coriander ,  seed                                    95
5 Spices     , coriander ,  leaf   , dried                          95
6 Coriander  , cilantro  , leaves , raw                            100 <- Winner
7 Seasoning  ,    mix    ,   dry   , sazon , coriander , annatto     80
```

Figura 6.2: Ejemplo de la búsqueda cilantro (*coriander*) en la base de datos, aplicando la heurística de división de los campos de características

Tras realizar las pruebas correspondientes a esta heurística, pudimos observar como los resultados mejoraban ligeramente y ahora obteníamos un 56 % de aciertos, por lo que decidimos seguir aplicando y diseñando más heurísticas.

6.3. Búsqueda sobre la descripción del ingrediente

Siguiendo con las heurísticas, planteamos hacer una nueva donde ya no necesitaremos los campos de características. Esta nueva heurística junta en una sola frase toda la información de los campos de características (eliminando comas y símbolos extraños), después realiza una búsqueda del ingrediente en esta frase, de manera que coincida exactamente con el ingrediente que buscamos. Inicialmente sabemos que, como máximo, un ingrediente en la base de datos USDA contiene 131 caracteres. Si el ingrediente coincide exactamente desde el primer carácter de la frase creada tendrá un valor de 150, mientras que si se encuentra en el segundo carácter tendrá un valor 149, y así sucesivamente. En el caso de que no se encuentre el ingrediente no sumaremos ningún valor.

Esta heurística, como en la heurística de la sección anterior (sección 6.2), puntúa mejor si encuentra los ingredientes en una posición más próxima, pero a diferencia de ésta, ahora buscamos el ingrediente completo. Si el ingrediente se componía de varias palabras se hacia una búsqueda con cada una de ellas. Esta vez se realizaría una búsqueda con todas las posibles combinaciones y después realizaríamos una media de los resultados.

Podemos observar en la figura 6.3 un ejemplo de la búsqueda del ingrediente "*puré de patatas*" donde, tras realizar las posibles combinaciones de su nombre, buscamos sobre los resultados obtenidos a través de la base de datos. Una vez calculados los valores para cada una de las combinaciones hacemos la media de estas, y el que mayor valor tenga será el elegido como solución, o en caso de empate, el primero.

```
1  Search:"mashed potatoes"
2  Possible combinations: "mashed potatoes", "potatoes mashed"
3  Exit:
4                                                Value   C1  C2 Total
5  Turkey mashed potatoes gravy assorted vegetables      136   0    68
6  Babyfood mashed cheddar potatoes & broccoli toddlers    0   0     0
7  Fast foods potatoes mashed                              0 139  69.5
8  Potatoes mashed home-prepared                           0 150    75 <- Winner
9  Potatoes mashed whole milk and margarine added          0 150    75
10 Potatoes mashed dehydrated flakes without milk dry      0 150    75
```

Figura 6.3: Ejemplo de la búsqueda puré de patatas (*mashed potatoes*) aplicando la heurística de búsqueda sobre la descripción del ingrediente

Después de realizar varias pruebas pudimos comprobar como esta heurística no nos proporcionaban buenos resultados. Aplicando esta heurística disminuimos el número de aciertos a un 53 % de los ingredientes.

6.4. Búsqueda orientada a su característica

Después de observar detenidamente los resultados, pudimos comprobar que la mayoría de los ingredientes tenían en alguno de los campos de características la palabra crudo (*raw*) o entero (*whole*). Esta heurística funcionaba de la siguiente forma, inicialmente realizábamos una división de los campos de características para obtener un nuevo campo de características que correspondieran a palabras. Más adelante, puntuaba si encontraba algún resultado con la palabra crudo o entero, si se encontraba en el primer campo de características obteníamos un valor de cien, si se encontraba en el segundo campo de características obteníamos un valor de noventa y cinco, y así sucesivamente. En el caso de que hubiera un empate, nos quedaríamos con el primer resultado. Podemos observar que su manera de actuar es muy similar a la heurística de la sección 6.2, pero esta heurística la podemos tener precalcula, ya que no necesitamos el ingrediente para saber que valores tiene cada resultado en la base de datos.

En la figura 6.4 podemos observar cómo quedaría realizar esta heurística sobre la búsqueda de "*cilantro*". Observamos cómo nos devuelve el tercer resultado, eliminando los demás, ya que es el único que contiene la palabra crudo (*raw*). En este ejemplo el resultado que nos devuelve coincide con el esperado, por lo que funciona correctamente.

```
1 Search:"coriander"
2 Exit:
3     C1     ,      C2     ,    C3    ,    C4   ,     C5     ,     C6      Value
4 Spices     , coriander  ,  seed                                    0
5 Spices     , coriander  ,  leaf    , dried                        0
6 Coriander  , cilantro   , leaves , raw                           85 <- Winner
7 Seasoning  ,    mix     ,    dry   , sazon , coriander , annatto     0
```

Figura 6.4: Ejemplo de la búsqueda cilantro (*coriander*) en la base de datos, aplicando la búsqueda orientada a su característica

Una vez probada la heurística, pudimos comprobar que no nos mejora el porcentaje de aciertos de los ingredientes. Sin aplicar ninguna heurística obteníamos el 55 % de aciertos, mientras que aplicando esta heurística solamente acertábamos un 51 %.

6.5. Heurística conjunta

Tras realizar las heurísticas anteriores, decidimos aplicar una heurística conjunta que contuviera todas las heurísticas que habían conseguido obtener el resultado esperado en más del 50 % de los ingredientes. Dicho esto la heurística final contendrá a la heurística de la división de los campos de características, la búsqueda sobre la descripción del ingrediente y la búsqueda orientada a su característica, correspondientes a las secciones 6.2, 6.3 y 6.4 respectivamente.

Para realizar esta heurística, teníamos que pensar la manera de normalizar los valores de las heurísticas anteriores, para ello dividíamos el valor obtenido por el valor máximo permitido. A continuación se explicarán los valores máximos obtenidos para normalizar estos resultados:

División de los campos de características. Obtenía unos pesos para los resultados obtenidos, en la base de datos, en función de la posición en la que se encontraba el ingrediente. Estos pesos podían ser como máximo cien por cada palabra dentro del ingrediente ($100 * n$ siendo n el número de palabras del ingrediente).

Búsqueda sobre la descripción del ingrediente. En esta heurística obteníamos un valor como máximo de 150, que corresponde a que el ingrediente se encuentre exactamente al principio del resultado.

Búsqueda orientada a su característica. Esta heurística funciona igual que la división de los campos de características, como hacemos una búsqueda de dos palabras (crudo y entero) podemos deducir fácilmente que el valor máximo obtenido será de 200.

Una vez realizada la normalización de los individuos, se deberá calcular un valor de pesos para cada heurística, con esto conseguimos darle prioridad a unas respecto a otras. Para realizar esto probamos a ejecutar unas pruebas donde se ejecutaban todas las permutaciones de los valores comprendidos entre 0 y 1 con saltos de 0,2 para los tres valores de las heurísticas.

En la ecuación 6.1 podemos observar como quedaría el valor de los resultados al aplicar esta heurística final. Cabe destacar que se le aplica la ecuación a cada resultado (i) de la búsqueda de ingredientes, siendo *words* el número de palabras del ingrediente que buscamos. También debemos destacar que

div, *desc* y *charac* hace referencia a aplicar las heurísticas de división de los campos de características, la búsqueda sobre la descripción del ingrediente y la búsqueda orientada a su característica, siendo W, U y V el valor de los pesos que obtendremos al realizar la prueba de las permutaciones.

$$\begin{aligned} Value_i = & W * div(i)/(100 * words) \\ & + U * desc(i)/150 \\ & + V * charac(i)/200 \end{aligned} \tag{6.1}$$

Tras realizar la prueba de las permutaciones vemos que los valores obtenidos no son prometedores, y que el número máximo de aciertos del resultado esperado es el de aplicar únicamente la división de los campos de características, es decir, un 56 %. Los resultados que obteníamos al mezclar las heurísticas empeoraban los resultados, como podemos observar en la figura 6.5.

W	**U**	**V**	**Porcentaje de aciertos (%)**
0.2	0.0	0.0	56
0.4	0.0	0.0	56
0.6	0.0	0.0	56
0.8	0.0	0.0	56
1.0	0.0	0.0	56
0.0	0.0	0.0	55
0.2	0.4	0.2	54
0.2	0.8	0.4	54
0.4	1.0	0.2	54

Figura 6.5: Porcentaje de aciertos del resultado esperado al realizar la búsqueda de los ingredientes, siendo W, U y V el valor de los pesos que obtendremos al realizar la prueba de las permutaciones sobre la ecuación 6.1

Capítulo 7

Conclusiones

En las recetas de cocina nos podemos encontrar diferentes formas de representar a los ingredientes. Este trabajo proponía aplicar diferentes técnicas soft computing para la búsqueda de diferentes ingredientes extraidos de las recetas frente a la base de datos de referencia mundial USDA. Como objetivo seguíamos la línea de establecer la mejor relación posible entre los ingredientes de la receta y la base de datos USDA.

En este capítulo se presentarán los objetivos cumplidos en este trabajo, además del posible trabajo futuro que se puede realizar.

7.1. Objetivos cumplidos

De acuerdo con los objetivos indicados inicialmente en este trabajo, hemos diseñado un algoritmo de búsqueda, concretamente un AG, que nos proporcione la mejor relación posible entre los ingredientes de la receta con la base de datos.

El primer problema que intentábamos resolver es que, debido al diseño de base de datos que nos proporciona USDA, cada ingrediente tenia diferentes campos de características, con un máximo de once campos. Si estos campos siguieran una estructura organizada podría ser sencillo realizar una búsqueda precisa, pero debido a la desestructuración de esta base de datos, necesitábamos desarrollar un algoritmo de búsqueda que nos proporcionara una combinación de pesos para poder encontrar el resultado esperado. Para resolver este problema diseñamos un AG para encontrar la mejor relación entre las prioridades de los campos de características que nos devuelva una mejor tasa de aciertos.

Para agilizar el cálculo de la función de la aptitud emprendida por el AG decidimos utilizar una RNA. Esta red obtenía rápidamente el valor aproximado de la aptitud, pero para diseñarla tardamos más de lo que esperábamos. Observamos que únicamente nos servia para aproximar el valor dado unos ingredientes específicos, pero que si deseábamos cambiar de ingredientes no nos servia y debíamos obtener y entrenar otra RNA. Además, el coste de obtener el método de aproximación era mayor que emplear directamente la función de aptitud.

Inicialmente, sin aplicar ningún tipo de mejora acertábamos un 36 % de los ingredientes que tratábamos de buscar, después de diseñar y aplicar nuestro AG conseguimos aumentar esta tasa de aciertos hasta un 55 %, que supone una mejora del 53 % de los aciertos iniciales. A pesar de realizar varios experimentos, pudimos observar que por muchas modificaciones que hiciéramos en el AG o directamente en las prioridades de los campos de características de los ingredientes, no podíamos mejorar la tasa de aciertos, por lo que decidimos aplicar distintas heurísticas sobre los resultados ofrecidos por la consulta a la base de datos.

Probamos un total de cinco heurísticas que aplicábamos a los resultados obtenidos por la base de datos tras realizar una búsqueda. Pudimos comprobar que en el 98 % de los casos, se obtenía los resultados dentro de los diez primeros resultados, de los cien que nos devolvía la base de datos. Decidimos aplicar estas heurísticas para encontrar el resultado esperado en los primeros diez resultados devuelto por la base de datos.

De todas las heurísticas implementadas, pudimos observar que únicamente obteníamos mejoras con una de ellas, la división de los campos de características. Esta heurística dividía las características del ingrediente por palabras y buscaba cada palabra del ingrediente sobre esta división. Cuanto más al próximo del inicio de la división se encontrara la palabra mayor puntuación obtenía el resultado. Con esta heurística mejorábamos hasta obtener una tasa del 56 % de aciertos, que supone una mejora del 56 % de los aciertos iniciales.

7.2. Líneas futuras de actuación

En este trabajo hemos trabajado con diversas técnicas *soft computing*, como son los algoritmos genéticos o las redes neuronales, y hemos desarrollado diversas heurísticas. Visto que las relaciones de pesos no se pueden mejorar, las líneas futuras de actuación para mejorar los resultados sobre este tema serán tres:

(a) **Crear una nueva base de datos.** Al crear una nueva base de datos más estructurada que la anterior quizás se consiga obtener unos mejores resultados en la búsqueda de ingredientes. Se podría hacer un análisis de los ingredientes y ver como se podría organizar en una nueva base de datos. Podríamos utilizar la base de datos MongoDB, y así podríamos aprovechar la característica de mongo sobre las búsquedas de texto. Se podrían comparar los resultados de ambas bases de datos y comparar que hace que la nueva sea peor o mejor que la anterior.

(b) **Clasificación de los ingredientes.** Podríamos realizar una clasificación de los ingredientes, con los que obtenemos resultados parecidos, y tratar de mejorar el resultado de su búsqueda.

(c) **Aplicar nuevas heurísticas.** Una de las posibles líneas de actuación seria la de aplicar nuevas heurísticas a los resultados obtenidos. Podríamos aplicar una nueva heurística que se quede con aquellos resultados que no excedan de un número fijo de palabras. Esta heurística podría ser interesante tratar debido a que cuanto más información contenga la descripción del ingrediente más probabilidad tendrá de ser un resultado no esperado. Aplicar heurísticas puede resultar un trabajo bastante complejo, ya que mejoraremos las soluciones de algunos ingredientes, pero también podemos empeorar el de otros.

Bibliografía

[Blickle and Thiele, 1995] Blickle, T. and Thiele, L. (1995). A comparison of selection schemes used in genetic algorithms. Technical report, Gloriastrasse 35, CH-8092 Zurich: Swiss Federal Institute of Technology (ETH) Zurich, Computer Engineering and Communications Networks Lab (TIK.

[Eshelman et al., 1989] Eshelman, L. J., Caruana, R. A., and Schaffer, J. D. (1989). Biases in the crossover landscape. In *Proceedings of the third international conference on Genetic algorithms*, pages 10–19, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Eshelman and Schaffer, 1993] Eshelman, L. J. and Schaffer, J. (1993). Real-coded genetic algorithms and interval-schemata. In Whitley, L. D., editor, *Foundations of Genetic Algorithms 2*, pages 187–202, San Mateo,CA. Morgan Kaufmann Publishers.

[Goldberg, 1989] Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

[Goldberg, 1991] Goldberg, D. (1991). Real-coded genetic algorithms, virtual alphabets, and blocking. *Complex Systems*, 5:139–157.

[Goldberg and Deb, 1991] Goldberg, D. E. and Deb, K. (1991). *A comparative analisys of selection schemes used in genetic algorithms*, pages 69–93. Morgan Kaufmann, San Mateo.

[Herdy, 1991] Herdy, M. (1991). Application of the evolutions strategy to discrete optimization problems. *Parallel problem solving from nature*, pages 187–192.

[Herrera et al., 2002] Herrera, F., Lozano, M., Pé, E., Sánchez, A., and Villar, P. (2002). Multiple crossover per couple with selection of the two best offspring: An experimental study with the blx-α crossover operator for real-coded genetic algorithms. *IBERAMIA 2002, LNAI, Springer Verlang Berlin Heidelberg*, 2527:392–401.

[Jin, 2005] Jin, Y. (2005). A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 9(1):3–12.

[Michalewicz, 1992] Michalewicz, Z. (1992). *Genetic algorithms+data structures=evolution programs*. Springer-Verlang, New York.

[Radcliffe, 1991] Radcliffe, N. J. (1991). Equivalence class analysis of genetic algorithms. *Complex Systems*, 5:183–205.

[Ripley, 1996] Ripley, B. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge.

[Rosenblatt, 1962] Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books.

[Sywerda, 1989] Sywerda, G. (1989). Uniform crossover in genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 2–9, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Valero Cubas, 2010] Valero Cubas, S. (2010). *Arquitectura de búsqueda basada en técnicas soft computing para la resolución de problemas combinatorios en diferentes dominios de aplicación*. PhD thesis, Universitat Politècnica de València.

[Zadeh, 1994] Zadeh, L. (1994). Fuzzy logic, neural networks and soft computing. *Communications of the ACM*, 37(3):77–84.