Práctica 2.b:

Técnicas de Búsqueda basadas en Poblaciones 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
```

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] < 0.2
                    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
```

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 P2.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.

○ 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 5.3: Resultados obtenidos por el algoritmo BL en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	74,5763	67,7419	71,1591	6,0588	85,9155	73,5294	79,7225	2,4674	87,2727	67,5000	77,3864	4,4930
Partición 2	73,6842	59,6774	66,6808	6,1186	90,0000	76,4706	83,2353	2,7486	86,3636	72,5000	79,4318	5,4115
Partición 3	68,4211	69,3548	68,8879	5,8822	88,5714	58,8235	73,6975	2,6828	90,9091	65,0000	77,9545	5,3148
Partición 4	68,4211	72,5806	70,5008	5,8923	94,2857	70,5882	82,4370	3,0638	91,8182	62,5000	77,1591	4,6655
Partición 5	75,4386	72,5806	74,0096	5,8723	94,2857	70,5882	82,4370	2,9222	90,0000	72,5000	81,2500	4,8400
Media	72,1082	68,3871	70,2477	5,9648	90,6117	70,0000	80,3058	2,7770	89,2727	68,0000	78,6364	4,9450

Tabla 5.4: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	77,9661	69,3548	73,6605	69,7021	84,5070	94,1176	89,3123	43,3600	90,0000	85,0000	87,5000	78,9397
Partición 2	73,6842	75,8065	74,7453	67,7807	84,2857	79,4118	81,8487	57,5159	90,0000	72,5000	81,2500	91,2642
Partición 3	70,1754	70,9677	70,5716	76,0186	90,0000	82,3529	86,1765	48,4244	92,7273	80,0000	86,3636	87,4968
Partición 4	71,9298	75,8065	73,8681	71,6827	82,8571	82,3529	82,6050	53,4732	88,1818	82,5000	85,3409	85,3934
Partición 5	80,7018	67,7419	74,2218	66,7205	90,0000	79,4118	84,7059	59,1918	91,8182	87,5000	89,6591	79,4115
Media	74,8915	71,9355	73,4135	70,3809	86,3300	83,5294	84,9297	52,3931	90,5455	81,5000	86,0227	84,5011

Tabla 5.5: Resultados obtenidos por el algoritmo AGG-CA en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	77,9661	70,9677	74,4669	58,6421	88,7324	82,3529	85,5427	41,6801	90,9091	80,0000	85,4545	73,7393
Partición 2	71,9298	83,8710	77,9004	59,7174	88,5714	67,6471	78,1092	52,9752	90,0000	82,5000	86,2500	76,2679
Partición 3	75,4386	66,1290	70,7838	57,3505	92,8571	70,5882	81,7227	53,2994	91,8182	72,5000	82,1591	79,8509
Partición 4	71,9298	70,9677	71,4488	65,7117	85,7143	82,3529	84,0336	46,8297	88,1818	70,0000	79,0909	83,2582
Partición 5	82,4561	70,9677	76,7119	63,8292	87,1429	70,5882	78,8655	55,0077	95,4545	70,0000	82,7273	79,2749
Media	75,9441	72,5806	74,2624	61,0502	88,6036	74,7059	81,6548	49,9584	91,2727	75,0000	83,1364	78,4782

Tabla 5.6: Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	74,5763	74,1935	74,3849	74,6263	90,1408	88,2353	89,1881	50,4520	93,6364	80,0000	86,8182	90,4923
Partición 2	68,4211	87,0968	77,7589	67,3204	88,5714	88,2353	88,4034	51,7266	90,9091	77,5000	84,2045	85,0274
Partición 3	66,6667	79,0323	72,8495	73,3198	87,1429	85,2941	86,2185	52,6602	90,9091	80,0000	85,4545	81,8141
Partición 4	75,4386	70,9677	73,2032	74,7141	85,7143	91,1765	88,4454	46,3400	87,2727	75,0000	81,1364	89,2910
Partición 5	78,9474	70,9677	74,9576	78,5647	85,7143	82,3529	84,0336	55,4981	92,7273	80,0000	86,3636	84,5006
Media	72,8100	76,4516	74,6308	73,7090	87,4567	87,0588	87,2578	51,3354	91,0909	78,5000	84,7955	86,2251

Tabla 5.7: Resultados obtenidos por el algoritmo AGE-CA en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	77,9661	69,3548	73,6605	57,5735	87,3239	73,5294	80,4267	45,8124	92,7273	65,0000	78,8636	80,9279
Partición 2	75,4386	62,9032	69,1709	62,6683	85,7143	79,4118	82,5630	53,7512	92,7273	62,5000	77,6136	78,3434
Partición 3	68,4211	64,5161	66,4686	61,0910	88,5714	82,3529	85,4622	39,4619	91,8182	62,5000	77,1591	83,2869
Partición 4	70,1754	70,9677	70,5716	65,7844	92,8571	82,3529	87,6050	45,0474	88,1818	80,0000	84,0909	74,8926
Partición 5	75,4386	64,5161	69,9774	63,4193	88,5714	79,4118	83,9916	54,7076	94,5455	65,0000	79,7727	79,1876
Media	73,4880	66,4516	69,9698	62,1073	88,6076	79,4118	84,0097	47,7561	92,0000	67,0000	79,5000	79,3277

Tabla 5.8: Resultados obtenidos por el algoritmo AM-(10,1.0) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	69,4915	82,2581	75,8748	63,2517	92,9577	91,1765	92,0671	42,2071	87,2727	87,5000	87,3864	74,9529
Partición 2	70,1754	83,8710	77,0232	61,5039	91,4286	88,2353	89,8319	37,8797	88,1818	82,5000	85,3409	75,4664
Partición 3	75,4386	82,2581	78,8483	61,1459	90,0000	91,1765	90,5882	45,9646	93,6364	82,5000	88,0682	76,9510
Partición 4	70,1754	83,8710	77,0232	57,6393	87,1429	91,1765	89,1597	34,4299	89,0909	87,5000	88,2955	67,1109
Partición 5	77,1930	79,0323	78,1126	67,1952	88,5714	88,2353	88,4034	40,4079	95,4545	87,5000	91,4773	67,5861
Media	72,4948	82,2581	77,3764	62,1472	90,0201	90,0000	90,0101	40,1778	90,7273	85,5000	88,1136	72,4135

Tabla 5.9: Resultados obtenidos por el algoritmo AM-(10,0.1) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	77,9661	87,0968	82,5314	60,1517	90,1408	91,1765	90,6587	39,6784	92,7273	82,5000	87,6136	86,3942
Partición 2	68,4211	82,2581	75,3396	58,4256	92,8571	91,1765	92,0168	36,6230	88,1818	85,0000	86,5909	66,4657
Partición 3	66,6667	82,2581	74,4624	56,4684	84,2857	85,2941	84,7899	52,1150	87,2727	85,0000	86,1364	72,7982
Partición 4	68,4211	85,4839	76,9525	63,1057	88,5714	88,2353	88,4034	48,5199	90,9091	85,0000	87,9545	70,5387
Partición 5	73,6842	85,4839	79,5840	65,5062	88,5714	88,2353	88,4034	50,7313	89,0909	85,0000	87,0455	74,1379
Media	71,0318	84,5161	77,7740	60,7315	88,8853	88,8235	88,8544	45,5335	89,6364	84,5000	87,0682	74,0670

Tabla 5.10: Resultados obtenidos por el algoritmo AM-(10,0.1mej) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	72,8814	80,6452	76,7633	63,6188	91,5493	91,1765	91,3629	37,8278	91,8182	85,0000	88,4091	67,2221
Partición 2	71,9298	79,0323	75,4810	56,8246	87,1429	91,1765	89,1597	38,6994	94,5455	85,0000	89,7727	73,3941
Partición 3	70,1754	79,0323	74,6038	59,1836	90,0000	91,1765	90,5882	35,5708	92,7273	85,0000	88,8636	66,1708
Partición 4	73,6842	85,4839	79,5840	58,6426	85,7143	85,2941	85,5042	38,3474	85,4545	87,5000	86,4773	64,4593
Partición 5	80,7018	77,4194	79,0606	65,3607	90,0000	88,2353	89,1176	38,8580	91,8182	85,0000	88,4091	65,9874
Media	73,8745	80,3226	77,0985	60,7261	88,8813	89,4118	89,1465	37,8607	91,2727	85,5000	88,3864	67,4467

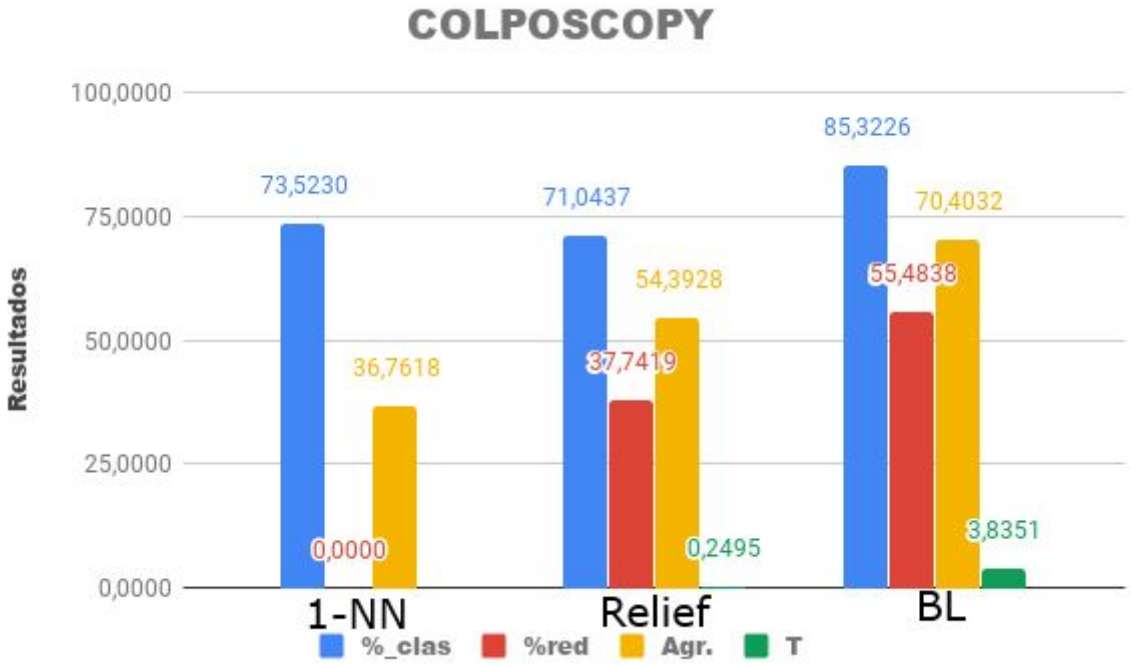
Tabla 5.11: Resultados globales en el problema del APC

	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
BL	85,3226	55,4838	70,4032	3,8351	96,8611	66,4705	81,6658	1,7979	91,8181	65,5000	78,6590	3,2822
AGG-BLX	76,6298	71,9355	73,4135	70,3809	86,3300	83,5294	84,9297	52,3931	90,5455	81,5000	86,0227	84,5011
AGG-CA	75,9441	72,5806	74,2624	61,0502	88,6036	74,7059	81,6548	49,9584	91,2727	75,0000	83,1364	78,4782
AGE-BLX	72,8100	76,4516	74,6308	73,7090	87,4567	87,0588	87,2578	51,3354	91,0909	78,5000	84,7955	86,2251
AGE-CA	73,4880	66,4516	69,9698	62,1073	88,6076	79,4118	84,0097	47,7561	92,0000	67,0000	79,5000	79,3277
AM-(10,1.0)	72,4948	82,2581	77,3764	62,1472	90,0201	90,0000	90,0101	40,1778	90,7273	85,5000	82,4227	72,4135
AM-(10,0.1)	71,0318	84,5161	77,7740	60,7315	88,8853	88,8235	88,8544	45,5335	89,6364	84,5000	87,0682	74,0670
AM-(10,0.1mej)	73,8745	80,3226	77,0985	60,7261	88,8813	89,4118	89,1465	37,8607	91,2727	85,5000	88,3864	67,4467

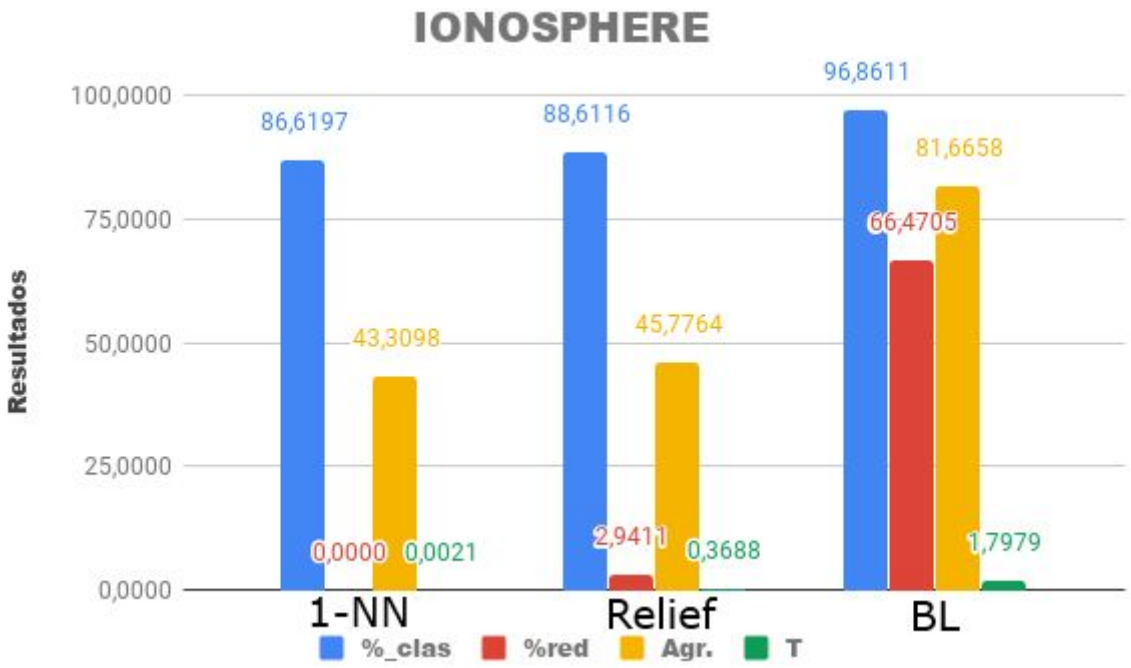
- **ANÁLISIS DE RESULTADOS**

PRÁCTICA 1

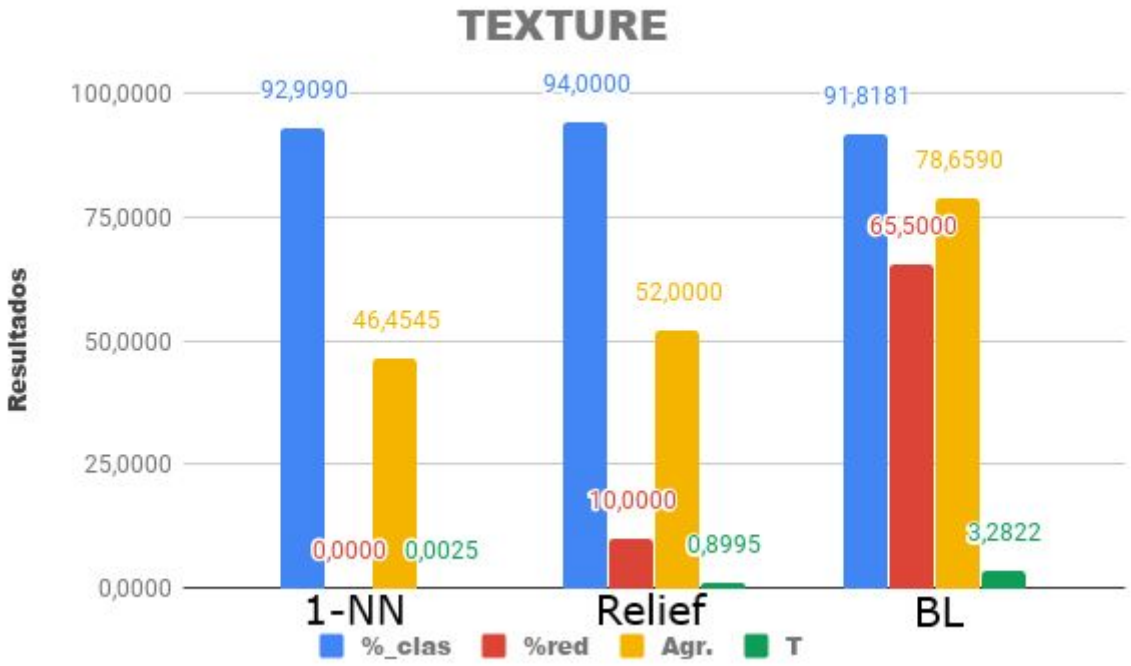
Para llevar a cabo el análisis de forma más ilustrativa, me voy a apoyar en 3 gráficas que he creado, en las que muestro una comparativa de los resultados obtenidos por cada algoritmo de forma individual en cada dataset.



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.



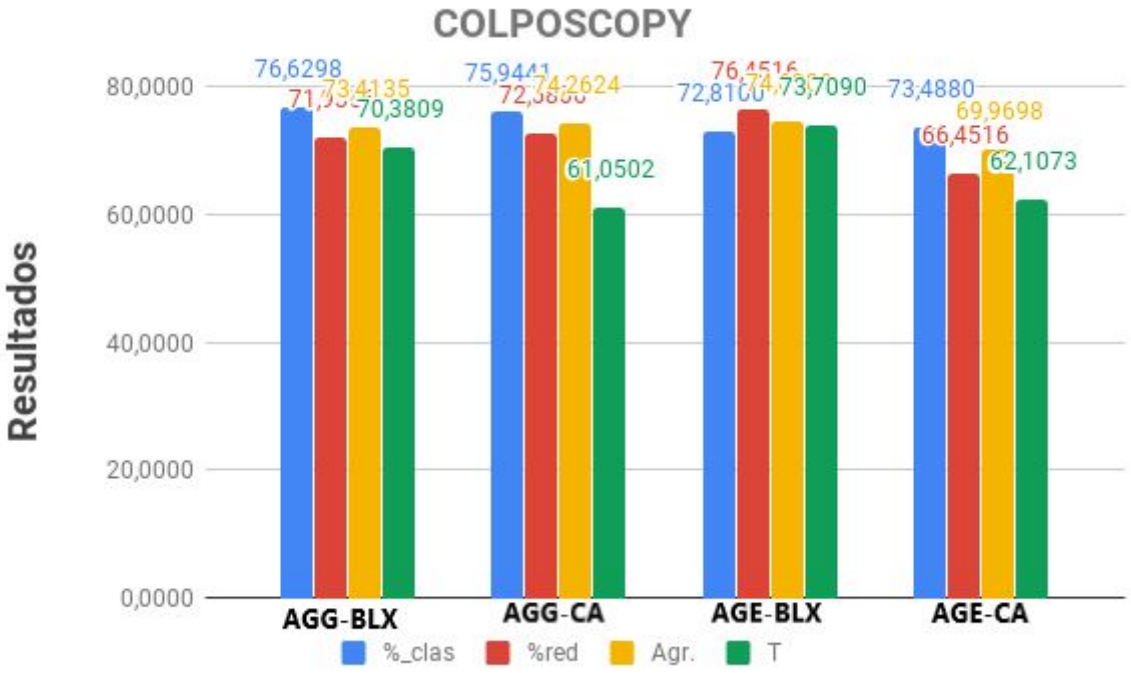
Para el último dataset, vemos que se repite todo lo comentado en los anteriores, la Búsqueda Local obtiene una mayor tasa de reducción, maximiza la función objetivo... pero en éste caso hay una particularidad, y es que para obtener esos valores de reducción y función objetivo, estamos sacrificando un 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

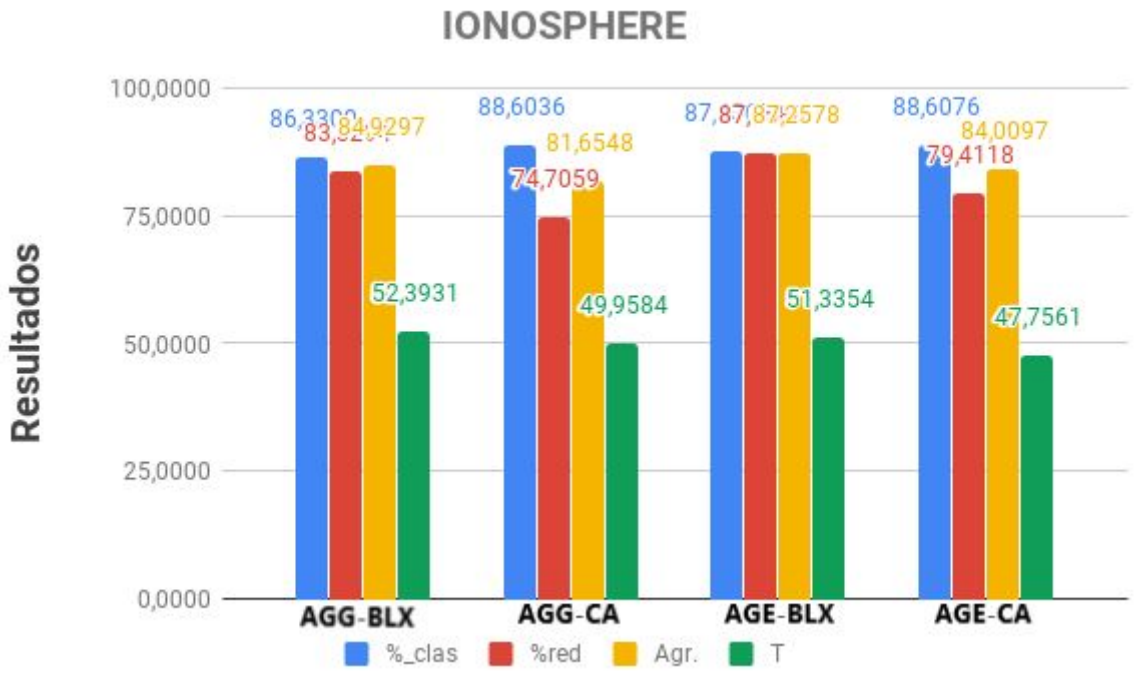
A la vista de los resultados, el algoritmo de Búsqueda Local es muy superior tanto al 1-NN como al Relief. Afronta con mayor facilidad la diversidad del dataset y siempre reduce considerablemente más que el Relief, manteniendo un nivel de tasa de aciertos más que aceptable que hace que sea el algoritmo 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

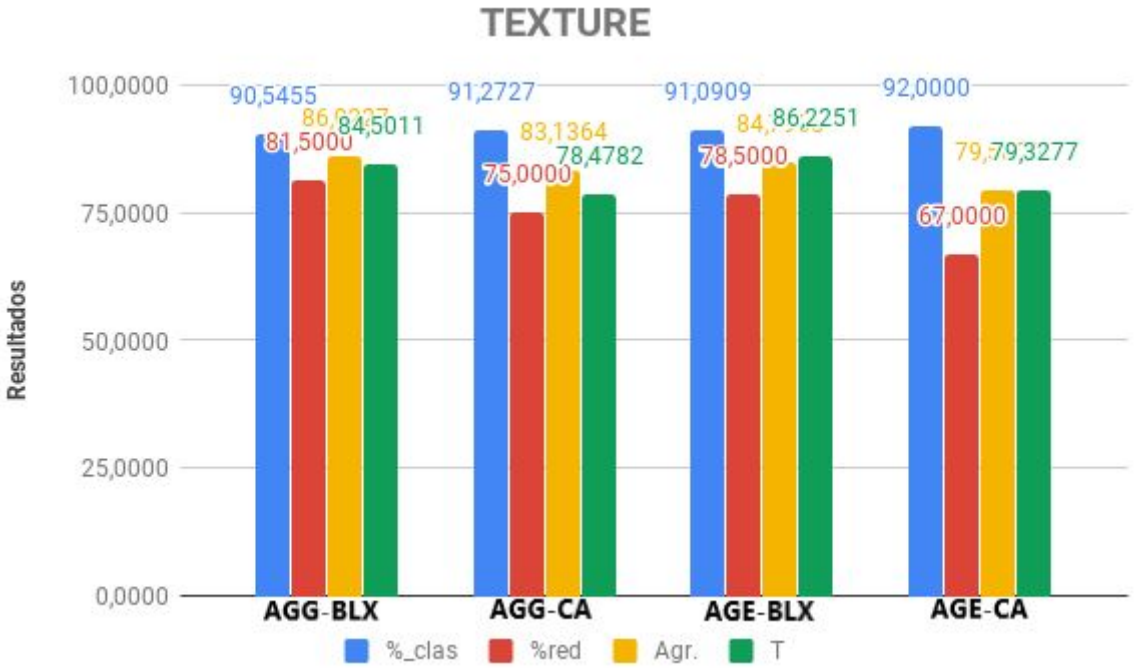
Nuevamente voy a apoyarme en gráficas de barras para analizar de forma más ilustrativa los resultados obtenidos. En primer lugar, voy a comparar el conjunto de algoritmos genéticos entre sí en los diferentes datasets, es decir, AGG-BLX, AGG-CA, AGE-BLX y AGE-CA.



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.



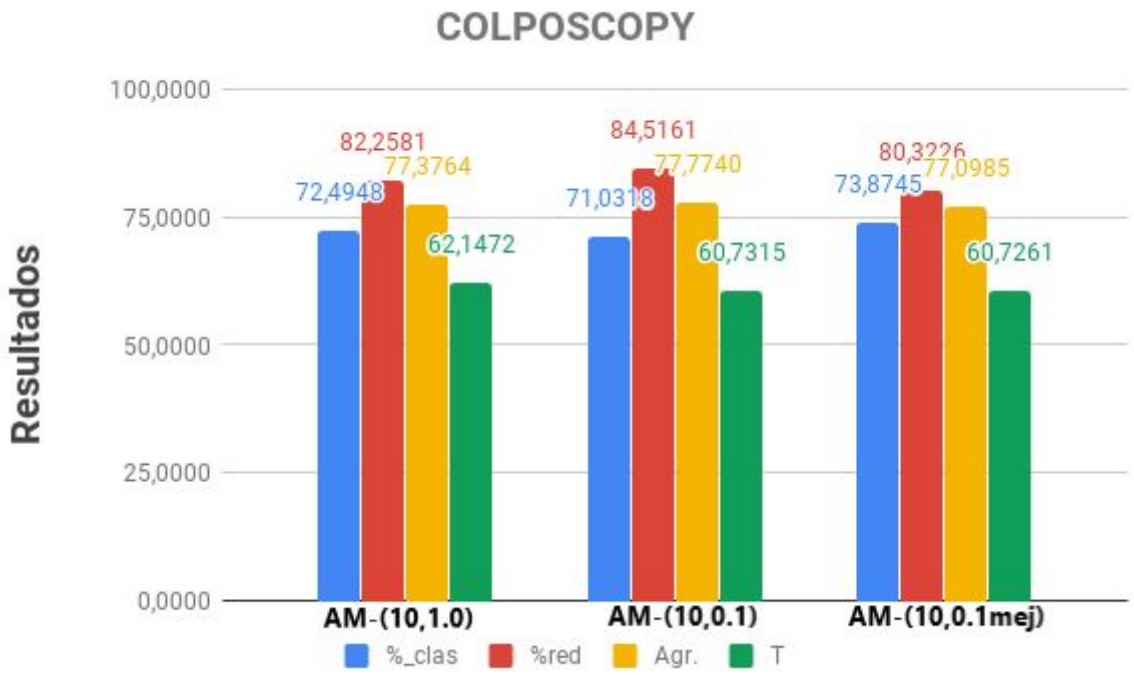
En éste dataset, vemos como se le da la vuelta totalmente al resultado anterior. El AGE es claramente el algoritmo que ofrece un mejor rendimiento para dicho dataset. No hay un sólo motivo por el que no decantarse por él; en cuanto a la tasa de acierto, supera a AGG tanto si comparamos versiones con 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.



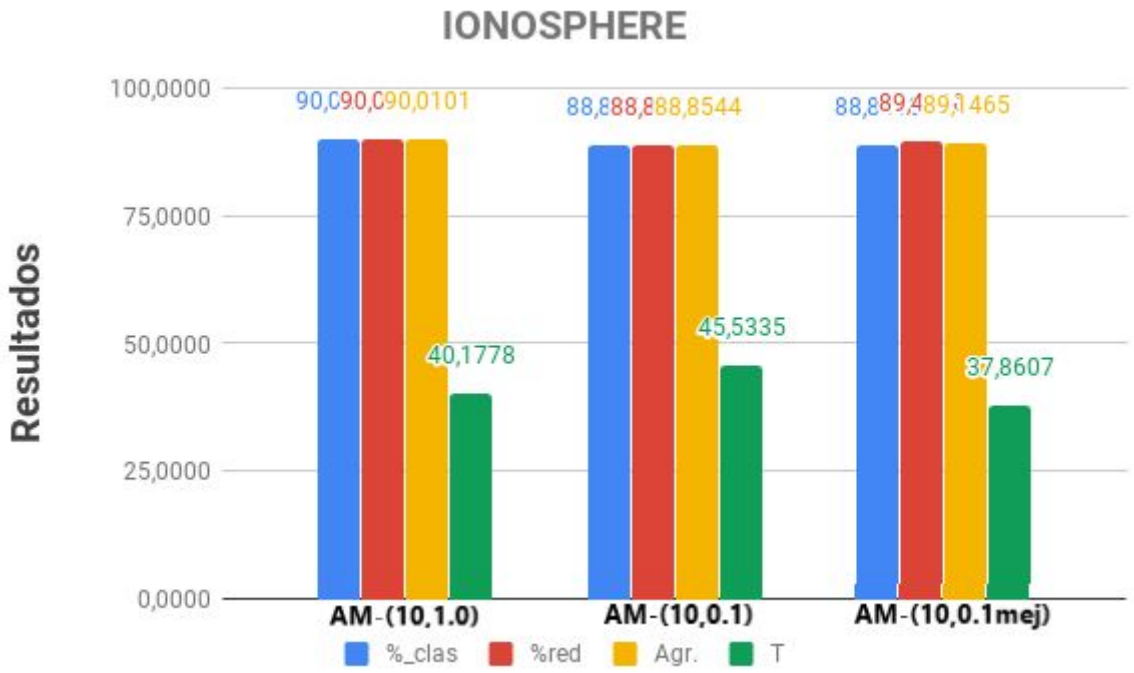
Nuevo dataset, y el resultado vuelve a cambiar, aquí el mejor algoritmo es el AGG. Si bien es cierto que el AGE consigue una mayor tasa de acierto en todas sus versiones con respecto a AGG, empeora en todo lo demás. El hecho de el agregado sea superior en AGG nos hace ver que el hecho de que AGE tenga 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.

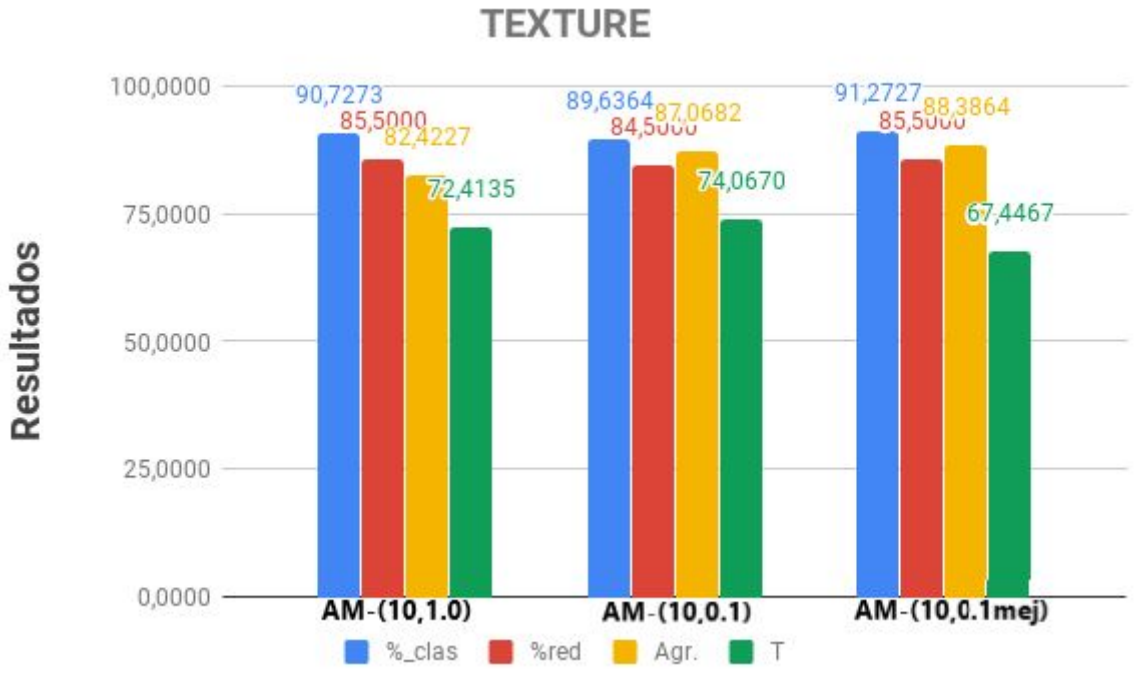
A continuación, procedo a comparar las diferentes versiones del Algoritmo Memético:



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.



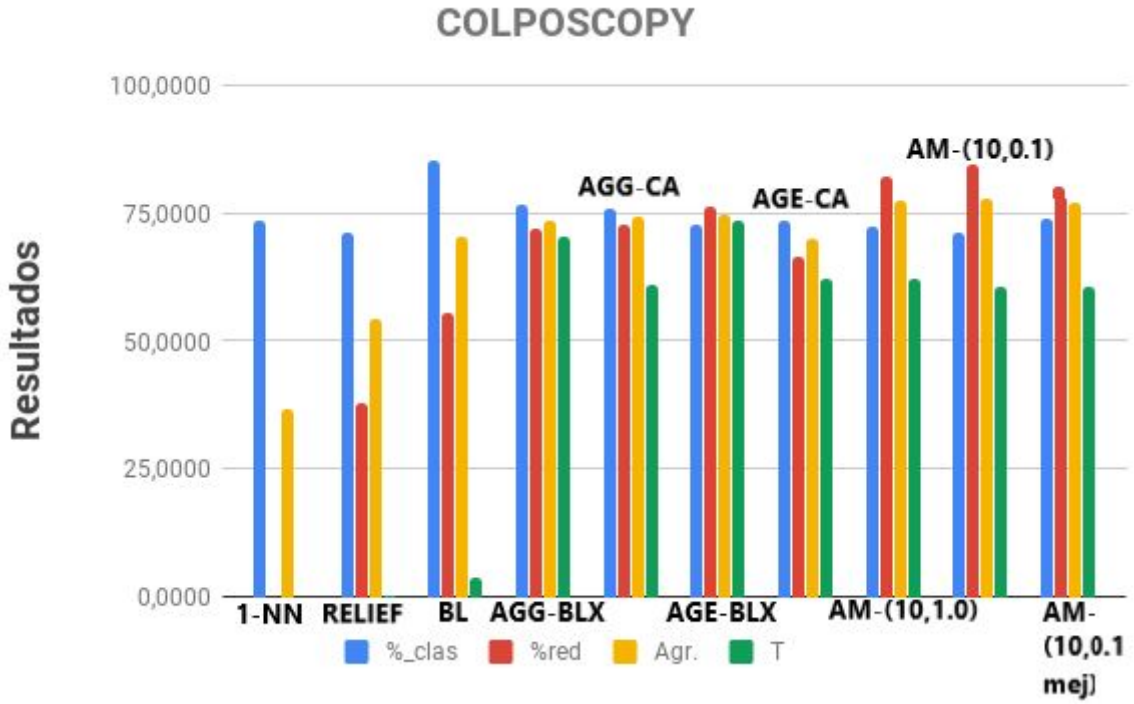
Aquí la disputa estaría entre AM-(10,1.0) y AM-(10,0.1mej), AM-(10,0.1) queda descartado automáticamente pues se observa a simple vista que es el que peor rendimiento ha obtenido de los 3. La diferencia entre los 2 primeros comentados anteriormente es mínima, quizá habría que tener en cuenta cuántas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



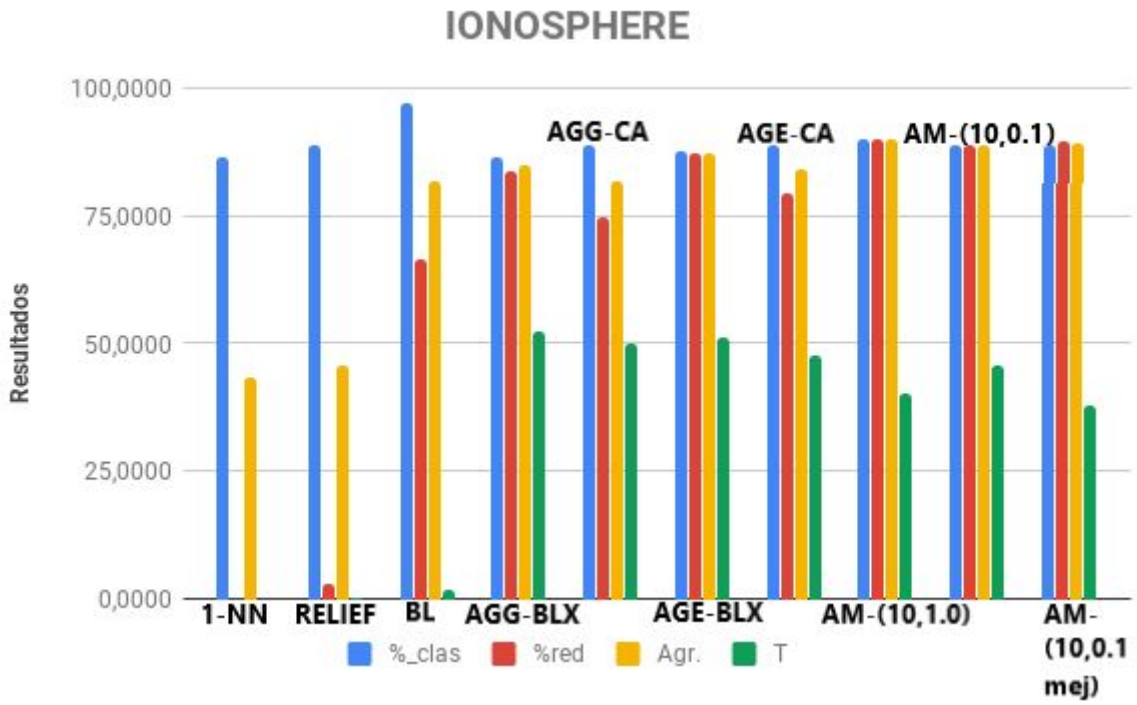
En éste dataset queda bastante claro que el mejor ha sido el AM-(10,0.1mej). No sólo ha obtenido una mejor tasa de acierto, de reducción y de agregado, sino que ha conseguido hacerlo en menor tiempo.

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

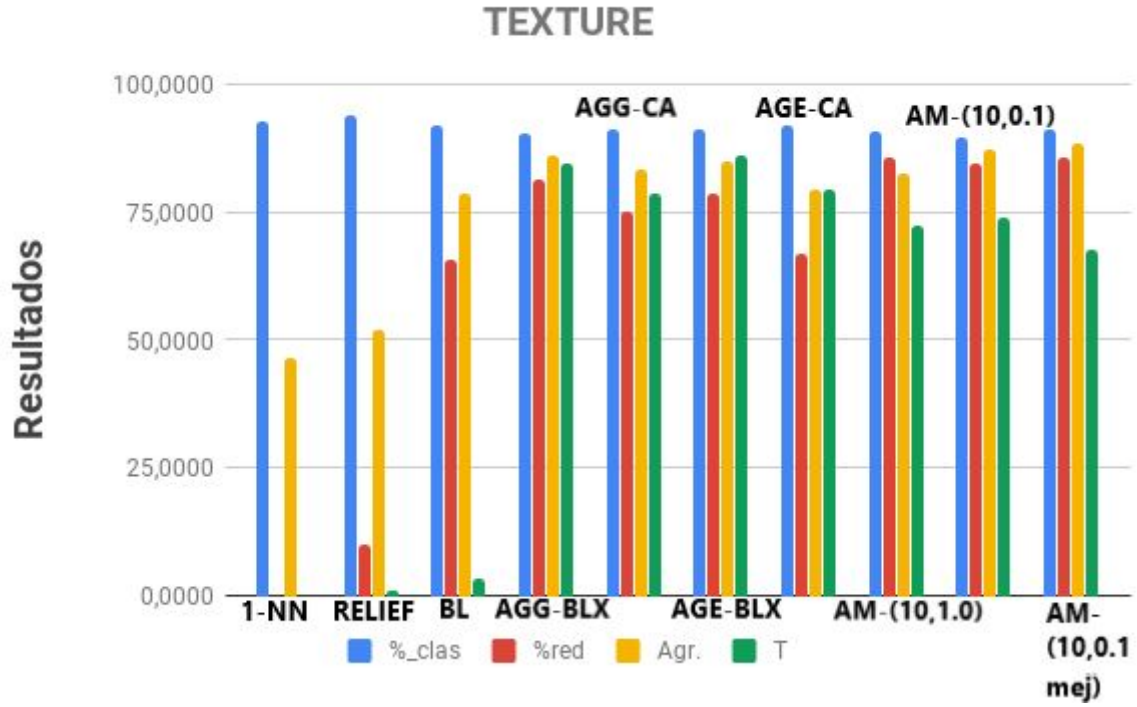
Por último, voy a hacer un breve análisis de todos los algoritmos estudiados hasta la fecha, presentando unas gráficas comparativas de su rendimiento sobre los diferentes datasets:



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

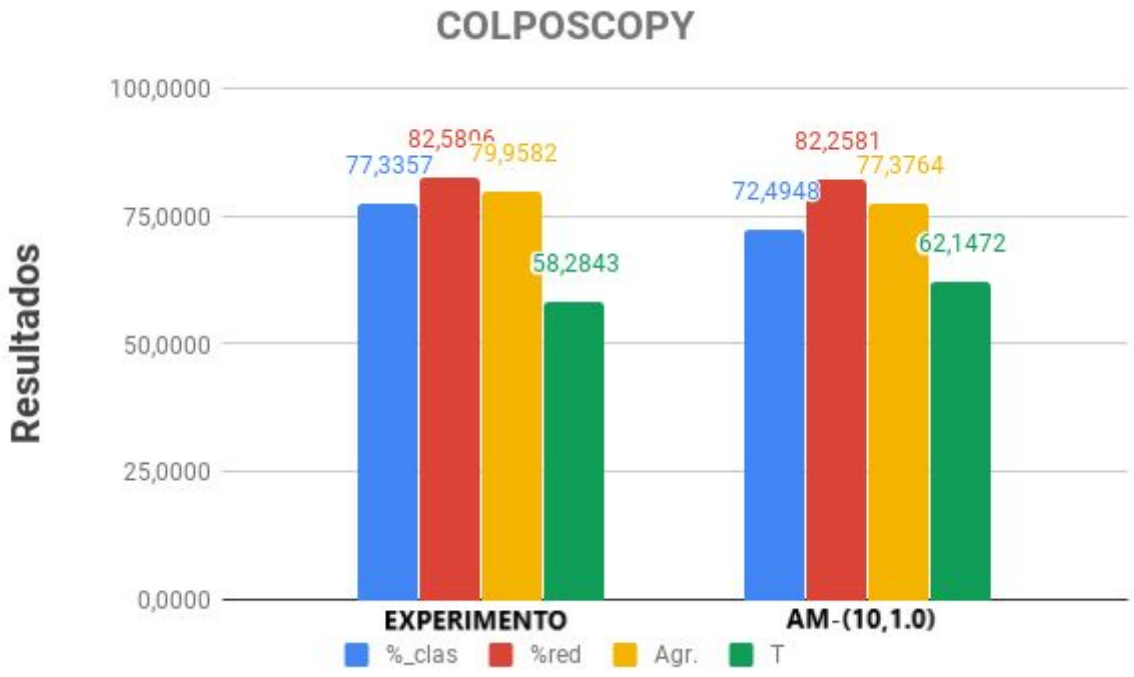
algoritmos. Si bien es cierto que el Relief obtiene una gran tasa de acierto, ni de lejos es suficiente para compararse con los demás.

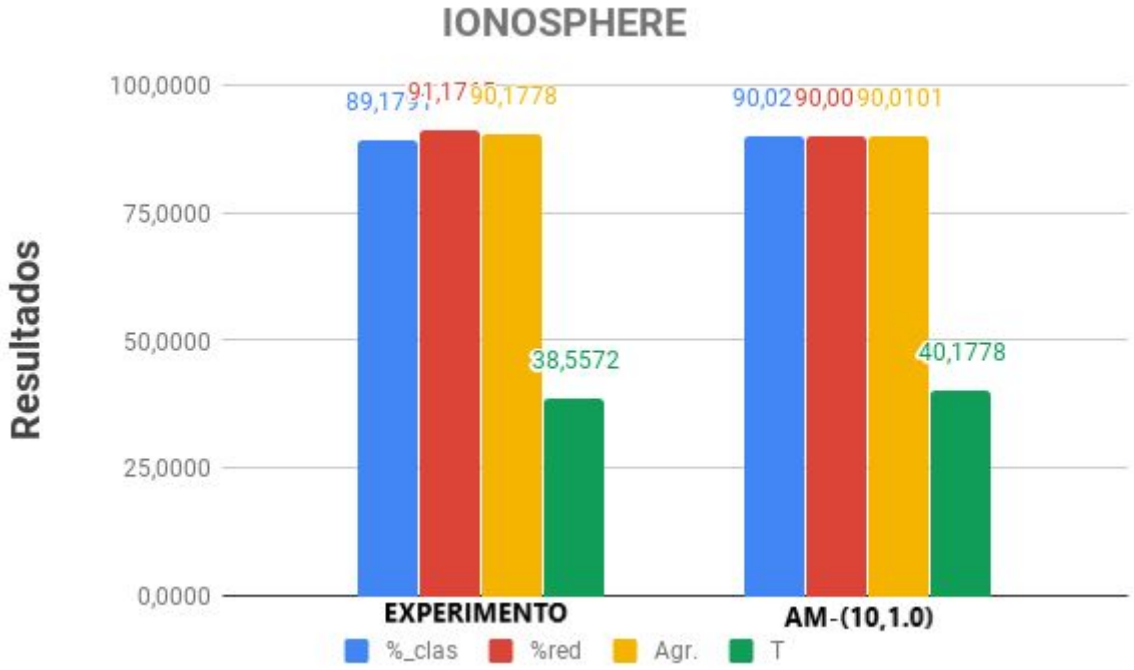
CONCLUSIÓN GENERAL

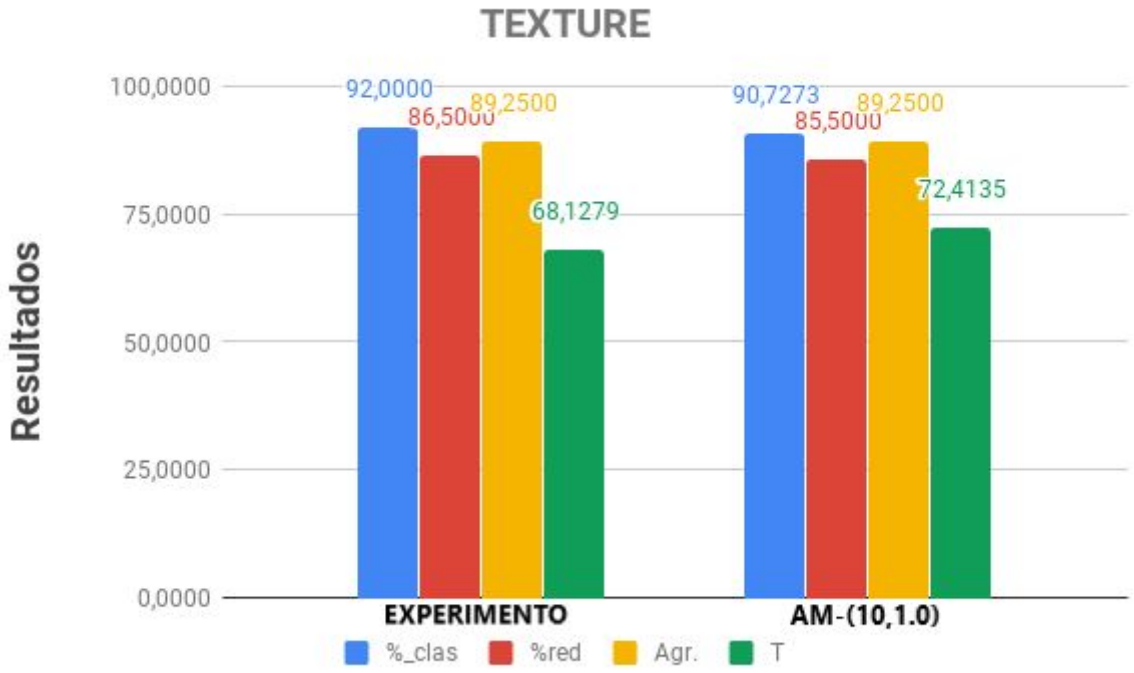
Los Algoritmos Meméticos son una combinación extraordinaria, pues combina un Algoritmo Genético, el cual ya obtiene buen rendimiento de por sí, con la Búsqueda Local que es el algoritmo con mejor rendimiento que conocíamos hasta la fecha de empezar ésta práctica. Tal y como reflejan los resultados, el 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

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.