UNIVERSIDAD DE GRANADA

METAHEURÍSTICAS

Prácticas Grupo 1, Lunes 17:30-19:30
Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Características

Nombre: Adrián Jesús
Apellidos: Peña Rodríguez
Correo: adrianprodri@correo.ugr.es
DNI: 45604927-K
Fecha de entrega: 04/05/18

1.- Descripción del problema

El problema que hemos elegido es el APC, o lo que es lo mismo el problema de aprendizaje de pesos en características.

Consiste en diseñar un clasificador el cual mediante la asignación de unos determinados pesos se consigue que este mejore en eficiencia y fiabilidad. Para diseñar este clasificador son necesarias dos tareas, aprendizaje y validación.

Los datos a tratar por nuestro clasificador se dividen en dos grupos:

- Entrenamiento: Utilizado para que el clasificador aprenda,
- Prueba: Se usa para validar el aprendizaje de nuestro clasificador. Se calculan el porcentaje de clasificación sobre los ejemplos de este conjunto, los cuales son desconocidos por la tarea de aprendizaje, para así conocer su poder de generalización.

Para determinar con mayor seguridad la fiabilidad de nuestro clasificador, usaremos un método de particiones llamado k-fold cross validation donde nuestra k será 5, esto quiere decir que dividiremos nuestros datos en 5 partes diferentes en las que cada una tendrá un 80% de los datos para aprender y el 20% restante serán para la validación (manteniendo la proporción de las distintas clases que tengan nuestros datasets).

Nuestro clasificador será un K Nearest Neighbors, es un método de clasificación supervisada, que sirve para estimar la función de densidad $F(\frac{x}{C_j})$ de las predicciones x por cada clase C_j

Una vez conocemos nuestro clasificador, lo que buscaremos es mejorar su capacidad de predicción mediante el uso de una ponderación de características(pesos), la cual iremos obteniendo en las diferentes sesiones de prácticas de la asignatura.

Determinaremos la validez de nuestros algoritmos con la siguiente función de evaluación donde α puede ser cualquier n ∈ [0, 1] aunque por defecto será 0.5:

$$F(W) = \alpha \cdot tasa - clas(W) + (1 - \alpha) \cdot tasa - red(W)$$

$$tasa - red = 100 \cdot \frac{nº\ valores\ w_i < 0.2}{nº\ características} \qquad tasa - clas = 100 \cdot \frac{nº\ instancias\ bien\ clasificadas\ en\ T}{nº\ instancias\ en\ T}$$

2.- Introducción a la práctica

Como sabemos la práctica que nos ocupa tiene como objetivo comparar los resultados de un clasificador K Nearest Neighbors, en adelante KNN, con las tres variables vistas en la práctica 1 además de con los algoritmos que hemos desarrollado en esta práctica 2:

1. Clasificador con todos los pesos a 1: Esta variable es básicamente el clasificador KNN sin ningún peso, es decir, tomando todas las características tal cual de los dataset y en base a estas encontrar los K vecinos más cercanos y devolver una predicción.
2. Clasificador con pesos obtenidos mediante RELIEF: para esta variable calcularemos un vector de pesos que multiplicaremos por nuestras características para de este modo priorizar a unas sobre otras o incluso eliminar las que según este algoritmo no son relevantes. Al ser un método greedy, no podemos garantizar que nos mejore la solución sin pesos por lo que veremos su comportamiento en las pruebas que se describen posteriormente.
3. Clasificador con pesos obtenidos mediante búsqueda local: esta variable es la primera que implementaremos que tendrá una heurística, sencilla pero que como veremos nos proporciona unos resultados interesantes.
4. Clasificador con pesos obtenidos mediante algoritmos genéticos generacionales con dos operadores de cruce diferentes (CA-Cruce aritmético y BLX-α donde $\alpha = 0.3$). Esta variable consiste en usar una estrategia basada en genéticos para así poder obtener con el paso de múltiples generaciones una mejor solución que las propuestas anteriormente. Tenemos que una de las características clave de esta estrategia con respecto a otros genéticos es que en este variable cada generación nueva sustituye a la anterior manteniendo el elitismo, esto es que si el mejor individuo de la generación actual va a ser sustituido por uno nuevo, el mejor individuo de la antigua generación sobrevivirá en la nueva echando al peor de esta. Veremos las bondades de los dos operadores de cruces usados en el análisis específico de esta variable.
5. Clasificador con pesos obtenidos mediante algoritmos genéticos estacionarios con dos operadores de cruce diferentes (CA-Cruce aritmético y BLX-α donde $\alpha = 0.3$). A diferencia de la estrategia anterior, tenemos que en esta ocasión no avanzaremos en base a las generaciones si no que esta vez se designarán dos padres y estos se cruzarán para dar lugar a dos hijos los cuales sustituirán a estos padres en la generación actual. Lo explicaremos más detalladamente en su correspondiente sección.

La implementación que hemos realizado a parte de las funcionalidades extra que tenía la práctica 1, se han añadido nuevas y además estos nuevos algoritmos se han diseñado para que sean lo más generales posibles, de manera que nos sirvan para más problemas de esta índole.
Las implementación son capaces de adaptarse a cualquier problema de este estilo, teniendo para ello un set de atributos totalmente configurables de tal manera que podemos experimentar con ellos para así poder, eventualmente, obtener una mejor solución en función de nuestras necesidades (distintos criterios de aceptación de una solución, etc).

Para la realización de esta segunda práctica hemos usados Python como lenguaje principal con las siguientes librerías:

1. scikit-learn: Para la implementación de nuestro KNN, lo implemente yo mismo completo pero ante la lenta ejecución en determinados datasets decidí usar esta implementación.
2. numpy: Librería para análisis matemático y manejo de datos.
3. scipy: Para tratamiento de archivos ARFF
4. Se importó operator para poder implementar por mi mismo el KNN aunque al final de este uso unas cosas y de la implementación de scikit-learn otras.
5. math: básico para nuestras operaciones.
6. time: Esto nos sirve para medir los tiempos de ejecución.

En esta ocasión hemos dividido en varios archivos nuestra práctica para así poder mantener y actualizar el código de una manera más simple, además de ayudar a la corrección de esta.

Los parámetros de ejecución de la práctica son los mismos que en la práctica 1 y para los nuevos algoritmos se han usado los parámetros que se indican en la práctica como valores por defecto pero pueden cambiarse a conveniencia para así poder investigar de manera más exhaustiva el comportamiento de estos algoritmos.

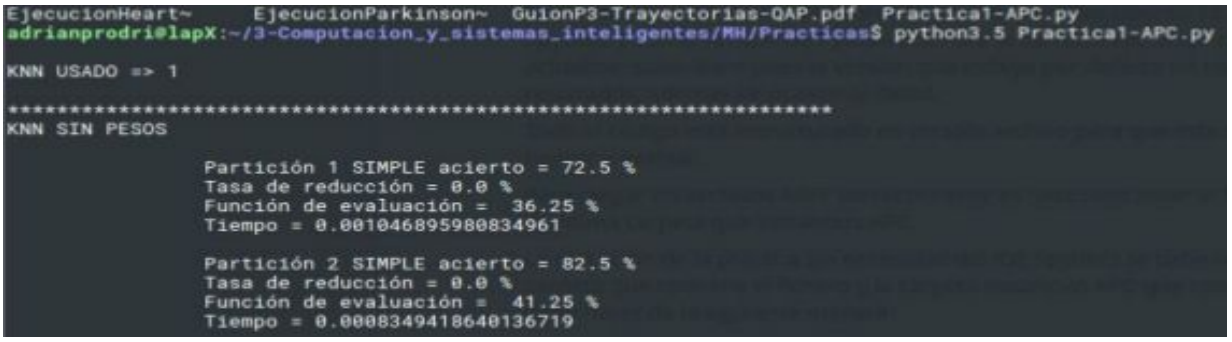
Además de esto se han corregido los problemas indicados en la práctica 1 como son el ruido causado por filas exactamente iguales, para suplir este problema se han usado un tipo de dato set, el cual no permite repetidos, cada vez que un dato no era introducido, el índice de este se guardaba para poder eliminarlo de las etiquetas.

Para la representación de nuestras soluciones, tenemos lo siguiente:

1. Pesos: Se usarán para intentar mejorar los resultados de nuestro KNN, para representarlo hemos optado por cambiar la representación en numpy por listas de python las cuales nos permiten un manejo más sencillo de los datos, además de una alta velocidad de modificado. Aún así seguimos usando la librería numpy pues nos sirve para multitud de tareas.
2. X_train, X_test: Son dos conjuntos de datos que contienen nuestras características, en X_train tenemos el conjunto de entrenamiento y en X_test, el conjunto con el que evaluaremos el aprendizaje de nuestro KNN que obtuvo con X_train.

3. y_train, y_test: Son los dos conjuntos de etiquetas, y_train contiene las etiquetas de las características de X_train, con este conjunto entrenamos nuestro KNN y hacemos las predicciones cuando cojemos X_test y con y_test comprobamos nuestros aciertos.
4. Para calcular la media de aciertos hacemos la sumatoria de los aciertos de todas las particiones y la dividimos por el número de particiones, hacemos esto mismo para los errores también y para la tasa de reducción.

La práctica ha sido realizada sobre el sistema operativo Elementary OS (basada en Ubuntu 16.04 LTS) con el IDE PyCharm y con la versión de Python 3.5. Para la correcta ejecución del código es preciso actualizar scikit-learn pues la versión que incluye por defecto no me proporcionaba buenos resultados, además de ocasionar fallos.
El código a sido reestructurado en más de un archivo para así poder mantenerlo y actualizarlo de manera más simple.
Para cargar los archivos ARFF correctamente es necesario tener el fichero de la práctica en la misma carpeta que Instancias APC.
La ejecución de la práctica sin necesidad del IDE Spyder3 se debe realizar situándose en la carpeta que contiene el fichero y la carpeta Instancias APC que contiene los datasets, se debe hacer de la siguiente manera:



3.- Descripción general de los algoritmos Genéticos y Meméticos

Comenzaremos indicando cuál ha sido el esquema de representación de las soluciones, para representar las soluciones se ha usado una lista de python en la que se encuentran todos los genes de la solución (gen = peso específico para la solución que ocupa ese lugar, GEN_i = característica$_i$).
Por tanto para nosotros, una generación de individuos será una lista de listas del tamaño que corresponda según el número de características y el número de individuos por generación.

En cuanto a nuestra función objetivo tenemos que es la siguiente:
Nuestra función objetivo es la siguiente:

$$F(W) = \alpha \cdot tasa - clas(W) + (1 - \alpha) \cdot tasa - red(W)$$

$$tasa - red = 100 \cdot \frac{nº\ valores\ w_i < 0.2}{nº\ características} \qquad tasa - clas = 100 \cdot \frac{nº\ instancias\ bien\ clasificadas\ en\ T}{nº\ instancias\ en\ T}$$

Por tanto un individuo de una población será mejor que otro si al aplicarle esta función, su valor es superior al de su rival, a esto lo denominaremos fitness.
Puesto que usamos un α de 0.5 quiere decir que damos la misma prioridad a lo bueno que es nuestro KNN para clasificar como a lo sencillo que es (número de características tenidas en cuenta), tenemos que un clasificador con un 100% de acierto puede ser peor que uno con 90% si este último usa menos características que el primero.
Para la realización de la práctica hemos dejado los dataset obligatorios que son Parkinsons, ozone-320 y spect-heart y hemos tenido que eliminar los añadidos a parte excepto el de diabetes debido a que en sus filas contenían errores y caracteres raros que provocan un fallo que impedía su ejecución.

A la hora de generar la población inicial, usamos un esquema de generación aleatorio entre un mínimo y un máximo elegidos (por defecto mínimo = 0.0 y máximo = 0.0), además tenemos una cota inferior mediante la cual si un gen del individuo está por debajo de esta lo colocamos a 0, al igual que una cota superior para colocarle el valor máximo, el pseudocódigo de esto es el siguiente:

```
def generaIndividuo:
        generados<-distribución uniforme entre min y max
        for n, i en aleatorios:
                si i < cota-inferior
```

```
            generados<-0.0
        si i > cota-superior
            generados = max
```

Podemos observar que la cota superior es prescindible pero esta puesta por si se da el caso de que queremos generar entre un min y un max más elevado y queremos que los valores no pasen de 0.0 y la cota superior.

```
def generaPoblación:
    for i en 0:tamañoPoblacion:
        generaIndividuo
```

Con esto generamos la población, en el código adjunto a este pdf se puede apreciar esto con todos sus parámetros.

En cuanto al esquema de representación de nuestro operador de selección, tenemos que usamos torneo binario, esto es, escogemos aleatoriamente dos individuos y de estos devolvemos el mejor de ellos. Su pseudocódigo es:

```
def torneoBinario:
    índices padres aleatorios = aleatorio entre 0 y el número de individuo de la población
    fitness padre 1 = calculamos su fitness
    fitness padre 2 = calculamos su fitness
    Devolvemos el índice del mejor padre
```

Por tanto en cada algoritmo lanzaremos tantos torneos binarios como sea necesario, por ejemplo en el modelo generación lanzaremos tantos torneos binarios como individuos se tengan que cruzar (prob de cruce * tamaño de la población). Cabe decir que es posible que aparezcan repetidos pues que sean elegidos una vez no quiere decir que no puedan ser candidatos a más cruces dentro de la misma generación.

A continuación, vamos a hablar de los distintos operadores de cruce. Se nos proponen dos tipos de cruce diferentes, unos aritmético y otro el cruce BLX-α (Eshelman, L. J. and Schaffer, J. D. 1993) donde $\alpha = 0.3$ en nuestro caso, aunque esto se puede modificar ya que la implementación lo permite.
Después de la experimentación con ambos cruces, llegamos a la conclusión de que el cruce aritmético de menor tasa de reducción en dos de los datasets (Parkinsons y Spect-Heart), funciona mejor en dos de los dataset y debido a esto es el que elegiremos para los cruces en nuestros algoritmos genéticos.
Adicionalmente aunque no se pida colocare la ejecución de los algoritmos meméticos con el cruce aritmético también para ver si efectivamente el comportamiento de este cruce sigue siendo peor o por el contrario no. Para hacer esto solo debemos de colocar el parámetro de BLX = 0.0. Vamos con el pseudocódigo de ambos:

```
def cruceAritmético:
```

Nos proporcionan dos individuos a cruzar
Gen a gen realizamos la media de ambos
Comprobamos cotas
Si estamos en generacional:
Nos quedamos cn el mejor padre(podríamos quedarnos también con el peor)
Devolvemos mejor padre e único hijo

def cruceBLX:
Calculamos C_{max}
Calculamos C_{min}
Calculamos I
Generamos hijos con cada gen entre C_{min}-I* α , C_{max}+I* α y el número de genes
Comprobamos cotas
Devolvemos hijos

Por tanto como podemos ver en los pseudocódigos, tenemos que en el cruce aritmético solo nos aparece un hijo de cada dos padres, por tanto nos quedamos con el mejor padre y ese hijo para no tener un déficit de individuos, cabe señalar que podríamos quedarnos también con el peor, esto es solo una opción.

Por último, definiremos el mecanismo de mutación. Para la mutación se ha elegido el mismo movimiento que en la práctica 1 el cual en la BL modificaba una componente, por tanto nuestra mutación nos servirá también para nuestra BL:

def mutación:
Generamos un movimiento con una distribución gaussiana con varianza (0.3, con posibilidad de modificarla)
Sumamos el movimiento al gen pasado
Comprobamos cotas
Devolvemos gen mutado

Con esto tenemos descritos los operadores básicos en estas técnicas de búsqueda.

4.- Pseudocódigos y explicación de estos

En este apartado se describirán los algoritmos desarrollados, así como sus peculiaridades y similitudes.
Comencemos con los algoritmos genéticos generacionales, con sus dos cruces correspondientes, aritmético y BLX-α:

def generacionales:
    Copiamos generacion actual en una nueva generacion sobre la que trabajaremos
    Mientras iteracion actual < iteracion maxima(15000, puede cambiarse):
        Repetimos según la presión establecida(prob de cruce * tamaño poblacion/2):
            Guardamos el mejor individuo actual
            Cruzamos dos individuos seleccionados con dos torneos binarios
            Cruzamos o bien con aritmético o con BLX-α
            Sustituimos los padres actuales por los dos nuevos individuos(En la nueva generación)
            Si el mejor a sido sustituido:
            sobreescrito <- True

        Mutamos aleatoriamente el número de cromosomas correspondiente según la probabilidad de mutación
        Si sobreescrito:
            peorNueva<-mejorActual (En la nueva generación)
        Sustituimos actual por nueva generación

    Devolvemos la mejor de las soluciones actuales

El esquema de generación que sigue este algoritmo es el de generación a generación, es decir, la generación actual será sustituida por una nueva generación en la cual existirán individuos de la actual (debido a la presión de cruce(<1.0)) y además mantendremos el elitismo pues tenemos que el mejor individuo actual siempre sobrevivirá y se pasará a la nueva generación eliminando el peor de esta. Esta variante de algoritmo genético tiene como característica que al sustituir gran parte de la generación pero no toda conseguimos mantener cierto grado de diversidad en las soluciones, a costa de que las nuevas soluciones son parecidas a sus padres pues son la media de estas y además uno de los hijos es uno de los padres por los que mantenemos cierto grado de diversidad y no tenemos una convergencia tan alta como en el modelo estacionario el cual analizaremos posteriormente.
Posteriormente analizaremos su comportamiento en los dataset obligatorios.

def estacionarios:

Copiamos generación actual en una nueva generación sobre la que trabajaremos
Mientras iteración actual < iteración máxima(15000, puede cambiarse):
Repetimos según la presión establecida(prob de cruce * tamaño población/2):
Seleccionamos dos padres con torneo binario
Cruzamos con unos de nuestros dos cruces
Obtenemos los dos peores de la generación actual
Evaluamos los peores
Evaluamos los nuevos generados a partir del cruce
Si mejoran a los peores
Los introducimos y sacamos los peores
Sino Se eliminan

Mutamos aleatoriamente el número de cromosomas correspondiente según la probabilidad de mutación
Sustituimos la generación actual por la nueva

Devolvemos la mejor de las soluciones actuales

En este modelo tenemos que la diversidad se mantiene bien al principio pero por la manera de introducir los nuevos individuos en la generación, tenemos que su convergencia es más alta que la del modelo generacional. Además de esto tenemos que puede que por puro azar no se lleguen a coger soluciones que podrían dar lugar a una cepa nueva que eventualmente llegase a dar una mejor solución. Este problema también ocurre en el modelo generacional pero es menos acusado pues tenemos un mayor número de cruces. Se podría decir que el modelo estacionario es capaz de actuar más en profundidad aunque esto depende en cierto modo del azar y el modelo generacional es menos propenso a esto. Unos mismos índices de la generación podrían llegar a cruzarse multitud de veces, esto es más probable que en el modelo generacional donde es más difícil que llegasen a una profundidad así de alta.

A continuación examinaremos el pseudocódigo de los algoritmos meméticos:

def meméticos:

def generacionales:
Copiamos generacion actual en una nueva generacion sobre la que trabajaremos
Mientras iteracion actual < iteracion maxima(15000, puede cambiarse):
Repetimos según la presión establecida(prob de cruce * tamaño poblacion/2):
Guardamos el mejor individuo actual
Cruzamos dos individuos seleccionados con dos torneos binarios
Cruzamos o bien con aritmético o con BLX-α
Sustituimos los padres actuales por los dos nuevos individuos(En la nueva generación)
Si el mejor a sido sustituido:
sobreescrito <- True

Mutamos aleatoriamente el número de cromosomas correspondiente según la probabilidad de mutación
Si sobreescrito:
peorNueva<-mejorActual (En la nueva generación)
Sustituimos actual por nueva generación
Introducimos búsqueda local

Devolvemos la mejor de las soluciones actuales

def búsquedaLocal:
Obtenemos cota de cromosomas
Ponemos la cota de 2 * n características
Mientras no realicemos las vueltas hasta llegar a la cota:
generamos aleatoriamente los genes a modificar
Para todos los genes:
Modificar gene en base a una mutación
Si mejora salimos
Si no mejora continuamos con el siguiente gen y volvemos el gen al estado original

En los algoritmos meméticos introducimos conocimiento en forma de una búsqueda local cada X número de iteraciones donde X en nuestro caso es 10. Como esqueleto para los algoritmos meméticos usamos un algoritmo genético generacional con cruce BLX como explique anteriormente. Al realizar esta modificación al algoritmo genético obtenemos unas mejores soluciones, por tanto podemos determinar que es una gran idea introducir conocimiento a un algoritmo genético. Nuestra búsqueda local funciona de tal manera que modificaremos el número de cromosomas que se determinen en base a la prob que le introducimos. Lo que hará es recorrer los genes en busca de una mejora en la solución en base a una modificación en un gen, una vez que la consigue se sale y comienza una nueva búsqueda local sobre esta nueva solución, así hasta llegar a 2 * n_características * El número de cromosomas escogidos para la BL.
Comentaremos las similitudes y diferencias de estos algoritmos con ellos mismos y con los genéticos anteriores.

5.- Procedimiento seguido para la realización de la práctica

En un inicio comenzamos la práctica estudiando las transparencias vistas en clase, posteriormente haciendo búsquedas exhaustivas por internet y una vez entendidos perfectamente los algoritmos comenzamos la práctica.

La implementación de esta práctica no ha sido difícil, sin embargo el tiempo dedicado a la experimentación ha sido muy elevado debido en parte al gran tamaño de los dataset, en particular el de ozono, esto sumado a que Python no es un lenguaje compilado por lo que su velocidad no es comparable a C por ejemplo han hecho que este tiempo sea muy elevado.

Aún así hay que señalar que hemos conseguido unos tiempo de ejecución comparables a otras implementaciones en C++ e incluso superar a estas, esto se debe a la inclusión de la librería Scikit-Learn la cual se traduce en una velocidad a la hora de ajustar nuestro KNN y de predecir extremadamente alta. Este es el motivo por el que Python es un lenguaje muy conveniente para este tipo de prácticas aunque en un principio pueda parecer lo contrario.

Con un poco de optimización en los bucles y una cuidada implementación se pueden llegar a conseguir tiempos comparables o mejores que implementaciones C++ gracias a esto y sobretodo a Scikit-Learn.

6.- Experimentación y explicación de los resultados

Para la realización de los experimentos se han usados los parámetros establecidos en el guión de prácticas.
Se ha ejecutado sobre los tres datasets obligatorios y se ha realizado un experimento extra que ha revelado ciertas cosas que veremos a continuación.

Ahora vamos a comentar los resultados obtenidos, empecemos con el algoritmo genético generacional con cruce aritmético. Como podemos apreciar en las tablas adjuntas a este archivo, este algoritmo genético barre a los algoritmos vistos con anterioridad (RELIEF y BL), consiguiendo una media de evaluación mucho más alta en todos los dataset. En cuanto a la diferencia con este mismo algoritmo pero en este caso con cruce BLX, tenemos que los resultados son parecidos en el dataset de ozono funciona mejor el cruce aritmético y en el de parkinsons funciona mejor el cruce BLX y en el de heart tenemos un empate prácticamente, de tener que elegir uno elegiría el cruce BLX pues tiene una mayor facilidad para reducir características que el cruce aritmético.

Si comparamos los resultados del generacional con el estacionario, tenemos que el generacional es mejor pero por muy poca diferencia en los dataset de parkinsons y heart pero no consigue batir al estacionario con cruce aritmético en el de ozono, esto probablemente se deba a que el dataset de ozono es más complicado tener una buena tasa de reducción, esto tiene sentido pues si lo pensamos es el que tiene más características y por tanto es más complicado llevar a cada una a cero teniendo en cuenta la naturaleza aleatoria de estos algoritmos.

Computacionalmente ambos son muy costosos y aunque el estacionario no vaya generación a generación ocupa prácticamente el mismo tiempo de ejecución pues al fin y al cabo ambos algoritmos tienen que realizar el tope de evaluaciones establecido en este caso (15000).

Según las pruebas realizadas usando prints, el algoritmo estacionario converge antes que el modelo generacional, por lo que pierde diversidad y debido a esto no es capaz de dar unos resultados claramente mejores que el modelo generacional. Ambos están muy cercanos en cuanto a tasa de evaluación.

En cuanto a los meméticos, tenemos que hemos usado como esqueleto el modelo generacional ya que para poder aplicar búsqueda local necesitamos una población digamos “actual” para tener unos resultados acordes.

Tenemos que el memético que da mejores resultados es en general el AM-(10,1.0) seguido muy de cerca por el AM-(10,0.1) y por último el mejorado no consigue unos resultados tan buenos. La explicación de esto que aparentemente no tiene sentido, es que puede que un individuo en una determinada generación X sea muy bueno pero eventualmente puede

ocurrir que dicho individuo desaparezca en unas cuantas iteraciones más porque sus compañeros comiencen a cruzarse y a obtener unas soluciones más prometedoras que este. En conclusión, ser el mejor en una determinada generación no es garantía de una buena solución. En cuanto al tiempo de ejecución, podemos ver que tienen un tiempo parecido aunque aparentemente pueda dar la impresión de que el 0.1 y el 0.1mej deban de tardar menos. Esto no es así pues igualmente el número de evaluaciones que tienen que hacer es el mismo en los tres, sí que ocurre que mientras que el memético 1.0 hará más evaluaciones en la BL que el 0.1 por ejemplo hará más como genético pues las evaluaciones realizadas en la BL se cuentan para el cómputo global, por tanto al final el 1.0 quitará vueltas como genético para dar más como BL y el 0.1 dará menos como BL y más como genético provocando unos resultados parecidos en los datasets de prueba, a veces es mejor uno y otras otro pero puesto a elegir elegiría el 1.0 pues no tira de aleatorios y puede mejorar una generación en gran medida aunque todo depende del problema al que nos enfrentemos, generalmente nos quedaremos con el que de media sea mejor.

Por último cabe señalar que usamos como base para los meméticos el cruce BLX y al realizar una prueba con CA en el dataset de parkinsons los resultados son bastante mejores que con BLX aunque aparentemente no lo parecía. Por falta de tiempo no puedo experimentar más pero habría que seguir probando para decidir con cual quedarse.