Práctica 2.b.
Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Características

UNIVERSIDAD
DE GRANADA

Curso 2022/23

Jesús Miguel Rojas Gómez. 31015253-Y.
jesusjrg1400@correo.ugr.es
Subgrupo A2 — Martes 17:30-19:30

Contents

1 Descripción del problema

El problema de Aprendizaje de Pesos en Características (APC) se refiere a la selección y ponderación de características relevantes para mejorar la precisión de los modelos de aprendizaje automático. El objetivo es encontrar una combinación óptima de pesos para cada característica que maximice la precisión del modelo en la tarea de predicción. El problema APC está relacionado con el aprendizaje supervisado, en el que se utiliza un conjunto de datos etiquetados para entrenar un modelo y luego hacer predicciones sobre nuevos datos.

Utilizaremos como clasificador la técnica del vecino más cercano. El clasificador 1-NN, o más general, k-NN (del inglés, *k-Nearest Neighbors*) busca determinar la etiqueta de un objeto desconocido basándose en las etiquetas de los objetos más cercanos a él en el espacio de características (sin tener en cuenta al propio objeto). En un clasificador k-NN, el valor de k representa el número de vecinos más cercanos que se tomarán en cuenta para la clasificación del objeto desconocido. Para medir la similitud entre los objetos, el clasificador k-NN utiliza una función de distancia. En nuestro caso, utilizaremos la función distancia euclídea ponderada, que tiene la siguiente expresión:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)},$$

donde e_j^i corresponde a la característica i del dato j, y w_i corresponde al valor del peso que pondera la característica i.

Evaluaremos el modelo combinando dos criterios: precisión y simplicidad:

- Para medir la precisión, utilizaremos el porcentaje de instancias correctamente clasificadas:

$$tasa\_clas = 100 \cdot \frac{\text{nº de instancias bien clasificadas}}{\text{nº total de instancias}}$$

- Para medir la simplicidad, utilizaremos el porcentaje de características descartadas:

$$tasa\_red = 100 \cdot \frac{\text{nº valores } w_i < 0.1}{\text{nº características}}$$

Para combinar ambos criterios, utilizaremos la siguiente función objetivo:

$$tasa\_fitness = \alpha \cdot tasa\_clas + (1 - \alpha) \cdot tasa\_red,$$

donde α es el parámetro que pondera precisión y reducción. En nuestro caso, utilizaremos $\alpha = 0.8$.

A lo largo de las distintas prácticas de la asignatura, estudiaremos e implementaremos diferentes algoritmos para obtener tales combinaciones de pesos, para después realizar una comparación entre ellos.

2 Descripción de la aplicación y algoritmos comúnes

Para realizar la aplicación y la implementación de los algoritmos utilizaremos Python, debido al manejo que hace este lenguaje de arrays y listas.

2.1 Representación de la solución

Utilizaremos arrays de la biblioteca numpy para representar la solución de APC. Concretamente, al ser un peso un elemento que pondera cada característica del conjunto de datos de entrada, la solución será representada por un array de pesos del tamaño del número de características de los datos, siendo la componente i del vector de pesos la ponderación correspondiente a la característica i de los datos de entrada. Cada componente del vector de pesos pertenecerá al intervalo $[0, 1]$.

2.2 Representación de los datos

Utilizaremos de nuevo arrays de numpy para representar los datos de entrada del problema APC. Para cada base de datos, disponemos de cinco ficheros con formato .arff. Dentro de cada fichero, hemos realizado tres divisiones: nombres de las características, datos, siendo estos las características, y clase. Ambos elementos del fichero serán almacenados en arrays de numpy, por separado. Estos, a su vez, se encontrarán en un array junto con los elementos del resto de ficheros, de nuevo por separado. He decidido realizar esta jerarquía para agilizar el proceso de ejecución y el número de llamadas al fichero main; de este modo, solo necesitaremos realizar una llamada por *dataset*. Por lo tanto, tenemos la siguiente estructura:

```
datosi = [ [...], [...], ..., [...] ]
clasesi = [ tipo1, tipo2, ..., tipo1 ]
...
datos_todos = [datos1, datos2, datos3, datos4, datos5]
clases_todos = [clases1, clases2, clases3, clases4, clases5]
```

Para leer los ficheros, he definido una función `leer_datos`, que operaría de la siguiente manera:

```
def leer_datos(ruta_fichero):
    abrimos ruta_fichero
    por cada linea en fichero:
        si no contiene @data la saltamos
        si contiene @atribute, guardamos el nombre del atributo
        si contiene @data: por cada linea sucesiva:
            dividimos la linea en datos y clase, y la almacenamos

    return nombres, datos, clases
```

Esta función toma como parámetros la ruta del fichero de datos y devuelve los nombres, los datos y las clases del fichero, atendiendo a su formato.

Una vez leídos todos los ficheros, se normalizan los datos de los mismos, de modo que toda característica de cualquier dato se encuentra en el intervalo $[0, 1]$. Para ello, he definido otra función `normalizar_datos`, que opera de la siguiente manera:

```
def normalizar_datos(datos):
    min = valor minimo de todas las caracteristicas
    max = valor maximo de todas las caracteristicas
    por cada dato en datos:
        por cada caracteristica en dato:
            caracteristica = (caracteristica-min) / (max-min)

    return datos_normalizados
```

Esta función toma como parámetro todos los datos y devuelve los datos normalizados. Destacar que esta normalización se realiza teniendo en cuenta todos los ficheros de un mismo *dataset*.

2.3 Funciones comunes

La implementación de esta práctica está organizada en tres archivos: `faux.py`, `P1.py` y `P2.py`. El archivo `faux.py` contiene las funciones comunes a los algoritmos implementados, el archivo `P1.py` contiene los algoritmos implementados y el main de la práctica 1, y el archivo `P2.py` contiene los algoritmos implementados y el main de la práctica 2.

Dentro del fichero `faux.py`, encontramos varias funciones. Concretamente, están las funciones ya mencionadas `leer_datos` y `normalizar_datos`, y las funciones necesarias para el calculo de la distancia, del clasificador 1-NN y la función objetivo.

Clasificador 1-NN

Para definir el clasificador necesitaríamos definir una función distancia ponderada, pero gracias al lenguaje que estamos utilizando y al manejo de arrays que realiza, no necesitamos explícitamente definir dicha función. Por lo tanto, tenemos el siguiente pseudocódigo:

```
def clasificador1nn(pesos, dato, datos, clases, indice=-1):
    distancias_sin_sumar = pesos x (datos-dato)^2
    # en este punto tenemos todas las caracteristicas de los datos
        con el resultado de la operacion
    distancias_sumadas = suma por filas de distancias_sin_sumar
    distancias = raiz cuadrada de distancias_sumadas
    si estamos utilizando el conjunto train para predecir la clase
        , implementamos leave-one-out y la distancia en la
        posicion indice la sustituimos por mas infinito

    return de la clase en clases relativa al dato_elegido en datos
        que hace minima la distancia entre dato (parametro) y
        dato_elegido
```

Esta función toma como parámetros un vector de pesos, el dato que queremos clasificar, y los datos y las clases del conjunto donde estamos realizando la comparación. Calcula la distancia ponderada al cuadrado del dato respecto a todos los datos, y devuelve la clase relativa al dato que hace mínima la distancia

anterior.

Esta función ha sido modificada respecto de la práctica 1 ya que presentaba errores. Concretamente, cuando sustituíamos las distancias que eran cero por infinito intentando implementar leave-one-out, sustituíamos más de un valor por infinito ya que había varias distancias que eran cero. Leave-one-out realmente solo se debe implementar cuando estamos entrenando un modelo, por tanto, he añadido un parámetro `indice` inicializado por defecto a -1 para distinguir cuando estamos entrenando y cuando testeando el modelo. Si el parámetro es distinto de su inicialización por defecto (es decir, -1), significa que estamos entrenando el modelo y por tanto debemos aplicar leave-one-out; esto lo hacemos sustituyendo la distancia relativa al dato que estamos comparando por infinito (utilizando el parámetro índice). De este modo, la implementación de esta función ya sería correcta.

Función objetivo

Como he mencionado en la introducción, la función objetivo es la función `tasa_fitness`. Para ello, necesitamos definir antes las funciones relativas a precisión y simplicidad:

```
def tasa_clas(pesos, datos_clasificar, clases_clasificar,
    datos_train, clases_train):
    si estamos entrenando:
        por cada dato,clase en datos_clasificar,clases_clasificar:
            si clasificador1nn(pesos,dato,datos_train,clases_train
                ,indice_dato_a_clasificar) == clase,
                contabilizamos un acierto
    si estamos testeando:
        por cada dato,clase en datos_clasificar,clases_clasificar:
            si clasificador1nn(pesos,dato,datos_train,clases_train
                ) == clase, contabilizamos un acierto

    return 100 x aciertos / no datos clasificar


def tasa_red(pesos):
    return 100 x no pesos con valor menor que 0.1 / n pesos
```

La función `tasa_clas` aplica el `clasificador1nn` que hemos definido previamente a cada dato a clasificar, y compara la clase obtenida con la clase verdadera del propio dato. Una vez hecho esto para cada dato, devuelve el porcentaje de aciertos. La función `tasa_red` devuelve el porcentaje del número de pesos con valor menor que 0.1.

La función `tasa_clas` por consiguiente también ha sufrido cambios de cara a la práctica 2. Como debemos distinguir entre entrenar y testear un modelo, debemos distinguir a la hora de realizar una llamada al clasificador. En este caso, he implementado esta nueva casuística comparando el tamaño del conjunto de datos a clasificar y a entrenar; si son del mismo tamaño, significa que estamos entrenando al modelo, si no, estamos testeando. De este modo la función ya estaría totalmente correcta.

Una vez definidas ambas funciones, podemos definir la función objetivo, realizando una ponderación según un valor alpha, que será fijado en el main a 0.8:

```
def tasa_fitness(alpha, pesos, datos_clasificar, clases_clasificar
    , datos_train, clases_train):
    return alpha x tasa_clas(pesos, datos_clasificar,
        clases_clasificar, datos_train, clases_train) + (1-alpha)
        x tasa_red(pesos)
```

Main

Todo lo relativo a la ejecución de los algoritmos y obtención de los resultados se encuentra en el main del archivo `P2.py`.

En primer lugar, leo los datos de los cincos ficheros, los junto en un mismo array, los normalizo y los vuelvo a dividir. En cuanto a las clases, dependiendo del *dataset* las clases tenian un formato diferente, por lo que decidí transformarlas a un valor numérico, ayudándome en la transformación de un diccionario. Una vez hecho esto, fijamos la semilla y con ayuda del módulo tabulate construimos la tabla con los resultados. Por cada algoritmo, realizamos cinco ejecuciones, cada una con un fichero de test distinto, cronometramos la ejecución y calculamos las métricas respecto al fichero test correspondiente. Una vez realizadas las cinco ejecuciones, imprimimos los resultados y pasamos al siguiente algoritmo.

Para cronometrar cada ejecución hemos utilizado la librería timeit.

3 Descripción de los algoritmos implementados

En total, para esta práctica, hemos implementado tres algoritmos principales: AGG, AGE y AM.

En esta práctica, como clases y funciones comunes a todos los algoritmos tenemos lo siguiente:

- Clase cromosoma. Representa un cromosoma de la población, es decir, un vector de pesos junto con las métricas de evaluación de una solución: clasificación, reducción y fitness.

```
class cromosoma:
    constructor(datos, clases, pesos=[]):
        si pesos == [] inicializamos pesos segun una
            distribucion uniforme
        si pesos != [], this.pesos = copia(pesos)
        this.tclas = tasa clasificacion
        this.tred = tasa reduccion
        this.tfit = tasa fitness
```

  La inicialización de la solución, como podemos ver en el pseudocódigo, la hemos realizado según una distribución uniforme, en el intervalo $[0, 1]$.

- Función cruce BLX. Implementa el operador de cruce BLX de parámetro α. Toma como parámetros dos cromosomas y realiza el cruce devolviendo dos nuevos cromosomas, que dependen de los vectores de pesos de los cromosomas proporcionados.

```
def cruce_BLX(c1, c2, datos, clases):
    cmax = maximo del valor de los pesos de c1 y c2
    cmin = minimo del valor de los pesos de c1 y c2
    a = cmin - (cmax-cmin)*alpha
    b = cmax - (cmax-cmin)*alpha
    h1,h2 = nuevos descendientes con distribucion uniforme en
        (a,b)

    return cromosoma(h1),cromosoma(h2)
```

- Función cruce aritmético. Implementa el operador de cruce aritmético. Toma como parámetros dos cromosomas y obteniendo una ponderación aleatoria realiza el cruce devolviendo dos nuevos cromosomas, que dependen de los pesos de los cromosomas proporcionados.

```
def cruce_aritmetico(c1, c2, datos, clases):
    a = primer valor de ponderacion
    b = segundo valor de ponderacion
    h1 = media ponderada de parametro a de c1 y c2
    h2 = media ponderada de parametro b de c1 y c2

    return cromosoma(h1),cromosoma(h2)
```

- Función torneo binario. Implementa el operador de selección de una población como un torneo binario. Elegimos dos índices aleatorios dentro de una población y devolvemos el elemento con mayor fitness.

```
def torneo_binario(poblacion):
    id1 = indice aleatorio 1
    id2 = indice aleatorio 2

    si poblacion[id1] es mejor que poblacion[id2]:
        return poblacion[id1]
    si no:
        return poblacion[id2]
```

Por otro lado, para poder utilizar los algoritmos meméticos, necesitamos funciones adicionales:

- Función búsqueda local de baja intensidad. Implementa una búsqueda local de baja intensidad, explorando solo $2\dot{n}$ vecinos, siendo n el número de características del dataset.

```
def busqueda_local_bi(datos_train, clases_train, cromosoma):
    mientras iteraciones < 2*n:
        mutamos un indice del cromosoma
        si mejora, sustituimos el cromosoma anterior por el
            actual y repetimos el proceso
```

```
        si no, mutamos otro indice del cromosoma

    return cromosoma_resultado
```

- Función AM-All. Implementa el tipo de algoritmo memético en el que a todos los cromosomas de la población se le aplica la búsqueda local de baja intensidad.

```
def AM_All(poblacion, datos_train, clases_train):
    por cada elemento de la poblacion:
        aplicamos la busqueda local de baja intensidad

    return nueva_poblacion
```

- Función AM-Rand. Implementa el tipo de algoritmo memético en el que al 10% de la población seleccionado aleatoriamente se le aplica la búsqueda local de baja intensidad.

```
def AM_Rand(poblacion, datos_train, clases_train):
    seleccionamos aleatoriamente n indices a aplicar la
        busqueda local, donde n es el numero esperado
    por cada indice seleccionado:
        aplicamos la busqueda local de baja intensidad

    return poblacion
```

- Función AM-Best. Implementa el tipo de algoritmo memético en el que al 10% de la población, seleccionando los mejores, se le aplica la búsqueda local de baja intensidad.

```
def AM_All(poblacion, datos_train, clases_train):
    ordenamos la poblacion por tfit
    a los n primeros elementos de poblacion:
        aplicamos la busqueda local de baja intensidad

    desordenamos de nuevo la poblacion

    return poblacion
```

Una vez definidas las clases y funciones necesarias, podemos describir los principales algoritmos de la práctica.

3.1 AGG. Algoritmo Genético Generacional

Este algoritmo toma como parámetros los datos de train y de test a utilizar, y el operador de cruce, y aplica un algoritmo genético generacional. Principalmente, consta de cuatro etapas, que se repiten hasta llegar a un número de iteraciones:

- Selección: mediante la función torneo binario previamente definida selecciona una nueva población de cromosomas de la población existente (en la primera iteración, esta es inicializada aleatoriamente).

- Cruce: mediante el operador de cruce proporcionado como parámetro, cruza cromosomas aleatoriamente con una probabilidad de 0.7.
- Mutación: muta cromosomas aleatoriamente con una probabilidad de 0.1.
- Elitismo: si de la nueva población el mejor fitness no mejora la población anterior, eliminamos el peor cromosoma de la nueva población y lo sustituimos por el mejor cromosoma de la población anterior.

```
def AGG(datos_train, clases_train, datos_test, clases_test,
    operador_cruce):
    poblacion = poblacion inicial aleatoria

    mientras iteraciones no haya llegado al maximo:
        mejor_id = indice mejor tfit de la poblacion
        # seleccion
        nueva_poblacion = [torneo_binario(poblacion) x tama o
            poblacion ]
        # cruce
        for i in elementos a cruzar:
            h1,h2 = operador_cruce(dos indices aleatorios)
            nueva_poblacion = nueva_poblacion sustituyendo los
                cruces
        # mutacion
        for i in elementos a mutar:
            elegimos un indice sin reemplazamiento
            nueva_poblacion = nueva_poblacion sustituyendo
                mutaciones
        mejor_id_actual = indice mejor tfit nueva_poblacion
        # elitismo
        si tfit de la poblacion antigua < tfit poblacion actual:
            sustituimos el peor elemento de la poblacion actual
                por el mejor de la poblacion antigua

    w_final = poblacion[mejor_id].w
    tclas = tasa_clasificacion de test
    tred = tasa_reducci n de test
    tfit = tasa_fitness de test

    return tclas,tred,tfit
```

Destacar de este algoritmo que los datos de test no se utilizan hasta que ha acabado el número máximo de iteraciones. Para el cuerpo del bucle, únicamente se utilizan los datos de train.

3.2 AGE. Algoritmo Genético Estacionario

Este algoritmo toma como parámetros los mismos que AGG, datos de train y de test, y el operador de cruce a utilizar, y aplica un algoritmo genético estacionario. Al igual que el anterior, principalmente consta de cuatro etapas, que se repiten hasta llegar a un número de iteraciones:

- Selección: mediante la función de torneo binario previamente definida seleccionamos dos padres.
- Cruce: mediante el operador de cruce seleccionado como parámetro, cruzamos los dos padres seleccionados, con una probabilidad de 1.

- Mutación: muta los padres aleatoriamente con una probabilidad de 0.1.
- Reemplazamiento: reemplazamos los dos padres seleccionados por los dos peores cromosomas de la población.

```
def AGE(datos_train, clases_train, datos_test, clases_test,
    operador_cruce):
    poblacion = poblacion inicial aleatoria

    mientras iteraciones no haya llegado al maximo:
        # seleccion
        nuevos_padres = [torneo_binario(poblacion), torneo_binario
            (poblacion)]
        # cruce
        nuevos_padres = nuevos padres cruzados
        # mutacion
        for i in 2:
            mutamos nuevos_padres[i] con probabilidad 0.1
        # reemplazamiento
        reemplazamos nuevos padres por los peores cromosomas de la
             poblacion

    w_final = poblacion[mejor_id].w
    tclas = tasa_clasificacion de test
    tred = tasa_reducci n de test
    tfit = tasa_fitness de test

    return tclas,tred,tfit
```

De nuevo, destacar de este algoritmo que los datos de test no se utilizan hasta que ha acabado el número máximo de iteraciones. Para el cuerpo del bucle, únicamente se utilizan los datos de train. La principal diferencia respecto al algoritmo anterior es que, en este sustituimos solo dos cromosomas de la población en cada generación, y en el anterior sustituimos todos los cromosomas de la población.

3.3 AM. Algoritmo Memético

Este algoritmo toma como parámetros los datos de train y test, el operador de cruce y el tipo de memético, y aplica un algoritmo memético basado en el algoritmo genético generacional. Tiene la misma estructura que AGE, solo que una etapa más: la búsqueda local. Por tanto:

- Selección: mediante la función torneo binario previamente definida selecciona una nueva población de cromosomas de la población existente (en la primera iteración, esta es inicializada aleatoriamente).
- Cruce: mediante el operador de cruce proporcionado como parámetro, cruza cromosomas aleatoriamente con una probabilidad de 0.7.
- Mutación: muta cromosomas aleatoriamente con una probabilidad de 0.1.
- Elitismo: si de la nueva población el mejor fitness no mejora la población anterior, eliminamos el peor cromosoma de la nueva población y lo sustituimos por el mejor cromosoma de la población anterior.

- Búsqueda local: por cada un número determinado de generaciones, se le aplica la búsqueda local de baja intensidad del tipo de memético proporcionado como parámetro.

```
def AM(datos_train, clases_train, datos_test, clases_test,
    operador_cruce, tipo_memetico):
    poblacion = poblacion inicial aleatoria

    mientras iteraciones no haya llegado al maximo:
        mejor_id = indice mejor tfit de la poblacion
        # seleccion
        nueva_poblacion = [torneo_binario(poblacion) x tamano
            poblacion ]
        # cruce
        for i in elementos a cruzar:
            h1,h2 = operador_cruce(dos indices aleatorios)
            nueva_poblacion = nueva_poblacion sustituyendo los
                cruces
        # mutacion
        for i in elementos a mutar:
            elegimos un indice sin reemplazamiento
            nueva_poblacion = nueva_poblacion sustituyendo
                mutaciones
        mejor_id_actual = indice mejor tfit nueva_poblacion
        # elitismo
        si tfit de la poblacion antigua < tfit poblacion actual:
            sustituimos el peor elemento de la poblacion actual
                por el mejor de la poblacion antigua
        # busqueda local
        si generacion % FRECUENCIA_MEMETICO == 0:
            poblacion = tipo_memetico(poblacion, datos_train,
                clases_train)

    w_final = poblacion[mejor_id].w
    tclas = tasa_clasificacion de test
    tred = tasa_reduccion de test
    tfit = tasa_fitness de test

    return tclas,tred,tfit
```

De nuevo, destacar de este algoritmo que los datos de test no se utilizan hasta que ha acabado el número máximo de iteraciones. Para el cuerpo del bucle, únicamente se utilizan los datos de train.

4 Desarrollo de la práctica y manual de usuario

En mi caso, he desarrollado la práctica desde cero apoyándome en lo entregado para la práctica 1, con las modificaciones pertinentes.

Para obtener correctamente los resultados, realizaremos la siguiente ejecución:

```
<interprete-python> P2.py <fichero-1> <fichero-2> <fichero-3> <
    fichero-4> <fichero-5> <semilla>
```

El orden de los ficheros determina el orden en el que aparecen los resultados de cada algoritmo. Si el orden es $[1, 2, 3, 4, 5]$, es decir, el que aparece en el recuadro, entonces en cada tabla, la primera linea corresponderá a la partición 1, la segunda a la partición 2, etc.

Para realizar tal ejecución deberemos tener instalado las librerías numpy, tabulate y timeit.

Para ejecutar los algoritmos, ya sean de la práctica 1 o de la práctica 2, debemos hacer una llamada con cada dataset.

5 Experimentos y análisis de resultados

Del análisis realizado de la práctica 1 concluímos lo siguiente:

- 1-NN es más rápido que el resto debido a su sencillez, pero tiene una tasa de reducción de 0 y esto hace que el fitness sea más bajo en comparación con el resto. La tasa de clasificación es similar al resto.
- Relief es más rápido que Búsqueda Local, pero aproximadamente diez veces más lento que 1-NN. Su tasa de reducción es baja y su tasa de clasificación es comparable a otros algoritmos.
- Búsqueda Local es significativamente más lenta en comparación con los otros algoritmos debido a su constante exploración de entornos de soluciones. Su enfoque en mejorar la función objetivo resulta en una mejor tasa fitness.
- El tiempo de ejecución en 1-NN y Relief es mayor en el dataset diabetes debido al mayor número de datos. El tiempo de ejecución en Búsqueda Local depende del número máximo de vecinos a explorar, que a su vez depende del número de características del dataset, por lo tanto, diabetes es el dataset más rápido, seguido por heart y ozone.
- La variabilidad de los resultados es baja en 1-NN y Relief, ya que solo dependen de la partición de los datos. Sin embargo, en Búsqueda Local hay más variabilidad debido a los fenómenos aleatorios involucrados, como la inicialización de la solución y la elección de mutación.
- En Búsqueda Local, el porcentaje de reducción siempre es mayor, lo cual mejora la tasa fitness. En el dataset diabetes, el porcentaje de reducción es especialmente alto debido a la estabilización de los pesos a valores menores que 0.1, lo que maximiza la función objetivo. En los otros datasets, esto no ocurre debido al mayor número de características.

En esta práctica, al igual que en la anterior, el único hiperparámetro utilizado, que no está indicado en el guión, es la semilla, inicializada a 42. Una vez resumido el análisis de la práctica 1, realizamos el análsis de la práctica 2. Destacar que para obtener los resultados del algoritmo memético hemos utilizado el operador de cruce BLX. A continuación vemos una tabla (Tabla 1) y dos gráficas con los resultados globales de los algoritmos (Figura 2).

	Diabetes			Diabetes	Ozone			Ozone	Spectf-			Spectf-
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
1-NN	68,36	0,00	54,69	0,02	80,31	0,00	64,25	0,01	85,38	0,00	68,30	0,01
RELIEF	67,83	5,00	55,27	0,15	76,87	5,83	62,66	0,05	85,08	0,00	68,06	0,05
BL	66,28	92,50	71,52	207,83	77,50	40,27	70,05	350,51	88,53	43,63	79,55	283,86
AGG-BLX	68,63	47,50	64,41	827,20	78,44	36,94	70,14	351,70	87,37	40,45	77,99	328,33
AGG-ARIT	67,96	57,50	65,87	836,95	80,00	8,06	65,61	330,32	84,52	10,91	69,80	332,15
AGE-BLX	65,24	85,00	69,19	829,48	76,88	37,22	68,94	338,76	87,37	40,45	77,99	328,33
AGE-ARIT	68,61	85,00	71,89	828,69	80,31	31,67	70,58	331,16	85,38	39,09	76,12	341,10
AM-ALL	68,10	67,50	67,98	821,17	77,50	37,78	69,56	350,80	84,23	40,91	75,57	344,61
AM-RAND	67,71	47,50	63,67	829,64	80,63	36,67	71,83	329,79	83,95	37,73	74,71	342,57
AM-BEST	67,85	67,50	67,78	840,83	79,38	39,44	71,39	333,26	83,09	20,00	70,47	342,66

Figure 1: Tabla de los resultados globales obtenidos

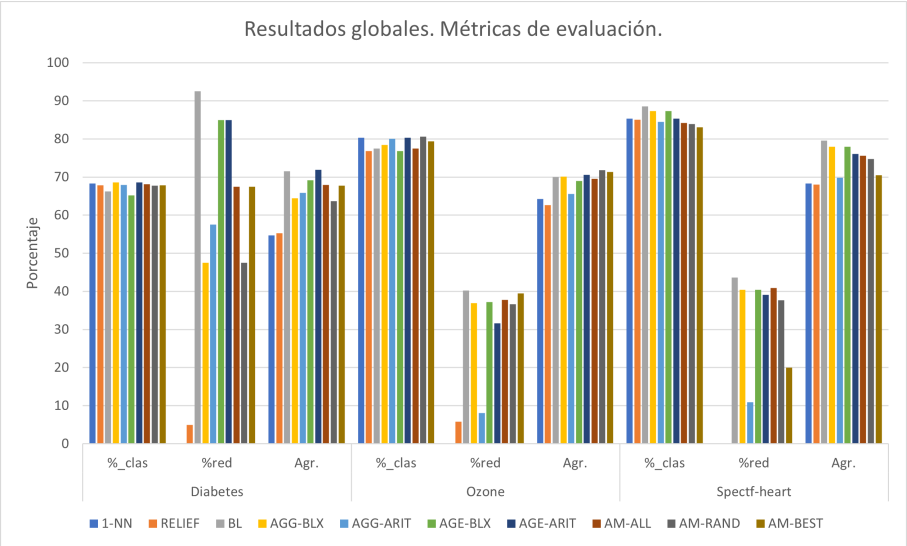
En una primera instancia, podemos apreciar que los tiempos de ejecución de la práctica 1 son en general muchísimo menores que los de la práctica 2. Si nos fijamos en la Figura 2b, los tiempos de esta práctica, en comparación con los de la anterior, en el caso de 1-NN y Relief ni aparecen en el gráfico ya que son órdenes de magnitud menor. En el caso de Búsqueda Local, vemos que en Diabetes el tiempo de ejecución es menor que el de los algoritmos de esta práctica, pero en el resto de datasets el tiempo es similar. Esta casuística en Diabetes se debe a lo ya mencionado anteriormente. Al ser el número de características más bajo que en el resto, el tiempo de ejecución es menor y es por ello por lo que presenta ese comportamiento. Además destacar que el tiempo de ejecución de los algoritmos de esta práctica es bastante similar.

Por otro lado, vemos que el comportamiento analizado en la práctica anterior sigue presente: Diabetes, al tener mayor número de datos, en general el tiempo de ejecución es mayor que en el resto de datasets. Recalco que, esto no ocurre en Búsqueda Local, ya que este algoritmo depende principalmente del número de vecinos a explorar, y en Diabetes es menor.

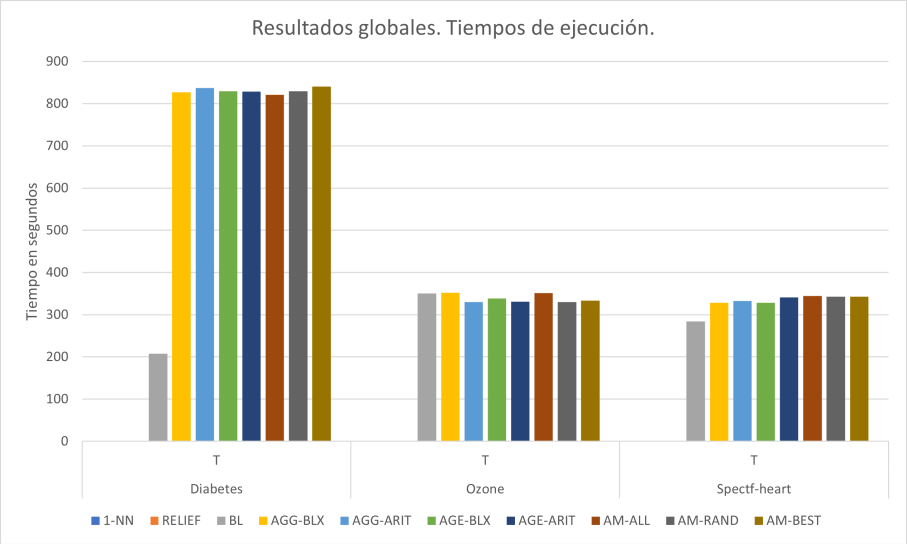
En cuanto a las métricas evaluadas (Figura 2a), vemos que, en general, la tasa de clasificación es bastante similar entre algoritmos, por lo que de nuevo en esta práctica es la tasa de reducción la que principalmente decide el desempeño del algoritmo. Búsqueda Local, como ocurría en la práctica anterior, al ir explorando entornos de soluciones que mejoren la solución actual, siempre presenta mayor tasa de reducción que el resto de algoritmos, por lo que el fitness de dicho algoritmo suele ser de los más altos. Por el contrario, 1-NN y Relief al tener tasas de reducción mucho más bajas, su fitness es en general peor que el del resto. En cuanto a los algoritmos de esta práctica, vemos que en general, dependiendo del dataset unos obtienen mejor desempeño que otros. Más adelante analizaremos dicho comportamiento.

Ahora realizaremos un análisis más en profundidad de los algoritmos de esta práctica. A continuación, en la Tabla 3 podemos ver los resultados obtenidos por los Algoritmos Genéticos Generacionales y en la Figura 4 las gráficas extraídas de dichos resultados.

Con esta comparación estamos intentando apreciar si en este tipo de algoritmos influye realmente el operador de cruce. En primer lugar, destacar que en cuanto a tiempo de ejecución no hay diferencias sustanciales ya que ambos algoritmos toman bastante tiempo en terminar, por lo tanto en términos de tiempo no hay diferencia. En cuanto a la tasa de clasificación, vemos que la diferencia tampoco es demasiado grande como para concluir si un operador



(a) Tasas.



(b) Tiempo.

Figure 2: Gráficas de los resultados globales obtenidos, divididos en tasas de evaluación y tiempo.

es mejor que otro. Dependiendo del dataset, unas veces supera AGG-BLX y otras veces AGG-Arit. En cuanto a la tasa de reducción, vemos que en los tres datasets utilizados la tasa de reducción de AGG-BLX es alta en general y oscila los mismos valores, y la de AGG-Arit dependiendo del dataset varía. Por tanto, podemos deducir que en este caso si influye el operador de cruce, y que AGG-BLX en general tiende a encontrar pesos menores que 0.1 haciendo que mejore el fitness global.

	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	66,88	50,00	63,51	825,29	79,69	43,06	72,36	351,67	91,43	40,91	81,32	329,04
Partición 2	68,18	50,00	64,55	824,49	78,13	37,50	70,00	351,95	87,14	47,73	79,26	327,99
Partición 3	68,18	50,00	64,55	824,24	79,69	30,56	69,86	354,90	92,86	36,36	81,56	329,10
Partición 4	66,88	37,50	61,01	823,77	70,31	37,50	63,75	350,75	85,71	38,64	76,30	327,79
Partición 5	73,03	50,00	68,42	838,22	84,38	36,11	74,72	349,24	79,71	38,64	71,50	327,73
Media	68,63	47,50	64,41	827,20	78,44	36,94	70,14	351,70	87,37	40,45	77,99	328,33

(a) Tabla de los resultados de AGG-BLX.

	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	70,78	75,00	71,62	829,74	79,69	12,50	66,25	329,27	82,86	2,27	66,74	325,81
Partición 2	66,88	62,50	66,01	830,41	81,25	5,56	66,11	332,31	87,14	11,36	71,99	344,85
Partición 3	69,48	37,50	63,08	828,90	81,25	11,11	67,22	328,66	90,00	4,55	72,91	334,88
Partición 4	66,23	50,00	62,99	852,06	76,56	4,17	62,08	332,52	81,43	22,73	69,69	327,07
Partición 5	66,45	62,50	65,66	843,65	81,25	6,94	66,39	328,82	81,16	13,64	67,65	328,12
Media	67,96	57,50	65,87	836,95	80,00	8,06	65,61	330,32	84,52	10,91	69,80	332,15

(b) Tabla de los resultados de AGG-Arit.

Figure 3: Resultados de los Algoritmos Genéticos Generacionales.

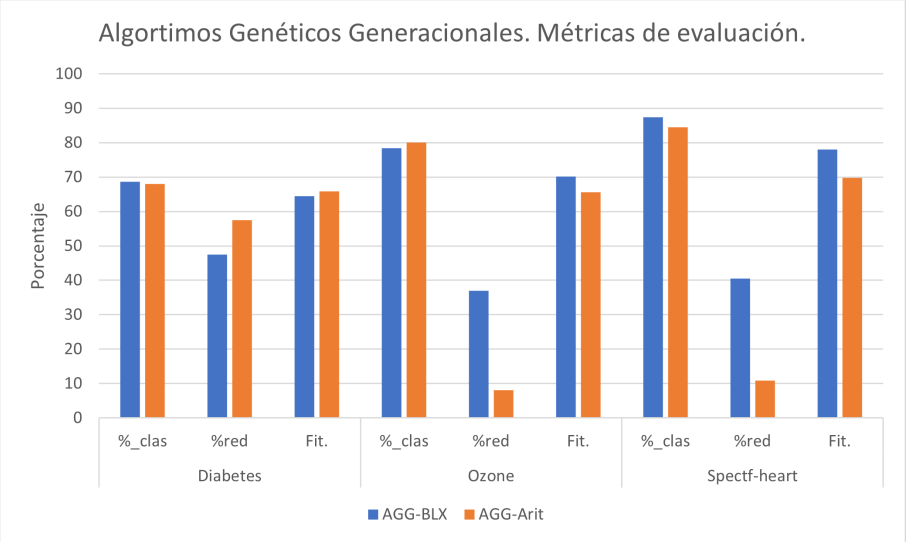
Veamos que ocurre con los Algoritmos Genéticos Estacionarios. A continuación, en la Tabla 5 podemos ver los resultados obtenidos por los Algoritmos Genéticos Generacionales y en la Figura 6 las gráficas extraídas de dichos resultados.

En primer lugar, destacar de nuevo que en cuanto a tiempo de ejecución no hay diferencias. En cuanto a la tasa de clasificación, vemos que otra vez la diferencia tampoco es demasiado grande como para concluir si un operador es mejor que otro, depende del dataset. En cuanto a la tasa de reducción, vemos algo distinto a lo anterior. En este caso, la tasa de reducción es en general alta tanto si se trata de AGE-BLX como si se trata de AGE-Arit. Por lo tanto, en este caso no podemos concluir que un operador es mejor que otro, ya que en realidad en todas las métricas las diferencias no son significativas, y como todas dependen de una componente estocástica pues probablemente dependan más de dicha componente que del propio operador de cruce.

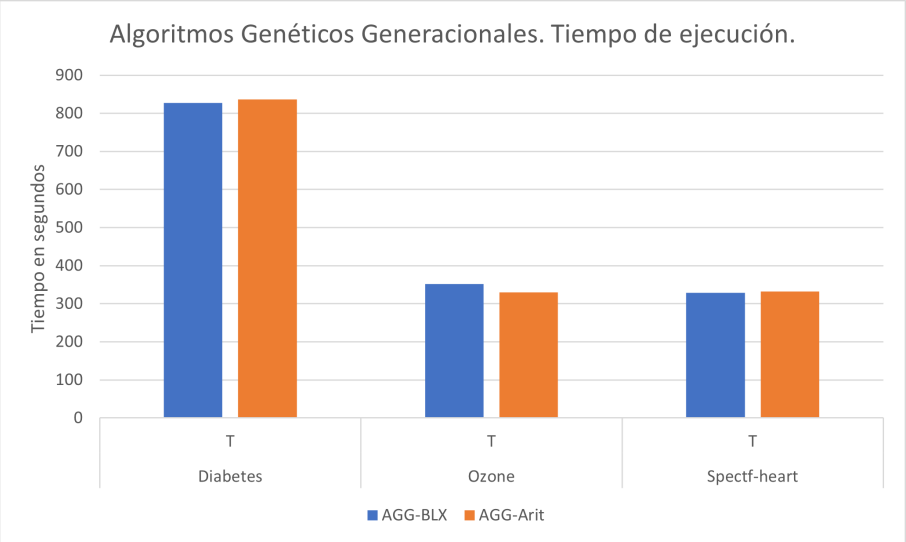
Realicemos la comparación ahora entre Algoritmos Genéticos. En la Figura 7 podemos ver los resultados acoplados en un mismo gráfico.

De nuevo, en tiempo no hay diferencias. En cuanto a porcentaje de clasificación, tampoco hay demasiada diferencia; dependiendo del dataset, unas veces son mejores los generacionales y otras veces los estacionarios. En cuanto a tasa de reducción, en general los algoritmos estacionarios tienen igual o mejor tasa que los algoritmos generacionales, aunque luego esto no se vea demasiado afectado en el fitness. Por tanto, no hay diferencias sustanciales para concluir si unos son mejores que otros. Podríamos destacar que los estacionarios son sutilmente mejores debido a la tasa de reducción, aunque de nuevo, al depender los algoritmos de una componente aleatoria, no es suficiente para decir que son mejores.

Analicemos ahora los resultados de los Algoritmos Meméticos. En la Tabla 8 podemos ver los resultados obtenidos y en la Figura 9 los gráficos extraídos.



(a) Tasas.



(b) Tiempo.

Figure 4: Gráficas de los Algoritmos Genéticos Generacionales.

En cuanto a tiempo de ejecución, no hay diferencias. Destacar que en Diabetes, que es el dataset con mayor número de datos, el tiempo de ejecución de AM-Best es mayor. Esto se debe a que el algoritmo, cada 10 generaciones, implica una ordenación del tamaño de la población. En nuestros datasets no es significativa la diferencia pero es algo a considerar a mayor número de datos y mayor tamaño de población. En cuanto a porcentaje de clasificación no hay diferencias significativas. En cuanto a porcentaje de reducción no hay una tendencia clara que poder concluir. Sería lógico pensar que AM-All y AM-Best

	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	65,58	87,50	69,97	824,48	76,56	38,89	69,03	334,10	91,43	40,91	81,32	329,04
Partición 2	59,74	100,00	67,79	823,27	84,38	34,72	74,44	330,18	87,14	47,73	79,26	327,99
Partición 3	69,48	75,00	70,58	830,90	78,13	40,28	70,56	351,65	92,86	36,36	81,56	329,10
Partición 4	65,58	75,00	67,47	835,45	68,75	38,89	62,78	344,07	85,71	38,64	76,30	327,79
Partición 5	65,79	87,50	70,13	833,28	76,56	33,33	67,92	333,78	79,71	38,64	71,50	327,73
Media	65,24	85,00	69,19	829,48	76,88	37,22	68,94	338,76	87,37	40,45	77,99	328,33

(a) Tabla de los resultados de AGE-BLX.

	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	67,53	87,50	71,53	823,47	75,00	27,78	65,56	329,66	82,86	43,18	74,92	327,58
Partición 2	69,48	87,50	73,08	822,66	84,38	34,72	74,44	333,85	88,57	40,91	79,04	342,02
Partición 3	72,08	87,50	75,16	833,28	81,25	36,11	72,22	329,52	87,14	40,91	77,90	347,04
Partición 4	67,53	87,50	71,53	828,03	78,13	25,00	67,50	333,53	85,71	38,64	76,30	344,23
Partición 5	66,45	75,00	68,16	836,00	82,81	34,72	73,19	329,22	82,61	31,82	72,45	344,64
Media	68,61	85,00	71,89	828,69	80,31	31,67	70,58	331,16	85,38	39,09	76,12	341,10

(b) Tabla de los resultados de AGE-Arit.

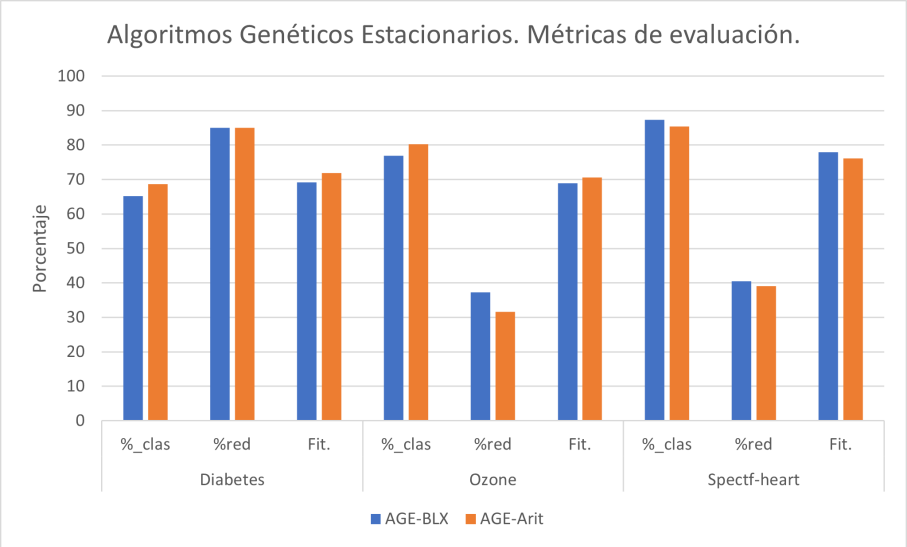
Figure 5: Resultados de los Algoritmos Genéticos Estacionarios.

obtuviesen la misma tasa de reducción ya que al final los peores cromosomas serán descartados, y esto ocurre en los dos primeros datasets, pero no en el último. En cuanto al fitness, destacar que tampoco podemos concluir nada. Cabría esperar que AM-All y AM-Best obtuviesen siempre mejores resultados que AM-Rand pero esto no ocurre ni en el segundo dataset ni en el tercero. Por tanto, en general, podemos concluir que es indiferente utilizar un algoritmo u otro según los resultados obtenidos. La lógica nos llevaría a escoger o AM-All o AM-Best ya que se seleccionan los cromosomas a los que aplicar la búsqueda local con un criterio, pero con los resultados obtenidos no podemos concluir nada.

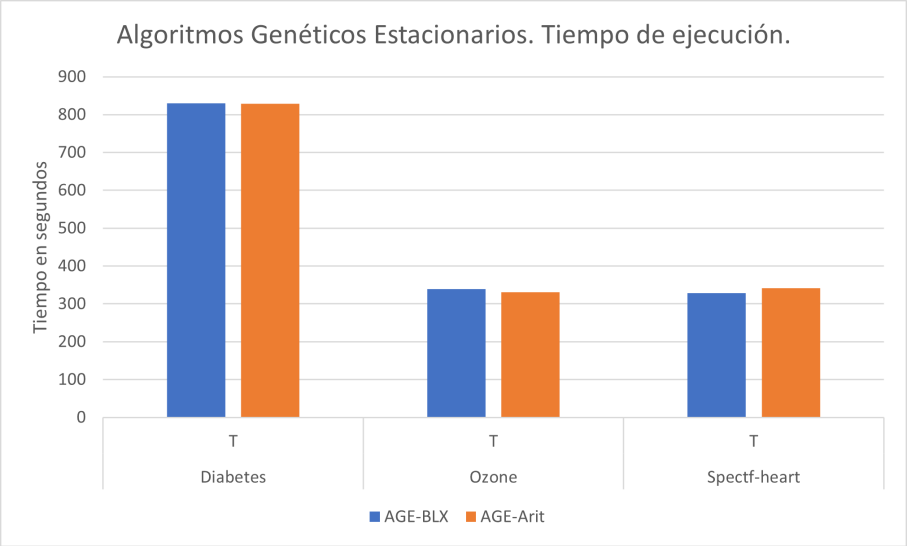
Por último, veamos las gráficas de los resultados de los algoritmos de esta práctica. En la Figura 10 podemos encontrar dicha información.

Cabría esperar que los algoritmos meméticos fuesen mejores que los genéticos, ya que en ellos aplicamos una búsqueda local para mejorar la población. Sin embargo, vemos en los resultados que esto no ocurre, ya que la búsqueda local aplicada no es demasiado significativa al terminar cuando hemos explorado $2n$ vecinos consecutivos sin mejora. Con dicho valor más alto, podríamos esperar obtener mejores resultados, aunque deberíamos analizar dicho caso ya que al aplicar búsqueda local estamos consumiendo iteraciones del algoritmo. Según los algoritmos aplicados y los resultados obtenidos, no hay diferencias sustanciales como para decantarnos por algún tipo; como vengo señalando a lo largo de todo el análisis, son algoritmos que dependen de eventos aleatorios y del dataset.

Lo que si podemos concluir es que los algoritmos de esta práctica son mucho más costosos y sin embargo no aportan mejoras significativas en el fitness, por lo que a quedarnos con un tipo de algoritmos, elegiríamos los de la práctica 1.

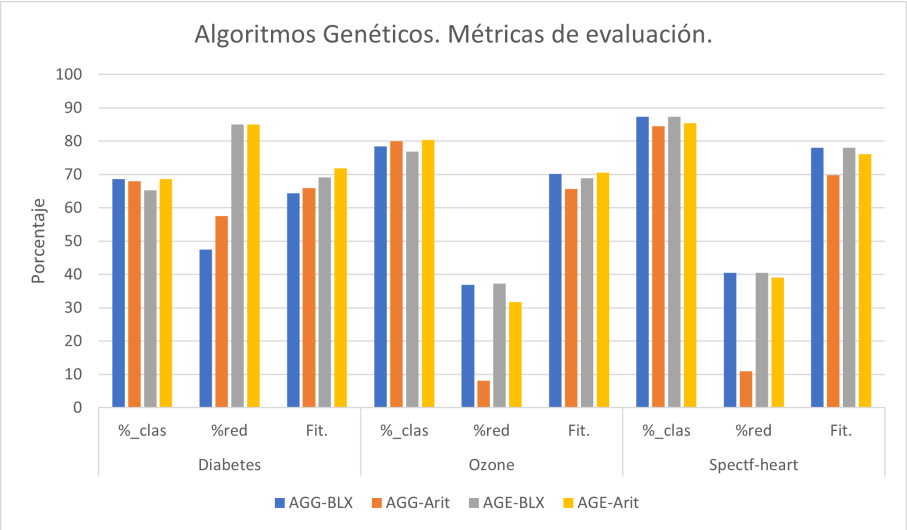


(a) Tasas.

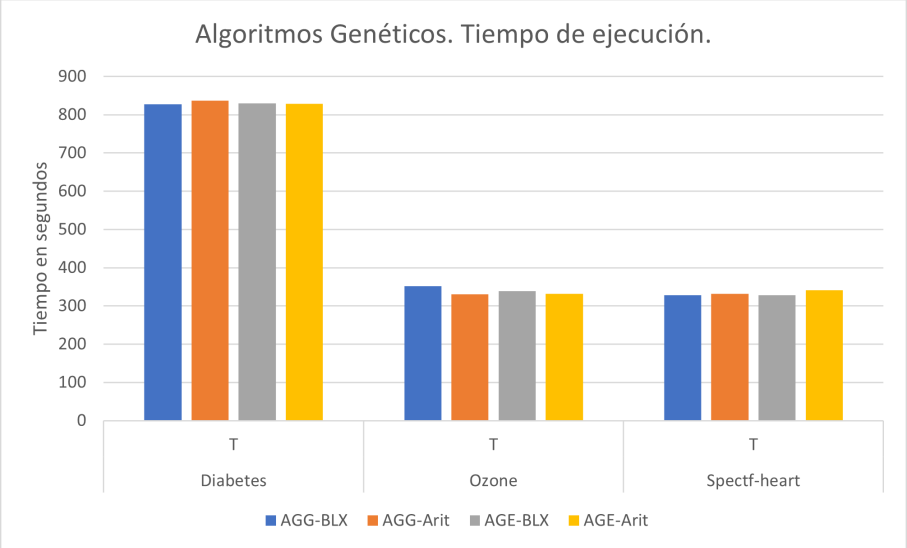


(b) Tiempo.

Figure 6: Gráficas de los Algoritmos Genéticos Estacionarios.



(a) Tasas.



(b) Tiempo.

Figure 7: Gráficas de los Algoritmos Genéticos.

	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	66,88	50,00	63,51	815,45	78,13	34,72	69,44	349,73	78,57	45,45	71,95	349,68
Partición 2	68,18	62,50	67,05	823,90	76,56	45,83	70,42	345,53	84,29	38,64	75,16	341,52
Partición 3	66,88	75,00	68,51	823,01	78,13	37,50	70,00	349,95	90,00	40,91	80,18	348,83
Partición 4	70,13	100,00	76,10	819,73	71,88	36,11	64,72	353,64	87,14	38,64	77,44	340,22
Partición 5	68,42	50,00	64,74	823,75	82,81	34,72	73,19	355,17	81,16	40,91	73,11	342,82
Media	68,10	67,50	67,98	821,17	77,50	37,78	69,56	350,80	84,23	40,91	75,57	344,61

(a) Tabla de los resultados de AM-All.

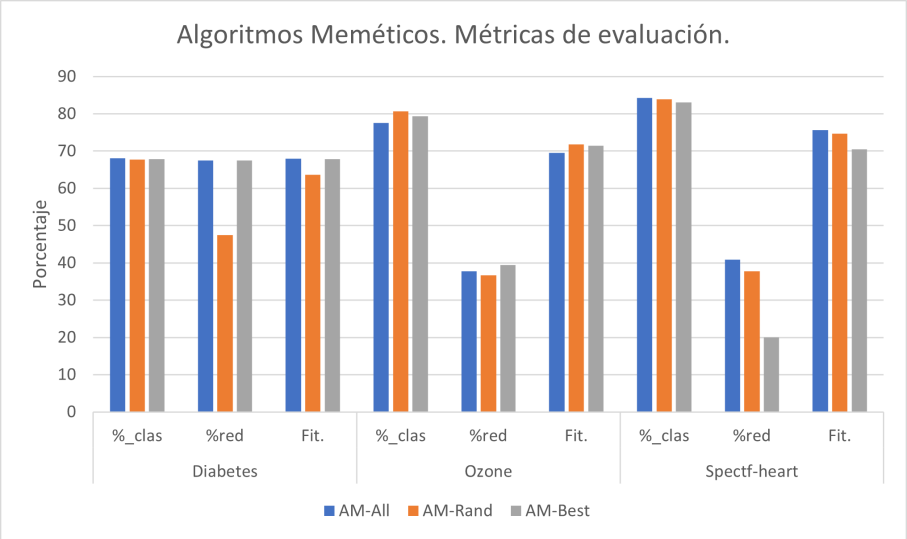
	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	65,58	62,50	64,97	834,26	81,25	38,89	72,78	328,26	82,86	40,91	74,47	337,66
Partición 2	66,88	50,00	63,51	828,73	79,69	30,56	69,86	331,98	82,86	31,82	72,65	344,66
Partición 3	70,13	37,50	63,60	828,04	84,38	37,50	75,00	328,37	87,14	36,36	76,99	343,63
Partición 4	67,53	50,00	64,03	826,54	76,56	36,11	68,47	331,91	84,29	36,36	74,70	347,81
Partición 5	68,42	37,50	62,24	830,62	81,25	40,28	73,06	328,41	82,61	43,18	74,72	339,08
Media	67,71	47,50	63,67	829,64	80,63	36,67	71,83	329,79	83,95	37,73	74,71	342,57

(b) Tabla de los resultados de AM-Rand.

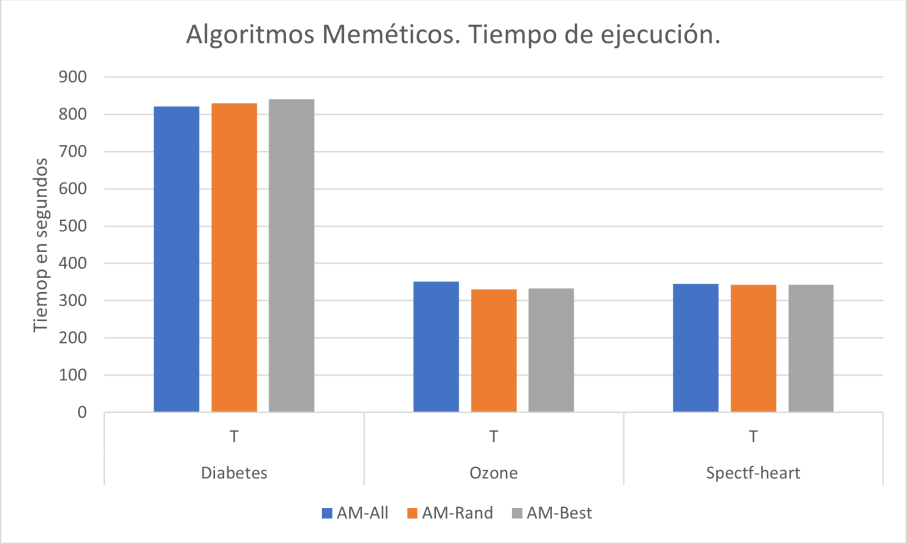
	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	68,83	75,00	70,06	837,39	76,56	37,50	68,75	331,89	74,29	22,73	63,97	346,03
Partición 2	65,58	75,00	67,47	828,89	82,81	40,28	74,31	328,29	85,71	29,55	74,48	337,57
Partición 3	64,94	75,00	66,95	830,02	81,25	41,67	73,33	331,91	90,00	27,27	77,45	346,51
Partición 4	68,18	50,00	64,55	838,71	71,88	45,83	66,67	328,46	84,29	4,55	68,34	341,38
Partición 5	71,71	62,50	69,87	869,15	84,38	31,94	73,89	345,73	81,16	15,91	68,11	341,81
Media	67,85	67,50	67,78	840,83	79,38	39,44	71,39	333,26	83,09	20,00	70,47	342,66

(c) Tabla de los resultados de AM-Best.

Figure 8: Resultados de los Algoritmos Meméticos.



(a) Tasas.



(b) Tiempo.

Figure 9: Gráficas de los Algoritmos Meméticos.

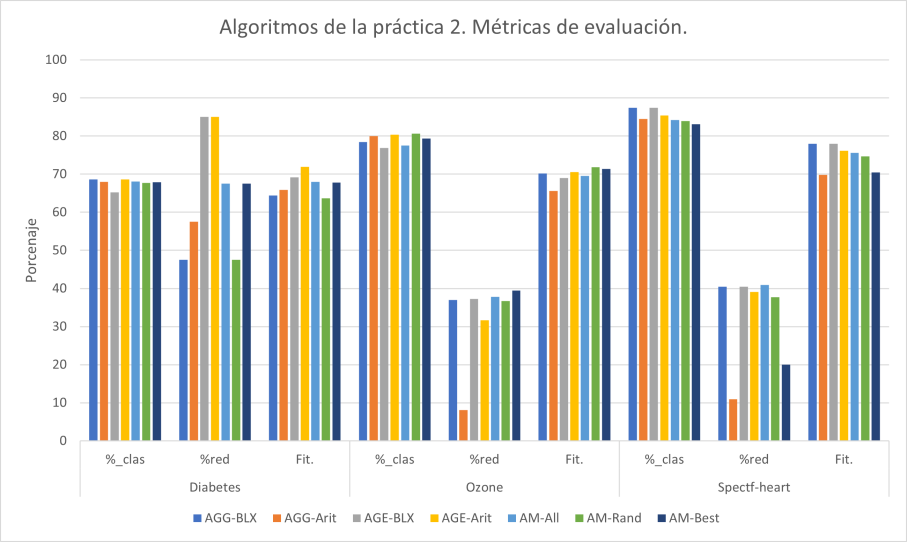


Figure 10: Gráficas de los algoritmos de la práctica 2.