Universidad Central “Marta Abreu” de Las Villas

Facultad de Matemática Física y Computación

Módulo de Experimentación Combinatoria para la Evaluación de Modelos Neuro-Borrosos Asociativos

Autores

Aylín Labrador Morales

Angel Miguel Navarro Moya

Tutores

MSc. Yanet Rodríguez Sarabia

MSc. Rafael Jesús Falcón Martínez

Santa Clara, 2006

Hago constar que el presente trabajo fue realizado en la Universidad Central Marta Abreu de Las Villas como parte de la culminación de los estudios de la especialidad de Ciencias de la Computación, autorizando a que el mismo sea utilizado por la institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos ni publicado sin la autorización de la Universidad.

_____

Firma del autor

Los abajo firmantes, certificamos que el presente trabajo ha sido realizado según acuerdos de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____ _____

Firma del tutor Firma del jefe del Seminario

El esfuerzo sólo proporciona plenamente su recompensa, después de que una persona se niega a darse por vencida.

Napoleón Hill

A mi mamita, la mejor del mundo

A mi papá, siempre cerca de mi corazón

A mi hermanita que ya creció pero sigue siendo mi hermanita

A la más linda de las abuelas

Aylín

A mi madre, por estar siempre ahí, a mi lado, en lo bueno y en lo malo.

A mi padre, por ser fuente de inspiración para mí.

A mi hermana, por ser todo un ejemplo de dedicación y entrega.

Angel

Agradecimientos

A mis tutores Yanet y Falcón por el esfuerzo que hicieron para ver este trabajo terminado.

A mi mamá , por todo.

A mi papá por estar al tanto de cada detalle.

A Annia por hacerme sentir una buena hermana mayor.

A mi abuela por quererme tanto

A Danilo por ayudarme en estos años como a cualquiera de sus hijos

A todos mis tíos y primos que siempre confiaron en mi

A Puchi por ayudarme a llegar hasta aquí y a tener confianza en mi

A Angel por soportarme todo este tiempo (debe haber sido más difícil que programar en C#)

A Ile y a mi sobri, a Sure y a Gill que aunque están lejos no me han dejado sola ni un momento.

A Damnito por estar siempre conmigo.

A Eikel por hacerme reir

A Yuneisy, Yunelsy, Dania, Annelis , Erylis, Anay, Liucel, Michel, Maikel y Lisbel por ser tan buenos amigos y apoyarme siempre.

A Abdel, Alain y El Chino por sus oportunas consultas.

A todos los que me ayudaron en estos difíciles años, profesores y compañeros de aula.

Aylin Labrador Morales

Agradecimientos

A mis tutores Yanet y Falcón por brindarme tanta ayuda y parte de su tiempo

A mi madre, mi padre y mi hermana por su apoyo incondicional y confianza en mí a lo largo de todos estos años.

A mi abuela paterna, que aunque no se encuentre a mi lado en estos momentos, siempre me dio su cariño como si fuera mi propia madre.

A mi novia Anileydi, por brindarme tanto amor y compresión en los momentos más difíciles.

A mi compañera de tesis Aylín, por su valiosa ayuda durante estos largos meses.

A mis compañeros de aula y año por regalarme tantos momentos gratos, en especial a Lisett, Damny, Michel y Maikel.

A todos mis amigos que de una forma u otra han contribuido a alcanzar esta meta, en especial a Alain Varela, Abdel y Roberto, “el chino”, por tantas consultas y paciencia conmigo.

A todas las personas que han compartido conmigo estos cinco años, a todos, gracias.

Angel Miguel Navarro Moya

Resumen

El presente trabajo consiste en el desarrollo de un módulo de experimentación combinatoria para incorporarlo a la plataforma computacional denominada *NeuroEvaluator* 2.0, que es capaz de validar la eficiencia de cuatro modelos de Redes Neuronales Asociativas implementados y empotrados en ella. Este módulo permite encontrar una configuración de los grados de libertad de un modelo de red ante un conjunto de datos de entrada, con la que dicho modelo garantice un buen desempeño. Para ello fue necesario implementar un módulo para el preprocesamiento de atributos simbólicos y numéricos, en el que se forman los grupo de neuronas con que trabaja posteriormente la Red Neuronal Asociativa; diseñar e implementar un sistema distribuido de tareas, utilizando las facilidades que brinda .Net Remoting, con un modelo cliente servidor, para evaluar al mismo tiempo variantes de solución para un mimo conjunto de datos; y una heurística para explorar solamente las combinaciones de las variantes de solución más prometedoras.

Abstract

The present paper consists on the development of a module of combinatory experimentation to be incorporated to a computing platform known as NeuroEvaluador 2.0 which is able to validate the efficacy of four models of Associative Neuronal Networks implemented and embodied in it. This module allows us to find a setup for the degrees of freedom of a network model before of a set of entry data, with which the model is expected to guarantee a good performance. For this purpose it was necessary to implement a module for the processing of symbolic and numerical attributes, in which the group of neurons with which the Associative Neuronal Networks is expected to work afterwards are formed. It was also necessary to design and implement a distributed system of tasks, using the facilities that .Net Remoting offers, with a user-server model, so as to assess, at the same time, a number of solution variants for a unique set of data; and a heuristics only to explore the combination of the most promising solution variants.

Índice

Introducción

Entre las técnicas de solución de problemas de la Inteligencia Artificial (IA), se encuentran las Redes Neuronales Artificiales (RNA). Las RNA no son más que un modelo artificial y simplificado del cerebro humano, que es el ejemplo más perfecto del que disponemos para un sistema que es capaz de adquirir conocimiento a través de la experiencia. Debido a su constitución y a sus fundamentos, las RNA presentan un gran número de características semejantes a las del cerebro. Por ejemplo, son capaces de aprender de la experiencia, de generalizar de casos anteriores a nuevos casos, de abstraer características esenciales a partir de entradas que representan información irrelevante, etc. Esto hace que ofrezcan numerosas ventajas y que este tipo de tecnología se esté aplicando en múltiples áreas.

Dentro del área vasta y compleja de la resolución de problemas, los modelos asociativos han aportado una parte considerable de la eficiencia en la solubilidad lograda a escala mundial por las RNA (Bello 2002). Este es un tema actual de investigación del Grupo de IA de la UCLV, que tuvo su inicio en la década del 90 con el desarrollo de un modelo híbrido simbólico-conexionista en el cual una RNA de este tipo se combina con un enfoque basado en casos (García, 1996). La topología de los modelos de RNA asociativos se forma por un grupo de neuronas por cada rasgo del dominio del problema. En estos grupos se coloca una neurona por cada valor que aparezca para ese rasgo en el conjunto de datos. Existen enlaces entre las neuronas de grupos diferentes. El problema con este tipo de topología surge cuando el rasgo tiene un dominio infinito o muy grande (García et al., 2000). Para los rasgos con esta característica se definen valores representativos, que son a los que se les coloca una neurona en el grupo.

Varios modelos de RNA asociativas fueron implementados en una herramienta denominada *NeuroDeveloper* (Falcón, 2003), donde era factible el procesamiento de atributos de tipo simbólico y numérico. Para estos últimos la selección de los valores representativos era enteramente manual, guiada por el criterio experto en el dominio de aplicación. Posteriormente se desarrolló el software *NeuroEvaluator* (Falcón 2006),

como una plataforma computacional para la evaluación de modelos neuro-borrosos asociativos. La herramienta tiene implementados cuatro modelos de RNA: IAC, SIAC, Hopfield y Brain-State-in-a-Box (BSB). En cada uno de ellos se especifica un modelo de neurona. Todos utilizan un aprendizaje basado en la regla de Hebb, y en particular se definen tres variantes para el cálculo de los pesos: Frecuencia Absoluta, Frecuencia Relativa y Coeficiente de Correlación de Pearson.

Para un conjunto de entrenamiento, *NeuroEvaluator* facilita encontrar cuál de estos modelos es el que mejor desempeño tiene. En esto no influyen solamente las características antes mencionadas de los modelos de red implementados en la herramienta, sino también la forma de codificar los datos que sirven de entrada a la RNA. Nótese que este último aspecto está estrechamente relacionado con la topología del modelo de red seleccionado.

El hecho de que un atributo numérico pueda ser modelado como una variable lingüística a partir de la discretización de este rasgo, presupone más grados de libertad a considerar en la búsqueda de un desempeño adecuado ante un conjunto de datos. Encontrar la mejor forma de codificar los datos de entrada para lograr un buen desempeño del modelo de red seleccionado, podría significar probar un alto número de combinaciones de las variantes de codificación (explosión combinatoria). Si esto se ejecuta en una sola estación de trabajo resultará muy demorado, y una variante alternativa pudiera ser hacer un procesamiento distribuido de tareas en varias terminales conectadas en red. No considerar todos los valores posibles para cada grado de libertad, es otra variante a considerar. Esto se logra si el problema de definir en *NeuroEvaluator* la variante más adecuada ante un archivo de datos, se modela como un problema de configuración que utilice una heurística para explorar solamente las configuraciones más prometedoras. Atendiendo a lo anteriormente expuesto, el presente trabajo tiene por:

Objetivo general

Desarrollar un Módulo de Experimentación Combinatoria para la plataforma NeuroEvaluator 2.0.

Objetivos Específicos

1. Diseñar e implementar un módulo para el preprocesamiento de atributos simbólicos y numéricos.

2. Diseñar e implementar un sistema distribuido que permita explotar los recursos de la red.

3. Proponer e implementar una heurística para explorar solamente las combinaciones de las variantes de codificación más prometedoras.

Este trabajo se presenta en tres capítulos, en cada uno de los cuales se aborda un tópico requerido en la realización del trabajo. El primer epígrafe del capítulo se refiere a la revisión bibliográfica, mientras que el segundo especifica la solución adoptada en la implementación. El capítulo 1 aborda el preprocesamiemto de los datos, el segundo se refiere al procesamiento distribuido de tareas, mientras que el último se dedica a formular y resolver este problema como un problema de búsqueda en IA.

CAPÍTULO 1. MÓDULO DE PREPROCESAMIENTO

La codificación de las entradas a la RNA está determinada por el preprocesamiento de los rasgos y la selección de una estrategia de asignación. El preprocesamiento de los rasgos determina la topología de la RNA asociativa. Atendiendo a la influencia que ejercen estos dos aspectos en el desempeño de la RNA, y a la necesidad de codificar los datos que le sirven de entrada, se diseñó e implementó el Módulo de Preprocesamiento de los Rasgos.

A continuación describen las formas de tratamiento de los rasgos numéricos, así como el diseño y la implementación de dicho módulo.

1.1 Tratamiento de los rasgos numéricos

El agrupamiento de un conjunto de patrones mediante la utilización de alguna métrica de similitud que resulte adecuada es un problema bastante estudiado en Inteligencia Artificial, también conocido como aprendizaje no supervisado en contraste con el aprendizaje supervisado realizado al construir modelos de clasificación a partir de ejemplos previamente etiquetados.

Para modelar atributos numéricos se puede seguir un enfoque duro utilizando discretizadores, o un enfoque borroso si estos rasgos se modelan como variables lingüísticas (Buckley, 2002).

Según (Östermark, 2000; Zadeh 1994), la lógica borrosa surge como la posibilidad de flexibilizar el arcaico concepto de que una cosa “o es, o no es”, pues en la mayoría de los problemas de la matemática, estas decisiones duras son inconsecuentes y el principio del tercio exclusivo nos puede llevar a un callejón sin salida ante una decisión en la que exista un por ciento de verdad y otro de falsedad (Falcón 2006).

Las técnicas de modelación borrosa tienen su fundamento en la definición de variables lingüísticas, en las que el universo de discurso es el dominio original del rasgo, y sus términos lingüísticos serán los valores representativos del rasgo (García et al., 2000). En

el rasgo Edad la variable lingüística correspondiente tiene como universo los números enteros y sus términos lingüísticos podrían ser los siguientes:

{*niño, joven, adulto*}

Para cada término lingüístico se define un conjunto borroso mediante su función de pertenencia (FP). Por ejemplo:

$$niño = B(x,\gamma,\beta) = \frac{1}{1+\left(\frac{x-\gamma}{\beta}\right)^2} \qquad B(x,165,5) = \frac{1}{1+\left(\frac{x-165}{5}\right)^2} \qquad (1)$$

Los tipos de FP más utilizadas dentro de la lógica borrosa son (Nauck, 1997):

- FP triangular.
- FP trapezoidal.
- FP gaussiana.
- FP sigmoidal.
- FP beta.
- FP campana.

Hablar de *niño* es más natural que decir que el valor 7 años representa a los valores próximos a él, lo que muestra una ventaja de usar esta alternativa. Además se puede determinar en qué medida un valor esta próximo a lo que dicho valor representa .Otra ventaja se manifiesta en el uso de calificadores (en inglés, hedges) para generar nuevos términos lingüísticos.

Al igual que en el lenguaje natural existen los adjetivos y los adverbios para calificar nombres, acciones, etc., se pueden colocar calificadores para los términos lingüísticos representados por conjuntos borrosos. Un calificador transforma un conjunto borroso en otro. Ejemplos de clasificadores son: muy, extremadamente, algo, bastante, más o menos.

A grandes rasgos, el efecto de un calificador sobre un conjunto borroso es intensificar, diluir o negar la función de pertenencia del mismo. Por ejemplo, *muy* intensifica la función, mientras que *algo* diluye la función, y *not* la niega.

Modelar atributos numéricos en forma de conjuntos borrosos ha servido como una de las herramientas para resolver problemas en presencia de vaguedad como el ejemplo que se vio anteriormente.

Dividir los valores de un atributo continuo en un conjunto de intervalos adyacentes corresponde al caso unidimensional de los métodos de agrupamiento. Este caso, conocido como discretización, es de especial importancia en Inteligencia Artificial, pues permite que muchos algoritmos de aprendizaje ideados para funcionar con atributos nominales o categóricos puedan también utilizarse con conjuntos de datos que incluyen valores numéricos (Hussain et al., 1999), algo esencial en la resolución de problemas reales.

En este caso se divide el rasgo en intervalos según el criterio de un experto o utilizando métodos automáticos. La correcta determinación de dichos intervalos se corresponde con el método empleado con este objetivo. Algunos métodos de discretización conocidos son:

- Equal Width (Hussain, 1999)
- Equal Frequency (Hussain, 1999)
- Chi-2 (Liu, 1997)
- CAIM (Kurgan 2004)
- K-Means (Jyh-Shing , 1998)

El primero de estos métodos de discretización es conocido en la literatura como "método de igual longitud" (Equal Width). Éste, divide el intervalo continuo en k particiones de igual tamaño mediante la fórmula siguiente:

$$Amplitud = \frac{\max - \min}{k} \qquad (2)$$

Es un método muy rápido y su implementación es sencilla, ventajas estas que lo caracterizan. Pero este método es superfluo, ya que no toma en cuenta la distribución real de los datos en la base de información.

Similar al anterior es el método conocido como "de igual frecuencia" (Equal Frequency). Con esta variante de discretización se logra que todas las particiones tengan la misma cantidad de elementos. Esto responde a la fórmula:

$$Frecuencia = \frac{N}{K} \qquad (3)$$

La N representa la cantidad de elementos del intervalo inicial [a, b] y la K, la cantidad de clases que se quieren formar con la discretización.

Se obtendrán k intervalos, que no tienen que ser de la misma longitud, pero sí tendrán la misma cantidad de elementos, es decir, la misma frecuencia de aparición de los valores. Sin embargo este método, aunque aparentemente tiende a considerar los elementos del intervalo a la hora de ordenarlos para determinar la frecuencia, en realidad no tiene en cuenta la distribución de los valores en el atributo.

Otra variante de discretización es el Chi-2, que discretiza atributos numéricos basados en el estadístico χ^2, y además permite eliminar los atributos redundantes y chequear inconsistencias. Se inicia considerando cada valor como un intervalo y se realiza un proceso de mezclado hacia arriba (bottom-up) hasta que se cumpla la condición de parada. Una discretización excesiva introduce muchas inconsistencias, por lo que se suministra como parámetro el por ciento de inconsistencias permitidas.

El CAIM, es otra de las variantes de discretización conocidas, con ella se discretiza un atributo en el menor número posible de intervalos y se maximiza la interdependencia entre el atributo y el rasgo objetivo (clase). Es el propio algoritmo quien selecciona de manera automática (al igual que el Chi-2) el número de intervalos discretos en los que

quedará finalmente particionado el atributo. Este método parte de un conjunto inicial que contiene un único intervalo: todo el dominio del atributo, y calcula los posibles valores a ser añadidos de uno en uno (top-down), hasta formar el esquema final de discretización. La adición de un nuevo valor al esquema se efectúa si dicho valor satisface el criterio de alcanzar la mejor interdependencia clase-atributo posible.

Se conoce además el método K-Means, también conocido como C-Means Clustering, el mismo particiona una colección de N vectores x_j, j = 1,…, N en c grupos o clusters G_i , i = 1,…, c y encuentra un centro de cluster en cada grupo de forma tal que se minimice una cierta función de costo. Dicha función de costo se calcula a partir de una distancia entre dos vectores, para lo cual proponemos dos medidas de distancia: la euclidiana y la de Manhattan (Wilson, 1997).

Es conveniente seguir este enfoque cuando se conoce, según criterios de expertos, que los intervalos en que puede ser dividido el rasgo a modelar, están bien delimitados.

1.2 Módulo de Preprocesamiento, diseño e implementación

El preprocesamiento de la información es la traducción del lenguaje natural en que se expresa el conocimiento experto en la base de casos, al lenguaje simbólico en que se representa el rango de valores que pueden tomar las neuronas de la RNA. El proceso de traducción de un lenguaje a otro está fuertemente condicionado por la forma en la que se hayan definido los tipos de rasgos.

En una BC aparecen valores como “lunes” del atributo discreto “Día” y “1.65” del atributo numérico “Talla”, los mismos tienen que ser convertidos en valores que la red pueda manipular y procesar, de acuerdo a su modelo de activación. Usualmente estos son binarios {0, 1} o bipolares {-1, 1}, aunque las neuronas correspondientes a los modelos continuos admiten un rango de valores.

Este proceso se desarrolla en el Módulo de Preprocesamiento que se encarga de conformar los clusters o grupos de neuronas de los que dispondrá la red, atendiendo al tipo de rasgo en específico. Esto arroja como resultado una matriz de preprocesamiento *MA*, cuyos elementos representan la activación inicial de cada neurona preprocesadora a

partir del valor presente en la BC., o sea, si se está lidiando con una base de casos de *M* ejemplos y fueron generadas *V* unidades preprocesadoras de la información, entonces la matriz inicial *MA* es de dimensión *M xV*.

Las unidades de preprocesamiento se generan de acuerdo al tipo de atributo:

- **Atributos simbólicos:** Se creará una unidad por cada valor simbólico del dominio del atributo. La activación de la neurona que representa al i-ésimo valor del dominio se fijará en uno si dicho valor está presente en el atributo para el caso actual, en caso contrario se asignará cero.

  Expresando matemáticamente la función de asignación para este tipo de neuronas, queda como sigue:

$$f(x_i) = \begin{cases} 1 & si\ x_i \in P.X \\ 0 & en\ caso\ contrario \end{cases} \qquad (4)$$

  Siendo X_i el i-ésimo valor del atributo X y $P.X$ el conjunto de valores para el problema en el atributo.

  La función facilitará el tratamiento de atributos multi-evaluados, pues si se trabaja con el atributo *Síntomas*, cuyo dominio de valores es el conjunto {fiebre, vómito, diarrea} y un ejemplo de la base de conocimientos para ese atributo tiene el valor "fiebre y diarrea", se activarán, según esta función, las neuronas correspondientes a los dos síntomas señalados, quedando con cero la que corresponde al vómito.

  Los nombres de las neuronas para este tipo de atributos coinciden con el de los valores simbólicos que ellas representan.

- **Atributos numéricos:** Los atributos numéricos se tratan de forma diferente, ya que el crear una neurona por cada valor presente en el rasgo no contribuye a mejorar el desempeño de la RNA sino todo lo contrario: aumenta el número de conexiones y se ralentiza el proceso de aprendizaje; lo correcto en este caso es

encontrar valores representativos y en base a estos proceder a la creación de las unidades de preprocesamiento.

- Variante discreta:

Se toman como valores representativos de un atributo numérico, al conjunto de intervalos que resulta de su discretización, creando tantas unidades de preprocesamiento como intervalos haya.

Por ejemplo, sea el atributo edad con las siguientes divisiones: [1-15], (15-32], (32-78], (78-120]. Consecuentemente, se crearían cuatro unidades de preprocesamiento y tomarían valores binarios, pero a diferencia de las unidades correspondientes a los rasgos simbólicos, estas serían mutuamente excluyentes entre sí, dado que un mismo valor numérico no puede pertenecer a más de un intervalo a la vez. Su función de asignación sería:

$$f(x_i) = \begin{cases} 1 & si\ \inf_0 \le x_0 \le \sup_0 \vee \inf_i < x_i \le \sup_i, \quad i > 0 \\ 0 & en\ otro\ caso \end{cases} \tag{5}$$

Donde:

x_i simboliza el i-ésimo intervalo en que quedó particionado el atributo numérico.

$\inf_i$ y $\sup_i$ son el ínfimo y el supremo, respectivamente, de cada intervalo del atributo.

- Variante borrosa:

La otra variante para tratar un atributo numérico es declararlo como borroso. Se definen entonces el conjunto de términos lingüísticos con sus respectivas FP. El preprocesador añadirá a la topología de la RNA una neurona por cada término lingüístico, cuyo valor de activación inicial se determina por el grado de

membresía que alcance el ejemplo presente en la BC en ese atributo a la FP asociada al término lingüístico.

Supongamos que el atributo "longitud del pétalo" es borroso, y tiene los términos lingüísticos asociados:

PEQUEÑO:	FP TRIANGULAR	con a = 10; b = 15; c=19.5
MEDIANO:	FP TRAPEZOIDAL	con a = 12; b = 14.8; c = 17.5; d = 20
GRANDE:	FP TRIANGULAR	con a = 18.5; b = 21; c=25.5

Para este ejemplo, se crearán tres neuronas preprocesadoras y se asociará a cada una de ellas el nombre del término lingüístico que representan. Además, cada valor presente en la base de casos correspondiente a este atributo, se evaluará para cada una de las neuronas creadas en dependencia de la función de membresía que posea la neurona.

Luego, la función de asignación de un valor borroso no es más que el valor de pertenencia a la curva de la neurona preprocesadora asociada al término lingüístico.

Para la asignación del valor inicial de la neurona preprocesadora no se está utilizando el principio de máxima membresía -el cual consiste en fijar en uno la mayor pertenencia de todas las alcanzadas, y poner cero en los valores de las demás neuronas- sino que a cada neurona de preprocesamiento le asociamos el grado de pertenencia del valor del atributo a su FP. Este criterio en la práctica ha arrojado mejores resultados que el principio antes mencionado. No obstante, sí emplearemos el criterio de máxima membresía como una variante más a la hora de inicializar el vector de entrada externa que le será presentado a la RNA.

El módulo de preprocesamiento está compuesto por los módulos de discretización, de construcción automática de FP y de ajuste de parámetros, además de poseer una estrecha interrelación con el módulo de datos (BC).

1.2.1 Módulo de Discretización.

El módulo de discretización está conformado por los cinco discretizadores:

- Equal Width
- Equal Frequency
- Chi-2
- CAIM
- K-Means

Estos ya existían como bibliotecas de enlace dinámico formando parte del *Neuro Evaluator*. Las dos primeras variantes de discretización no toman en cuenta los valores presentes en la BC. Pueden emplearse en cualquier tipo de problema de asociación, al igual que el algoritmo de agrupamiento K-Means, quien implementa una heurística mucho más compleja.

Como se muestra en (Falcón 2006), las variantes Chi-2 y CAIM contribuyen notablemente a elevar el desempeño de la RNA en comparación con las restantes, pero lamentablemente sólo pueden emplearse en problemas de clasificación donde el rasgo objetivo pueda tomar un solo valor a la vez.

1.2.2 Módulo de construcción automática de FP

En la construcción de sistemas inteligentes que empleen conjuntos borrosos en la modelación de sus datos, juegan un papel muy importante los expertos humanos. Cuando no se cuenta con dichos expertos para que modelen el conocimiento y en particular, definan las FP para cada atributo del problema, o estos se muestran inseguros con respecto al tipo de FP a aplicar, el tiempo de desarrollo puede incrementarse, o los sistemas difusos desarrollados pueden no tener un buen desempeño.

Este módulo utiliza un procedimiento para proponer, de forma automática, a partir de ejemplos de entrenamiento, FP apropiadas. En él se usan métodos de construcción automática de FP:

- Triangulares
- Trapezoidales
- Gaussianas
- Sigmoidales

Dichos métodos ya existían como bibliotecas de enlace dinámico formando del paquete de funciones implementadas para el preprocesamiento (Varela, 2005).

El proceso por el cual se determina si los elementos de un conjunto *X* (Universo de discurso de la variable lingüística) pertenecen o no a un intervalo (crisp set) se puede definir mediante una función característica o discriminante (Hong 1988). La misma se puede extender no solamente para indicar lo anterior, sino para cuantificar en qué medida, utilizando el grado de pertenencia (fuzzy set).

El procedimiento para obtener los conjuntos borrosos asociados a una variable consta de dos etapas. En la primera etapa se definen los términos lingüísticos, particionando el universo de la variable lingüística *X* en grupos. Luego, para cada uno de estos términos se estiman los parámetros del tipo de FP seleccionado. Los conjuntos borrosos asociados a un atributo resultante de este procedimiento deben ser fáciles de entender por los expertos del dominio de aplicación.

Primera Etapa: Selección de términos lingüísticos

El universo de discurso (discreto o continuo) de una variable lingüística *X* asociada a un atributo lineal, está formado por los valores que aparecen para ese atributo en el conjunto de entrenamiento. Este se particiona en grupos, asociando un termino lingüístico a cada uno de los grupos obtenidos. El j-ésimo grupo (Gj) se representa por [Aj, Bj], donde Aj se corresponde con el menor valor del intervalo asociado al grupo Gj y Bj con el mayor.

Esto se logra aplicando cualquier método de discretización o agrupamiento de los ya explicados en el módulo de discretización.

Los tres últimos métodos seleccionados consideran la similitud de los valores, en contraste con otros métodos que definen intervalos de igual amplitud o de igual frecuencia, lo cual propicia definir términos relacionados con los datos que se están procesando

Segunda Etapa: Estimación de los parámetros de las FP.

A continuación se describen las heurísticas propuestas para estimar, a partir de ejemplos, de manera automática los parámetros de una FP. Se enfatiza en la heurística propuesta por Hong (Hong 1988), un método de aprendizaje para derivar automáticamente reglas borrosas y FP de un conjunto de casos de entrenamiento, facilitando la adquisición de conocimiento. La heurística de Hong considera la pertenencia a una clase o no (crisp set), comparando la similitud entre los datos adyacentes con un parámetro de entrada alfa (α). Si la similitud entre estos valores excede el valor de α, entonces estos se ubican en la misma clase; de lo contrario, se ubican en clases diferentes; es decir, este valor determina el umbral de pertenencia a una misma clase (para mayor valor de α se tiene un mayor número de grupos). A cada clase obtenida se asocia un término lingüístico, por tanto, la pericia para proponer un valor adecuado para este parámetro influirá en la modelación que se está haciendo de la variable lingüística.

Para encontrar el valor de similitud entre datos adyacentes, primeramente se ordena el rasgo y se convierte cada diferencia a un número real entre 0 y 1, donde se tiene en cuenta la desviación estándar (σ_s) y otro parámetro de control (C). La heurística que se propone para determinar los tres parámetros que caracterizan una FP triangular es la siguiente:

$$b_j = \frac{y_i * S_i + y_{i+1} * \frac{S_i + S_{i+1}}{2} + \ldots + y_{k-1} * \frac{S_{k-2} + S_{k-1}}{2} + y_k * S_{k-1}}{S_i + \frac{S_i + S_{i+1}}{2} + \ldots + \frac{S_{k-2} + S_{k-1}}{2} + S_{k-1}} \tag{6}$$

$$\mu_j(y_i) = \mu_j(y_k) = \min\{S_i, S_{i+1}, .., S_{k-1}\} \tag{7}$$

$$S_i = 1 - \frac{diff_i}{C * \sigma_s} \tag{8}$$

Donde: $diff_i = y_{i+1} - y_i$

$$a_j = b_j - \frac{b_j - y_i}{1 - \mu_j(y_i)} \tag{9}$$

$$c_j = b_j - \frac{y_i - b_j}{1 - \mu_j(y_k)} \tag{10}$$

Donde:

j: representa el j-ésimo intervalo [Aj, Bj]

i: representa el primer índice de los datos en [Aj, Bj]

k: representa el último índice de los datos en [Aj, Bj]

yi : valor de i-ésimo dato en [Aj, Bj]

Si: similaridad entre yi y yi+1

Como puede verse, la FP resultante no es necesariamente simétrica. Este proceso de agrupamiento es muy primitivo. El algoritmo propone solamente FP de tipo triangular. En problemas reales no todos los rasgos son descritos con un mismo tipo de función, es probable que para algún rasgo la función triangular no sea la apropiada para describirlo, provocando que los valores de la FP no sean los deseados y esto pueda influir en el desempeño del sistema.

- FP triangular

Una FP triangular se especifica por tres parámetros {a, b, c}, donde a < b < c. Para estimar estos parámetros se aplica una generalización de la variante propuesta por Hong y explicada anteriormente, asumiendo la forma que él emplea para calcular la similitud entre dos valores adyacentes a partir de la diferencia entre ellos como caso particular ($\beta_i = S_i$).

Si no se considera la similitud entre datos adyacentes (β_i=1 $\forall$i), entonces el centro sería la media de los valores del intervalo. La misma idea propuesta por Hong es aplicable al resto de los parámetros. En los extremos del intervalo (Aj y Bj), la FP tiene un valor igual al mínimo de la similitud entre todos los puntos en [Aj, Bj], y luego se obtienen los parámetros aj y cj de la FP por interpolación.

- FP trapezoidal

La FP trapezoidal es especificada por cuatro parámetros {a, b, c, d}, donde a < b < c < d. Note que esta función se reduce a la función triangular cuando b es igual a c. Por esta razón, el siguiente procedimiento es aplicado para estimar los parámetros:

A partir del intervalo [Aj, Bj] calculamos a1, b1 y c1 por la heurística para FP triangulares.

De la misma forma con [Aj, b1] obtenemos a2, b2, c2.

De la misma forma con [b1, Bj] obtenemos a3, b3, c3.

Finalmente definimos la función trapezoidal con los parámetros (a2, b2, b3, c3).

- FP Gaussiana

La FP gaussiana esta definida por dos parámetros {c, σ}, donde c representa el centro de la FP y σ determina el ancho (σ > 0). Se aplican las ideas anteriores para la estimación de los parámetros de la FP triangular. El valor de cj debe ser considerado como la media de los valores de los puntos en [Aj, Bj], considerando el caso particular de la ecuación (6) donde βi es igual a uno para todos los puntos yi en [Aj, Bj]. El parámetro "ancho" esta relacionado con cj (ver expresión 10), asumiendo las propiedades de simetría de esta función.

$$\sigma = \frac{Ln(2)}{|c - v|}$$

Donde

$$v = \begin{cases} A_j & si \quad Min(|Centro - A_j|, |Centro - B_j|) = |Centro - A_j| \\ B_j & en\ otro\ caso \end{cases} \tag{11}$$

- FP Sigmoidal

La FP sigmoidal está definida por dos parámetros {c, α}. Dependiendo del signo del parámetro α (α≠0), esta función abre a la derecha (positivo) o a la izquierda (negativo) y es apropiada para representar conceptos como "muy largo" (término lingüístico Tk, 0 < j < k) o "muy negativo" (término lingüístico T_0). El parámetro c es el mismo que el de la anterior FP.

$$= \ln(0.25) * |v - c|$$

Donde

$$v = \begin{cases} A_j & si \quad Min(|Centro - A_j|, |Centro - B_j|) = |Centro - A_j| \\ B_j & en\ otro\ caso \end{cases} \quad (12)$$

1.2.3 Módulo de ajuste dinámico de parámetros

Adaptar manualmente los parámetros de una FP a un problema es complicado, pequeñas modificaciones pueden causar cambios considerables en el desempeño del modelo.

Este módulo se compone de una heurística para ajustar dinámicamente los parámetros de funciones triangulares, cuya idea central puede extenderse para abarcar también funciones trapezoidales, gaussianas y sigmoidales.

El algoritmo propuesto sigue ideas similares a los sistemas NEFCLASS y ANFIS (Nauck, 1997). También se asemeja a los algoritmos de entrenamiento utilizados en las RNA; y tiene gran similitud con el algoritmo backpropagation, porque las modificaciones se realizan en la dirección del gradiente del error. Se puede considerar que este algoritmo es supervisado y correccional.

Este algoritmo requiere una definición inicial de las funciones *f* para todos los rasgos, que significa, para los rasgos lineales modelados como variables lingüísticas, proponer FP iniciales y definir el criterio a seguir para determinar la pertenencia de un valor a un término lingüístico. Estas pueden ser sugeridas por el usuario u obtenidas de forma automática siguiendo las heurísticas explicadas anteriormente.

El algoritmo ofrece como salida los nuevos parámetros de las FP (μ) iniciales, que mejoran el desempeño del híbrido RNA-Borroso con respecto a las FP iniciales; y por tanto también variarían las funciones *f* correspondientes en la capa de preprocesamiento.

1.3 Implementación computacional del Módulo de Preprocesamiento

El proceso de preprocesamiento ya fue descrito anteriormente, así como las componentes que se utilizarán en el mismo. A continuación se describe la implementación del Módulo de Preprocesamiento de los Rasgos. En dicho módulo se hizo uso de algunas de las facilidades que nos brinda la plataforma .NET Framework.

La primera, debido a la capacidad para descubrir información de tipo en tiempo de ejecución, fue el uso de espacio de nombres *Reflectio*n para ver la información de tipo que contienen los ensamblados que, más tarde, podrá enlazar a objetos e incluso puede usar este espacio de nombres para generar código en tiempo de ejecución, es decir, que no se necesita hacer referencias a los diferentes modelos de discretización y fuzzificación que se usan en nuestro problema de forma estática, ya que mediante este mecanismo se obtiene de forma dinámica toda la información que se necesita de los ensamblados para invocar mas tarde los diferentes métodos que posee el mismo. Esta facilidad nos brinda la posibilidad de incorporar o eliminar nuevos modelos de discretizacion y fuzzificación sin tener que recompilar el código fuente, además de realizar estas tareas en tiempo de ejecución Este mecanismo fue implementado en una clase llamada *ReflectionManager*, la cual contiene los métodos necesarios para utilizar las facilidades mencionadas.

Fueron usados además ficheros XML a través de una clase que se implementó con ese fin, *XMLManager*. Esta clase es la encargada del el trabajo con ficheros XML, ya sea cargar o salvar este tipo de ficheros. Es aquí donde se implementaron los métodos para salvar los resultados de la ejecución del preprocesador, es decir, la matriz de preprocesamiento y el fichero de preprocesamiento. Este último contiene el nombre de las neuronas generadas para cada rasgo, además de todas las funciones necesarias para el manejo de la entrada de datos a partir de ficheros XML. El uso de este tipo de ficheros nos da la posibilidad de la validación de los ficheros con que trabaja nuestro módulo, ya que a través de esquemas definidos con anterioridad podemos detectar cualquier error que pudiera contener un fichero que será usado.

Como ya se explicó en el modulo de preprocesamiento, *Preprocessor*, está compuesto por los módulos de discretización, de construcción automática de FP y de ajuste de parámetros, además de poseer una estrecha interrelación con el módulo de datos (BC).

Teniendo en cuenta esta composición, el primer objetivo fue la posibilidad de obtener la información relacionada con la forma de tratamiento de los rasgos correspondientes a la base de casos con la cual vamos a trabajar, de dos formas diferentes. Estas son:

- De forma manual, generando un fichero de definición *(.defx).*
- De forma automática, a partir de un vector que se genera combinando las formas de preprocesar los rasgos.

De esta forma se diseñaron métodos que fueran capaces de ejecutar el módulo teniendo en cuenta ambas variantes, posteriormente se describirán.

Si la información sobre el tratamiento de los rasgos se obtiene de forma manual, se "parseará" el fichero de definición y se extraerá toda la información contenida en él, para construir los arreglos que se usarán con posterioridad en la construcción de la matriz. En caso de que sea de forma automática, se obtendrán los datos a partir del vector generado.

Antes de describir la implementación del Módulo de Preprocesamiento es importante definir con exactitud la estructura de los datos que tiene como entrada este módulo. Cuando los datos de entrada son especificados de forma manual, el fichero XML que se genera tiene la siguiente estructura:

La etiqueta raíz del documento <*DEFINITIONS_FILE*> contiene como atributo el nombre del fichero de definición de rasgos, el cual se distingue por su extensión *(.defx).* Seguidamente se lista la descripción de cada rasgo mediante una etiqueta <*FEATURE*>.

De cada rasgo contenido en la base de casos se guarda su nombre. A continuación y en forma de elementos XML contenidos dentro de este se procede a describir la forma en la que se ha modelado el rasgo. Las etiquetas creadas para este propósito son

<DISCRETE>, <INTERVAL> y *<FUZZZY>,* las cuales corresponden a las tres maneras de interpretar un rasgo en el contexto actual.

Para el caso de las dos variantes de modelación de rasgos numéricos, hay que diferenciar la variante manual de la automática.

- Rasgo continúo modelado como discreto, variante manual.

  La etiqueta *<INTERVALS>* está conformada por un conjunto de valores a partir de los cuales se construyen los intervalos especificados manualmente por el experto. Dichos valores están contenidos en la etiqueta *<VALUE>*.

- Rasgo continuo tratado como discreto, variante automática.

  La etiqueta *<DISCRETIZER>* es la encargada de indicar el nombre del método de discretización (discretizador) que se empleará para construir el conjunto de intervalos de forma automática. En caso de que el discretizador conlleve el uso de parámetros asociados, se indica el nombre y el valor de estos mediante un conjunto de etiquetas *<DISC_PARAM>*.

- Rasgo numérico tratado como borroso de forma manual.

  La etiqueta *<MANUAL>* indica que el atributo será tratado como borroso pero de forma manual y la misma contiene un conjunto de etiquetas *<LINGUISTIC_TERM>,* las cuales tienen como atributo el nombre de los términos lingüísticos definidos para el atributo. El nombre de los parámetros pertenecientes a los términos lingüísticos, así como los valores de los mismos, son definidos con la etiqueta <PARAM>,

  Existen 6 variantes con las cuales se puede definir un término lingüístico, ellas son:

  1. Trapezoidal (Trapeze)

  2. Triangular (Triangle)

3. Gaussiana (Gauss)

4. Beta (Beta)

5. Sigmoideal (Sigmoid)

6. Bell (Bell)

- Rasgo numérico tratado como borroso de forma automática.

  La etiqueta <*METHOD_NAME*> es la encargada de indicar el nombre del método de fuzzzificacion (fuzzzificador) que se empleará para construir el conjunto de términos lingüísticos de forma automática. Antes de fuzzificar un rasgo hay que discretizar el mismo para generar los intervalos con los cuales de definirán para cada uno de ellos el término lingüístico correspondiente, el discretizador a usar en este caso estará contenido en la etiqueta <*BASE_DISCRETIZER*>.

En el Anexo #1 se muestra un ejemplo de cada una de las variantes antes mencionadas.

Si se generan de forma automática los datos de entrada, el vector que se recibe será un arreglo que tendrá los componentes que se muestran a continuación:

$$[\boldsymbol{R1}, \boldsymbol{R2}, \boldsymbol{R3}, \ldots, \boldsymbol{Rn}]$$

Donde cada una de las R_i $\quad i = \overline{1..n-1}$ componentes representa la forma en que se va a modelar el rasgo i.

En caso de que uno o más rasgos sean tratados como discretos, el vector recibirá en la posición correspondiente al mismo el valor *"DISCRETE"*. Para los rasgos que sean tratados como lineales, las diferentes variantes con las que los mismos pueden ser tratados se muestran en la tabla # 1.

Variante	Descripción
All Triangle	Todas las funciones de membresías serán modelas por la variante de fuzzificación triangular.
All Gaussian	Todas las funciones de membresías serán modelas por la variante de fuzzificación gausiana.
Sigmoidea-Gaussian -Sigmoidea	La primera y última función de membresía serán modeladas por la variante de fuzzificación sigmoidal, el resto por la variante Gaussiana.
Triangle–Trapeze–Triangle	La primera y última función de membresía serán modeladas por la variante de fuzzificacion Triangular, el resto por la variante Trapezoidal.
Trapeze – Triangle – Trapeze	La primera y última función de membresía serán modeladas por la variante de fuzzificacion Trapezoidal, el resto por la variante Triangular.
CAIM	El rasgo será discretizado por el método CAIM.
Chi-2	El rasgo será discretizado por el método Chi-2
K-Means	El rasgo será discretizado por el método K-Means.
Equal_Width	El rasgo será discretizado por el método Equal_Width.
Equal_Frequency	El rasgo será discretizado por el método Equal_Frequency.

Tabla # 1.

La última componente del vector R representa el discretizador base a usar para obtener los intervalos a partir de los cuales se construirán los términos lingüísticos mediante las heurísticas de construcción automática de FP anteriormente citadas.

Un ejemplo de configuración para este vector modelándolo a partir de la base de casos *Iris.data*, y teniendo en cuenta que la misma tiene cinco rasgos y el último es el rasgo objetivo, puede ser el del siguiente ejemplo:

[Sigmoid-Gaussian-Sigmoid, Equal Frequency, CAIM, All Triangle, DISCRETE, Equal Frequency]

Donde se indica que el primero y cuarto rasgos serán modelados a partir de las variantes de fuzzificación Sigmoid-Gaussian-Sigmoid y All Triangle respectivamente, ambas haciendo uso del discretizador Equal Frequency que aparece en la última posición del vector. Los rasgos dos y tres serán modelados por las variantes de discretización Equal_Frequency y CAIM. Como el último rasgo de la base de casos es simbólico el vector en esa posición toma el valor DISCRETE, indicando que ese es el rasgo que se modela de forma discreta en nuestra base de casos.

Para realizar el tratamiento de los rasgos haciendo uso de las variantes anteriores se implementó la clase *Preprocessor*, la cual realiza todas las operaciones necesarias a fin de transformar la información en lenguaje natural expresada en la base de conocimientos en información en lenguaje simbólico que sea interpretable por la RNA. Esta clase genera como resultado una matriz de preprocesamiento. Esta matriz va a tener tantas columnas como neuronas preprocesadoras hayan sido generadas y filas como casos tenga la base de casos que se este usando. En dicha clase se implementaron los métodos *LoadDiscreteFeature ( )* y *LoadIndexes_Remote ( ),* los cuales son los encargados de obtener toda la información necesaria a partir de la base de casos relacionada con los rasgos discretos y lineales respectivamente; es decir, para cada rasgo definido como discreto se obtiene el conjunto de valores que este toma en la base de casos, incluyendo aquellos rasgos discretos que sean multievaluados.

Para los rasgos lineales se obtienen, aplicando alguna de las variantes de discretización o fuzzificación, el conjunto de valores representativos del rasgo (intervalos o términos lingüísticos, respectivamente).

Para ejecutar el primer método se necesita pasar como parámetro la dirección de la base de casos que se va a usar; mientras que el segundo necesita, además de dicha dirección, un arreglo que contiene la forma de tratamiento de los rasgos. Esta información se almacena en los arreglos *discrete_values* e *index_traits_lineal*, donde en cada posición se guarda una estructura para almacenar la información relevante al rasgo. Las estructuras que contienen dichos arreglos contienen diferentes campos, esto lo determina el tipo de atributo. En las figuras 1 y 2 se muestran como están formadas dichas estructuras.

```
/// <summary>
/// Estructura para guardar los indices y la informacion de los rasgos lineales.
/// </summary>
///
public struct struct_traits_lineal
{
        public string pos;
        public string traist;
        public string type_traits;
        public ArrayList information_traits;
}
```

Fig. 1 Estructura que representa al rasgo lineal

```
/// <summary>
/// Estructura para guardar los valores de los rasgos discretos.
/// </summary>

public struct struct_values
{
        public string pos;
        public ArrayList lista_valores;
}
```

Fig. 2 Estructura que representa al rasgo discreto.

Como las bases de conocimiento pueden estar almacenadas en diversos formatos (*.DATA, *.CSV, *.TXT, etc.) es preciso conocer cuál es el carácter delimitador que separa los valores de los rasgos para un caso dado. Con este fin, se implementó el método *Return_Delimiter (string cad, int count_trait),* que posibilita obtener dicho delimitador presente en la base de casos, así como la cantidad de veces que este está presente de forma sucesiva. Esta información es útil a la hora de invocar a funciones propias del lenguaje C# como el método *Split ( )* de la clase *String*, mediante el cual podemos

obtener un arreglo con los valores correspondientes a cada rasgo para un caso de la base de conocimiento.

Como se explicó anteriormente, en un caso pueden existir atributos discretos que tomen más de un valor a la vez (multievaluados), es decir, que si el valor para un rasgo determinado en el caso que estamos analizando está compuesto por más de uno de los valores pertenecientes al dominio de valores del rasgo, entonces se necesita obtener todos esos valores y almacenarlos en caso de que no hayan sido guardados con anterioridad. Para ello se implementó el método booleano *Search (string cadena),* el cual verifica si el valor del atributo que se le pasa como parámetro es multievaluado o no. En caso afirmativo, devuelve verdadero y llamamos entonces al método *VerifySep (string cadena)*, que recibe una cadena que contiene el valor multievaluado y retorna un arreglo con todos los valores presentes en la cadena ya separados.

Otro de los métodos implementados en el módulo de preprocesamiento es *Create_Begin_End_Bounds (),* el cual construye, a partir de los arreglos discrete_values e index_traits_lineal, dos nuevos arreglos llamados *beginBounds* y *endBounds*, cuyo propósito es guardar el índice de la neurona donde comienza y termina un determinado rasgo, respectivamente.

De todos los métodos implementados en el módulo preprocesador, el más importante es *CreateMatrixPreprocessor (string case_base),* ya que este se encarga de preprocesar los datos y generar la matriz de preprocesamiento. A este método se le pasa como parámetro la base de casos, pero internamente se trabaja con los arreglos discrete_values e *index_traits_lineal* generados anteriormente.

Mediante el uso de los métodos *Execute_Prep (string cbFile, string defFile, bool saveMatrix)* y *Execute_Prep (string [] Values, string cbfile)* implementados también en el módulo de preprocesamiento se garantiza la ejecución de los diferentes métodos antes comentados en su orden para dar salida a la matriz. En el primer caso, la información acerca del tratamiento dado a los rasgos está definida en un fichero de definición *(.defx),* mientras que en el segundo, a diferencia de la anterior, dicha información viene contenida en un arreglo pasado como parámetro.

Estos dos últimos métodos son los encargados de agrupar y ejecutar de forma lógica los anteriormente descritos. Lo primero que se hace es cargar de forma dinámica en varias tablas hash, toda la información necesaria de los diferentes discretizadores y fuzzificadores con los cuales se trabajará, para después ejecutar los métodos contenidos en los mismos. Ya cargada esa información se ejecuta el método *LoadIndexes_Remote* o *LoadIndexes*, los cuales generan un arreglo con toda la información relacionada a los rasgos lineales. El uso de una de estas dos funciones está determinado por la forma en que se va a ejecutar el módulo de preprocesamiento, es decir, si se ejecuta a partir de la aplicación de la generación automática, se usa el primero de ellos, en caso contarios el segundo. Seguidamente se pasa a ejecutar el método *LoadDiscreteFeature*, es aquí donde se obtiene entonces otro arreglo pero con la información relacionada con los rasgos definidos como discretos. Ya creados los dos arreglos se confeccionan los arreglos *beginBounds* y *endBounds* aplicando el método *Create_Begin_End_Bounds*. Luego de haber generado toda la información necesaria se construye la matriz de preprocesamiento ejecutando el método *CreateMatrixPreprocessor*, la cual se da como resultado final de todo el proceso y que es almacenada en ficheros generados por los métodos implementados en la clase *XMLManager*.

CAPÍTULO 2. EL MÓDULO DE EXPERIMENTACIÓN CON DISTRIBUCIÓN DE TAREAS

En el Módulo de Experimentación Combinatoria se requiere calcular el desempeño de la RNA para cada una de las combinaciones de las variantes de codificación de los datos que se explorarán, en conjunto con los posibles modelos a seleccionar, así como las variantes para el cálculo de los pesos. Este proceso se tornaría lento, por lo que resultó conveniente hacer una distribución de tareas en varios host. Esto fue posible hacerlo con un sistema distribuido utilizando un modelo cliente servidor. A continuación se describen las variantes estudiadas para la implementación de dicho sistema.

2.1 Tecnologías para la comunicación remota

Los sistemas distribuidos surgen para dar solución a las siguientes necesidades:

- Repartir el volumen de información.
- Compartir recursos, ya sea en forma de software o hardware.

La construcción de sistemas distribuidos brinda la posibilidad de utilizar los recursos de toda una red. Los dos tipos principales de sistemas distribuidos son los sistemas computacionales distribuidos y los sistemas de procesamiento paralelo. En programación distribuida, un conjunto de ordenadores conectados por una red son usados colectivamente para realizar tareas distribuidas. Por otro lado en los sistemas paralelos, la solución a un problema importante es dividida en pequeñas tareas que son repartidas y ejecutadas para conseguir un alto rendimiento.

Los sistemas distribuidos se pueden implementar usando dos modelos: el modelo cliente – servidor y el modelo basado en objetos.

El modelo de cliente-servidor contiene un conjunto de procesos clientes y un conjunto de procesos servidor. Cliente y Servidor deben hablar el mismo lenguaje para conseguir una comunicación efectiva. En el modelo Orientado a Objetos hay una serie de objetos que solicitan servicios (clientes) a los proveedores de los servicios (servidores) a través de

una interfaz de encapsulación definida. Un cliente envía un mensaje a un objeto (servidor) y éste decide qué ejecutar. RMI y CORBA son algunos de esos sistemas basados en objetos.

La cuestión básica en la programación de un sistema distribuido es como comunicarse con otras máquinas a través de la red. Se resumen a continuación diversas formas de atacar la programación de aplicaciones a través de la red.

Sockets

Los Sockets proveen al usuario de una interfaz para comunicarse a través de la red con otro sistema. Cada dirección de identificación para Sockets consiste en una dirección de Internet y en un puerto. Se suelen utilizar los conocidos protocolos TCP/IP o UDP/IP.

Una de las ventajas principales de la programación con Sockets es su sencillez.

Dce/Rpc

El DCE es un ambiente de Cálculo Distribuido diseñado para proporcionar herramientas y servicios que faciliten el desarrollo de aplicaciones distribuidas. Para su implementación, las interfaces tienen que ser especificadas en el Lenguaje de Definición de Interfaz (IDL) y compiladas por un compilador IDL.

El uso de lenguajes diferentes al C/C++, esta restringido por la dependencia en los servicios bases, como Hilos/DCE, con la consecuencia de que no podremos usar servidores multihilos si no usamos C/C ++.

Dcom

DCOM permite la distribución de componentes entre computadoras diferentes. Usa un proceso de ping para controlar las vidas de sus objetos; todos los clientes que usan un objeto servidor enviarán mensajes después de ciertos intervalos. Cuando un servidor recibe estos mensajes sabe que el cliente está todavía disponible; en otro caso destruirá el objeto remoto.

Además, los requerimientos del protocolo binario DCE/RPC plantean la necesidad de conexiones TCP directas entre el cliente y su servidor. DCOM solo está disponible para Microsoft Windows y para algunas implementaciones de UNIX.

Mts/Com+

COM+, anteriormente Servidor de Transacción de Microsoft (MTS), no sólo sirvió como una plataforma remota, sino que también proporcionó transacción, seguridad, escalabilidad, y un despliegue de servicios.

A pesar de sus ventajas, COM+ no soporta todavía el ordenamiento automático de objetos pasados por valor entre aplicaciones; sino que tenemos que pasar las estructuras de datos usando recordsets de ADO o algún otro medio de serialización. Otra desventaja que impide el amplio uso de COM+ es su difícil configuración y desarrollo, lo que complica su uso para aplicaciones reales.

Java Rmi (Remote Method Invocation)

RMI fue el primer framework con la idea de crear sistemas distribuidos que apareció para Java. Además, viene integrado en cualquier máquina virtual Java posterior a la versión 1.0 y está pensado para hacer fácil la creación de sistemas distribuidos a partir de una aplicación cuyas clases ya están implementadas.

La invocación de métodos remotos permite que un objeto que se ejecuta en una máquina pueda invocar métodos de un objeto que se encuentra en ejecución bajo el control de otra máquina (por supuesto no hay problemas para las relaciones entre los objetos cuando ambos son locales). En definitiva, RMI permite crear e instanciar objetos en máquinas locales y al mismo tiempo crearlos en otras máquinas (máquinas remotas), de forma que la comunicación se produce como si todo estuviese local. RMI se convierte así en una alternativa muy viable a los Sockets de bajo nivel con una serie de particularidades destacables:

- Permite abstraer las interfaces de comunicación a llamadas locales, no necesitamos fijarnos en el protocolo y las aplicaciones distribuidas son de fácil desarrollo.

- Permite trabajar olvidando el protocolo.

- Es flexible y extensible.

- A diferencia de DCE/RPC y DCOM, las interfaces no son escritas en IDL abstracto, sino en Java. Esto es posible debido a que Java es el único lenguaje que implementa RMI.

CORBA

CORBA tiene como objetivo ser el software intermedio para comunicar sistemas heterogéneos. Es una colección de estándares; la implementación de agentes de solicitud de objetos (ORBs) es hecha por terceras partes. Dado que algunas partes del estándar son opcionales y a los programadores de agentes ORB se les permiten incluir rasgos adicionales que no están en las especificaciones, es fácil encontrase con agentes incompatibles. Además las peticiones al servidor pueden ser extremadamente lentas.

No obstante después de implementar la primera aplicación CORBA, se podrán integrar muchos lenguajes de programación y plataformas. Existen capas hasta para la integración de COM o EJB.

.Net Remoting

Ante todo se debe comenzar expresando que esta tecnología forma parte de la más reciente plataforma de desarrollo: Microsoft .Net; lanzada por el gigante informático oficialmente en el año 2001.

Microsoft .NET Remoting provee un marco que permite que una aplicación pueda comunicarse con otra, tanto si ambas residen en el mismo equipo como si residen en equipos distintos de la misma red de área local o en redes distintas separadas por una gran distancia, incluso si los sistemas operativos que se ejecutan en dichos equipos son

diferentes; con la peculiaridad de necesitar ejecutarse sobre Microsoft .Net Framework, del cuál existen versiones para Linux y sus homólogos así como para todas las versiones de Windows. Microsoft .NET Framework es un artefacto importante dentro de la arquitectura .Net. Como las aplicaciones desarrolladas sobre esta arquitectura tienen que correr sobre Microsoft .Net Framework, en sus inicios muchos vieron su instalación como algo incómodo. Esta molestia está cambiando rápidamente ya que los propietarios de los sistemas operativos están incorporando la correspondiente versión Microsoft .Net Framework en sus productos siendo totalmente transparente para el usuario.

La interfaz de programación de .NET Remoting en la plataforma .NET, tiene otra filosofía respecto a las comunicaciones y formatos de mensajes de otras APIs, como son COM distribuido (DCOM) o la invocación remota de métodos (RMI). En lugar de basarse en un protocolo y en mensajes propietarios, .NET Remoting utiliza estándares muy bien establecidos como SOAP para mensajería y HTTP y TCP como protocolo de comunicación. Incluso se pueden utilizar canales propios o de terceras partes.

Microsoft .NET Remoting oculta todos los detalles de implementación del trabajo con Sockets, entregando al programador final una interfaz de programación que se distingue por las facilidades de uso y su potente alcance. En el siguiente esquema se ejemplifican claramente las diferentes partes que conforman una típica llamada a un objeto remoto en .NET.

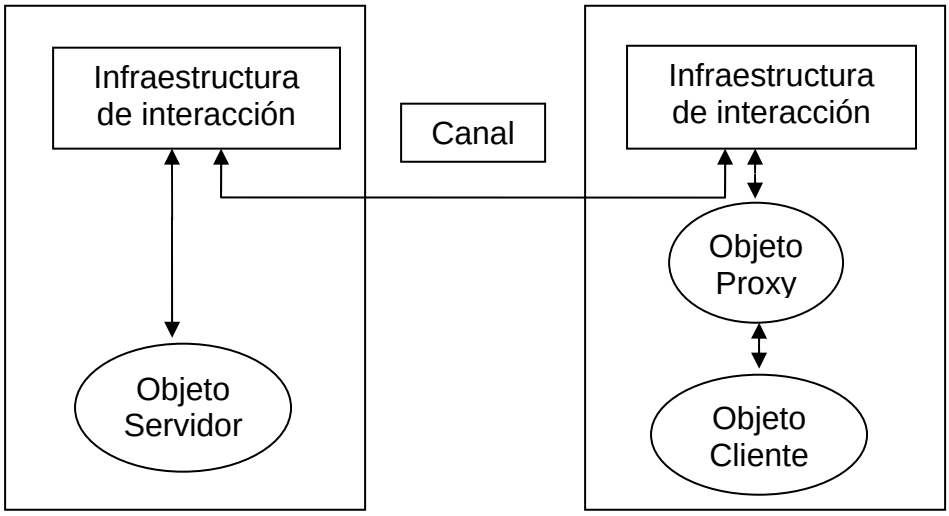

Figura # 3. Visión general de una comunicación a través de .Net Remoting

2.2 Diseño e implementación de la distribución de tareas

De las variantes analizadas anteriormente, fue seleccionada .Net Remoting atendiendo a sus características y las ventajas que reporta al módulo.

El entorno remoto .NET permite a las aplicaciones comunicarse entre objetos que están en servidores diferentes, procesos diferentes o dominios de aplicación IBI diferentes. El entorno remoto permite implementar un servidor o una aplicación de servidor y una aplicación cliente. En el servidor o el cliente, la aplicación puede ser cualquiera de las plantillas de aplicación .NET disponibles, incluyendo aplicaciones de consola, aplicaciones de servicios Windows, aplicaciones ASP.NET, aplicaciones WindowsForms y aplicaciones IIS. En el servidor, se configura mediante programación (o se emplea un archivo de configuración para especificarlo) el tipo de activación que permitirán los clientes. Se pueden usar uno de los varios tipos de métodos de activación, incluyendo Singleton y SingleCall (Ferguson, 2002).

El modelo mas eficiente en el consumo de recurso es la activación en el servidor de un Singleton. Estas son sus ventajas:

- La activación en el servidor consume menos recursos en el que la activación en el cliente. Esta última requiere un objeto distinto para cada cliente, y cada objeto debe mantener su estado de una llamada a la siguiente.

- El uso de un Singleton en comparación con el modelo SingleCall, evita tener que crear un objeto por cada llamada. Normalmente, la creación de estos objetos remotos no es demasiado costosa, pero al final, el tiempo acumulado puede ser significativo.

- Si varios clientes ejecutan simultáneamente métodos de Singleton, el servidor creará un hilo por cada cliente para que el mismo objeto sea manipulado a la vez por estos hilos.

Se deberá especificar el canal y el puerto a través de los cuales se comunica el objeto y el formato que tendrán los datos cuando pasen entre el servidor y el cliente. El formato de

serialización de los datos es muy importante pues según el diseño del sistema, se pueden usar datos binarios, SOAP o un formato personalizado, para recodificar los datos (Rammer, 2002). El formato binario es más eficiente, SOAP por su parte es más conocido, ofrece más posibilidades de compatibilidad con otros sistemas, pero a costa de más redundancia.

Un canal es el conducto por el que un cliente y un servidor remoto intercambian mensajes. .Net Remoting tiene dos tipos de canales predefinidos: uno que utiliza protocolo HTTP y otro que trabaja a más bajo nivel, directamente a través de zócalos de TCP/IP. En el espacio de nombres System.Runtime.Remoting.Channels hay los dos formateadores predeterminados: el canal TCP y el canal HTTP. En ambos casos debemos especificar el puerto por el que se comunicarán los dos extremos. HTTP se asocia normalmente con el puerto 80 pero .NET Remoting puede usar este protocolo por cualquier otro puerto que indiquemos.

La codificación predeterminada para TCP es la codificación binaria, lo que hace que sea un modo eficaz de pasar datos entre objetos remotos. Los datos binarios siempre ocupan menos espacio que los datos XML equivalentes pasados mediante SOAP en un canal HTTP.

Uniendo la mayor eficiencia de la serialización binaria al hecho de que el canal TCP es mas rápido que uno HTTP, queda claro cual elegir en caso de que la compatibilidad no sea un problema.

En el cliente, se crea una instancia del objeto que publica el servidor, en el canal y puerto especificados y se esperan peticiones del servidor. Esto se puede lograr mediante programación o mediante un archivo de configuración. En este punto, las llamadas de método no son diferentes de cualquier otro objeto de una aplicación .NET que se pueda usar. Tras crear objetos, se llaman a los métodos, se establece y recuperan propiedades y se desencadenan eventos igual que con un objeto que no esté usando el entorno remoto.

Lo anterior se puede resumir en las siguientes tareas:

- Especificar los canales y puertos que recodifican los objetos entre el servidor y el cliente.
- Usar formateadotes para especificar el formato en que los datos son serializados y deserializados entre el servidor y el cliente.
- Determinar como se activan los objetos del servidor y cuanto dura la activación.

Para empezar con la aplicación remota, se necesita crear un ensamblado que contenga las llamadas de método reales que usará la aplicación servidora. Una vez creado el ensamblado, se creará la aplicación servidora que acepte peticiones de los métodos del ensamblado por parte de los clientes.

2.2.1 Diseño de la distribución de tareas

En el módulo experimentador, primeramente fue necesario definir una aplicación servidora y otra cliente. La aplicación servidora fue diseñada con el objetivo de distribuirla por los host y la cliente para controlar las peticiones a cada uno de los servidores.

La aplicación cliente cuenta con una clase controladora *Client* la cual se encarga de desencadenar el proceso de experimentación combinatoria. Cuenta además con la clase *HillClimbing* que se encarga de ejecutar el método de búsqueda que será descrito en el siguiente capítulo. Se usa la dll *CasesBaseManager* para el manejo de las BC y la clase *System.Threading* para el trabajo con hilos de la aplicación.

La aplicación servidora cuenta con una clase *Server* en la que se publica el objeto servidor en el método *PublicServerObject*, este objeto será instanciado en la aplicación cliente. Cuenta además con la clase *RemoteTask* en la que implementa el método *Execut* el cual es el encargado de calcular el rendimiento de la RNA a partir de los datos de entrada que le son enviados desde la aplicación cliente.

El cliente tendrá el control de todos los host en los que se encuentran las aplicaciones servidoras y estas a su vez se encontrarán escuchando las peticiones de la aplicación cliente.

2.2.2 Implementación computacional de la distribución de tareas

Para la implementación de la distribución de tareas fue necesario definir un conjunto de aspectos importantes que determinan la forma de comunicación cliente servidor. Se especificó el canal y los puertos que recodifican los objetos, el formato en que los datos serán serializados y deserializados y el modo de activación del objeto.

La comunicación cliente servidor se establecerá a través del canal TCP, ya que éste es el más eficiente para la transferencia de datos debido a su codificación binaria, además de ser más rápido. El HTTP pudiera ser más conveniente en caso de tener que atravesar cortafuegos pero no es necesario para la aplicación que se desea desarrollar. La codificación de los datos para este canal es binaria como ya se dijo. Los puertos serán determinados dependiendo de los que estén disponibles en el momento de la ejecución En el caso del modo de activación del objeto se usa el modo Singleton pues con él se crea como máximo una instancia de la clase remota, independientemente del número de clientes que accedan a ella.

Fue definida además la interfaz *IServerObject* y ésta fue alojada en un ensamblado compartido para el cliente y el servidor. En la misma se encuentra la definición del método *Execute(string[] combin, string [,] CasesBase)* que se implementa en la aplicación servidora y a su vez es invocado desde la aplicación cliente.

Definidos estos aspectos se describirán los métodos de las clases de las aplicaciones cliente y servidor.

La aplicación servidora cuenta con una clase *Server* en la que se publica el objeto servidor que será instanciado en la aplicación cliente mediante el método *PublicServerObject*, el cual es llamado desde que comienza a ejecutarse la aplicación. A continuación se muestra dicho método.

```
public bool PublicServerObject(int port)
{
        try
        {
                TcpServerChannel ChannelServer = new TcpServerChannel(1234);
                RemotingConfiguration.RegisterWellKnownServiceType(typeof(RemoteTask),
                                                                   "TestClient",
                                                                   WellKnownObjectMode.Singleton);
                return true;
        }
        catch
        {
                return false;
        }
}
```

Fig. # 4 Método que publica el objeto servidor.

Además de la clase anteriormente descrita, la aplicación servidor cuenta con una clase *RemoteTask* que implementa el método *Execute ( ),* el cual es el encargado de calcular el rendimiento de la RNA a partir de los datos de entrada que le son enviados desde la aplicación cliente.

Este método usa dos parámetros: el primero es un arreglo que contiene la información relacionada a la forma de tratamiento de los rasgos, el modelo de red a utilizar, la forma del cálculo de los pesos y la estrategia de asignación; mientras que el segundo parámetro es una matriz con los valores de la base de casos a procesar.

Esta matriz es previamente salvada para el trabajo con ella usando un método implementado en la clase *XMLManager* (uno de los módulos que usa esta aplicación y cuya responsabilidad es la manipulación de todos los ficheros en formato XML) llamada *Save_Matrix ( )*, pasándole dicha matriz así como también el camino donde se va a salvar la misma. Para obtener la dirección de donde se va a salvar usamos dos de las funciones propias del lenguaje, *Path.GetDirectoryName ( )* y *Directory.GetCurrentDirectory ( )* mediante las cuales podemos obtener la ruta del ejecutable de la aplicación.

Después de haber salvado la matriz en el camino indicado, se obtiene la forma de tratamiento que se le dará a los rasgos a partir del arreglo de entrada y con ambas cosas se hace una llamada al método *Execute_Prep* ( ), el cual esta implementado en el módulo *Preprocessor* y que es el encargado de preprocesar los datos de la base de casos a partir

de la forma de tratamiento. Como resultado del preprocesamiento se generan el fichero de grupos o *pools* (*.POOLS) y el fichero que contiene la matriz resultante (*.PREP).

El fichero *.pools* contiene los grupos de neuronas que, como resultado de la forma de tratar los rasgos, fueron generadas para conformar la topología de la RNA. El fichero *.prep* almacena la matriz de preprocesamiento cuyos valores están entre 0 y 1 y que sirve como entrada al modelo de red. En el Anexo #2 se muestra un ejemplo del mismo.

Ya preprocesada toda la información, se procede a cargar dinámicamente, haciendo uso de la clase *ReflectionManager*, los modelos de red con que cuenta la aplicación para su funcionamiento y se extrae del arreglo pasado como parámetro a la función el modelo a usar, así como también los índices de los rasgos objetivos. Con toda esta información se ejecuta el modelo de red seleccionado y se devuelve al cliente el valor obtenido como rendimiento (*performance*) de la red. La figura 5 muestra el diagrama de clase de la la aplicación servidora.

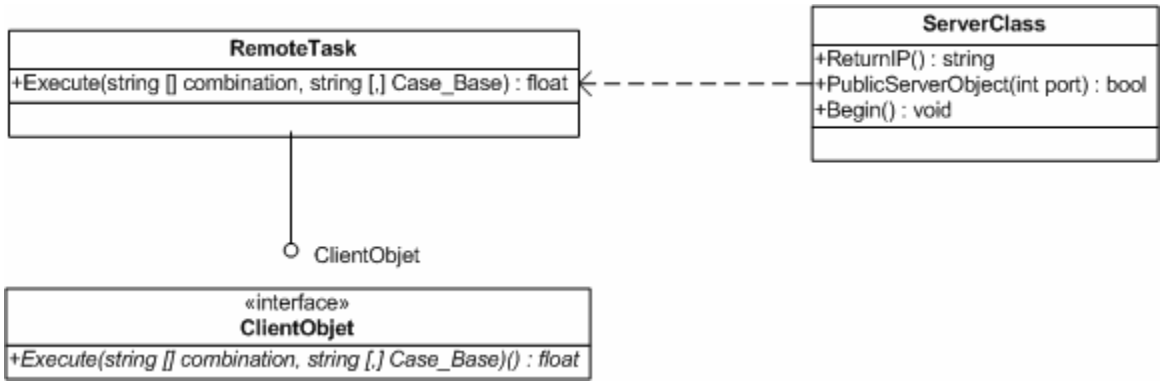

Fig # 5 diagrama de clases de la aplicación servidora

La clase *Client* de la aplicación cliente contiene los métodos *LoadCasesBase, GetData* e *InitProcess.* El primero de ellos se encarga de cargar en una matriz el contenido de la BC con la que se quiere experimentar. Fue necesaria su implementación para poder pasar esta información al cliente en forma de objeto serializado. El método *GetData* carga en memoria los datos que el usuario ha seleccionado para combinar.

InitProcess recibe como parámetro un *ArrayList* de ocho elementos al que se le llamó *Config.* El mismo contiene los siguientes elementos en cada una de sus posiciones:

Config[0]= <arreglo con los modelos de RNA seleccionados por el usuario>

Config[1]= < arreglo con los métodos de cálculo de los pesos seleccionados por el usuario >

Config[2]= < arreglo con las estrategias de asignación seleccionadas por el usuario >

Config[3]=<arreglo con los fuzzificadores seleccionados por el usuario>

Config[4]=< arreglo con los discretizadores seleccionados por el usuario >

Config[5]= < cadena (*string*) con el discretizzador base >

Config[6]=< arreglo el listado de host seleccionado por el usuario >

Config[7]=<cadena (*string*) con la dirección de la BC seleccionada>

Por otra parte en *InitProccess* se invocan lo métodos *LoadCasesBase* y *GetData,* posteriormente se ejecuta un hilo por cada modelo de RNA seleccionado y en cada uno se invoca al método *Run* de la clase *HillClimbing*.

HillClimbing es otra de las clases contenidas en la aplicación cliente. En ella se cuenta con los métodos *Run, SearchChild y GetPerformance*. La descripción de dichos métodos se hará en el siguiente capítulo. Es necesario especificar que el método *GetPerformance* es el encargado de invocar al método *Execute* de la aplicación servidora, en el que se efectúa el cálculo del desempeño de la RNA. La siguiente figura muestra el diagrama de clase de la aplicación cliente.

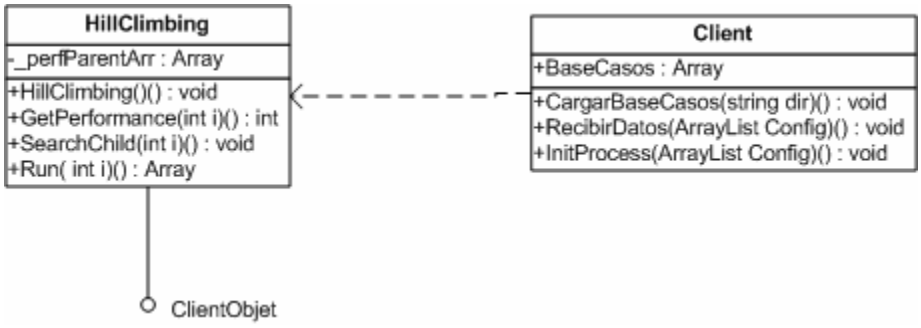

Fig. 6 diagrama de clases de la aplicación cliente

Las integración de las aplicaciones cliente y servidor se muestra en el siguiente diagrama de clases.

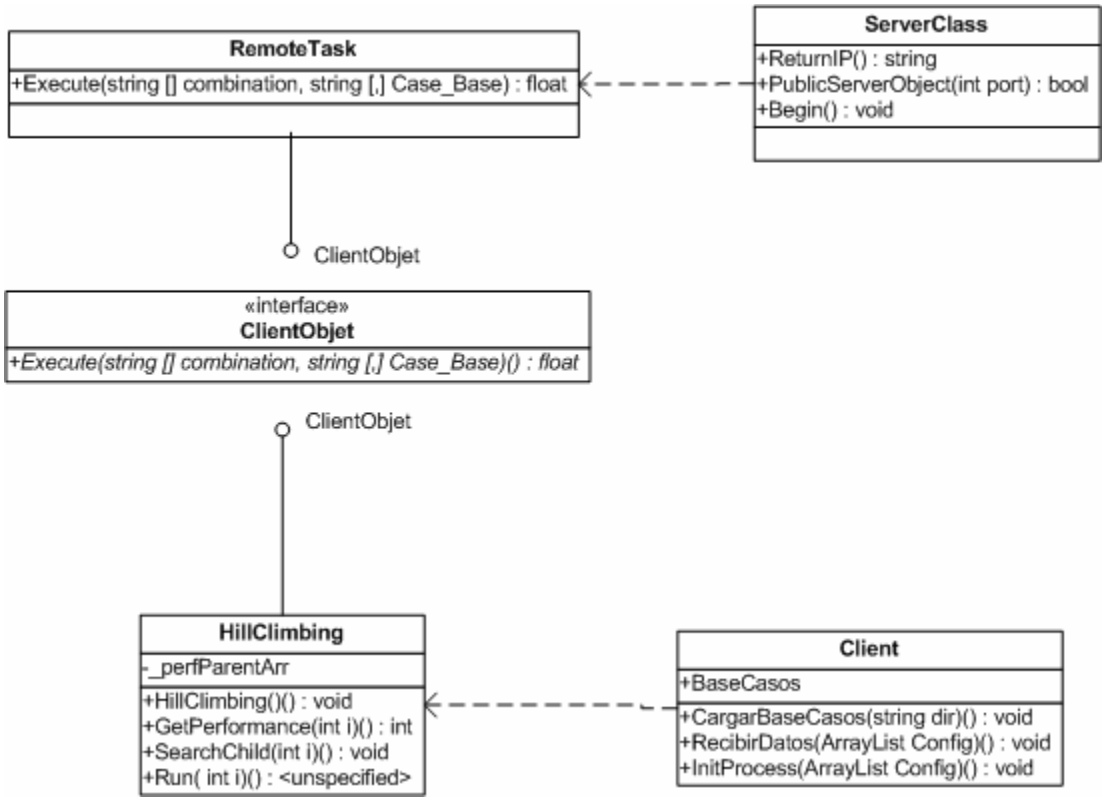

Fig. 7 Diagrama de clases Cliente-Servidor

CAPÍTULO 3. EL MÓDULO DE EXPERIMENTACIÓN COMO UN PROBLEMA DE CONFIGURACIÓN

Como ya se explicó la herramienta *NeuroEvaluator* tiene implementados cuatro modelos de red. Los datos que reciben dichos modelos son previamente transformados atendiendo a una estrategia de asignación y un preprocesamiento de los rasgos considerados. Para el preprocesamiento de los rasgos numéricos se puede seguir un enfoque duro utilizando discretizadores, o un enfoque borroso si estos rasgos se modelan como variables lingüísticas. Cada uno de estos criterios es un grado de libertad a especificar previo a que la red neuronal pueda ser utilizada para resolver un problema. Luego, para un conjunto de datos, el problema consistiría en encontrar la configuración óptima. Una configuración se puede representar por un vector de cinco componentes atendiendo a los aspectos antes mencionados como se muestra a continuación.

(Modelo de red,	Trat. Pesos,	Est. Asignación,	Prep. Rasgos)
• **IAC** • **SIAC** • **Hopfield** • **BSB**	• **Frec. Abs** • **Frec. Rel** • **Coef. de Correlación de Pearson**	• **Todas las membresías** • **Máx. Membresía** • **Membresía>δ**	• **5 discretizar** • **5 fuzzificar**

Con el módulo de experimentación combinatoria se pretende hallar una configuración de los grados de libertad de los factores mencionados, que permita un buen desempeño de la RNA ante un problema concreto.

A pesar de implementar una variante distribuida para el cálculo del rendimiento, es apropiado pensar en otra forma para reducir el número de estados de configuración en los que se calcule dicho rendimiento, ya que este es un problema de explosión combinatoria.

Para esto el usuario puede seleccionar las opciones con las que desea experimentar y además es conveniente utilizar una estrategia de búsqueda con la que se exploren sólo las configuraciones que prometan un mejor desempeño de la RNA. A continuación se analizarán algunas variantes de solución para el problema planteado.

3.1 Estrategias de búsqueda

En computación, dos objetivos fundamentales para la mayoría de casos son encontrar algoritmos con buenos tiempos de ejecución y buenas soluciones, usualmente las óptimas. Una heurística es un algoritmo que ofrece uno o ambos objetivos; por ejemplo, normalmente encuentran buenas soluciones, aunque en ocasiones no hay pruebas de que la solución no pueda ser arbitrariamente errónea; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que deba ser así.

A menudo, pueden encontrarse instancias concretas del problema donde la heurística producirá resultados muy malos o se ejecutará muy lentamente. Aún así, estas instancias concretas pueden ser ignoradas porque no deberían ocurrir nunca en la práctica por ser de origen teórico, y el uso de heurísticas es muy común en el mundo real.

Es útil a este respecto recordar la distinción entre tres tipos de estrategias para la resolución de problemas: búsquedas ciegas, búsquedas heurísticas (basadas en la experiencia) y búsquedas racionales (usando un sistema formal de razonamiento).

De acuerdo con ANSI/IEEE Std 100-1984, la heurística trata de aquellos métodos o algoritmos exploratorios para la resolución de problemas en los que las soluciones se descubren por la evaluación del progreso logrado en la búsqueda de un resultado final. Se trata de métodos en los que, aunque la exploración se realiza de manera algorítmica, el progreso se logra por la evaluación puramente empírica del resultado. Se gana eficacia, sobre todo en términos de eficiencia computacional, a costa de la precisión. Las técnicas heurísticas son usadas por ejemplo en problemas en los que la complejidad de la solución algorítmica disponible es función exponencial de algún parámetro; cuando el valor de éste crece, el problema se vuelve rápidamente inabordable. Una alternativa heurística será

practicable si la complejidad de cómputo depende, por ejemplo, polinómicamente del mismo parámetro.

Las técnicas heurísticas no aseguran soluciones óptimas sino solamente soluciones válidas, aproximadas; y frecuentemente no es posible justificar en términos estrictamente lógicos la validez del resultado.

La heurística no garantiza que siempre se tome la dirección de la búsqueda correcta, por eso este enfoque no es óptimo sino suficientemente bueno. Frecuentemente son mejores los métodos heurísticos que los métodos de búsquedas a ciegas. Las desventajas y limitaciones principales de la heurística son:

- La flexibilidad inherente de los métodos heurísticos pueden conducir a errores o a manipulaciones fraudulentas.

- Ciertas heurísticas se pueden contradecir al aplicarse al mismo problema, lo cual genera confusión y hacen perder credibilidad a los métodos heurísticos.

- Soluciones óptimas no son identificadas. Las mejoras locales determinadas por las heurísticas pueden cortar el camino a soluciones mejores por la falta de una perspectiva global. La brecha entre la solución óptima y una generada por heurística puede ser grande.

La popularización del concepto se debe al matemático George Polya, con su libro “Cómo resolverlo” (How to solve it). Habiendo estudiado tantas pruebas matemáticas desde su juventud, quería saber como los matemáticos llegan a ellas. El libro contiene la clase de recetas heurísticas que trataba de enseñar a sus alumnos de matemáticas.

Actualmente, la heurística es más frecuentemente usada como un adjetivo para referirse a cualquier técnica que mejore la media del proceso de solución de problemas.

Para problemas de búsqueda del camino más corto el término tiene un significado más específico. En este caso una heurística es una función matemática, h(n) definida en los

nodos de un árbol de búsqueda, que sirve como una estimación del coste del camino más económico de un nodo dado hasta el nodo objetivo.

Una función de evaluación heurística es una función que hace corresponder situaciones del problema con números. Es decir, da una medida conceptual de la distancia entre un estado dado y el estado objetivo. Estos valores son usados para determinar cual operación ejecutar a continuación, típicamente seleccionando la operación que conduce a la situación con máxima o mínima evaluación (Russell, 1995).

Se verán a continuación ejemplos de búsquedas heurísticas que serán analizados para la solución del problema de configuración que se desea solucionar.

3.1.1 Algunos métodos de búsqueda heurística.

Algoritmo Best-First

La búsqueda por el mejor nodo es una forma de combinar las ventajas de las búsquedas en profundidad y a lo ancho en un único método. En cada paso del proceso de búsqueda se selecciona el más prometedor de aquellos nodos que se han generado hasta el momento. Entonces este se expande usando los operadores para generar sus sucesores. Si uno de ellos es una solución, se termina. Si no, todos esos nuevos nodos se añaden al conjunto de nodos generados hasta ese momento. Se selecciona de nuevo el nodo más prometedor y el proceso continúa. La selección del nodo a expandir es independiente de la posición en que se encuentra en el árbol de búsqueda y de la posición del nodo más prometedor. Lo que sucede usualmente es que se realiza un poco de búsqueda a profundidad mientras se explora una rama prometedora. En un momento dado esa rama comienza a ser menos prometedora que otras de más alto nivel que se han ignorado hasta ese momento.

Búsqueda A*

La variante anterior minimiza el costo estimado al objetivo, h(n) y reduce el costo de la búsqueda considerablemente, pero no es ni optima ni completa. Por otro lado, una búsqueda con costo uniforme minimiza el costo del camino hasta el nodo n desde el nodo

inicial S, g(n). Ella es óptima y completa pero puede ser muy ineficiente. Una buena alternativa es combinar ambas estrategias para tomar sus ventajas. Eso puede hacerse simplemente combinando las dos funciones de evaluación mediante la suma, así se obtiene: $f(n) = g(n) + h(n)$.

Como g(n) da el costo del camino del nodo inicial al nodo n, y h(n) es el estimado del costo del camino más barato desde n hasta el nodo objetivo, entonces f(n) es el costo estimado de la solución más barata a través de n. Se puede probar que esta estrategia es completa y óptima siempre que la función h nunca sobrestime el costo de alcanzar el objetivo. En este caso h se denomina una heurística admisible. Las heurísticas admisibles son por naturaleza optimistas porque ellas piensan que el costo de resolver el problema es menor que lo que es realmente. Este optimismo se transfiere a la función f; si h es admisible f(n) nunca sobrestimará el costo real de la mejor solución a través de n.

La búsqueda primero el mejor que usa f como función de evaluación y una función h admisible se conoce como búsqueda A*. La distancia aérea (equivalente a la distancia en línea recta entre dos puntos del mapa) es un ejemplo de heurística admisible.

Ascensión de colinas (Hill-Climbing)

Existe un conjunto de problemas para los cuales el camino que conduce a la solución es irrelevante. Usualmente esto ocurre en aquellos problemas en los que la descripción del estado contiene toda la información necesaria para una solución. La idea general es comenzar con una configuración completa y hacer modificaciones para mejorar su calidad. El método de búsqueda conocido por ascensión de colinas es un bucle que se mueve en dirección del valor creciente y termina cuando alcanza un pico en el que ningún vecino tiene valor mejor.

Con este método la estrategia es repetidamente expandir un nodo, inspeccionar sus sucesores recién generados, y seleccionar y expandir el mejor entre los sucesores sin mantener referencias a los padres.

- Escalada simple: Se busca cualquier operación que suponga una mejora respecto al padre.

- Escalada estocástica: Se escoge aleatoriamente entre los movimientos ascendentes.

- Escalada por máxima pendiente (steepestascent hill climbing, gradient search): Se selecciona el mejor movimiento (no el primero de ellos) que suponga mejora respecto al estado actual.

Este método es una variación de la búsqueda primero en profundidad en la cual se usa la función heurística para estimular la distancia entre el nodo actual y el nodo objetivo. La siguiente descripción del método muestra la semejanza con la búsqueda en profundidad:

- P1. Formar una pila de un elemento consistente del nodo raíz.

- P2. Iterar de (a) a (c) hasta que la pila este vacía.

    a) Chequear si el elemento del tope de la pila es una solución.

    b) SALIR con ÉXITO si el elemento chequeado es solución.

    c) Si el primer elemento no es solución ordenar los descendientes del mismo según la distancia restante al objetivo y luego añadir el hijo al tope de pila.

- P3. Salir con Falla.

Como se aprecia la diferencia entre la descripción de ambos métodos radica en el paso 2c.

Otra descripción es la siguiente:

- P1. Evaluar el estado inicial. Si es objetivo SALIR con ÉXITO.

- P2. Iterar hasta encontrar una solución o hasta que no haya nuevos operadores aplicables al estado actual.

a) Seleccionar un operador todavía no aplicado al estado actual y aplicarlo para producir un nuevo estado.

b) Evaluar el nuevo estado.

c) Si es objetivo SALIR con ÉXITO.

d) Si es mejor que el estado actual hacerlo estado actual.

e) Si no es mejor que el actual continuar en el lazo.

El algoritmo ascensión de colinas es típicamente local, ya que la decisión para continuar la búsqueda se toma mirando únicamente a las consecuencias inmediatas de sus opciones. Puede que nunca lleguen a encontrar una solución, si son atrapados en estados que no son el objetivo y desde donde no se puede hallar mejores estados, por ejemplo:

- Máximo local: Ningún vecino tiene mejor coste.
- Meseta: Todos los vecinos son iguales.
- Cresta: La pendiente de la función sube y baja (efecto escalón).

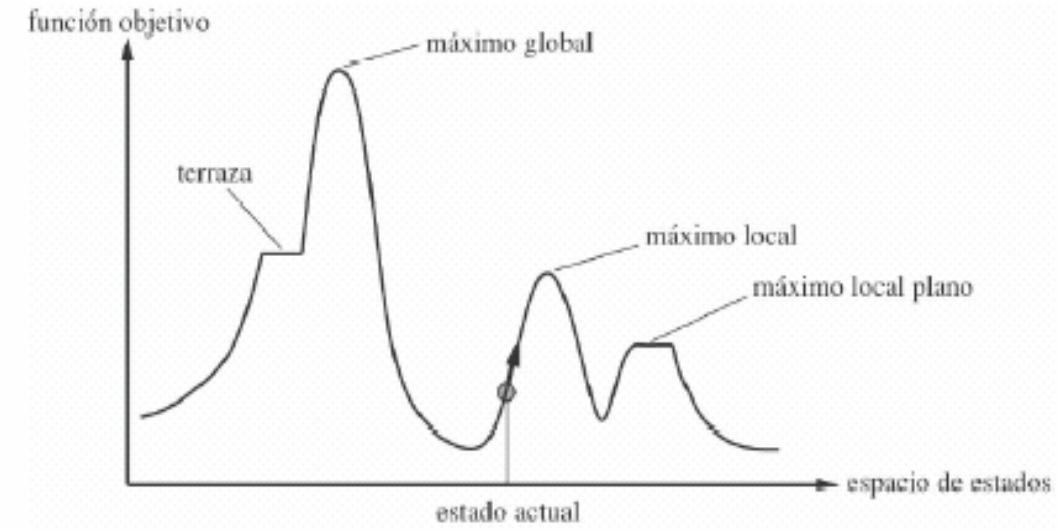

Fig. # 8 Ascensión de colinas.

Hay algunas formas que pueden ayudar a solventar estos problemas, aunque no existe garantía:

- Hacer backtracking a un nodo anterior y seguir el proceso en otra dirección.
- Reiniciar la búsqueda en otro punto buscando mejorar la solución actual.
- Aplicar dos o más operaciones antes de decidir el camino.
- Hacer HC en paralelo (p.ej. Dividir el espacio de búsqueda en regiones y explorar las más prometedoras).

Algunas observaciones sobre el método son:

- Cuando se llega a un nodo muerto no hay forma de hacer retroceso (salvo generar otro nodo raíz).
- Es usado cuando existe una buena función heurística.
- La estrategia es llamada irrevocable porque el proceso no nos permite virar la atención hacia alternativas previamente suspendidas.

La estrategia es también útil en problemas donde los operadores poseen un cierto tipo de independencias llamada conmutatividad, una condición en la cual la aplicación de un operador no afecta la aplicabilidad de otros.

3.2 Diseño e implementación del Hill Climbing

Para encontrar una configuración óptima de los grados de libertad descritos en este capítulo, la primera idea pudiera ser, generar todas las configuraciones posibles del vector, descrito también al inicio del capítulo, comprobar el desempeño de la red con cada una de ellas y seleccionar la que proporcione un mejor resultado. Esta seria una búsqueda exhaustiva, ineficiente para el problema que se pretende resolver. En este caso seria conveniente utilizar una heurística sencilla que permita obtener una buena configuración sin tener que generarlas todas. De las variantes analizadas fue seleccionado el método Hill Climbing, del que se hizo una detallada descripción en el epígrafe anterior.

Este método de búsqueda cumple con los requerimientos del problema a resolver pues cada nodo (instancia de un vector conformado por los grados de libertad) está unívocamente determinado, no se necesita saber el camino recorrido para llegar al nodo actual. Esta característica permite generar una configuración del vector sin tener que almacenar las que ya fueron generadas.

Una de las variantes del Hill Climbing es la de retroceder al nodo que ofreció un mejor valor al ser evaluado mediante la función heurística, lo cual permite continuar el proceso de búsqueda en otra dirección. Esta variante será aplicada al problema que se pretende resolver. A continuación se hará una descripción del diseño y la implementación de la heurística para el problema en particular.

3.2.1 Diseño del Hill Climbing

Como ya se explicó, el vector de configuración esta formado por cinco componentes que pueden tomar diferentes valores. La primera componente del vector son los modelos de RNA; se formará un vector por cada modelo fijando en la primera posición el modelo de RNA correspondiente y manteniendo el resto de las posiciones con valores iniciales iguales para tolos los vectores que se formen. Estos valores iniciales fueron seleccionados previamente atendiendo a la incidencia que ellos tienen sobre el rendimiento de la RNA. Cada uno de los vectores formados será un estado inicial a partir del cual se comenzarán a generar los siguientes nodos según la heurística planteada.

Un estado en este caso será una instancia del vector y el criterio objetivo, maximizar el desempeño de la RNA, el cálculo del desempeño de la RNA en el estado actual. Esto determina el próximo nodo a generar. El estado final estará determinado por la obtención de un desempeño muy alto, superior a 95% o cuando todas las componentes del vector sean exploradas en todo su dominio. El orden que se determinó para asignar los valores a cada grado de libertad fue el siguiente:

Forma de tratamiento de los pesos

1. Frecuencia Absoluta

2. Frecuencia Relativa

3. Coeficiente de Correlación de Pearson

Estrategia de asignación

1. Todas las membresías

2. Máxima Membresía

3. Membresía>δ

Preprocesamiento de los rasgos

1. Rasgo a fuzzificar mediante la variante Triangle.

2. Rasgo a fuzzificar mediante la variante All Guassian.

3. Rasgo a fuzzificar mediante la variante Sigmoidea, Gaussian, Sigmoidea.

4. Rasgo a fuzzificar mediante la variante * Triangle.

5. Rasgo a fuzzificar mediante la variante Triangle, Trapezoid, Triangle.

6. Rasgo a discretizar mediante la variante CAIM.

7. Rasgo a discretizar mediante la variante Chi_Square.

8. Rasgo a discretizar mediante la variante K-Means.

9. Rasgo a discretizar mediante la variante Equal_Width.

10. Rasgo a discretizar mediante la variante Equal_Frequency.

Se seleccionó además un orden para variar las componentes del vector, basándonos en las afectaciones que provoca la variación de cada uno de ellos en el rendimiento de la RNA. Por lo tanto se variará primero la forma de preprocesar cada rasgo, luego la forma de tratar los pesos y por último la estrategia de asignación.

Inicialmente se calcula el rendimiento de la RNA con la configuración inicial del vector, luego se varía la componente del vector que corresponde y si el rendimiento de la RNA es mejor con la nueva configuración, se comenzarían a generar nuevos nodos fijando la componente con el valor actual y variando la que le sigue en el orden predeterminado. En caso que se obtenga un menor desempeño se explorará el próximo valor del dominio de la componente que se está variando. Este proceso se observa en la figura # 9. Asumamos que cada rasgo (R1 y R2) pueden tomar solo dos valores, es decir, solo se desea probar con una variante de discretización y una de fuzzificación.

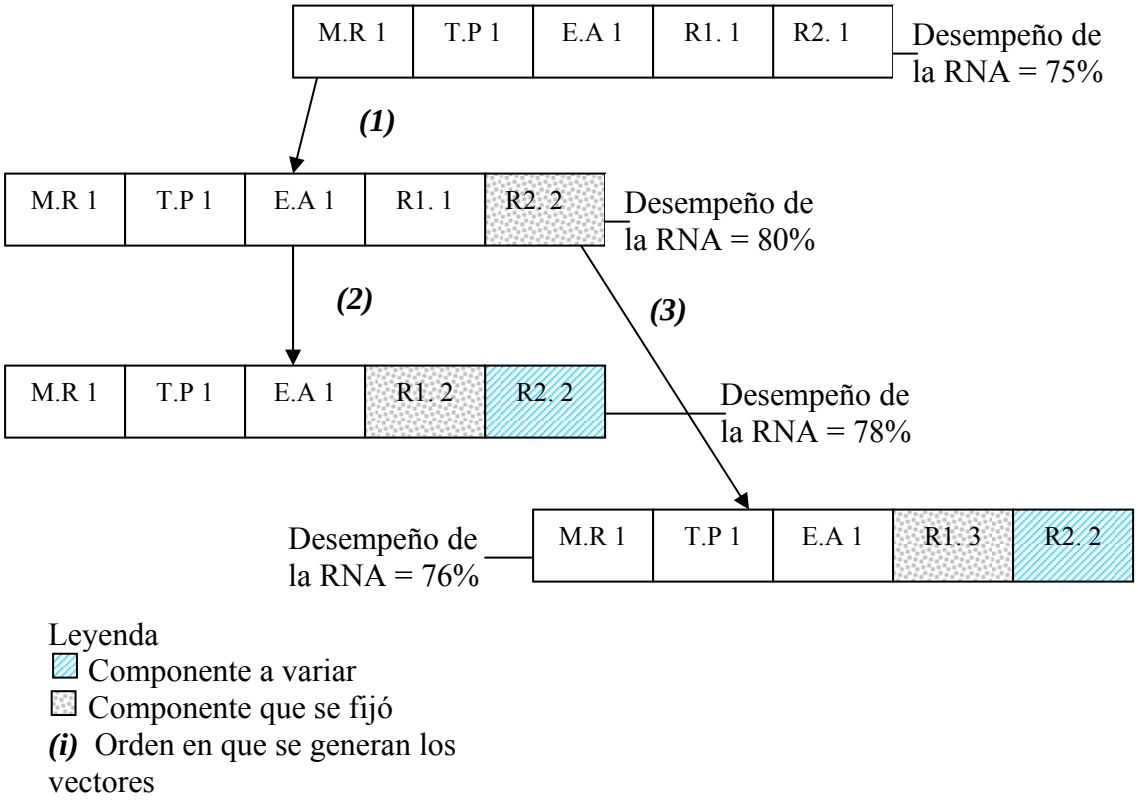

Fig. 9 Ejemplo Hill Climbing

Con la variante de Hill Climbing que incluye backtracking, si se explora todo el dominio de valores de una componente y no se obtiene un resultado superior al mejor desempeño obtenido hasta el momento, se procede a probar con los valores del dominio de la componente que se exploró anteriormente, que no han sido probados. La componente del vector que se exploró completamente sin obtener un mejor resultado, tomará el valor que tenía antes de comenzar a explorarse su dominio por última vez. Esto se hará siempre que

la cantidad de valores explorados no exceda al 70%. Asumamos ahora que los rasgos pueden ser tratados de cinco formas diferentes entre variantes de discretización y fuzzificación. La figura 10 muestra como funcionaría el backtraking en este ejemplo.

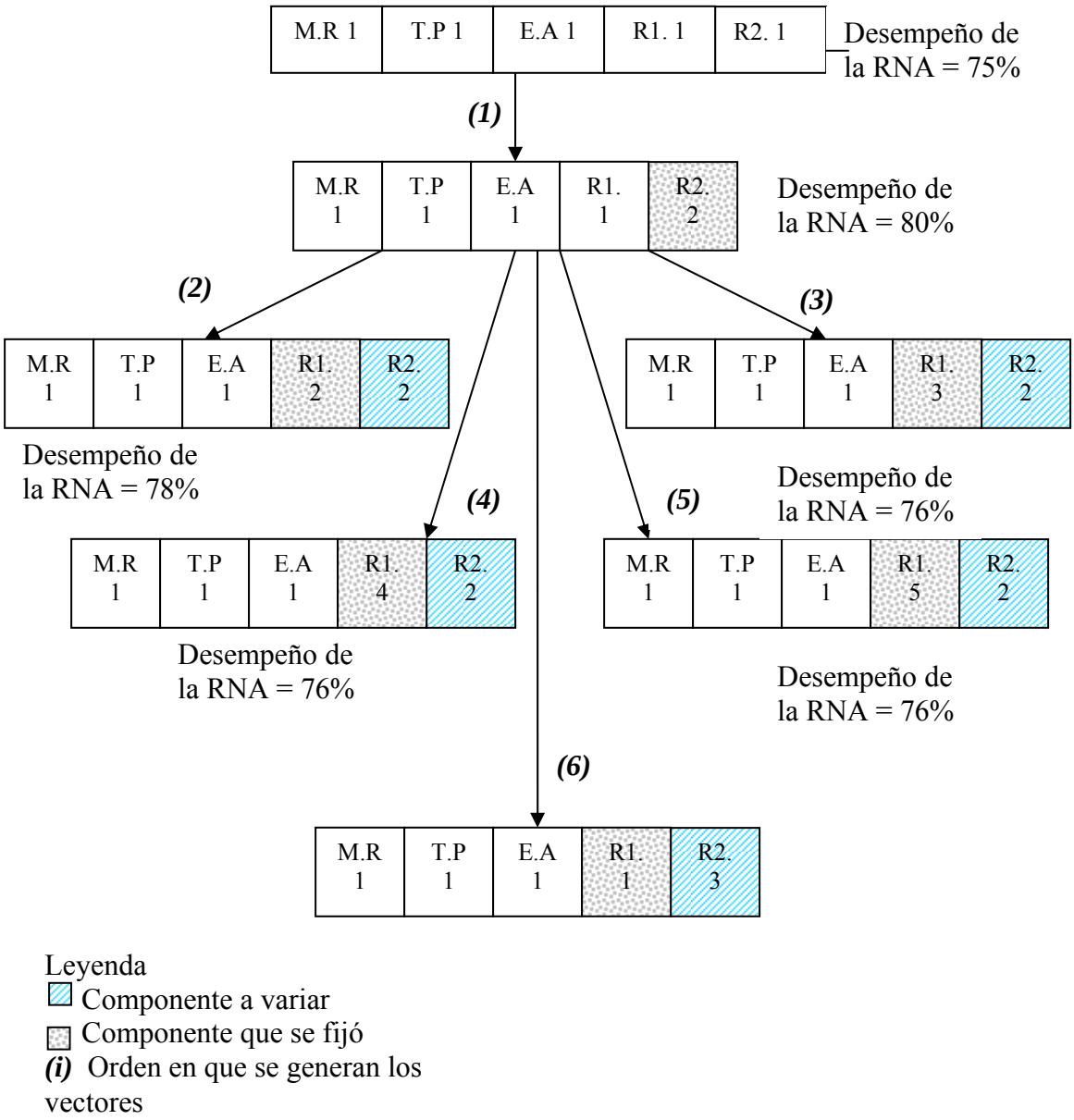

Fig. 10 Ejemplo de backtraking en Hill Climbing.

Este proceso se detendrá cuando el dominio de valores de cada componente del vector se haya explorado totalmente o cuando se encuentre un rendimiento suficientemente bueno en una instancia del vector (>95%).

El mismo procedimiento se ejecuta a partir de cada uno de los vectores iniciales, es decir para cada uno de los modelos de RNA. Si alguno de ellos se detiene por obtener un rendimiento muy bueno en una de las instancias del vector, los otros detendrán la búsqueda. Si esto no ocurre, entonces se compararán los desempeños que brindaron mejores resultados con cada uno de los modelos de RNA y se seleccionará el mejor así como la configuración del vector que le corresponda.

Con este diseño se reduce considerablemente el número de configuraciones del vector a explorar. Teniendo en cuenta que al determinar que un elemento de éste no ofrece una mejora en el desempeño de la RNA, el mismo no volverá a ser examinado al combinarlo con otro de los elementos no explorados aun.

3.2.2 Implementación del Hill Climbing

Para la implementación de la heurística descrita se construyó la clase *HillClimbing* (fig. 5). En ella se define el método *SearchChild* en el que se genera el nodo que corresponde según la definición de la heurística. En éste se definen los elementos que determinan el estado siguiente y se controla el backtracking. Luego de la generación de un nodo se invoca al método *GetPerformance* para conocer el rendimiento de la RNA y así determinar cual es el próximo nodo a generar. El método *GetPerformance* invoca al método *Execute* de la clase *Client* en el que se realiza el cálculo del rendimiento de la RNA, preprocesando primeramente los rasgos y utilizando este resultado como entrada de la RNA correspondiente, esto se explicó con mas detalles el capítulo anterior. Se define además el método *Run* quien se encarga de iniciar la generación de nodos invocando a *SearchChild* hasta llegar al estado final. Este método es ejecutado de forma paralela por tantos hilos como vectores iniciales se hayan formado.

La clase cuenta con un arreglo de estructuras que hace corresponder la cantidad de elementos del mismo con la cantidad de vectores iniciales. Cada elemento del arreglo

contiene el vector que representa al estado actual el desempeño de la red en dicho estado y un campo *_code* que señálala el estado de la ejecución de cada hilo: en ejecución, detenido por hallar un desempeño superior a 95%, o detenido por haber explorado totalmente todas las componentes del vector tanto como fuera posible. Dicha estructura se muestra en la figura 11.

```
public struct _perfParentDat
{
    public int _perfParent;
    public string [] _rama;
    public int _code;
}
```

Fig. 11 Estructura utilizada en la clase Hill Climbing

Las estructuras poseen un comportamiento muy similar a las clases, pero a diferencia de éstas, son tipos por valor y se almacenan en la pila. En esta aplicación ha sido empleada en la clase, mayormente, como objetos auxiliares que almacenan información temporal con las que se opera posteriormente.

Si durante la ejecución de uno de los hilos, se obtiene un desempeño superior al 95%, se detiene su ejecución y envía un mensaje que detenga la ejecución de los restantes. En caso contrario, se comparan las soluciones todos los hilos y se da como resultado final la configuración del vector que proporcionó un mayor rendimiento de la RNA.

Conclusiones

Con la realización de este trabajo se desarrolló un Módulo de Experimentación Combinatoria para la plataforma NeuroEvaluator 2.0, lográndose así dar cumplimiento al objetivo planteado a partir de que:

1. Se diseñó e implementó un módulo para el preprocesamiento de atributos simbólicos y numéricos.

2. Se diseñó e implementó un sistema distribuido de tareas que cumple con los requerimientos del Módulo de Experimentación Combinatoria, utilizando las facilidades que brinda .Net Remoting. Esta variante permite que se puedan evaluar al mismo tiempo variantes de solución que brinda la herramienta para un mismo conjunto de datos, una por cada terminal que se esté utilizando.

3. Se diseñó e implementó una heurística para explorar solamente las combinaciones de las variantes de codificación más prometedoras. De esta manera, en la solución de este problema de configuración, no se examinan todas las variantes de solución posibles con esta herramienta para un conjunto de datos, podándose el árbol de búsqueda.

Recomendaciones

Teniendo en cuenta los resultados alcanzados se recomienda:

1. Validar la factibilidad del Módulo de Experimentación Combinatoria

2. Explorar otras variantes de búsqueda mediante la aplicación de algoritmos de búsqueda estocásticos basados en ACO o AG

Referencias Bibliográficas

Bello, R.E., et. al, *Aplicaciones de la Inteligencia Artificial.* Ed. Universidad de Guadalajara. 2002

Buckley J. J., Eslami E., (2002) *An Introduction to Fuzzy Logic and Fuzzy Sets.* pp 31-35

Falcón, R.J., (2003) *Herramienta para el desarrollo de sistemas conexionistas en presencia de rasgos difusos.* Trabajo de Diploma.

Ferguson, J. et al., (2002) *La Biblia de C#.*

García, M.M. y P.R. Bello, (1996) *A model and its different applications to case-based reasoning. Knowledge-based systems 9* 465-473.

García, M.M.; Rodríguez, Y. y R.E. Bello, (2000) *Usando conjuntos borrosos para implementar un modelo para sistemas basados en casos interpretativos*, Congreso Iberoamericano de Inteligencia Artificial (IBERAMIA).

Hong, J., (1998) *On the handling of fuzziness for continuous-valued attributes in decision tree generation.* Fuzzy Sets and Systems 99.

Hussain, F. Liu, H., (1999) *Discretization: An Enabling Technique*. TRC6/99., disponible en: http://www.comp.nus.edu.sg/~liuh

Jyh-Shing, R. et. al., (1998) *Neuro-Fuzzy and Soft Computing*. Prentice Hall

Kurgan L.; et. al, (2004): *CAIM Discretization Algorithm.* IEEE Transactions on Knowledge and Data Engineering. Vol 16. No. 2

Liu H. and Setiono R., (1997): *Chi^2: Attribute selection and discretization of numeric attributes*. In Proceedings of the IEEE 7th Conference on tools with AI.

Nauck, D. et al. (1997) *Foundations of neuro fuzzy systems, John Wiley & Sons, Inc*. ISBN 0-471-97151-0

Östermark, R., (2000) *A hybrid genetic fuzzy neural network algorithm designed for classification*.

Rammer, I., (2002) *Advanced .NET Remoting.*

Russell, S. Norvig, P., (1995) *Artificial Intelligence A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ.

Varela, A. J., (2005) *Estimación automática de parámetros de funciones de pertenencia.* Tesis de Grado. Universidad Central "Marta Abreu" de Las Villas.

Wilson, R y T.R Martinez, (1997) *Improved Heterogeneous Distance Functions. Journal of AI Research No 6*.

Zadeh, L.A., (1994) *The fuzzy systems handbook: a practitioner's guide to building, using and maintaining fuzzy systems.* Academic Press.

Anexos

Anexo #1 Fichero de definición de rasgos (.defx)


```
<DEFINITIONS_FILE>
  <FEATURE name="feature1">
    <INTERVAL>
      <INTERVALS>
        <VALUE>4.3</VALUE>
        <VALUE>5.4</VALUE>
        <VALUE>5.9</VALUE>
      </INTERVALS>
    </INTERVAL>
  </FEATURE>
  <FEATURE name="feature2">
    <INTERVAL>
      <DISCRETIZER name="Chi Square">
        <DISC_PARAM name="INCONSISTENCY">7</DISC_PARAM>
      </DISCRETIZER>
    </INTERVAL>
  </FEATURE>
  <FEATURE name="feature3">
    <FUZZY>
      <MANUAL>
        <LINGUISTIC_TERM name="small" mfType="TRIANGLE">
          <PARAM name="A">0.5</PARAM>
          <PARAM name="B">0.7</PARAM>
          <PARAM name="C">1.1</PARAM>
        </LINGUISTIC_TERM>
        <LINGUISTIC_TERM name="large" mfType="TRIANGLE">
          <PARAM name="A">2.2</PARAM>
          <PARAM name="B">4.3</PARAM>
          <PARAM name="C">5.5</PARAM>
        </LINGUISTIC_TERM>
      </MANUAL>
    </FUZZY>
  </FEATURE>
  <FEATURE name="feature4">
    <FUZZY>
      <AUTOMATIC>
        <METHOD_NAME>All Triangle</METHOD_NAME>
        <BASE_DISCRETIZER>Equal Frequency</BASE_DISCRETIZER>
        <TUNING>False</TUNING>
      </AUTOMATIC>
    </FUZZY>
  </FEATURE>
  <FEATURE name="feature5">
    <DISCRETE />
  </FEATURE>
</DEFINITIONS_FILE>
```

Anexo #2 Fichero .pools (neuronas generadas por el preprocesador por cada rasgo)


```
<POOLS>
  <POOL>
    <NEURON>4.3;5.55</NEURON>
    <NEURON>5.55;6.25</NEURON>
    <NEURON>6.25;7.9</NEURON>
  </POOL>
  <POOL>
    <NEURON>2.1;2.95</NEURON>
    <NEURON>2.95;3.05</NEURON>
    <NEURON>3.05;4.4</NEURON>
  </POOL>
  <POOL>
    <NEURON>1.1;2.45</NEURON>
    <NEURON>2.45;4.75</NEURON>
    <NEURON>4.75;6.9</NEURON>
  </POOL>
  <POOL>
    <NEURON>0.1;0.8</NEURON>
    <NEURON>0.8;1.75</NEURON>
    <NEURON>1.75;2.5</NEURON>
  </POOL>
  <POOL>
    <NEURON>IRIS-SETOSA</NEURON>
    <NEURON>IRIS-VERSICOLOR</NEURON>
    <NEURON>IRIS-VIRGINICA</NEURON>
  </POOL>
</POOLS>
```