Aprendizaje de Pesos en Características

Metaheurísticas: Práctica 3
Titulo
Grupo 1
M:17.30-19.30

Luis Antonio Ortega Andrés
76425628D
ludvins@correo.ugr.es

29 de mayo de 2019

ÍNDICE

1 NOTAS SOBRE PSEUDOCÓDIGO.

El lenguaje de programación utilizado en la práctica es `Rust`, para el pseudocódigo utilizaré una versión simplificada de este. Por ejemplo la siguiente cabecera de función, acepta una referencia a un vector del tipo genérico `T`, una referencia a un vector de flotantes y un booleano, además devuelve un elemento del tipo `Results`.

```rust
pub fn foo<T: Clone + Copy>(
    foo1: &Vec<T>,
    foo2: &Vec<f32>,
    foo3: bool,
) -> Results
```

El pseudocódigo correspondiente será el siguiente, donde se ha simplificado todo lo correspondiente al tipo genérico y las referencias.

```rust
fn foo(
    foo1: Vec<Datos>,
    foo2: Vec<f32>,
    foo3: bool,
) -> Results
```

Notemos que al indicar los tipos de una variable lo haré en el siguiente formato `var: Tipo`, también se indicará el tipo que devuelve una función con `->tipo`, despues de los parámetros. Además a lo largo de la memoria, se hará referencia a tipos de la forma `u32` y `f32`, donde el primero corresponde a un entero sin signo con 32 bits, y el segundo a un flotante de 32 bits.

2 DESCRIPCIÓN DEL PROBLEMA.

Dado un número natural $n \in \mathbb{N}$ y un conjunto de clases C, un **clasificador** es una aplicación $f : \mathbb{R}^n \to C$ que asigna a cada punto de $\mathbb{R}^n$ (vector de atributos) una clase de C.
El problema de clasificación consiste en, dado un conjunto de datos ya clasificados, obtener un clasificador que permita clasificar otros datos.
El método **k-NN**, es un método de clasificación supervisada, que asigna a cada elemento a clasificar $v \in \mathbb{R}^n$, la clase que mas se repita de entre los k vecinos mas cercanos a v.
Para nuestro problema consideramos $k = 1$, y la distancia euclídea *ponderada con un vector de pesos*.
Para el problema de clasificación tendremos en cuenta dos características sobre el vector de pesos:

- **Tasa de clasificación:** Nos indica como de bueno es el clasificador midiendo la proporción de elementos que se clasifican correctamente.
- **Tasa de reducción:** Las características con un peso menor en el vector de pesos serán menos importantes, la tasa de reducción nos dice que proporción de pesos tiene poca importancia (menor a 0.2).

El aprendizaje consistirá en encontrar el vector de pesos que maximice ambas, es decir, encontrar $\omega \in [0,1]^n$ que maximice:

$$F(\omega) = \alpha T_{class}(\omega) + (1 - \alpha) T_{red}(\omega)$$

Para cierto $\alpha \in [0,1]$, en nuestro caso $\alpha = 0{,}5$.

3 descripción de la aplicación de los algoritmos (práctica 1).

En este apartado se describen las consideraciones, tipos y operaciones comunes a los algoritmos de la práctica.

La primera consideración que hay que tener es que los archivos `.arff` han sido sustituidos por archivos `.csv` equivalentes, donde los datos ya se encuentran normalizados por columnas, además, se ha considerado que si en el archivo de datos existen dos elementos con los mismos atributos, estos son distintos, para ello se ha usado un identificador (el orden de aparición en el fichero).

Un elemento del `.csv` se encuentran encapsulado en una estructura genérica `Data`, esta estructura obliga al dato a tener un identificador `u32`, un vector de atributos `Vec<f32>` y una clase a la que pertenece `u32`. En uno de los conjuntos de dato, la clase no correspondía a un valor numérico, sino a una letra, esto se ha modificado en el `.csv` por un valor entero (0 y 1). Además permite calcular la distancia euclídea del elemento a otro dado.

Para cada conjunto de datos específico, existe una implementación de dicha estructura que se ajusta a la cantidad de atributos (tener el vector de atributos como un array de tamaño fijo aumenta la eficiencia del programa, por ello se ha optado por esta implementación en lugar de utilizar un vector).

Antes de llamar a los algoritmos se realizan las particiones del conjunto de datos en una función auxiliar, estas particiones se usarán para todas las ejecuciones de ese conjunto de datos.

```
fn make_partitions(data: Vec<Datos>, folds: u32) -> Vec<Vec<Datos>> {
    categories_count = new HashMap();
    partitions = new Vec<Vec<Datos>>;

    for i in 0..folds {
        partitions.push(new Vec());
    }
    for example in data {
        counter = categories_count.entry(example.class).or_insert(0);
        partitions[*counter].push(example);

        counter = (counter + 1) % folds;
    }

    return partitions;
}
```

La función consiste en lo siguiente, para cada clase del conjunto de datos existe una entrada en el Hash, esa entrada indica en que partición se ha de insertar el siguiente valor de dicha clase. Cada vez que se inserte un elemento de una clase en una partición se aumenta el valor del Hash en 1 módulo el número de particiones. De esta forma los elementos de la misma clase se van repartiendo entre las particiones.

Todos los algoritmos de la práctica devuelven un vector de pesos de longitud el número de atributos. Este vector de pesos se le pasa luego al clasificador que devuelve un objeto de la estructura `Results`, que encapsula el número de aciertos, el número de pesos de baja importancia y los tamaños para poder calcular tanto la tasa de reducción como la tasa de clasificación, por ello, la función objetivo se calcula en esta estructura tal y como se ha indicado en la descripción del problema.

La funcion de clasificación toma dos vectores de datos `knowledge` correspondiente a aquellos datos que sabemos su clasificación y `exam` correspondiente a aquellos que se quiere clasificar, además acepta el vector de pesos `weights` y un valor booleano `discard_low_weights` que nos permite elegir si queremos que se descarten los pesos menores a 0,2 al calcular la distancia. Se podría haber optado por hacer que el clasificador aceptara un solo elemento a clasificar en lugar de todo un vector, pero como en nuestra práctica siempre se utiliza un vector me ha parecido mas conveniente esta op-

ción. En todo caso siempre se podria pasar un vector con un solo elemento o cambiar la implementación.

En la misma función `classifier_1nn` se calcula si la respuesta dada es la correcta o no, de esta forma se evita que el clasificador tenga que devolver un vector con las respuestas y luego calcular el número de acertos, esto lo podemos hacer porque disponemos de las clases de los elementos que queremos clasificar, en caso de no disponer de ellas devolveriamos las respuestas dadas por el clasificador.

Veamos como funciona el clasificador, lo primero que hacemos es inicializar el número de respuestas correctas a 0 y recorremos el vector de elementos a clasificar.

```
pub fn classifier_1nn(
    knowledge: Vec<Datos>,
    exam: Vec<Datos>,
    weights: Vec<f32>,
    discard_low_weights: bool,
) -> Results {
    correct: u32 = 0;
    for test in exam {
        ...
    }
```

En cada iteración buscamos el elemento mas cercano de entre los conocidos.

```
nearest_example = new Datos;
min_distance = MAX;

for known in knowledge {
    if known.id == test.id {
        continue;
    }
    distance = Calculate_distance();

    if distance < min_distance {
        min_distance = distance;
        nearest_example = known;
    }
}
```

Veamos como calculamos la distancia, aquí he añadido la funcionalidad de optar a no descartar los pesos, veremos mas tarde la razón.

```
distance = 0.0;
for index in 0..weights.len{
    if !discard_low_weights || weights[index] >= 0.2 {
           distance += weights[index]
           * (test.attr(index) - known.attr(index))
           * (test.attr(index) - known.attr(index))
    }
}
distance.sqrt();
```

Despues comprobamos si la respuesta es la correcta y devolvemos los resultados.

```
if nearest_example.class == test.class {
      correct += 1;
}
return new Results(weights, correct, exam.len);
```

4 descripción de la aplicación de los algoritmos (práctica 2)

En esta sección como en la anterior describiré los operadores comunes a los algoritmos utilizados en la práctica 2, además de la estructura de datos utilizada para encapsular un cromosoma.

Un cromosoma se compone de un vector de pesos y un valor flotante correspondiente al valor de la función fitness (tasa de agregado), como en la práctica se ha creado un tipo de dato genérico para los conjuntos de datos y no quería que el cromosoma dependiera de ese conjunto de datos, se puede crear un cromosoma con valor fitness `-1`, que indica que no ha sido evaluado.

Se han sobrecargado los operadores de comparación necesarios para poder mantener la población ordenada, hay que tener en cuenta que en `rust`, la ordenación por defecto es de menor a mayor, por lo que el mejor cromosoma de la generación será el último.

La estructura del cromosoma se encuentra declarada en el fichero `structs.rs`.

Veremos ahora los distintos operadores y funciones utilizadas, las separaremos según su finalidad.

4.1 *Función objetivo*

Su utilidad es, dado un cromosoma y el conjunto de entrenamiento, evaluar su puntuación.

```
fn fitness_function(training: Vec<Data>, chromosome: Chromosome) {
    chromosome.result =
        classifier_1nn(training, training, chromosome.weights).evaluation_function();
}
```

4.2 *Generar población inicial.*

Aquí se consideran dos opciones, la primera de ellas genera cromosomas aleatorios utilizando una distribución uniforme, los clasifica y ordena la población.

```
pub fn initial_generation(
    generation_size: u8,
    n_attrs: u8,
    training: Vec<Data>,
    rng: Rng,
) -> Vec<Chromosome> {
    generation = new Vec<Chromosome>();
    for _ in 0..generation_size {
        weights = [0.0; n_attrs];
        uniform = new Uniform(0.0, 1.0);
        for attr in 0..n_attrs {
            weights[attr] += uniform.sample(rng);
        }
        res = classifier_1nn(training, training, weights);
        generation.push(new Chromosome(weights, res.evaluation_function()));
    }
    generation.sort();
    return generation;
}
```

El otro generador se trata de una variante de este, en el que despues de hacer la población aleatoria, insertamos en ella los pesos que nos devuelve `RELIEF` y el otro algoritmo greedy realizado en la práctica anterior (eliminamos los dos peores pesos generados aleatoriamente).

```
generation.remove(0);
generation.remove(0);
w = calculate_relief_weights(training, n_attrs);
res = classifier_1nn(training, training, w);
generation.push(new Chromosome(w, res.evaluation_function()));
w = alter_greedy_weights(training, n_attrs);
res = classifier_1nn(training, training, w);
generation.push(new Chromosome(w, res.evaluation_function()));

generation.sort();
```

4.3 *Operadores de selección.*

Veamos los distintos operadores de selección que he considerado en esta práctica. El primero de ellos se trata del torneo binario.

Le pasamos como argumento la población y cuántos cromosomas tiene que seleccionar.

```
fn binary_tournament(
    generation: Vec<Chromosome>,
    select_n: u8,
    rng: Rng,
) -> Vec<Chromosome> {
    ret = new Vec<Chromosome>;
    for _ in 0..select_n {
        ret.push(compite(
            generation[rng.gen_range(0, generation.len())],
            generation[rng.gen_range(0, generation.len())],
        ));
    }
    return ret;
}
```

Luego selecciona parejas de elementos y se queda con el mejor de ellos. También he considerado otro algoritmo de selección basado en aumentar la probabilidad de seleccionar aquellos elementos de la población que son mejores.

```
fn weighted_selection(
    generation: Vec<Chromosome>,
    select_n: usize,
    rng: Rng,
) -> Vec<Chromosome> {
    ret = new Vec<Chromosome>;
    uniform = new Uniform(0.0, 1.0);
    total_sum = generation.map(|x| x.result).sum();
```

Lo primero que hacemos es declarar el vector que vamos a devolver (`ret`), y calcular la suma total de todos los valores de la función fitness de nuestra población.

Luego declaramos un vector de flotantes, a cada elemento de la población le corresponderá uno. Cada cromosoma tendrá un valor asignado igual al valor acumulado de la función fitness entre el valor total.

```
weights = new Vec<f32>;
acumulative = 0.0;
for chromosome in generation {
    acumulative = acumulative + chromosome.result / total_sum;
    weights.push(acumulative);
}
```

Ya tenemos inicializado un vector de flotantes entre 0 y 1, creciente, donde la distancia entre los elementos va aumentando (ya que como la población está ordenada, a lo largo de esta la función fitness aumenta).

Ahora generamos un número aleatorio entre 0 y 1, y nos quedamos con aquel cromosoma cuyo peso se quede justo por encima.

```
for _ in 0..select_n {
    random = uniform.sample(rng);
    for i in 0..weights.len() {
        if random < weights[i] {
            let parent = generation.get(i);
            ret.push(parent1);
            break;
        }
    }
}
```

Pongamos un caso de ejemplo, supongamos que en la población tenemos 2 elementos, el primero con un valor fitness de 0.5 y el segundo de 1.0. La suma total sería 1.5 y tendrían asignados los pesos 0.33 y 1 respectivamente. De forma que es más probable que el segundo sea elegido.

En el análisis de resultados discutiremos como ha funcionado este operador de selección.

4.4 *Operadores de cruce*

En la práctica se nos pedía realizar dos operadores de cruces distintos. el primero de ellos es una media ponderada de los pesos de los padres. En un principio se debia calcular para cada peso el punto medio de los de sus padres, sin embargo esto resulta en que para cada 2 padres se genera un solo hijo (no queremos repetirlos), para no tener que pasar mas valores a las funciones he considerado mejor retocar este operador y que en lugar de devolver 1 hijo con la media de los valores, devuelva 2 con una media ponderada. El primero se parecerá mas a un padre y el segundo a otro.

```
fn aritmethic_cross(
    parents: Vec<Chromosome>,
    n_childs: u8,
    n_attrs: u8,
    _rng: Rng,
) -> Vec<Vec<f32>> {

    children = new Vec<Vec<f32>>;

    for _ in 0..(n_childs / 2) {
        parent2 = parents.pop();
        parent1 = parents.pop();
        weights1 = [0.0; n_attrs];
        weights2 = weights1;

        for i in 0..n_attrs {
            weights1[i] += parent1.weights[i] * 0.4 + parent2.weights[i] * 0.6;
            weights2[i] += parent1.weights[i] * 0.6 + parent2.weights[i] * 0.4;
        }
        children.push(weights1);
        children.push(weights2);
    }
    return children;
}
```

Vemos que el operador acepta como parámetros el conjunto de padres, el número de hijos que tiene que generar y el número de atributos.

Cogemos dos padres del vector, sacandolos de este ya que no los vamos a volver a utilizar y para cada atributo vamos haciendo la suma ponderada.

El otro operador considerado es el BLX-α. Veamos su funcionamiento en detalle.

El operador empieza haciendo lo mismo que el cruce aritmético, cogiendo dos elementos del vector de padres. Y comenzamos un bucle sobre los atributos de estos.

```
fn blx_alpha_cross(
    parents: Vec<Chromosome>,
    n_childs: u8,
    n_attrs: u8,
    rng: Rng,
) -> Vec<Vec<f32>> {
    alpha = 0.3;
    children = new Vec<Vec<f32>>;
    for _ in 0..(n_childs / 2) {
        parent2 = parents.pop();
        parent1 = parents.pop();
        weights1 = vec![0.0; n_attrs];
        weights2 = vec![0.0; n_attrs];

    for i in 0..n_attrs {
      ...
     }

  return children;
```

Dentro de cada iteración del bucle calculamos que padre tiene el atributo mas alto y cual el mas pequeño, y los almacenamos. En caso de que los pesos sean iguales nos ahorramos calculos ya que los dos hijos tendrán ese mismo peso.

```
if parent1.weights[i] < parent2.weights[i] {
    c_max = parent2.weights[i];
    c_min = parent1.weights[i];
} else if parent1.weights[i] > parent2.weights[i] {
    c_max = parent1.weights[i];
    c_min = parent2.weights[i];
} else {
    weights1[i] = parent1.weights[i];
    weights2[i] = parent1.weights[i];
    continue;
}
```

Ahora calculamos los límites superior e inferior del intervalo donde vamos a generar el peso de los hijos.

```
lower_bound = c_min - alpha * (c_max - c_min);
upper_bound = c_max + alpha * (c_max - c_min);

value1 = rng.gen_range(lower_bound, upper_bound);
value2 = rng.gen_range(lower_bound, upper_bound);
```

Esos serán los valores del peso correspondiente en los hijos, sin embargo hay que considerar que `upper_bound` podría ser mayor que 1 y `lower_bound` menor que 0. Una solución sería capar las cotas directamente pero entonces estariamos bajando la probabilidad de que el peso resultante fuera 1. Es decir, si el intervalo fuera (0.8, 1.2), la probabilidad de que un peso quede por encima de 1 (y luego

haya que truncarlo) es mas alta que si cambiamos el intervalo a (0.8, 1). De forma que no cambiamos el intervalo.

Por ello lo que hacemos es caparlo a la hora de insertarlo.

```
    weights1[i] = truncate(value1);
    weights2[i] = truncate(value2);
}
children.push(weights1);
children.push(weights2);
```

5 descripción de los algoritmos considerados (práctica 1).

5.1 *Algoritmo greedy RELIEF.*

El algoritmo greedy RELIEF recorre todo el conjunto, modificando el vector de pesos en función del enemigo y el aliado mas cercanos a cada elemento, utilizando la diferencia entre los atributos. Se consideran enemigos a aquellos que pertenecen a otra clase y aliados a los que pertenecen a la misma. La idea del algoritmo es incrementar el peso de aquellas características que mejor separan elementos de distintas clases y reducir los pesos que separan los de la misma clase.

Este algoritmo una función auxiliar `normalize_and_truncate_negative_weights`, que dado un vector de pesos, pone a `0.0` aquellos pesos que sean negativos y luego normaliza el vector.

```
fn normalize_and_truncate_negative_weights(weights: Vec<f32>) {
    for attr in 0..weights.length {
        if weights[attr] > highest_weight
            highest_weight = weights[attr];
        if weights[attr] < 0.0
            weights[attr] = 0.0;
    }
    for attr in 0..weights.length
        weights[attr] = weights[attr] / highest_weight;
}
```

Finalmente, el algoritmo RELIEF se encuentra estructurado de la siguiente forma,

Primero inicializamos el vector de pesos a 0, e iteramos sobre cada elemento de `knowledge`.

```
fn calculate_greedy_weights(knowledge: Vec<Datos>, n_attrs: u8) -> Vec<float> {

    weights = [0.0; n_attrs];

    for known in knowledge {
      ...
    }
```

En cada una de estas iteraciones, inicializamos una serie de variables y buscamos el aliado y el enemigo.

```
enemy_distance = MAX;
ally_distance = MAX;
ally_index = 0;
enemy_index = 0;

for (index, candidate) in knowledge.enumerate() { // Iterate over pair<index,element>
    // NOTE Skip if cantidate == known
    if candidate != known {
        // NOTE Pre-calculate distance
```

```
        dist = euclidean_distance(known, candidate);
        // NOTE Ally
        if known.class == candidate.class
            if dist < friend_distance {
                ally_index = index;
                ally_distance = dist;
            }
        // NOTE Enemy
        else
            if dist < enemy_distance {
                enemy_index = index;
                enemy_distance = dist;
            }
    }
}
enemy = knowledge[enemy_index];
ally = knowledge[ally_index];
```

Una vez encontrados ajustamos el vector de pesos.

```
for attr in 0..n_attrs {
    weights[attr] += (known.attrs(attr) - enemy.attrs(attr)).abs()
        - (known.attrs(attr) - ally.attrs(attr)).abs();
}
```

Para finalizar normalizamos el vector y truncamos los valores negativos.

```
normalize_and_truncate_negative_weights(weights);
return weights;
```

5.2 *Algoritmo búsqueda local*

El algoritmo de búsqueda local, realiza una serie de mutaciones sobre el vector de pesos, y prueba el nuevo vector sobre el conjunto de entrenamiento, realizando *leave one out*.

La función de mutación tiene la siguiente forma.

```
fn mutate_weights(weights: Vec<f32>, desv: f32, index_to_mutate: u32) {

    weights[index_to_mutate] += Normal(0.0,desv).sample();

    if weights[index_to_mutate] > 1.0
        weights[index_to_mutate] = 1.0;
    if weights[index_to_mutate] < 0.0
        weights[index_to_mutate] = 0.0;
}
```

Donde realiza una mutación utilizando una distribución normal de media 0 y desviación típica `desv`, sobre el elemento deseado del vector. Luego comprueba que el valor no se salga de los limites (0 y 1).

El algoritmo de búsqueda local realiza un máximo de 15000 mutaciones parando si se llegan a realizar $20 * n\_atributos$ sin que ninguna presente mejoría. Se mutará siempre un índice distinto del vector de pesos, sin repetirse hasta que todos hayan sido mutados.

Veamos el algoritmo en detalle, lo primero que hacemos es inicializar el vector de pesos, dependiendo del valor de `initial_weights` lo haremos de una forma u otra, si es 1 utilizamos valores aleatorios, si es 2, los pesos de RELIEF, y si es 3, los pesos de un algoritmo que explicaré en la siguiente sección.

```
fn calculate_local_search_weights(
    training: Vec<Datos>,
    n_attrs: u32,
    discard_low_weights: bool,
    initial_weights: u8,
) -> Vec<f32> {

    weights = [0.0; n_attrs];

    match initial_weights {  //This is like a switch
        2 => weights = calculate_relief_weights(training, n_attrs),
        3 => weights = alternative_greedy_weights(training, n_attrs),
        1 => {
            let uniform = new Uniform(0.0, 1.0);
            for attr in 0..n_attrs {
                weights[attr] += uniform.sample();
            }
        }
    }
```

Una vez inicializado el vector de pesos, inicializamos el vector de índices a mutar, precalculamos el valor de la función de evaluación con los pesos actuales y comenzamos el bucle de mutaciones.

```
index_vec = (0..n_attrs).shuffle();

best_result = classifier_1nn(training, training, weights, discard_low_weights);

max_neighbours_without_muting = 20 * n_attrs;
n_neighbours_generated_without_muting = 0;

for i in 0..15000 {
  ...
}
```

En cada iteracion del bucle de mutaciones, llamamos a la funcion `mutate_weights` sobre el índice que nos marque el vector, tambien calculamos el valor de la función de evaluación sobre el conjunto de entrenamiento con los nuevos pesos.

```
index_to_mute = index_vec.pop();
muted_weights = weights;
mutate_weights(muted_weights, 0.3, index_to_mute);

muted_result =
    classifier_1nn(training, training, muted_weights, discard_low_weights);
```

En caso de que la mutación suponga una mejoría, guardamos los nuevos valores, reseteamos el contador de mutaciones sin mejora y volvemos a inicializar el vector de índices.

```
if muted_result.evaluation_function > best_result.evaluation_function {
    n_neighbours_generated_without_muting = 0;
    weights = muted_weights;
    best_result = muted_result;
    index_vec = (0..n_attrs).shuffle();
}
```

En caso de no mejorar, aumentamos el contador de mutaciones sin mejorar, si alcanzamos el máximo salimos del bucle. Tambien comprobamos que el vector de índices no este vacío, de estarlo lo rellenamos.

```
else {
    n_neighbours_generated_without_muting += 1;
```

```
    if n_neighbours_generated_without_muting == max_neighbours_without_muting {
        break;
    }
    //NOTE If no more index to mutate, recharge them.
    if index_vec.is_empty {
        index_vec = (0..n_attrs).shuffle;
    }
}
```

Finalmente devolvemos el vector de pesos.

5.3 *Otros algoritmos considerados*

El primer cambio realizado es la opción de no descartar los pesos bajo el umbral (0,2) en el clasificador, esta modificación solo la vamos a probar en RELIEF, durante el desarollo de la práctica, pude observar que tras implementar el descarte de pesos los resultados empeoraron, de forma que decidí añadir la opción de no hacerlo, luego veremos los resultados.

La modificación sobre la búsqueda local consiste en añadir la opción de inicializar el vector de pesos utilizando los que devuelve el algoritmo RELIEF o los que devuelve el siguiente algoritmo, este se aprovecha de que la tasa de reducción vale un 50 % de la función de evaluación, de modo que devuelve un vector de pesos donde solo 1 de ellos no es nulo. Este vector se podía haber elegido de forma aleatoria pero he decidido hacerlo de la siguiente manera.

Inicializamos el vector de pesos que vamos a devolver y un vector de pesos auxiliar `attr_sum`, iteramos sobre los elementos del conjunto de entrenamiento.

```
fn alternative_greedy_weights(
    knowledge: Vec<Datos>,
    n_attrs: u8,
) -> Vec<f32> {

     weights = [0.0; n_attrs];
     attr_sum = [0.0; n_attrs];

    for known in knowledge {
      ...
    }
```

Calculamos el mejor enemigo del vector igual que haciamos en RELIEF.

```
enemy_distance = MAX;
enemy_index = 0;

for (index, candidate) in knowledge.enumerate() {
  if known.class != candidate.class {
      dist = known.euclidean_distance(candidate);
      if dist < enemy_distance {
          enemy_index = index;
          enemy_distance = dist;
      }
  }
}
enemy = knowledge[enemy_index];
```

Una vez encontrado, sumamos las distancias de sus atributos en el vector auxiliar.

```
for attr in 0..n_attrs {
    attr_sum[attr] += (enemy.get_attr(attr) - known-get_attr(attr)).abs();
}
```

Cuando hemos terminado de recorrer el bucle y hemos sumado los atributos de todos los enemigods, buscamos el atributo mas grande, y en esa posición ponemos un 1,0 en el vector de pesos. La idea es solo darle importancia al atributo que mejor separa los elementos de distintas clases, dejando los demás a 0, para obtener una alta tasa de reducción.

```
max_value = 0.0;
max_index = 0;
for attr in 0..n_attrs {
    if attr_sum[attr] > max_value {
        max_index = attr;
        max_value = attr_sum[attr];
    }
}
weights[max_index] = 1.0;
return weights;
```

6 descripción de los algoritmos considerados (práctica 2).

6.1 *Algoritmo genético generacional*

En esta sección veremos como funciona y como esta implementado el algoritmo genético generacional de la práctica.

El código se encuentra estructurado de forma que existe un método `genetic_generational_algorithm`, cuya unica labor es gestionar en número de generaciones que se deben realizar y las operaciones existentes entre ellas. Esto tendrá mas sentido en el algoritmo memético cuando haya que llamar a la búsqueda local.

La función tiene la siguiente forma.

```
fn genetic_generational_algorithm(
    training: Vec<Data>,
    n_attrs: u32,
    cross_prob: f32,
    mut_prob: f32,
    generation_size: u8,
    selection_operator: fn,
    cross_operator: fn,
    rng: Rng,
) -> Vec<f32> {
    generation =
        initial_generation(generation_size, n_attrs, training, rng);
    n_calls_to_ev = generation_size;
    _n_generation = 0;
```

Donde vemos que acepta como parámetros los operadores de seleccion y cruce, ademas de las respectivas probabilidades y el conjunto de entrenamiento. Lo primero que hace es generar una población inicial aleatoria, el número de llamadas a la función objetivo y lo que será nuestro contador de generaciones.

Luego la función entra en un bucle utilizando el criterio de parada que se nos ha indicado.

```
while n_calls_to_ev < 15000 {
    iteration = generational_iteration(
        generation,
        training,
        n_attrs,
        cross_prob,
        mut_prob,
        generation_size,
        selection_operator,
        cross_operator,
        rng,
    );
    _n_generation += 1;
    generation = iteration.generation;
    n_calls_to_ev += iteration.calls;
}

return generation
    .last()
    .weights
```

Como la generación se encuentra ordenada, para saber que elemento es el mejor solo debemos devolver el último (esta ordenada de menor a mayor).

Veamos ahora como funciona cada iteración del algoritmo.

Lo primero que hacemos es inicializar el número de llamadas a la función de evaluacion que vamos a hacer en la iteración. Luego le decimos al operador de selección que nos devuelva un vector con tantos padres como elementos hay en la población.

Ese mismo vector de padres se lo pasamos al operador de cruce, que en nuestro caso (`cross_prob = 0.7`) utilizamos la esperanza matemática para decirle que nos devuelva directamente 20 hijos (`cross_prob * generation_size`).

Añadimos esos hijos a la siguiente generación sin evaluarlos (si lo hacemos y luego mutara, estariamos desperdiciando una evaluación).

```
n_calls_to_ev = 0;

parents = selection_operator(&generation, generation_size, rng);

children = cross_operator(
     parents,
     cross_prob * generation_size,
     n_attrs,
     rng,
 );

next_generation = children;
next_generation.append(parents);
```

Ahora, como el operador de cruce elimina los padres que ha utilizado, solo nos queda añadir los resultantes a la población (son aquellos en los que la `prob_cruce` "falla").

Veamos ahora como calcular el número de mutaciones a realizar. Primero calculamos la esperanza matemáticas de mutaciones (`n_muts`). Y truncamos su valor (`trunc`), calculando tambien su parte decimal (`dec`). Esta claro que mínimo se deben realizar tantas mutaciones como indique `trunc`, pero el decimal también es importante, por ello la estrategia seguida es la siguiente, se genera un número aleatorio y si es menor que la parte decimal se aumenta en 1 el número de mutaciones.

```
n_muts = mut_prob * n_attrs * generation_size;
trunc = (u8)n_muts;
dec = n_muts - trunc;
if rng.gen_range(0.0, 1.0) < dec {
    trunc += 1;
}
```

Ya hemos calculado el número de mutaciones que vamos a realizar, ahora vamos a ver nuestra población como una matriz (vectores de vectores de pesos) y vamos a seleccionar posiciones aleatorias en dicha matriz. Hay que tener una consideración y esque el método clasico de calcular mutaciones no permite que se mute 2 veces el mismo atributo del mismo vector, para tener esto en cuenta vamos a generar tantos números aleatorios como necesitamos y los vamos a insertar en un `set`, así nos aseguramos de no mutar la misma posición dos veces.

```
nums = new Set();
while nums.len() < trunc {
    nums.insert(rng.gen_range(0, generation_size * n_attrs));
}
```

El siguiente paso es realizar las mutaciones. Para ello nos podemos aprovechar de la función que habiamos creado para la práctica anterior `mutate_weights`. Ponemos el valor del cromosoma mutado a `-1`, que nos indica que ha cambiado y es necesario volver a evaluarlo.

```
for random_value in nums {
    chromosome = random_value / n_attrs;
    attr = random_value % n_attrs;
    mutate_weights(next_generation[chromosome].weights, 0.3, attr, rng);
    next_generation[chromosome].result = -1.0;
}
```

Ahora procedemos a evaluar todos los cromosomas de nuestra población y la ordenamos.

```
for chromosome in next_generation {
    if chromosome.result == -1.0 {
        fitness_function(training, chromosome);
        n_calls_to_ev += 1;
    }
}
next_generation.sort();
```

Ahora es el momento de mantener el elitismo de la población, para ello cogemos los mejores elementos de la anterior y la actual.

```
best_of_last_generation = generation.last()
best_of_this_generation = next_generation.last()
```

Comprobamos si nuestra nueva generación es mejor que la anterior y en caso contrario lo arreglamos, eliminando el peor elemento (`0`).

```
if best_of_this_generation.result < best_of_last_generation.result {
    next_generation.remove(0);
    next_generation.push(best_of_last_generation);
}
return (next_generation, n_calls_to_ev);
```

6.2 *Algoritmo genético estacionario*

En esta sección veremos como funciona y como esta implementado el algoritmo genético estacionario.

Aunque como ya se ha explicado el generacional y tienen varios aspectos en común. solo se detallarán las diferencias entre ellos.

Lo primero es que ahora el algoritmo de seleccion solo tiene que coger 2 elementos, y por tanto el de cruce solo devolver 2 hijos.

```
parents = selection_operator(generation, 2, rng);
children = cross_operator(parents, 2, n_attrs, rng);
```

El proceso de mutación es el mismo, teniendo en cuenta que nuestra "nueva generación"tiene 2 elementos solo.

```
n_muts = mut_prob * n_attrs * 2 ;
trunc = (u8)n_muts;
dec = n_muts - trunc;
if rng.gen_range(0.0, 1.0) < dec{
    trunc += 1;
}

let mut nums = new Set();
while nums.len() < trunc {
    nums.insert(rng.gen_range(0, 2 * n_attrs));
}
```

Luego evaluamos los dos hijos que ya han podido ser mutados y los ordenamos en `next_generation`.

Ahora lo que tenemos que ver es si estos dos hijos van a formar parte de la nueva generación o no. Para ello hacemos el siguiente razonamiento. Cogemos los dos peores elementos de la generación anterior, los llamaré `best` y `worst`, indicando que `best` es el mejor de ellos y `worst` el peor. Existen tres opciones (recordemos que la generación esta ordenada al revés).

- Ambos son mejores que los nuevos hijos, en ese caso `worst` será mejor que `best_child`. Entonces sacaremos a los hijos de la generación e insertaremos a estos dos.
- Alguno es mejor que el peor hijo, en particuñar `best` será mejor que `worst_child`, con lo cual eliminamos a este de la población e insertamos a `best`.
- Los hijos son mejores que ambos, entonces no hacemos ningún cambio en la población.

```
worst_child = next_generation.get(0):
best_child = next_generation.get(1);
worst = generation.get(0);
best = generation.get(1);
if best_child.result < worst.result {
    next_generation.clear();
    next_generation.push(best);
    next_generation.push(worst);
} else if worst_child.result < best.result {
    next_generation.remove(0);
    next_generation.push(best);
}
```

Ahora nuestra población `next_generation` tiene los 2 mejores elementos de aquellos 4. Nos falta completarla con el resto de la generación anterior.

```
next_generation.extend(generation[2..]);
next_generation.sort();
return (next_generation, n_calls_to_ev);
```

6.3 *Algoritmo memético*

Vamos a ver ahora el funcionamiento del algoritmo memético, para ello tenemos que estudiar también el funcionamiento del la búsqueda local de baja intensidad. La unica diferencia con la búsqueda local de la práctica anterior es que esta acepta un cromosoma en lugar de un peso, escribe el resultado en ese mismo cromosoma y devuelve el número de evaluaciones realizadas.

```
fn memetic_local_search_weights>(
    training: Vec<Data>,
    chromosome: Chromosome,
    n_attrs: u8,
    rng: Rng,
) -> u8 {
    n_evaluations = 0;
    index_vec = [0..n_attrs];
    index_vec.shuffle();

    for _ in 0..2 * n_attrs {
        if index_vec.is_empty() {
            index_vec = (0..n_attrs);
            index_vec.shuffle();
        }
        index_to_mutate = index_vec.pop();
        muted_weights = chromosome.weights;
```

```
        mutate_weights(muted_weights, 0.3, index_to_mutate, rng);

        muted_result =
            classifier_1nn(training, training, &muted_weights).evaluation_function();
        n_evaluations += 1;
        if muted_result > chromosome.result {
            index_vec.clear();
            chromosome.weights = muted_weights;
            chromosome.result = muted_result;
        }
    }
    return n_evaluations;
}
```

Veamos ahora el funcionamiento del algoritmo memético, la estructura es la misma que en los anteriores.

Inicializamos la población y comenzamos dos bucles, uno exterior con una etiqueta de `outer`.

```
generation =
    initial_generation(generation_size, n_attrs, training, rng);

n_calls_to_ev = generation_size;
_n_generation = 0;
'outer: loop {
    for _ in 0..10 {
      ...
    }
 ...
 }
```

Veamos que hacemos en cada iteración de bucle interior. Si el número de iteraciones es mayor que el número de evaluaciones fijado, entonces paramos el bucle exterior. En otro caso calculamos la siguiente generación igual que en el generacional. También podriamos llamar al estacionario, en el análisis discutiremos esta posibilidad.

```
if n_calls_to_ev >= 15000 {
    break 'outer;
}

let iteration = generational_iteration(
    generation,
    training,
    n_attrs,
    cross_prob,
    mut_prob,
    generation_size,
    selection_operator,
    cross_operator,
    rng,
);

_n_generation += 1;
generation = iteration.0;
n_calls_to_ev += iteration.1;
```

Después del bucle de 10 iteraciones realizamos la parte de explotación del algoritmo. Para ello distinguimos los 3 casos que se nos plantean.

```
match memetic_type {
    2 => {
```

```
        let selected
            (0..generation_size).choose_multiple(rng, generation_size / 10);

        for index in selected {
            n_calls_to_ev +=
                memetic_local_search_weights(training, generation[index], n_attrs, rng)
        }
    }
```

En el caso de que el tipo sea el segundo, llamamos a la busqueda local para un 10 % aleatorio de la población. En nuestro caso es 1 solo elemento.

```
3 => {
    for index in generation_size - generation_size / 10..generation_size {
        n_calls_to_ev +=
            memetic_local_search_weights(training, generation[index], n_attrs, rng)
    }
}
```

Si el tipo es 3, se llama sobre el 10 % final de la generación (los 10 % mejores). En otro caso se hace el tipo 1, llamandose a la BL sobre todos los elementos.

```
    _ => {
        for index in 0..generation_size {
            n_calls_to_ev +=
                memetic_local_search_weights(training, generation[index], n_attrs, rng)
        }
    }
}

generation.sort();
```

7 PROCEDIMIENTO CONSIDERADO PARA DESARROLLAR LA PRÁCTICA.

Como ya he explicado en las notas sobre el psudocódigo, a implementación se ha hecho en el lenguaje de programación `Rust`. Todo el código necesario para ejecutar el programa se encuentra en el directorio. `Rust` dispone de una herramienta gestora de paquetes `Cargo`, es posible ejecutar el programa sin utilizarla pero no se recomienda.

Para ejecutarlo utilizando `Cargo` basta con ejecutar el comando `cargo run --release` en el mismo directorio donde se encuentran `src/` y `data/`. Se puede ejecutar sin `--release` pero la ejecución tardará bastante más. Para la ejecución se puede pasar como parámetro la semilla a utilizar, de no hacerlo el programa cogerá como semilla 1, con la que se ha hecho el análisis de resultados. El programa ejecuta todos los algoritmos en los 3 conjuntos de datos, es fácil no ejecutar algún conjunto de datos comentando unos valores booleanos en la función `main` de `src/main.rs`, por ejemplo, `do_texture`, o no ejecutar algún algoritmo cambiando otros en la función `run` del mismo fichero.

El código de la practica se encuentra dividido en 2 archivos, `main.rs` y `structs.rs`. En el primero de ellos se encuentran todas las funciones auxiliares y algoritmos utilizados, y en el segundo todas las estructuras. En la carpeta `data/` se encuentran los archivos `.csv`.

Tambien se puede utilizar `Cargo` para generar un `.html` con toda la documentación de las funciones, utilizando el comando `cargo doc`. La documentación se crearía automaticamente una carpeta `target/doc`.

8 experimentos y análisis de resultados.

En esta sección se muestran los resultados obtenidos por cada uno de los algoritmos. El análisis de resultados se realiza en la siguiente seción.

El ordenador sobre el que se han realizado las ejecuciones tiene sistema operativo Manjaro Linux 64-bit, con procesador Intel Core i7-5700HQ(8) @3.50GHz

Los resultados se encuentran divididos en tablas de la siguiente forma, primero para cada algoritmo existe una tabla donde se comparan los resultados de cada una de las particiones en cada conjunto de datos. El algoritmo RELIEF aparece dos veces y la búsqueda local 3, la primera tabla del algoritmo RELIEF son los resultados obtenidos al descartar los pesos menores que el umbral prefijado (0,2), mientras que la segunda tabla son los resultados obtenidos si no se descartan. La primera tabla del algoritmo de búsqueda local corresponde a los resultados donde el vector de pesos inicial es el generado por la distribucion uniforme, es decir, pesos aleatorios entre 0 y 1, la segunda tabla corresponde a los resultados cuando el vector de pesos inicial es el que nos devuelve el algoritmo RELIEF y la tercera tabla corresponde a cuando el vector es el que nos devuelve el algoritmo greedy que he añadido.

Luego hay 3 tablas más, correspondientes a los valores medios obtenidos por cada algoritmo en cada conjunto de datos.

Un factor a tener en cuenta es que, al realizar varias llamadas a la búsqueda local, los resultados de la búsqueda local 2 dependen de si se ha realizado la búsqueda local 1, debido al generador de números aleatorios. Por ello los resultados que aparecen en las tablas son los correspondientes a las ejecuciones aisladas de cada una de ellas, si se ejecutan todas seguidas no saldrán los mismos resultados.

Respecto a las tablas de la segunda práctica, se encuentran en el msimo formato que las de la primera apareciendo luego junto al resto de algoritmos. El apartado de mutaciones aparecerá en blanco ya que al estar usandose la esperanza matemática no tiene sentido comparar las que realiza uno u otro.

8.1 *Tablas detalladas por algoritmo (Práctica 1).*

8.1.1 *1-NN*

Texture

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	93.636364	0	46.818182	1
Partición 2	89.090910	0	44.545455	1
Partición 3	94.545454	0	47.272727	1
Partición 4	92.727274	0	46.363637	1
Partición 5	92.727274	0	46.363637	1
Media	92.545455	0	46.272728	1

Colposcopy

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	74.57627	0	37.288135	0
Partición 2	70.17544	0	35.087720	0
Partición 3	73.68421	0	36.842105	0
Partición 4	75.43859	0	37.719298	0
Partición 5	82.45614	0	41.228070	0
Media	75.26613	0	37.633066	0

Ionosphere

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	90.140843	0	45.070422	0
Partición 2	80.000000	0	40.000000	0
Partición 3	82.857144	0	41.428572	0
Partición 4	92.857144	0	46.428572	0
Partición 5	87.142855	0	43.571428	0
Media	86.599597	0	43.299799	0

8.1.2 *RELIEF descartando*

Texture

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	91.818184	15	53.409092	6
Partición 2	91.818184	2.5	47.159092	6
Partición 3	95.454544	2.5	48.977272	6
Partición 4	92.727274	2.5	47.613637	6
Partición 5	93.636364	5	49.318182	6
Media	93.090910	5.5	49.295455	6

Colposcopy

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	72.881360	40.322580	56.601970	2
Partición 2	75.438595	27.419356	51.428976	2
Partición 3	77.192980	32.258064	54.725522	3
Partición 4	71.929824	51.612900	61.771362	2
Partición 5	82.456140	30.645162	56.550651	2
Media	75.979780	36.451612	56.215696	2.2

Ionosphere

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	90.140843	2.9411765	46.541010	2
Partición 2	81.428572	2.9411765	42.184874	2
Partición 3	82.857144	2.9411765	42.899160	2
Partición 4	92.857140	2.9411765	47.899158	2
Partición 5	90.000000	2.9411765	46.470588	2
Media	87.456740	2.9411765	45.198958	2

8.1.3 *RELIEF sin descartar*

Texture

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	93.636364	15	54.318182	6
Partición 2	90.909094	2.5	46.704547	6
Partición 3	95.454544	2.5	48.977272	6
Partición 4	92.727274	2.5	47.613637	6
Partición 5	93.636364	5	49.318182	6
Media	93.272728	5.5	49.386364	6

Colposcopy

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	72.881360	40.322580	56.601970	2
Partición 2	71.929824	27.419356	49.674590	2
Partición 3	78.947370	32.258064	55.602717	3
Partición 4	73.684210	51.612900	62.648555	2
Partición 5	84.210527	30.645162	57.427845	2
Media	76.330658	36.451612	56.391135	2.2

Ionosphere

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 1	90.140843	2.9411765	46.541010	2
Partición 2	81.428572	2.9411765	42.184874	2
Partición 3	82.857144	2.9411765	42.899160	2
Partición 4	92.857140	2.9411765	47.899158	2
Partición 5	90.000000	2.9411765	46.470588	2
Media	87.456740	2.9411765	45.198958	2

8.1.4 *Alternativa greedy*

Texture

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 0	33.636364	97.5	65.568182	5
Partición 1	36.363637	97.5	66.931819	5
Partición 2	38.181818	97.5	67.840909	5
Partición 3	40.000000	97.5	68.750000	5
Partición 4	35.454544	97.5	66.522727	5
Media	36.745455	97.5	67.122727	5

Colposcopy

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 0	77.966100	98.3871	88.176600	1
Partición 1	66.666667	98.3871	82.526884	1
Partición 2	66.666667	98.3871	82.526884	1
Partición 3	57.894737	98.3871	78.140919	1
Partición 4	68.421054	98.3871	83.404077	1
Media	67.523045	98.3871	82.955073	1

Ionosphere

Partición	Tasa clasificación	Tasa reducción	Agregado	Tiempo (ms)
Partición 0	64.788735	97.05882	80.923778	1
Partición 1	75.714284	97.05882	86.386552	1
Partición 2	71.428573	97.05882	84.243697	1
Partición 3	81.428570	97.05882	89.243695	1
Partición 4	75.714284	97.05882	86.386552	1
Media	73.814889	97.05882	85.436855	1

8.1.5 *Búsqueda local*

Texture

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	87.27273	85	86.136365	12579	60
Partición 1	87.27273	82.5	84.886365	13432	59
Partición 2	89.09091	87.5	88.295455	24891	62
Partición 3	84.54546	82.5	83.522730	15794	67
Partición 4	83.63636	82.5	83.068180	20386	69
Media	86.363638	84.0	85.181819	17416.4	63.4

Colposcopy

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	76.27119	75.80645	76.03882	7487	75
Partición 1	75.43859	83.87096	79.65478	6097	64
Partición 2	78.94737	69.35484	74.15110	9439	63
Partición 3	71.92982	74.19355	73.06168	11751	59
Partición 4	73.68421	85.48387	79.58404	12121	63
Media	75.25423	77.741935	76.498086	9379	64.8

Ionosphere

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	92.957747	88.23529	90.596521	4927	59
Partición 1	74.285716	91.17647	82.731093	3103	48
Partición 2	85.714287	91.17647	88.445379	2934	42
Partición 3	87.142855	85.29411	86.218486	3961	54
Partición 4	88.571430	82.35294	85.462185	2948	33
Media	85.734407	87.647058	86.690733	3574.6	47.2

8.1.6 *Búsqueda local con vector inicial de RELIEF*

Texture

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	89.09091	82.5	85.79545	7948	44
Partición 1	90.00000	85	87.50000	10993	63
Partición 2	93.63636	87.5	90.56818	14406	70
Partición 3	84.54546	87.5	86.02273	10536	50
Partición 4	87.27273	85	86.13636	10248	56
Media	88.90909	85.5	87.204546	10826.2	56.6

Colposcopy

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	74.57627	80.64516	77.610715	5823	57
Partición 1	80.70175	83.87096	82.286361	7318	64
Partición 2	78.94737	85.48387	82.215620	11034	67
Partición 3	73.68421	88.70967	81.196943	4243	38
Partición 4	71.92982	83.87096	77.900395	6406	70
Media	75.96788	84.51612	80.242007	6964.8	59.2

Ionosphere

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	80.28169	88.235295	84.258493	4384	67
Partición 1	81.42857	85.294116	83.361343	4025	52
Partición 2	84.28571	91.176470	87.731090	4501	68
Partición 3	91.42857	88.235295	89.831933	8425	71
Partición 4	88.57143	82.352940	85.462185	4428	55
Media	85.19919	87.058823	86.129009	5152.6	62.6

8.1.7 *Búsqueda local con vector inicial de greedy.*

Texture

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	90.909094	85	87.954547	9321	15
Partición 1	88.181820	85	86.590910	10303	22
Partición 2	95.454544	85	90.227272	5142	14
Partición 3	91.818184	85	88.409092	7802	18
Partición 4	86.363630	85	85.681815	6105	8
Media	90.545454	85	87.772727	7734.6	15.4

Colposcopy

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	81.355930	95.16129	88.258610	3227	2
Partición 1	66.666670	90.32258	78.494625	8461	12
Partición 2	75.438595	95.16129	85.299943	3670	5
Partición 3	71.929824	93.54839	82.739107	6485	12
Partición 4	70.175440	91.93548	81.055460	5242	13
Media	73.113292	93.22580	83.169549	5417	8.8

Ionosphere

Partición	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
Partición 0	91.549295	88.23529	89.892295	2195	5 5
Partición 1	80.000000	91.17647	85.588235	1903	8
Partición 2	91.428570	91.17647	91.302520	3356	11
Partición 3	90.000000	88.23529	89.117648	1954	6
Partición 4	82.857114	91.17647	87.016792	2305	8
Media	87.166996	90.00000	88.583498	2342.6	11.6

8.2 *Tablas detalladas por algoritmo (Práctica 2).*

8.2.1 *Algoritmo genético generacional con cruce aritmético.*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	90.000000	80.000000	85.000000	98248
Partición 1	89.090910	67.500000	78.295455	105582
Partición 2	92.727274	70.000000	81.363637	101701
Partición 3	89.090910	77.500000	83.295455	103449
Partición 4	92.727274	77.500000	85.113637	101813
Media	90.727274	74.5	82.613637	102158.6

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	74.57627	59.677420	67.126845	44335
Partición 1	70.17544	62.903225	66.539333	45499
Partición 2	77.19298	58.064514	67.628747	48600
Partición 3	73.68421	64.516103	69.100157	47317
Partición 4	78.94737	58.064514	68.505942	44717
Media	74.915254	60.645155	67.780205	46093.6

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	81.690140	82.352940	82.02154	34761
Partición 1	81.428570	73.529410	77.47899	36104
Partición 2	87.142855	64.705884	75.924370	45563
Partición 3	90.000000	73.529410	81.764705	36090
Partición 4	92.857140	76.470590	84.663865	37077
Media	86.623741	74.117647	80.370694	37919

8.2.2 *Algoritmo genético generacional con cruce BLX.*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	90.909094	87.5	89.204547	92083
Partición 1	92.727274	85.0	88.863637	97394
Partición 2	90.909094	77.5	84.204547	101117
Partición 3	86.363640	82.5	84.431820	99005
Partición 4	92.727274	82.5	87.613637	93546
Media	90.727275	83.0	86.863638	96629

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	77.966100	75.80645	76.886275	39326
Partición 1	80.701756	72.58065	76.641203	40180
Partición 2	71.929824	72.58065	72.255237	41990
Partición 3	68.421054	77.41935	72.920202	38860
Partición 4	73.684210	77.41935	75.551780	44683
Media	74.540589	75.16129	74.850939	41007.8

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	90.140843	88.235295	89.188069	33884
Partición 1	87.142855	88.235295	87.689075	31525
Partición 2	85.714287	88.235295	86.974791	32595
Partición 3	94.285715	91.176470	92.731093	32796
Partición 4	88.571430	85.294116	86.932773	33914
Media	89.171026	88.235294	88.703160	32942.8

8.2.3 *Algoritmo genético estacionario con cruce aritmético.*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	84.54546	85.0	84.772730	95803
Partición 1	87.27273	70.0	78.636365	110003
Partición 2	87.27273	72.5	79.886365	111586
Partición 3	85.45455	77.5	81.477275	106585
Partición 4	84.54546	82.5	83.522730	98446
Media	85.81818	77.5	81.659093	104484.6

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	76.271190	59.677429	67.974310	44600
Partición 1	75.438595	61.290324	68.364460	44005
Partición 2	66.666670	66.129035	66.397853	43702
Partición 3	64.912283	67.741936	66.327110	43759
Partición 4	78.947370	64.516130	71.731750	43835
Media	72.447222	63.870971	68.159097	43980.2

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	87.323946	64.705884	76.014915	37402
Partición 1	85.714287	58.823530	72.268909	36564
Partición 2	87.142855	70.588240	78.865548	35552
Partición 3	91.428570	82.352940	86.890755	33691
Partición 4	88.571430	79.411760	83.991595	35665
Media	88.036218	71.176471	79.606344	35774.8

8.2.4 *Algoritmo genético estacionario con cruce BLX.*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	90.00000	77.5	83.75000	99536
Partición 1	89.09091	82.5	85.79545	89095
Partición 2	91.81818	77.5	84.65909	93316
Partición 3	89.09091	82.5	85.79545	93924
Partición 4	88.18182	82.5	85.34091	97261
Media	89.636364	80.5	85.068182	94626.4

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	69.491524	67.741936	68.616739	41245
Partición 1	71.929824	75.806450	73.868137	41433
Partición 2	77.192980	72.580650	74.886815	41987
Partición 3	75.438595	69.354840	72.396718	42027
Partición 4	78.947370	66.129035	72.538203	43901
Media	74.600059	70.322582	72.461321	42118.6

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	92.957747	85.294116	89.125932	31042
Partición 1	87.142855	85.294116	86.218486	30797
Partición 2	87.142855	82.352940	84.747898	31681
Partición 3	92.857140	67.647060	80.252100	34877
Partición 4	90.000000	88.235295	89.117648	30125
Media	90.020119	81.764705	85.892413	31704.4

8.2.5 *Algoritmo memético 1*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	90.00000	85.0	87.500000	88018
Partición 1	85.45455	85.0	85.227275	89085
Partición 2	90.00000	87.5	88.750000	88598
Partición 3	88.18182	87.5	87.840910	87827
Partición 4	87.27273	85.0	86.136365	87771
Media	88.18182	86.0	87.09091	88259.8

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	77.966100	80.645160	79.305630	39342
Partición 1	68.421054	82.258064	75.339559	40581
Partición 2	71.929824	82.258064	77.093944	38556
Partición 3	68.421054	75.806450	72.113752	39884
Partición 4	71.929824	87.096775	79.513300	37742
Media	71.733571	81.612903	76.673237	39221

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	85.915494	91.176470	88.545982	29843
Partición 1	85.714287	91.176470	88.445379	30881
Partición 2	82.857144	85.294116	84.075630	31252
Partición 3	85.714287	94.117650	89.915969	31921
Partición 4	81.428570	85.294116	83.361343	30932
Media	84.325956	89.411764	86.868861	30965.8

8.2.6 *Algoritmo memético 1 con pesos iniciales retocados.*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	89.090910	87.5	88.295455	86398
Partición 1	92.727274	85.0	88.863637	88002
Partición 2	88.181820	85.0	86.590910	88565
Partición 3	89.090910	87.5	88.295455	87628
Partición 4	92.727274	87.5	90.113637	85799
Media	90.363638	86.5	88.431819	87278.4

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	61.016950	93.54839	77.282670	37577
Partición 1	77.192980	93.54839	85.370685	38801
Partición 2	71.929824	95.16129	83.545557	36384
Partición 3	78.947370	93.54839	86.247880	37565
Partición 4	68.421054	90.32258	79.371817	34199
Media	71.501636	93.225808	82.363722	36905.2

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	84.507040	91.176470	87.841755	29170
Partición 1	78.571427	91.176470	84.873949	30449
Partición 2	84.285710	91.176470	87.731090	28998
Partición 3	94.285715	88.235295	91.260505	29440
Partición 4	82.857144	91.176470	87.016807	31771
Media	84.901407	90.588235	87.744821	29965.6

8.2.7 *Algoritmo memético 2*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	86.363640	87.5	86.931820	82119
Partición 1	91.818184	85.0	88.409092	83517
Partición 2	90.909094	80.0	85.454547	87325
Partición 3	85.454550	87.5	86.477275	85724
Partición 4	89.090910	87.5	88.295455	84351
Media	88.727276	85.5	87.113638	84607.2

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	72.881360	80.645160	76.763260	36203
Partición 1	75.438595	83.870965	79.654780	36975
Partición 2	70.175440	80.645160	75.410300	37098
Partición 3	75.438595	87.096775	81.267685	35388
Partición 4	80.701756	79.032260	79.867008	39430
Media	74.927149	82.258064	78.592607	37018.8

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	87.323946	91.176470	89.250208	28337
Partición 1	84.285710	88.235295	86.260503	32157
Partición 2	85.714287	88.235295	86.974791	28655
Partición 3	90.000000	88.235295	89.117648	29336
Partición 4	88.571430	88.235295	88.403363	29093
Media	87.179075	88.82353	88.001303	29515.6

8.2.8 *Algoritmo memético 3*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	86.363640	85.0	85.681820	86035
Partición 1	93.636364	85.0	89.318182	83503
Partición 2	89.090910	85.0	87.045455	82768
Partición 3	88.181820	85.0	86.590910	85647
Partición 4	94.545454	87.5	91.022727	82129
Media	90.363638	85.5	87.931819	84016.4

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	77.966100	87.096775	82.531438	34015
Partición 1	73.684210	85.483870	79.584040	36246
Partición 2	80.701756	82.258064	81.479910	36135
Partición 3	78.947370	85.483870	82.215620	37334
Partición 4	75.438595	83.870965	79.654780	36268
Media	77.347606	84.838709	81.093158	35999.6

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	85.915494	85.294116	85.604805	29627
Partición 1	84.285710	85.294116	84.789913	30035
Partición 2	85.714287	88.235295	86.974791	33477
Partición 3	94.285715	91.176470	92.731093	34594
Partición 4	82.857144	94.117650	88.487397	33916
Media	86.61167	88.823529	87.717600	32329.8

8.2.9 *Algoritmo memético 3 con pesos iniciales retocados*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	90.909094	87.5	89.204547	81846
Partición 1	92.727274	87.5	90.113637	82093
Partición 2	88.181820	87.5	87.840910	80126
Partición 3	87.272730	87.5	87.386365	81643
Partición 4	87.272730	85.0	86.136365	83795
Media	89.272730	87.0	88.136365	81900.6

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	74.576270	95.161290	84.868780	35961
Partición 1	70.175440	91.935486	81.055463	35307
Partición 2	73.684210	91.935486	82.809848	37517
Partición 3	75.438595	90.322580	82.880588	36316
Partición 4	73.684210	88.709676	81.196943	33586
Media	73.511745	91.612904	82.562324	35737.4

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
Partición 0	80.281690	91.176470	85.72908	28318
Partición 1	81.428570	91.176470	86.30252	28082
Partición 2	82.857144	88.235295	85.546220	28534
Partición 3	95.714283	91.176470	93.445377	28006
Partición 4	87.142855	91.176470	89.159663	35068
Media	85.484908	90.588235	88.036572	29601.6

8.3 *Tablas detalladas por algoritmo (Práctica 3).*

8.3.1 *Enfriamiento Simulado*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	90.000000	87.5	88.750000	58039
1	88.181820	87.5	87.840910	82668
2	90.000000	87.5	88.750000	83469
3	89.090910	87.5	88.295455	82597
4	90.909094	85.0	87.954547	83268
Media	89.636365	87.0	88.318182	78008.2

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	74.576270	90.322580	82.449425	38279
1	70.175440	87.096775	78.636108	44575
2	68.421054	83.870965	76.146010	40638
3	71.929824	88.709676	80.319750	44362
4	75.438595	87.096775	81.267685	41976
Media	72.108237	87.419354	79.763796	41966

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	88.732390	91.176470	89.954430	24864
1	81.428570	88.235295	84.831933	27357
2	87.142855	91.176470	89.159663	26891
3	94.285715	88.235295	91.260505	25478
4	90.000000	88.235295	89.117648	28250
Media	88.317906	89.411765	88.864836	26568

8.3.2 *Iterative Local Search*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	87.272730	85.0	86.136365	86745
1	88.181820	87.5	87.840910	86350
2	87.272727	87.5	87.386364	90772
3	90.909094	87.5	89.204547	88256
4	90.000000	87.5	88.750000	88302
Media	88.727274	87.0	87.863637	88085

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	76.271190	83.870965	80.071078	36873
1	73.684210	85.483870	79.584040	42682
2	73.684210	88.709676	81.196943	41835
3	71.929824	80.645160	76.287492	48716
4	77.192980	83.870965	80.531973	42500
Media	74.552483	84.516127	79.534305	42521.2

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	85.915494	91.176470	88.545982	25526
1	82.857144	91.176470	87.016807	27449
2	84.285710	88.235295	86.260503	25751
3	95.714283	88.235295	91.974789	27057
4	87.142855	91.176470	89.159663	28079
Media	87.183097	90.000000	88.591549	26772.4

8.3.3 *Evolución Diferencial 1*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	89.090910	87.5	88.295455	93232
1	89.090910	87.5	88.295455	99444
2	86.363640	90.0	88.18182	96713
3	85.454550	87.5	86.477275	98191
4	93.636364	87.5	90.568182	91191
Media	88.727275	88.0	88.363637	95754.2

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	74.576270	95.161290	84.868780	41559
1	70.175440	95.161290	82.668365	43632
2	71.929824	95.161290	83.545557	48562
3	70.175440	91.935486	81.055463	45881
4	71.929824	93.548390	82.739107	42226
Media	71.757360	94.193549	82.975454	44372

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	83.098590	91.17647	87.137530	30877
1	84.285710	91.17647	87.731090	34105
2	84.285710	91.17647	87.731090	32814
3	94.285715	94.11765	94.201683	34180
4	80.000000	94.11765	87.058825	32113
Media	85.191145	92.352942	88.772044	32817.8

8.3.4 *Evolución Diferencial 2*

Texture

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	92.727274	75.0	83.863637	88489
1	83.636360	85.0	84.31818	84385
2	89.090910	77.5	83.295455	92522
3	88.181820	80.0	84.09091	87733
4	88.181820	77.5	82.84091	89318
Media	88.363637	79.	83.681818	88489.4

Colposcopy

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	0.7457627	0.7419355	0.7438491	43545
1	0.68421054	0.6451613	0.66468592	48643
2	0.7017544	0.6935484	0.6976514	50231
3	0.7368421	0.6935484	0.71519525	45194
4	0.7719298	0.7580645	0.76499715	39518
Media	0.72809991	0.70645162	0.71727576	45426.2

Ionosphere

Partición	Tasa de clasificación	Tasa de reducción	Agregado	Tiempo (ms)
0	0.943662	0.85294116	0.89830158	30303
1	0.8	0.7941176	0.7970588	34016
2	0.85714287	0.88235295	0.86974791	34843
3	0.94285715	0.85294116	0.89789916	29651
4	0.9142857	0.7941176	0.85420165	30424
Media	0.89158954	0.83529409	0.86344182	31847.4

8.4 *Tablas generales por conjunto de datos.*

8.4.1 *Texture*

Algoritmo	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
1-NN	92.545455	0	46.272728	1	0
RELIEF	93.090910	5.5	49.295455	6	0
RELIEF 2	**93.272728**	5.5	49.386364	6	0
Greedy	36.745455	**97.5**	67.122727	5	0
Búsqueda local	86.363638	84.0	85.181819	17416.4	63.4
Búsqueda local 2	88.909092	85.5	87.204546	10826.2	56.6
Búsqueda local 3	90.545454	85.0	87.772727	7734.6	15.4
AGG-CA	90.727274	74.5	82.613637	102158.6	-
AGG-BLX	90.727275	83.0	86.863638	96629	-
AGE-CA	85.818180	77.5	81.659093	104484	-
AGE-BLX	89.636364	80.5	85.068182	94626.4	-
Memético 1	88.181820	86.0	87.090910	88259.8	-
Memético 1 2	90.363638	86.5	**88.431819**	87278.4	-
Memético 2	88.727276	85.5	87.113638	84607.2	-
Memético 3	90.363638	85.5	87.931819	84016	-
Memético 3 2	89.272730	87.0	88.136565	81900.6	

8.4.2 *Colposcopy*

Algoritmo	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
1-NN	75.266131	0	37.633066	0	0
RELIEF	75.979780	36.451612	56.215696	2.2	0
RELIEF 2	76.330658	36.451612	56.391135	2.2	0
Greedy	67.523045	**98.387100**	82.955073	1	0
Búsqueda local	75.254238	77.741935	76.498086	9379	64.8
Búsqueda local 2	75.967886	84.516127	80.242007	6964.8	59.2
Búsqueda local 3	73.113292	93.548387	**83.169549**	5417	8.8
AGG-CA	74.915254	60.645155	67.782050	46093.6	-
AGG-BLX	74.540589	75.161290	74.850939	41007.8	-
AGE-CA	72.447222	63.870971	68.159097	43980	-
AGE-BLX	74.600059	70.322582	72.461321	42118.6	-
Memético 1	71.733571	81.612903	76.673237	39221	-
Memético 1 2	71.501636	93.225808	82.262722	36905	-
Memético 2	74.927149	82.250864	78.592607	37018.8	-
Memético 3	**77.347606**	84.838709	81.093158	35999.6	-
Memético 3 2	73.511745	91.612904	82.562324	35737.4	-

8.4.3 *Ionosphere*

Algoritmo	Tasa clas.	Tasa red.	Agregado	Tiempo (ms)	Mutaciones
1-NN	86.599597	0	43.299799	0	0
RELIEF	87.456740	2.9411765	45.198958	2	0
RELIEF 2	87.456740	2.9411765	45.198958	2	0
Greedy	73.814889	**97.058820**	85.436855	1	0
Búsqueda local	86.032190	84.705881	85.369036	4203	60.6
Búsqueda local 2	87.179074	85.882352	86.526689	4874.8	63.6
Búsqueda local 3	87.166996	90.000000	88.583498	2342.6	11.8
AGG-CA	86.623741	74.117647	80.370694	37919	-
AGG-BLX	89.171026	88.235294	**88.703160**	32942	-
AGE-CA	88.036218	71.176471	79.606344	35774.8	-
AGE-BLX	**90.020119**	81.764705	85.892413	31704	-
Memético 1	84.325956	89.411764	86.868861	30965.8	-
Memético 1 2	84.901407	90.588235	87.744821	29965.6	-
Memético 2	87.179075	88.823530	88.001303	29515	-
Memético 3	86.611670	88.823530	87.717600	32329	-
Memético 3 2	85.484908	90.588235	88.035720	29601	-

8.5 *Analisis de los resultados (Práctica 1)*

Comenzamos primero con el **clasificador 1-NN**, todos los vectores de pesos son 1, por lo que su tasa de reducción es mínima, aun así, la tasa de clasificación en el mejor caso, *Texture*, es de un 92,5 % de media, la cual está a la altura del resto de algoritmos, incluso quedando por encima de la búsqueda local. Sin embargo, en el conjunto de datos *Colposcopy*, la tasa de clasificación es de un 75,2 %, aunque debemos tener en cuenta que el algoritmo con mejor tasa de clasificación en este conjunto ha obtenido 76,3 %. De forma que vemos que la clasificación es bastante buena. Debido a su tasa de reducción, el agregado no es muy alto, siendo siempre el mas bajo de todos los algoritmos. Los tiempos de ejecución vemos que son de $1ms$ en el conjunto de datos mas grande y menos de esto en el resto de conjuntos.

Veamos ahora los resultados del metodo **RELIEF**, mirando el primer conjunto de datos, *Texture*, vemos que la mejoría respecto a 1-NN no ha sido apenas notable, apenas un 0,5 % de mejora en la tasa de clasificación y solo un 5,5 % de los pesos se han visto reducidos, por ello el agregado es similar al de 1-NN, algo similar ocurre en el conjunto de datos *Ionosphere*, sin embargo, si miramos los resultados de *Cosposcopy* vemos que aquí la tasa de redución llega al 36 %, aunque la tasa de clasificación sigue cerca de los resultados del 1-NN. Esto es debido a que el algoritmo RELIEF no nos asegura una reducción de los pesos, esto dependerá del conjunto de datos y las partitiones. Aun así en todos los casos tiene un resultado en la función de evaluación mayor que el 1-NN, tardando aún milésimas de segundo en ejecutarse.

Podemos comparar ahora que ocurre si **no descartamos los pesos en RELIEF**, vemos que el tiempo y la tasa de reducción son los mismos en todos los conjuntos de datos, si que se produce una pequeña mejoría en la tasa de clasificación, lo cual me lleva a pensar que quizá el umbral de 0,2 es aún demasiado alto para descartar el peso. Cabe destacar que en los 3 conjuntos de datos es el que tiene mejor tasa de clasificación de media.

Miremos los resultados del algoritmo **greedy** (no RELIEF), la tasa de clasificación en el conjunto *Texture* no es muy alta, pero en los otros dos conjuntos aun estando por debajo del resto de algoritmos no es demasiado baja, la naturaleza del algoritmo le permite tener siempre la mayor tasa de reducción de entre todos los algoritmos. Los tiempos son similares a los de RELIEF debido a que se recorren los mismos bucles. Si la ponderación utilizada en la función de evaluación fuera otra, este algoritmo no tendría una puntuación tan alta.

Si miramos ahora los resultados de la **búsqueda local**, lo primero que llama la atención es que la tasa de clasificación no es mejor que las de 1-NN o RELIEF, en todo caso es peor que ambas, lo que si observamos es el gran cambio en la tasa de reducción que supera el 80 % en gran parte de las ejecuciones. Analizemos esto, en la evaluación estamos usando que un vector de pesos es mejor que otro cuando su función de evaluación es mayor, teniendo en cuenta que el vector de pesos es bastante mas pequeño que el vector de entrenamiento, reducir un peso tiene el mismo resultado que acertar varias clasificaciones más, lo cual es mas dificil, además, con el resto de algoritmos vemos que con estos conjuntos de datos, acertar gran parte de las clases no es dificil con unos pesos decentes, por ello no es sorprendente que los resultados tiendan a reducir un peso antes que mejorar la clasificación.

Ahora podemos comparar la búsqueda local normal, con la realizada utilizando **el vector inicial de RELIEF**, vemos que en muchas de las ejecuciones la tasa de clasificación y la de reducción son mejores, aun así, la diferencia no es demasiado significativa, decidí tener en cuenta el número de mutaciones aceptadas, y esperaba observar que en este segundo caso fuera menor, pero por lo general no ha sido así. Otra cosa a observar es la relación entre las mutaciones y el tiempo de ejecución.

Podemos ver que aunque esta segunda variante tarda menos que el original tanto en *Texture* como en *Colposcopy*, el número de mutaciones sigue siendo parecido, es decir, en la variante hay menos iteraciones entre mutaciones aceptadas y se llega antes el máximo local.

En este punto de la práctica fue cuando decidí probar a ejecutar la búsqueda local con los pesos que devuelve el algoritmo greedy que he añadido a la práctica, la razón es comprobar si de verdad las mutaciones estan dirigidas a aumentar la tasa de reducción. Efectivamente, si miramos los resultados vemos que esta segunda variante de las búsqueda local realiza muchas menos mutaciones y tarda menos tiempo que las demás, teniendo resultados similares y obteniendo la mejor puntuación de todos los algoritmos. Otra cosa que me llamó la atención es que en algunas particiones el resultado del greedy es mejor que el de la búsqueda local que utiliza sus pesos, esto me llevó a pensar que quizá la implementación estaba mal, pero luego reflexioné que la búsqueda local solo considera mejoras sobre el conjunto de entrenamiento, lo cual no quiere decir que sea una mejora respecto al conjunto de test posterior, aquí tenemos un ejemplo entonces de que la búsqueda local tambien puede empeorar los resultados dependiendo del conjunto de datos.

Vamos a ver ahora una gráfica comparando la búsqueda local normal con la que utiliza los pesos de RELIEF, (podríamos haber comparado también la otra variante, pero esta tiene una alta tasa de reducción desde el principio y como vamos a ver no es lo interesante de la gráfica). Tengamos en cuenta que la gráfica es solo con los datos de *Texture*.

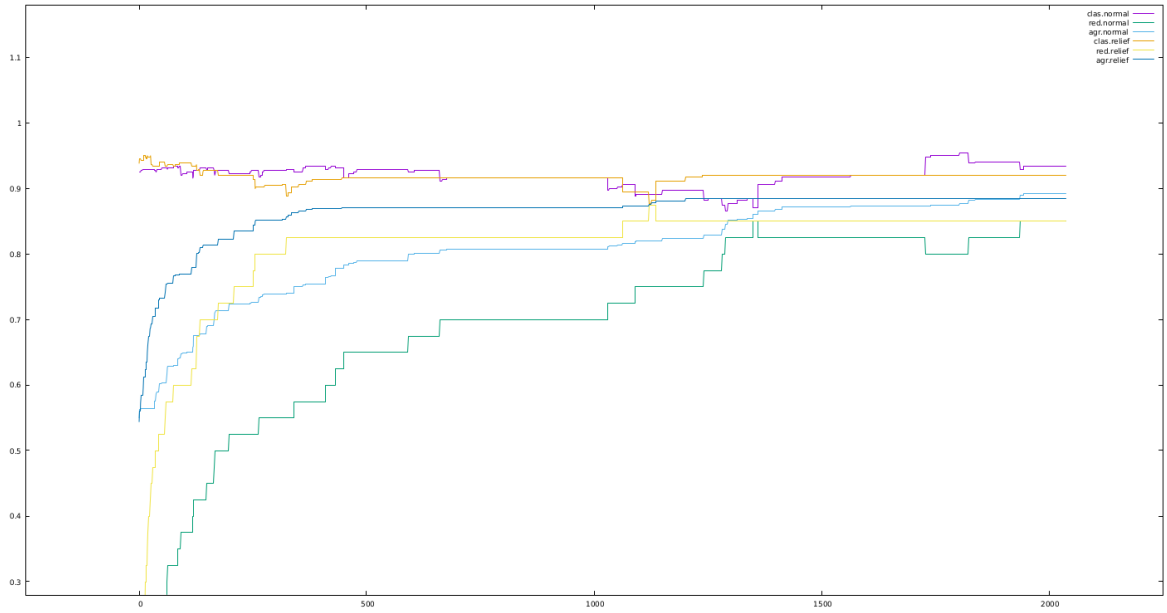

Figura 1: Evolución

Como podemos ver ambas tasas de reduccion comienzan en valores cercanos a 0, pero se puede ver como la tasa de reducción cuando los pesos iniciales son de RELIEF crece mas rapidamente, en consecuencia lo hace la función de evaluación. Esto se debe a que los pesos de RELIEF están más cercanos a la solución óptima, con lo cual las mutaciones de estos siguen siendo buenas soluciones y es mas fácil que sean aceptadas.

Además vemos que en los dos casos no se mejora apenas la tasa de clasificación, con iteraciones en las que se sacrifica parte de la tasa de clasificación en favor de aumentar la tasa de reducción, quizá sería buena idea cambiar la ponderación de 50 % que estamos usando si no se quiere perder tasa de clasificación.

La conclusión general sobre los resultados es que la tasa de clasificación debería tener un peso mayor en la función de evaluación para evitar que algoritmos como el greedy que he implementado tengan una puntuación tan alta, además la tasa de clasificación en 1-NN, RELIEF y la búsqueda local normal no son muy distintas, lo cual reduce el análisis a hablar sobre la tasa de reducción.

8.6 *Análisis de los resultados (Práctica 2)*

Comencemos hablando de los resultados de los algoritmos genéticos. Tanto en el generacional como en el estacionario podemos observar el siguiente fenónemo, los resultados con el cruce **BLX-α son mejores que con el cruce aritmético**, ya que si miramos los resultados con detalle vemos que la diferencia no se encuentra tanto en la tasa de clasificación sino en la de reducción, donde por ejemplo en `Colposcopy` la diferencia entre AGG-BLX y AGG-CA es de casi un 15 %. Esto se debe a que los resultados de cruce aritmético son siempre un valor entre los pesos de los padres, de forma que para que un peso quede por debajo de 0.2 ambos padres deberian tener dicho peso bastante bajo, lo cual no siempre ocurre. Sin embargo cuando utilizamos el cruce BLX-α si los pesos de los padres son 0.2 y 0.3, el hijo tiene la posibilidad de tener el peso reducido.

Comparemos ahora resultados entre el genético generacional y el genético estacionario. Por lo general el modelo generacional se ha desarrollado mejor que el estacionario en los 3 conjuntos de datos, sobre todo si nos fijamos en los resultados utilizando el cruce BLX, de todas formas las diferencias no son muy significativas. Por ello vamos a ver como evolucionan los pesos a lo largo de las generaciones en ambos. En el eje X se representa la generación a la que pertenece el peso y en el eje Y su valor de agregado.

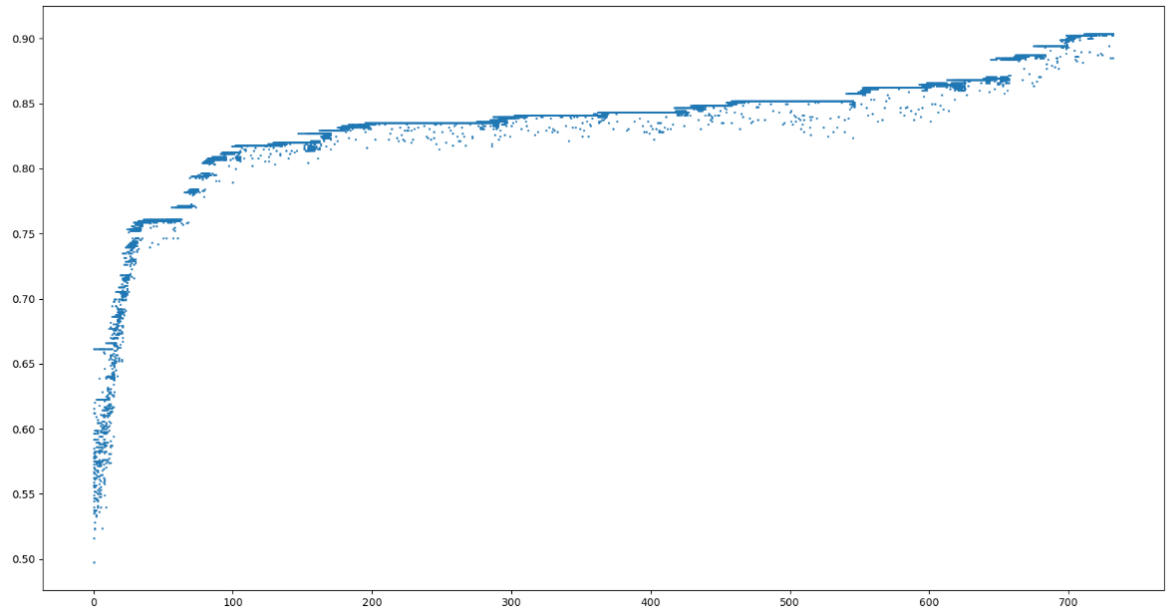

Figura 2: Generacional BLX

En el modelo **generacional** los pesos se encuentran algo mas dispersos que en el **estacionario** aunque a lo largo de las generaciones si se concentran mas en los valores más altos. El modelo de crecimiento de ambos es similar, en las primeras generaciones el crecimiento es muy rápido hasta encontrarse con un óptimo local, a partir de ahí, el crecimiento se basa en esperar a que una mutación o cruce saque a la generación de dicho óptimo, para volver a estancarse en otro.

También podemos observar que los pesos del generacional aún no habían convergido cuando ha terminado la ejecución, mientras que los del estacionario si. Vamos entonces a aumentar el número de llamadas a la función de evaluación a 25.000 para comprobar cuanto tarda el generacional en converger a un solo peso.

Ahora ya si podemos ver que el modelo generacional tambien acaba alcanzando un momento en el que todos los pesos son casi iguales, aunque no a tanto nivel como el estacionario.

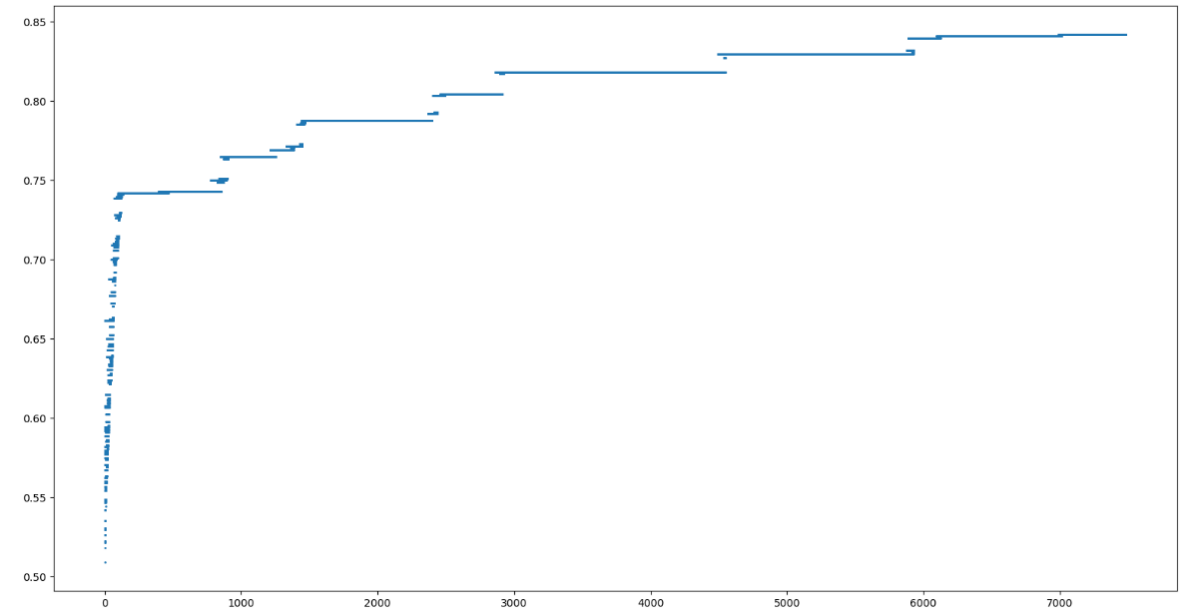

Figura 3: Estacionario BLX

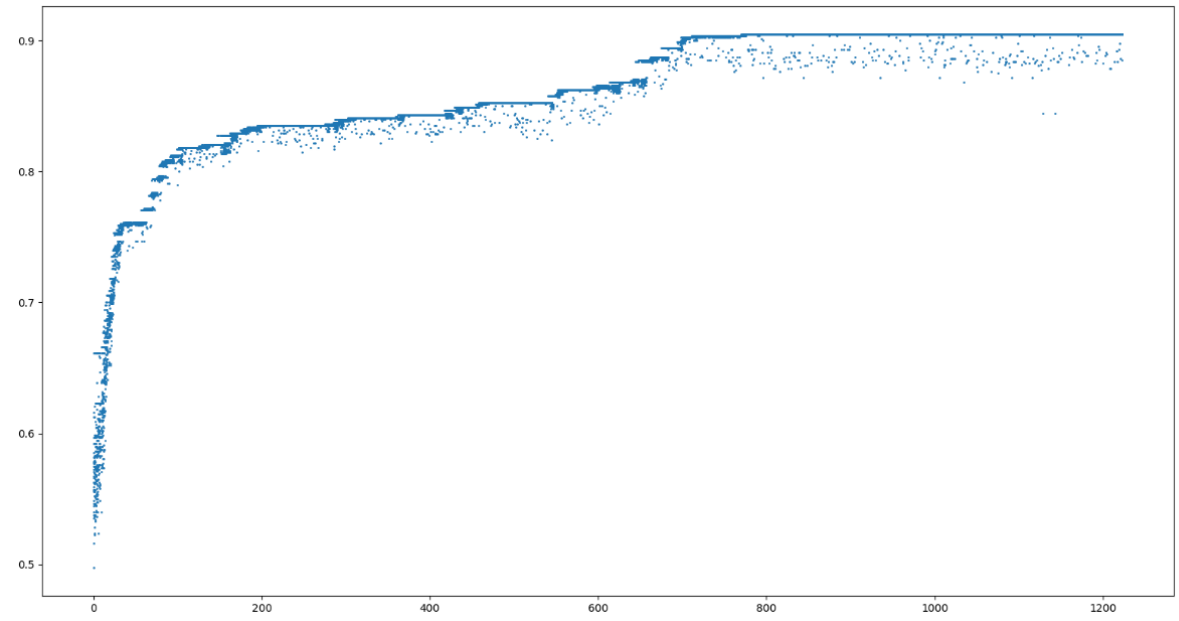

Figura 4: Generacional 2 BLX

Este es el motivo por el que en el algoritmo memético he utilizado el modelo generacional en lygar del estacionario. En caso de haber resultado mejor el estacionario que el generacional, habría optado por utilizarlo.

Llegados a este punto podemos estudiar los resultados de utilizar el operador de selección que he implementado, la idea es dar más importancia en la selección a aquellos elementos de la población que son mejores. Sin embargo, al utilizarlo los resultados obtenidos no han sido mejores que los del otro operador (por eso no aparecen en las tablas). Comprobando su funcionamiento vi que en la población

inicial todos los pesos tienen un valor de entre 0.4 y 0.6, de forma que todos tienen aproximadamente la misma proporción de ser elegidos. Además en el resto de generaciones los pesos se parecen cada vez más con lo que esto se acentúa. Si todos los operadores tienen mas o menos el mismo valor, este operador hace esencialmente lo mismo que una elección aleatoria. Este debe ser el motivo por el que los resultados no han mejorado. En otro conjunto de datos en el que la diferencia entre los elementos sea mayor debería dar mejores resultados.

Veamos ahora los resultados que han obtenido los diferentes algoritmos **meméticos**. Si no tenemos en consideración aquellos en cuya generación inicial están los pesos de RELIEF y del algoritmo Greedy, entonces los mejores resultados los tiene aquel que sólo aplica la búsqueda local al mejor elemento de la población en todos los conjuntos de datos. La razón de esto es que en los otros casos estamos gastando llamadas a la función de evaluacion en cromosomas que quizá son muy malos y no merece la pena, mientras que en otro caso sólo explotamos la mejor solución.

Veamos las gráficas de crecimiento de cada uno de ellos.

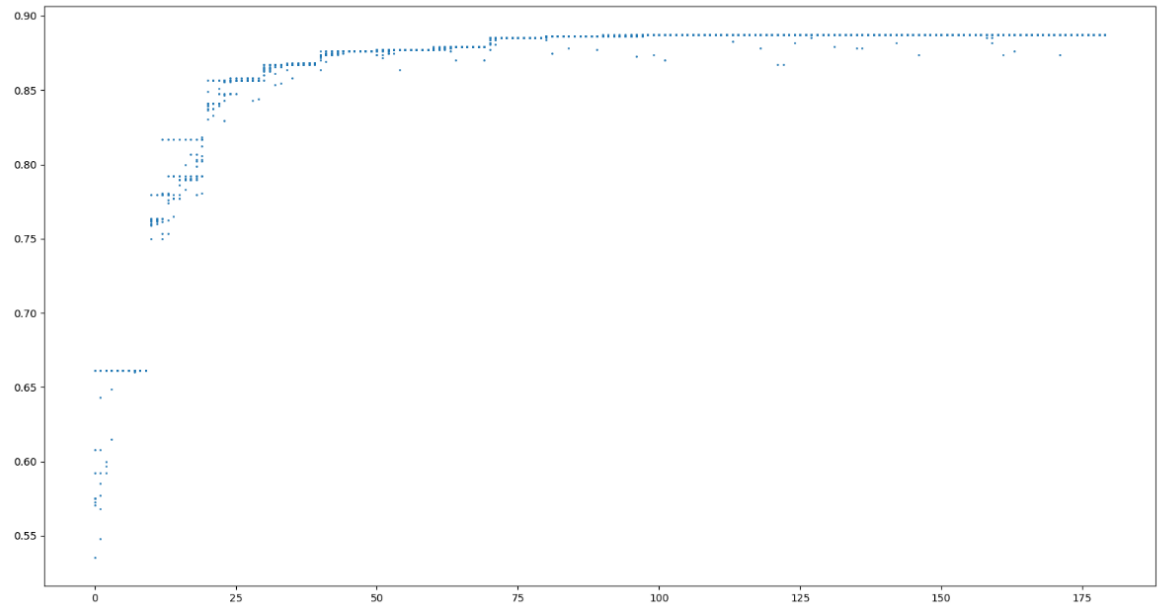

Figura 5: Memético 1

Vemos que en cada uno de ellos el crecimiento es diferente. Tanto en el primero como en el segundo, la solución explota las primeras 100 generaciones para luego estancarse.

Mientras que en el caso del tercero la solución alcanza un mínimo local en menos generaciones, unas 50, y luego consigue salir del él reiteradas veces.

En todos los casos los pesos se encuentran en un óptimo local al final de la ejecución.

Ahora podemos discutir como mejoran o empeoran los resultados si en la población inicial insertamos los pesos que nos devuelve RELIEF y el algoritmo Greedy.

Vemos que en todos los casos el resultado ha sido mejor que su alternativo sin dichos pesos. Podemos ver también como afecta su presencia al crecimiento del resto de elementos de la población. Como vemos ahora los pesos convergen mucho mas rápido al máximo local, si lo hacemos respecto a la tercera variante del memético el resultado es que se aplica la búsqueda local directamente al cromosoma del algoritmo Greedy, por ello es el que mejor resultados ha obtenido de entre todos los meméticos.

Comparemos ahora los resultados obtenidos por los algoritmos de esta segunda práctica tambien con los de la primera. Respecto a la tasa de clasificación, los mejores han sido RELIEF, la tercera

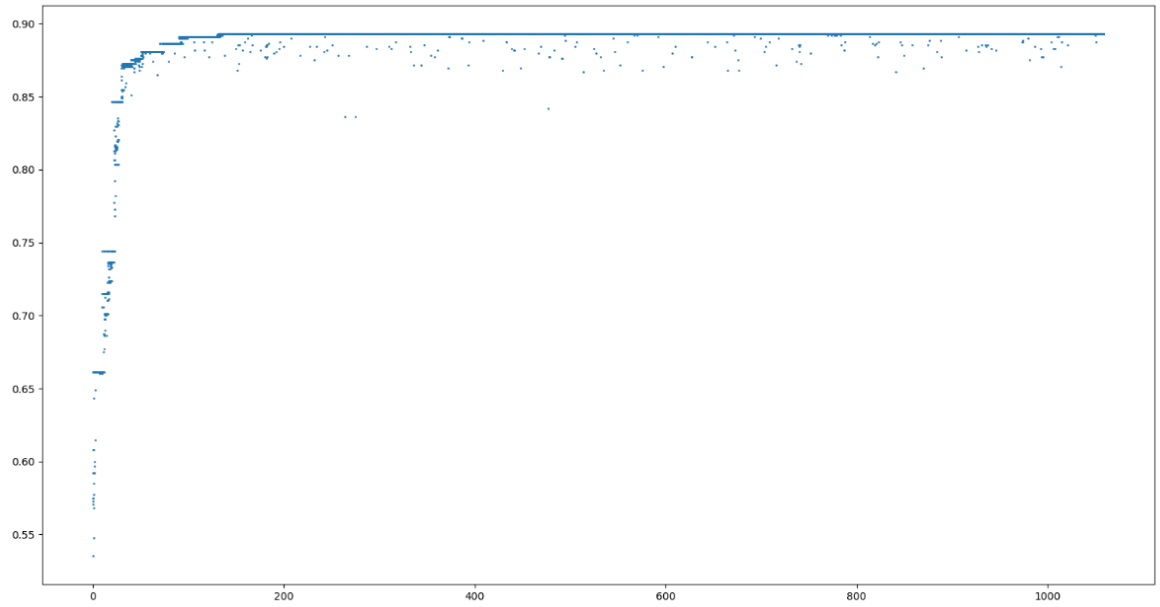

Figura 6: Memético 2

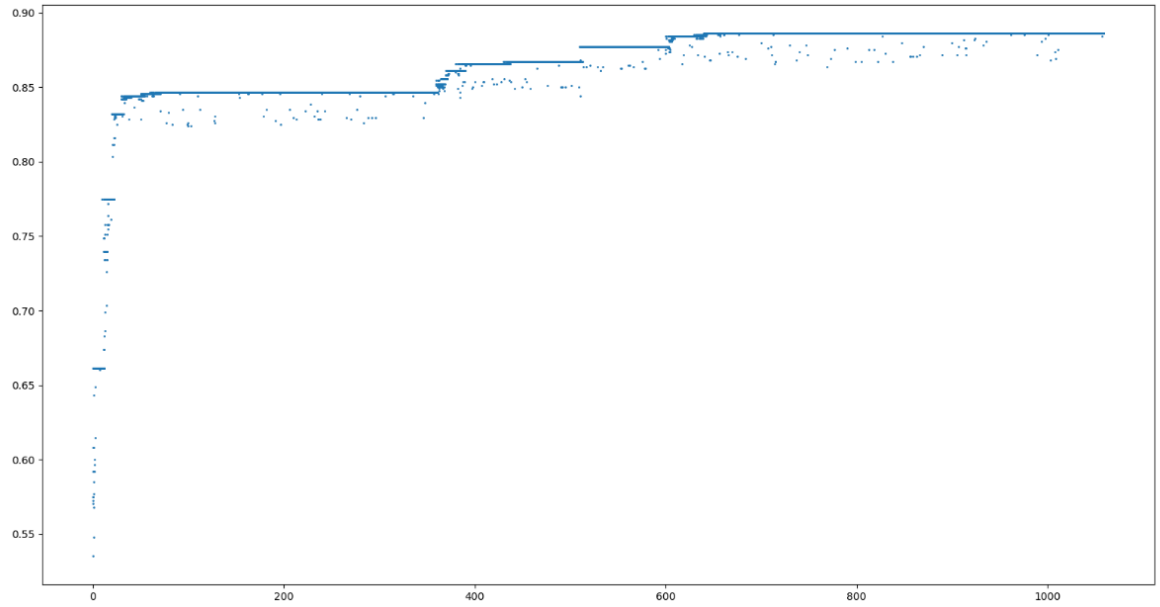

Figura 7: Memético 3

variante del memético y AGE-BLX. La tasa de reducción siempre va a ser óptima en el algoritmo Greedy desarrollado (recordemos que solo tiene 1 peso), pero por lo general los meméticos tienen mejor resultado que los genéticos, esto tiene sentido ya que en la práctica anterior ya vimos que la búsqueda local tiende a reducir los pesos. Respecto a los resultados en el agregado, en `Texture` el mejor resultado lo ha obtenido la primera variante del memético con los pesos iniciales retocados, aunque todos los meméticos han tenido resultados muy cercanos. En `Colposcopy` ninguno de los neuvos algoritmos ha conseguido llegar al valor que tenía la búsqueda local 3, aunque las variantes del memético con pesos retocados se han quedado muy cerca. En el último conjunto de datos, el mejor resultado lo tiene el AGG-BLX volviendo a quedarse muy cerca todos los meméticos.

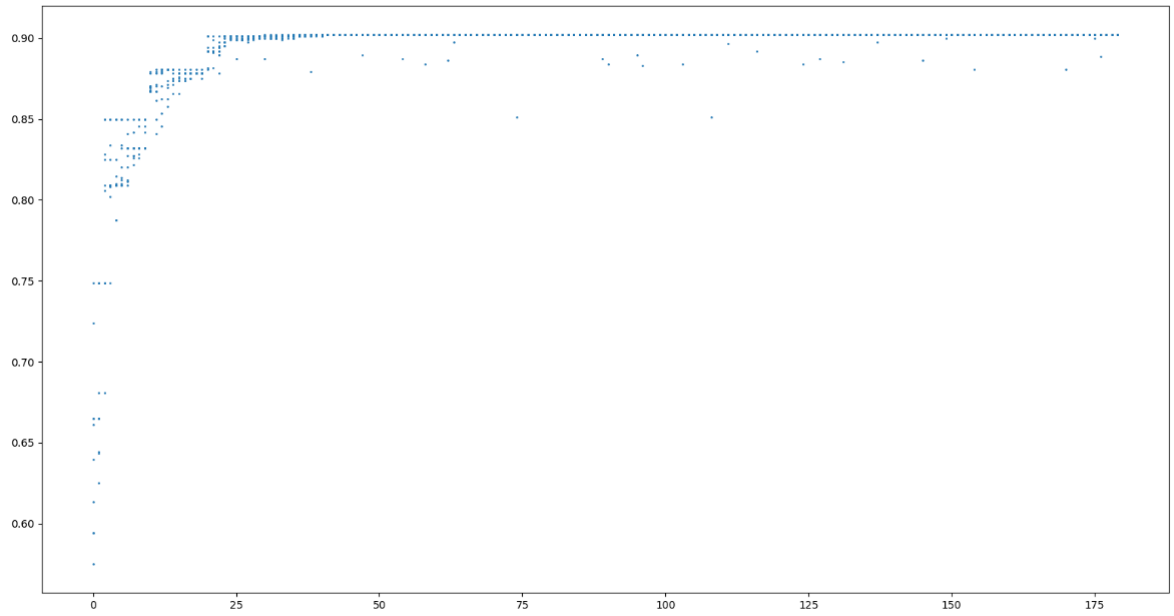

Figura 8: Memético 1 con pesos iniciales retocados

En resumen, podemos ver que los resultados de los meméticos son mejores que los de los algoritmos genéticos, y que aunque los segundos a veces sacan un mejor resultado, los meméticos no se quedan muy lejos. El tiempo de ejecución de todos es similar como cabría esperar y bastante mayor que la búsqueda local. Con lo cual en estos 3 conjuntos de datos vemos que si nos importa el tiempo de ejecución de nuestro algoritmo deberiamos quitarle llamadas a la función de evaluacion a los meméticos (ya que se estancan muy rapido y no salen de ahí) o cambiar el criterio de parada a uno que considere que se ha estancado el tiempo suficiente.