Proyecto Final

INTELIGENCIA COMPUTACIONAL

Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Características

Julián Garrido Arana
alu0101759401@ull.edu.es

Curso Académico: 2024-2025

19 de abril de 2025

Índice

1. Descripción del Problema APC

El problema del Aprendizaje de Pesos en Características (APC) [2] consiste en optimizar el rendimiento de un clasificador a través de variar y evaluar la importancia de cada una de las caráterísticas de los datos en un dataset.

1.1. Introducción al problema de clasificación

Supongamos un conjunto de datos o *dataset* dividido en distintos subgrupos o *clases*. En dicho dataset, cada muestra individual o *sample* posee una serie de atributos o *features* que lo caracterizan. Entre esos atributos, uno de ellos establecido como etiqueta o *label* es el que identifica al sample con una de las clases del conjunto. El objetivo del problema de clasificación consiste en diseñar un sistema (clasificador) para poder predecir con la mayor precisión posible la clase de un elemento perteneciente al dataset o de un dato con la misma estructura.

Para abordar este problema planteado, vamos a utilizar un enfoque de aprendizaje supervisado ya que se conocen las clases existentes y se conoce la clase concreta a la que pertenece cada sample del dataset. En este ámbito, existen numerosas técnicas para clasificar los objetos de manera automática y en este proyecto vamos a centrarnos en el clasificador 1-NN [1]. Este clasificador utiliza el criterio del vecino más cercano, es decir, a un nuevo ejemplo se le asignará el label del elemento del dataset más cercano (similar en cuanto a features).

1.2. Funcionamiento del Aprendizaje de Pesos en Características

Para determinar la similaridad (distancia) entre dos elementos, el APC asigna a cada característica de los datos una importacia o *peso*, normalmente un valor real entre 0 y 1, de manera que la tarea de clasificación se ve sesgada por dicha importancia. Entonces, aplicando APC, un clasificador 1-NN buscará el vecino más cercano valorando más algunos atributos concretos. Este funcionamiento permite al clasificador identificar las características más influyentes e incluso descartar algunos atributos irrelevantes para la clasificación.

Dependiendo del algoritmo que se utilice para aprender los pesos, se pueden obtener distintos clasificadores. Así pues, el problema del APC planteado consite en evaluar y comparar varios algoritmos de aprendizaje para elegir la *mejor* ponderación de las características, tratando de optimizar cada calsificador.

En este caso, para decidir que pesos son *mejores*, el APC se centrará en maximizar una función de evalauación basada en dos criterios: optimizar la **precisión** (***tasa_class***) y la **complejidad** del clasificador (***tasa_red***).

Los algoritmos que se compararán en esta proyecto para la búsqueda de dichos pesos seran algoritmos genéticos y meméticos basados en heurísticas poblacionales.

2. Estructura Base del Proyecto

El desarrollo de este proyecto consiste en programar 7 clasificadores 1-NN para encontrar soluciones al problema APC. Por un lado, 4 de estos clasificadores se basarán en *Algoritmos Genéticos* y los otros 3 en *Algoritmos Meméticos*. Finalmente los evaluaremos en 3 conjuntos de datos diferentes y los compararemos entre sí para la obtención de conclusiones.

2.1. Conjuntos de Datos

Para la obtención de resultados vamos a evaluar los clasificadores en 3 datasets diferentes. De esta manera, el criterio de comparación no se verá tan determinado por las posibles particularadades de cada conjunto de datos y podremos realizar un análisis un poco más imparcial.

Cada dataset viene especificado en **5 ficheros de texto con formato .arff**. Cada fichero contiene dos secciones: una cabecera de la que se extraen los atributos, su tipo y las clases del dataset y un cuerpo donde cada fila es un sample del dataset. Cada conjunto se divide en 5 ficheros para aplicar de manera preestablecida **5-fold cross validation**.

Los datasets escogidos y sus características son:

- **Ecoli:** 366 ejemplos, 8 atributos (clase incluida) y 8 clases. Remarcar que es un dataset desbalanceado. (algunas clases son minoritarias)
- **Parkinsons:** 195 ejemplos, 23 atributos (clase incluida) y 2 clases. Remarcar que este dataset está desbalanceado.
- **Breast-Cancer:** 569 ejemplos, 31 atributos (clase incluida) y 2 clases.

2.2. Esquema de representación de las soluciones

En este problema de APC una solución se representa como un vector (W) n-dimensional donde n es el número de features del dataset evaluado y los valores del vector son números reales entre 0 y 1.

Remarcar que un cuando el valor de una componente es menor que 0.1, no se tiene en cuenta dicha caráterística en la solución, es decir, se interpreta como un peso igual a cero.

$$W = (w_1, w_2, \ldots, w_n), \text{ donde } w_i \in [0, 1] \tag{1}$$

2.3. Esquema de representación de los datos

Para trabajar con los datos de manera cómoda he implementado una estructura de datos (*DataInstance*) que me permite almacenar los features de cada sample y su clase en un solo elemento:

Pseudocódigo 1 Estructura de datos DataInstance

```
Struct DataInstance:
    features : vector de números reales
    classLabel : string
```

Una vez hecho esto, para almacenar el conjunto de datos completo y a su vez tenerlo dividido en 5 subconjuntos de cara a la validación, utilizo un vector de vectores de elementos tipo DataInstance.

$$datos = (data_1, data_2, \ldots, data_n), \text{ donde } n = nfolds \tag{2}$$

$$datos[i] = data_i = (DataInstance_1, Datainstance_2, ..., DataInstance_k) \tag{3}$$

2.4. Programa principal (main)

Mi programa principal se encarga de leer los argumentos de entrada haciendo las comprobaciones pertinentes, leer los datos de los ficheros y de llamar al clasificador correspondiente según los argumentos.

Sigue la siguiente estructura:

1. Verificar nº de argumentos ($3 \leq nArgs \leq 4$).
2. Leer nombre del dataset (argumento 1).
3. Leer el tipo de algoritmo de búsqueda (argumento 2).
4. Leer la semilla para la generación aleatoria (argumento 3, opcional).
5. Determinar los nombres de los ficheros ARFF correspondientes a leer (*getArffFilenames(dataset)*).
6. Leer los datos de los ficheros (Instanciar ARFFParser).
7. Normalizar los datos leídos (*ARFFParser::normalizeData()*).
8. Imprimir los datos normalizados (opcional).
9. Crear una cabecera de texto para el output de los resultados.
10. Instanciar el modelo del clasificador correspondiente y completar su cabecera.
11. Validar el clasificador (Llamada a *Classifier::kFoldCrossValidation()*).

2.5. Clase ARFFParser

Para leer los datos de los ficheros e introducirlos en mi programa implemento la clase ***ARFFParser***. Esta clase se encarga de almacenar el nombre del dataset proporcionado y de procesar los datos de los ficheros dándoles el formato adecuado para los algoritmos de clasificación.Implementa los siguientes métodos:

- ***parse() :*** Se encarga de leer cada uno de los archivos .arff del dataset recogiendo en un vector todos samples que contengan.
- ***getDataInSets() :*** Devuelve los datos almacenados.
- ***normalizeData() :*** **Normaliza los datos** para que todos los features tomen valores en [**0-1**]. Crucial para **no priorizar unos atributos sobre otros** en la clasificación.
- ***printDaata() :*** Imprime los datos almacenados.

2.6. Clase Classifier (Clasificador 1-NN)

Para implementar el algoritmo de **clasificación 1-NN** y realizar la la **validación cruzada** 5-fold cross validation utilizo la clase ***Classifier***. Como estas dos técnicas son comunes a todos los clasificadores a desarrollar independientemente del algoritmo de aprendizaje que empleen, esta clase es una **clase padre para el resto de clasificadores**. Así pues, almacena toda la informacion útil y necesaria para un clasificador además de sus resultados de rendimiento obtenidos.

Esta clase **no cuenta con un algoritmo de aprendizaje concreto** sino que implementa por defecto un clasificador 1-NN corriente, es decir, no hay APC y todos **los pesos se establecen a 1.0 por defecto**.

Los métodos que implementa son:

- ***kFoldCrossValidation()***: Realiza la validación cruzada k-fold sobre el conjunto de datos para evaluar el rendimiento del modelo e imprime los resultados obtenidos.

- ***train(índice_fold)***: Es un método **virtual** para que cada clase hija implemente su propio algoritmo de aprendizaje en cada fold. Entrena el modelo para ajustar los pesos y los devuelve junto con su *fitness*. Por defecto, para dar lugar al Clasificador 1-NN, devuelve un vector entero de unos (1.0) y el fitness asociado.
- ***classify(DataInstance, pesos)***: Clasifica una instancia de datos (vector de features) y devuelve su clase predicha (string) aplicando la estrategia de **Nearest Neighbour** y la **distancia euclidea ponderada**.
- ***calculateClassRate(pesos)***: Calcula la **precisión del modelo (tasa de clasificación)**, es decir, el porcentaje de instancias correctamente clasificadas respecto al total de instancias del testSet para unos pesos dados.
- ***calculateReductionRate(pesos)***: Calcula la **tasa de reducción**, es decir, el porcentaje de características no consideradas (peso cercano a cero) por el modelo, indicando la simplicidad del clasificador para unos pesos dados.
- ***funcionObjetivo(pesos, índice_fold)***: Calcula el fitness del clasificador en cada fold para unos pesos dados llamando a *calculateClassRate* y a *calculateReductionRate* y actualiza los datos de rendimiento correspondientes. Vease sección 2.7.
- ***printResults()***: Imprime el rendimiento de un clasificador para cada fold y también las medias finales. Incluyendo las tasas de clasificación y reducción, el fitness y el tiempo de ejecución.
- ***resultsToCSV()***: Extrae los mismos resultados del modelo que printResults en un fichero csv.

Pseudocódigo 2 Calcular Tasa de Clasificación

```
1:  function CALCULATECLASSRATE(weights)
2:      numCorrect ← 0.0
3:      numTestInstances ← tamaño de testSet
4:      for i ← 0 to numTestInstances − 1 do
5:          if CLASSIFY(testSet[i], weights) == testSet[i].classLabel then
6:              numCorrect ← numCorrect + 1
7:          end if
8:      end for
9:      return (100.0 × numCorrect / numTestInstances)
10: end function
```

Pseudocódigo 3 Clasificar una Instancia

```
1:  function CLASSIFY(instance, weights)
2:      minDistance ← ∞
3:      nearestNeighborClass ← cadena vacía
4:      for cada trainingInstance en trainingSet do
5:          distance ← WEIGHTEDEUCLIDEANDISTANCE(instance.features, trainingInstance.features, weights)
6:          if distance < minDistance then
7:              minDistance ← distance
8:              nearestNeighborClass ← trainingInstance.classLabel
9:          end if
10:     end for
11:     return nearestNeighborClass
12: end function
```

Pseudocódigo 4 Calcular Tasa de Reducción

```
1: function calculateReductionRate(weights)
2:     numReductions ← 0.0
3:     for i ← 0 to numFeatures − 1 do
4:         if weights[i] < 0.1 then
5:             numReductions ← numReductions + 1
6:         end if
7:     end for
8:     return (100.0 × numReductions / numFeatures)
9: end function
```

2.7. Función objetivo (Fitness)

Para evaluar el rendimiento del clasificador se aplica la siguiente fórmula:

$$\text{fitness}(pesos) = (\text{ALPHA} \times \text{tasa\_class}(pesos) + (1.0 - \text{ALPHA}) \times \text{tasa\_red}(pesos)) \tag{4}$$

Esta fórmula fomenta la precisión y la simplicidad del clasificador. Se corresponde con la función objetivo puesto que el propósito del clasificador es maximizarla.

ALPHA se corresponde con un valor real de 0.75 y tasa_class y tasa_red se calculan con los métodos calculateClassRate(pesos) y calculateReductionRate(pesos) respectivamente.

Pseudocódigo 5 Función Objetivo

```
1: function funcionObjetivo(weights, fold)
2:     tasa_class[fold] ← calculateClassRate(weights)      ▷ Actualizar la tasa de clasificación
3:     tasa_red[fold] ← calculateReductionRate(weights)    ▷ Actualizar la tasa de reducción
4:     return (ALPHA × tasa_class[fold] + (1.0 − ALPHA) × tasa_red[fold])
5: end function
```

2.8. Funciones adicionales

Aparte de los métodos vistos, utilizo un par de funciones declaradas en el archivo *Classifier.h* que realizan calculos comunes en distintos archivos y clases:

- ***euclideanDistance(vector1, vector2)*** : Devuelve la distancia euclidea entre dos vectores.
- ***mean(vector)*** : Devuelve la media de las componentes de un vector.
- ***weightedEuclideanDistance(vector1, vector2, pesos)***: Calcula la distancia euclidea ponderada entre dos vectores usando el vector de pesos de los parámetros.

3. Algoritmos Evolutivos escogidos y Esquema común de implementación

En este proyecto se abordará la optimización del problema mediante el uso de algoritmos evolutivos, específicamente algoritmos genéticos y meméticos. Estas técnicas están inspiradas en procesos naturales de evolución y aprendizaje, y son especialmente adecuadas para explorar espacios de búsqueda complejos y no lineales. El algoritmo genético se utilizará para generar y evolucionar soluciones candidatas, mientras que el algoritmo memético incorporará estrategias de mejora local para refinar dichas soluciones, combinando así exploración global con explotación local de forma eficiente.

3.1. Clase RandomToolsClassifier

Para implementar todo el código relativo a la generación aleatoria de números en los clasificadores diseñados y encapsular el código común de los métodos de búsqueda de soluciones, la evaluación de estos y sus criterios de parada, he utilizado la ***clase RandomToolsClassifier***.

Esta clase hereda de Classifier y básicamente define los atributos necesarios para generar soluciones aleatorias, soluciones iniciales, mutar soluciones y evaluar soluciones en el conjunto de entrenamiento.

De esta manera, ahora para crear un clasificador basado en un AG o el de búsqueda local mismamente, **las clases correspondientes deberán heredar de RandomToolsClassifier**.

Esta clase almacena los estadísticos comunes a todos los clasificadores como la media, la varianza y la desviación estandar y también el número máximo de evaluaciones de la función objetivo (en el training-set) que puede realizar un clasificador (***MAX_EVALUATIONS = 15.000***). Además, utiliza un atributo para contabilizar el número de llamadas a dicha función para que se actualice de forma automática y no tengamos que incrementar el contador en cada parte del código que hagamos una evaluación.

Los métodos que implementa son:

- ***generaSolucionInicial()***: Genera una solución inicial aleatoria para los pesos del modelo. Devuelve un vector de pesos donde cada peso es un valor aleatorio entre 0.0 y 1.0.
- ***calculateClassRateInTrain(weights)***: Calcula la tasa de clasificación del modelo usando la técnica de validación cruzada leave-one-out en el conjunto de entrenamiento. Evalúa cuántas instancias del conjunto de entrenamiento son correctamente clasificadas por el modelo.
- ***funcionObjetivoLocal(weights)***: Calcula el valor de la función objetivo del modelo, combinando la tasa de clasificación y la tasa de reducción sobre el propio conjunto de entrenamiento. Actualiza el número de evaluaciones realizadas.
- ***mutacionMovNormal(weights)***: Realiza una mutación sobre un vector de pesos seleccionando un peso al azar y ajustándolo usando una distribución normal. Se asegura que los pesos mutados se mantengan dentro del rango [0, 1].

Pseudocódigo 6 mutacionMovNormal

```
1: function MUTACIONMOVNORMAL(weights)
2:     normalDistribution ← NORMAL_DISTRIBUTION(MEDIA, DESVIACION_ESTANDAR)
3:     index ← RANDOM::GET(0, numFeatures − 1)
4:     weights[index] ← weights[index]+ RANDOM::GET(normalDistribution)
5:     if weights[index] < 0.0 then
6:         weights[index] ← 0.0
7:     else if weights[index] > 1.0 then
8:         weights[index] ← 1.0
9:     end if
10: end function
```

Pseudocódigo 7 generaSolucionInicial

```
1: function generaSolucionInicial
2:     weights ← vector(numFeatures)
3:     for i ← 0 to numFeatures − 1 do
4:         weights[i] ← Random::get(0.0, 1.0)
5:     end for
6:     return weights
7: end function
```

Pseudocódigo 8 funcionObjetivoLocal

```
1: function funcionObjetivoLocal(weights)
2:     evaluationsDone ← evaluationsDone + 1
3:     return (ALPHA × calculateClassRateInTrain(weights) + (1.0 − ALPHA) ×
   calculateReductionRate(weights))
4: end function
```

Pseudocódigo 9 calculateClassRateInTrain - Algoritmo **LEAVE-ONE-OUT**

```
1: function calculateClassRateInTrain(weights)
2:     numCorrects ← 0.0
3:     numTrainingInstances ← size(trainingSet)
4:     for i ← 0 to numTrainingInstances − 1 do
5:         minDistance ← numeric_limits::max(double)
6:         nearestNeighborClass ←
7:         for j ← 0 to numTrainingInstances − 1 do
8:             if i ≠ j then
9:                 distance ← weightedEuclideanDistance(trainingSet[i].features, trainingSet[j].features, weights)
10:                if distance < minDistance and distance > 0.0 then
11:                    minDistance ← distance
12:                    nearestNeighborClass ← trainingSet[j].classLabel
13:                end if
14:            end if
15:        end for
16:        if nearestNeighborClass == trainingSet[i].classLabel then
17:            numCorrects ← numCorrects + 1
18:        end if
19:    end for
20:    return (100.0×numCorrects / numTrainingInstances)
21: end function
```

3.2. Clase GeneticClassifier

Para implementar todo el código común y relativo a los algoritmos genéticos utilizo la clase ***GeneticClassifier***. Esta clase **hereda de RandomToolsClassifier** y es la **clase padre tanto de AGG como de AGE**, por lo que encapsula el esquema de evolución común a ambos y almacena las carácterísticas relevantes como el **tamaño de la población** o la **probabilidad de mutación**. Además, esta clase también almacena el **fitness asociado a cada individuo** de la población en un vector para poder realizar consultas en lugar de calcular reiteradamente este valor. También almacena el mejor fitness (***bestFitness***) y el índice del individuo asociado (***bestIndividual***). Esto hace más eficiente el esquema de reemplazo en cada algoritmo genético.

Los métodos que implementa son:

- ***GeneticClassifier(datos, type, blx_alpha)***: Constructor de la clase GeneticClassifier. Inicializa los parámetros del algoritmo genético mencionados. También establece si se usará el cruce BLX con el parámetro *blx_alpha*.
- ***train(fold)***: Entrena el modelo usando un algoritmo genético (llama al método AG) y devuelve los mejores pesos y su fitness asociado para un *fold* específico.
- ***AG(int fold)***: Implementa el esquema de evolución general para un algoritmo genético (AGG y AGE). Calcula el fitness real en el conjunto de prueba (*test set*) y devuelve los mejores pesos y su fitness para un *fold*.
- ***void evaluarPoblacion(population)***: Evalúa la población calculando y actualizando el fitness de cada individuo usando el método *funcionObjetivoLocal*. También actualiza el mejor individuo de la población mediante una llamada al método *mejorIndividuo*.
- ***void mejorIndividuo()***: Encuentra el individuo con el mejor fitness en la población actual y actualiza los atributos *bestIndividual* y *bestFitness*.
- ***seleccionTorneo(population)***: Realiza la selección por torneo para elegir el mejor individuo entre tres seleccionados al azar. Devuelve el individuo con el mejor fitness de los tres.
- ***void generarPadres(numPadres, population, sucesores)***: Según el parámetro *numPadres*, genera los padres para la siguiente generación llamando al método de selección por torneo. Los padres seleccionados se almacenan en el vector *sucesores*.
- ***cruce(sucesores)***: Realiza el cruce entre los padres en *sucesores* actualizandolos. Dependiendo del valor del parámetro *blx*, llama al cruce BLX (*cruceBLX*) o al cruce aritmético (*cruceAritmetico*).
- ***void cruceBLX(padre1, padre2)***: Realiza el cruce BLX para dos individuos y actualizandolos (vectores por referencia).
- ***void cruceAritmetico(padre1, padre2)***: Realiza el cruce aritmético para dos individuos y actualizandolos (vectores por referencia).

Pseudocódigo 10 AG

```
1:  function AG(fold)
2:      evaluationsDone ← 0                          ▷ Restablecer el número de evaluaciones
3:      population ← VECTOR(TAM_POBLACION)           ▷ Generar la población inicial
4:      for i ← 0 to TAM_POBLACION − 1 do
5:          population[i] ← GENERASOLUCIONINICIAL
6:      end for
7:      EVALUARPOBLACION(population)                 ▷ Evaluar la población inicial
8:      sucesores ← VECTOR(numPadres)                ▷ Crear el vector de sucesores
9:      while evaluationsDone < MAX_EVALUATIONS do
10:         GENERARPADRES(numPadres, population, sucesores)   ▷ Generación de los padres
11:         CRUCE(sucesores)                         ▷ Cruce de los padres
12:         MUTACION(sucesores)                      ▷ Mutación para generar los hijos
13:         REEMPLAZO(population, sucesores)         ▷ Reemplazo de la población
14:     end while
15:     bestWeights ← population[bestIndividual]     ▷ Calcular los mejores pesos
16:     realFitness ← FUNCIONOBJETIVO(bestWeights, fold)   ▷ Calcular real fitness (en test-set)
17:     return MAKE_TUPLE(bestWeights, realFitness)
18: end function
```

** No se realiza la evaluación de la población explicitamente en cada iteración del bucle porque se realiza dentro de REEMPLAZO para cada tipo de algoritmo genético.

Pseudocódigo 11 generarPadres

```
1: function generarPadres(numPadres, population, sucesores)
2:     for i ← 0 to numPadres − 1 do
3:         sucesores[i] ← seleccionTorneo(population)
4:     end for
5: end function
```

Pseudocódigo 12 seleccionTorneo

```
1: function seleccionTorneo(population)
2:     bestFit ← __DBL_MIN__
3:     index ← 0
4:     bestIndex ← 0
5:     fitness ← 0.0
6:     for i ← 0 to 3 do
7:         index ← Random::get(0, TAM_POBLACION - 1)
8:         fitness ← fitnesses[index]
9:         if fitness > bestFit then
10:            bestFit ← fitness
11:            bestIndex ← index
12:        end if
13:    end for
14:    return population[bestIndex]
15: end function
```

Pseudocódigo 13 cruce

```
1: function cruce(sucesores)
2:     for i ← 0 to numEsperadoCruces − 1 do
3:         if blx then
4:             cruceBLX(sucesores[i ∗ 2], sucesores[i ∗ 2 + 1])
5:         else
6:             cruceAritmetico(sucesores[i ∗ 2], sucesores[i ∗ 2 + 1])
7:         end if
8:     end for
9: end function
```

Pseudocódigo 14 cruceBLX

```
1: function CRUCEBLX(padre1, padre2)
2:     for i ← 0 to numFeatures − 1 do
3:         maxGen ← máx(padre1[i], padre2[i])
4:         minGen ← mín(padre1[i], padre2[i])
5:         rango ← maxGen − minGen
6:         rangoCruce ← 0.3 × rango
7:         cruce1 ← RANDOM::GET(minGen − rangoCruce, maxGen + rangoCruce)
8:         cruce2 ← RANDOM::GET(minGen − rangoCruce, maxGen + rangoCruce)
9:         cruce1 ← máx(0.0, mín(1.0, cruce1))
10:        cruce2 ← máx(0.0, mín(1.0, cruce2))
11:        padre1[i] ← cruce1
12:        padre2[i] ← cruce2
13:    end for
14: end function
```

Pseudocódigo 15 cruceAritmetico

```
1: function CRUCEARITMETICO(padre1, padre2)
2:     alpha ← RANDOM::GET(0.0, 1.0)
3:     for i ← 0 to numFeatures − 1 do
4:         aux ← padre1[i]
5:         padre1[i] ← alpha × aux + (1 − alpha) × padre2[i]
6:         padre2[i] ← alpha × padre2[i] + (1 − alpha) × aux
7:     end for
8: end function
```

4. Algoritmos Genéticos

Son métodos de **búsqueda basados en poblaciones** que dependen de la generación y selección de valores aleatorios [3].

El objetivo de estos algoritmos es generar un conjunto de soluciones (vectores de pesos) a partir de una población inicial de soluciones aleatorias. Cada solución se evalúa utilizando una función de aptitud, que en este caso se corresponde con funcionObjetivoLocal.

Para implementar ambas versiones de estos algoritmos, he seguido el esquema de evolución general mostrado en la Figura 1 implementado en la clase GeneticClassifier en el método AG descrito en la sección anterior.

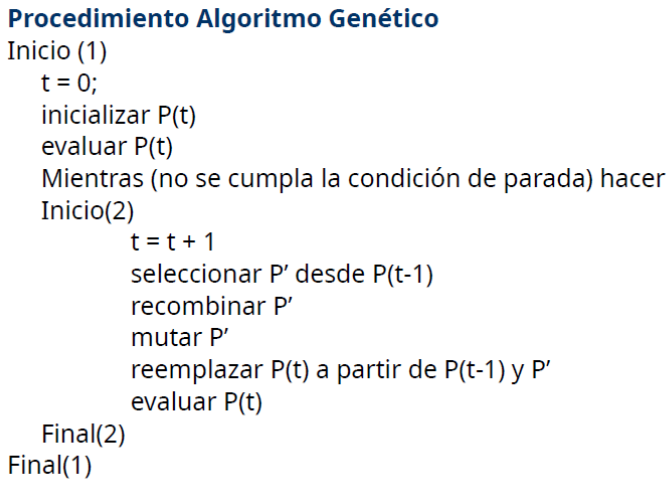

Figura 1: Esquema general del proceso evolutivo con algoritmo genético.

4.1. Algoritmo Genético Generacional (AGG)

Este método considera tantos padres como individuos haya en la población y renueva toda la pablación con elitismo, es decir, sustituye a todos salvo al mejor individuo en caso de que no haya mejora [4].

Para implementar un clasificador que aplique este algoritmo he creado la **clase *AGGClassifier*** que **hereda de la clase GeneticClasifier**. La única novedad que incluye esta clase es la **sobreescritura de los métodos mutacion y reemplazo**. Vease en el pseudocódigo. Además, esta clase define una probabilidad de cruce igual a 0.68 y establece el número esperado de cruces según la fórmula: round(PROB_CRUCE * numPadres/2). También establece el número de padres a 50 y el número esperado de mutaciones según la fórmula: PROB_MUT * TAM_POBLACION.

Pseudocódigo 16 Mutación en AGG

```
1: function MUTACION(sucesores)
2:     for i ← 0 to numEsperadoMutaciones − 1 do
3:         index ← RANDOM::GET(0, TAM_POBLACION - 1)
4:         MUTACIONMOVNORMAL(sucesores[index])    ▷ Aplicar mutación en el sucesor con índice index
5:     end for
6: end function
```

Pseudocódigo 17 Reemplazo en AGG

```
1: function REEMPLAZO(population, sucesores)
2:     bestOldFitness ← bestFitness ▷ Guardar el mejor fitness de la generación anterior
3:     bestOldIndex ← bestIndividual ▷ Guardar el índice del mejor individuo de la generación anterior
4:     EVALUARPOBLACION(sucesores) ▷ Evaluar la población de sucesores
5:     if bestFitness < bestOldFitness then
6:         worstIndividual1 ← DISTANCE(fitnesses.begin(), min_element(fitnesses.begin(), fitnesses.end()))
       ▷ Encontrar el peor individuo en los sucesores
7:         sucesores[worstIndividual1] ← population[bestOldIndex] ▷ Reemplazar el peor sucesor con el
       mejor de la generación anterior
8:         bestFitness ← bestOldFitness ▷ Actualizar el mejor fitness
9:         bestIndividual ← worstIndividual1 ▷ Actualizar el índice del mejor individuo
10:        fitnesses[worstIndividual1] ← bestFitness ▷ Actualizar el fitness del mejor individuo
11:    end if
12:    population ← MOVE(sucesores) ▷ Reemplazar la población por los sucesores
13:    RESIZE(sucesores, TAM_POBLACION, vector¡double¿(numFeatures)) ▷ Redimensionar el vector de
       sucesores
14: end function
```

4.2. Algoritmo Genético Estacionario (AGE)

Este método considera solo 2 padres y renueva como mucho 2 individuos de la población, es decir, sustituye los peores individuos por los 2 hijos generados a partir de los cruces y mutaciones de los padres si el fitness obtenido es mejor [4].

Para implementar un clasificador que aplique este algoritmo he creado la **clase *AGEClassifier*** que **hereda de la clase GeneticClasifier**. La única novedad que incluye esta clase es la **sobreescritura de los métodos mutacion y reemplazo**. Vease en el pseudocódigo. Además, esta clase define una probabilidad de cruce igual a 1.0 para que siempre se crucen los padres y establece el número esperado de cruces según la fórmula: round(PROB_CRUCE * numPadres/2). También establece el número de padres a 2.

Pseudocódigo 18 Mutación AGE

```
1: function MUTACION(sucesores)
2:     for i ← 0 to numPadres − 1 do
3:         muta ← RANDOM::GET(0.0, 1.0)
4:         if muta < PROB_MUT then
5:             MUTACIONMOVNORMAL(sucesores[i]) ▷ Mutación para el sucesor i
6:         end if
7:     end for
8: end function
```

Pseudocódigo 19 Reemplazo AGE

```
1: function REEMPLAZO(population, sucesores)
2:     fitness_h1 ← FUNCIONOBJETIVOLOCAL(sucesores[0])                ▷ Evaluar el primer sucesor
3:     fitness_h2 ← FUNCIONOBJETIVOLOCAL(sucesores[1])                ▷ Evaluar el segundo sucesor
4:     worstIndividual1 ← DISTANCE(fitnesses.begin(), min_element(fitnesses.begin(), fitnesses.end()))  ▷
   Encontrar el peor individuo
5:     worstIndividual2 ← −1
6:     worstFitness2 ← 101.00                          ▷ Inicialización con un valor suficientemente grande
7:     for i ← 0 to TAM_POBLACION − 1 do
8:         if i ≠ worstIndividual1 and fitnesses[i] < worstFitness2 then
9:             worstIndividual2 ← i
10:            worstFitness2 ← fitnesses[i]
11:        end if
12:    end for     ▷ Comparar los sucesores con los peores individuos de la población y reemplazarlos si es
   necesario
13:    if MIN(fitness_h1, fitness_h2) > fitnesses[worstIndividual2] then
14:        population[worstIndividual1] ← sucesores[0]
15:        fitnesses[worstIndividual1] ← fitness_h1
16:        population[worstIndividual2] ← sucesores[1]
17:        fitnesses[worstIndividual2] ← fitness_h2
18:        if MAX(fitness_h1, fitness_h2) ¿bestFitness then
19:            if fitness_h1 > fitness_h2 then
20:                bestFitness ← fitness_h1
21:                bestIndividual ← worstIndividual1
22:            else
23:                bestFitness ← fitness_h2
24:                bestIndividual ← worstIndividual2
25:            end if
26:        end if
27:    else if MAX(fitness_h1, fitness_h2) >fitnesses[worstIndividual1] then
28:        if fitness_h1 > fitness_h2 then
29:            population[worstIndividual1] ← sucesores[0]
30:            fitnesses[worstIndividual1] ← fitness_h1
31:            if fitness_h1 > bestFitness then
32:                bestFitness ← fitness_h1
33:                bestIndividual ← worstIndividual1
34:            end if
35:        else
36:            population[worstIndividual1] ← sucesores[1]
37:            fitnesses[worstIndividual1] ← fitness_h2
38:            if fitness_h2 > bestFitness then
39:                bestFitness ← fitness_h2
40:                bestIndividual ← worstIndividual1
41:            end if
42:        end if
43:    end if
44: end function
```

5. Algoritmos Meméticos

Son métodos de **búsqueda basados en AGG** que dependen de la generación y selección de valores aleatorios, pero incorporan además técnicas de optimización como por ejemplo búsqueda local [6] para refinar las soluciones generadas [5].

Para implementar las versiones de estos algoritmos, he creado la **clase *AMProbClssifier*** que **hereda de la clase AGGClasifier**. Esta clase encapsula todo el código común a los 3 algoritmos meméticos y establece el **esquema de funcionamiento en el método *AM_Prob***. También **sobreescribe el método train para llamar a AM_Prob en lugar de a AG**.

La idea de funcionamiento de esta clase es simplemente cambiar el conjunto de individuos al que se le aplica BL en función de una probabilidad. Por ejemplo, si la probabilidad es 1.0, se aplica BL sobre toda la población y si la porbabilidad es 0.1, sobre el 10 % de la población. Para ello el constructor de la clase recibe un parámetro: probNumIndividuos.

Implementa estos métodos:

- ***AM_Prob(int fold)***: Implementa un algoritmo memético probabilístico. Durante el proceso evolutivo, alterna entre la búsqueda local aplicada a un subconjunto de individuos y la evolución genética estándar. Devuelve los mejores pesos y su fitness asociado para un *fold* específico.
- ***busquedaLocalIndividuo(individuoBL, index_individuo, numIteraciones)***: Aplica búsqueda local a un individuo específico de la población con un numero límite de iteraciones pasado como parámetro. Actualiza el mejor individuo y su fitness en la población si se encuentra una mejor solución a la actual.
- ***mutacionMovNormalBL(peso_pos)***: Modifica un peso de característica con un valor aleatorio generado a partir de una distribución normal. Trunca el valor resultante al rango [0, 1]. Devuelve el peso modificado.

Pseudocódigo 20 mutacionMovNormalBL

```
1: function MUTACIONMOVNORMALBL(peso_pos)
2:     normalDistribution ← NORMAL_DISTRIBUTION<DOUBLE(MEDIA, DESVIACION_ESTANDAR)
3:     peso_pos ← peso_pos + RANDOM::GET(normalDistribution)        ▷ Modificar el peso con un valor aleatorio
4:     if peso_pos < 0.0 then
5:         return 0.0
6:     else if peso_pos > 1.0 then
7:         return 1.0
8:     else
9:         return peso_pos
10:    end if
11: end function
```

Pseudocódigo 21 AM_Prob

```
1: function AM_Prob(fold)
2:     evaluationsDone ← 0
3:     poblacion ← nueva lista de listas de longitud TAM_POBLACION
4:     for i ← 0 to TAM_POBLACION − 1 do
5:         poblacion[i] ← generaSolucionInicial
6:     end for
7:     evaluarPoblacion(poblacion)
8:     sucesores ← nueva lista de listas de longitud numPadres
9:     contador_generaciones ← 0
10:    numIndividuosBL ← round(probNumIndividuos × TAM_POBLACION)
11:    indices_poblacion ← nueva lista de longitud TAM_POBLACION
12:    for i ← 0 to TAM_POBLACION − 1 do
13:        indices_poblacion[i] ← i
14:    end for
15:    while evaluationsDone < MAX_EVALUATIONS do
16:        if contador_generaciones = numGeneraciones then
17:            Random::shuffle(indices_poblacion)
18:            for i ← 0 to numIndividuosBL − 1 do
19:                ind ← indices_poblacion[i]
20:                busquedaLocalIndividuo(poblacion[ind], ind, ITERS_BL)
21:            end for
22:            contador_generaciones ← 0
23:        else
24:            generarPadres(numPadres, poblacion, sucesores)
25:            cruce(sucesores)
26:            mutacion(sucesores)
27:            reemplazo(poblacion, sucesores)
28:            contador_generaciones ← contador_generaciones + 1
29:        end if
30:    end while
31:    bestWeights ← poblacion[bestIndividual]
32:    realFitness ← funcionObjetivo(bestWeights, fold)
33:    return < bestWeights, realFitness >
34: end function
```

Pseudocódigo 22 busquedaLocalIndividuo

```
1: function BUSQUEDALOCALINDIVIDUO(individuoBL, index_individuo, numIteraciones)
2:     consecutiveNeighbors ← 0
3:     maxConsecutiveNeighbors ← MAX_NEIGHBORS
4:     evaluationsBL ← 0
5:     maxEvalsGlobal ← MAX_EVALUATIONS
6:     bestFitBL ← fitnesses[index_individuo]                    ▷ Evaluar la solución inicial
7:     index ← vector¡int(numFeatures)
8:     for i ← 0 to numFeatures − 1 do
9:         index[i] ← i
10:    end for
11:    neighborWeights ← vector¡double()
12:    neighborFitness ← 0.0
13:    while evaluationsDone < maxEvalsGlobal and consecutiveNeighbors < maxConsecutiveNeighbors and evaluationsBL < numIteraciones do
14:        RANDOM::SHUFFLE(index)                                ▷ Generar una permutación aleatoria
15:        for i ← 0 to numFeatures − 1 do
16:            if consecutiveNeighbors < maxConsecutiveNeighbors and evaluationsDone < maxEvalsGlobal and evaluationsBL < numIteraciones then
17:                neighborWeights ← individuoBL
18:                neighborWeights[index[i]] ← MUTACIONMOVNORMALBL(individuoBL[index[i]])
19:                consecutiveNeighbors ← consecutiveNeighbors + 1
20:                neighborFitness ← FUNCIONOBJETIVOLOCAL(neighborWeights)    ▷ Evaluar la solución vecina
21:                evaluationsBL ← evaluationsBL + 1
22:                if neighborFitness > bestFitBL then
23:                    individuoBL ← neighborWeights
24:                    bestFitBL ← neighborFitness
25:                    consecutiveNeighbors ← 0
26:                    break
27:                end if
28:            end if
29:        end for
30:    end while
31:    if bestFitBL > bestFitness then
32:        bestFitness ← bestFitBL
33:        bestIndividual ← index_individuo
34:    end if
35: end function
```

5.1. AM-ALL y AM-RAND

Cada 10 generaciones, se aplica la BL sobre todos los cromosomas de la población o solo sobre un 10 % seleccionado de manera aleatoria.

Para implementar estos algoritmos simplemente creo un objeto AMProbClassifier con la probabilidad correspondiente, 1.0 o 0.1.

5.2. AM-BEST

Cada 10 generaciones, aplicar la BL sobre los 0.1·N mejores cromosomas de la población actual (N es el tamaño de ésta).

Para implementar un clasificador que aplique este algoritmo he creado la **clase *AMBestClassifier*** que

hereda de la clase AMProbClasifier. La única novedad que incluye esta clase es la **sobreescritura del método train (llama a AM_Best)** y la **implementación del método AM_Best** para definir el esquema de funcionamiento aplicando BL sobre los mejores individuos. AM_Best se apoya en la **función auxiliar *ordenarPorFitness***. Vease en el pseudocódigo.

Pseudocódigo 23 AM_Best

```
1: function AM_Best(fold)
2:     evaluationsDone ← 0
3:     poblacion ← nueva lista de listas de longitud TAM_POBLACION
4:     for i ← 0 to TAM_POBLACION − 1 do
5:         poblacion[i] ← generaSolucionInicial
6:     end for
7:     evaluarPoblacion(poblacion)
8:     sucesores ← nueva lista de listas de longitud numPadres
9:     contador_generaciones ← 0
10:    numIndividuosBL ← round(probNumIndividuos × TAM_POBLACION)
11:    while evaluationsDone < MAX_EVALUATIONS do
12:        if contador_generaciones = numGeneraciones then
13:            ordenarPorFitness(poblacion, fitnesses)
14:            sort(fitnesses.begin(), fitnesses.end(), [](doublea, doubleb){returna > b;})
15:            bestFitness ← fitnesses[0]
16:            bestIndividual ← 0
17:            for i ← 0 to numIndividuosBL − 1 do
18:                busquedaLocalIndividuo(poblacion[i], i, ITERS_BL)
19:            end for
20:            contador_generaciones ← 0
21:        else
22:            generarPadres(numPadres, poblacion, sucesores)
23:            cruce(sucesores)
24:            mutacion(sucesores)
25:            reemplazo(poblacion, sucesores)
26:            contador_generaciones ← contador_generaciones + 1
27:        end if
28:    end while
29:    bestWeights ← poblacion[bestIndividual]
30:    realFitness ← funcionObjetivo(bestWeights, fold)
31:    return make_tuple(bestWeights, realFitness)
32: end function
```

Pseudocódigo 24 ordenarPorFitness

```
1: function ORDENARPORFITNESS(population, fitnesses)
2:     combined ← nueva lista de pares < vector(double), double >
3:     tam ← SIZE(population)
4:     for i ← 0 to tam − 1 do
5:         COMBINED.PUSH_BACK(make_pair(population[i], fitnesses[i]))
6:     end for
7:     SORT(combined.begin(), combined.end(), [](constpair < vector < double >, double > &a,
       constpair < vector < double >, double > &b){return a.second > b.second; })
8:     sorted_vector_a ← nueva lista de vectores de doble precisión
9:     for const auto&pair : combined do
10:        SORTED_VECTOR_A.PUSH_BACK(pair.first)
11:    end for
12:    population ← MOVE(sorted_vector_a)
13: end function
```

6. Algoritmo de comparación (5-fold cross validation)

En nuestro problema, para poder validar y comparar los clasificadores con sus respectivos algoritmos de aprendizaje consideraremos el método de validación ***5-fold cross validation***. Para ello, cada conjunto de datos ha sido dividido en 5 particiones (*folds*) disjuntas al 20 % manteniendo la distribución de clases. Aprenderemos un clasificador utilizando hasta un total del 80 % de los datos disponibles de cada dataset (4 particiones de las 5) y validaremos con el 20 % restante (la partición restante). Finalmente, la calidad del método de clasificación se medirá con un único valor, correspondiente a la media de los 5 porcentajes de clasificación del conjunto de prueba. Vease imagen a continuación.

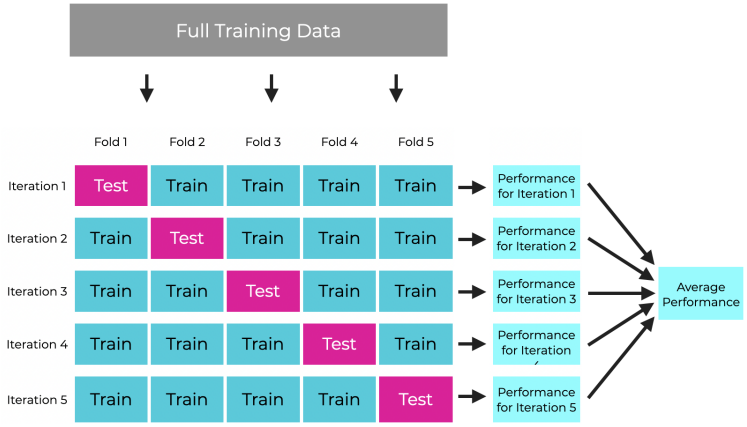

Figura 2: Funcionamiento de 5-fold cross validation

Para llevar a cabo esta técnica, he implementado el **método *kFoldCrossValidation()*** dentro de la **clase *Classifier***.

Pseudocódigo 25 Validación Cruzada k-Fold del Clasificador

```
1:  function kFoldCrossValidation
2:      for fold = 0 to k_Folds − 1 do
3:          iniciarTemporizador()
4:          testSet.clear()
5:          trainingSet.clear()
6:          testSet ← data[fold]
7:          for i = 0 to k_Folds − 1 do
8:              if i ≠ fold then
9:                  trainingSet.insert(trainingSet.end(), data[i].begin(), data[i].end())
10:             end if
11:         end for
12:         trained ← train(fold)
13:         trainedWeights[fold] ← getWeights(trained)
14:         featureWeights ← trainedWeights[fold]
15:         fitness[fold] ← getFitness(trained)
16:         detenerTemporizador()
17:         tiempo[fold] ← calcularDuracion()
18:     end for
19:     printResults
20: end function
```

* featureWeights son los pesos del clasificador en cada momento (atributo de instancia).

7. Procedimiento para el desarrollo y ejecución del proyecto:

Para el desarrollo de este proyecto he llevado a cabo una implementación desde cero aunque me he apoyado en consejos de *ChatGPT* para decidir la estructura de ficheros y clases del proyecto (como por ejemplo la clase ARFFParser). Para programar he utilizado el editor de código Visual Studio Code (VSCode) porque estoy familiarizado con su uso y como lenguaje de programación he escogido C++ porque es un lenguaje que manejo con soltura y por ser un lenguaje más apropiado para la optimización y reducir el tiempo de ejecución.

En cuanto a la estructura de ficheros y clases, se explica en las secciones anteriores el razonamiento y procedimiento seguidos. Vease también el Readme.

7.1. Manual de usuario

En esta sección se indica como compilar y ejecutar el programa (main) del proyecto.

La estructura del proyecto es la siguiente:

Para **compilar el programa** debemos abrir una terminal dentro del directorio FUENTES y ejecutar la orden make. Esta orden ejecuta el archivo Makefile de FUENTES generando los archivos objeto dentro de una carpeta builds en FUENTES y creando también el archivo ejecutable del proyecto (main) dentro del directorio BIN.

Para **ejecutar el programa** debemos abrir una terminal en el irectorio BIN y ejecutar la orden ./main con los argumentos correspondientes:

```
./main <nombre_dataset> <modelo> <semilla (opcional)>
```

Veamos los detalles sobre los argumentos.

- **nombre_dataset**: Nombre en minúsculas del dataset a utilizar. Las opciones son: ***ecoli***, ***parkinsons*** o ***breast-cancer***. Es **obligatorio**.
- **modelo**: Opción a modo de flag del tipo de clasificador a utilizar. Las opciones son: ***-1nn*** para indicar clasificador 1-NN, ***-gr*** para greedy, ***-bl*** para búsqueda local, ***-agg*** para algoritmo genético generacional con cruce aritmético, ***-agg_blx*** para algoritmo genético generacional con cruce blx, ***-age*** para algoritmo genético estacionario con cruce aritmético, ***-age_blx*** para algoritmo genético estacionario con cruce blx, ***-am_all*** para algoritmo memético aplicadondo bl en toda la población, ***-am_rand*** para algoritmo memético aplicando bl al 10 % de la población o ***-am_best*** para utilizar el algoritmo memético aplicando bl al 10 % mejor de la población,. Es **obligatorio**.
- **semilla**: Número entero a utilizar como seed. Es **opcional**. Si no se introduce semilla se establece por defecto a 12345.

Ejemplos válidos de ejecución:

- `./main ecoli -1NN`
- `./main parkinsons -agg 42`
- `./main breast-cancer -am_all`
- `./main ecoli -bl 42`

8. Experimentos y Análisis de resultados

De forma adicional, previo a la implementación de los algoritmos evolutivos descritos, se ha requerido implementar tanto el clasificador 1-NN simple (sin mejora evolutiva) así como otros algoritmos de búsqueda de soluciones como Búsqueda Local (BL) o un Algoritmo Greedy (Búsqueda Voraz) para abordar el problema del APC. Por tanto, mostramos también el rendimiento de estas estrategias de búsqueda de soluciones y procedemos a evaluar y comparar los clasificadores elaborados: Clasificador 1-NN simple, Clasificador 1-NN con algoritmo de aprendizaje Greedy, el Clasificador 1-NN con algoritmo de aprendizaje Búsqueda Local y los 7 Clasificadores 1-NN con algoritmos evolutivos integrados (4 genéticos y 3 meméticos).

8.1. Casos del problema

Para la obtención de resultados, hemos ejecutado cada clasificador en cada dataset utilizando de semilla 42 en las ejecuciones.

8.2. Resultados obtenidos

Para comparar el rendimiento de cada clasificador, recogemos los siguientes datos de cada uno:

- tasa de clasificación
- tasa de reducción
- fitness
- tiempo de ejecución

Mostrando en las tablas siguientes su valor tanto en cada fold como la media en las 5 ejecuciones.

Partición	**Ecoli**				**Breast-Cancer**				**Parkinsons**			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	80.00	0	60.00	0.00e+00	97.39	0	73.04	4.00e+00	97.50	0	73.12	0.00e+00
2	72.86	0	54.64	0.00e+00	96.52	0	72.39	3.00e+00	87.50	0	65.62	0.00e+00
3	82.35	0	61.76	0.00e+00	92.17	0	69.13	4.00e+00	97.50	0	73.12	0.00e+00
4	83.82	0	62.87	0.00e+00	98.26	0	73.70	3.00e+00	100.00	0	75.00	0.00e+00
5	85.00	0	63.75	0.00e+00	93.58	0	70.18	4.00e+00	91.43	0	68.57	0.00e+00
Media	80.81	0	60.61	0.00e+00	95.59	0	71.69	3.60e+00	94.79	0	71.09	0.00e+00

Cuadro 1: Resultados del **clasificador 1-NN**

Partición	**Ecoli**				**Breast-Cancer**				**Parkinsons**			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	77.14	28.57	65.00	2.00e+00	97.39	3.33	73.88	1.20e+01	97.50	0.00	73.12	1.00e+00
2	74.29	28.57	62.86	1.00e+00	97.39	3.33	73.88	1.30e+01	90.00	0.00	67.50	1.00e+00
3	79.41	28.57	66.70	1.00e+00	89.57	13.33	70.51	1.00e+01	95.00	0.00	71.25	1.00e+00
4	80.88	28.57	67.80	1.00e+00	97.39	0.00	73.04	1.00e+01	100.00	0.00	75.00	1.00e+00
5	86.67	28.57	72.14	1.00e+00	94.50	0.00	70.87	1.00e+01	91.43	0.00	68.57	1.00e+00
Media	79.68	28.57	66.90	1.20e+00	95.25	4.00	72.44	1.10e+01	94.79	0.00	71.09	1.00e+00

Cuadro 2: Resultados del **clasificador Greedy**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	71.43	57.14	67.86	8.29e+02	95.65	80.00	91.74	9.46e+03	90.00	72.73	85.68	9.49e+02
2	68.57	42.86	62.14	5.86e+02	95.65	83.33	92.57	1.10e+04	85.00	72.73	81.93	1.01e+03
3	66.18	71.43	67.49	5.52e+02	84.35	80.00	83.26	9.55e+03	95.00	77.27	90.57	9.38e+02
4	75.00	57.14	70.54	5.80e+02	93.91	90.00	92.93	1.32e+04	92.50	81.82	89.83	1.37e+03
5	85.00	57.14	78.04	5.56e+02	95.41	86.67	93.23	1.91e+04	94.29	81.82	91.17	8.24e+02
Media	73.24	57.14	69.21	6.21e+02	93.00	84.00	90.75	1.25e+04	91.36	77.27	87.84	1.02e+03

Cuadro 3: Resultados del **clasificador BL**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	75.71	57.14	71.07	2.95e+04	94.78	80.00	91.09	1.48e+05	100.00	68.18	92.05	1.52e+04
2	68.57	42.86	62.14	3.04e+04	92.17	86.67	90.80	1.52e+05	87.50	72.73	83.81	1.53e+04
3	66.18	71.43	67.49	3.08e+04	86.09	86.67	86.23	1.47e+05	95.00	81.82	91.70	1.50e+04
4	73.53	57.14	69.43	3.00e+04	92.17	86.67	90.80	1.46e+05	92.50	77.27	88.69	1.50e+04
5	86.67	42.86	75.71	3.24e+04	94.50	80.00	90.87	1.49e+05	88.57	77.27	85.75	1.59e+04
Media	74.13	54.29	69.17	3.06e+04	91.94	84.00	89.96	1.48e+05	92.71	75.45	88.40	1.53e+04

Cuadro 4: Resultados del **clasificador AGG-Aritmético**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	78.57	57.14	73.21	2.99e+04	95.65	80.00	91.74	1.44e+05	95.00	72.73	89.43	1.55e+04
2	74.29	42.86	66.43	3.00e+04	95.65	90.00	94.24	1.39e+05	95.00	77.27	90.57	1.48e+04
3	66.18	71.43	67.49	3.11e+04	87.83	90.00	88.37	1.43e+05	95.00	72.73	89.43	1.49e+04
4	79.41	57.14	73.84	3.00e+04	91.30	90.00	90.98	1.40e+05	92.50	81.82	89.83	1.46e+04
5	85.00	57.14	78.04	3.20e+04	92.66	83.33	90.33	1.46e+05	97.14	81.82	93.31	1.57e+04
Media	76.69	57.14	71.80	3.06e+04	92.62	86.67	91.13	1.43e+05	94.93	77.27	90.51	1.51e+04

Cuadro 5: Resultados del **clasificador AGG-BLX**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	75.71	57.14	71.07	2.90e+04	93.04	83.33	90.62	1.44e+05	97.50	72.73	91.31	1.50e+04
2	71.43	42.86	64.29	2.94e+04	93.91	86.67	92.10	1.45e+05	92.50	77.27	88.69	1.47e+04
3	77.94	57.14	72.74	2.94e+04	88.70	90.00	89.02	1.43e+05	97.50	77.27	92.44	1.47e+04
4	83.82	57.14	77.15	2.94e+04	94.78	83.33	91.92	1.43e+05	92.50	81.82	89.83	1.47e+04
5	85.00	57.14	78.04	3.11e+04	94.50	83.33	91.70	1.48e+05	97.14	81.82	93.31	1.57e+04
Media	78.78	54.29	72.66	2.97e+04	92.99	85.33	91.07	1.45e+05	95.43	78.18	91.12	1.50e+04

Cuadro 6: Resultados del **clasificador AGE-Aritmético**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	77.14	57.14	72.14	2.93e+04	93.91	83.33	91.27	1.42e+05	95.00	72.73	89.43	1.50e+04
2	68.57	57.14	65.71	2.93e+04	95.65	83.33	92.57	1.44e+05	87.50	77.27	84.94	1.47e+04
3	77.94	57.14	72.74	2.97e+04	90.43	86.67	89.49	1.40e+05	97.50	77.27	92.44	1.47e+04
4	72.06	71.43	71.90	3.04e+04	92.17	83.33	89.96	1.40e+05	95.00	81.82	91.70	1.46e+04
5	85.00	57.14	78.04	3.15e+04	94.50	83.33	91.70	1.44e+05	97.14	81.82	93.31	1.55e+04
Media	76.14	60.00	72.11	3.00e+04	93.33	84.00	91.00	1.42e+05	94.43	78.18	90.37	1.49e+04

Cuadro 7: Resultados del **clasificador AGE-BLX**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	74.29	57.14	70.00	2.97e+04	93.04	80.00	89.78	1.49e+05	100.00	77.27	94.32	1.51e+04
2	68.57	57.14	65.71	2.98e+04	95.65	90.00	94.24	1.43e+05	87.50	72.73	83.81	1.50e+04
3	77.94	57.14	72.74	3.00e+04	93.04	80.00	89.78	1.45e+05	92.50	77.27	88.69	1.50e+04
4	80.88	57.14	74.95	3.02e+04	92.17	86.67	90.80	1.43e+05	90.00	77.27	86.82	1.51e+04
5	85.00	57.14	78.04	3.15e+04	95.41	90.00	94.06	1.47e+05	97.14	81.82	93.31	1.57e+04
Media	77.34	57.14	72.29	3.02e+04	93.87	85.33	91.73	1.46e+05	93.43	77.27	89.39	1.52e+04

Cuadro 8: Resultados del **clasificador AM-ALL**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	74.29	57.14	70.00	2.95e+04	95.65	83.33	92.57	1.51e+05	90.00	77.27	86.82	1.46e+04
2	54.29	71.43	58.57	2.99e+04	98.26	83.33	94.53	1.43e+05	95.00	77.27	90.57	1.46e+04
3	77.94	57.14	72.74	3.00e+04	94.78	90.00	93.59	1.40e+05	97.50	77.27	92.44	1.49e+04
4	80.88	57.14	74.95	3.01e+04	93.04	90.00	92.28	1.39e+05	92.50	81.82	89.83	1.44e+04
5	71.67	71.43	71.61	3.18e+04	95.41	76.67	90.73	1.48e+05	91.43	68.18	85.62	1.60e+04
Media	71.81	62.86	69.57	3.03e+04	95.43	84.67	92.74	1.44e+05	93.29	76.36	89.06	1.49e+04

Cuadro 9: Resultados del **clasificador AM-RAND**

Partición	Ecoli				Breast-Cancer				Parkinsons			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1	75.71	57.14	71.07	2.97e+04	95.65	90.00	94.24	1.42e+05	100.00	77.27	94.32	1.48e+04
2	54.29	71.43	58.57	3.09e+04	92.17	73.33	87.46	1.45e+05	95.00	77.27	90.57	1.49e+04
3	77.94	57.14	72.74	2.98e+04	88.70	86.67	88.19	1.40e+05	92.50	68.18	86.42	1.62e+04
4	80.88	57.14	74.95	2.96e+04	92.17	90.00	91.63	1.39e+05	87.50	77.27	84.94	1.48e+04
5	85.00	57.14	78.04	3.14e+04	92.66	90.00	92.00	1.42e+05	85.71	81.82	84.74	1.56e+04
Media	74.76	60.00	71.07	3.03e+04	92.27	86.00	90.70	1.42e+05	92.14	76.36	88.20	1.53e+04

Cuadro 10: Resultados del **clasificador AM-BEST**

	ECOLI				Parkinsons				Breast-Cancer			
	%_class	%_red	Fit	T	%_class	%_red	Fit	T	%_class	%_red	Fit	T
1NN	80.81	0.00	60.61	0.00e+00	94.79	0.00	71.09	0.00e+00	95.59	0.00	71.69	3.60e+00
Greedy	79.68	28.57	66.90	1.20e+00	94.79	0.00	71.09	1.00e+00	95.25	4.00	72.44	1.10e+01
BL	73.24	57.14	69.21	6.21e+02	91.36	77.27	87.84	1.02e+03	93.00	84.00	90.75	1.25e+04
AGG	74.13	54.29	69.17	3.06e+04	92.71	75.45	88.40	1.53e+04	91.94	84.00	89.96	1.48e+05
AGG_BLX	76.69	57.14	71.80	3.06e+04	94.93	77.27	90.51	1.51e+04	92.62	86.67	91.13	1.43e+05
AGE	78.78	54.29	72.66	2.97e+04	95.43	78.18	91.12	1.50e+04	92.99	85.33	91.07	1.45e+05
AGE_BLX	76.14	60.00	72.11	3.00e+04	94.43	78.18	90.37	1.49e+04	93.33	84.00	91.00	1.42e+05
AM_ALL	77.34	57.14	72.29	3.02e+04	93.43	77.27	89.39	1.52e+04	93.87	85.33	91.73	1.46e+05
AM_RAND	71.81	62.86	69.57	3.03e+04	93.29	76.36	89.06	1.49e+04	95.43	84.67	92.74	1.44e+05
AM_BEST	74.76	60.00	71.07	3.03e+04	92.14	76.36	88.20	1.53e+04	92.27	86.00	90.70	1.42e+05

Cuadro 11: Comparación de resultados entre diferentes métodos de clasificación

8.3. Análisis de los resultados

Comparación general

Vamos a calcular la media en los tres conjuntos de datos para cada característica (%class, %red, Fit. y T) para cada clasificador para hacer una comparación general de los clasificadores mediante gráficos de barras.

En primer lugar, es importante destacar que la clasificación (%_class) representa la precisión del algoritmo en la clasificación de las instancias del conjunto de datos. En este aspecto vemos que el mejor clasificador es el 1-NN, cosa que puede tener sentido dado que considera todas las características (no hace reducción) y coge el vecino más cercano. Sin embargo, todos los demás valores están muy próximos (no debe confundir el intervalo de valores escogido para el eje Y) por lo que todos los clasificadores son buenos y válidos para esta tarea.

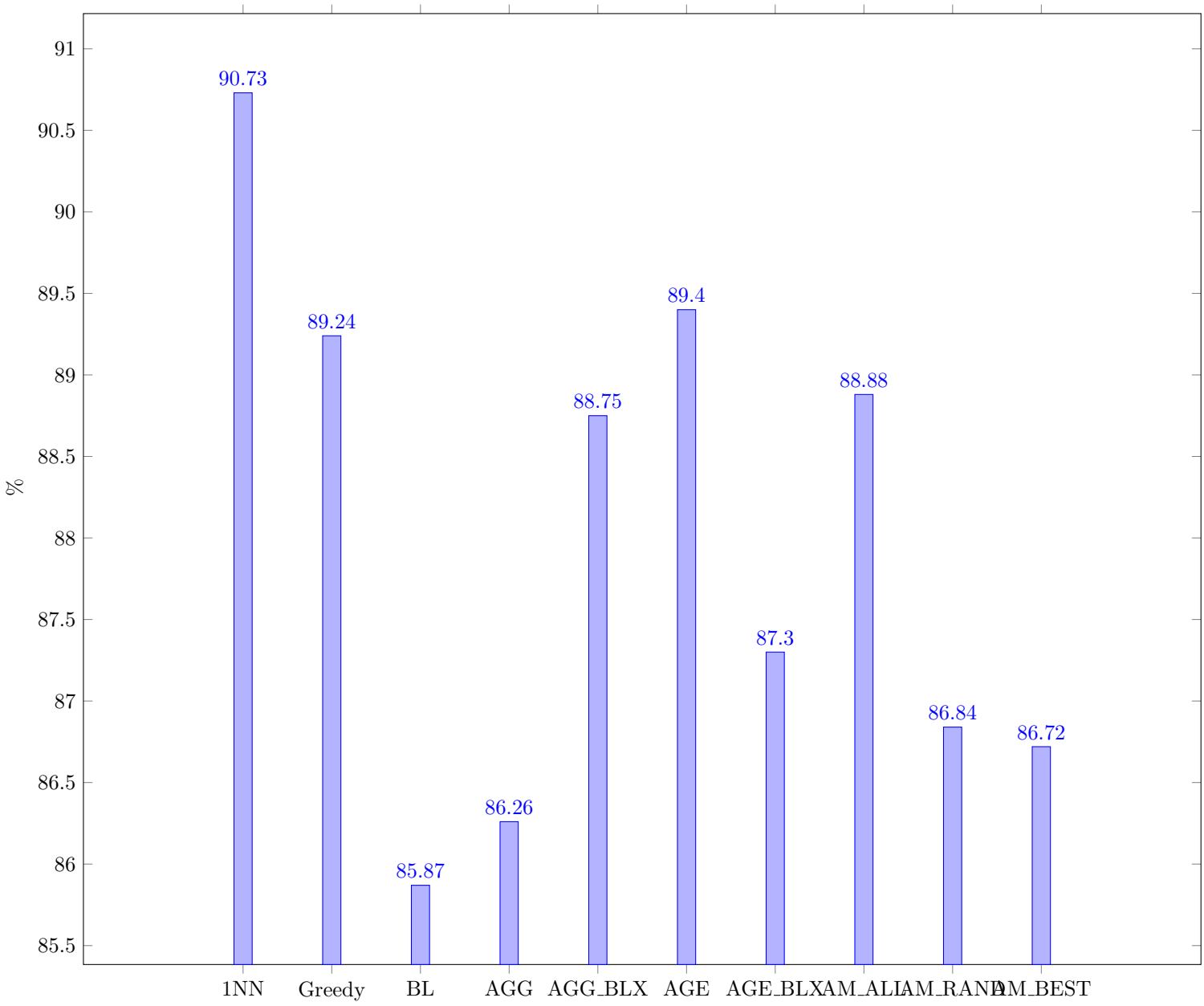

Figura 3: % class

Por otro lado, la reducción de características (%_red) indica la eficacia del algoritmo para reducir el número de características mientras mantiene un nivel aceptable de precisión en la clasificación. En este aspecto, 1-NN destaca negativamente, ya que no emplea reducción seguido de Greedy, que también reduce muy poco. El resto de clasificadores realizan un porcentaje de reducción muy similar por lo que comparar estos clasificadores según esta característica no sería apropiado ya que no es tan crucial como %clas. Todos ellos pueden ser efectivos en la reducción del número de características redundantes o irrelevantes, lo cual es una ventaja en conjuntos de datos con alta dimensionalidad. Destaca AM-RAND.

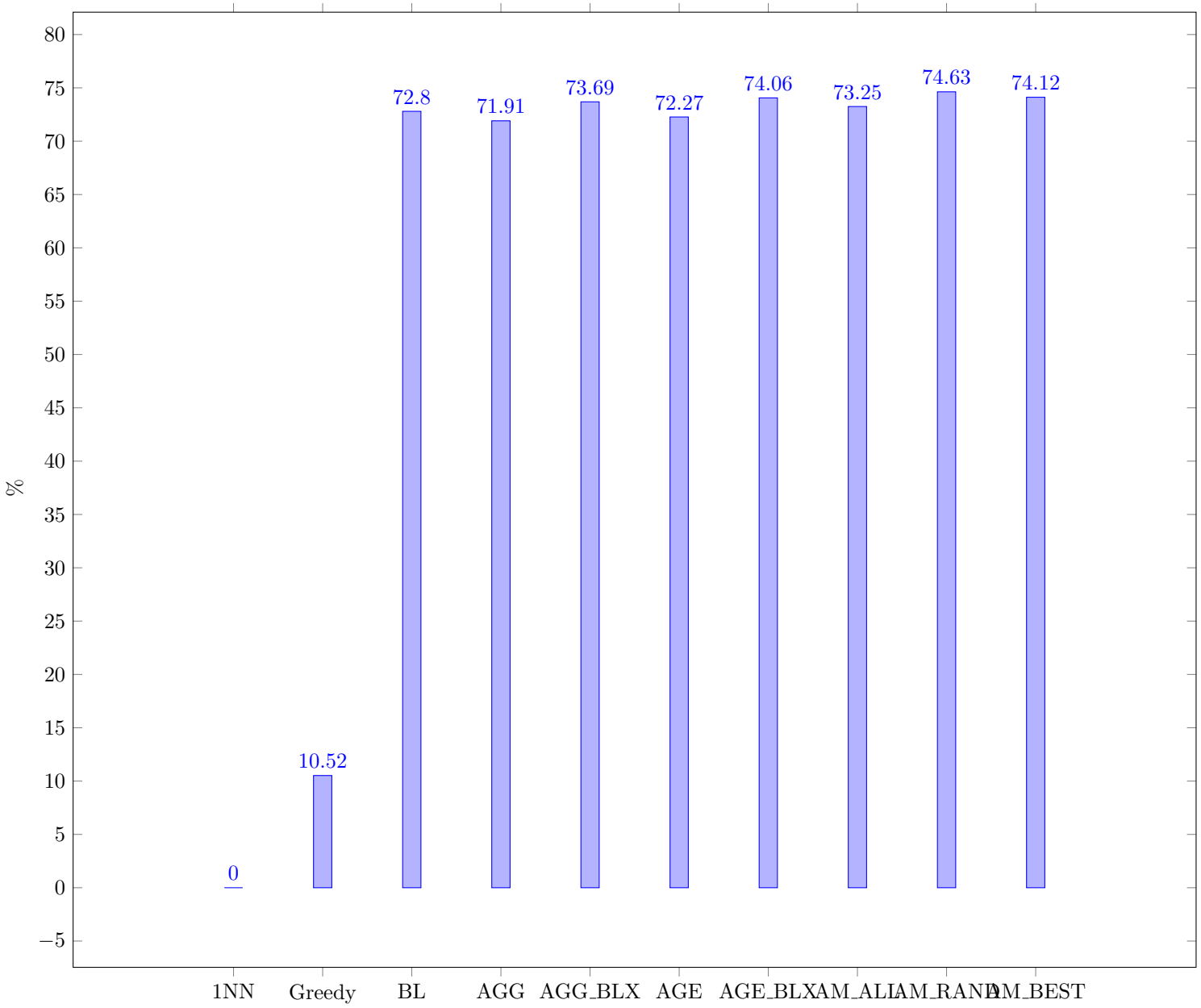

Figura 4: % red

En cuanto al Fitness, que es una medida compuesta que considera tanto la clasificación como la reducción de características, nos aporta información según la ponderación que nosotros consideremos para cada una de ellas. En nuestro caso, es un 75 % para %class y 25 % para %red, y así observamos que 1-NN y Greedy exhiben un rendimiento deficiente como cabía esperar debido a la reducción y que destacan AM-RAND

y AGE-Aritmético con valores cercanos al resto. Esto indica que, a pesar de que las tasas relativamente menores de clasificación frente a 1-NN o Greedy, el resto de clasificadores puede lograr un equilibrio muy competitivo entre precisión y reducción de características.

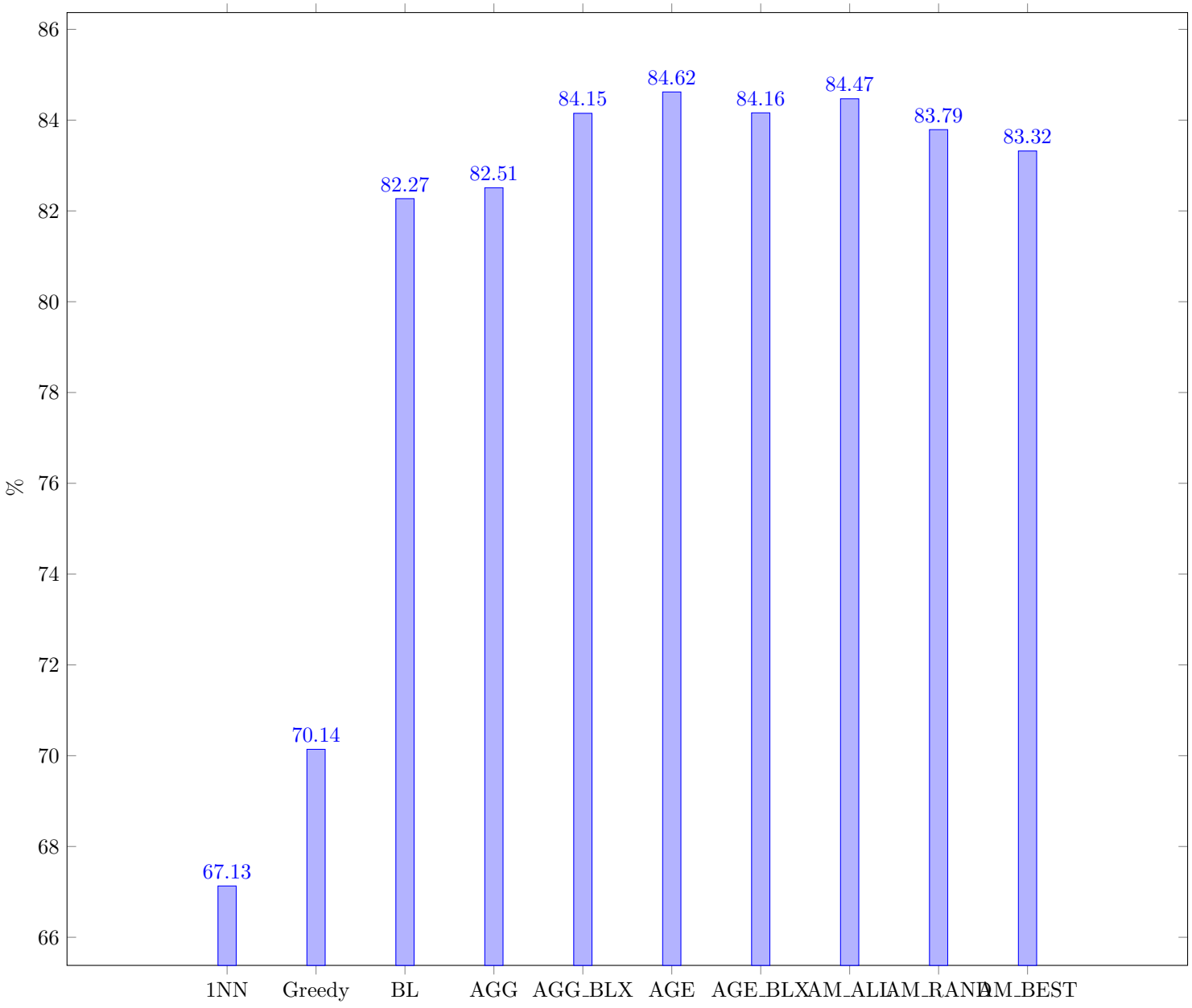

Figura 5: % Fit

Sin embargo, es importante tener en cuenta que este mejor rendimiento del resto de clasificadores en términos de Fitness se logra a expensas de un tiempo de ejecución significativamente mayor (T). Vemos que BL es una gran opción intermedia, ya que compite en cuanto a Fitness con todos los algoritmos genéticos y meméticos pero es el que menor tiempo consume de ellos, y con muchísima diferencia.

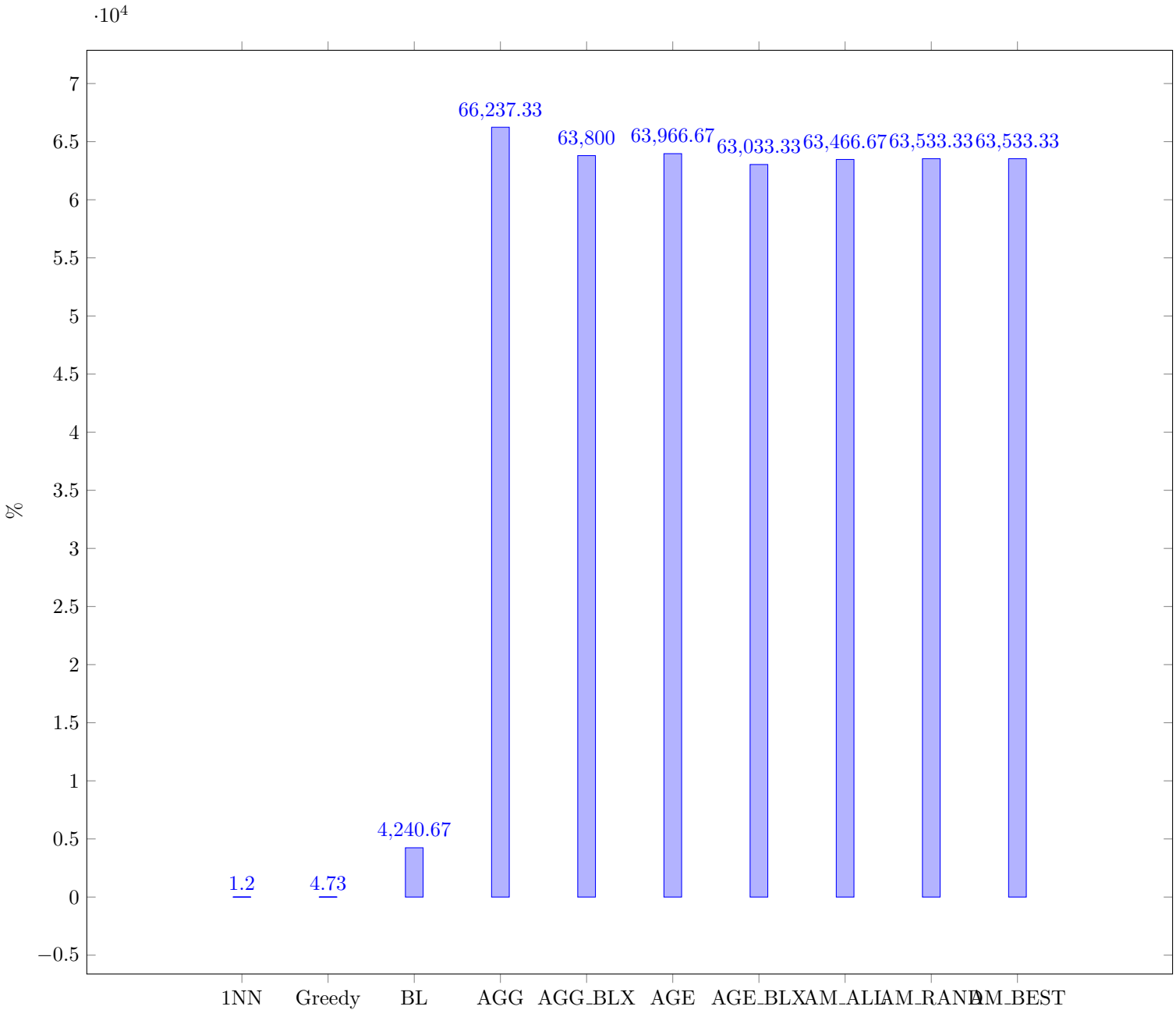
$\cdot 10^4$
7
6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
−0.5
%
1.2
4.73
4,240.67
66,237.33
63,800
63,966.67
63,033.33
63,466.67
63,533.33
63,533.33
1NN
Greedy
BL
AGG
AGG_BLX
AGE
AGE_BLX
AM_ALI
AM_RAND
AM_BEST

Figura 6: % T

Comparación entre algoritmos genéticos

AGE tiende a tener una precisión de clasificación y un ajuste ligeramente más altos en comparación con AGG.

AGG y AGE también muestran una reducción de dimensionalidad más baja en comparación con AGG_BLX y AGE_BLX, lo que sugiere que podrían estar conservando más características del conjunto de datos original.

En términos de tiempo de ejecución, AGG y AGE parecen ser menos eficientes que AGG_BLX y AGE_BLX en la mayoría de los casos, aunque la diferencia no es significativa. Sin embargo esta diferencia es más notable en el conjunto de datos Breast-Cancer, que es el conjunto más voluminoso, por lo que esto nos sugiere que utilizar el cruce BLX es más rápido que utilizar el cruce Aritmético.

Comparación entre algoritmos meméticos

Los algoritmos meméticos (AM_ALL, AM_RAND, AM_BEST) también muestran variaciones en los resultados en los tres conjuntos de datos.

AM_ALL generalmente muestra la mayor precisión de clasificación y el ajuste más alto entre los tres algoritmos meméticos. Esto tiene sentido ya que AM-ALL aplica BL a toda la población, por lo que todos los individuos mejoran y por tanto, los futuros padres y cruces serán mejores, no solo unos pocos.

AM_BEST tiende a tener una precisión de clasificación ligeramente menor y un ajuste más bajo en comparación con AM_RAND y AM_ALL. Por un razonamiento análogo al anterior pero de argumento contrario, como en AM-BEST solo se mejora la élite de la población, en la elección de padres y en los cruces se empeoran resultados.

AM_RAND parece ser el intermedio dado que su componente aleatoria permite mejorar toda la población pero el hecho de que no se aplique siempre BL sobre toda la población hace que esta mejore más lento.

Comparación entre algoritmos genéticos y meméticos

En comparación con los algoritmos genéticos, los meméticos muestran una tendencia a obtener tasas de clasificación ligeramente superiores, cosa totalmente lógica ya que los meméticos mejoran la población incluyendo BL sobre los individuos.

Los algoritmos genéticos y meméticos muestran tiempos de ejecución más largos en comparación con los métodos clásicos, pero en términos de eficiencia relativa entre ellos, como los algoritmos meméticos implementan el AGG-Aritmético, vemos que en todos los conjuntos de datos son más rápidos los meméticos que el algoritmo genético generacional. Esto tiene mucho sentido ya que los algoritmos genéticos, al implementar también BL, tendran que hacer menos iteraciones generacionales que es lo realmente costoso de estos algoritmos.

Conclusiones

En resumen, los algoritmos meméticos tienden a ofrecer tasas de clasificación ligeramente superiores en comparación con los genéticos además de un menor tiempo de ejecución. Los métodos clásicos de clasificación, como 1-NN, Greedy y BL, también son competitivos en términos de tasas de clasificación y son mucho más eficientes en tiempo de ejecución en comparación con los enfoques basados en metaheurísticas. Sin embargo, difieren mucho en reducción de dimensionalidad.

En conclusión, la elección del algoritmo más adecuado dependerá de las necesidades específicas del problema y las prioridades del usuario. Si se valora la precisión de clasificación sobre la eficiencia computacional, 1-NN, Greedy pueden ser opciones preferibles. Sin embargo, si la reducción de características es una consideración crítica pero queremos no excedernos mucho en tiempo de ejecución, BL podría ser la opción más adecuada. Si se puede tolerar un mayor tiempo de ejecución, escogeríamos sin duda alguno de los algoritmos meméticos ya que son una versión "mejorada"de los genéticos.

Referencias

[1] Wikipedia. K vecinos más próximos

[2] Tahir, M. A., Bouridane, A., & Kurugollu, F. (2007). Simultaneous feature selection and feature weighting using hybrid Tabu Search/K-nearest neighbor classifier

[3] Wikipedia. Algoritmos Genéticos.

[4] Universidad de Granada. Algoritmos Genéticos Generacionales y Algoritmos Genéticos Estacionarios.

[5] Wikipedia. Algoritmos Meméticos.

[6] Universidad de Granada. Algoritmo de Búsqueda Local.