Departamento de Informática

PROYECTO FIN DE CARRERA

Ingeniería Técnica en Informática de Gestión

"OPTIMIZACIÓN EVOLUTIVA DE DISTANCIAS PARA CLASIFICADORES BASADOS EN PROTOTIPOS"

Autor: Rubén Villagarcía Vicente

Tutor: José María Valls Ferrán

Director: Ricardo Aler Mur

Leganés, marzo de 2011

A mis padres,
por su infinita paciencia.

CONTENIDOS

Capítulo 3: Sistema desarrollado

Capítulo 4: Experimentación

ILUSTRACIONES

TABLAS

ECUACIONES

Capítulo 1

Introducción

Mediante el siguiente documento se pretende dar al lector una idea global del proyecto llevado a cabo. Por ello este capítulo resumirá el contenido completo del proyecto, que será expuesto en los sucesivos capítulos del documento de manera más amplia, así como se expondrán los objetivos perseguidos para la realización del mismo. Además de dar un resumen global del proyecto también serán resumidos los diferentes apartados mediante la explicación de la estructura del mismo.

1.1.- Resumen y objetivos del proyecto

El proyecto consistirá en la implementación de un algoritmo que mejore la técnica de clasificación del vecino más cercano, utilizando técnicas evolutivas.

El algoritmo del vecino más cercano es una técnica de clasificación cuyo mecanismo de aprendizaje consiste simplemente en almacenar los datos de aprendizaje. Para clasificar nuevos datos, les asigna la clase del dato más cercano de entre los de aprendizaje. Uno de los problemas de esta técnica de clasificación es que para cada dato que se desea clasificar, necesita computar todas las distancias a los datos ya almacenados. Una segunda cuestión es que es muy sensible a la función de distancia utilizada (típicamente la euclídea, pero no siempre esta es la más adecuada).

Una manera de solventar el primer problema es sustituir los datos de entrenamiento por unos pocos prototipos bien elegidos, pero tiene el problema de que hay que determinar el número de prototipos y situarlos en el espacio de datos. El segundo problema se puede atacar modificando la función de distancia para adaptarla al problema.

En este proyecto se pretenden abordar ambas cuestiones mediante una técnica de búsqueda local, inspirada en algoritmos evolutivos. Es decir, dicha técnica encontrará el mejor posicionamiento de los prototipos y la mejor función de distancia para el problema de clasificación concreto que se quiera resolver. La heurística que guiará la búsqueda será la maximización del porcentaje de aciertos del algoritmo del vecino más cercano.

Un punto del espacio de búsqueda consiste en un conjunto de prototipos (cuya cantidad es un parámetro del algoritmo) y una función de distancia representada mediante una matriz (que a su vez representa una distancia euclídea plenamente ponderada, como se describirá en la sección 2.2.3). La definición de dicha distancia representada por una matriz M, entre un punto P y un centroide C, se puede ver en la ecuación 1.

La búsqueda local partirá de un punto inicial del espacio de búsqueda, cuyos prototipos son los centroides (centros de masas) de una de las clases y la distancia inicial es la distancia euclídea. El algoritmo irá modificando la posición de los prototipos y la matriz que representa la función de distancia, intentando maximizar la tasa de aciertos del algoritmo del vecino más cercano.

$$d(P,C) = \sqrt{(P-C)^T M^T M (P-C)}$$

Ecuación 1: Distancia euclídea ponderada.

Por tanto, el algoritmo implementado irá modificando las variables C (prototipo) y M (matriz) para conseguir la fórmula de la distancia más adecuada para sus objetivos. Para la modificación de las variables se cuenta con dos atributos configurables que son la probabilidad que tiene una posición de un elemento modificable sea mutada y otra que dará valor a esa mutación, ponderada con la varianza del elemento a mutar y un valor aleatorio devuelto por una distribución normal.

Una mejora en la tasa de aciertos hará que la matriz y el prototipo responsables de ella pasen a ser los objetos mutables en la siguiente generación.

Tras la primera mejora del porcentaje de aciertos, el cálculo de la distancia sí pasará a ser ponderado ya que la matriz dejará de ser la matriz identidad y no se hará entre el individuo y el punto medio de la clase sino con la mutación de este último.

El objetivo del proyecto por tanto será que tras el transcurso de N generaciones para un dominio dado lleguemos a encontrar las matrices que representen la mejor distancia, así como el prototipo o prototipos que mejor describan el conjunto de datos original. El objetivo último es optimizar la tasa de aciertos del algoritmo del vecino más cercano.

1.2.- Estructura del proyecto

A lo largo de este documento se pasará a explicar el algoritmo desarrollado con más detenimiento así como los resultados obtenidos tras su ejecución. El documento queda por tanto dividido en los siguientes capítulos:

1.2.1.- Capítulo 2: Contexto

En este capítulo se explicará el contexto en el que ha sido desarrollado el proyecto. En él se dará una descripción teórica de las técnicas que influyen en la realización, se hablará de los problemas de clasificación así como de la búsqueda local y de los algoritmos evolutivos de los que se recogen algunas de sus características.

1.2.2.- Capítulo 3: Sistema desarrollado

En este punto se tratará el sistema desarrollado, el algoritmo llevado a cabo será explicado en detalle así como la codificación del mismo y las herramientas utilizadas para ello.

1.2.3.- Capítulo 4: Experimentación

Mostrará los resultados obtenidos con la utilización del algoritmo con diferentes dominios. Para comprobar la eficacia del mismo, estos resultados serán comparados con los que dan la situación inicial de los puntos medios de cada clase y la matriz identidad ponderando la distancia para poder comprobar la evolución de los resultados con la ejecución del algoritmo así como poder compararlos con los arrojados por otros tipos de algoritmos con la misma finalidad.

1.2.4.- Capítulo 5: Conclusiones

Para finalizar se expondrán las conclusiones a las que se ha llegado con la realización del algoritmo y se propondrán líneas de investigación que podrían ser llevadas a cabo en el futuro.

Capítulo 2

Contexto

2.1.- Búsqueda local

En este apartado se dará una visión general de los algoritmos de búsqueda local, poniendo algunos ejemplos de tipos de métodos algorítmicos de búsqueda local.

2.1.1.- Introducción

Los métodos algorítmicos de búsqueda local se utilizan para la búsqueda de la solución óptima entre un grupo de soluciones candidatas. Para ello recoge los datos de la información local y los individuos cercanos para poder llegar mediante una serie de iteraciones a una solución final o una condición de satisfacción. El camino para llegar a este punto final es irrelevante, lo importante en estos métodos es conseguir llegar al objetivo.

El camino que siguen estos algoritmos para buscar la solución puede ser dirigido por diferentes heurísticas que les permiten innovar en los sucesivos pasos para llegar al objetivo buscado. El funcionamiento genérico se resume en que a partir de un estado inicial tenemos que llegar a una solución que mejore este estado inicial y para ello utilizaremos los medios explicados en esta introducción.

El gran problema de estos métodos es el peligro de un estancamiento producido por un mínimo local de la solución del problema.

Como se comentó anteriormente, se pueden seguir diferentes heurísticas para la obtención de la solución, debido a ello encontramos diferentes tipos de métodos de búsqueda local. A continuación se hablará de algunos de ellos.

2.1.2.- Método de búsqueda por escalada (hill-climbing)

Los métodos de búsqueda local mediante escalada [Russel y Norvig, 2003], trazan el camino a seguir mediante una función objetivo, el resultado de esta función objetivo será la heurística utilizada. Dado un estado inicial se comprobarán los datos más cercanos buscando el vecino que nos dé una heurística más satisfactoria, pasando este a ser el estado actual, continuando con este funcionamiento hasta que llegamos a la función objetivo.

Una característica de este algoritmo es que no "recuerda" el camino seguido para llegar al estado actual, es decir, una vez se pasa de un estado a otro, tanto el estado anterior como sus vecinos son olvidados. Sólo conoce su estado actual y el estado final, a partir de estos se calculan los vecinos y los valores para cada uno de ellos de la función objetivo.

El estado final puede buscar tanto un máximo como un mínimo, depende del problema en cuestión. Es muy importante la heurística que decidamos para llegar al resultado ya que será la que nos dará un mejor o peor algoritmo.

Una gran ventaja de este tipo de búsquedas es que el estado final puede ser alcanzado rápidamente y ahorra bastante memoria debido a que no guarda el árbol de estados. Sin embargo, cuenta con el inconveniente de que podemos llegar a un estancamiento debido a diferentes razones:

- Máximo (o mínimo) local: cuando ninguno de los vecinos mejora la heurística del estado actual.
- Meseta: todos los vecinos devuelven un valor igual para la función objetivo.
- Cresta: se produce un efecto escalón, la pendiente de la función sube y baja.

El algoritmo de este método se puede resumir de la siguiente forma:

```
Procedure Hill-Climbing
        estadoActual = estadoInicial
        fin = false
        while no fin do
                hijos = Buscar_Vecinos (estadoActual)
                mejorHijo = Buscar_Mejor_Heuristica (hijos)
                if mejorHijo > estadoActual then estadoActual = mejorHijo
                else fin = true
        end while
end Procedure
```

Para dar una visión gráfica del método usaremos los últimos pasos de un problema 8-puzzle con una heurística que se basará en que el mejor estado es el que tenga más piezas bien colocadas, es decir, el estado objetivo es H=8.

Este problema consiste en un tablero de nueve posiciones en el que hay ocho números (del 1 al 8) y una casilla vacía. Para resolver el problema se debe conseguir que los ocho números queden ordenados de izquierda a derecha y de arriba a abajo ascendentemente y que el hueco vació quede en la casilla inferior derecha. Para llegar a esta solución tan solo se puede mover en cada paso uno de los números contiguos a la casilla vacía hasta esta posición.

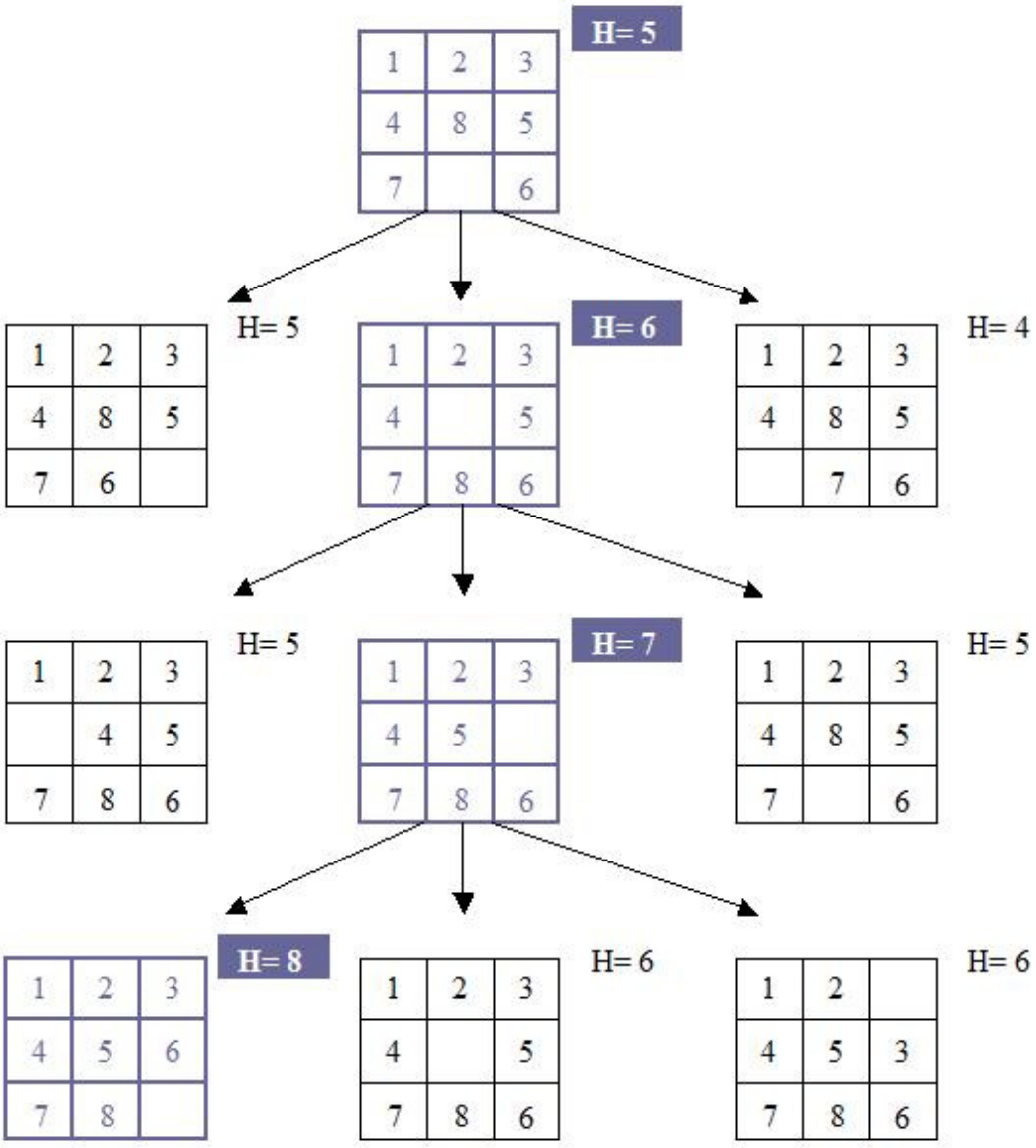

Ilustración 1: Resolución mediante algoritmo de escalada de un 8-puzzle.

2.1.2.1.- Variaciones del método de búsqueda hill-climbing

Como hemos visto el método de búsqueda por escalada tiene algunos inconvenientes, por ello existen unas variaciones de dicho algoritmo además de la explicada anteriormente (buscar el mejor vecino), algunas de ellas son:

- Escalada estocática (Stochastic hill climbing): escoge aleatoriamente uno de los vecinos con mejor heurística, no necesariamente el que tenga la mejor.
- Escalada de primera opción (Simple hill climbing): escoge el primer vecino que se encuentre con mejor resultado en el función objetivo.
- Escalada con reinicio aleatorio (Random restart hill climbing): se realiza la búsqueda varias veces, buscando un nuevo estado inicial en cada ocasión.

2.1.2.- Algoritmos evolutivos

Los algoritmos evolutivos [Bäck, 1996; Eiben y Smith, 2003] son una mejora dentro de los algoritmos de búsqueda local y están basados en la propia naturaleza. Su funcionamiento es una versión computerizada de las poblaciones de seres vivos dentro de su hábitat, es decir, emulan la reproducción de los individuos así como la selección natural de éstos.

Los primeros antecedentes históricos de estas técnicas los encontramos en la obra *El origen de las especies* [Darwin, 1859], de las leyes descritas por Darwin se derivan los principios básicos de los algoritmos genéticos [Holland, 1975]:

- Existe una población de individuos con diferentes características y habilidades. El número de individuos de una población está limitado.

- Aparecen nuevos individuos en la naturaleza con propiedades similares a sus antecesores.

- Debido a la selección natural los individuos más prometedores tienen más probabilidades de ser seleccionados para llevar a cabo la reproducción.

Volviendo a centrarnos en los algoritmos, los individuos de una población son en realidad posibles soluciones del problema dado, así como las características de cada uno de ellos vienen dadas por los diferentes atributos del dato. Estos atributos forman el código genético los individuos hijos heredarán de sus progenitores [Mendel, 1865].

Los algoritmos evolutivos comienzan con la generación de una población, todos los individuos de esta población estarán mejor o peor adaptados al medio, esto vendrá dado por el fitness que determina cuan buena o mala es una solución. Tras la generación de esta población se pasará a crear las siguientes generaciones, estas surgirán de la primera generación mediante la reproducción, mutación y selección natural de sus progenitores. En el siguiente ciclo la generación hija pasará a ser la progenitora y se repetirá el

proceso. Llegaremos al estado final cuando se hayan producido un número de generaciones que ha sido establecidas antes de la ejecución.

2.1.2.1.- Ejemplo de algoritmo evolutivo

A continuación se mostrará en forma de pseudocódigo el funcionamiento básico de un algoritmo evolutivo:

```
Procedure EvolutiveAlgorithm
        generacionActual = Generar_Poblacion_Inicial()
        fin = false
        generacion = 0
        generacionFinal = N
        while no fin do
                CalcularFitness(generacionActual)
                mejoresIndividuos = Seleccion (generacionActual)
                generacionActual = Reproducir_y_Mutar(mejoresIndividuos)
                generacion++
                if generacionFinal == generacion then fin = true
        end while
end procedure
```

2.1.2.2.- Operadores evolutivos

A continuación se pasará a explicar los diferentes operadores que hacen que consigamos las nuevas generaciones diferentes a sus padres:

- **Selección de individuos**: los nuevos individuos deben ser generados por los mejores habitantes de la generación anterior. El hecho de usar este operador está reflejado en la naturaleza en que los seres que consiguen reproducirse son los mejores adaptados al medio, es decir, tendrán descendencia aquellos que hayan sobrevivido, deben haber conseguido eludir a sus depredadores así como haberse procurado alimento y ser capaces de encontrar pareja para, por último, también ser aptos para la reproducción.
  En la naturaleza esto no siempre es así, puede que un individuo peor adaptado sea capaz de reproducirse, en el paso a algoritmo de este operador podemos simplemente escoger a los individuos con mejor fitness y reproducirlos. Esto se puede hacer de varias maneras:

  - Selección proporcional a la fitness: los individuos son seleccionados de manera proporcional a su fitness.
  - Método de torneo: los individuos compiten en grupos de cierto tamaño (dos o más) y es selecciónado aquel que tiene la fitness más alta.

- **Reproducción (clonación)**: en estos algoritmos también encontramos el operador de reproducción que en este caso se refiere a la clonación de un individuo de la población (la reproducción mediante dos individuos será llamada cruce). Este operador es utilizado debido a la necesidad de mantener a los individuos que tengan un fitness mayor. Una forma particular de usar este tipo de operador es el elitismo, que consiste en que para la siguiente generación mantendremos un número de individuos que se corresponderán con los mejores que encontremos durante el ciclo actual.

- **Cruce**: como se dijo en el anterior punto, la reproducción "sexual" en el contexto que nos ocupa será llamada cruce, mientras que el término reproducción quedará para las clonaciones. Con este operador lo que conseguimos es que los datos de los dos progenitores se mezclen entre sí para dar lugar a los individuos hijos. Los cruces se pueden hacer de diversas maneras, por ejemplo, los más comunes son los cruces binarios en los que un descendiente tendrá en cada atributo el valor que tenía uno de los padres para ese atributo mientras que el otro hijo tendrá los valores de los padres que no tomó su "hermano". Algunos ejemplos de cruces binarios son:

    - Cruce basado en un punto: una vez elegidos los individuos progenitores se decide un punto que será el punto de corte, darán lugar a dos descendientes, el primero tendrá los datos del primer progenitor del punto de corte hacia la izquierda y los del segundo padre del corte hacia la derecha, y viceversa para el otro hijo.

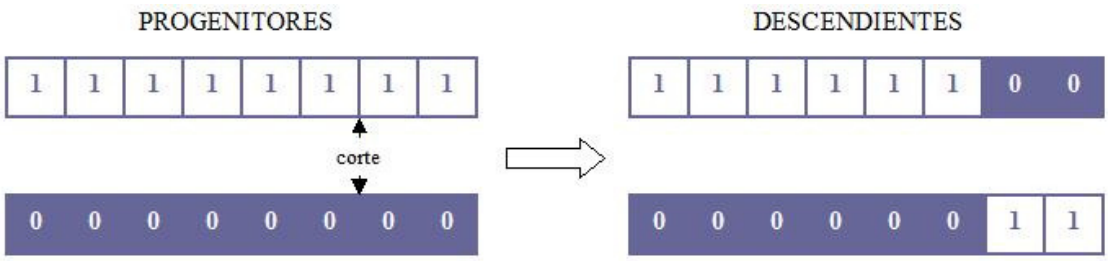

Ilustración 2: Ejemplo de cruce basado en un punto.

- Cruce punto a punto: en este caso un descendiente recogerá los datos impares del primer padre y los pares del segundo, mientras que el otro tendría los pares del progenitor número uno y los impares del número dos.

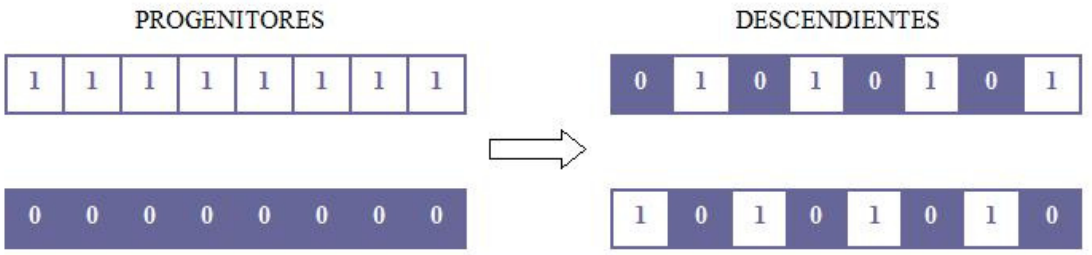

Ilustración 3: Ejemplo de cruce punto a punto.

- Cruce multipunto: será parecido al primero pero en vez de asignar un punto de corte asignaremos un número aleatorio de cortes.

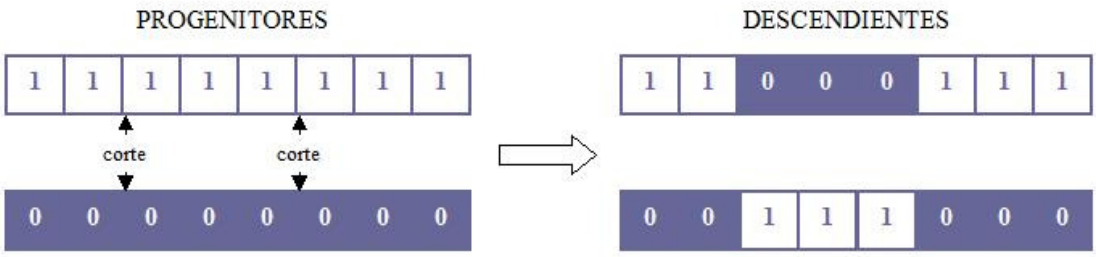

Ilustración 4: Ejemplo de cruce multipunto.

Existen otras variantes como el cruce de dos puntos o el cruce uniforme, en este último el cruce se realiza siendo el patrón dado por una máscara de cruce. Además de estos cruces binarios, también existen otros tipos de cruces no binarios que utilizan las medias de los atributos de los progenitores o las diferencias entre estos para calcular los valores que tendrán los descendientes.

- **Mutación:** el otro operador básico de estos algoritmos es la mutación, consiste en cambiar uno o varios atributos del individuo. Es un operador que nos proporciona un camino para poder expandir el espacio de búsqueda de la solución, ya que los anteriores operadores dependen por completo de los datos que ya tenemos en la población, sin embargo, al mutar algún elemento podemos encontrar atributos totalmente nuevos.

  La mutación suele venir asociada a un valor que nos diga el porcentaje de atributos que mutarán dentro de la población aunque podemos asociar por ejemplo ese valor al individuo o a un atributo en particular. La probabilidad de mutación puede ser fija para todas las generaciones o ir variando, así por ejemplo, se puede hacer que vaya disminuyendo en el tiempo con lo que conseguiremos que al principio el espacio de búsqueda sea más amplio para buscar en un espectro mayor a los mejor adaptados y a medida que los vamos encontrando reducir la búsqueda a los seres con características más próximas a estos individuos.

  En un dominio de datos binario el valor cambiará de un valor al otro (de 1 a 0 o de 0 a 1). Para los dominios con codificaciones reales tendremos diferentes tipos de mutación, como puede ser:

  - Mutación al azar: se define un rango para la mutación y la posición mutada tomará un valor cualquiera dentro de este rango, sin importar el valor que tenía anteriormente.

  - Mutación gaussiana: al atributo que se va a mutar se le aplicará una función con una distribución normal de media π y con una desviación estándar dada σ.

Ilustración 5: Ejemplo de mutación binaria.

2.2.- Algoritmos de clasificación. Distancias y transformaciones

2.2.1.- Algoritmos de clasificación

Mediante los algoritmos de clasificación es posible identificar de forma automática a qué clase dentro de las posibles soluciones de un problema pertenece una instancia, caracterizada por unos atributos determinados.

Existen dos grandes grupos de algoritmos de clasificación:

- Algoritmos no supervisados: al comienzo del algoritmo no se conocen las clases en las que se van a clasificar los datos. Por lo que es necesario buscar un método de organización automática de los datos.

- Algoritmos supervisados: se conocen las clases a las que pertenecen los datos, por tanto, es posible conocer a posteriori si la clasificación ha sido correcta o no.

2.2.2.- Algoritmos de clasificación supervisados

Como se ha explicado en el punto anterior, en este tipo de algoritmos conocemos las clases a las que pertenecen los datos. El funcionamiento de estos algoritmos se basa entonces en crear el modelo a partir de una serie de ejemplos – *instancias* – con unas características propias – *atributos* – etiquetados dentro de una determinada etiqueta o categoría – *clase* –.

Debido a que conocemos la clase a la que pertenecen los datos, es posible conocer si los resultados obtenidos con el modelo creado mediante el algoritmo son correctos o no.

2.2.3.- Algoritmo KNN (K Nearest Neighbors / K Vecinos más cercanos)

El algoritmo KNN [Cover y Hart, 1967] se engloba dentro de los llamados algoritmos perezosos. Este tipo de algoritmo se caracteriza porque no genera un modelo de manera explícita como sí que hacen otros métodos. Dado a que no se ha generado un modelo durante la fase de entrenamiento, la toma de decisiones comienza cuando se trata la primera instancia de test a clasificar.

Este algoritmo en particular basa su funcionamiento en la búsqueda de los vecinos más cercanos a la instancia de test. La cercanía de los vecinos está dada por la distancia de estos al individuo a clasificar, como se verá más adelante, la formula de la distancia puede ser calculada de diversas maneras. Los datos de entrenamiento son almacenados y cuando es necesario clasificar una instancia de test, se calculan las distancias a todas las instancias de entrenamiento, una vez calculadas se recogen las k más cercanas y en función de estas se clasifica el dato de test.

A continuación, se muestra una ilustración que servirá para mostrar con ejemplos el funcionamiento de dicho algoritmo:

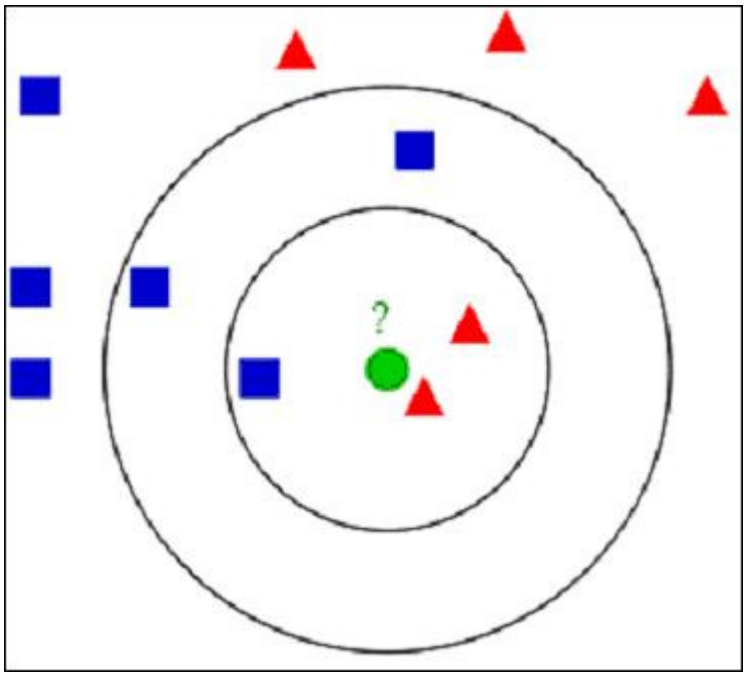

Ilustración 6: Ejemplo de algoritmo KNN.

El círculo verde de la imagen representa la instancia de test a clasificar. En este ejemplo, tenemos dos clases diferentes, por un lado, están los cuadrados azules que llamaremos clase *X* y, por el otro, los triángulos rojos o clase *Y*. Para simplificar la explicación, para el cálculo de la distancia se utilizará una distancia euclídea no ponderada, así como instancias con dos dimensiones. Con estos datos, si el parámetro *K* del algoritmo KNN fuese 3, es decir, que la clasificación de la instancia de test viniese dada por los tres vecinos más cercanos a esta, podemos comprobar que los vecinos más cercanos son los tres contenidos dentro del círculo más pequeño, por tanto, existen más vecinos triángulos rojos que cuadrados azules con lo que la instancia será clasificada como perteneciente a la clase *Y*. Sin embargo, si tomamos el algoritmo con los 5 vecinos más cercanos, tenemos que hay mayoría de vecinos de la clase *X* lo que da como resultado una clasificación de la instancia de test dentro de la clase *X*.

Un inconveniente de este algoritmo es que al no realizar explícitamente la búsqueda de un modelo de clasificación, cuando existe un número de datos elevado, la clasificación es demasiado costosa ya que es necesario el cálculo de la distancia con respecto a todos los datos de entrenamiento del problema.

2.2.4.- Algoritmos basados en prototipos

Debido a lo costoso del uso de un algoritmo como el KNN cuando nos encontramos con un alto número de individuos en la población, existe la posibilidad del uso de algoritmos basados en prototipos.

La idea principal de los algoritmos basados en prototipos es que utilizando los datos de entrenamiento se buscará un modelo de clasificación que utilizaremos para etiquetar posteriormente los datos de test. Consistirá en la creación de uno o varios prototipos por clase con los que se calcularán las distancias a las que se encuentran de la instancia de test y con ellas será posible clasificar el dato.

Un ejemplo de algoritmo basado en prototipos es el algoritmo LVQ [Somervuo y Kohonen, 1999], Learning Vector Quantization o Redes de cuantización vectorial. Es un algoritmo supervisado por vecindad, en el que se generan uno o varios prototipos por clase distribuidos aleatoriamente por el espacio de entrada. Durante la fase de entrenamiento se conseguirá la colocación final de los prototipos que van siendo desplazados con la entrada de nuevas instancias de entrenamiento. Cada nueva instancia de test será clasificada etiquetándola como el prototipo más cercano a esta. Por tanto, durante la fase de test sólo se calculan las distancias a los prototipos lo que hace que estas comprobaciones sean mucho más ligeras que las necesarias para la fase de test del algoritmo KNN.

2.2.5.- Distancias

Para poder saber cuan cerca está un dato de otro vamos a usar distancias, como se ha comentado anteriormente, comenzaremos usando una distancia euclídea no ponderada desde el dato a los puntos medios de las diferentes clases del problema. Posteriormente esta distancia se irá ponderando para encontrar una solución mejor [Atkeson, Moore y Schaal, 1997; Valls, Aler y Fernández, 2007].

Por esto, en este apartado se hablará de distintos tipos de distancias euclídeas:

- Distancia euclídea no ponderada.
- Distancia euclídea ponderada en diagonal.
- Distancia euclídea plenamente ponderada.

2.2.3.1.- Distancia euclídea no ponderada

Es la distancia más conocida. Se define como la longitud del segmento que une dos puntos.

Se calcula de la siguiente manera:

$$d(A,B) = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2} = \sqrt{(A-B)^T(A-B)}$$

Ecuación 2: Distancia euclídea no ponderada.

Podemos representarla gráficamente con el siguiente ejemplo. La distancia entre dos puntos es igual al espacio que los separa, así todos los puntos de una misma circunferencia están a la misma distancia del centro.

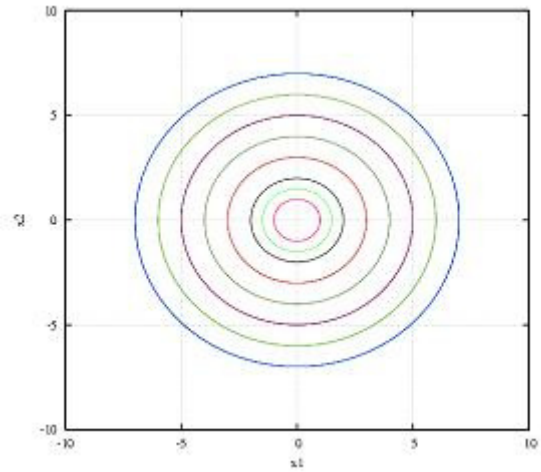

Ilustración 7: Ejemplo de distancia euclídea no ponderada.

2.2.3.2.- Distancia euclídea ponderada en diagonal

Es una distancia basada en la distancia euclídea no ponderada pero para cada diferencia que se realiza entre los puntos se multiplica por un factor de ponderación positivo.

Por tanto, la ecuación para calcularla es la siguiente:

$$d(A,B) = \sqrt{\sum_{i=1}^{n} (m_i (A_i - B_i))^2} = \sqrt{(A-B)^T M^T M (A-B)}$$

Ecuación 3: Distancia euclídea ponderada en diagonal.

Donde M es una matriz diagonal y $m_i = M_{ii}$ es el factor de ponderación que escalará la dimensión i.

En un ejemplo gráfico similar al anterior tenemos que los puntos que están a la misma distancia no forman circunferencias sino que pasan a formar elipses con los ejes paralelos a los ejes de coordenadas. Con esto, todo punto que esté en una elipse más interna estará más cerca del centro que cualquier punto de otra elipse que sea más

externa. Por ejemplo, todos los puntos de la elipse azul estarán más cercanos al centro que los de la elipse amarilla.

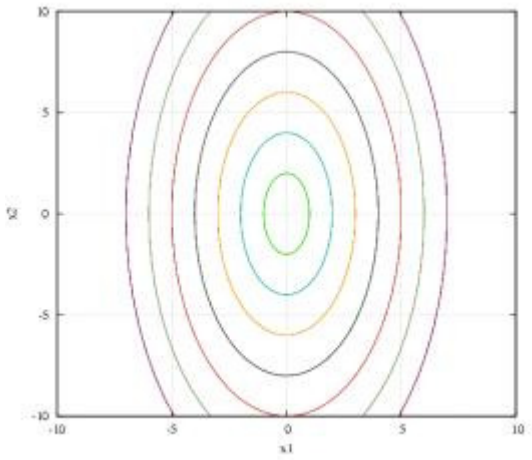

Ilustración 8: Ejemplo de distancia euclídea ponderada diagonalmente.

2.2.3.3.- Distancia euclídea totalmente ponderada

Cuando la matriz M ya no es diagonal sino que es una matriz cualquiera, obtenemos la distancia euclídea totalmente ponderada o generalizada.

Su fórmula es la que aparece a continuación:

$$d(A,B) = \sqrt{(A-B)^T M^T M (A-B)}$$

Ecuación 4: Distancia euclídea totalmente ponderada.

Donde M pasa a ser una matriz arbitraria que puede tomar cualquier valor en cualquiera de sus posiciones.

En el ejemplo gráfico podemos comprobar que los puntos que se encuentran a una misma distancia forman una elipse con respecto al centro, de igual manera que los

puntos de las elipses más internas estaban más cercanos al centro, los puntos de una elipse más interna serán más cercanos al punto central que los puntos que forman una elipse más externa.

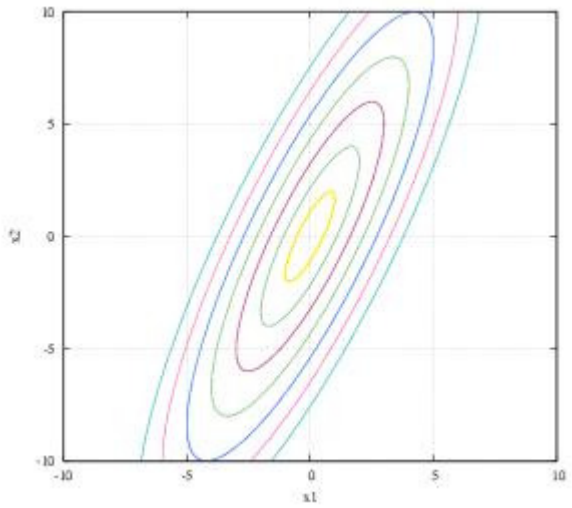

Ilustración 9: Ejemplo de distancia euclídea totalmente ponderada.

2.3.4.- Transformaciones

Una distancia euclídea generalizada que use la matriz M es equivalente a una distancia euclídea no ponderada sobre datos transformados con la misma matriz M:

$$d(A,B) = \sqrt{(A-B)^T M^T M (A-B)} = \sqrt{(MA-MB)^T (MA-MB)}$$

Ecuación 5: Distancia euclídea generalizada como transformación de distancia no ponderada.

La transformación de un dato A sería A' = MA.

Una de las ventajas de usar transformaciones en vez de distancias es que se puede aplicar cualquier algoritmo de clasificación aunque no esté explícitamente basado en distancias.

Para mostrar esto con más claridad utilizaremos el siguiente ejemplo, en el que tenemos un dato *b* al que se le aplica una matriz *M* para dar lugar al dato transformado *b'*.

$$\begin{pmatrix} b_1 & b_2 & \dots & b_n \end{pmatrix} * \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} b'_1 & b'_2 & \dots & b'_n \end{pmatrix}$$

Ilustración 10: Ejemplo de dato transformado.

Capítulo 3

Sistema Desarrollado

3.1.- Introducción

Este apartado servirá para dar una visión general del desarrollo que se ha realizado para llevar a cabo el sistema que nos ocupa.

3.1.1.- Objetivo del sistema

El objetivo de este desarrollo será conseguir una aplicación que codifique un algoritmo capaz de mejorar, mediante técnicas evolutivas, la clasificación conseguida con la técnica del vecino más cercano.

El algoritmo del vecino más cercano clasifica los datos buscando la solución del dato más cercano a este. Este sistema de clasificación se encuentra con dos problemas: uno de ellos es que es necesario comprobar las distancias a cada uno de los diferentes datos del problema. Por otro lado, este algoritmo es muy sensible a la distancia utilizada para el cálculo de la cercanía a los demás puntos y no siempre se usa la mejor fórmula para determinarla.

Para poder minimizar en cierta manera estos dos problemas se realizará un sistema que clasifique los nuevos datos utilizando un número configurable de prototipos en vez de comprobar la distancia a todos los datos de entrenamiento. Con esto solucionaremos el primer problema del problema del vecino más cercano. Para atajar el segundo problema, realizaremos modificaciones en la función de distancia que se irá utilizando.

La realización de esta solución comienza por la codificación de los datos para que la aplicación sea capaz de trabajar con ellos. Cada dato estará representado por sus atributos y la clase a la que pertenece, mediante un fichero de configuración le diremos al programa, entre otros parámetros, cuantas dimensiones y clases resultado posibles va a tener el problema. Tras la realización de la lectura y preparación de los datos para ser utilizados, calcularemos un dato por cada una de las clases del problema que será el centro de masas de los datos pertenecientes a dicha clase, este dato será el prototipo inicial con el que hará la clasificación.

El algoritmo codificado utilizará un bucle de validación cruzada, con un número configurable de particiones. Por tanto, cada uno de los datos tendrá asociado el identificador de partición a la que pertenece y un dato que dirá si esa partición es de entrenamiento o de test. Este valor irá cambiando cuando sea necesario en cada nueva pasada del bucle. El cálculo anteriormente explicado de los prototipos iniciales se realizará solo teniendo en cuenta los datos de entrenamiento.

Tras estos pasos, tendremos todos los datos preparados para comenzar el algoritmo de clasificación. Este algoritmo comenzará realizando el cálculo de la distancia euclídea de cada uno de los puntos de entrenamiento a cada uno de los prototipos de clase. Una vez calculada la distancia al prototipo de cada clase, se comprobará cual de todas ellas es la más pequeña, lo que nos dará la clase resultado esperada. Este valor esperado será comparado con la clase a la que realmente pertenece el dato, en caso de coincidir se contabilizará como un acierto. Cuando se ha terminado la contabilización de todos los aciertos que se han obtenido, se calculará el fitness, esto es, el porcentaje de aciertos sobre el total de datos tratados.

Realizado el cálculo del fitness inicial el algoritmo intentará encontrar un prototipo más adecuado así como una fórmula de la distancia que se adapte mejor a los datos tratados. Es aquí donde comienzan la evolución de los prototipos y de las funciones de distancias basada en los algoritmos evolutivos. El prototipo inicial dará paso a tantos prototipos como diga el número dado en el fichero de configuración y se realizará en cada pasada la evolución de estos, así como la distancia vendrá ponderada por una matriz. En el cálculo inicial esta matriz será la matriz identidad por lo que la distancia utilizada en comienzo será la distancia euclídea. Sin embargo, en cada pasada cualquiera de las posiciones de esta matriz puede ser mutada con lo que en las sucesivas generaciones la distancia usada será una distancia euclídea plenamente ponderada.

Estas mutaciones serán llevadas a cabo un número de ciclos dado, al final de cada ciclo se comprobará el fitness resultante de la distancia obtenida hasta cada uno de los prototipos con la fórmula mutada de la distancia. Si alguno de estos fitness da un porcentaje de acierto superior al del anterior ciclo se realizará un cambio del prototipo y

matriz a mutar por los que han dado este porcentaje más alto. En caso de no ser mejorado dicho porcentaje se realizará la mutación sobre los mismos que en la generación actual.

Una vez alcanzado el número de generaciones para esa ejecución del programa, tendremos el fitness para los datos de entrenamiento. Para comprobar que estos resultados obtenidos son realmente buenos, se cogerán el prototipo y la matriz resultado para calcular el fitness que nos dará con los datos de test. Este valor será guardado para que al finalizar todo el proceso de validación cruzada se realice la media de las tasas de acierto de las particiones de test, lo que nos dará el fitness final, que será el resultado a comparar con el dado por el algoritmo del vecino más cercano y comprobar si realmente hemos alcanzado el objetivo del proyecto.

3.1.2.- Desarrollo del sistema

El sistema ha sido desarrollado desde cero. Para la realización del mismo se ha utilizado el lenguaje de programación Java. Para la creación del código tanto como para realizar las pruebas ha sido usado el IDE Eclipse con un JDK para la compilación y ejecución del mismo en su versión 1.5.

3.1.3.- Requisitos del sistema

El sistema desarrollado cumple los siguientes requisitos:

- El sistema leerá los ficheros de configuración desde el subdirectorio “config” y los de datos desde el llamado “data”. El fichero elegido en cada uno de los casos podrá ser personalizado cambiando el valor de una constante.

- Cuando finalice la ejecución creará dos ficheros dentro del subdirectorio "result", uno contendrá las soluciones obtenidas en cuanto a matrices, prototipos y tasas de aciertos, el otro archivo tendrá la evolución que ha tenido el fitness a lo largo de la ejecución. Estos ficheros serán utilizados para el análisis posterior, al ser generados su nombre contendrá el nombre del fichero de datos utilizado así como la fecha completa en la que comenzó la ejecución que los generó, esto nos servirá para diferenciar diferentes ejecuciones, si esto no fuese suficiente para la identificación el fichero de resultados contendrá los datos de configuración utilizados durante esa ejecución.

- La evolución del fitness irá siempre creciendo ya que mediante elitismo en cada nueva generación, nos quedaremos con los resultados óptimos para la siguiente ejecución. Esto es, si alguno de los individuos de la nueva generación es mejor al individuo progenitor, utilizaremos este nuevo individuo como progenitor en el siguiente ciclo, en caso contrario, el progenitor seguirá siendo el mismo.

- Siempre que los requisitos de la máquina y sistema operativo en que corren las ejecuciones del sistema lo permitan, se podrá trabajar con problemas de cualquier dimensión.

- En principio, el sistema sólo trabajará con soluciones discretas, aunque es escalable a problemas de clasificación en caso de ser necesario.

- El sistema trabaja con validación cruzada, para ello podemos utilizar el número de particiones que creamos óptimo indicándolo en el sistema de configuración.

- El cálculo de la distancia siempre se realizará tan solo con una matriz que irá mutando para conseguir la mejor solución posible. Sin embargo, los prototipos de centroide utilizado para la generación de esta solución pueden

ser tantos como queramos, siendo indicado también en el fichero de configuración.

- La matriz inicial podrá ser elegida, las opciones serán la matriz diagonal (para comenzar con una distancia euclídea que irá evolucionando) o una matriz generada aleatoriamente. Durante la experimentación en este proyecto finalmente sólo se utilizó la matriz diagonal, la opción de la matriz aleatoria queda disponible para posteriores experimentaciones.

- El número de ciclos que se realizarán hasta que el sistema de por finalizada la ejecución será configurable.

- En caso de que en la mutación generada nos dé como resultado la misma matriz o el mismo prototipo, ésta será desechada y se entrará en un bucle del que no se saldrá hasta que no se haya encontrado una matriz o prototipo que difieran al menos en una posición con los originales. Esto será gestionado por un flag que se levantará en caso de que algún individuo haya mutado en alguna de sus posiciones.

- Otras dos características configurables serán tanto la probabilidad con la que una posición podrá ser mutada como la desviación que tendrá esta mutación. El valor en que esta mutación cambia el elemento puede ser el mismo que se ha pasado mediante el fichero de configuración o este valor multiplicado por la desviación del atributo actual, para elegir una u otra opción ya que también existe otro campo configurable de ponderación.

3.2.- Estructura del sistema

Se va a realizar en este apartado una visión general, a alto nivel, de la estructura del sistema. Tanto el funcionamiento como la descripción de la implementación serán tratadas en posteriores apartados.

3.2.1.- Arquitectura del sistema

Una descripción a muy alto nivel del sistema, se podría hacer con una descomposición en dos subsistemas. Una visión esquemática de esto podría ser la siguiente:

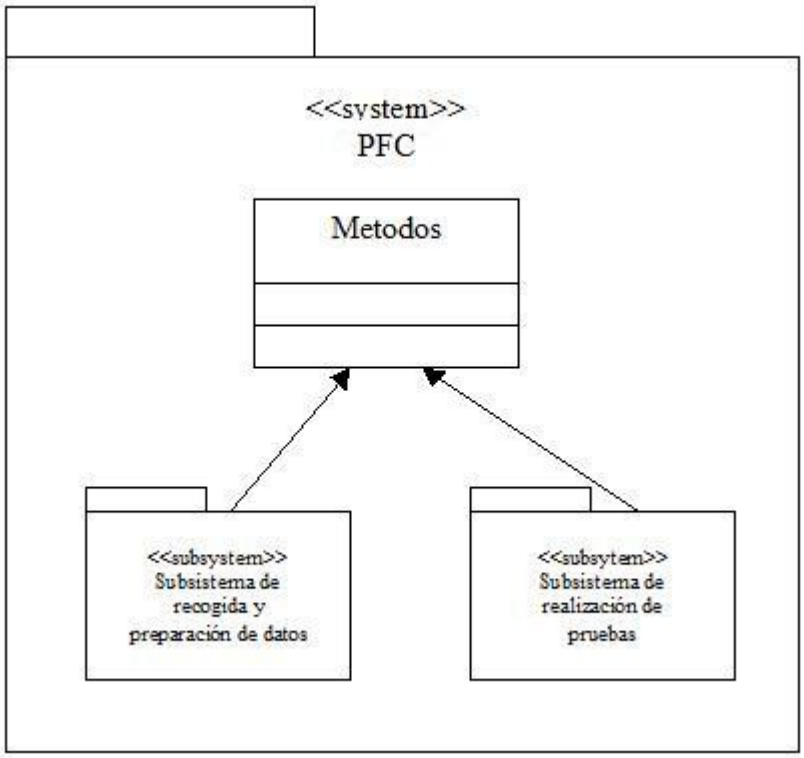

Ilustración 11: Esquema a alto nivel de la aplicación.

Los dos subsistemas principales se pueden resumir de la siguiente manera:

- **El subsistema de recogida y preparación de datos**
  En este subsistema se utilizarán las clases *Config*, *Dato*, y *Clase*. Servirán para realizar la lectura de los ficheros y su preparación para que el siguiente subsistema pueda realizar las pruebas oportunas en busca de la solución óptima.

- **El subsistema de realización de pruebas**
  Lo componen las clases *Matriz* y *Prueba*. Estas se ocuparán de que la información proporcionada por los anteriores sistemas sea tratada en búsqueda de la solución al problema.

Estos dos subsistemas, utilizarán una clase que les proporcionará la funcionalidad necesaria a cada una de ellas:

- **La clase *Metodos***
  Esta clase contendrá todos los métodos de utilidad general dentro del programa para la búsqueda de una óptima solución. Ambos subsistemas, por tanto, dependerán de ella. A su vez existirá una clase principal que sea la que organice toda la ejecución.

3.2.2.- Diagrama de clases del sistema

En este apartado se representa de manera más detallada como está organizada la estructura estática del programa.

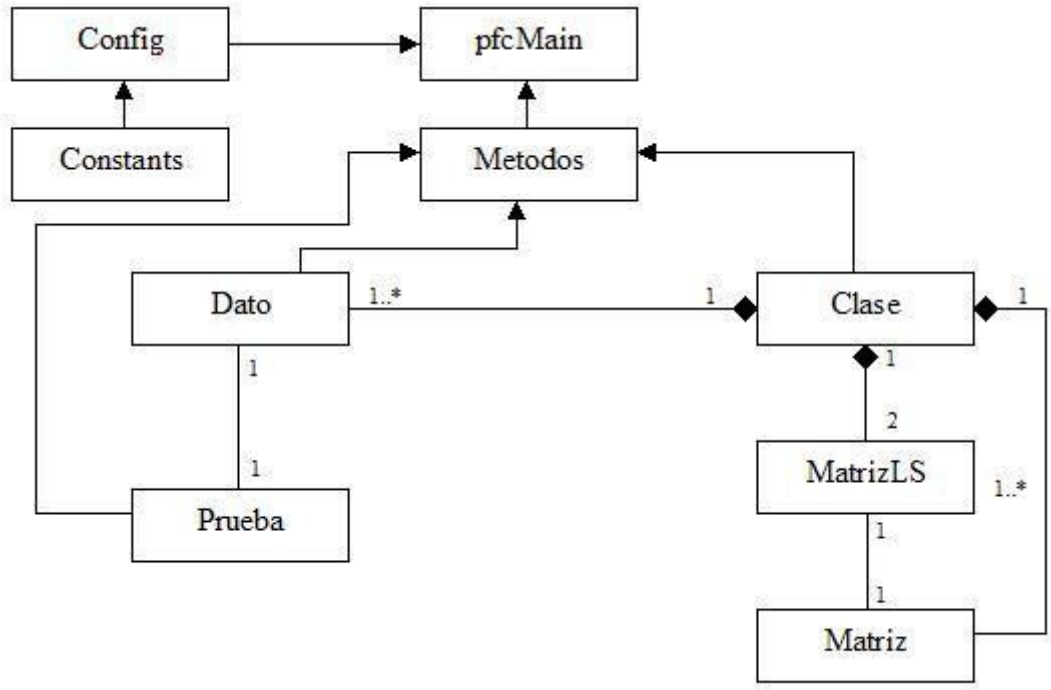

Ilustración 12: Diagrama de clases de la aplicación.

La clase *pfcMain* controlará la ejecución del sistema en general, por ello la clase *Config* y la clase *Constants* le proporcionarán los datos necesarios para conocer de dónde leer los datos y con que tipo de problema nos vamos a encontrar, así como conocer cual será la manera de buscar la solución.

La clase *Metodos* le proporcionará a dicha clase los métodos con los que realizará todas las funcionalidades necesarias durante la ejecución del problema. Esta clase necesitará de las demás clases para realizar las operaciones pertinentes, así como la clase principal necesitará de las demás clases para poder realizar una inicialización de las diferentes variables en las que se irá guardando la información necesaria.

Para la realización de las pruebas necesitaremos, en un primer lugar la clase *Dato* ya que gracias a ellas tendremos todos los datos que nos han sido proporcionados. Esta clase, a su vez dependerá de la clase *Clase* ya que cada uno de los datos es perteneciente a cada clase según la solución que tiene asociada.

Cada uno de los objetos de la clase *Dato* tendrá asociado a su vez un objeto de la clase *Prueba* en los que se calcularán las soluciones esperadas para finalmente comprobar si se corresponden con la solución real para contabilizar un acierto o no.

Durante la lectura de datos de configuración sabremos cuantos objetos de la clase *Clase* vamos a tener, así como los identificadores (posibles soluciones) que tendrán. A su vez, cada una de estas clases tendrá dos matrices del tipo *MatrizLs*, es decir, matrices que servirán para el cálculo de las distancia, una de ellas se corresponderá a la matriz perteneciente a la generación anterior y la otra a la matriz resultante de la mutación. Además, dependerán de la *Clase* x matrices de la clase *Matriz*, esta x vendrá dada por el fichero de configuración y se cada uno de uno objetos se corresponderán con los prototipos que se usarán en la búsqueda de la solución.

Como se ha dicho este apartado simplemente sirve para dar una visión global de cuales son las clases que han sido creadas para la realización del sistema, en próximos apartados se dará una visión más amplia y específica de como trabajan estas.

3.3.- Funcionamiento del sistema

En este apartado se dará una visión más completa del sistema implementado, así como de las clases que lo componen y la interacción entre ellas.

3.3.1.- Función principal

Este punto describirá el diseño y la implementación del método principal del sistema, este método *main* se encuentra en un fichero con código Java llamado *'pfcMain.java'*.
La funcionalidad del método principal del sistema se basa en la creación de las variables necesarias para la resolución del problema, así como de la organización de las llamadas a los distintos métodos específicos en el momento adecuado.

A continuación, se explicarán en diferentes subapartados el funcionamiento básico mediante diagramas de actividad.

3.3.1.1.- Bucle principal

El siguiente diagrama de actividad nos muestra el funcionamiento del bucle principal del método *main*:

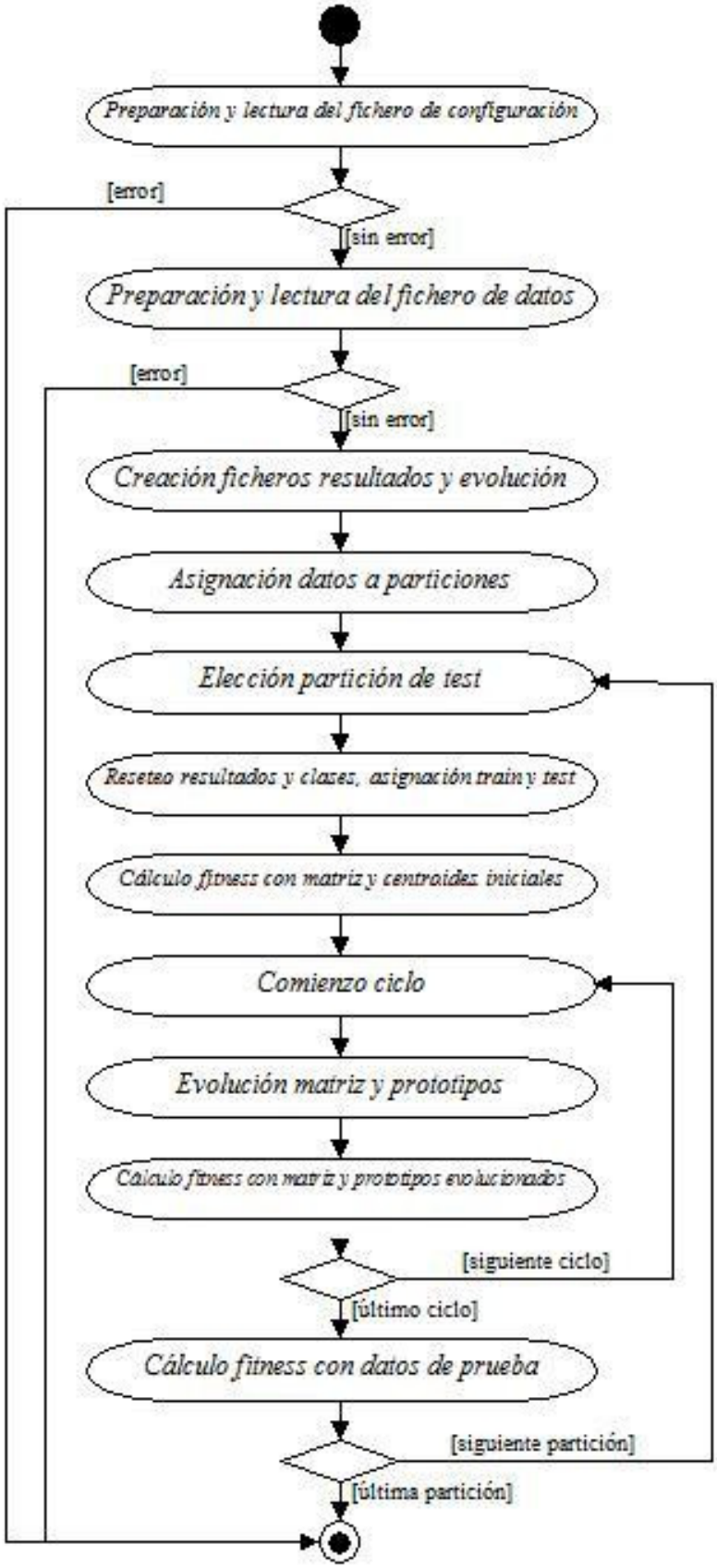

Ilustración 13: Diagrama de actividad de la aplicación.

A continuación, se pasa a detallar cada uno de los pasos en los que se divide el programa:

3.3.1.2.- Preparación y lectura del fichero de configuración

Al comenzar la ejecución se lee el fichero de configuración correspondiente según el valor de la constante creada a tal efecto. Durante la lectura, se desechan los comentarios que pueda contener el fichero y cada línea con información que haya se corresponderá con un dato de la configuración según su posición en el fichero.

Todos estos datos son guardados en una variable de la clase *Config*, en caso de que falten datos o que estos tengan valores no válidos, devolverá un error y la ejecución será finalizada en ese momento.

A continuación se muestra un ejemplo de fichero de configuración, en los comentarios tiene los nombres de los atributos, que serán desechados al instanciarlos en java. Realmente lo que determina que atributo es cada uno es su posición en el fichero.

```
#Particiones
10
//Dimensiones
2
#Ciclos
3000
#Probabilidad de mutación
0.25
#Mutación
0.1
#Clases
0 1
#Número de centroides
6
//Ponderar
S
```

Ilustración 14: Ejemplo de fichero de configuración.

3.3.1.3.- Preparación y lectura del fichero de datos

Si la lectura de los datos de configuración ha sido correcta, se pasa a recoger los valores y soluciones dadas para trabajar con ellos. Para ello, se crea un array de datos del tipo *Dato*, en ella quedan guardados los atributos, así como la solución y se prepara la estructura para las futuras pruebas.

Como en el caso anterior, si existe algún error durante la lectura, por ejemplo, que el número de dimensiones no se corresponda con lo que decía el fichero de configuración, entonces la ejecución finaliza.

3.3.1.4.- Creación de los ficheros de resultados y evolución

A lo largo de toda la ejecución del sistema se irán grabando los resultados que se vayan encontrando así como la evolución de estos en sendos archivos. En este momento son creados vacíos con un nombre que los haga identificable (con el nombre del fichero de datos y la fecha actual) para poder ir escribiendo en ellos cuando sea necesario. Para mejorar la identificación, en el fichero de resultados se añadirán todos los datos proporcionados por el fichero de configuración.

3.3.1.5.- Asignación de datos a particiones

En este paso del proceso, se agrupan los datos recogidos en particiones. Por tanto, cada dato quedará marcado con un atributo de tipo numérico que diferenciará la partición a la que pertenece. Gracias a este atributo cuando durante la validación cruzada la partición de test cambie también cambiarán de rol los datos asignados a esta, quedando todos los demás como datos de entrenamiento.

3.3.1.6.- Elección de partición de test

Como se ha dicho en el anterior punto, necesitaremos que en cada pasada de la validación cruzada una partición tome el rol de partición de test, en este punto se comenzará un bucle en el que en cada iteración, se pasará a asignar la siguiente partición como la de test.

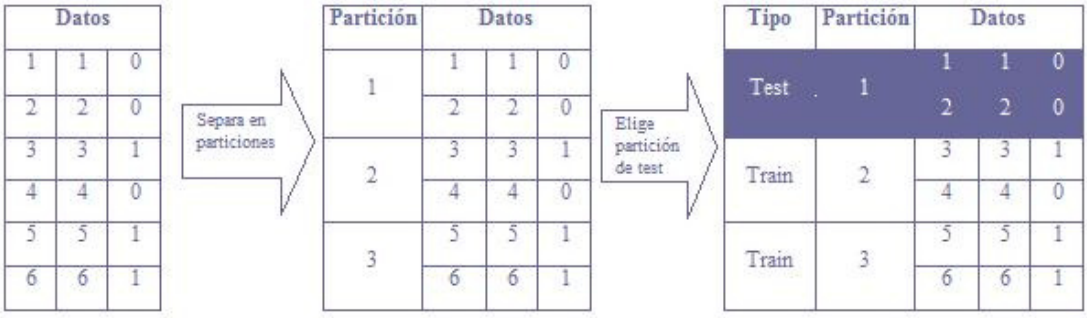

Ilustración 15: Separación de los datos en particiones y elección de test en el primer ciclo.

3.3.1.7.- Reseteo de resultados y clases, asignación train y test

Una vez elegida la partición de test, asignamos el tipo test a todos los datos que pertenezcan a ella y el tipo train a todos los demás datos recogidos. Además los resultados son puestos a cero para que sean independientes de los de las demás particiones.

Con los datos de train, recogemos la suma de los que tienen la misma solución y con ello generamos el array de clases, asignándole los id's que coincidirán con las diferentes soluciones posibles (vienen dadas en el fichero de configuración) y generando la matriz inicial para el cálculo de la distancia (en principio, la matriz identidad aunque es configurable utilizar una aleatoria) y el prototipo inicial, que vendrá dado por el punto medio de los datos de cada clase.

3.3.1.8.- Cálculo fitness con matriz y centroide iniciales

Independientemente del número de prototipos que se hayan designado en el fichero de configuración, en un principio para calcular un fitness inicial que nos sirva para saber si los siguientes mejoran o no las pruebas anteriores hacemos una prueba cogiendo la matriz y centroide que han resultado de la inicialización de cada una de las clases.
Para este cálculo calculamos la distancia de cada uno de los datos que hemos recogido y cogemos como solución esperada, el identificador de la clase que nos dé una distancia menor. Posteriormente esta solución esperada la comparamos con la solución real que venía en el archivo de datos y si coinciden la contabilizamos como acierto, al final contabilizamos el tanto por ciento de aciertos con respecto al total de datos, este será nuestro fitness inicial.

$$d(P,C) = \sqrt{(P-C)^T M^T M (P-C)} = sumarCuadrados((P-C)M)$$

Ecuación 6: Fórmula convencional de la distancia euclidea ponderada y su simplificación.

3.3.1.9.- Comienzo ciclo

El archivo de configuración nos ha dado el número de ciclos que tenemos, esto es el número de generaciones que tenemos para conseguir encontrar el mejor fitness. En este momento comenzamos un nuevo bucle para ir realizando las pruebas hasta que se nos agoten estos intentos.

3.3.1.10.- Evolución matriz y prototipos

Con cada nuevo ciclo realizamos una evolución tanto de la matriz como de los prototipos de cada una de las clases, para ello usamos los datos de configuración con la probabilidad de mutación y la desviación de esta mutación.

Con la probabilidad de mutación recorremos todos los datos de la matriz y generamos un número aleatorio, si este número es inferior a la probabilidad de mutación, entonces esa posición deberá mutar. La mutación tendrá un valor resultante de la multiplicación de la desviación dada por un número aleatorio de una distribución normal, este número aleatorio vendrá dado por la función *nextGaussian()* de la clase *java.util.Random*, también existe la opción de que esta mutación sea ponderada por la desviación del atributo actual. Esta mutación será sumada al valor de la posición actual, con ello tendremos una posición mutada.

$$Mut(x) = D_{mx} \cdot (M \cdot r.nextGaussian())$$

Mut(x) = Mutación generada para el dato x.
Dmx = Desviación media del atributo al que pertenece el dato x.
M = Valor de la mutación indicado en el fichero de configuración.
r.nextGaussian() = función de java que da un valor aleatorio de una distribución normal.

Ecuación 7: Cálculo de la mutación.

Tendremos además un flag que será marcado en caso de que alguna posición de la matriz haya sido mutada, si no se ha mutado ninguna, se repetirá el proceso hasta que al menos una posición haya sido mutada con lo que tendremos una matriz hija resultante diferente a la matriz padre.

Este proceso es repetido con los x prototipos dados, en el caso de la primera generación los x prototipos antes de mutar corresponderán con el prototipo inicial de la clase.

3.3.1.11.- Cálculo fitness matriz y prototipos evolucionados

Mediante el mismo proceso llevado a cabo en el punto 3.3.1.8 se calculará el fitness usando la matriz hija y los diferentes prototipos generados en el anterior paso.
Si alguno de los fitness generados mejora el valor del mejor fitness que tengamos en ese momento, la matriz hija pasará a ser la matriz padre y el prototipo usado pasará a ser el mejor, en caso contrario, quedarán como matriz padre y mejor prototipo los mismos que había en el anterior ciclo.

En este momento, volvemos con los mejores datos de nuevo al punto 3.3.1.9 para intentar mejorar aun más el fitness en un nuevo ciclo.

3.3.1.12.- Cálculo fitness con datos prueba

Una vez agotados los ciclos cogemos la matriz y el prototipo que nos han dado un mejor fitness y con ellos calculamos que tasa de acierto tenemos sobre los datos de prueba.
Finalizado el proceso guardamos esta tasa de acierto y volvemos al punto 3.3.1.6 para continuar con la siguiente partición. La tasa de acierto conseguida en la ejecución del programa vendrá dada por la media de todos fitness conseguidos sobre los datos de test de cada una de las particiones de la validación cruzada.

3.3.2.- Clase "Config"

La clase *"Config"* está formada por los atributos que vienen dados por la lectura del fichero de configuración, que son:

- Atributo *particiones*: número de particiones que se realizarán para llevar a cabo la validación cruzada.

- Atributo *dimensiones*: número de dimensiones que tendrán los datos que se manejarán en la ejecución de la aplicación.

- Atributo *ciclos*: número de generaciones que habrá antes de que se pare la ejecución, es decir, número que hará que el bucle para la búsqueda de la mejor solución se pare.

- Atributo *probMutacion*: probabilidad que tendrá cada posición de las matrices y prototipos de mutar.

- Atributo *desviación*: valor máximo que podrá variar una posición cuando es mutada.

- Atributo *clases*: listado de soluciones que podrán tomar cada uno de los individuos que son tratados por la aplicación.

- Atributo *centroides*: número de prototipos que se generarán en cada nueva generación.

- Atributo *ponderar*: opción que si está en estado afirmativo, las mutaciones serán ponderadas según la desviación del atributo a mutar.

Además de estos atributos existirán dos atributos con la aparición o no de un error así como del mensaje que este error genera. Estos dos atributos se usarán en los setters de la clase, ya que antes de asignar un valor a cada uno de los atributos, comprobará que los valores de estos atributos son correctos.

En esta clase también habrá un método encargado de la lectura del fichero de configuración que desechará los comentarios que se encuentren en él e irá asignando los valores de cada uno de los atributos según el orden en que se vayan leyendo.

A continuación se muestra una vista general de la clase:

Ilustración 16: Clase Config.

3.3.3.- Clase "Prueba"

Esta clase irá siempre asignada a un dato de entrenamiento o test, su función consiste en guardar los valores que se van a ir consiguiendo mientras se está realizando la búsqueda del mejor fitness. Debido a ello tendremos los siguientes atributos:

- Atributo *distancias*: este atributo consiste en una matriz con el resultado del cálculo de distancias que se consigue desde el dato actual a cada uno de los prototipos de cada clase. La matriz en un problema de N prototipos y M clases quedaría de la siguiente manera:

Distancia del dato actual al prototipo 1 de la clase 1.	Distancia del dato actual al prototipo 2 de la clase 1.	…	Distancia del dato actual al prototipo N de la clase 1.
Distancia del dato actual al prototipo 1 de la clase 2.	Distancia del dato actual al prototipo 2 de la clase 2.	…	Distancia del dato actual al prototipo N de la clase 2.
…	…	…	…
Distancia del dato actual al prototipo 1 de la clase M.	Distancia del dato actual al prototipo 2 de la clase M.	…	Distancia del dato actual al prototipo N de la clase M.

- Atributo *clasesObtenidas*: es un vector que servirá para guardar una vez relleno el atributo anterior cual es la solución (clase) obtenida comprobando cual de todas las distancias de cada prototipo a cada una de las clases es la menor. Con los datos anteriormente comentados este vector quedará de la siguiente manera:

Clase obtenida con el prototipo 1.	Clase obtenida con el prototipo 2.	…	Clase obtenida con el prototipo N.

- Atributo *aciertos*: con un vector que se corresponde con el anterior en sus posiciones guardamos, no la clase obtenida, sino si esa clase obtenida corresponde con la solución que el fichero de datos nos da como real, por tanto si son iguales será un acierto y si no lo es será un fallo. Este vector, por tanto, estará formado por datos de tipo boolean, acierto = true / fallo = false.

Una visión general de la clase:

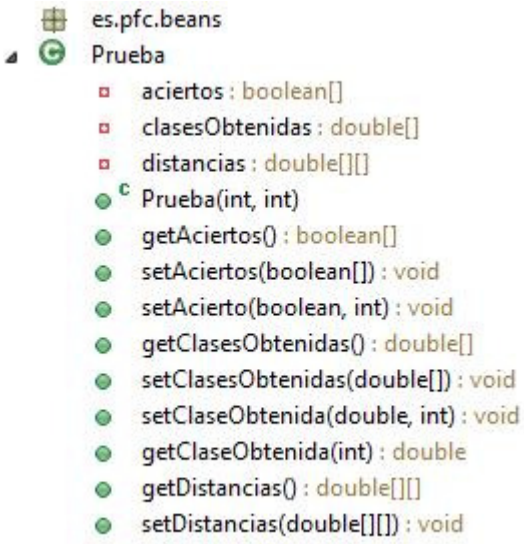

Ilustración 17: Clase Prueba.

3.3.4.- Clase "Dato"

Cada objeto de esta clase se corresponderá con cada una de las líneas del fichero de datos, es decir, con cada uno de los datos que tendremos para que sean usados como datos de entrenamiento y test.

Los atributos que deberemos guardar para cada uno de estos datos, serán:

- Atributo *atributos*: cada uno de los valores que hay en cada línea del fichero, salvo el último que se corresponderá con la solución, este atributo es un array cuya longitud viene dado por el valor *dimensiones* que haya en el fichero de configuración.

- Atributo *solucion*: como se ha dicho en el anterior guión se corresponde con el último valor, y será la clase solución que realmente se consigue con los atributos de ese dato.

- Atributo *particion*: este dato nos dará a que partición corresponde un dato dado, este valor es asignado después de leer todos los datos del fichero y comprobar con el número total de datos que se tienen donde se deben hacer los cortes de cada una de estas particiones.

- Atributo *tipo*: los dos tipos posibles a los que puede pertenecer un dato son *entrenamiento* y *test*, este valor cambiará a lo largo de la ejecución de la aplicación, ya que durante la validación cruzada se realizarán tantas pasadas como particiones haya y en cada una de estas una de las particiones actuará como test, esto es que todos los datos que pertenezcan a esa partición serán del tipo *test*, mientras que todos los demás serán del tipo *entrenamiento*.

- Atributo *prueba*: datos relacionados con las pruebas que se hagan con ese dato, ver *apartado 3.3.3*.

Además de estos atributos esta clase estará dotada de una serie de métodos para poder trabajar con estos datos. Estos métodos servirán para realizar las pruebas con estos datos, es decir, conocer cual es el valor que nos dará al hacer los cálculos con las matrices y prototipos dados por los objetos del tipo *Clase* del problema actual. Estas funcionalidades tendrán como objetivo, rellenar los valores de la clase *Prueba* asociada a cada uno de estos datos, es decir, rellenar la matriz de distancias calculadas, así como obtener la clase esperada como solución según las distancias guardadas y almacenar si estos cálculos son un acierto o un fallo.

La clase está formada de la siguiente manera:

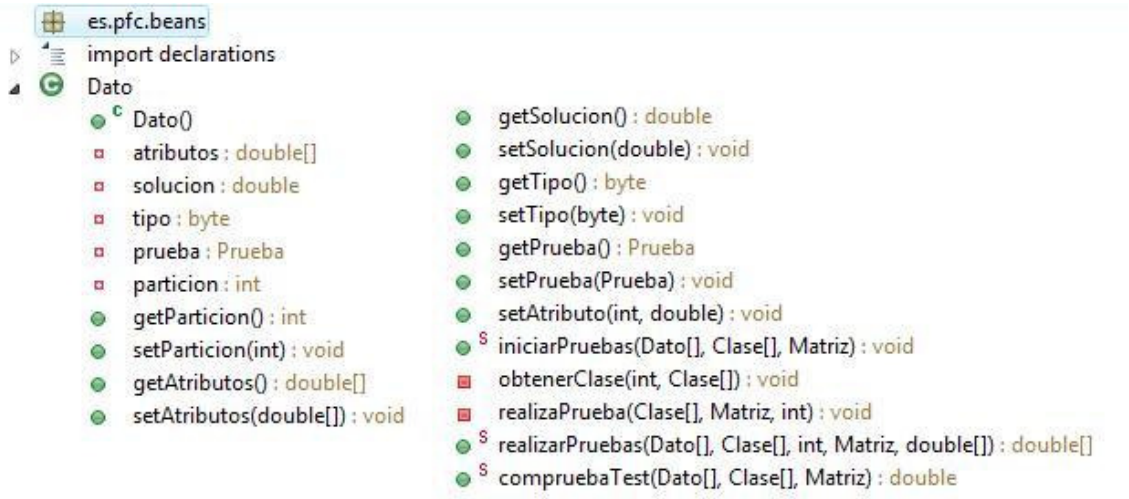

Ilustración 18: Clase Dato.

3.3.5.- Clase "Matriz"

Esta clase nos dará la funcionalidad necesaria para trabajar con las matrices, la estructura vendrá dada por dos números que nos dará el número de filas y columnas y un array con las dimensiones dadas por esos dos números. Una vez construida una matriz con esa estructura, tendrá métodos para el cálculo de suma, resta, producto, matriz traspuesta y determinante de matrices.

Los atributos, constructores y métodos de la clase son los siguientes:

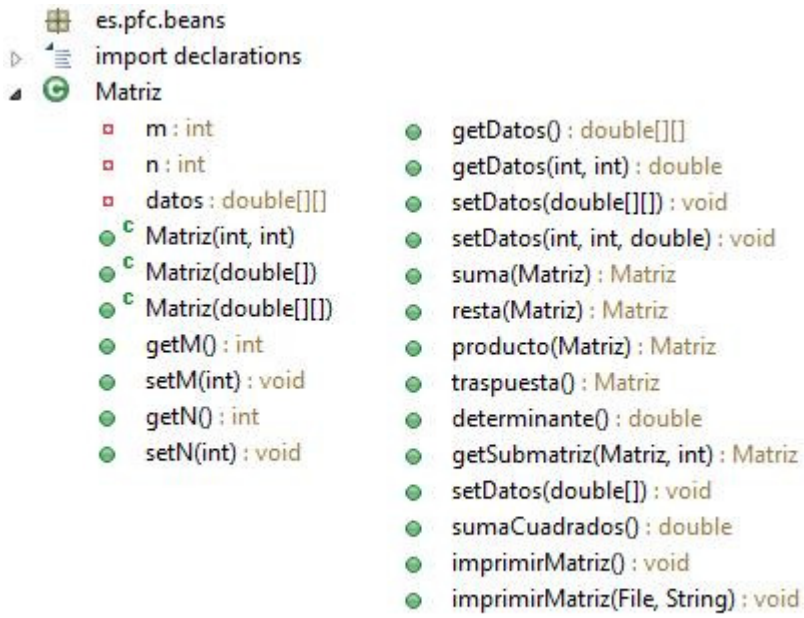

Ilustración 19: Clase Matriz.

3.3.6.- Clase "MatrizLS"

Será usada para las matrices que se utilizan en la búsqueda local, que son las matrices que irán asociadas a las clases como matriz padre y matriz hija, es decir, la matriz para el cálculo de la distancia que se tendrá de la generación actual y la matriz mutada de la generación actual. Para ello, heredará de la clase *Matriz* (ver *apartado 3.3.5*) los atributos y métodos pero tendrá la particularidad de que será una matriz.

La clase esta formada por los siguientes elementos junto con los que la proporciona la clase *Matriz*:

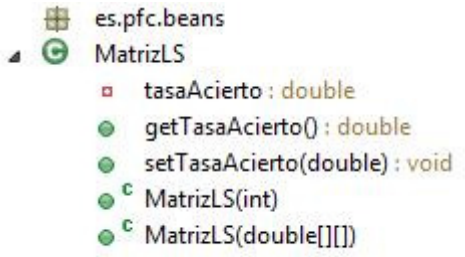

Ilustración 20: Clase MatrizLS.

3.3.7.- Clase "Clase"

Cada una de las soluciones posibles que nos pueda dar el sistema tendrá una serie de datos asociados, por ello, esta clase tendrá los siguientes atributos:

- Atributo *id*: las clases vendrán identificadas por este atributo, este será una de las posibles soluciones que tendrá la ejecución actual.

- Atributo *centroides*: será un array de objetos de la clase *Matriz* que guardará los diferentes prototipos que se vayan obteniendo en cada una de las generaciones. La particularidad de las matrices correspondientes es que son vectores, es decir, serán matrices de una fila. Estos centroides serán los prototipos que irán mutando en cada una de las generaciones.

- Atributo *sumaCoordenadas*: para el cálculo del centroide inicial se realizará la suma de cada una de las coordenadas de todos los valores pertenecientes a esta clase.

- Atributo *numDatos*: también se guardará el número de datos que pertenecen a esta clase, con este atributo y el anterior podremos calcular un prototipo que será el usado durante la primera generación.

- Atributo m*atrizPadre*: será la mejor matriz encontrada hasta el momento actual, la cual en cada nueva generación será mutada para dar la matriz hija.

- Atributo m*atrizHija*: esta matriz será la resultante de la mutación del anterior atributo.

- Atributo *mejorCentroide*: el mejor prototipo encontrado hasta el momento será guardado para que pueda ser usado por la siguiente generación.

- Atributo *mejorFitness*: la mejor tasa de acierto también es guardada para poder comprobar posteriormente si las nuevas pruebas realizadas nos dan un resultado mejor y podemos pasar a usar otra matriz y otro prototipo.

Para la generación de los objetos de este tipo, necesitaremos los datos que nos da el fichero de configuración, primero necesitamos las diferentes soluciones que nos puede dar el problema, con cada una de ellas crearemos un objeto de esta clase y con el número de dimensiones y prototipos, daremos formato a los demás atributos, las matrices por defecto serán creadas como matrices diagonales pero podrán ser creadas como matrices aleatorias mediante un simple cambio de llamada al método de creación en el constructor de la clase.

Otros métodos que nos da esta clase son el cálculo de los centroides iniciales que se calculará con los atributos *sumaCoordenadas* y *numDatos* que van siendo calculados gracias a un método que suma cada uno de los datos que se van encontrando pertenecientes a esta clase e incrementado el valor del número de datos de la clase. Como se ha dicho anteriormente, tenemos la opción de crear matrices diagonales o aleatorias para usarlas en la primera generación para el cálculo de la distancia, las funciones que nos devuelven estos dos tipos de matrices también se encuentran en la propia clase.

La clase está formada por los siguientes elementos:

Ilustración 21: Clase Clase.

De todos estos, a continuación se van a explicar dos de los métodos más importantes, esto son:

3.3.7.1.- Método calculaDistancia

Este método nos la distancia que existe de un punto dado a uno de los prototipos de la clase. Para ello, se le pasará como atributos el número de prototipo al que se quiere calcular la distancia y, por otro, el punto que representa el dato al cual se quiere saber la distancia.

El cálculo de la distancia se hará de la siguiente manera: necesitamos el punto que representa el dato, lo llamaremos como P y el centroide C que será el prototipo dado por

el atributo pasado que se corresponde al índice del listado de prototipos, además necesitaremos la matriz hija *M*. Con estos datos, calcularemos la distancia de la siguiente manera:

$$d(P,C) = sumarCuadrados((P-C)M)$$

Ecuación 8: Función cálculo distancia simplificada.

Este cálculo, es una manera simplificada de realizar el cálculo de la distancia de la siguiente manera:

$$d(P,C) = \sqrt{(P-C)^T M^T M (P-C)}$$

Ecuación 9: Función cálculo distancia.

3.3.7.2.- Método evolucionar

Se necesitará que para cada nueva generación se recoja el mejor prototipo así como la mejor matriz y sean evolucionados para ir buscando la mejor solución para el programa. El diagrama de flujo de este método se puede resumir de la siguiente manera (el diagrama sólo se corresponde con la evolución de la matriz, luego se realiza el mismo procedimiento con cada uno de los prototipos):

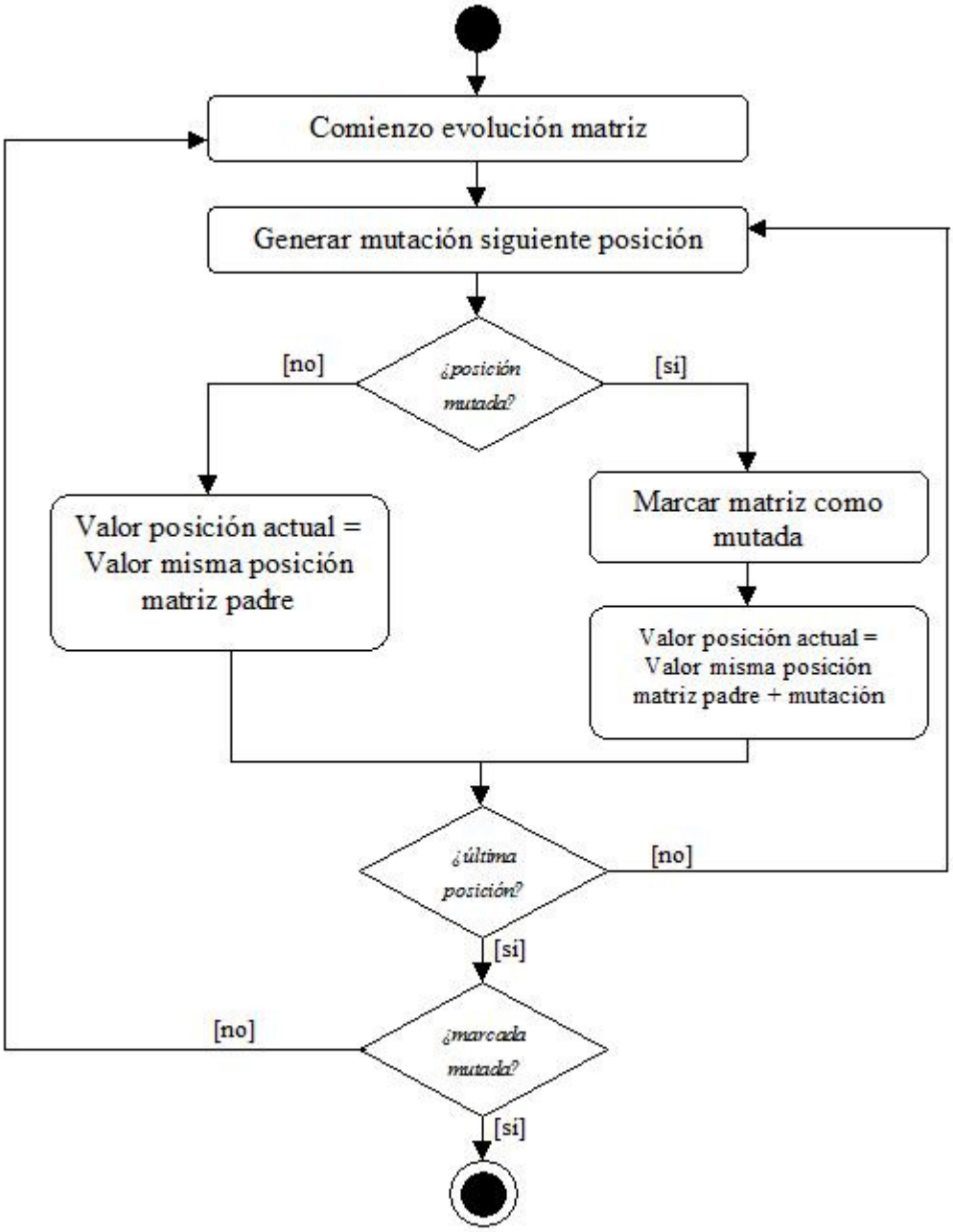

Ilustración 22: Diagrama de actividad del método evolucionar

A continuación, se explica como funciona la mutación o no de cada una de las posiciones, es decir, el proceso del diagrama anterior llamado *Generar mutación siguiente posición*:

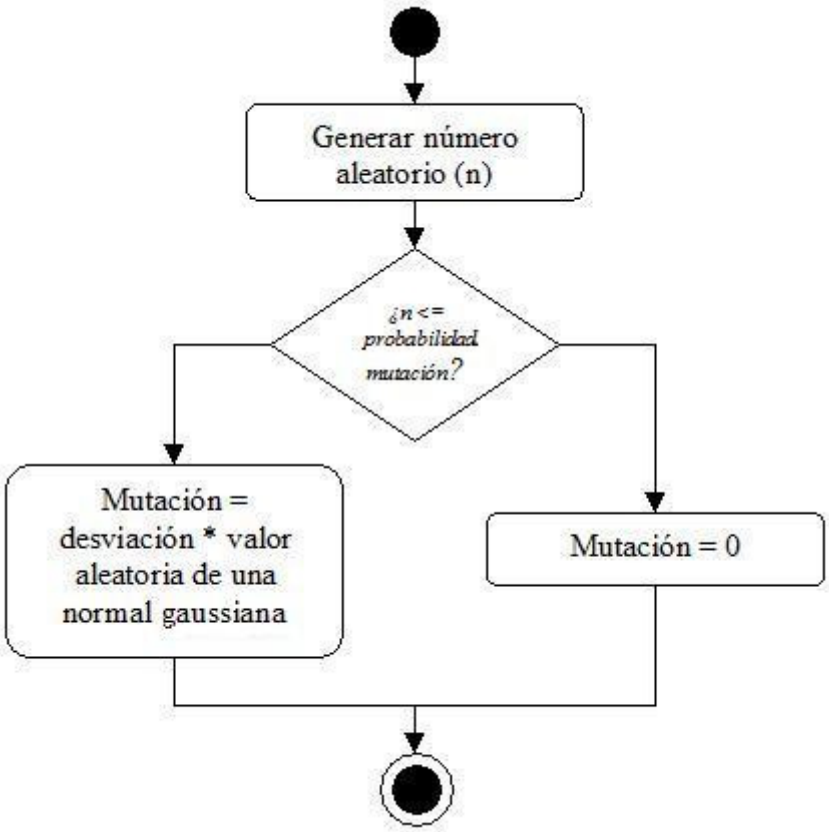

Ilustración 23: Diagrama de actividad de la mutación

3.3.8.- Clase "Metodos"

Por último, la clase *"Metodos"* tendrá como su propio nombre indica una serie de métodos necesarios para encontrar la solución al problema. Estos son:

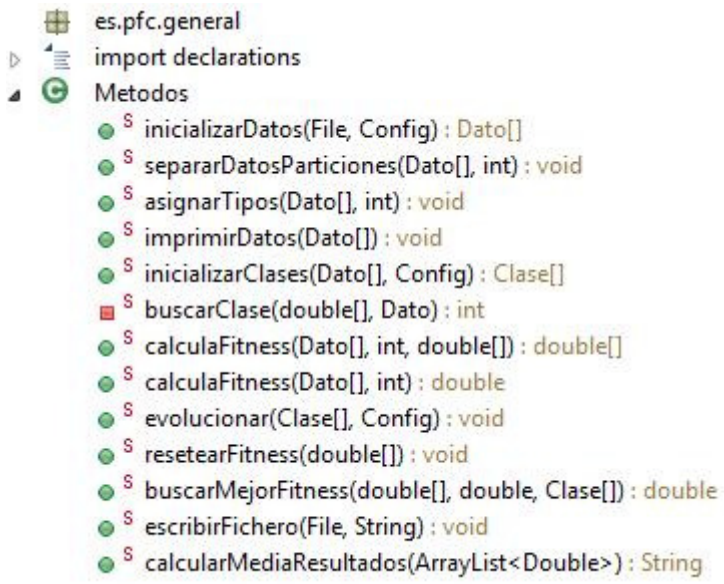

Ilustración 24: Clase Metodos.

Debido a la importancia de estos métodos serán explicados más detalladamente:

3.3.8.1.- Método inicializarDatos

A este método se le pasa el fichero de datos y los datos de configuración, recorre cada una de las líneas del fichero de datos que son pasadas a objetos del tipo *Dato*, separando los atributo de la solución e inicializando el objeto de la clase *Prueba* que cada uno de los datos tiene asociada.

3.3.8.2.- Método separarDatosParticiones

Es necesario separar los datos en diferentes particiones para realizar la validación cruzada, para ello tomamos el número total de datos así como el número de particiones, de la división de estos redondeada obtenemos el número de datos que habrá por partición.

Con este valor, se recorre la lista de datos y a cada se les asigna la partición a la que pertenecen, cuando se llega al número obtenido de datos por partición, se reinicia el contador y los siguientes datos serán asignados a la siguiente partición, así sucesivamente.

El funcionamiento del método de forma resumida es el siguiente:

```
Procedure separarDatosParticiones
        numDatos = length(datos)
        datosPorParticion = rint(numDatos / particiones)
        particionActual = 1
        while contador < numDatos
                datos[contador].particion = particionActual
                if (contador % datosPorParticion == 0)  && no ultimaParticion then
                        particionActual++
                fin if fin = true
        end while
end Procedure
```

3.3.8.3.- Método asignarTipos

Recorre la lista de datos y les asigna el tipo *"train"* a todos salvo a los que pertenezcan a la partición de test.

```
Procedure asignarTipos
        for i = 0; i < length(datos); i++
                if  datos[i].particion == particionTest then
                        datos[i].tipo = TEST
                else
                        datos[i].tipo = TRAIN
        end for
end Procedure
```

3.3.8.4.- Método inicializarClases

Con las clases que nos da el fichero de configuración llamamos al constructor de las clases y creamos estas, con las matrices y prototipos de las dimensiones requeridas.

Una vez construidas todas las clases, recorremos los datos de entrenamiento y buscamos la clase a la que pertenece (ver *apartado 3.3.8.5*). A la vez que se van asignando los datos a cada una de las clases, se va haciendo el sumatorio de los atributos de los datos de cada una de las clases y contabilizamos los datos que pertenecen a cada una de ellas, con estos datos calcularemos los prototipos iniciales de cada una de las clases.

```
Procedure inicializarClases
        for i = 0; i < numeroClases; i++
                clases[i] = new Clase(numeroDimensiones, numeroPrototipos)
        end for
        for j = 0; j < length(datos); j++
                c = buscarClase(datos[j])
                sumatorioDatos(c)+= datos[j]
        end for
        for k = 0; k < numeroClases; k++
                calcularPrototipoInicial(sumatorioDatos(k))
                calcularDesviacionesAtributos(k)
        end for
end Procedure
```

3.3.8.5.- Método buscarClase

Busca la clase a la pertenece un dato dado, para ello comprueba con que identificador de clase coincide la solución que fue leída en el fichero de datos.

Este método es llamado desde el anterior como se puede ver en el pseudocódigo, el funcionamiento para buscar la clase de un dato es el siguiente:

```
Procedure buscarClase
        indiceClase = 0
        while no encontrado do
                if clases[indiceClase] == dato.solucion then
                        encontrado = true
                if no encontrado then
                        indiceClase++
        end while
        return indiceClase
end Procedure
```

3.3.8.6.- Métodos calcularFitness

Existen dos métodos con este nombre, uno es usado para el cálculo del fitness mientras estamos en la fase de entrenamiento, mientras que el otro se usa para el cálculo final con los datos de test.

El funcionamiento de estos es bastante similar, ambos reciben el array de datos como parámetro, además de un número relacionado con los prototipos, pero mientras que en el caso de los datos de entrenamiento este se refiere al número total de prototipos con los que trabaja el programa en esa ejecución, en el caso del método referente a los datos de test, este número hace referencia al índice del que ha sido el mejor prototipo. Con los datos de entrenamiento, también se pasa una matriz de datos de tipo *"double"* en la que se guardarán los fitness referentes a cada uno de los prototipos, para los datos de test tan sólo se realiza el cálculo con un prototipo por lo que devolverá un número con el fitness resultado de la ejecución.

El funcionamiento consiste en recorrer todos los datos (los de entrenamiento en un caso y los de test en el otro) comprobando en las variable de tipo *Prueba*, que han sido rellenadas en el método de la clase *Dato* (ver métodos del *apartado 3.3.4*) que llama a su vez a este método, si han sido marcados como acierto o no en la prueba con el prototipo correspondiente. En este proceso se realiza el conteo del número total de aciertos, así como del número total de datos recorridos, con estos dos valores obtenemos el tanto por ciento de acierto que es el dato buscado.

En el caso de los datos de entrenamiento, la funcionalidad descrita en el párrafo anterior estará incluida dentro de un bucle que recorrerá todos los prototipos guardando cada tanto por ciento en la posición correspondiente del array de salida.

El funcionamiento del procedimiento será el siguiente:

```
Procedure calcularFitness
        for i=0; i < numPrototipos; i++
                contTrain = 0
                for j=0; j < numDatos; j++
                        if datos[j].tipo == TRAIN then
                                contTrain++
                                if esAcierto(datos[j], prototipo[i]) then
                                        fitness[i]+= 1
                                end if
                        end if
                end for
                fitness[i] = fitness[i] * 100 / contTrain
        end for
end Procedure
```

3.3.8.7.- Método evolucionar

Consiste en un bucle que irá llamando al método *evolucionar* de la clase *Clase* (ver *apartado 3.3.7.2*), recorriendo todas las clases del problema para que sus matrices y prototipos sean mutados.

3.3.8.8.- Método buscarMejorFitness

Busca si durante el ciclo actual se ha encontrado algún fitness mejor con alguno de los prototipos y nos devolverá el mejor fitness encontrado hasta el momento.

Para ello recogerá como parámetros el array de fitness con los tantos por ciento de acierto de cada uno de los prototipos (ver *apartado 3.3.8.6*), el mejor fitness encontrado hasta el momento, así como el array de clases.

El método recorre el array con los fitness calculados y compara cada uno de ellos con el que es el mejor hasta el momento, en caso de que alguno supere a este pasará a ser el

mejor y por tanto debemos recorrer las clases para que las matrices hijas de estas pasen a ser las matrices padres con lo que en el próximo ciclo estas serán las que muten y por la misma razón, los prototipos correspondientes al índice actual pasarán también a ser los mejores prototipos de cada una de sus clases. En caso de que ninguna posición del array supere el mejor tanto por ciento, las matrices padre y los mejores prototipos de todas las clases serán los mismos en el próximo cálculo.

```
Procedure buscarMejorFitness
        mejorFitness = fitnessDelPadre
        for i=0; i < numPrototipos; i++
                if fitnessPruebas[i] > mejorFitness then
                        mejorFitness = fitnessPruebas[i]
                        for j=0; j < numClases; j++
                                clases[j].matrizPadre = clases[j].matrizHija
                                clases[j].mejorPrototipo = clases[j].prototipo[i]
                        end for
                end if
        end for
end Procedure
```

3.3.8.9.- Método calcularMediaResultados

Calcula la media de los resultados obtenidos con cada una de las particiones de la validación cruzada (que han sido anteriormente guardados en una variable de tipo "*ArrayList*") para dar un resultado final al problema.

Capítulo 4

Experimentación

4.1.- Dominios utilizados en la experimentación

El presente capítulo mostrará los resultados obtenidos por la aplicación con los datos correspondientes a una serie de dominios. Estos resultados nos servirán para llegar a conclusiones mediante la comparación entre ellos.

El sistema desarrollado devolverá unos resultados que serán comparados con el método de clasificación inicial. Este clasificador "base" consistirá en el uso de la matriz identidad y prototipos iniciales correspondientes al centro de masas de los individuos de cada una de las clases resultado que tenga el dominio. En otras palabras, la experimentación parte con la distancia euclídea al valor medio de los datos de cada una de las clases existentes y los resultados devueltos en este primer cálculo serán los que se compararán con los resultados finales obtenidos.

El resultado final que nos devolverá el sistema será alcanzado mediante la evolución de la matriz identidad a lo largo de un número dado de generaciones. Así como se irá mutando la posición de los prototipos con respecto a los que es calculada la proximidad de los datos. Los experimentos serán realizados con dos configuraciones distintas para todos los dominios, la primera experimentación se llevará a cabo con tres prototipos, mientras que para la segunda se hará uso de seis prototipos.

A continuación, pasamos a enumerar los dominios que se han utilizado durante esta fase de experimentación [Frank y Asuncion, 2010]:

- Ripley
- Blood
- Bupa
- Ionosphere
- Iris
- Wine

Como se dijo anteriormente, para cada uno de los dominios anteriores se llevarán a cabo dos experimentaciones diferentes. En una de ellas se utilizarán tres prototipos y en la otra seis.

Así mismo, para cada dominio también serán descritos todos los datos que son proporcionados a la aplicación en el fichero de configuración y que influirán en la solución final. Además se realizará una breve descripción de los datos pertenecientes a dichos dominios.

4.2.- Ripley

4.2.1.- Descripción del dominio

El dominio Ripley [Ripley, 1996] está compuesto por dos coordenadas con valores reales y existen dos resultados posibles: 0 y 1. Las instancias del dominio han sido generadas artificialmente. Cada una de las dos clases citadas anteriormente corresponde a una distribución bimodal que es una composición balanceada de dos distribuciones normales. Las matrices de covarianza son idénticas en todas las distribuciones, mientras que los centros son diferentes. Existe un gran solapamiento entre las dos clases, lo que hace interesante el estudio de este dominio.

Información acerca de los datos utilizados:

- Instancias: 1250
- Atributos: 2 (valores continuos)
    - Coordenada X
    - Coordenada Y
- Clases: 2
    - 0
    - 1

4.2.2.- Configuración

Los datos de configuración que recibirá el programa para las pruebas son los siguientes:

- Número de particiones para la validación cruzada: 10
- Número de dimensiones: 2
- Número de generaciones: 3000
- Probabilidad de mutación: 0’25, resultante de la división de 1 entre el número de dimensiones al cuadrado, para que en cada generación la mutación de la matriz sea realizada en una de sus posiciones (aproximadamente).
- Valor de la mutación: 0’1, este peso será multiplicado por una distribución gaussiana aleatoria, posteriormente también se ponderará por la desviación del atributo (ver siguiente dato de la configuración).
- Ponderación de las mutaciones: cada valor de la mutación obtenido mediante el dato anterior será ponderado por la desviación del atributo que se está mutando.
- Clases: 0 y 1
- Número de prototipos: 3 y 6 en cada una de las respectivas ejecuciones.

4.2.3.- Resultados

La siguiente tabla nos muestra un resumen de los datos obtenidos. Las columnas 2 y 3 corresponderán a los resultados de entrenamiento y test obtenidos con el clasificador inicial. Las columnas 4 y 5 muestran los resultados obtenidos por el sistema usando tres prototipos. Y, por último, las columnas 6 y 7 muestran las tasas de acierto con el uso de seis prototipos. En todas las columnas se muestran los resultados en cada una de las particiones de la validación cruzada, así como finalmente la media de todas ellas.

	Distancia Euclídea		3 Prototipos		6 Prototipos	
Partición	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
1	74.84	79.2	88.98	91.2	89.87	89.6
2	75.56	68.8	90.13	87.2	90.05	86.4
3	74.49	80.8	88.45	86.4	89.87	90.4
4	76.27	76.8	90.31	83.2	90.49	84.8
5	75.38	70.4	89.78	68.8	90.4	87.2
6	74.93	76.8	89.42	88.0	88.89	84.8
7	75.2	76.0	89.96	91.2	89.78	91.2
8	75.11	78.4	89.42	90.4	89.87	89.6
9	75.47	71.2	89.96	92.0	89.51	87.2
10	75.47	74.4	90.04	84.0	90.4	74.4
Media	**75.27**	**75.28**	**89.65**	**86.24**	**89.91**	**86.56**

Tabla 1: Resultados experimentación Ripley.

En la anterior tabla podemos comparar las tasas de acierto tanto de entrenamiento como de test utilizando la distancia euclídea, que será el punto de partida, y dos ejecuciones de la aplicación con tres y seis prototipos respectivamente.

Se puede comprobar que la ejecución de la aplicación consigue que los resultados mejoren de manera bastante apreciable cuando nos fijamos en los datos de entrenamiento y que esta mejora sigue siendo apreciable cuando realizamos la fase de test.

En comparación a la utilización de más o menos prototipos, se ve una ligera mejora utilizando el doble de prototipos en el inicio, cada camino recorrido para buscar la solución con un prototipo parece que arroja unos resultados parecidos, pero la opción de seguir más caminos hace que se pueda encontrar uno algo mejor.

Para poder ilustrar mejor la evolución que nos lleva a dar con estos resultados, mostraremos gráficamente la evolución de una de las particiones, en este caso será la partición 8 en su ejecución con 6 prototipos. Esta evolución se corresponde con la tasa de acierto que obtenemos en cada generación con el uso del mejor individuo. En los casos en los que el fitness no crece, se corresponden con generaciones en las que ninguno de los descendientes consigue ser mejor que el individuo progenitor:

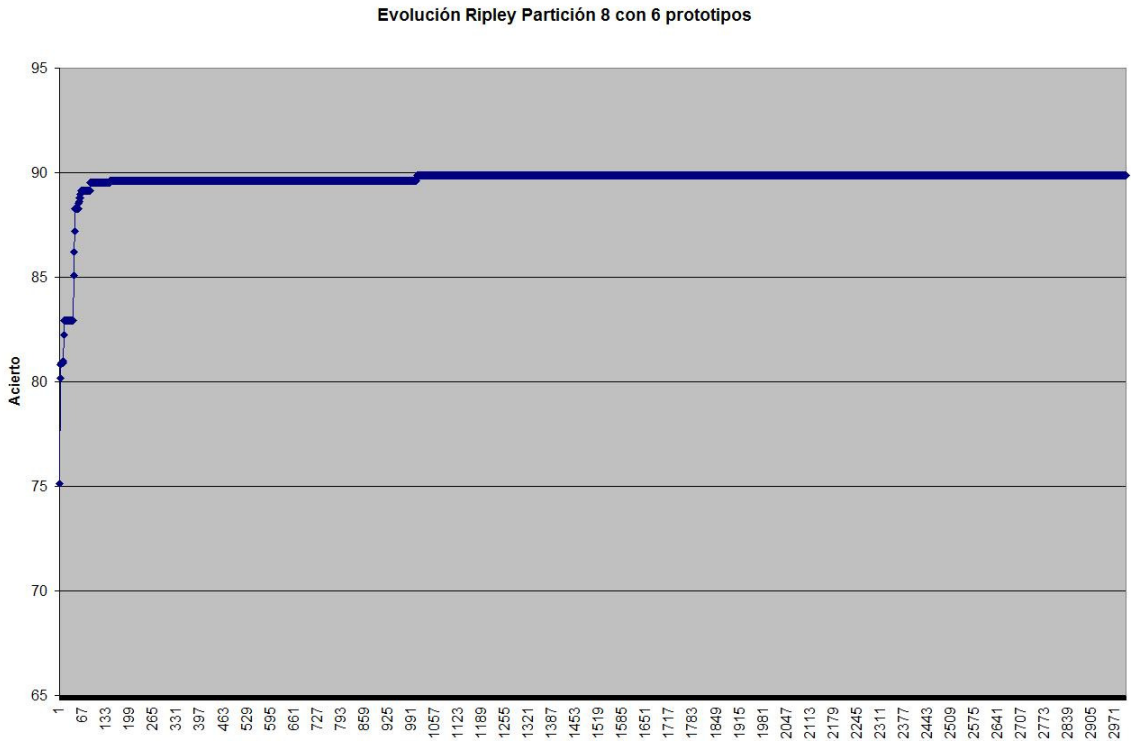

Ilustración 25: Evolución partición 8 de la experimentación con Ripley

Como podemos comprobar en las primeras generaciones el aumento del fitness es bastante rápido, sin embargo, la mejora en la tasa de acierto a partir de, aproximadamente, el ciclo 100 se convierte en un resultado difícil de conseguir, debido a esto no existe gran variación entre los resultados de las diferentes particiones tanto con tres como con 6 prototipos.

A continuación, comparamos los resultados obtenidos mediante el algoritmo de los k-vecinos más cercanos con k = 1, estos resultados han sido obtenidos mediante la ejecución de otra aplicación y recogidos aquí para la comparación de los resultados finales de ambas formas de búsqueda:

3 Prototipos	6 Prototipos	K-Vecinos (K = 1)
86’24	86’56	88’5760

Tabla 2: Comparación con k-vecinos Ripley.

Podemos comprobar que el algoritmo K-vecinos da unos resultados mejores que con el algoritmo utilizado en este caso, la mejora es significativa aunque el sistema que nos ocupa tiene un aumento de la tasa de acierto bastante significativo con respecto al clasificador base.

4.3.- Blood

4.3.1.- Descripción del dominio

Este dominio de datos [Yeh, Yang y Ting, 2008] recoge los datos del día 3 de octubre de 2008 de 748 donantes elegidos aleatoriamente en un autobús de donaciones de sangre enviado a una universidad de Hsin-Chu City (Taiwan). En el dominio se recogen los datos de las donaciones que las personas estudiadas realizaron en el pasado (estos atributos serán enumerados en detalle más adelante). El resultado final será si un determinado individuo realizó una donación el día 7 de marzo de 2007.

El dominio fue creado para demostrar el modelo de marketing RFMTC, que es una versión ampliada del modelo RFM, que se basa en los parámetros dados por la siglas de su nombre: Recency (compra reciente), Frecuency (frecuencia), Monetary (monetario), Time (tiempo desde la primera compra) y Churn (cancelación de clientes). Esta definición está realizada en términos financieros, en la descripción de los atributos del dominio veremos su equivalente en este caso.

Información acerca de los datos utilizados:

- Instancias: 748
- Atributos: 4 (valores discretos)
  - R: meses desde la última donación.
  - F: número de donaciones totales.
  - M: total de sangre donada en centímetros cúbicos.
  - T: meses desde la primera donación.
- Clases: 2
  - 0: no donó en marzo de 2007.
  - 1: sí donó en marzo de 2007.

4.3.2.- Configuración

Los datos de configuración que recibirá el programa para las pruebas son los siguientes:

- Número de particiones para la validación cruzada: 10
- Número de dimensiones: 4
- Número de generaciones: 3000
- Probabilidad de mutación: 0’0625, resultante de la división de 1 entre el número de dimensiones al cuadrado, para que en cada generación la mutación de la matriz sea realizada en una de sus posiciones (aproximadamente).
- Valor de la mutación: 0’1, este peso será multiplicado por una distribución gaussiana aleatoria, posteriormente también se ponderará por la desviación del atributo (ver siguiente dato de la configuración).
- Ponderación de las mutaciones: cada valor de la mutación obtenido mediante el dato anterior será ponderado por la desviación del atributo que se está mutando.
- Clases: 0 y 1
- Número de prototipos: 3 y 6 en cada una de las respectivas ejecuciones.

4.3.3.- Resultados

La siguiente tabla nos muestra un resumen de los datos obtenidos, mostrando los resultados por cada partición así como la media final de estos. Comparando los resultados obtenidos en la primera generación (distancia euclídea), los resultados de este clasificador inicial se corresponden con las columnas 2 y 3 de la tabla. Con los resultados obtenidos tras el paso de todas las generaciones indicadas en el fichero de configuración ejecutadas con tres y seis prototipos respectivamente. Así tenemos que los datos recogidos en las columnas 4 y 5 se corresponden con la experimentación con tres prototipos, mientras que las dos últimas columnas nos muestran los porcentajes obtenidos para seis prototipos:

	Distancia Euclídea		3 Prototipos		6 Prototipos	
Partición	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
1	68.1	62.16	77.15	68.92	77.15	71.62
2	67.36	68.92	76.41	82.43	76.7	79.73
3	67.36	68.92	77.0	74.32	77.0	77.02
4	67.80	64.86	76.85	78.38	77.3	77.03
5	66.47	77.03	76.71	79.73	76.41	81.08
6	67.06	71.62	76.71	75.68	76.71	78.38
7	68.1	62.16	77.6	68.92	77.45	72.97
8	67.95	63.51	76.71	77.03	76.56	75.68
9	67.51	67.57	77.45	70.27	77.45	70.27
10	67.42	68.29	76.43	79.29	76.58	79.27
Media	**67.51**	**67.50**	**76.90**	**75.50**	**76.93**	**76.31**

Tabla 3: Resultados experimentación Blood.

Como vemos la ejecución consigue que los resultados mejoren con respecto a la distancia euclídea original.

Al realizar el test podemos comprobar que el problema mejora con seis prototipos, sin embargo, al comprobar los resultados con los datos de entrenamiento vemos que esta mejora es casi inexistente. A continuación se pasará a mostrar de manera gráfica las diferencias en las tasas de acierto con los datos de test.

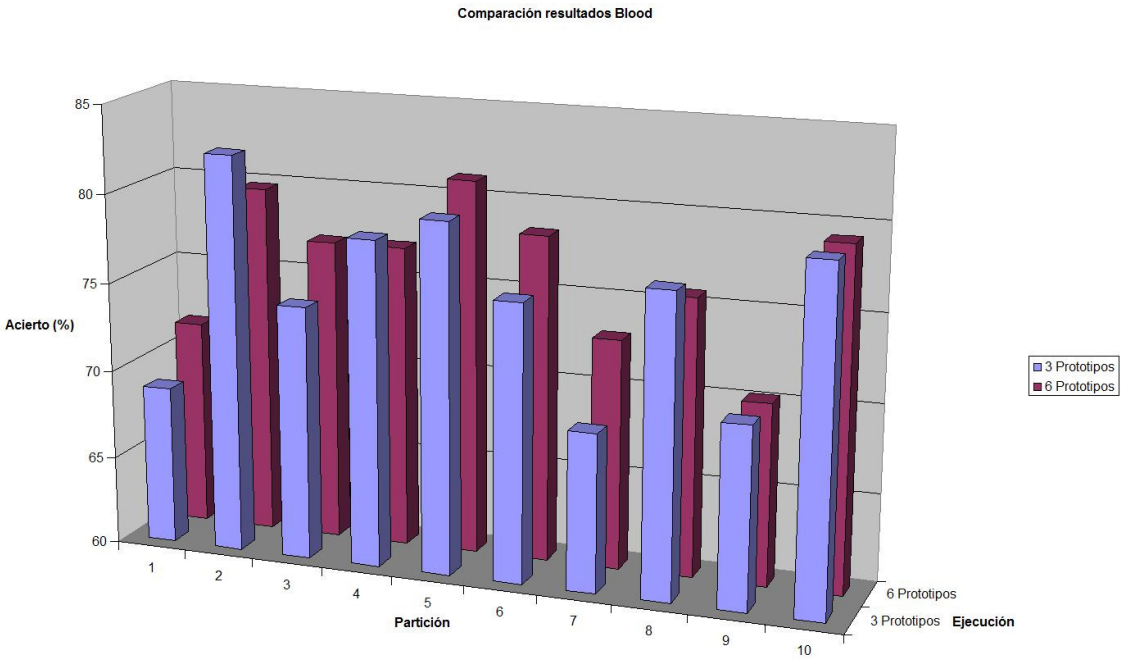

Ilustración 26: Comparación de la tasa de aciertos en test por partición con cada ejecución para el dominio Blood.

El gráfico nos muestra que las pruebas realizadas con seis prototipos tienen mejores resultados en más particiones, aunque existen algunos casos en los que la ejecución con menos prototipos tiene mejores resultados. Pero en general no existen grandes diferencias entre ambos.

A continuación, comparamos los resultados obtenidos mediante el algoritmo de los k-vecinos más cercanos con k = 1, estos resultados han sido recogidos gracias a una aplicación diferente a la que tratamos aquí pero que nos sirven para comparar nuestros resultados con otras posibles soluciones:

3 Prototipos	6 Prototipos	K-Vecinos (K = 1)
75’4944	76’3052	68’4091

Tabla 4: Comparación con k-vecinos Blood.

Podemos comprobar que para la búsqueda de un algoritmo optimo de cálculo de resultados para este dominio de datos, la solución utilizada en este caso mejora sustancialmente los resultados dados por las pruebas con k-vecinos.

4.4.- Bupa

4.4.1.- Descripción del dominio

Este dominio recoge datos sobre afecciones del hígado recogidos por BUPA Medical Research Ltd. Cada una de las instancias corresponden a un adulto varón, cinco de los atributos son datos recogidos mediante los análisis de sangre, el otro corresponde a la cantidad de bebida que ingiere el individuo. Los individuos serán separados en dos tipos, la aplicación calculará a que grupo pertenece el sujeto.

Información acerca de los datos utilizados:

- Instancias: 345
- Atributos: 6 (valores discretos)
    - Volumen Corpuscular Medio (mcv: Mean Corpuscular Volume): $VCM = (Hct / RBC) * 10$, donde Hct es el hematocrito, en porcentaje y RBC es el conteo de eritrocitos, expresado en millones de células por microlitro.
    - Fosfatasa alcalina (alkphos: alkaline phosphatase): enzima hidrolasa encargada de eliminar grupos de fosfatos de ciertos tipos de moléculas.
    - Alanina aminotransferasa (sgpt: alamine aminotransferase): enzima aminotransferasa de gran concentración en el hígado que es liberada en gran volumen a la sangre cuando existe una lesión del organo en el que se encuentra.
    - Aspartato aminotransferasa (sgot: aspartate aminotransferase): enzima aminotransferasa alojada en el tejido muscular hepático que también sirve como indicador de lesión.
    - Gamma-glutamil transpeptidasa (gammagt: gamma-glutamyl transpeptidase: enzima hepática que en niveles elevados puede indicar una anormalidad en el hígado.

    - Cantidad diaria de bebida alcohólica ingerida cuantificada por el equivalente en medias pintas.
- Clases: 2
    - 1: Grupo 1
    - 2: Grupo 2

4.4.2.- Configuración

Los datos de configuración que recibirá el programa para las pruebas son los siguientes:

- Número de particiones para la validación cruzada: 10
- Número de dimensiones: 6
- Número de generaciones: 3000
- Probabilidad de mutación: 0’02778, resultante de la división de 1 entre el número de dimensiones al cuadrado, para que en cada generación la mutación de la matriz sea realizada en una de sus posiciones (aproximadamente).
- Valor de la mutación: 0’1, este peso será multiplicado por una distribución gaussiana aleatoria, posteriormente también se ponderará por la desviación del atributo (ver siguiente dato de la configuración).
- Ponderación de las mutaciones: cada valor de la mutación obtenido mediante el dato anterior será ponderado por la desviación del atributo que se está mutando.
- Clases: 1 y 2
- Número de prototipos: 3 y 6 en cada una de las respectivas ejecuciones.

4.4.3.- Resultados

La siguiente tabla nos muestra un resumen de los datos obtenidos, mostrando los resultados por cada partición así como la media final de estos, en las diferentes columnas podemos distinguir los resultados de entrenamiento y test tanto para la distancia euclídea, es decir, punto de partida del problema (columnas 2 y 3), como los resultados obtenidos tras la ejecución con tres prototipos en primera instancia (columnas 4 y 5) y con seis en segundo lugar (columnas 6 y 7):

	Distancia Euclídea		3 Prototipos		6 Prototipos	
Partición	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
1	53.7	79.41	70.74	52.94	74.28	50.0
2	57.56	32.35	67.2	52.94	71.7	70.59
3	55.31	61.76	72.67	67.64	69.45	64.70
4	56.91	44.11	70.1	64.71	69.13	67.64
5	57.23	52.94	72.99	70.59	73.95	70.59
6	55.95	70.59	71.38	67.64	73.31	70.59
7	54.66	55.88	67.84	64.71	71.7	82.35
8	56.27	55.88	69.13	67.64	71.7	61.76
9	56.27	50.0	68.49	82.35	69.45	73.53
10	55.88	56.41	69.28	51.28	72.22	51.28
Media	**55.97**	**55.93**	**69.98**	**64.24**	**71.69**	**66.3**

Tabla 5: Resultados experimentación Bupa.

Para este dominio, la búsqueda de una solución mediante el uso de la distancia euclídea nos devuelve unos resultados bastante pobres, utilizando el uso de la mutación de la matriz y los prototipos no conseguimos una tasa de acierto excesivamente elevada, pero sí que es una mejora muy apreciable con respecto a la otra solución. Además de comprobar que la utilización de 6 prototipos nos da unos mejores resultados.

Otra cosa a tener en cuenta es que aunque los resultados durante las pruebas son más o menos homogéneos, tenemos que cuando se realiza el test conseguimos unos resultados bastante dispares con los que una tasa elevada de acierto en el entrenamiento no tiene porque arrojar un fitness elevado en el test, así como al revés, por tanto, los resultados en el entrenamiento para este dominio no son muy fiables a la hora de conocer cuales van a ser los resultados finales. La razón de estos porcentajes puede ser que la población es más reducida que las de otros dominios utilizados con anterioridad, por ello las particiones de test tendrán 34 o 35 individuos en cada caso y los fallos afectarán en un porcentaje mayor en los resultados finales. Añadido al hecho de lo significativo en el fitness que puede resultar un fallo, las tasas de acierto en la parte de entrenamiento son algo menores que en los otros dominios, con esto tenemos, que además de que cada fallo afecta más al resultado, la tasa de acierto a priori va a ser menor al realizar el test. Resumiendo, este dominio tiene más posibilidades de fallo y cada uno de estos fallos afectará más significativamente al resultado de la partición, por ello tenemos resultados más dispares.

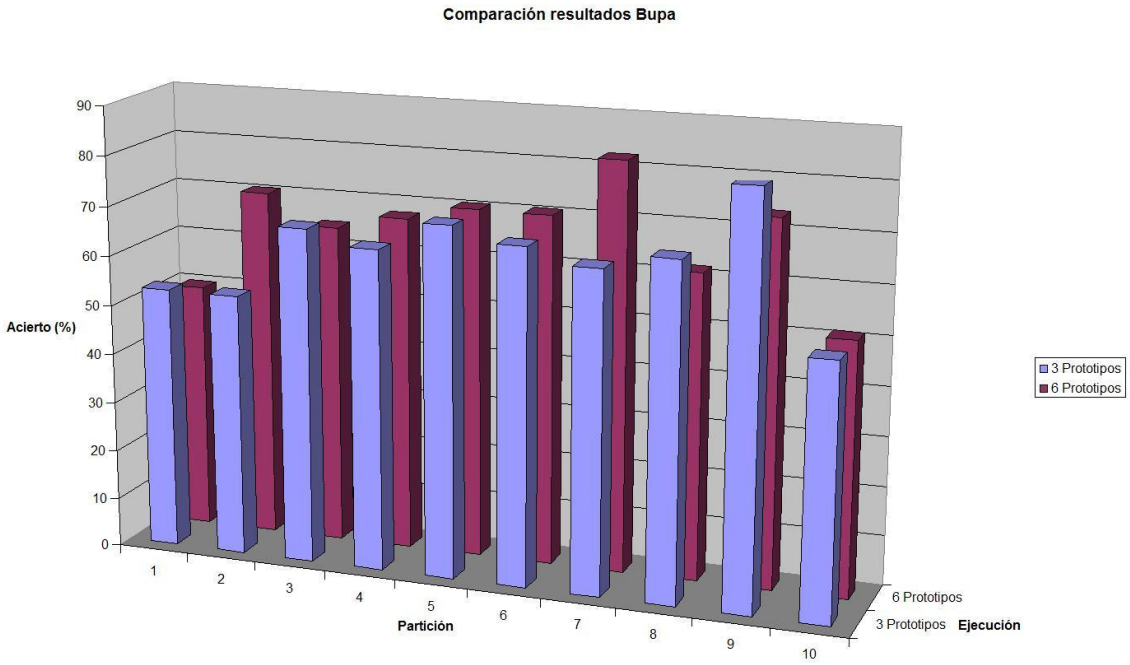

Ilustración 27: Comparación de la tasa de aciertos en test por partición con cada ejecución para el dominio Bupa.

La anterior gráfica nos ayuda mejor a comprobar las diferencias de porcentajes de acierto entre las diferentes particiones para ambas ejecuciones. También podemos comprobar unos resultados bastante similares en las dos pruebas, salvo en un par donde los resultados con seis prototipos son mayores.

A continuación, comparamos los resultados obtenidos mediante el algoritmo de los k-vecinos más cercanos con k = 1 proporcionados por otra aplicación para comprobar como de buenos han sido los obtenidos por esta:

3 Prototipos	6 Prototipos	K-Vecinos (K = 1)
64’2459	66’3047	61’4783

Tabla 6: Comparación con k-vecinos Bupa.

Para este dominio, aun sin llegar a tener un porcentaje de acierto muy elevado, comprobamos que sí que mejoramos los resultados que se obtienen con la ejecución de un algoritmo de k-vecinos con un vecino.

4.5.- Ionosphere

4.5.1.- Descripción del dominio

Esta colección de datos fue recogida por un sistema de captura de datos en Goose Bay, Labrador, consistente en 16 antenas de alta frecuencia con una energía de unos 6,4 kilowatios, cuyos objetivos eran electrones libres en la ionosfera. Las señales pueden ser buena o malas según si existe algún tipo de estructura en la ionosfera o no.

Las señales recibidas son procesadas mediante una función de autocorrelación cuyos argumentos son el tiempo y número de cada uno de los pulsos para cada antena, cada número de pulso tendrá dos atributos que corresponden a valores complejos dados por la función resultante de la señal electromagnética.

Información acerca de los datos utilizados:

- Instancias: 351
- Atributos: 33 (valores continuos y uno discreto)
    - Cada uno de los atributos vienen dados por la función aplicada a las señales electromagnéticas recibidas.
- Clases: 2
    - 0: Mala señal, no se encontró evidencia de estructura en la ionosfera.
    - 1: Buena señal, existe evidencia de estructura en la ionosfera.

4.5.2.- Configuración

Los datos de configuración que recibirá el programa para las pruebas son los siguientes:

- Número de particiones para la validación cruzada: 10
- Número de dimensiones: 33

- Número de generaciones: 3000
- Probabilidad de mutación: 0’000865, resultante de la división de 1 entre el número de dimensiones al cuadrado, resultante de la división de 1 entre el número de dimensiones al cuadrado, para que en cada generación la mutación de la matriz sea realizada en una de sus posiciones (aproximadamente).
- Valor de la mutación: 0’1, este peso será multiplicado por una distribución gaussiana aleatoria, posteriormente también se ponderará por la desviación del atributo (ver siguiente dato de la configuración).
- Ponderación de las mutaciones: cada valor de la mutación obtenido mediante el dato anterior será ponderado por la desviación del atributo que se está mutando.
- Clases: 0 y 1
- Número de prototipos: 3 y 6 en cada una de las respectivas ejecuciones.

4.5.3.- Resultados

La siguiente tabla nos muestra un resumen de los datos obtenidos, mostrando los resultados por cada partición así como la media final de estos. Las columnas 2 y 3 corresponderán a los resultados de entrenamiento y test obtenidos con el clasificador inicial. Las columnas 4 y 5 muestran los resultados obtenidos por el sistema usando tres prototipos. Y, por último, las columnas 6 y 7 muestran las tasas de acierto con el uso de seis prototipos:

	Distancia Euclídea		3 Prototipos		6 Prototipos	
Partición	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
1	73.1	85.71	89.24	80.0	87.34	85.71
2	73.73	80.0	87.97	77.14	87.97	77.14
3	74.05	80.0	89.87	82.86	87.66	82.86
4	73.41	68.57	88.92	77.14	87.97	77.14
5	74.05	65.71	86.08	68.57	88.29	71.43
6	71.83	91.43	86.08	88.57	87.97	82.85
7	72.78	71.43	87.97	82.86	87.87	77.14
8	72.78	68.57	87.34	94.29	86.39	94.29
9	74.68	48.57	86.71	94.28	87.34	94.29
10	74.92	66.67	87.94	94.44	87.3	94.44
Media	**73.53**	**72.67**	**87.81**	**84.02**	**87.61**	**83.73**

Tabla 7: Resultados experimentación Ionosphere.

Para este dominio comprobamos que los resultados obtenidos por la distancia euclídea son mejorados con ambas ejecuciones de la aplicaciones, pero con la particularidad de que los resultados son ligeramente mejores con 3 prototipos que con el doble de ellos, la diferencia no es demasiado significativa pero sí que existe, cuando normalmente los resultados con seis prototipos mejoran a los obtenidos con menos prototipos.

A continuación se van a comparar los fitness durantes las fases de test al finalizar ambas ejecuciones. En la gráfica podremos apreciar una gran similitud entre los resultados para diferente número de prototipos.

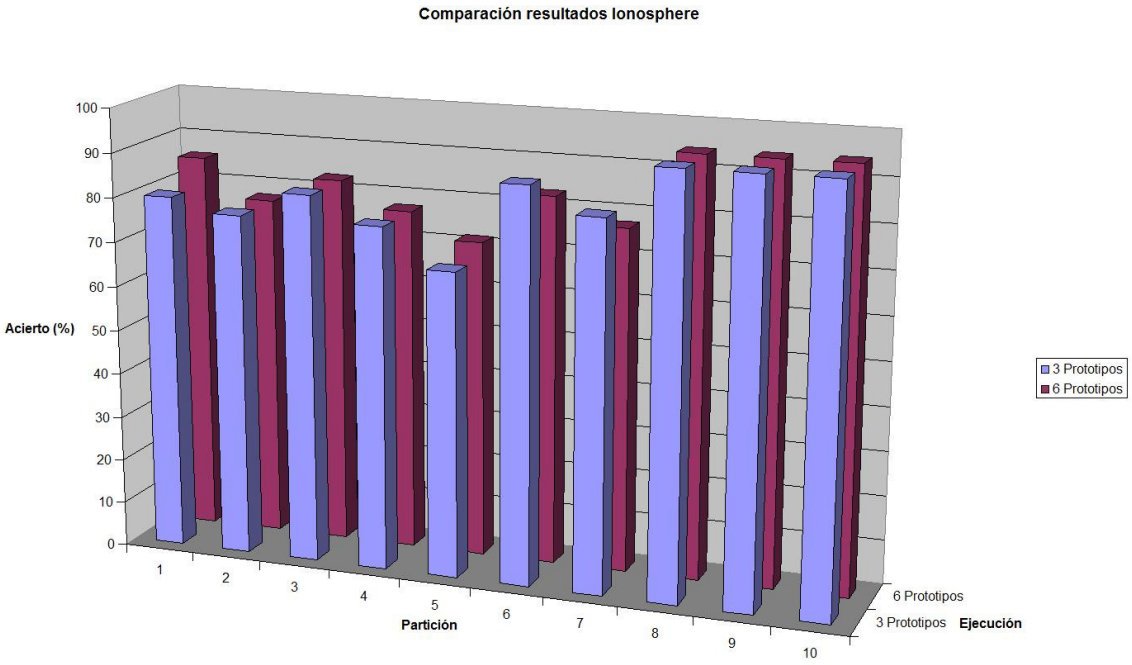

Ilustración 28: Comparación de la tasa de aciertos en test por partición con cada ejecución para el dominio Ionosphere.

Por último, comparamos los resultados obtenidos mediante el algoritmo de los k-vecinos más cercanos con k = 1, con ello podremos comprobar las soluciones arrojadas por la aplicación tratada con otra aplicación con la misma finalidad:

3 Prototipos	6 Prototipos	K-Vecinos (K = 1)
84’0159	83’7302	86’1823

Tabla 8: Comparación con k-vecinos Ionosphere.

Los resultados con el algoritmo de k-vecinos son superiores a los de ambas ejecuciones de la aplicación. Aunque la tasas de acierto no son bajas sí que no son todo lo buenas que se podrían esperar ya que, por un lado, el otro algoritmo es superior y, por otro, es extraño que no se mejoren con la experimentación con más prototipos.

4.6.- Iris

4.6.1.- Descripción del dominio

Este dominio es uno de los más sencillos a la vez que de los más conocidos dentro de la literatura sobre reconocimiento de patrones, fue creada por R. A. Fischer, los datos están separados en 3 clases de 50 instancias cada una, correspondiente a un tipo de planta del género iris.

Información acerca de los datos utilizados:

- Instancias: 150
- Atributos: 4 (valores continuos)
  - Longitud del sépalo en cm.
  - Anchura del sépalo en cm.
  - Longitud del pétalo en cm.
  - Anchura del pétalo en cm.
- Clases: 3
  - 0: Iris Setosa.
  - 1: Iris Versicolor.
  - 2: Iris Virginica.

4.6.2.- Configuración

Los datos de configuración que recibirá el programa para las pruebas son los siguientes:

- Número de particiones para la validación cruzada: 10
- Número de dimensiones: 4
- Número de generaciones: 3000
- Probabilidad de mutación: 0’0625, resultante de la división de 1 entre el número de dimensiones al cuadrado, para que en cada generación la mutación de la matriz sea realizada en una de sus posiciones (aproximadamente).

- Valor de la mutación: 0'1, este peso será multiplicado por una distribución gaussiana aleatoria, posteriormente también se ponderará por la desviación del atributo (ver siguiente dato de la configuración).
- Ponderación de las mutaciones: cada valor de la mutación obtenido mediante el dato anterior será ponderado por la desviación del atributo que se está mutando.
- Clases: 0, 1 y 2
- Número de prototipos: 3 y 6 en cada una de las respectivas ejecuciones.

4.6.3.- Resultados

La siguiente tabla nos muestra un resumen de los datos obtenidos. Las columnas 2 y 3 corresponderán a los resultados de entrenamiento y test obtenidos con el clasificador inicial. Las columnas 4 y 5 muestran los resultados obtenidos por el sistema usando tres prototipos. Y, por último, las columnas 6 y 7 muestran las tasas de acierto con el uso de seis prototipos. En todas las columnas se muestran los resultados en cada una de las particiones de la validación cruzada, así como finalmente la media de todas ellas:

	Distancia Euclídea		3 Prototipos		6 Prototipos	
Partición	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
1	92.59	93.33	97.03	100.0	97.78	100.0
2	91.85	100.0	98.52	100.0	97.04	100.0
3	92.59	93.33	98.52	93.33	97.78	93.33
4	94.81	86.67	97.78	86.67	99.26	86.67
5	91.85	100.0	96.3	100.0	99.26	100.0
6	93.33	93.33	97.04	93.33	97.04	100.0
7	93.33	86.67	98.52	100.0	97.04	93.33
8	91.11	93.33	98.52	93.33	99.26	100.0
9	94.07	93.33	100.0	80.0	97.78	80.0
10	92.59	93.33	97.78	93.33	97.78	93.33
Media	**92.81**	**93.33**	**98.0**	**94.0**	**98.0**	**94.67**

Tabla 9: Resultados primera experimentación Iris.

Como se comentó en la descripción del dominio, este es uno de los más sencillos que existen y es reflejado en los resultados obtenidos, incluso ya con la distancia euclídea conseguimos resultados por encima del 85%. Estos altos coeficientes los podemos comprobar en el siguiente gráfico:

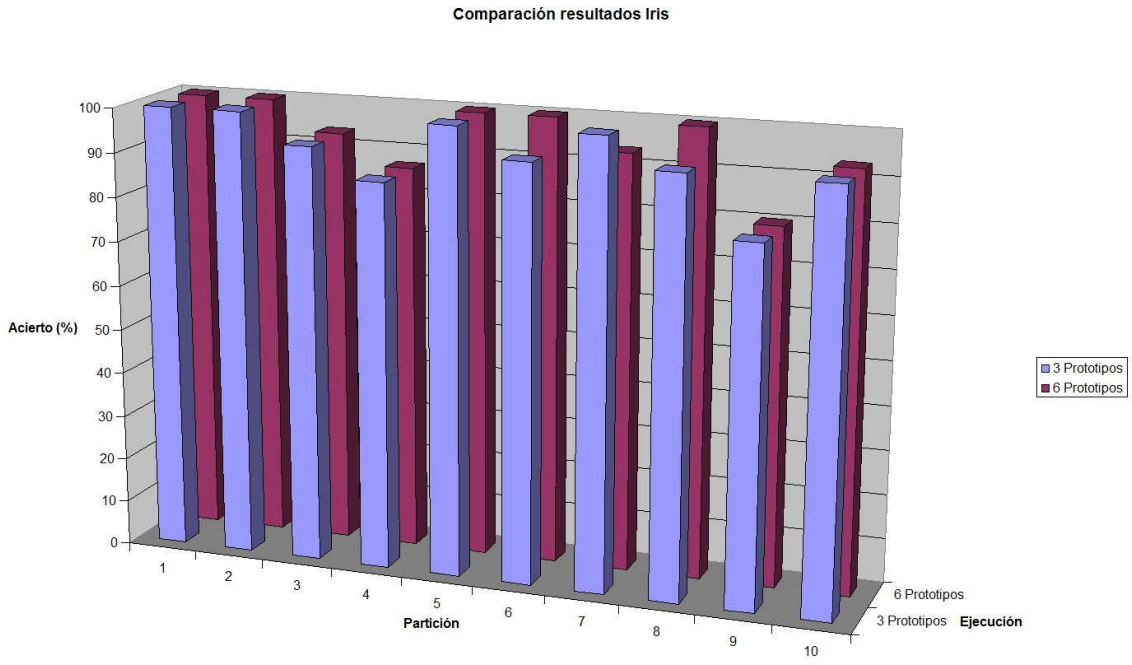

Ilustración 29: Comparación de la tasa de aciertos en test por partición con cada ejecución para el dominio Iris.

La experimentación con los algoritmos hacen que los resultados mejoren de manera significativa, podemos verlo muy bien en los resultados de los entrenamientos, estos son cercanos al 100%, de hecho en ninguna partición bajan de un fitness del 97%.

Pero si nos fijamos en los resultados finales en la experimentación con los datos de test, estos bajan en ambas. Repasando las particiones comprobamos que la partición 4 y, sobre todo, la número 9 son las únicas que bajan de una tasa del 90%. La explicación de que esto aparezca puede ser el hecho de que este dominio es muy poco poblado, lo que hace que las particiones de test sean de unos 15 individuos, es decir, que cada fallo encontrado durante la fase de test genera una disminución de aproximadamente un 7% en el fitness de esa partición.

A continuación, comparamos los resultados obtenidos mediante el algoritmo de los k-vecinos más cercanos con k = 1, para comprobar si esta forma de obtener un mejor fitness es mejor o peor que otra manera diferente de obtener el mismo objetivo:

3 Prototipos	6 Prototipos	K-Vecinos (K = 1)
94’0000	94’6667	96’0000

Tabla 10: Primera comparación con k-vecinos Iris.

En esta ocasión comprobamos que el valor de la experimentación es también inferior a la del algoritmo de k-vecinos. Sin embargo, podemos comprobar que los datos de entrenamiento sí que tienen unos tantos por ciento superiores por lo que obtenemos unos resultados aceptables.

4.7.- Wine

4.7.1.- Descripción del dominio

Mediante análisis químico de los componentes de los vinos cultivados en una misma región de Italia, este dominio determina a que variedad cultivada pertenece entre 3 diferentes [Cortez y col., 2009].

El dominio inicialmente tenía un mayor número de atributos, pero en el conjunto de datos utilizado (el encontrado en UCI Machine Learning Repository) los componentes químicos se reducen a 13.

Información acerca de los datos utilizados:

- Instancias: 178
- Atributos: 13 (valores continuos)
  - Alcohol.
  - Ácido málico.
  - Ceniza.
  - Alcalinidad de la ceniza.
  - Magnesio.
  - Número total de fenoles.
  - Flavonoides.
  - Fenoles no-flavonoides.
  - Proanthocyanidins.
  - Intensidad del color.
  - Tono del color.
  - OD280/OD315 de vinos diluidos.
  - Prolina.

- Clases: 3
  - 1: Variedad 1.
  - 2: Variedad 2.
  - 3: Variedad 3.

4.7.2.- Configuración

Los datos de configuración que recibirá el programa para las pruebas son los siguientes:

- Número de particiones para la validación cruzada: 10
- Número de dimensiones: 13
- Número de generaciones: 3000
- Probabilidad de mutación: 0’0059, resultante de la división de 1 entre el número de dimensiones al cuadrado, para que en cada generación la mutación de la matriz sea realizada en una de sus posiciones (aproximadamente).
- Valor de la mutación: 0’1, este peso será multiplicado por una distribución gaussiana aleatoria, posteriormente también se ponderará por la desviación del atributo (ver siguiente dato de la configuración).
- Ponderación de las mutaciones: cada valor de la mutación obtenido mediante el dato anterior será ponderado por la desviación del atributo que se está mutando.
- Clases: 1, 2 y 3
- Número de prototipos: 3 y 6 en cada una de las respectivas ejecuciones.

4.7.3.- Resultados

La siguiente tabla nos muestra un resumen de los datos obtenidos, mostrando los resultados por cada partición así como la media final de estos, comprobamos también la evolución que han tenido los resultados añadiendo los obtenidos con la distancia euclídea que es el clasificador inicial, que pueden ser vistos en la segunda y tercera columna. Para comprobar los resultados tras la ejecución, tenemos que en las columnas

4 y 5 podemos ver los porcentajes de la ejecución con tres prototipos, y en la última y penúltima columnas aparecen los fitness de la experimentación usando seis prototipos:

	Distancia Euclídea		3 Prototipos		6 Prototipos	
Partición	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
1	71.43	82.35	75.78	76.47	75.78	76.47
2	72.05	76.47	73.91	70.59	76.40	64.71
3	72.05	82.35	79.5	70.59	74.53	82.35
4	72.05	76.47	75.77	64.71	75.15	76.47
5	73.29	64.71	74.53	52.94	74.53	53.94
6	73.91	58.82	75.78	64.71	75.16	70.59
7	74.53	52.94	75.78	47.06	77.64	52.94
8	73.29	64.71	75.78	52.94	75.16	52.94
9	71.43	82.35	72.67	82.35	75.16	88.23
10	71.24	80.0	71.90	80.0	71.90	80.0
Media	**73.53**	**72.12**	**75.14**	**66.24**	**75.14**	**69.86**

Tabla 11: Resultados primera experimentación Wine.

Con la ejecución de nuestro sistema conseguimos que el fitness en la fase de entrenamiento aumente, pero esta subida no es muy significativa. Además, en la fase de test la tasa de acierto disminuye con respecto a la distancia euclídea con lo que no estamos consiguiendo buenos resultados.

Durante el entrenamiento no se aprecian grandes diferencias en los fitness de las diferentes particiones, todos se mueven en torno al 75%, sin bajar del 70% y sin sobrepasar el 80%. Sin embargo, si observamos las tasas de acierto en las fases de test tienen unos resultados muy dispares. En la siguiente ilustración se podrán comprobar estas grandes fluctuaciones:

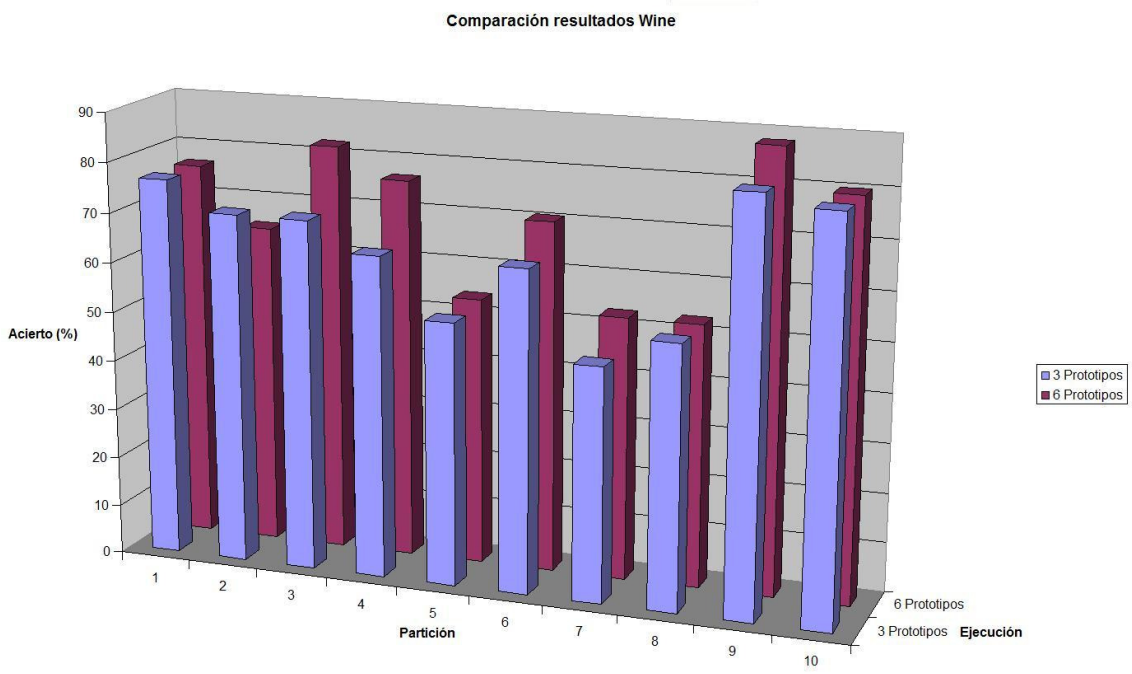

Ilustración 30: Comparación de la tasa de aciertos en test por partición con cada ejecución para el dominio Wine.

A continuación, comparamos los resultados obtenidos mediante el algoritmo de los k-vecinos más cercanos con k = 1, estos resultados con el algoritmo de k-vecinos han sido obtenidos por otros medios y nos sirven para comparar los resultados obtenidos por dos aplicaciones con el mismo objetivo pero diferente algoritmo para la búsqueda de la solución:

3 Prototipos	6 Prototipos	K-Vecinos (K = 1)
66’2353	69’7647	75’9551

Tabla 12: Comparación con k-vecinos primera experimentación Wine.

La anterior gráfica mostró que los resultados tanto con seis como con tres prototipos eran inferiores a los conseguidos con la distancia euclídea, por tanto, si comparamos con los resultados obtenidos por el algoritmo de k-vecinos es aun inferior en ambos casos.

En todos los dominios se ha realizado todas las ejecuciones con la misma configuración. A continuación, se va a proceder a mostrar de manera informativa los datos de se

consiguen cambiando alguno de los datos de la configuración. En este caso, cambiaremos el peso de la mutación de un 0'1 a un 0'01 con lo que las mutaciones de los prototipos y de la matriz serán menos significativas en cada generación.

Se muestra en una tabla análoga a la anterior de los resultados por particiones que esta segunda ejecución de la aplicación devuelve los siguientes resultados:

	Distancia Euclídea		3 Prototipos		6 Prototipos	
Partición	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
1	71.43	82.35	73.91	82.35	74.53	82.35
2	72.05	76.47	73.29	70.59	77.02	76.47
3	72.05	82.35	74.53	82.35	77.64	76.47
4	72.05	76.47	75.15	64.71	76.40	64.71
5	73.29	64.71	74.53	64.71	75.15	76.47
6	73.91	58.82	74.53	58.82	75.15	58.82
7	74.53	52.94	77.02	52.94	77.64	52.94
8	73.29	64.71	75.15	58.82	77.02	64.71
9	71.43	82.35	73.91	82.35	74.53	82.35
10	71.24	80.0	76.47	80.0	72.55	84.0
Media	**73.53**	**72.12**	**74.84**	**69.76**	**75.76**	**71.93**

Tabla 13: Resultados experimentación Wine modificando la mutación.

Podemos comprobar que los resultados han mejorado en ambos casos, sin embargo, si comprobamos los datos obtenidos durante el entrenamiento, no existe apenas diferencia entre los resultados de la anterior tabla y los de esta, simplemente, algunas de las matrices y prototipos encontrados en algunas particiones fueron más propicios a la hora de comprobar con los resultados de test. Por lo que la modificación de este dato no ha supuesto un cambio significativo a la hora de experimentar con los datos de entrenamiento.

Comparando los resultados de la segunda ejecución con el algoritmo de k-vecinos:

3 Prototipos	6 Prototipos	K-Vecinos (K = 1)
69’7647	71’9294	75’9551

Tabla 14: Comparación con k-vecinos segunda experimentación Wine.

Podemos ver que los resultados aun siguen siendo inferiores a los que se consiguen con el algoritmo de k-vecinos, aunque ya resultan más cercanos en el caso de los seis prototipos, aunque como se dijo antes la mejora solo ha sido detectada en los datos de test, el entrenamiento tuvo unos fitness similares en ambas experimentaciones.

Capítulo 5

Conclusiones y líneas futuras

5.1.- Conclusiones

El objetivo del proyecto era comprobar la efectividad que tendría el realizar la búsqueda de soluciones en diferentes dominios mediante la mutación de una matriz que pondere la distancia y de los prototipos de cada una de las clases que se puedan dar en los diferentes problemas a realizar. Se ha probado con tres y con seis prototipos de mutación de los centroides.

Hemos podido comprobar que el algoritmo utilizado mejora los resultados que daría el cálculo de la distancia euclídea a los diferentes prototipos de las clases. También hemos podido comprobar como es este algoritmo en comparación a otro con el mismo objetivo, en este caso, con un algoritmo de k-vecinos con k=1.

A continuación presentamos un resumen de los resultados recogidos durante el proyecto, es decir, la media de las 10 particiones de test realizadas durante la validación cruzada, comparadas con la de la distancia euclídea (comienzo de la ejecución) y con el algoritmo de k-vecinos (con k=1):

	Euclídea	3 Prototipos	6 Prototipos	K-Vecinos
Ripley	75.28	**86.2400**	**86.5600**	88.5760
Blood	67.50	**75.4944**	**76.3052**	68.4091
Bupa	55.93	**64.2459**	**66.3047**	61.4783
Ionosphere	72.67	**84.0159**	**83.7302**	86.1823
Iris	92.00	**94.6667**	**96.0000**	96.00
Wine	72.12	**66.2353**	**69.7647**	75.9551
Media	**72.5833**	**78.4830**	**79.7775**	**79.4335**

Tabla 15: Comparación resultados finales.

Podemos comprobar que en casi todas las ocasiones (salvo en el caso de Wine) el uso de los prototipos consigue mejorar el resultado de la distancia euclídea de manera notable.

También podemos comprobar que el uso de seis prototipos mejora casi siempre los resultados obtenidos con tan solo tres prototipos.

Por último, comparando los resultados con el algoritmo de k-vecinos comprobamos que no hay unidad en los resultados, dependiendo del dominio un método o el otro arroja mejores resultados. Se comprueba que finalmente con las medias una visión general de los resultados nos diría que ambos algoritmos tienen una efectividad bastante parecida, pero con la ventaja de que el método de prototipos reduce el número de datos a 3 o 6 por clase, lo que hace que la determinación de si un dato de test pertenece a una u otra clase sea mucho más rápida.

Por tanto, la aplicación realizada durante el proyecto mejora a la distancia euclídea no ponderada y que tiene unos resultados similares a los que se podrían obtener mediante el uso de k-vecinos.

Para comprobar costo en tiempo de ejecución que supone el uso del doble de prototipos mostraremos una tabla comparativa de los tiempos obtenidos con cada dominio durante la experimentación:

	3 Prototipos	6 Prototipos
Ripley	544927 ms.	1074488 ms.
Blood	307416 ms.	484840 ms.
Bupa	233528 ms.	365758 ms.
Ionosphere	1666455 ms.	1875827 ms.
Iris	166396 ms.	226407 ms.
Wine	538073 ms.	886996 ms.
Resultado	X	1’4216X

Tabla 16: Comparación de tiempos con diferente número de prototipos.

En resumen, para poder llevar a cabo la mejora que supone el uso de seis prototipos necesitamos aproximadamente una vez y media el tiempo que requeriría la ejecución de la aplicación con tres prototipos.

5.2.- Líneas futuras

En un futuro sería posible explotar diferentes caminos de investigación partiendo de este proyecto como punto de partida. A continuación, se exponen unos cuantos ejemplos:

- **Búsqueda de configuraciones más eficaces:** utilizar diferentes a la estándar usada en la experimentación, consistente en 3.000 generaciones, mutaciones de 0'1, probabilidades de mutación de uno entre el cuadrado de las dimensiones del dominio, 3 o 6 prototipos, etc. Ya que quizá alguna otra configuración mejore el algoritmo en general o el problema para algún dominio en particular.

- **Evitar el elitismo "cerrado"**: en el presente proyecto se ha utilizado para la siguiente generación el mejor hijo o el padre en caso de no haberse mejorado, esto ha dado lugar a muchas generaciones sin ninguna mejora del fitness, quizá se podría investigar los diferentes caminos que seguirían uno o varios hijos aun teniendo peor tasa de acierto porque posiblemente más adelante se encuentren mejoras, pudiendo poner un punto de reseteo si en un número de generaciones no se ha conseguido aumentar el fitness.

- **Uso de matrices no identidad como comienzo:** aunque no se ha utilizado durante la experimentación, el propio código tiene la opción de inicializar las clases con matrices aleatorias, en las que el comienzo ya no sería la distancia euclídea sin ponderar sino una ponderada.

- **Dominios con soluciones reales:** en este proyecto se han utilizado dominios en los que el atributo a predecir era discreto, se podría hacer que se utilizase para problemas de tipo real.

Anexos

Anexo A: Presupuesto

Para la realización del proyecto se han requerido nueve meses de trabajo de un ingeniero a razón de 5 hombres mes.

Además de estos costes de personal, también se han necesitado materiales para la realización. Estos costes materiales se ven reducidos a un ordenador de sobremesa para realización de la experimentación y un ordenador portátil para las tareas de desarrollado, así como unos pequeños costes en bienes fungibles debido a la utilización de material de oficina.

En los costes materiales no se ven reflejadas licencias de software debido a que se han utilizado programas libres, tanto el sistema operativo como el IDE Eclipse para la programación del código fuente.

Con todo esto, el presupuesto total del proyecto asciende a 25.968 €.

PRESUPUESTO DE PROYECTO

1.- Autor: Rubén Villagarcía Vicente

2.- Departamento: Informática, Inteligencia Artificial

3.- Descripción del Proyecto:

- Título: Optimización evolutiva de distancias para clasificadores
- Duración (meses): 9
- Tasa de costes indirectos: 20%

4.- Presupuesto total del Proyecto (valores en Euros):

Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a título informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
					0,00	
Villagarcía Vicente, Rubén		Ingeniero Senior	5	4.289,54	21.447,70	
		Ingeniero		2.694,39	0,00	
					0,00	
					0,00	
		Hombres mes	5	Total	21.447,70	

[a] 1 Hombre mes = 131,25 horas. Máximo anual
Máximo anual para PDI de la Universidad Carlos III

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [c]
Ordenador para pruebas	450,00	100	9	60	67,50
Portátil de trabajo	500,00	100	9	60	75,00
		100		60	0,00
		100		60	0,00
		100		60	0,00
					0,00
				Total	142,50

[c] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses
B = periodo de d
C = coste del eq
D = % del uso qu

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Fungible		50,00
	Total	50,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,...

6.- Resumen de costes

Presupuesto Costes Totales	Presupuesto Costes Totales
Personal	21.448
Amortización	143
Subcontratación de tareas	0
Costes de funcionamiento	50
Costes Indirectos	4.328
Total	25.968

Anexo B: Referencias

[Atkeson, Moore y Schaal, 1997] C.G.Atkeson, A.W.Moore, and S.Schaal. Locally weighted learning. Artificial Intelligence Review, 11:11-73, 1997.

[Bäck, 1996] T. Bäck. Evolutionary algorithms in theory and practice. Oxford New York: Oxford University Press, 1996.

[Cortez y col., 2009] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553. ISSN: 0167-9236.

[Cover y Hart, 1967] T.M. Cover and P.E. Hart. Nearest neighbor pattern classication. IEEE Trans. Inform. Theory, 13(1):21--27,1967.

[Darwin, 1859] Darwin, C. (1859). On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life. John Murray, London, (1), 1–556.

[Eiben y Smith, 2003] A. E. Eiben and J. E. Smith. Introduction to Evolutionary Computing. Natural Computing Series. Springer, October 2003.

[Frank y Asuncion, 2010] Frank, A. & Asuncion, A. (2010). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[Holland, 1975] John H. Holland. Adaptation in Natural and Artificial Systems. The University of Michigan Press, 1975.

[Mendel, 1865] Gregor Mendel. (1865). Experiments in Plant Hibridization. Read at the February 8th, and March 8th, 1865, meetings of the Brünn Natural History Society.

[Ripley, 1997] B.D. Ripley. Pattern Recognition and Neural Networks Cambridge: Cambridge University Press, 1996.

[Russell y Norvig, 2003] Russell, Stuart J.; Norvig, Peter. Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 111–114

[Somervuo y Kohonen, 1999] Somervuo, P. and Kohonen, T. (1999). Self-organizing maps and learning vector quantization for feature sequences. Neural Process. Lett., 10(2):151--159.

[Valls, Aler y Fernández, 2007] J.M. Valls, R. Aler and O. Fernández. Evolving Generalized Euclidean Distances for Training RBNN. Computing and Informatics. 26:33-43. 2007

[Yeh, Yang y Ting, 2008] Yeh, I-Cheng, Yang, King-Jang, and Ting, Tao-Ming, "Knowledge discovery on RFM model using Bernoulli sequence," Expert Systems with Applications, 2008