Práctica 3.b:

Enfriamiento Simulado, Búsqueda Local Reiterada y Evolución Diferencial para el Problema del Aprendizaje de Pesos en Características

Curso: 2018 - 2019
Alumno: Christian Vigil Zamora
DNI: [redacted]
Correo: [redacted]
Grupo: A2: Miércoles 17:30 - 19:30

Índice

1. DESCRIPCIÓN DEL PROBLEMA

El problema del Aprendizaje de Pesos en Características tiene como objetivo ajustar un conjunto de pesos asociados al conjunto total de características, con la finalidad de obtener un clasificador que de mejores resultados. Los datos de nuestro problema estarán formados por un conjunto de elementos, los cuáles tienen asociados a su vez un conjunto de características.
Para entrenar a nuestro clasificador recurrimos a la técnica de validación cruzada **5-fold cross validation**, con la que el conjunto de datos total se divide en 5 particiones manteniendo la distribución de clases equilibrada. Nuestro clasificador se entrenará con con la concatenación de 4 particiones, y será evaluado con la partición restante. En nuestro caso, vamos a usar un clasificador muy simple, el 1-NN cuyo funcionamiento está basado en la Regla del vecino más próximo, por lo que vamos a tener que calcular la distancia entre dos elementos e ir comparando el valor obtenido. Para realizar el cálculo de la distancia entre dos elementos recurrimos a la **distancia euclídea**, que viene dada por la fórmula:

$$d_e(e_1, e_2) = \sqrt{\sum_i w_i \cdot (e_1^i - e_2^i)^2 + \sum_j w_j \cdot d_h(e_1^j, e_2^j)}$$

Siendo Wi un elemento del vector de pesos asociado a cada características, representando un valor real en [0, 1].
Para analizar el rendimiento del clasificador utilizamos una **función de evaluación**:

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

que viene a ser una combinación de la **tasa_clas** (porcentaje de instancias correctamente clasificadas pertenecientes a un conjunto de datos, T):

$$tasa - clas = 100 \cdot \frac{n^o \text{ instancias bien clasificadas en } T}{n^o \text{ instancias en } T}$$

de la **tasa_red** (porcentaje de características descartadas):

$$tasa - red = 100 \cdot \frac{n^o \text{ valores } wi < 0.2}{n^o \text{ características}}$$

y de $\alpha \varepsilon [0, 1]$ (pondera la importancia entre el acierto y la reducción de la solución encontrada), teniendo como objetivo obtener el conjunto de pesos W que maximiza esta función.

2. APLICACIÓN DE LOS ALGORITMOS EMPLEADOS AL PROBLEMA

Operadores comunes y Función Objetivo

- **euclidean_distance** : Función que calcula la distancia euclídea entre 2 elementos mediante la sumatoria de la resta de las características de un elemento con otro, al cuadrado. Para reducir el tiempo de cálculo, he suprimido el cálculo de la raíz cuadrada. "np.sum((elem1 - elem2)**2)" devuelve la sumatoria de lo anterior comentado, directamente. Pseudocódigo:

```
Function euclidean_distance (elem1,elem2):
        distance = 0.0
        distance = np.sum((elem1 - elem2)**2)
        Return distance
```

- **closest_examples** : Función que dado un ejemplo, devuelve su amigo más cercano (distancia mínima entre el ejemplo y él, y ambos comparten clase) y su enemigo más cercano (distancia mínima entre el ejemplo y él, y ambos con clase diferente). "**np.inf**" otorga a una variable el valor infinito. Su funcionamiento se basa en que por cada elemento de los datos, se calcula la distancia entre ese elemento y el ejemplo dado como argumento. Si esa distancia es la mínima posible, se comprueba la etiqueta de ambos elementos, si coinciden significa que es el amigo más cercano, sino, el enemigo. Pseudocódigo:

```
Function closest_examples (datos, ejemplo, etiquetas)
        closest_enemy = 0
        closest_friend = 0
        etiqueta_ejemplo = etiquetas[ejemplo]
        enemy_dist = np.inf(∞)
        friend_dist = np.inf(∞)

        for i = 0 until n°datos
                if i != ejemplo
                        dist = euclidean_distance(ejemplo,datos[i])
                        if dist < enemy_dist and etiqueta_ejemplo != etiquetas[i]
                                enemy_dist = dist
                                closest_enemy = i
                        if dist < enemy_dist and etiqueta_ejemplo == etiquetas[i]
                                friend_dist = dist
                                closest_friend = i

        Return datos[closest_enemy], datos[closest_friend]
```

- **evaluation_function** : Función de evaluación en la que se mide el rendimiento del clasificador. Recibe las etiquetas de los valores con los que se va a testear, la predicción dada por el clasificador, el vector de pesos asociado a las características y alpha). "**np.count.nonzero(condicion)**" devuelve el resultado de contabilizar cuántas veces se cumple una condición dada. En éste caso, para la tasa_clas devuelve el número de veces que la predicción se corresponde con el valor de la etiqueta asociado a cada elemento (success), y para calcular el porcentaje de tasa_clas, se divide el valor de 'success' entre el número total de etiquetas. Para la tasa red, devuelve el número de elementos del vector de pesos que tienen un valor por debajo de 0.2 (under_value), y para calcular el porcentaje de tasa_red, se divide el valor de 'under_value' entre el número total de etiquetas. Una vez obtenidos éstos valores, es posible calcular la función objetivo, cuyo resultado se almacena como Agregado. Por último, se devuelve el valor de la tasa_clas, la tasa_red y el Agregado para esos datos. Pseudocódigo:

```
Function evaluation_function (prediccion, etiquetas_test, W, α)
        success = 0
        under_value = 0

        success = np.count_nonzero(prediccion == etiquetas_test)
        tasa_clas = (100 * (success / n°etiquetas_test)

        under_value = np.count.nonzero(W < 0.2)
        tasa_red = (100 * (under_value / n°caracteristicas(W))

        agr = α * tasa_clas + (1 - α) * tasa_red

        Return tasa_clas, tasa_red, agr
```

- **mean_calculus** : Función que calcula la media de los resultados obtenidos por la tasa_clas, tasa_red, agregaciones y tiempos en las 5 particiónes de cada ejecución. "**np.sum(valores)**" devuelve la sumatoria de los valores que se le pasan. Pseudocódigo:

```
Function mean_calculus(valores)
        summ = np.sum(valores)
        Return summ / 5.0
```

- **display_table** : Función que lleva a cabo la representación de las soluciones. Recibe la tasa_clas, la tasa_red, las agregaciones y los tiempos de ejecución de cada partición tras la ejecución de un algoritmo y las representa. Por último, se muestra la media de los resultados obtenidos en las 5 particiones. Pseudocódigo:

```
Function display_table (tasa_clas, tasa_red, agregaciones, times)

        print(tasa_clas[0], tasa_red[0], agregaciones[0], times[0])
        .
        .
        print(tasa_clas[4], tasa_red[4], agregaciones[4], times[4])
        print(mean_calculus(tasa_clas),mean_calculus(tasa_red),
                mean_calculus(agregaciones),mean_calculus(times))
```

- Procedimiento común a todos los Algoritmos:

Para realizar la división del conjunto de datos en 5 particiones manteniendo las proporciones de cada clase, utilizo la función:

```
skf = StratifiedKFold(n_splits=5,shuffle=True, random_state=1)
```

Dicha función divide los datos mediante la técnica de validación cruzada. Los datos son tomados aleatoriamente. Devuelve los índices de la división de los datos para separar en datos de entrenamiento y datos de testeo:

```
for indice_train, indice_test until skf.split(datos,etiquetas)
        datos_train= datos[indice_train]
        etiquetas_train = etiquetas[indice_train]
        datos_test = datos[indice_test]
        etiquetas_test = etiquetas[indice_test]
```

Para construir el clasificador K-NN, utilizo:

```
neigh = KNeighborsClassifier(n_neighbors = 1)
```

El entrenamiento del clasificador se realiza con los datos y etiquetas de entrenamiento, utilizo:

```
neigh.fit(datos_train,etiquetas_train)
```

Una vez entrenado el clasificador, predice la etiqueta de los datos de testeo, utilizo: `prediction = neigh.predict(datos_test)`.

PRÁCTICA 2

1. Esquema de representación de soluciones empleado

Las soluciones las he representado de igual forma, mediante un vector de pesos W asociados a cada característica del conjunto de datos, con unos valores entre [0,1]. Como novedad, para gestionar los elementos de una población, que no son más que vectores de pesos W, he creado una clase 'element' con la que cada objeto de la clase es un vector de pesos W y cuyos atributos son el valor de tasa_clas, tasa_red y agr para ese vector.

```
class element
        self.w
        neigh = KNeighborsClassifier(n_neighbors=1)
        w_c = np.copy(self.w)
        w_c[w_c < 0.2] = 0.0
        neigh.fit(x_train*w_c,y_train)
idx = neigh.kneighbors(x_train*w_c,n_neighbors=2,return_distance=False)
        pred = y_train[idx[:,1]]
        Clas, Red, Agr = evaluation_function(pred,y_train,self.w,0.5)
        self.agr = Agr
        self.clas = Clas
        self.red = Red
```

Para obtener los valores de tasa_clas, tasa_red y agr mantengo el mismo procedimiento, es decir, entreno al clasificador con los datos de entrenamiento eliminando los pesos con valor por debajo de 0.2 y a la hora de evaluar la función objetivo, el segundo vecino más cercano con la finalidad de cumplir con el **'leave one out'**.

2. Descripción en pseudocódigo de la función objetivo

Mantengo la descripción de la práctica anterior:
- **evaluation_function** : Función de evaluación en la que se mide el rendimiento del clasificador. Recibe las etiquetas de los valores con los que se va a testear, la predicción dada por el clasificador, el vector de pesos asociado a las características y alpha). "**np.count.nonzero(condicion)**" devuelve el resultado de contabilizar cuántas veces se cumple una condición dada. En éste caso, para la tasa_clas devuelve el número de veces que la predicción se corresponde con el valor de la etiqueta asociado a cada elemento (success), y para calcular el porcentaje de tasa_clas, se divide el valor de 'success' entre el número total de etiquetas. Para la tasa red, devuelve el número de elementos del vector de pesos que tienen un valor por debajo de 0.2 (under_value), y para calcular el porcentaje de tasa_red, se divide el valor de 'under_value' entre el número total de etiquetas. Una vez obtenidos éstos valores, es posible calcular la función objetivo, cuyo resultado se almacena como Agregado. Por último, se devuelve el valor de la tasa_clas, la tasa_red y el Agregado para esos datos. Pseudocódigo:

```
Function evaluation_function (prediccion, etiquetas_test, W, α)
        success = 0
        under_value = 0

        success = np.count_nonzero(prediccion == etiquetas_test)
        tasa_clas = (100 * (success / nºetiquetas_test)

        under_value = np.count.nonzero(W < 0.2)
        tasa_red = (100 * (under_value / nºcaracteristicas(W))

        agr = α * tasa_clas + (1 - α) * tasa_red

        Return tasa_clas, tasa_red, agr
```

3. Pseudocódigo del proceso de generación de soluciones aleatorias

El proceso de generación de soluciones aleatorias es muy sencillo. Dado el tamaño que deseamos que tenga nuestra población, generamos tantas soluciones como indique el tamaño. Las soluciones (w) las generamos mediante una distribución uniforme entre 0 y 1, cuyo tamaño es el número de características que tenga que el conjunto de datos considerado. Tras ésto, se crea un objeto de la clase 'elemento' pues cada solución generada es un elemento de la población y se inserta posteriormente en la población. Pseudocódigo:

```
poblacion= []
For 0 to tamaño_poblacion
        w = np.random.uniform(0,1,num_caracteristicas)
        ele = element(w,x_train,y_train)
        poblacion.append(ele)
End
```

4. Pseudocódigo de la selección de los AGs y los operadores de cruce y mutación

La selección de los AGs es llevada a cabo mediante la función **binaryTournament**, la cual se encarga de generar 2 números aleatorios para seleccionar 2 elementos de la población. Comprueba que no sean el mismo para evitar comparar el mismo elemento. Tras ésto se evalúa cuál de los 2 elementos seleccionados tiene un mayor valor de agregado, y se devuelve dicho elemento. Si nos encontramos en un algoritmo AGG, la función será llamada tantas veces como elementos tenga la población en cada iteración, pues se selecciona el mismo número de padres que de elementos tiene la población, mientras que si estamos en un AGE, la función será llamada exclusivamente 2 veces iteración, pues se seleccionan sólo 2 padres.

```
Function binaryTournament(población)
        r1 = genero número aleatorio entre 0 y tamaño población
        r2 = genero número aleatorio entre 0 y tamaño población

        While r1 == r2
                r2 = genero número aleatorio entre 0 y tamaño población
        End

        If población[r1].agr > población[r2].agr
                Return población[r1]
        End
        Else Return población[r2]
        End
End
```

En cuánto a los operadores de cruce, el primero de ellos es el Cruce BLX. Su funcionamiento consiste en generar 2 descendientes fruto del cruce de 2 padres. Para ello, por cada elemento de la 'w' de cada padre, se obtiene el máximo y el mínimo valor entre ambos, y se genera un valor perteneciente al intervalo que se muestra en el pseudocódigo. Se normaliza ese valor y se inserta en el hijo. Así sucesivamente hasta recorrer todos los elementos de 'w' de los padres. Realizando el mismo procedimiento 2 veces para obtener los 2 descendientes. Pseudocódigo:

```
Function BLXCross(padre1, padre2, alpha, x_train, y_train)
        descendientes = []
        For 0 to 2
                hijo = []
                For 0 to tamaño_solucion
                        Cmax = max(padre1.w, padre2.w)
                        Cmin = min(padre1.w, padre2.w)
                        I = Cmax - Cmin
                        valor = np.random.uniform(Cmin-I*alpha, Cmax+I*alpha)

                        Se normaliza valor
                        hijo.append(valor)
                End
                he = element(hijo,x_train,y_train)
                descendientes.append(hijo)
        End
        Return descendientes
End
```

El segundo es el Cruce Aritmético. Con el objetivo de mejorar los resultados, he modificado al método de cruce con respecto del seminario. En éste cruce, se genera un número aleatorio entre 0 y 1, alpha. A continuación, se generan los 2 descendientes. Ambos se generan sumando el valor de 'w' del padre 1 más el valor de 'w' del padre 2, con la diferencia de que para uno, los valores del padre1 se multiplican por alpha y los del padre2 por (1 - alpha), mientras que para el otro ocurre al revés, los valores del padre1 se multiplican por (1 - alpha) y los del padre2

por alpha a secas. De ésta forma se generan 2 hijos, cuyos valores son normalizados posteriormente e incluídos como elementos de la población. Pseudocódigo:

```
Function arithmeticCross(padre1,padre2,x_train,y_train)
        descendientes = []
        descendiente = []
        descendiente2 = []

        alpha = genero un número aleatorio entre 0 y 1
        descendiente = (padre1.w * alpha + padre2.w * (1 - alpha)
        descendiente2 = (padre1.w * (1 - alpha) + padre2.w * alpha)

        Normalizo los valores de ambos descendientes

        descendiente, descendiente2 = element(descendiente, descendiente2, x_train,y_train)

        descendientes.append(descendiente, descendiente2)

        Return descendientes
```

Por último voy a hablar del proceso de mutación. En el AGG, calculamos el número de mutaciones a realizar multiplicando el tamaño de la población por el tamaño del vector 'w' por la probabilidad de mutar un gen, que en éste caso es 0.001. Dicho valor se redondea por si se obtiene un valor de la forma 0.X . Una vez tenemos el número de mutaciones, escogemos aleatoriamente que cromosoma y que gen van a mutar, y se realiza la mutación de la misma forma que en la Búsqueda Local, mediante un valor 'z' que sigue una distribución normal. Posteriormente se normaliza el valor de la mutación obtenido. Pseudocódigo:

```
numero_mutacion = 0.001 * (tamaño_poblacion * tamaño_solucion(w))
For 0 to numero_mutaciones
        cro = número aleatorio para decidir que cromosoma muta
        gen = número aleatorio para decidir que gen muta
        z = np.random.normal(0,0.3,1)
        poblacion[cro].w[gen] += z

        Normalizo el valor obtenido
End
```

En el AGE, la cosa cambia. Puesto que tenemos sólo 2 hijos, sabemos que el número de mutaciones como máximo va a ser 2. Se calcula la probabilidad de mutar a nivel de cromosoma multiplicando la probabilidad de mutar a nivel de gen (0.001) por el tamaño de 'w'. Luego, para cada hijo, se calcula un número aleatorio entre 0 y 1, y si ese número obtenido es menor que la probabilidad de mutar a nivel de cromosoma, se muta. Se genera otro número aleatorio para conocer que gen va a mutar y el resto, es el mismo procedimiento que en el AGG. Pseudocódigo:

```
pm_cromosoma = 0.001 * tamaño_solucion(w)
h = número aleatorio entre 0 y 1
If h < pm_cromosoma
        gen = número aleatorio entre 0 y tamaño_solucion(w)
        z = np.random.normal(0,0.3,1)
        hijo.w[gen] += z

        Normalizo el valor obtenido
End
```

PRÁCTICA 3

1. Esquema de representación de soluciones empleado

Las soluciones las he representado de igual forma, mediante un vector de pesos W asociados a cada característica del conjunto de datos, con unos valores entre [0,1]. Para obtener los valores de tasa_clas, tasa_red y agr mantengo el mismo procedimiento, es decir, entreno al clasificador con los datos de entrenamiento eliminando los pesos con valor por debajo de 0.2 y a la hora de evaluar la función objetivo, el segundo vecino más cercano con la finalidad de cumplir con el **'leave one out'**. Para los 2 modelos de algoritmo Evolución Diferencial, las soluciones han sido representadas mediante el esquema seguido por los algoritmos de la práctica 2, es decir, la clase **element**, la cual representa a un individuo de la población. Pseudocódigo incluído en la página 6 de la memoria.

2. Descripción en pseudocódigo de la función objetivo

Mantengo la descripción de la práctica anterior:
- **evaluation_function** : Función de evaluación en la que se mide el rendimiento del clasificador. Recibe las etiquetas de los valores con los que se va a testear, la predicción dada por el clasificador, el vector de pesos asociado a las características y alpha). "**np.count.nonzero(condicion)**" devuelve el resultado de contabilizar cuántas veces se cumple una condición dada. En éste caso, para la tasa_clas devuelve el número de veces que la predicción se corresponde con el valor de la etiqueta asociado a cada elemento (success), y para calcular el porcentaje de tasa_clas, se divide el valor de 'success' entre el número total de etiquetas. Para la tasa red, devuelve el número de elementos del vector de pesos que tienen un valor por debajo de 0.2 (under_value), y para calcular el porcentaje de tasa_red, se divide el valor de 'under_value' entre el número total de etiquetas. Una vez obtenidos éstos valores, es posible calcular la función objetivo, cuyo resultado se almacena como Agregado. Por último, se devuelve el valor de la tasa_clas, la tasa_red y el Agregado para esos datos. Pseudocódigo:

```
Function evaluation_function (prediccion, etiquetas_test, W, α)
        success = 0
        under_value = 0

        success = np.count_nonzero(prediccion == etiquetas_test)
        tasa_clas = (100 * (success / nºetiquetas_test)

        under_value = np.count.nonzero(W < 0.2)
        tasa_red = (100 * (under_value / nºcaracteristicas(W))

        agr = α * tasa_clas + (1 - α) * tasa_red

        Return tasa_clas, tasa_red, agr
```

3. Pseudocódigo del proceso de generación de soluciones aleatorias

El proceso de generación de soluciones aleatorias es muy sencillo, para los algoritmos ES e ILS las soluciones se generan mediante una distribución uniforme entre 0 y 1. El tamaño de esa solución será el número de características de nuestros datos. Pseudocódigo:

```
w = np.random.uniform(0,1,num_caracteristicas)
```

Para los algoritmos de DE, mantengo la descripción de la práctica anterior: Dado el tamaño que deseamos que tenga nuestra población, generamos tantas soluciones como indique el tamaño. Las soluciones (w) las generamos mediante una distribución uniforme entre 0 y 1, cuyo tamaño es el número de características que tenga que el conjunto de datos considerado. Tras ésto, se crea un objeto de la clase 'elemento' pues cada solución generada es un elemento de la población y se inserta posteriormente en la población. Pseudocódigo:

```
poblacion= []
For 0 to tamaño_poblacion
        w = np.random.uniform(0,1,num_caracteristicas)
        ele = element(w,x_train,y_train)
        poblacion.append(ele)
End
```

4. Pseudocódigo del algoritmo de BL empleado, incluyendo el método de exploración del entorno y el operador de generación de vecino

```
Function LS_ILS(x_train, y_train, initial_solution)

w = initial_solution

neigh = KNeighborsClassifier(n_neighbors=1)
w_c = copy(w)
w_c[w_c < 0.2] = 0.0
neigh.fit(x_train*w_c, y_train)
vecinos = neigh.kneighbors(x_train*w_c,n_neighbors=2,return_distance=False)
prediccion = y_train[vecinos[:, 1]]

last_clas, last_red, last_agr = evaluation_function(predicción,y_train,w,0.5)
ev = 1

While ev  < 1000
        For i to tamaño_w
                w_copy  = copy(w)
                z = np.random.normal(0,0.3,1)
                w[i] += z

                Normalizo el valor

                neigh = KNeighborsClassifier(n_neighbors=1)
                w_c = copy(w)
                w_c[w_c < 0.2] = 0.0
                neigh.fit(x_train*w_c, y_train)
                vecinos = neigh.kneighbors(x_train*w_c,n_neighbors=2,return_distance=False)
                prediccion = y_train[vecinos[:, 1]]
                new_clas, new_red, new_agr = evaluation_function(predicción,y_train,w,0.5)
                ev += 1
                If new_agr > last_agr
                        last_agr = new_agr
                        break
                End
                Else
                        w = w_copy
                End
        End
End

Return w,
```

3. DESCRIPCIÓN EN PSEUDOCÓDIGO DE LA ESTRUCTURA DEL MÉTODO DE BÚSQUEDA

```
Function local_search (datos, etiquetas):

    tasa_clas = []
    tasa_red = []
    times = []
    agregacion = []

    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
    for train_index, test_index in skf.split(datos, etiquetas):
        start = time.time()
        x_train, y_train = datos[train_index], etiquetas[train_index]
        x_test, y_test = datos[test_index], etiquetas[test_index]

        w = np.random.uniform[0,1]
        n = 1
        iteraciones = 1
        neigh = KNeighborsClassifier(n_neighbors=1)
        w_c = np.copy(w)
        w_c[w_c < 0.2] = 0.0
        neigh.fit(x_train*w_c, y_train)
        vecinos = neigh.kneighbors(x_train*w_c,n_neighbors=2,return_distance=False)
        prediccion = y_train[vecinos[:, 1]]
        last_clas, last_red, last_agr = evaluation_function(predicción,y_train,w,0.5)

        while n < 20*n°caracteristicas and iteraciones < 15000
            for i=0  until n°caracteristicas
                w_copy = w.copy()
                z = np.random.normal[0,0.3]
                w[i] += z

                if w[i] < 0.0
                    w[i] = 0.0
                else if w[i] > 1.0
                    w[i] = 1.0

            neigh = KNeighborsClassifier(n_neighbors=1)
                w_c = np.copy(w)
                w_c[w_c < 0.2] = 0.0
                neigh.fit(x_train*w_c,y_train)
                idx =
            neigh.kneighbors(x_test*w_c,n_neighbors=2,return_distance=False)

                prediccion= y_train[idx[:,1]]
                new_clas, new_red, new_agr =
                evaluation_function(predicción,y_train,w,0.5)
                iteraciones += 1

                if new_agr > last_agr
                    last_agr = new_agr
```

```
                    break
                else
                    w = w_copy
                    n += 1

        prediccion = neigh.predict(x_test*w)
        new_clas, new_red, new_agr = evaluation_function(y_test,predicción,w,0.5)

        end = time.time()
        final_clas = new_clas
        final_red = new_red
        final_agr = new_agr

        tasa_clas.append(final_clas)
        tasa_red.append(final_red)
        agregacion.append(final_agr)
        times.append(end-start)

    Return tasa_clas, tasa_red, agregacion, times
```

El algoritmo de la Búsqueda Local tiene dos condiciones de salida: generar 20*n°caracteristicas vecinos o evaluar la función objetivo 15.000 veces. Mientras no se cumpla alguna de las condiciones, el algoritmo va mutando el vector de pesos en busca de una solución que mejore los resultados anteriores. Su ejecución es tan sencilla como que en cada iteración va mutando un peso, si la solución mejora los resultados, se pasa a mutar el siguiente peso dejando el peso anterior con el valor de la mutación, si no mejora, se pasa a mutar el siguiente peso sin considerar el valor de la mutación anterior. Finalmente, el algoritmo devuelve los valores tasa_clas, tasa_red, agregacion y times obtenido en cada iteración considerando la mejor solución posible, es decir, el vector de pesos que maximiza la función de evaluación. Las funciones que no explico en éste apartado es porque han sido explicadas en el apartado **2.)**, ya que son comunes a todos los algoritmos. **Funciones a explicar:**
"**np.random.uniform[0,1]**" genera el vector de pesos inicial de forma aleatoria, mediante una distribución uniforme en el intervalo [0, 1].
"**neigh.kneighbors**" devuelve el índice del vecino más cercano dados unos datos. El hecho de recibir como argumento que el número de vecinos sea 2 es para cumplir la técnica de validación **leave-one-out**, es decir, como el vecino más próximo vamos a ser nosotros mismos, obtengo 2 vecinos y me quedo con el segundo. "**np.random.normal[0,0.3]**" genera el valor para llevar a cabo la mutación de un peso y es generado mediante una distribución normal de media 0 y desviación típica 0.3. "**copy()**" es usada para obtener una copia del vector de pesos. "**append**" es la función usada para insertar los resultados obtenido en cada partición.

* ALGORITMO GENÉTICO GENERACIONAL *

Voy a comenzar comentando el esquema de evolución y reemplazo seguido en éste algoritmo. Primeramente, partimos de una **población inicial** generada aleatoriamente. En ése momento ya habremos realizado tantas evaluaciones de la función objetivo como elementos tenga la población inicial. Tras ésto, se entra en un bucle del que no se sale hasta que se completen 15.000 evaluaciones de la función objetivo. Dentro ya del bucle, lo primero que tiene lugar es el proceso de **SELECCIÓN**, en el que se van a elegir mediante la función **binaryTournament** tantos padres como elementos tenga la población inicial. De esa forma, la población actual estaría formada por 30 padres. Una vez seleccionados los padres, tiene lugar el proceso de **CRUCE**. En éste proceso, independientemente del método de cruce que se haya elegido, BLX o Aritmético, el número de parejas que van a cruzar es el mismo, en éste caso 0.7 por el número de parejas. Por lo tanto, ahora la población estaría formada por tantos hijos como se haya generado en el cruce MÁS los padres que no han cruzado. Llegados a éste punto, tiene lugar el proceso de **MUTACIÓN**. En éste proceso nuevamente sabemos el número de mutaciones que van a tener lugar multiplicando la probabilidad de que mute un gen, que es 0.001 en éste caso por el producto del tamaño de población actual y tamaño del vector de pesos. En cada mutación, se selecciona aleatoriamente el cromosoma que va a mutar y el gen que va a mutar. Una vez seleccionados, se muta mediante una distribución normal de media 0 y desviación típica 0.3, se normaliza el valor obtenido en la mutación y se actualiza ese valor en la población. Llegados a éste punto, tiene lugar el proceso de **REEMPLAZO** con elitismo. Hasta ahora, la población actual que teníamos era la población tras la mutación, a la que llamaré mutaciones. Antes de pasar a una nueva iteración, el reemplazo con elitismo realiza lo siguiente: se busca al mejor elemento de la población inicial y al peor elemento de la población mutaciones, tomando como factor de comparación el valor agregado. Se inicializa como peor y mejor elemento el primero de sus respectivas poblaciones, y se almacena su índice. A continuación, se entra en un bucle en el que se recorren el resto de elementos de ambas poblaciones y va comparando el elemento actual con el mejor/peor ya almacenado. Si se da el caso de que el elemento actual es mejor/peor, se actualiza como mejor/peor elemento el actual y se actualiza el índice también. De ésta forma, cuando finaliza el bucle tenemos el mejor elemento de la población original y su índice, y el peor elemento de la población mutaciones y su índice. Ahora se compara si el valor agregado del mejor elemento de la población original es mejor que el peor de la población mutaciones, sí lo es, el peor elemento de la población mutaciones es reemplazado por el mejor elemento de la población original, sino, la población mutaciones se deja tal cual. De ésta forma estamos cumpliendo llevando a cabo el reemplazo con elitismo, pues le damos la oportunidad al mejor elemento de la población original de seguir en la población. Por último decir, que de cara a una nueva iteración, la población inicial que considerará ahora el algoritmo será la heredada de la población mutaciones. Éste método es el encargado de realizar el reemplazo con elitismo:

```
Function replacement(poblacion_original, mutaciones)
        mejor = poblacion_original[0].agr
        indice_mejor = 0
        peor= mutaciones[0].agr
        indice_peor = 0

        For i to tamaño_poblacion_original
                If poblacion_original[i].agr > mejor
                        mejor = poblacion_original[i].agr
                        indice_mejor = i
                End
                If mutaciones[i].agr < peor
                        peor= mutaciones[i].agr
                        indice_peor = i
                End
        End

        If mejor > peor
                mutaciones[indice_peor] = poblacion_original[indice_mejor]
        End

        Return mutaciones
```

Cuando el algoritmo alcanza las 15.000 evaluaciones de la función objetivo, devuelve la mejor solución obtenida, y eso es gracias a la siguiente función:

```
Function bestElement(poblacion)
        best = poblacion[0]
        For i to tamaño_poblacion
                If poblacion[i].agr > best.agr
                        best = poblacion[i]
                End
        End

        Return best
```

Ésta función básicamente consiste en que dada una población, toma como mejor elemento inicial el primer elemento de la población y posteriormente va comparando si el elemento actual de la población es mejor que el ya prefijado, sí lo es, se actualiza el mejor elemento y así hasta recorrer toda la población.

* ALGORITMO GENÉTICO ESTACIONARIO *

De nuevo, partimos de una **población inicial** generada aleatoriamente. En ése momento ya habremos realizado tantas evaluaciones de la función objetivo como elementos tenga la población inicial. Tras ésto, se entra en un bucle del que no se sale hasta que se completen 15.000 evaluaciones de la función objetivo. Dentro ya del bucle, lo primero que tiene lugar es el proceso de **SELECCIÓN**, en el que se van a elegir mediante la función **binaryTournament** la nueva población que va a estar compuesta por los padres. En éste caso no se seleccionan tantos padres como elementos tiene la población, sino que se seleccionan sólo 2. Una vez seleccionados los padres, nuestra población pasa a tener tamaño 2, únicamente tenemos 2 padres en ella. Ahora, tiene lugar el proceso de **CRUCE**. En éste proceso, independientemente del método de cruce que se haya elegido, BLX o Aritmético, sabemos que solamente va a haber 1 cruce pues nuestra población está compuesta por 2 padres, y por tanto, por sólo una pareja. Por lo que ahora la población estaría formada los 2 hijos que hayan resultado de proceso de cruce. Llegados a éste punto, tiene lugar el proceso de **MUTACIÓN**. Aquí aparece otra diferencia con respecto al AGG, puesto que tenemos una población actual formada por sólo 2 hijos, si mantuviéramos el mismo procedimiento para calcular el número de mutaciones que en AGG, siempre sería 0, por lo tanto, aquí se muta a nivel de cromosoma, así que obtenemos la probabilidad de que un cromosoma mute como la multiplicación de la probabilidad de que mute un gen (0.001 en éste caso) por la longitud del vector de pesos. Una vez que tenemos la probabilidad de que un cromosoma mute, se genera un número aleatorio por cada hijo, entre 0 y 1 y si ese número es menor que la probabilidad comentada anteriormente, ese hijo muta. Elegir un cromosoma aleatoriamente en éste algoritmo no tiene sentido, puesto que tenemos sólo 2, los 2 hijos, asi que lo que se elige aleatoriamente es el gen que muta. El resto de procedimiento es el mismo que en el algoritmo AGG. Nuestra población actual por tanto sigue siendo 2 hijos, que quizá han mutado y quizás no. Llega ahora el turno del proceso de **REEMPLAZAMIENTO**. Antes de llevar a cabo el reemplazo, a la población original, por ejemplo en nuestro caso de tamaño 30, le añado los elementos de la población actual, que son los 2 hijos con o sin mutaciones. Por lo tanto, me queda una población de 32 elementos. En ese momento, entra en acción la función descrita en pseudocódigo más abajo cuyo funcionamiento es el siguiente: Genera una copia de la población formada por 32 elementos. Entra en un bucle que por cada elemento de la población, inserta en una lista auxiliar cada valor de Agregado de la población acompañado de su índice. Tras ésto, se ordena dicha lista de menor a peor en función del valor Agregado, es decir los primeros elementos de la lista ordenada serán los PEORES, y los últimos los MEJORES. Por último, eliminamos de la población pasada por argumento los 2 peores elementos que vienen a ser el primero y el segundo obtenidos con la lista ordenada, y se devuelve la población. De ésta forma, lo que hemos hecho es que los hijos generado en el proceso de evolución compitan con la población inicial, de forma que si son mejores que los peores elementos de la población inicial, pasarán a sustituirlos en ella. A la población inicial pueden entrar los 2 hijos, 1 hijo

o ninguno, es decir, el reemplazo no está asegurado y sólo se llevará a cabo si son mejores. Función que realiza el reemplazo:

```
Function worstElements(población)
        poblacion_c = genero una copia de la población pasada como argumento
        aux = []

        For i to tamaño_poblacion
                aux.append((i, poblacion[i].agr))
        End

        aux = sort(aux, order='agr')

        poblacion.remove(aux[0])
        poblacion.remove(aux[1])

        Return poblacion
```

Finalmente, cuando el algoritmo alcanza las 15.000 evaluaciones de la función objetivo, devuelve la mejor solución obtenida, gracias a la función comentada en el algoritmo anterior: **bestElement.**

* ALGORITMOS MEMÉTICOS *

Los algoritmos genéticos están compuestos de un algoritmo AGG más la integración de la búsqueda local cada X generaciones. Por lo tanto, para dichos algoritmos he reutilizado el AGG descrito anteriormente y me he decantado por usar el cruce BLX pues es el que me ha rendido mejor en la mayoría de casos. He modificado levemente el algoritmo de **Búsqueda Local** para que pueda ser llamado correctamente por el algoritmo memético, de forma que ya no es necesario evaluar la función objetivo antes de entrar al bucle while, puesto que ya está evaluada la 'w' inicial que recibe ahora la BL del algoritmo memético. También se ha reducido su condición de salida y se devuelve a parte de la solución obtenida, el número de evaluaciones, para que esas evaluaciones contribuyan a las del AM. Su pseudocódigo es el siguiente:

```
Function LS(x_train, y_train, initial_solution)

w = initial_solution
n = 1
ev = 1
last_agr = initial_solution.agr

While n < 2 * tamaño_w
        For i to tamaño_w
                w_copy  = copy(w)
                z = np.random.normal(0,0.3,1)
                w[i] += z

                Normalizo el valor

                neigh = KNeighborsClassifier(n_neighbors=1)
                w_c = copy(w)
                w_c[w_c < 0.2] = 0.0
                neigh.fit(x_train*w_c, y_train)
                vecinos = neigh.kneighbors(x_train*w_c,n_neighbors=2,return_distance=False)
                prediccion = y_train[vecinos[:, 1]]
                new_clas, new_red, new_agr = evaluation_function(predicción,y_train,w,0.5)
                ev += 1
                If new_agr > last_agr
                        last_agr = new_agr
                        break
                End
                Else
                        w = w_copy
                        n += 1
                End
        End
End

Return w, ev
```

El **AM-(10, 1.0)** funciona igual que el AGG a excepción de cuando se alcanza un número de generaciones divisible exacto de 10. Se considera una generación completada cuando se realiza todo el esquema de evolución incluído el reemplazo. Por tanto, cuando el número de generaciones es divisible exacto de 10, por cada elemento de la población actual, se llama a Búsqueda Local, pasándole como argumento inicial la solución de ese elemento, es decir, el vector de pesos 'w' actual y ese mismo vector de pesos es sustituido por la solución aportada por la Búsqueda Local. De esa forma obtenemos una población totalmente nueva formada por las soluciones aportada por la Búsqueda Local. Pseudocódigo:

```
If generation % 10== 0
        For i to tamaño_población
                w, eva = LS(x_train, y_train, poblacion[i])
                poblacion[i] = w
        End
End
```

El **AM-(10, 0.1)** funciona nuevamente igual que el AGG a excepción de cuando se alcanza un número de generaciones divisible exacto de 10. En ese momento, aplicamos la Búsqueda Local sobre 0.1 * tamaño_población elementos, pero como en nuestro caso el tamaño de la población es 10, indicamos directamente que la BL se aplica sobre un elemento. El elemento que va a ser sustituído por la solución aportada por la Búsqueda Local se elige de forma aleatoria entre todos los elementos de la población. Por tanto, la población se mantiene intacta excepto un elemento que es sustituido por la solución de la BL. Pseudocódigo:

```
If generation % 10 == 0
        cromosoma = número aleatorio entre 0 y tamaño_población
        w, eva = LS(x_train, y_train, poblacion[cromosoma])
        poblacion[cromosoma] = w
End
```

Por último, el **AM-(10, 0.1mej)**, el cual funciona exáctamente igual que el AM-(10, 0.1) con una única diferencia, y esa es que el elemento que va a ser sustituído por la solución de la Búsqueda Local no es elegido aleatoriamente, sino que se aplica sobre el mejor de la población actual. El mejor de la población lo obtenemos con el método **bestElementIndex**, que hace lo mismo que el método bestElement, ya comentado en ésta memoria, con la particularidad de que en vez de devolver el mejor elemento como tal, devuelve su índice. Pseudocódigo de AM-(10, 0.1mej):

```
If generation % 10 == 0
        best = bestElementIndex(poblacion)
        w, eva = LS(x_train, y_train, poblacion[best])
        poblacion[best] = w
End
```

* ENFRIAMIENTO SIMULADO (ES) *

- Cálculo de la Temperatura Inicial

La Temperatura Inicial va a ser calculada mediante la fórmula:

$$T_0 = \frac{\mu * C(S_0)}{-ln(\phi)}$$

En nuestro caso, consideramos tanto μ como ϕ con valor = 0.3 y el Coste de la solución inicial lo consideramos como el valor agregado de la primera solución que generamos aleatoriamente, por lo tanto la fórmula de la Temperatura Inicial quedaría así:

$$T_0 = \frac{0.3 * agregado(soluciónInicial)}{-ln(0.3)}$$

- Esquema de Enfriamiento

En cuánto al Esquema de Enfriamiento, utilizamos el esquema de Cauchy modificado que viene delimitado por las expresiones:

$$T_{k+1} = \frac{T_k}{1 + \beta * T_k} \quad \text{y} \quad \beta = \frac{T_0 - T_f}{M * T_0 * T_f}$$

siendo T_f la Temperatura Final, fijada a 10^{-3} y M el número de enfriamientos a realizar. Pseudocódigo:

```
Function ES(datos, etiquetas):

        tasa_clas = []
        tasa_red = []
        times = []
        agregacion = []

        skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
        for train_index, test_index in skf.split(datos, etiquetas):
                start = time.time()
                x_train, y_train = datos[train_index], etiquetas[train_index]
                x_test, y_test = datos[test_index], etiquetas[test_index]

                w = np.random.uniform[0,1,num_caracteristicas]

                neigh = KNeighborsClassifier(n_neighbors=1)

                w_c = np.copy(w)
                w_c[w_c < 0.2] = 0.0

                neigh.fit(x_train*w_c, y_train)
                vecinos = neigh.kneighbors(x_train*w_c,n_neighbors=2,return_distance=False)
```

```
prediccion = y_train[vecinos[:, 1]]
last_clas, last_red, last_agr = evaluation_function(predicción,y_train,w,0.5)

best_agr = last_agr
best_w = w
Tf = 1e-3
T0 = (0.3 * best_agr) / (-np.log(0.3))
max_neighbors = 10 * len(w)
max_successes = 0.1 * max_neighbors
max_evaluations = 15000
M = max_evaluations / max_neighbors
B = (T0 - Tf) / (M * T0 * Tf)
T = T0
ev = 0
n_successes = 1
K = 1

while ev < max_evaluations and n_successes > 0 and T > Tf
        n_successes = 0
        int_ev = 0
        while int_ev < max_neighbors and n_successes < max_successes
                int_ev += 1
                z = np.random.normal[0,0.3]
                caract = número_aleatorio(tamaño_w)
                w_mut = np.copy(w)
                w_mut[caract] += z

                if w_mut[caract] < 0.0
                        w_mut[caract] = 0.0
                else if w_mut[caract] > 1.0
                        w_mut[caract] = 1.0

                neigh = KNeighborsClassifier(n_neighbors=1)
                w_c = np.copy(w_mut)
                w_c[w_c < 0.2] = 0.0
                neigh.fit(x_train*w_c,y_train)
                idx = neigh.kneighbors(x_test*w_c,n_neighbors=2,return_distance=False)

                prediccion= y_train[idx[:,1]]
                new_clas, new_red, new_agr =
                evaluation_function(predicción,y_train,w_mut,0.5)

                difference = new_agr - last_agr

                if difference > 0 or np.random.uniform() <= np.exp(difference / (K*T))
                        w = w_mut
                        last_agr = new_agr
                        n_successes += 1

                        if last_agr > best_agr
                                best_agr = last_agr
                                best_w = w
        ev += int_ev
```

```
                    T = T / (1 + B * T)

            neigh = KNeighborsClassifier(n_neighbors=1)
            neigh.fit(x_train*best_w,y_train)
            prediccion = neigh.predict(x_test*best_w)
            new_clas, new_red, new_agr = evaluation_function(y_test,predicción,best_w,0.5)

            end = time.time()
            final_clas = new_clas
            final_red = new_red
            final_agr = new_agr

            tasa_clas.append(final_clas)
            tasa_red.append(final_red)
            agregacion.append(final_agr)
            times.append(end-start)

        Return tasa_clas, tasa_red, agregacion, times
```

* BÚSQUEDA LOCAL REITERADA (ILS) *

- Operador de Mutación

Puesto que se va a cambiar el valor del peso en un 10 % de las características, **t** contiene el número de exacto de mutaciones que se van a realizar, es decir, cuántas características van a cambiar. Una vez se tiene **t**, se seleccionan tantos índices de características como indique **t** de forma aleatoria y garantizando que no se repitan características. Por lo demás, funciona de la misma forma que la mutación en algoritmos anteriores a excepción de que la distribución normal ahora considera $\sigma = 0.4$. Por último, se normaliza el valor de las características que hayan sido mutadas. Pseudocódigo:

```
t = 0.1 * tamaño_w
w_mut = np.copy(w)
n = np.arange(tamaño_w)
n = set(n)
index = random.sample(n,t)

z = np.random.normal(0,0.4,1)
w_mut[index] += z

for i in index:
        if w_mut[i] < 0.0:
            w_mut[i] = 0.0
        elif w_mut[i] > 1.0:
            w_mut[i] = 1.0
```

```
Function ILS(datos, etiquetas):

    tasa_clas = []
    tasa_red = []
    times = []
    agregacion = []

    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
    for train_index, test_index in skf.split(datos, etiquetas):
        start = time.time()
        x_train, y_train = datos[train_index], etiquetas[train_index]
        x_test, y_test = datos[test_index], etiquetas[test_index]

        w = np.random.uniform[0,1,num_caracteristicas]

        w = BUSQUEDA_LOCAL_ILS(x_train,y_train,w)

        neigh = KNeighborsClassifier(n_neighbors=1)

        w_c = np.copy(w)
        w_c[w_c < 0.2] = 0.0

        neigh.fit(x_train*w_c, y_train)
        vecinos = neigh.kneighbors(x_train*w_c,n_neighbors=2,return_distance=False)
        prediccion = y_train[vecinos[:, 1]]
        last_clas, last_red, last_agr = evaluation_function(predicción,y_train,w,0.5)

        max_iter = 15
        it = 1
        t = int(0.1 * len(w))
        best_agr = last_agr

        while it < max_iter
            it += 1

            w_mut = np.copy(w)
            n = np.arange(tamaño_w)
            n = set(n)
            index = random.sample(n,t)
            z = np.random.normal(0,0.4,1)
            w_mut[index] += z

            for i in index:
               if w_mut[i] < 0.0
                    w_mut[i] = 0.0
               else if w_mut[i] > 1.0
                    w_mut[i] = 1.0

            w_mut = BUSQUEDA_LOCAL_ILS(x_train,y_train,w_mut)

            neigh = KNeighborsClassifier(n_neighbors=1)
            w_c = np.copy(w_mut)
            w_c[w_c < 0.2] = 0.0
```

```
                    neigh.fit(x_train*w_c,y_train)
                    idx = neigh.kneighbors(x_test*w_c,n_neighbors=2,return_distance=False)

                    prediccion= y_train[idx[:,1]]

                    new_clas, new_red, new_agr =
            evaluation_function(predicción,y_train,w_mut,0.5)

                    if new_agr > best_agr
                            best_agr = new_agr
                            w = w_mut

            neigh = KNeighborsClassifier(n_neighbors=1)
            neigh.fit(x_train*w,y_train)
            prediccion = neigh.predict(x_test*w)
            new_clas, new_red, new_agr = evaluation_function(y_test,predicción,w,0.5)

            end = time.time()
            final_clas = new_clas
            final_red = new_red
            final_agr = new_agr

            tasa_clas.append(final_clas)
            tasa_red.append(final_red)
            agregacion.append(final_agr)
            times.append(end-start)

    Return tasa_clas, tasa_red, agregacion, times
```

* EVOLUCIÓN DIFERENCIAL (DE) *

- Común a ambos modelos

- La Recombinación se realiza gen a gen, y en éste caso nos encontramos ante una Recombinación Binomial, de forma que si un número aleatorio $\varepsilon[0,1]$ es menor o igual que la probabilidad de Cruce, 0.5 en éste caso, se introduce el gen obtenido con la fórmula anterior. En caso contrario, se mantiene el gen del individuo actual.

- El Reemplazamiento en ambos casos es **one-to-one**, lo que significa que una vez se tiene el individuo fruto de la mutación, es decir el hijo, se compara directamente con su padre, quedándose en la población aquel con mayor valor agregado.

```
if offspring.agr > population[i].agr
        population[i] = offspring
```

El primer modelo usado es el DE/Rand/1. En él, el nuevo individuo obtenido fruto de la mutación se obtiene en base a la siguiente fórmula:

$$V_{i,G} = X_{r1,G} + F * (X_{r2,G} - X_{r3,G})$$

siendo r1,r2 y r3 los índices de cada individuo en cada Generación, escogidos aleatoriamente de forma mutuamente excluyente. Dentro de la función **rand_1** realizo tanto el proceso de Mutación como el Recombinación. Pseudocódigo:

```
Function rand_1(population, index, CR,F,x_train,y_train):

        offspring = []

        r1 = np.random.randint(tamaño_poblacion)
        While r1 == index
                r1 = np.random.randint(tamaño_poblacion)

        r2 = np.random.randint(tamaño_poblacion)
        While r2 == index or r2 == r1
                r2 = np.random.randint(tamaño_poblacion)

        r3 = np.random.randint(tamaño_poblacion)
        While r3 == index or r3 == r1 or r3 == r2
                r3 = np.random.randint(tamaño_poblacion)

        For k in range(numero_caracteristicas):
                if np.random.uniform(0,1,1) <= CR
                        offspring.append(population[r1].w[k] + F * (population[r2].w[k] - population[r3].w[k]))
                else
                        offspring.append(population[index].w[k])

        offspring = np.array(offspring)

        offspring[offspring > 1.0] = 1.0
        offspring[offspring < 0.0] = 0.0

        offspring = element(offspring,x_train,y_train)

        Return offspring
```

El primer modelo usado es el DE/current-to-best/1. En él, el nuevo individuo obtenido fruto de la mutación se obtiene en base a la siguiente fórmula:

$$V_{i,G} = X_{i,G} + F * (X_{best,G} - X_{i,G}) + F * (X_{r1,G} - X_{r2,G})$$

siendo r1 y r2 los índices de cada individuo en cada Generación, escogidos aleatoriamente de forma mutuamente excluyente. Xbest en éste modelo hace referencia al mejor individuo de la Generación, hasta el momento. Dentro de la

función **current_to_best_1** realizo tanto el proceso de Mutación como el Recombinación. Pseudocódigo:

```
Function current_to_best_1(population, index, CR,F,x_train,y_train):

        offspring = []

        r1 = np.random.randint(tamaño_poblacion)
        While r1 == index
                r1 = np.random.randint(tamaño_poblacion)

        r2 = np.random.randint(tamaño_poblacion)
        While r2 == index or r2 == r1
                r2 = np.random.randint(tamaño_poblacion)

        best = bestElement(population)

        For k in range(numero_caracteristicas):
                if np.random.uniform(0,1,1) <= CR
                        offspring.append(population[index].w[k] + F * (best.w[k] - population[index].w[k])
                        + F * (population[r1].w[k] - population[r2].w[k]))
                else
                        offspring.append(population[index].w[k])

        offspring = np.array(offspring)

        offspring[offspring > 1.0] = 1.0
        offspring[offspring < 0.0] = 0.0

        offspring = element(offspring,x_train,y_train)

        Return offspring
```

```
Function ED(datos, etiquetas):

        tasa_clas = []
        tasa_red = []
        times = []
        agregacion = []

        skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
        for train_index, test_index in skf.split(datos, etiquetas):
                start = time.time()
                x_train, y_train = datos[train_index], etiquetas[train_index]
                x_test, y_test = datos[test_index], etiquetas[test_index]

                population = []
                for i in range(50):
                        w = np.random.uniform(0,1,x.shape[1])
                        ele = element(w,x_train,y_train)
                        population.append(ele)

                ev = 50
                CR = 0.5
                F = 0.5
                while ev < 15000
                        for i in range(tamaño_poblacion)
                                if version == 'RAND_1'
                                        offspring = rand_1(population,i,CR,F,x_train,y_train)
                                        ev += 1
                                elif version == 'CURRENT_TO_BEST_1'
                                        offspring = current_to_best_1(population,i,CR,F,x_train,y_train)
                                        ev += 1
                                if offspring.agr > population[i].agr
                                        population[i] = offspring

                end = time.time()
                w = bestElement(population)

                neigh = KNeighborsClassifier(n_neighbors=1)
                neigh.fit(x_train*w.w,y_train)
                prediccion = neigh.predict(x_test*w.w)
                new_clas, new_red, new_agr = evaluation_function(y_test,predicción,w.w,0.5)

                final_clas = new_clas
                final_red = new_red
                final_agr = new_agr

                tasa_clas.append(final_clas)
                tasa_red.append(final_red)
                agregacion.append(final_agr)
                times.append(end-start)

        Return tasa_clas, tasa_red, agregacion, times
```

4. DESCRIPCIÓN EN PSEUDOCÓDIGO DE LOS ALGORITMOS DE COMPARACIÓN

* ALGORITMO RELIEF *

```
Function relief (datos, etiquetas):

        tasa_clas = []
        tasa_red = []
        agregacion = []
        times = []

        skf = StratifiedKFold(n_splits=5,shuffle=True, random_state=1)

        for train_index, test_index until skf.split(datos, etiquetas)
                start = time.time()

                x_train, y_train = datos[train_index], etiquetas[train_index]
                x_test, y_test = datos[test_index], etiquetas[test_index]

                w = np.zeros(datos.shape[1])

                for i=0 until nºdatos_train
                        enemigo, amigo = closest_examples(x_train,i,y_train)

                        w = w + np.abs(x_train[i] - enemigo) - np.abs(x_train[i] - amigo)

                wm = np.max(w)
                for i=0 until nºcaracteristicas
                        if w[i] < 0.0
                                w[i] = 0.0
                        else
                                w[i] = w[i] / wm

                x_train = x_train * w
                 x_test = x_test * w

                neigh = KNeighborsClassifier(n_neighbors=1)
                neigh.fit(x_train,y_train)
                prediccion = neigh.predict(x_test)
                clas, red, agr = evaluation_function(y_test,prediccion,w,0.5)

                end = time.time()

                tasa_clas.append(clas)
                tasa_red.append(red)
                agregacion.append(agr)
                times.append(end-start)

        Return tasa_clas, tasa_red, agregacion, times
```

El algoritmo greedy Relief parte de un vector de pesos inicial, el cual trata de mejorar para obtener una solución mejor. Su funcionamiento para mejorarlo es el siguiente: Por cada valor en los datos de entrenamiento, obtiene su amigo y enemigo más cercano. Una vez obtenidos ambos, modifica el vector de pesos de forma que si la distancia entre el enemigo más cercano es mayor que la distancia con respecto al amigo más cercano, se incrementa el valor de los pesos, mientras que cuando ocurre al revés, se decrementa. Finalmente, el algoritmo devuelve los valores tasa_clas, tasa_red, agregacion y times obtenido en cada iteración considerando la mejor solución posible, es decir, el vector de pesos que maximiza la función de evaluación. Las funciones que no explico en éste apartado es porque han sido explicadas en el apartado **2.)**, ya que son comunes a todos los algoritmos. **Funciones a explicar:**
"**np.zeros**" es una función que inicializa el vector de pesos con el tamaño que se le indique a valor 0. "**np.abs**" devuelve el resultado de sus argumentos en valor absoluto. "**np.max**" devuelve el valor máximo hallado en el vector de pesos.

*** ALGORITMO 1-NN ***

```
Function k-nn (datos, etiquetas):

        tasa_clas = []
        tasa_red = []
        times = []
        agregacion = []

        skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)

        for train_index, test_index until skf.split(x,y)

                start = time.time()
                x_train, y_train = datos[train_index], etiquetas[train_index]
                x_test, y_test = datos[test_index], etiquetas[test_index]

                w = np.ones(x.shape[1])

                neigh = KNeighborsClassifier(n_neighbors=1)
                neigh.fit(x_train,y_train)
                prediccion = neigh.predict(x_test)
                clas, red, agr = evaluation_function(y_test,prediccion,w,0.5)
                end = time.time()

        tasa_clas.append(clas)
        tasa_red.append(red)
        agregacion.append(agr)
        times.append(end-start)

        Return tasa_clas, tasa_red, agregacion, times
```

Las funciones que usa éste algoritmo ya han sido explicadas anteriormente. En éste caso, todos los valores del vector de pesos se inicializan a 1 para que todos los elementos tengan la misma importancia.

Funciones a explicar:

"**np.ones**" es la función que inicializa el vector de pesos a valor uno.

5. BREVE EXPLICACIÓN DEL PROCEDIMIENTO CONSIDERADO PARA DESARROLLAR LA PRÁCTICA

- Para desarrollar la práctica he usado el lenguaje de programación Python 3.7.1, siendo necesario tenerlo instalado.

- El desarrollo lo he llevado a cabo en el IDE Spyder.

- Para ejecutar la práctica es necesario tener instalados los siguientes paquetes:

- Scikit-learn
- numpy (para realizar todo tipo de operaciones)
- pandas (para leer los ficheros en formato CSV)
- StratifiedKFold (para realizar las 5 particiones, train, test...)
- KneighborsClassifier (para obtener vecinos, y métodos derivados...)
- time (para contabilizar el inicio y fin de ejecución)
- MinMaxScaler (para normalizar los datos extraídos de los ficheros CSV)

- Los ficheros de datos están dentro de un directorio llamado 'datos'. En el propio fichero ya he dejado escrita la ruta de lectura, por lo que sería necesario que el fichero y la carpeta 'datos' estuvieran en el mismo nivel antes de ejecutar.

- Personalmente, recomiendo instalar Anaconda, puesto que ya trae por defecto la mayoría de los paquetes que he usado.

- Práctica realizada en el SO: Windows 10.

- Para ejecutar el fichero, bastaría con correrlo si se usa un IDE, o en CMD(también en otras terminales) teclear: '**python P3.py'**.

6. EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

○ CASOS DEL PROBLEMA Y VALORES DE LOS PARÁMETROS EMPLEADOS EN LAS EJECUCIONES DE CADA ALGORITMO

- Los datasets han sido convertidos a CSV previamente a su lectura. En la lectura de ellos, almaceno en arrays de Numpy los datos por un lado y las etiquetas por otro. Una vez leídos los datos, son normalizados. Posteriormente, se van llamando a los diferentes algoritmos y se van mostrando sus resultados.

- En cuanto a la pseudoaleatoriedad, la semilla está fijada al principio del fichero a 1. "**np.random.seed(1)**"

- En los algoritmos, a la hora de dividir el conjunto en 5 particiones, he utilizado 'shuffle' en los datos. El hecho de haber considerado 'shuffle' conlleva que los datos se barajan con cierto índice de aleatoriedad, por lo que el valor de "**random_state**" ha sido puesto a 1 también durante la inicialización para que concuerde con el valor de la semilla.

- Para obtener el valor Agregado, es decir la función objetivo, he usado un valor de $\alpha = 0.5$.

- En el algoritmo de Búsqueda Local y AGs y AMs, el valor Z que provoca la mutación del vector de pesos ha sido generado mediante una distribución normal de media 0 y desviación típica = 0.3 para todas las ejecuciones.

- Para el algoritmo de Enfriamiento Simulado (ES), se consideran $\varphi = \mu = 0.3$ y la Temperatura Final se fija a $T_f = 10^{-3}$.

- Para el algoritmo Búsqueda Local Reiterada (ILS), se mutarán el 10 % de las características. En ésta ocasión, la mutación se genera siguiendo una distribución normal de media 0 y desviación típica = 0.4.

- Para las diferentes versiones de Evolución Diferencial, se considera la Probabilidad de Cruce = 0.5 al igual que F.

○ RESULTADOS OBTENIDOS

Tabla 5.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	72,8810	0,0000	36,4406	0.001	83,0985	0,0000	41,5492	0,0029	92,7272	0,0000	46,3636	0,0029
Partición 2	71,9290	0,0000	35,9649	0.001	87,1428	0,0000	43,5714	0,0019	93,6363	0,0000	46,8181	0,0029
Partición 3	70,1750	0,0000	35,0877	0.001	85,7142	0,0000	42,8571	0,0019	91,8181	0,0000	45,9090	0,0019
Partición 4	71,9290	0,0000	35,9649	0.001	91,4285	0,0000	45,7142	0,0020	90,9090	0,0000	45,4545	0,0019
Partición 5	80,7010	0,0000	40,3508	0.001	85,7142	0,0000	42,8571	0,0019	95,4545	0,0000	47,7272	0,0029
Media	73,5230	0,0000	36,7618	0.001	86,6197	0,0000	43,3098	0,0021	92,9090	0,0000	46,4545	0,0025

Tabla 5.2: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	76,2711	35,4838	55,8775	0,2573	85,9154	2,9411	44,4283	0,3739	96,3636	2,5000	49,4318	0,9175
Partición 2	68,4210	40,3225	54,3718	0,2503	87,1428	2,9411	45,0420	0,3700	93,6363	7,5000	50,5681	0,9025
Partición 3	71,9298	40,3225	56,1262	0,2483	91,4285	2,9411	47,1848	0,3700	93,6363	15,0000	54,3181	0,9005
Partición 4	68,4210	27,4193	47,9202	0,2413	91,4285	2,9411	47,1848	0,3630	89,0909	20,0000	54,5454	0,8866
Partición 5	70,1754	45,1612	57,6683	0,2503	87,1428	2,9411	45,0420	0,3670	97,2727	5,0000	51,1363	0,8906
Media	71,0437	37,7419	54,3928	0,2495	88,6116	2,9411	45,7764	0,3688	94,0000	10,0000	52,0000	0,8995

Tabla 5.11: Resultados obtenidos por el algoritmo ES en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	67,7966	69,3548	68,5757	15,3232	88,7324	91,1765	89,9544	7,5263	94,5455	85,0000	89,7727	13,5819
Partición 2	71,9298	69,3548	70,6423	17,8954	87,1429	88,2353	87,6891	7,5118	89,0909	85,0000	87,0455	11,9989
Partición 3	73,6842	87,0968	80,3905	10,5747	88,5714	85,2941	86,9328	8,5915	93,6364	82,5000	88,0682	14,3463
Partición 4	64,9123	82,2581	73,5852	10,8501	85,7143	88,2353	86,9748	7,0726	90,9091	85,0000	87,9545	14,4076
Partición 5	78,9474	70,9677	74,9576	12,7170	87,1429	91,1765	89,1597	6,9115	93,6364	77,5000	85,5682	13,0866
Media	71,4541	75,8065	73,6303	13,4721	87,4608	88,8235	88,1421	7,5227	92,3636	83,0000	87,6818	13,4843

Tabla 5.12: Resultados obtenidos por el algoritmo ILS en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	72,8814	82,2581	77,5697	57,3836	88,7324	91,1765	89,9544	34,3113	88,1818	87,5000	87,8409	68,7422
Partición 2	64,9123	79,0323	71,9723	66,3960	82,8571	88,2353	85,5462	35,8860	93,6364	85,0000	89,3182	71,0795
Partición 3	71,9298	88,7097	80,3198	53,5897	97,1429	91,1765	94,1597	40,7924	93,6364	85,0000	89,3182	72,3430
Partición 4	73,6842	90,3226	82,0034	57,4484	94,2857	88,2353	91,2605	38,8691	86,3636	87,5000	86,9318	69,1477
Partición 5	80,7018	82,2581	81,4799	58,0613	88,5714	91,1765	89,8739	34,9236	95,4545	87,5000	91,4773	68,2027
Media	72,8219	84,5161	78,6690	58,5758	90,3179	90,0000	90,1590	36,9565	91,4545	86,5000	88,9773	69,9030

Tabla 5.13: Resultados obtenidos por el algoritmo DE/rand/1 en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	77,9661	91,9355	84,9508	74,7201	88,7324	94,1176	91,4250	39,0367	92,7273	85,0000	88,8636	80,8144
Partición 2	61,4035	91,9355	76,6695	62,7482	85,7143	91,1765	88,4454	48,6953	93,6364	87,5000	90,5682	77,8037
Partición 3	75,4386	93,5484	84,4935	63,7683	88,5714	91,1765	89,8739	44,4313	93,6364	87,5000	90,5682	76,4611
Partición 4	64,9123	93,5484	79,2303	72,8426	88,5714	91,1765	89,8739	42,8902	91,8182	87,5000	89,6591	81,4337
Partición 5	75,4386	93,5484	84,4935	72,9576	88,5714	94,1176	91,3445	43,9054	92,7273	87,5000	90,1136	89,6138
Media	71,0318	92,9032	81,9675	69,4073	88,0322	92,3529	90,1926	43,7918	92,9091	87,0000	89,9545	81,2253

Tabla 5.14: Resultados obtenidos por el algoritmo DE/current-to-best/1 en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	74,5763	72,5806	73,5785	56,3414	91,5493	91,1765	91,3629	36,8625	90,0000	72,5000	81,2500	81,2323
Partición 2	64,9123	79,0323	71,9723	63,0508	90,0000	82,3529	86,1765	44,6546	90,9091	82,5000	86,7045	70,7678
Partición 3	66,6667	72,5806	69,6237	62,6451	84,2857	82,3529	83,3193	43,0055	91,8182	80,0000	85,9091	74,1209
Partición 4	71,9298	67,7419	69,8359	66,3835	84,2857	88,2353	86,2605	45,6101	85,4545	80,0000	82,7273	71,2640
Partición 5	77,1930	79,0323	78,1126	59,5397	91,4286	79,4118	85,4202	47,8611	95,4545	82,5000	88,9773	78,0973
Media	71,0556	74,1935	72,6246	61,5921	88,3099	84,7059	86,5079	43,5988	90,7273	79,5000	85,1136	75,0965

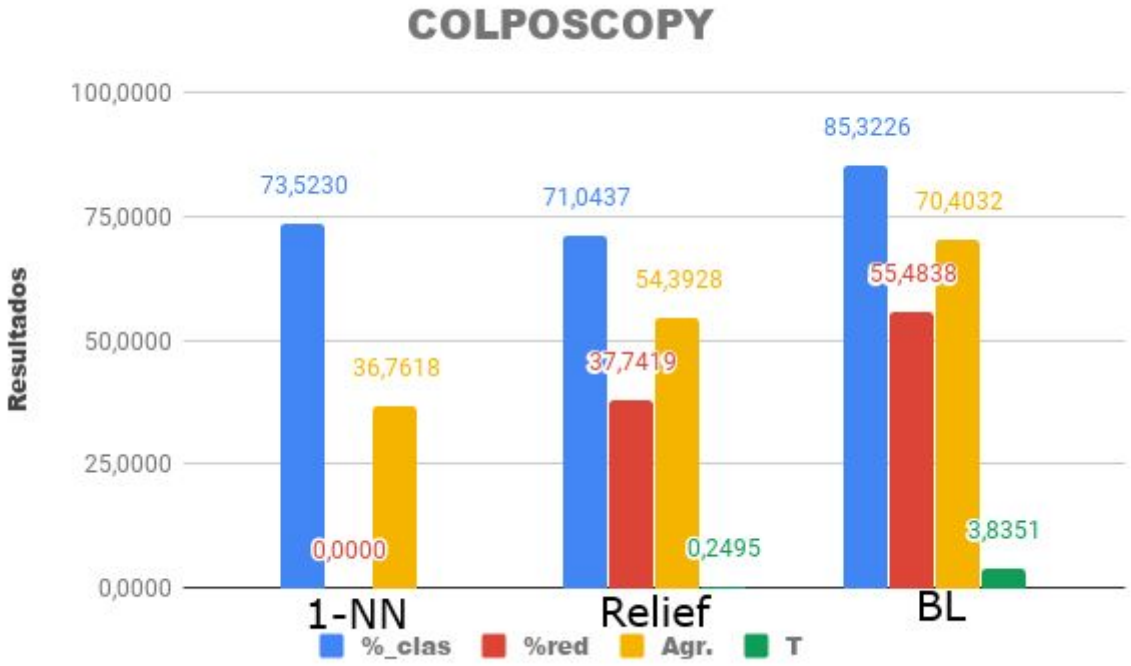
Tabla 5.15: Resultados globales en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
1-NN	73,5230	0,0000	36,7618	0.001	86,6197	0,0000	43,3098	0,0021	92,9090	0,0000	46,4545	0,0025
RELIEF	71,0437	37,7419	54,3928	0,2495	88,6116	2,9411	45,7764	0,3688	94,0000	10,0000	52,0000	0,8995
ES	71,4541	75,8065	73,6303	13,4721	87,4608	88,8235	88,1421	7,5227	92,3636	83,0000	87,6818	13,4843
ILS	72,8219	84,5161	78,6690	58,5758	90,3179	90,0000	90,1590	36,9565	91,4545	86,5000	88,9773	69,9030
DE/rand/1	71,0318	92,9032	81,9675	69,4073	88,0322	92,3529	90,1926	43,7918	92,9091	87,0000	89,9545	81,2253
DE/current-to-best/1	71,0556	74,1935	72,6246	61,5921	88,3099	84,7059	86,5079	43,5988	90,7273	79,5000	85,1136	75,0965

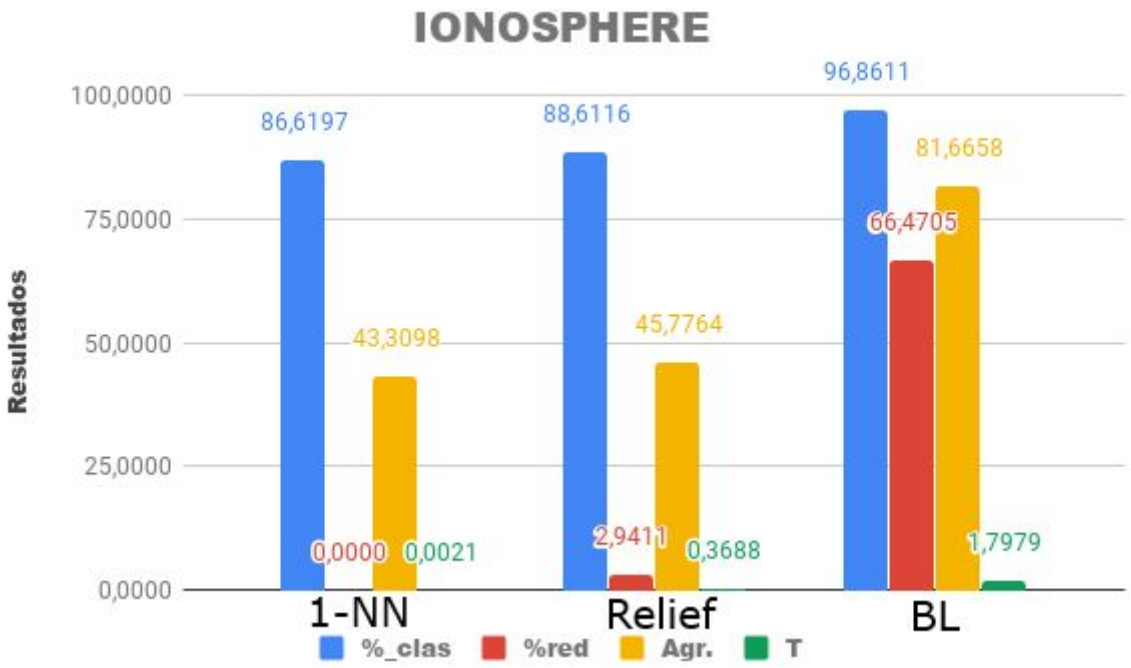
- **ANÁLISIS DE RESULTADOS**

PRÁCTICA 1

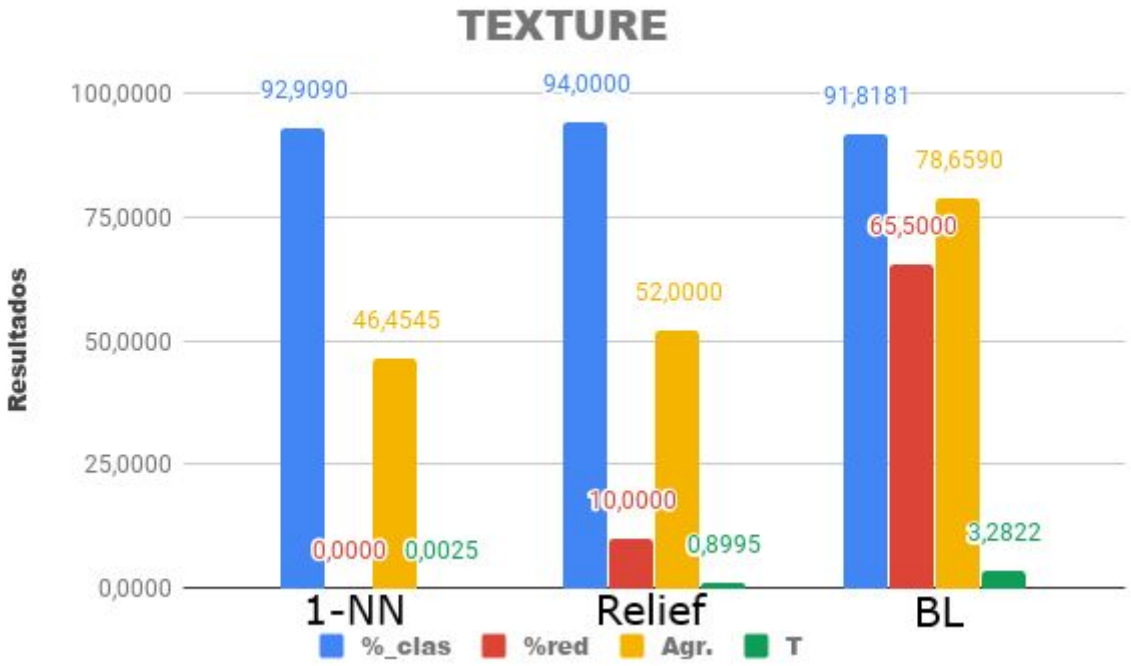
Para llevar a cabo el análisis de forma más ilustrativa, me voy a apoyar en 3 gráficas que he creado, en las que muestro una comparativa de los resultados obtenidos por cada algoritmo de forma individual en cada dataset.



En éste primer dataset, vemos como se evidencia que el algoritmo 1-NN es el más simple de todos y el que peores resultados obtiene. Si bien tiene una tasa de acierto aceptable, no consigue reducir a la hora de elegir al subconjunto de pesos, cosa que sí consigue Relief. Como vemos también, el valor de la función objetivo es mayor con el Relief, denotando así que obtiene mejores resultados que 1-NN. Si hablamos del algoritmo de Búsqueda Local, vemos que la tasa de aciertos se ve incrementada con respecto a los algoritmos de comparación, es decir que clasifica de forma más precisa ante unos datos de test. Por otra parte, consigue una reducción mayor y es el algoritmo que mejor maximiza la función objetivo. La conclusión general que se saca de éste gráfico es que la Búsqueda Local es el mejor algoritmo de los 3, con la única pega de que a cambio de obtener mejores resultados, el tiempo de ejecución es mayor.



En éste dataset de nuevo, la Búsqueda Local es muy superior con respecto a su comparativa. Obtiene una tasa de acierto rozando el 100%, lo que da una credibilidad a éste algoritmo a un mayor pues el fallo de predicción no llega al 5%. En cuanto a la reducción, vemos que al Relief le cuesta particularmente con estos datos, por lo que obtiene un valor de función objetivo similar al 1-NN en el que no es posible la reducción.



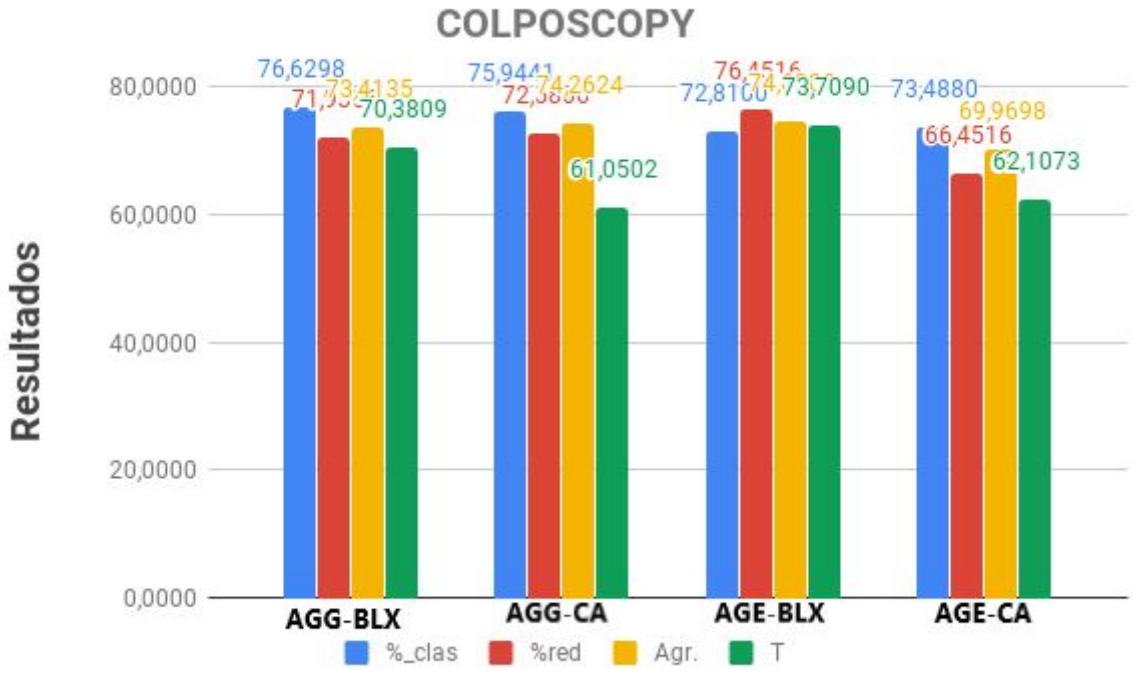
Para el último dataset, vemos que se repite todo lo comentado en los anteriores, la Búsqueda Local obtiene una mayor tasa de reducción, maximiza la función objetivo... pero en éste caso hay una particularidad, y es que para obtener esos valores de reducción y función objetivo, estamos sacrificando un pequeño porcentaje de tasa de acierto, dando lugar a que tanto Relief como 1-NN posean una tasa de acierto mayor. ¿Sería peor entonces la Búsqueda Local en éste caso? La respuesta es NO. Si bien es cierto que su tasa de acierto es inferior, la diferencia es mínima y no por ello lo hace peor, ya que estamos reduciendo los datos un 55% más que en el Relief y obteniendo un resultado de función objetivo que casi duplica al 1-NN y supera holgadamente al Relief.

Conclusión General

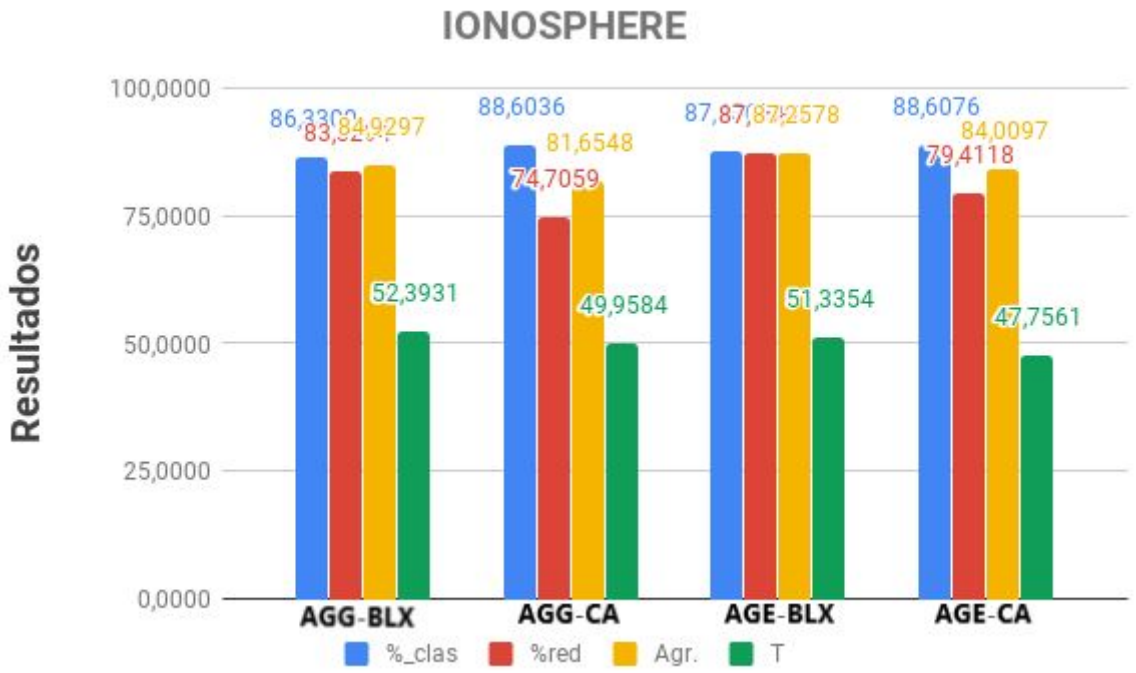
A la vista de los resultados, el algoritmo de Búsqueda Local es muy superior tanto al 1-NN como al Relief. Afronta con mayor facilidad la diversidad del dataset y siempre reduce considerablemente más que el Relief, manteniendo un nivel de tasa de aciertos más que aceptable que hace que sea el algoritmo que más maximiza la función objetivo en todos los casos. En cuánto al tiempo de ejecución, considero que he realizado una implementación óptima que permite que el algoritmo no tarde mucho más que los de comparación.

PRÁCTICA 2

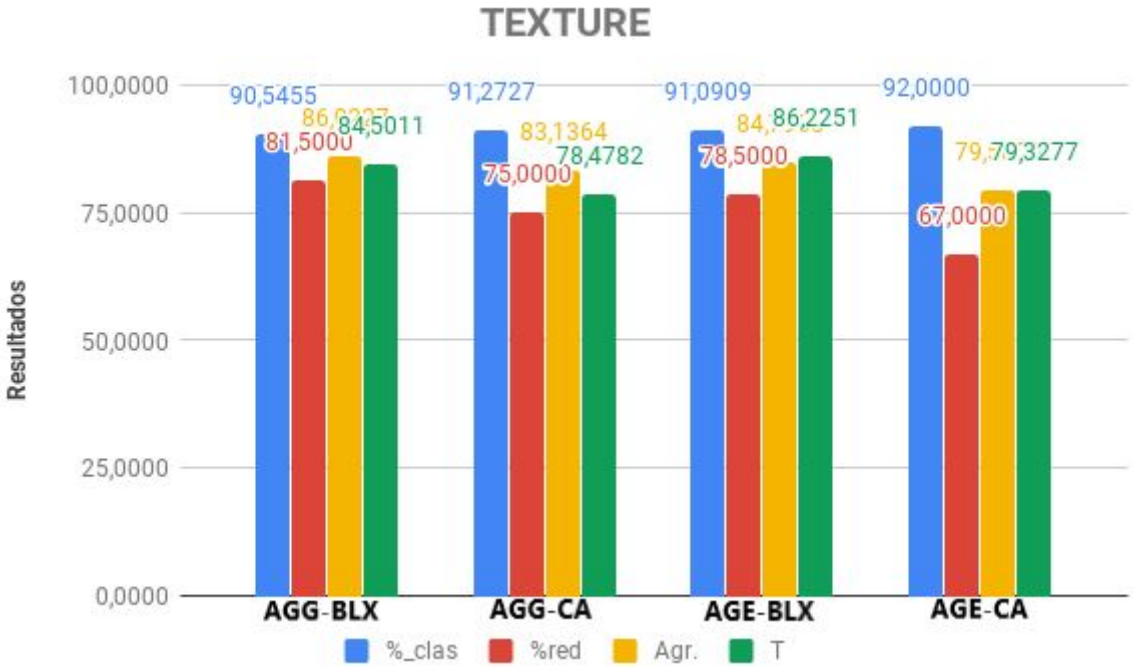
Nuevamente voy a apoyarme en gráficas de barras para analizar de forma más ilustrativa los resultados obtenidos. En primer lugar, voy a comparar el conjunto de algoritmos genéticos entre sí en los diferentes datasets, es decir, AGG-BLX, AGG-CA, AGE-BLX y AGE-CA.



Para éste primer dataset, observamos que en líneas generales el AGG obtiene ligeramente mejores resultados. En cuanto a la tasa de acierto, AGG obtiene mejores resultados en sus dos variantes. La tasa de reducción como se ve en AGE-BLX es la máxima de las 4 gráficas, pero no considero que por ello esa variante sea mejor porque a cambio de esa reducción estamos sacrificando un porcentaje de tasa de acierto y un pellizco de tiempo con respecto a AGG-BLX. Si hablamos de la agregación, el valor es bastante parejo en todos excepto en el AGE-CA. En términos generales, considero que el motivo por el que el AGE no es mejor algoritmo en éste dataset es porque en su variante con cruce aritmético, no consigue dar unos resultados que estén a la altura de ninguno de los otros 3.



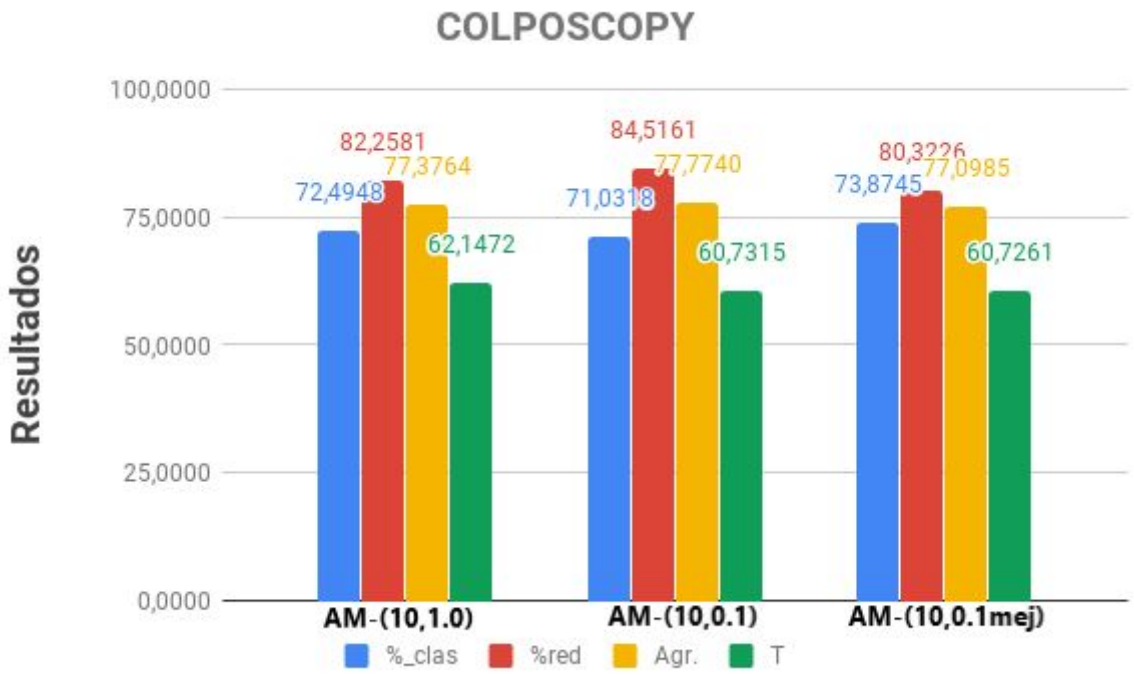
En éste dataset, vemos como se le da la vuelta totalmente al resultado anterior. El AGE es claramente el algoritmo que ofrece un mejor rendimiento para dicho dataset. No hay un sólo motivo por el que no decantarse por él; en cuanto a la tasa de acierto, supera a AGG tanto si comparamos versiones con BLX como con CA. La tasa de reducción también es mayor, al igual que la agregación y todo ello en un tiempo menor, no con una diferencia muy significativa, pero es un motivo más por el que sale victorioso el AGE.



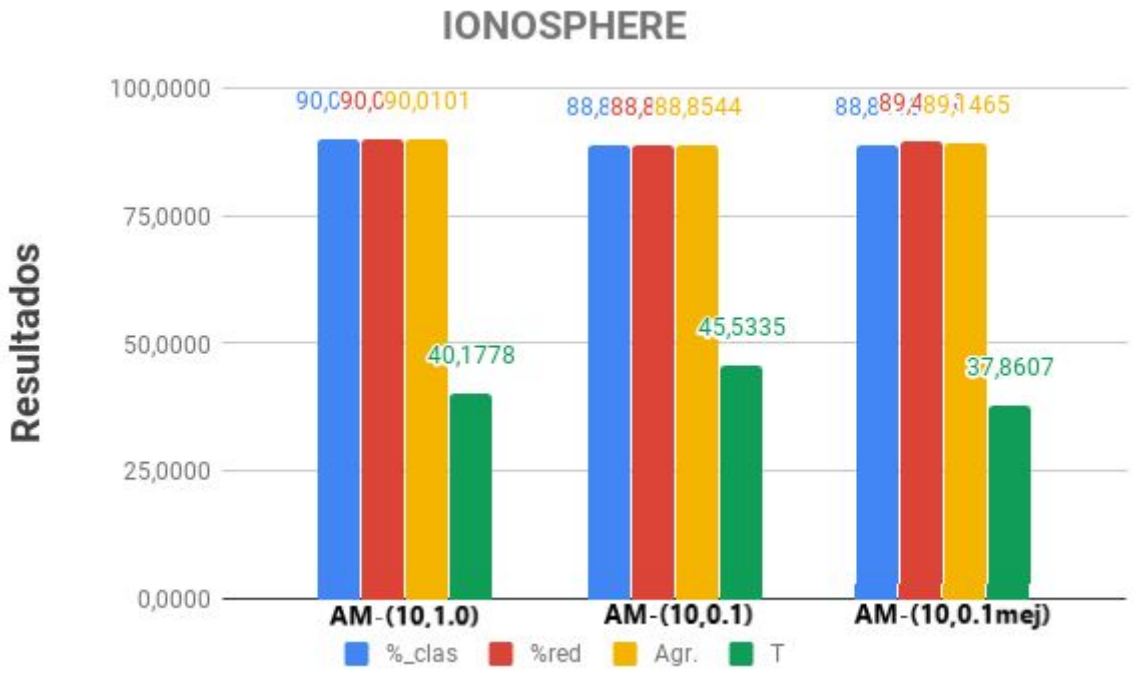
Nuevo dataset, y el resultado vuelve a cambiar, aquí el mejor algoritmo es el AGG. Si bien es cierto que el AGE consigue una mayor tasa de acierto en todas sus versiones con respecto a AGG, empeora en todo lo demás. El hecho de el agregado sea superior en AGG nos hace ver que el hecho de que AGE tenga mayor tasa de acierto no es suficiente como para obtener mejor rendimiento, puesto que no es suficiente tener una estupenda tasa de acierto pero no estar a la altura en la tasa de reducción. Además, el tiempo de cómputo para el AGE es superior.

Conclusión: Tras analizar los 3 datasets, hemos podido comprobar que es difícil establecer como mejor o peor un algoritmo sin conocer el dataset sobre el que se va a aplicar, puesto que según la diversidad que presente el dataset, AGG o AGE se adapta mejor o peor. Aún así, en términos generales, el AGG ha obtenido un mejor rendimiento para la mayoría de los casos.

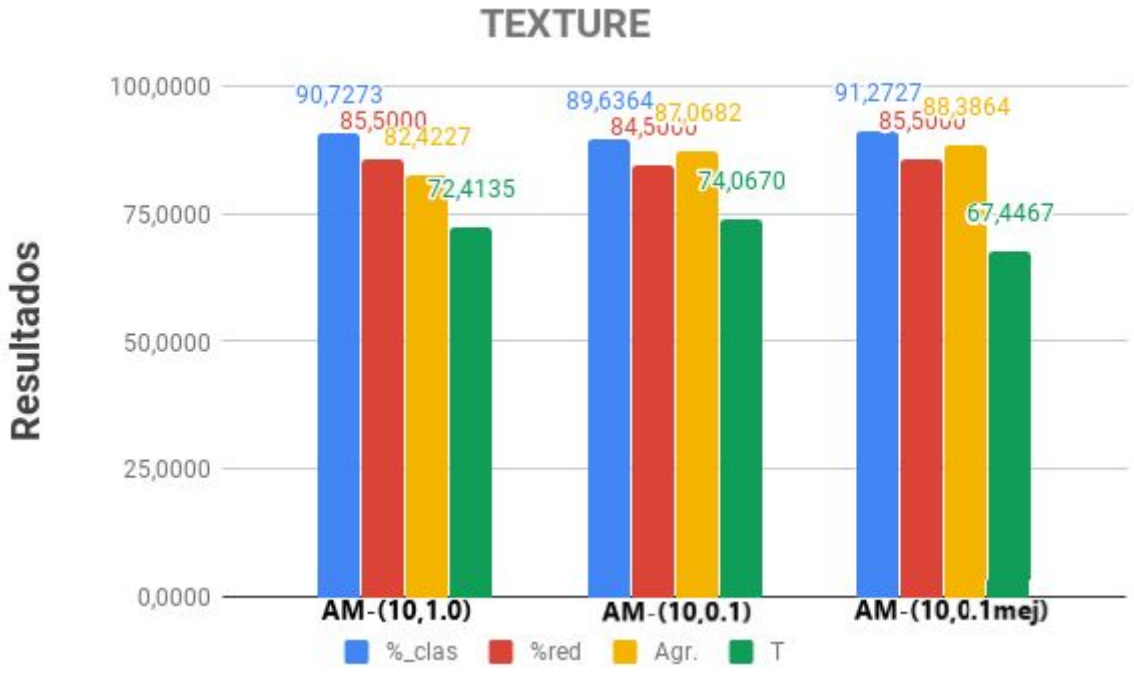
A continuación, procedo a comparar las diferentes versiones del Algoritmo Memético:



Para éste dataset vemos que es difícil comparar los resultados pues son muy próximos entre sí. Si nos ceñimos estrictamente el valor de la función objetivo, el agregado, si que es cierto que la versión del AM-(10,0.1) sale victoriosa, aunque no por mucho. Sí que podríamos descartar el AM-(10,0.1mej) pues en éste el hecho de tener mejor tasa de acierto no ha sido suficiente para contrarrestar el déficit de tasa de reducción. En general, el AM-(10,0.1) sería el mejor para éste caso, no sólo por su agregado sino por su superioridad tanto en tasa de reducción como en tiempo.



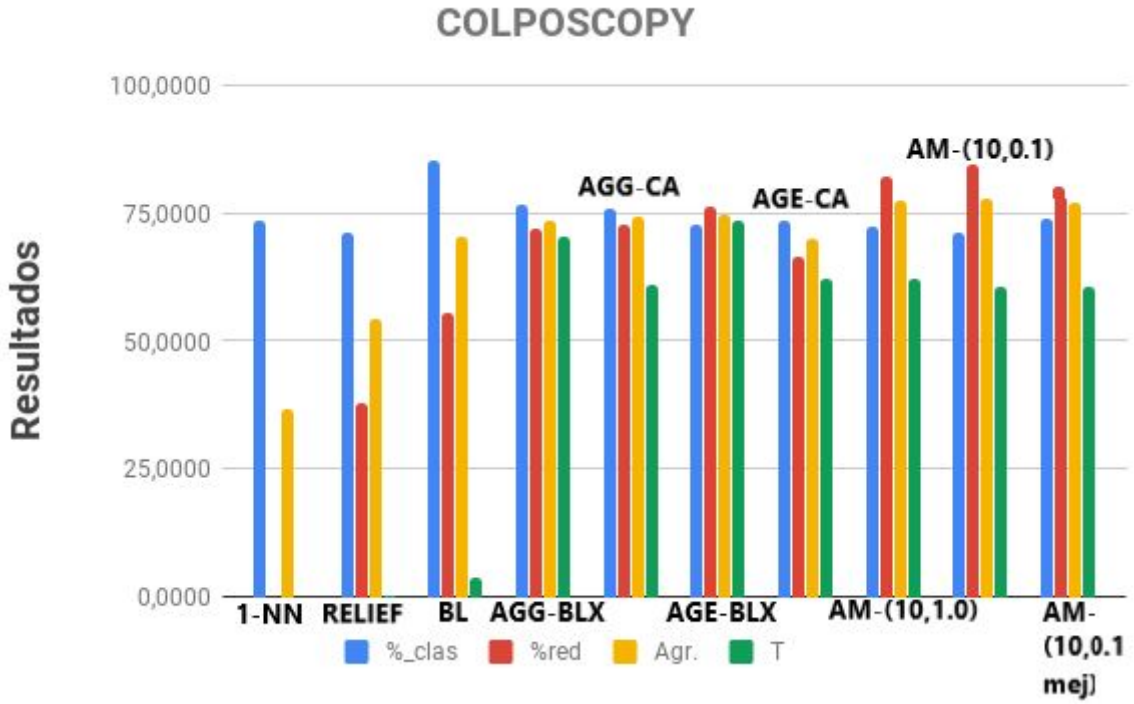
Aquí la disputa estaría entre AM-(10,1.0) y AM-(10,0.1mej), AM-(10,0.1) queda descartado automáticamente pues se observa a simple vista que es el que peor rendimiento ha obtenido de los 3. La diferencia entre los 2 primeros comentados anteriormente es mínima, quizá habría que tener en cuenta cuántas veces va a ser ejecutado el algoritmo, para intentar decantar la balanza hacia un lado u otro por el tiempo de cómputo.



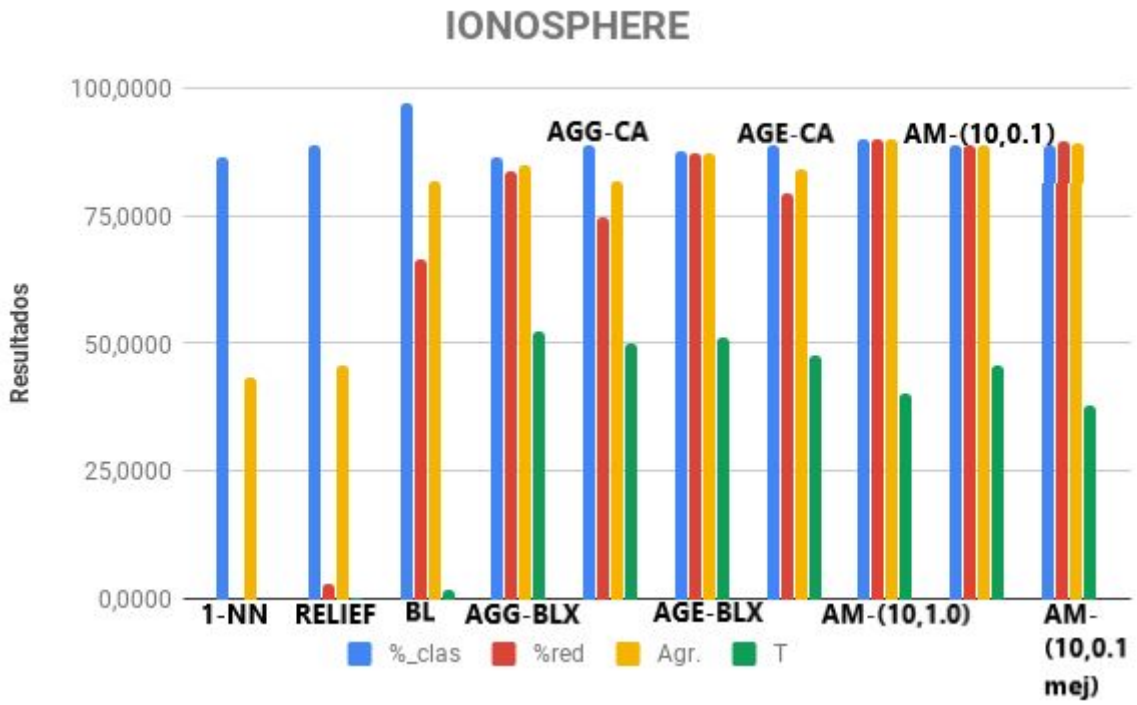
En éste dataset queda bastante claro que el mejor ha sido el AM-(10,0.1mej). No sólo ha obtenido una mejor tasa de acierto, de reducción y de agregado, sino que ha conseguido hacerlo en menor tiempo.

Conclusión: Depende mucho de la diversidad del dataset el hecho de elegir un variante del AM u otra.

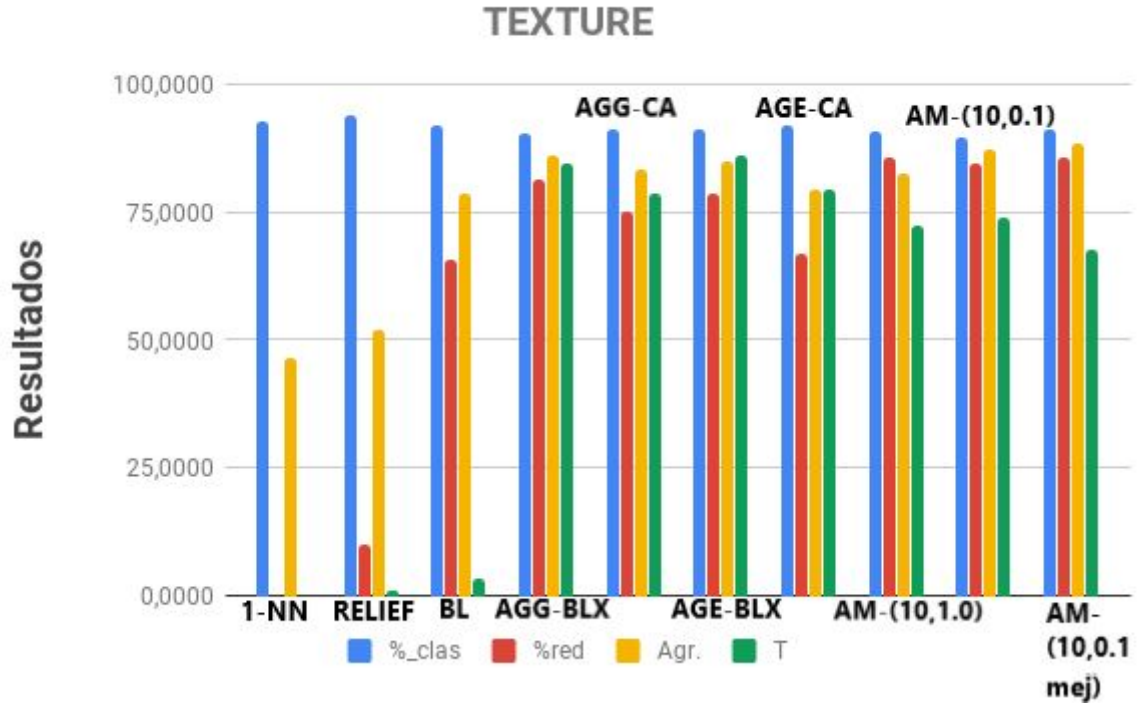
Por último, voy a hacer un breve análisis de todos los algoritmos estudiados hasta la fecha, presentando unas gráficas comparativas de su rendimiento sobre los diferentes datasets:



Como vemos, la Búsqueda Local ha sido el algoritmo con el que mayor tasa de acierto hemos tenido para éste dataset. Eso está claro, pero a medida que hemos introducido vemos que la tendencia está en mejor el valor de la función objetivo. Es fácil observar el progreso creciente que ha tenido, pero como también se observa con facilidad, a cambio de mejorar el agregado, se han sacrificado ciertos factores. El primero de ellos y más evidente, es el tiempo de ejecución, que es decenas de veces mayor en los algoritmos Genéticos y Meméticos con respecto a la Búsqueda Local. Observando los resultados de todos los algoritmos, da la sensación de que el mejor equilibrio se obtiene sacrificando algo más la tasa de acierto que la tasa de reducción.



Para éste dataset, prácticamente podríamos repetir lo ya comentado. La Búsqueda Local es el claro competidor de los nuevos algoritmos, pero insuficiente en cuanto a resultados. Se ve superada por ese equilibrio que obtienen por ejemplo los algoritmos meméticos con la tasa de acierto, tasa de reducción y agregado. De nuevo, el mayor sacrificio, el tiempo.



Por último, tenemos éste dataset, en el que el análisis es muy similar a los anteriores. La Búsqueda Local ha sido el rival más digno para éstos nuevos

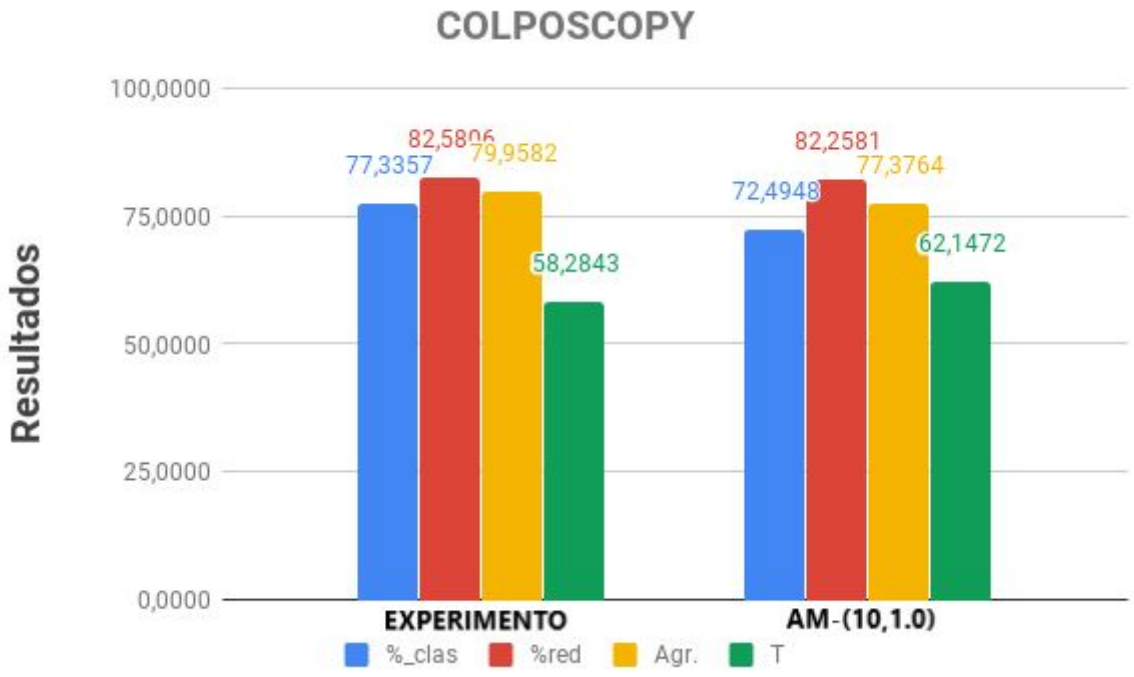
algoritmos. Si bien es cierto que el Relief obtiene una gran tasa de acierto, ni de lejos es suficiente para compararse con los demás.

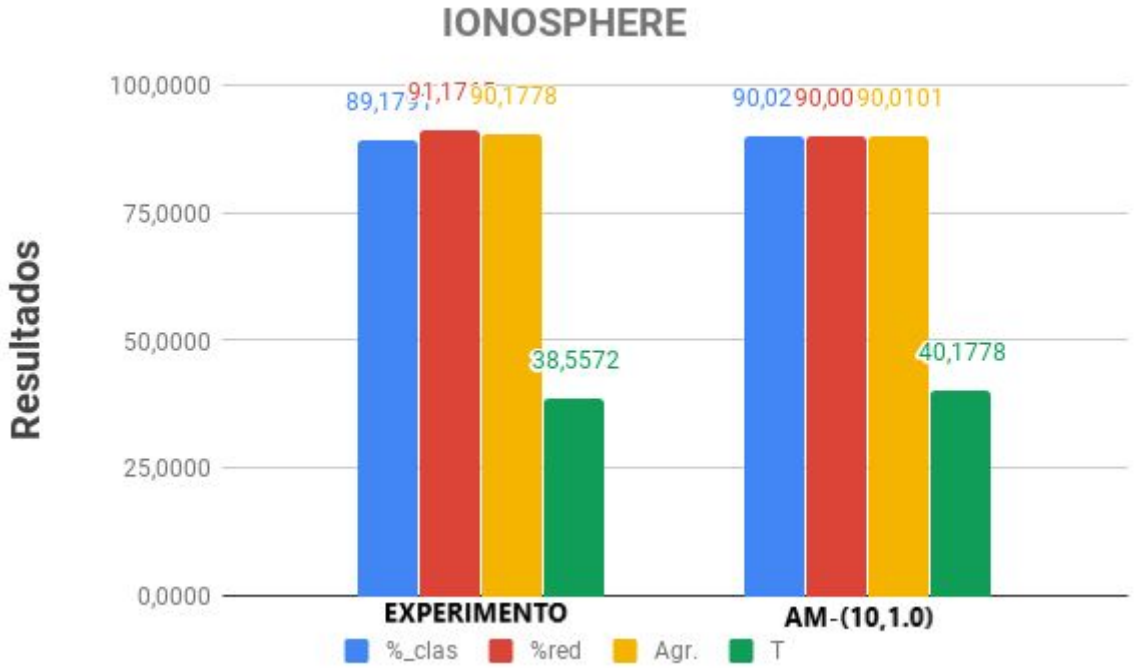
CONCLUSIÓN GENERAL

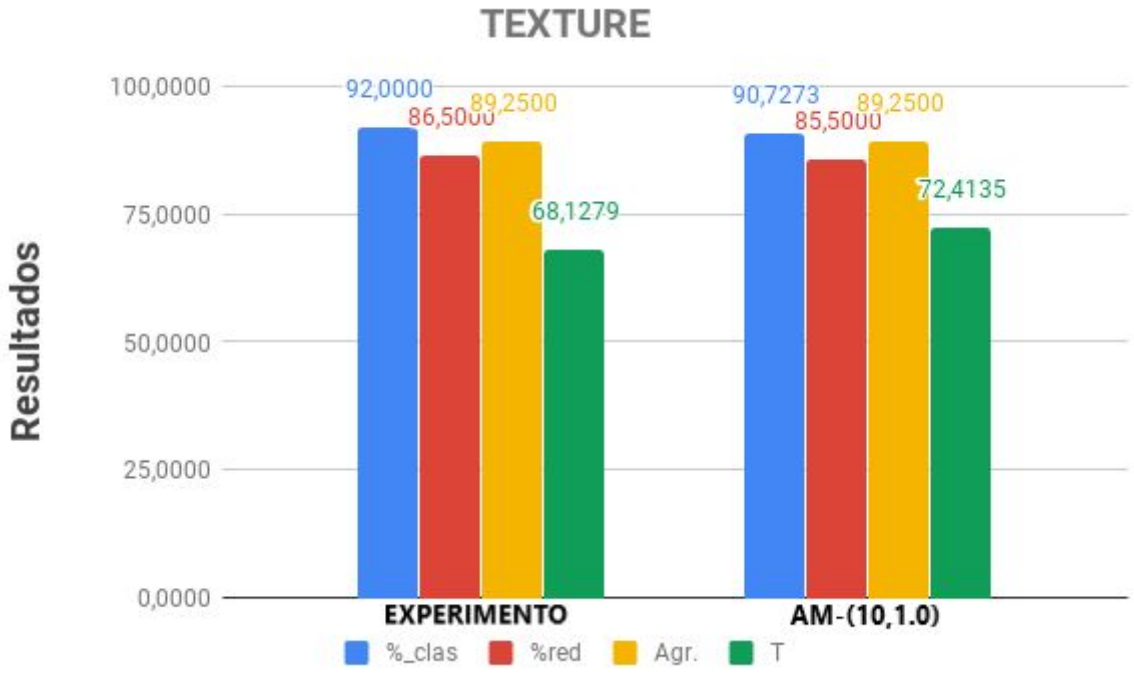
Los Algoritmos Meméticos son una combinación extraordinaria, pues combina un Algoritmo Genético, el cual ya obtiene buen rendimiento de por sí, con la Búsqueda Local que es el algoritmo con mejor rendimiento que conocíamos hasta la fecha de empezar ésta práctica. Tal y como reflejan los resultados, el rendimiento de los Meméticos es bastante bueno, consiguiendo unos valores muy compensados de tasa de acierto, tasa de reducción y agregado. El único pero, el tiempo de ejecución, que se incrementado bastante con respecto a la ejecución de la Búsqueda Local.

EXPERIMENTO EXTRA

El hecho de que el Algoritmo Memético llámase a Búsqueda Local cada 10 generaciones, me parecía quizá demasiado, así que he decidido experimentar con el AM-(10-1.0) llamando a Búsqueda Local cada 2 generaciones y comprobar si mejora los resultados:







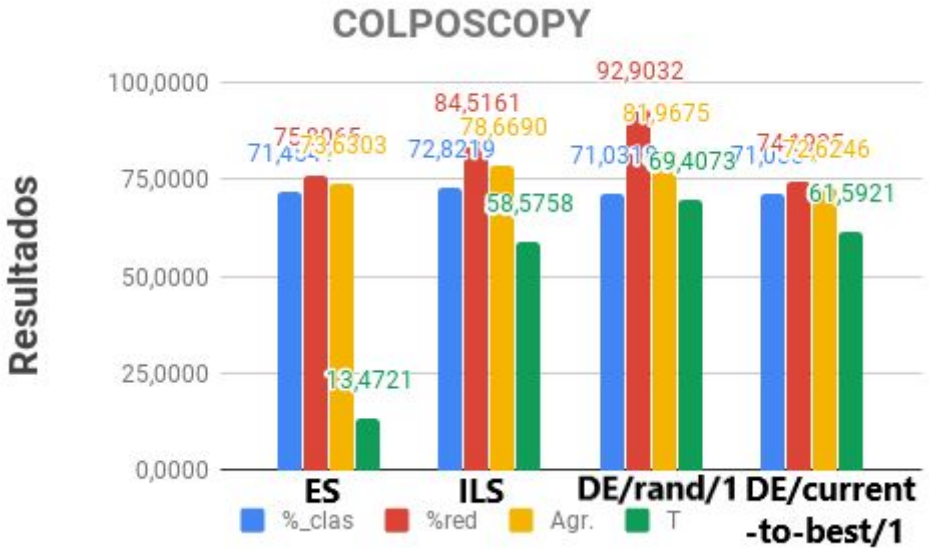
Como se puede ver en las 3 gráficas, hemos mejorado los resultados. A la izquierda tenemos el rendimiento obtenido en el experimento y a la derecha los datos del AM-(10-1.0). En términos generales, la tasa de acierto, la tasa de reducción y el agregado por tanto se ven aumentados, es decir, reduciendo el número de generaciones necesarias para aplicar Búsqueda Local obtenemos un mejor rendimiento. Como curiosidad, el tiempo de ejecución en los 3 datasets ha sido menor en el experimento, lo cuál es positivo.

Tabla de comparación de resultados:

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Experimiento	77,3357	82,5806	79,9582	58,2843	89,1791	91,1765	90,1778	38,5572	92,0000	86,5000	89,2500	68,1279
AM-(10,1.0)	72,4948	82,2581	77,3764	62,1472	90,0201	90,0000	90,0101	40,1778	90,7273	85,5000	89,2500	72,4135

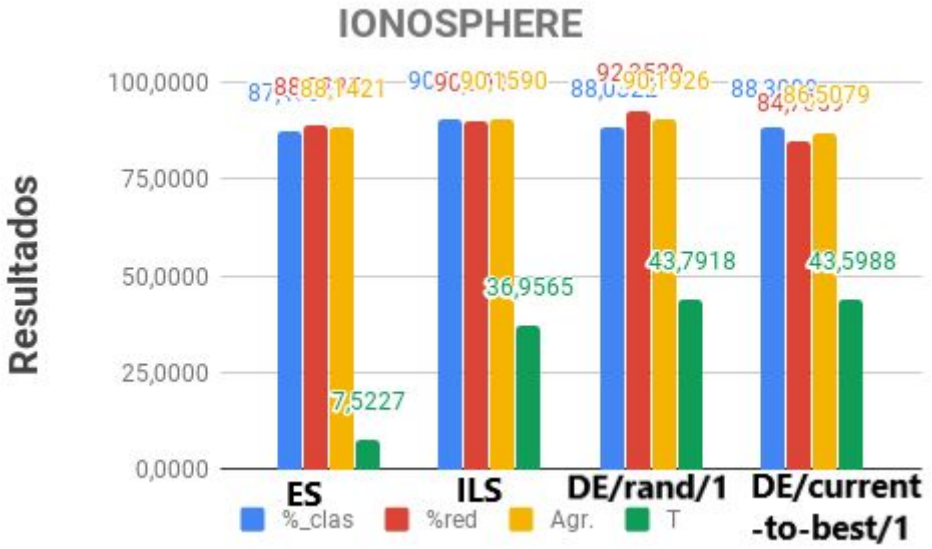
PRÁCTICA 3

Para seguir con la dinámica de las anteriores prácticas, voy a fundamentar mis conclusiones a parte de en los resultados obtenidos en las tablas, en gráficos de barras. En primer lugar, voy a hacer un análisis de los algoritmos propios de ésta práctica entre ellos, como son ES, ILS, ED/rand/1 y ED/current-to-best/1 y posteriormente pasaré a compararlos con los algoritmos de la primera práctica 1-NN y Relief.

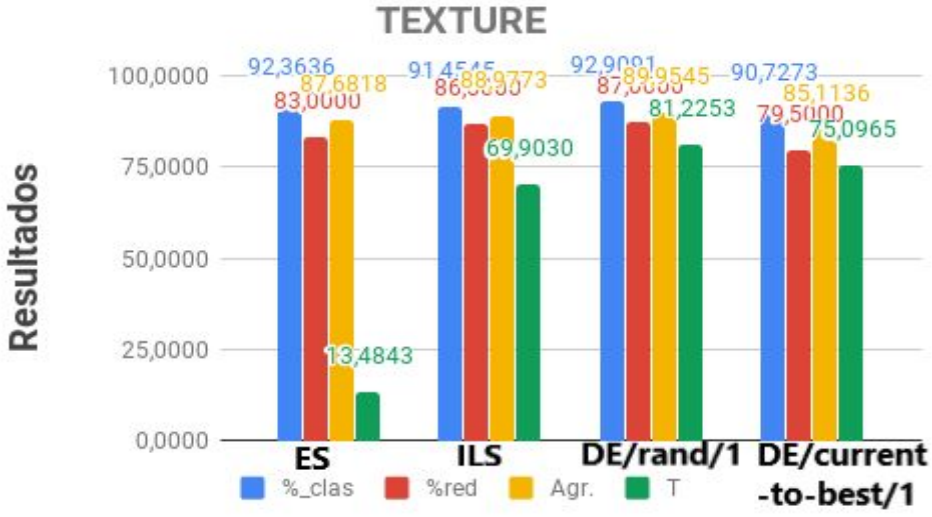


Partimos comentando el dataset Colposcopy. A simple vista apreciamos que la tasa de clasificación es común a los 4 algoritmos implementados, por lo que en éste aspecto no vemos ningún rendimiento mejor por parte de uno u otro. El resto de características sí nos permite apreciar diferencias.
Vemos que el algoritmo de búsqueda evolutiva DE/rand/1 es el que mejor rendimiento ha obtenido. Principalmente ha conseguido una tasa de reducción muy superior a la de los demás algoritmos, lo que implica que pese a que la tasa de clasificación sea pareja, el valor de la función objetivo aumenta gracias a esa reducción. Es curioso que éste mismo algoritmo pero con la variante current-to-best/1 obtiene resultados bastante más bajos que el ya comentado.
También mencionar el buen rendimiento del algoritmo basado en trayectorias múltiples ILS, el cual ha conseguido sacar pecho frente a los demás algoritmos exceptuando DE/rand/1. Un factor a tener en cuenta es el tiempo necesario para obtener la solución, ya que ILS pese a no obtener un valor de la función objetivo

tan alto como DE/rand/1, sí que se acerca bastante y consigue hacerlo ahorrando 10 segundos, por lo tanto es un aspecto a tener en cuenta. El ES pese a tardar considerablemente menos en obtener una solución, sus resultados no están a la altura de los demás.

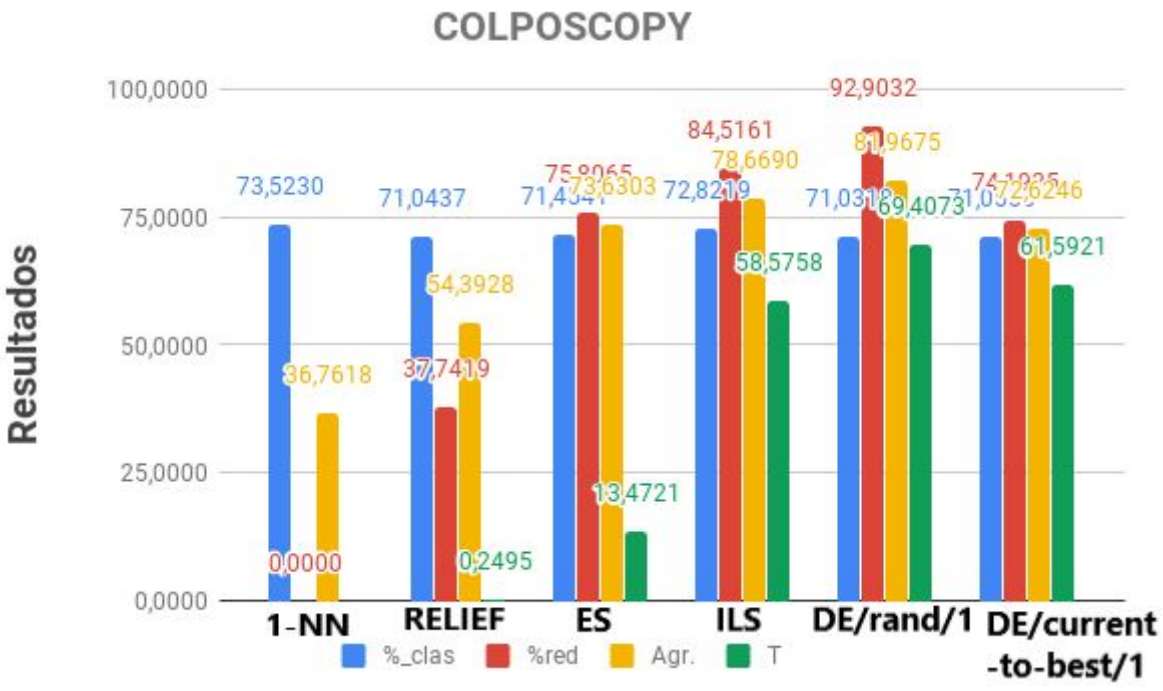


En éste dataset, podríamos aplicar muchos de los aspectos comentados en el anterior. DE/rand/1 vuelve a ser el algoritmo que más despunta gracias a que consigue una excelente tasa de reducción. En cambio, ésta vez no está tan claro que merezca la pena frente a los demás, ya que si tenemos en cuenta el algoritmo ES, sus resultados son muy muy parejos en ésta ocasión pero consigue obtenerlos 6 veces más rápido. En general, los 4 algoritmos han obtenido resultados muy parejos, pero sin duda, ES presenta la ventaja de la rapidez en obtener las soluciones sin sacrificar demasiada calidad en lo que a la función objetivo se refiere.

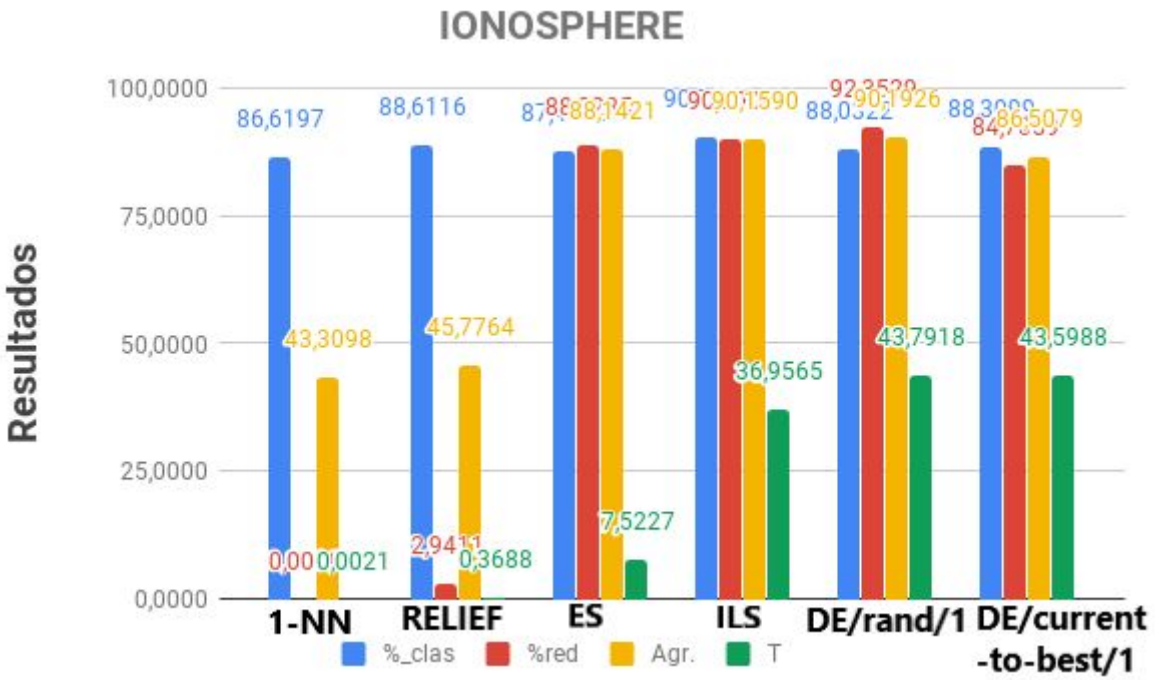


Se repite la misma historia, DE/rand/1 vuelve a ser el algoritmo que mejores resultados presenta, aunque eso sí, con un tiempo requerido para obtener la solución bastante superior a los demás algoritmos. ILS vemos que en los 3 dataset ofrece unos resultados muy aceptables, consigue buscar un equilibrio entre tasa de clasificación y tasa de reducción que hace que tenga un valor de la función objetivo elevado. DE/current-to-best/1 como vemos en los 3 dataset, no consigue adaptarse correctamente a ninguno, siendo el peor de los 4 algoritmos.

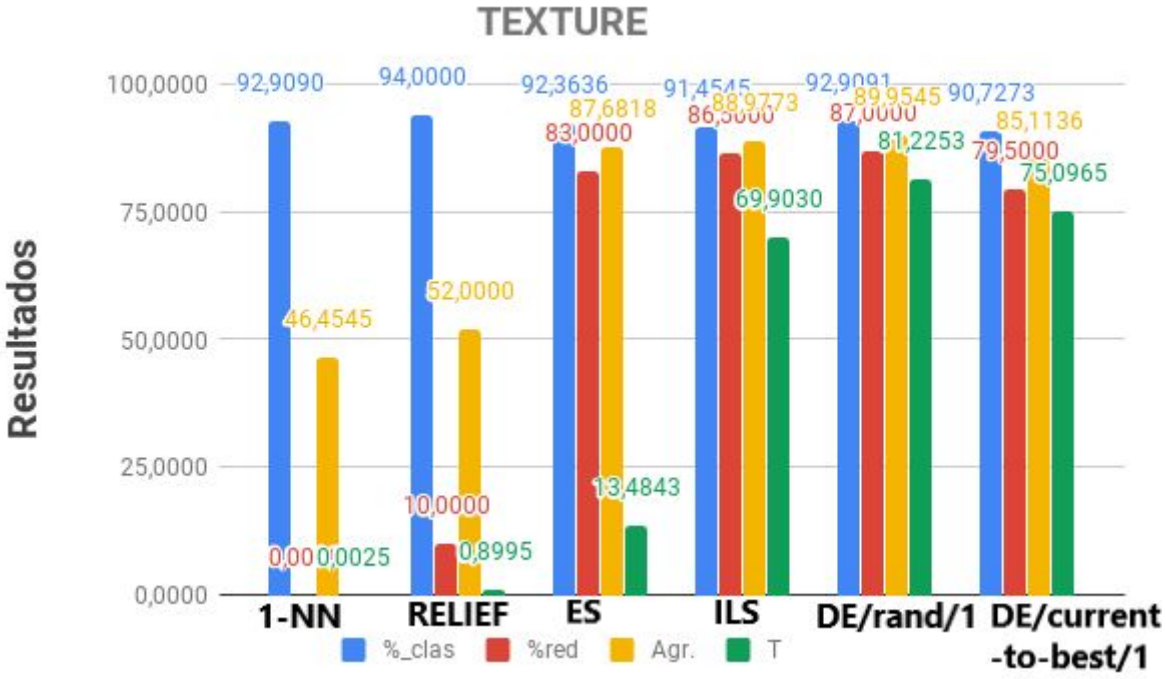
A continuación, voy a realizar una breve comparación entre los 4 algoritmos ya comentados y los algoritmos de la primera práctica 1-NN y Relief:



Como se puede ver, el algoritmo más simple, 1-NN es el que mayor tasa de clasificación obtiene. Su problema está en que al no conseguir reducción, la función objetivo toma valor 0, como es lógico. Con respecto a Relief, vemos que es un competidor, consigue una reducción notable y gracias a su buena tasa de clasificación, obtiene un valor agregado de la función objetivo que sin ser el mejor, no está mal. En ésta gráfica se evidencia la evolución de las metaheurísticas conforme aplican unas técnicas u otras, es decir pasamos de los algoritmos 1-NN y Relief bastantes simples a metaheurísticas más complejas como las de la ésta práctica cuyo objetivo es despuntar todo lo que se pueda para conseguir una función objetivo mayor.



De nuevo aquí se repita lo ya comentado, todos los algoritmos obtiene una tasa de clasificación pareja, que permite que sean ser comparados, pero en cuánto tenemos en cuenta aspectos como la tasa de reducción, ahí es dónde vemos las mayores diferencias.



En ésta ocasión, el algoritmo Relief obtiene una tasa de clasificación superior a todos los algoritmos y de nuevo su carencia, la tasa de reducción. A continuación, voy a comentar las conclusiones finales.

CONCLUSIONES FINALES

Como hemos podido comprobar, el algoritmo de Búsqueda Evolutiva DE/rand/1 es el algoritmo que mejores resultados ha obtenido en términos generales, es decir, es el algoritmo al que menos le ha influído la diversidad que puedan presentar los diferentes datasets pues se ha mantenido constante en todos. Por otro lado, su versión current-to-best/1 ha sido la que peor rendimiento ha obtenido en todos los dataset.

Por otra parte, debemos tener en cuenta el algoritmo basado en trayectorias simples ES, sin ser el mejor en términos de agregado o tasa de reducción, consigue unos resultados muy competitivos en un tiempo mucho más reducido que el resto de los algoritmos, por lo tanto concluímos que su desempeño en éstas pruebas ha sido muy positivo.

Por último comentar que el algoritmo ILS, es un algoritmo muy completo capaz de buscar un equilibrio entre tasa de clasificación y tasa de reducción que le permite llegar a valores de agregación altos, llegando a ser la competencia directa de DE/rand/1 además de conseguirlo con un tiempo menor al recién mencionado.

7. REFERENCIAS BIBLIOGRÁFICAS

- Consultas de funciones de Numpy: https://docs.scipy.org/doc/numpy/reference/index.html

- Consultas de funciones de Scikit-Learn: https://scikit-learn.org/stable/

- Consulta referente a la lectura de datos: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

- Material de clase.