Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Aprendizaje de Pesos en Características

Alejandro Borrego Megías[1]

07/04/2022

[1]26504975M, alejbormeg@correo.ugr.es, Grupo 3 Martes 17:30-19:30

Contents

Chapter 1

Descripción del problema APC.

El problema que trataremos es el APC (Aprendizaje de Pesos en Características). Antes de explicar el problema tendremos en cuenta los siguientes puntos:

- Conjunto de datos: Utilizaremos tres bases de datos que se explicarán más adelante y se representarán como usa sucesión de elementos sobre los cuales se han llevado a cabo una serie de mediciones o se aporta una serie de características y tienen asociada una etiqueta. Por lo tanto estamos en el ámbito de problemas de Aprendizaje Supervisado.
- Clasificador: Utilizaremos el clasificador de los K vecinos más cercanos con K=1 para clasificar ejemplos, de esta manera la etiqueta que asignaremos al ejemplo que clasificamos será la del elemento más cercano a él en el conjunto de datos del que disponemos.
- Distancia: Para medir la distancia entre elementos del conjunto usaremos la distancia euclídea ponderada:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)^2}$$

  Como podemos ver, cada elemento del sumatorio viene multiplicado por una constante w_i.

De esta manera, establecido el contexto del problema, nuestro objetivo será hallar ese vector w de pesos que dará prioridad a unos atributos sobre otros, para ello usaremos diversas técnicas aprendidas en clase.

Por lo tanto, el APC es un problema que pretende optimizar el rendimiento del clasificador KNN.

Chapter 2

Descripción de la aplicación de los algoritmos.

Métricas

En todos los algoritmos que empleemos vamos a utilizar un clasificador 1-NN (como se dijo anteriormente). De esta forma las métricas que usaremos para comprobar la bondad del vector de pesos w obtenido por cada algoritmo serán:

- **Tasa de clasificación**: que medirá el porcentaje de instancias correctamente clasificadas aplicando validación sobre el conjunto T de datos correspondiente (lo haremos generalmente sobre el conjunto de datos que usamos para entrenar). Su expresión es la siguiente:

$$tasa_{clas} = 100 \cdot \frac{n\ instancias\ bien\ clasificadas\ en\ T}{n\ instancias\ en\ T}$$

- **Tasa de reducción**: medirá el porcentaje de características descartadas, es decir, aquellas cuyo peso esté cercano a 0. Pondremos el umbral en 0.1. Su expresión es la siguiente:

$$tasa_{red} = 100 \cdot \frac{n\ valores\ <\ 0.1\ en\ W}{n\ caracteristicas}$$

Finalmente, las dos métricas anteriores se combinarán en la que será nuestra **función objetivo**, que intentaremos maximizar, y que tiene como expresión la siguiente:

$$F(W) = \alpha \cdot tasa_{clas}(W) + (1 - \alpha) \cdot tasa_{red}(W)$$

En nuestro aso $\alpha = 0.5$ de manera que daremos la misma importancia a la tarea de reducción como la de clasificación.

Validación Cruzada

Por otro lado, como sabemos de las técnicas de Aprendizaje Automático, para diseñar un clasificador, es necesario realizar dos tareas: **Aprendizaje y Validación**.

De esta forma, el conjunto de los ejemplos que tratamos se divide en dos, en conjunto de entrenamiento (que usaremos para entrenar el clasificador) y en el conjunto de Validación (que lo usaremos para validarlo).

Para mayor seguridad y garantías de generalización, se suelen realizar varias particiones Entrenamiento-Validación sobre los datos originales, es por ello que nosotros usaremos la técnica denominada **Cross Validation 5-Fold**, que consiste en los siguiente:

- El conjunto de datos se divide en 5 partes (5-fold) disjuntas al 20% con distribución de clases equilibrada.
- Aprendemos un clasificador con el 80% de los datos (es decir, con 4 de las 5 particiones) y se valida en el 20% restante (la partición que no hemos usado). Esto lo realizaremos 5 veces cambiando cada vez el conjunto de validación.
- De esta manera obtenemos 5 valores distintos de pocentaje de clasificación, por lo que para medir la calidad del método se realizará la media de los 5 obtenidos.
- Lo explicado anteriomente es el método general, pero en nuestro caso en cada iteración obtendremos un resultado de $tasa_{clas}$, $tasa_{red}$, del tiempo de ejecución del algoritmo y de la función objetivo. Por lo que nosotros al final promediaremos todas estas cantidades para obtener la bondad de nuestro clasificador final.

Clasificador KNN

Finalmente, como ya se ha comentado, usaremos el **clasificador KNN** con K=1, por lo que en esta última parte de la sección vamos a explicar algunos detalles del clasificador.

El proceso de aprendizaje consiste en mantener en memoria una tabla con los ejemplos de entrenamiento junto con la clase que tienen asociada, de esta forma, dado un nuevo ejemplo se calculará la distancia con los n elementos de la tabla y se escogen los k elementos más cercanos, de esta forma la clase que asignaremos al nuevo ejemplo sería la mayoritaria entre estas k clases. En nuestro caso como K=1 se simplifica el procesp, pues asignamos la clase del elemento más cercano.

La descripción en Pseudocódigo sería:

```
Clasificador1NN(datos, elemento):
  cmin=clase del primer elemento en entrenamiento e1
  dmin=distancia entre e1 y el nuevo ejemplo

  Para i=2 hasta m hacer:
    Se calcula la distancia entre ei y el nuevo ejemplo.

    Se comprueba si la nueva distancia es menor que dmin:
      - Si es menor se elige cmin=clase de ei y se actualiza dmin.
      - Si no es menor se continúa.
Devolver cmin.
```

Por otro lado la distancia que usaremos para el clasificador será la comentada en el apartado anterior, que denominaremos **distancia euclídea ponderada**, recordamos su expresión:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)^2}$$

Finalmente usaremos dos técnicas diferentes para comprobar la precisión de nuestro clasificador según nos encontremos en la fase de entrenamiento o la de validación en cada partición realizada en el 5-Fold Cross Validation:

- Entrenamiento: Usaremos el método **Leave One Out**, su pseudocódigo es el siguiente:

```
LeaveOneOut(Entrenamiento):
  Para cada elemento e en Entrenamiento:
    etiqueta_predicha=Clasificador1NN(Entrenamiento-{e},e)

    Si etiqueta_predicha=etiqueta_real de e
```

```
      correctos ++

  return 100*(correctos/num_datos)
```

Es decir, en cada paso eliminamos un elemento del conjunto de datos de entrenamiento y tratamos de clasificarlo, si las etiquetas predicha y real coinciden sumamos un acierto, si no se continúa. Finalmente se devuelve el porcentaje de acierto.

- Validación: En este caso método será intentar clasificar cada elemento del conjunto de validación usando los del conjunto de entrenamiento:

```
Evaluacion(Entrenamiento, Validacion):
  Para cada elemento e en Validacion:
    etiqueta_predicha=Clasificador1NN(Entrenamiento,e)

    Si etiqueta_predicha=etiqueta_real de e
      correctos ++

  return 100*(correctos/num_datos)
```

Para este método es muy importante que los datos estén normalizados entre [0,1] para no priorizar unos atributos sobre otros. Por ello, para cualquier conjunto de datos que utilicemos, lo primero que se hará será normalizar los datos y mezclarlos (para evitar que las clases estén desbalanceadas). El algoritmo para normalizar es el siguiente:

```
Normalizar(Datos):
  Para cada atributo a:
    Se calcula el máximo y mínimo de entre todos los datos

    Para cada elemento del conjunto de datos:
      Cambiar valor de atributo a por (x-min)/(max-min)
```

Bases de datos a usar

Las bases de datos que utilizaremos son:

- **Ionosphere**: datos de radar recogidos por un sistema en Goose Bay, Labrador. Los objetivos eran electrones libres en la ionosfera. Los "buenos" retornos de radar son aquellos que muestran evidencia de algún tipo de estructura en la ionosfera. Los retornos "malos" son aquellos que no lo hacen.

  Tiene 352 ejemplos con 34 atributos (señales procesadas) y 2 clases (good o bad).

- **Parkinsons**:contiene datos para distinguir entre lapresencia y la ausencia de la enfermedad de Parkinson en una serie de pacientes a partir de medidas biomédicas de la voz. Tiene 195 ejemplos con 23 atributos (incluyendo la clase). Las clases están desbalanceadas (147 enfermos 48 sanos). Algunos atributos son: Frecuencia mínima, máxima y media de la voz, medidas absolutas y porcentuales de variación de la voz, medidas de ratio de ruido en las componentes tonales.

- **Spectf-heart**:contiene atributos calculados a partir de imágenes médicas de tomografía computerizada (SPECT) del corazón de pacientes humanos. La tarea consiste en determinar si la fisiología del corazón analizado es correcta o no. Tiene 267 ejemplos, 45 atributos (incluyendo la clase) y 2 clases (sano o con patología).

Detalles técnicos

En nuestro problema, el conjunto de datos se va a representar por medio de un vector de pares (atributos, etiqueta).

El esquema de representación de soluciones será un vector de doubles que representará el vector de pesos w solución, de manera que si una característica tiene asociado un peso de 1, significa que se considera completamente en el cálculo de la distancia, si tiene un 0.1 o menos no se considera y para cualquier otro valor intermedio pondera la importancia del atributo que tiene asociado.

Chapter 3

Descripción de los Métodos de Búsqueda

En esta sección vamos a describir las distintas técnicas que emplearemos para resolver el problema APC.

Algoritmo RELIEF

Se trata de un algoritmo Voraz (greedy) de búsqueda secuencial, en el cual se parte de un vector de pesos inicializado a 0 que incrementa cada componente en función del enemigo más cercano a cada ejemplo y la reduce en función del amigo más cercano a cada ejemplo.

Entendemos el enemigo más cercano como el elemento más próximo al que estemos considerando que tiene asociada una clase distinta a la suya. Del mismo modo consideramos el amigo más cercano como aquel que es más próximo al elemento que estamos considerando y tiene misma clase asociada.

La implementación del algoritmo es la siguiente en pseudocódigo:

```
RELIEF(Entrenamiento,w):
    Inicializamos w a 0

    Para cada elemento e en el conjunto de entrenamiento E:
    se calcula distancia componente a componente al enemigo más cercano de e en E
    se calcula distancia componente a componente al amigo más cercano de e en E
    se calcula w=w+distancia_enemigo-distancia_amigo componente a componente

  Normalizamos w

  return w
```

Por su lado las implementaciones para las funciones que calculan las distancias al enemigo y amigo más cercano son las siguientes:

```
DistanciaEnemigoMasCercano(Entrenamiento,a):
  elemento de entrada a
  dmin=distancia euclídea entre a y el primer elemento de entrenamiento
  distancia=0

  Para cada elemento e en entrenamiento E:
    distancia=distancia euclídea entre e y a
```

```
    Si distancia < dmin && clase de e!= clase de a
      dmin=distancia
      enemigo_mas_cercano=e

  Calculamos distancias componente a componente entre a y enemigo_mas_cercano

  return distancias_componente_componente
```

Para el amigo más cercano sería:

```
DistanciaAmigoMasCercano(Entrenamiento,a):
  elemento de entrada a
  dmin=distancia euclídea entre a y el primer elemento de entrenamiento
  distancia=0

  Para cada elemento e en entrenamiento E:
    distancia=distancia euclídea entre e y a

    Si distancia < dmin && clase de e== clase de a
      dmin=distancia
      amigo_mas_cercano=e

  Calculamos distancias componente a componente entre a y amigo_mas_cercano

  return distancias_componente_componente
```

En el código, la estructura de datos que hemos empleado han sido los vectores de la STL. Además hemos tenido que sobrecargar los operadores de suma y resta para realizar sumas y restas componente a componente entre dos vectores.

Algoritmo de Búsqueda Local

En segundo lugar hemos empleado la técnica de búsqueda por trayectorias simples para implementar el algoritmo de **búsqueda local del primer mejor**.

Antes de describir el algoritmo vamos a aclarar algunos puntos necesarios para su comprensión:

- Definimos el entorno de una solución w como el conjunto formado por las solucuiones accesibles desde ella a través de un movimiento, que en nuestro caso será una mutación de una componente del vector w por medio de un valor aleatorio generado por una Distribución normal de media 0 y desviación típica $\sigma = 0.3$.

$$Mov(W, \sigma) = W' = (w_1, ..., w_i + z_i, ..., w_n)$$

$$z_i \sim N(0, \sigma^2)$$

- Para asegurarnos de que tras aplicar una mutación, nuestro vector resultante siga cumpliendo las restricciones del problema debemos truncar la componente modificada para que su valor se encuentre entre [0,1].

El tamaño del entorno de cada solución es infinito por ser un problema de codificación real. Para solucionar este problema vamos a mutar cada componente del vector w en un orden aleatorio y sin repetición hasta que **haya mejora en la función objetivo** o se hayan mutado todas las componentes. Si se produce mejora aceptamos la solución vecina y comenzamos de nuevo. Si no se produce ninguna mejora tras mutar las n componentes del vector se vuelve a repetir el proceso sobre la solución actual.

Este método que hemos descrito se denomina **Búsqueda Local del primer Mejor**, y en nuestro caso concreto partimos de un vector cuyas componentes son valores aleatorios de una distribución uniforme en el intervalo [0,1] y repetiremos el proceso descrito anteriormente hasta que se hayan realizado un máximo de 15000 llamadas a la función de evaluación o bien hasta que se hayan realizado un máximo de $20 \cdot$ n mutaciones sobre la solución actual sin que haya mejora (se han visitado $20 \cdot$ n vecinos sin que haya mejora).

Vamos a presentar a continuación los pseudocódigos del algoritmo y sus funciones.

- Inicialización Búsqueda Local: Es la función para inicializar el vector w antes del algoritmo. Utilizamos el generador de números pseudoaleatorios basado en el algoritmo de Marsenne Twister, que funciona muy bien como generador de números aleatorios[1].

```
InicializacionBL(int dimension, int i){
  Declaramos el vector w
  Inicializamos el generador mt19937 con la semilla i
  Inicializamos la distribución uniforme en [0,1]

  Mientras que i<dimension:
      elem_generado=dist(gen)
      añadimos a w elem_generado

  return w
}
```

- Movimiento: es la función que realiza la generación de vecinos en el entorno de la solución actual, volvemos a usar el generador de números pseudoaleatorios Marsenne Twister.

```
Mov(vector w, double sigma, int pos, int i){
  Inicializamos el generador mt19937 con la semilla i
  Inicializamos la distribución Normal(0,sigma^2)
```

[1] se denomina mt19937 debido a que se basa en el primo $2^{19937} - 1$

```
  z=elemento generado por la distribucion Normal
  w[pos]=w[pos]+z

  Si (w[pos]>1.0){
      w[pos]=1.0
  }

  Si(w[pos]<0.0){
      w[pos]=0.0;
  }
}
```

- Algoritmo de BL: Como aclaración al pseudocódigo, el vector w viene ya inicializado con la función anterior.

```
BusquedaLocal(vector de pares datos, vector w, semilla){
  Establecemos semilla(semilla)
  creamos vector con orden de mutaciones
  contador_evaluaciones=contador_mutaciones=0
  mejpra=false
  Vector w_mutado=w

  Mientras(contador_evaluaciones<15000 && contador_mutaciones<20*tam(w))
    mezclar orden de mutaciones
    mejora=false

    for(i=0; i<tam(w)&& mejora==false; i++)
      w_mutado=w
      Mov(w_mutado,0.3,orden_mutaciones[i])
      contador_mutaciones++
      Obtenemos precision en entrenamiento con LeaveOneOut
      Obtemenos tasa de reduccion
      Obtenemos valor de función evaluación
      contador_ev++

      Si mejora la funcion evaluacion
        w=w_mutado
        mejora=true

        contador_mutaciones=0
}
```

Metaheurísticas Basadas en trayectorias

El algoritmo de Búsqueda local es muy buen explotador en el sentido de que alcanza con precisión extremos locales, pero precisamente este también es un inconveniente del algoritmo, y es que es muy propenso a caer en extremos alejados del óptimo global.

Ante esta problemática se suele optar por modificaciones, las que exploraremos en esta práctica se caracterizan por:

- Permitir empeoramientos de la solución actual (Enfriamiento simulado).
- Comenzar la búsqueda desde otra solución inicial (ILS y BMB).

Algoritmo de Enfriamiento Simulado (ES)

El algoritmo de Enfriamiento simulado es un algoritmo de búsqueda por entornos con un criterio de aceptación de soluciones probabilístico basado en la termodinámica.

La filosofía del algoritmo es tratar de no descartar siempre soluciones peores a la actual, y permitir algunos movimientos hacia peores soluciones en ciertos momentos para evitar quedar atrapados en extremos locales.

En el caso concreto del algoritmo de Enfriamiento simulado, habrá una función de probabilidad que hará disminuir la probabilidad de estos movimientos a peores soluciones conforme la búsqueda avance y estemos (en teoría) más cerca del óptimo local. En cierto modo se busca *diversificar* al principio e *intensificar* al final.

El algoritmo realiza una búsqueda por entornos en el que el criterio de aceptación se adapta a lo largo de la ejecución. Para esto se usa una variable que simboliza la **Temperatura (T)** cuyo valor determina en qué medida se aceptan nuevas soluciones. Esta variable comienza con una temperatura inicial T_0 elevada y se reduce según una función de enfriamiento hasta alcanzar una temperatura final T_f.

En cada iteración del algoritmo generamos un número de vecinos concreto **L(T)** cada vez que se genera un vecino se aplica el criterio de aceptación probabilístico y si lo pasa sustituye a la solución actual (en caso de que sea mejor que la solución actual no es necesario el criterio probabilístico).

Así, la probabilidad de aceptación en nuestro caso será $e^{\Delta f/T}$ que depende del incremento del fitness entre la solución generada y la solución actual (Δf) y de la temperatura actual (T). De esta manera se aceptan muchas soluciones peores al principio, cuando la probabilidad es mayor, y menos al final, cuando es menor la probabilidad.

En caso de que se alcance el número máximo de hijos que se pueden generar **L(T)**, o se alcance un número **máximo de éxitos** (cambios de solución de acuerdo al criterio probabilístico) se enfriará la temperatura y se pasa a la siguiente iteración.

De esta forma, el algoritmo final queda de la siguiente manera:

```
EnfriamientoSimulado(matriz datos,vector w, generador_num_aleatorios &generator,
double T_final,double mu, double phi){

    //Variables que usaremos
    s=InicializacionBL(dim,generator)
    fitness_s = funcionEvaluacion(tasa_clas_s,tasa_red_s)
    w=s //Solución por ahora
    fitness_max=fitness_s
    T_inicial = CalculaTempInicial(fitness_max,mu,phi)
    T_actual = T_inicial
    contador_vecinos=0
    max_vecinos=10*dim
    contador_exitos=-1 //Para entrar al bucle
    max_exitos=0.1*max_vecinos
    contador_evaluaciones=0
```

```
    M=15000/max_vecinos
    incremento=0.0

    //Bucle principal
    //Salimos si se alcanza la temperatura final,
    //el máximo de evaluaciones o no hemos mejorado nada
    while(T_final<T_actual && contador_evaluaciones<15000
    && contador_exitos!=0){
        contador_vecinos=0;
        contador_exitos=0;
        while(contador_exitos<max_exitos && contador_vecinos<max_vecinos){
            //Generamos nueva solución mutando una componente
            s'=s;
            Mov(s',0.3,generator()%dim,generator);
            //Contamos un nuevo vecino
            contador_vecinos++;

            //Calculamos fitness
            fitness_s'=funcionEvaluacion(tasa_clas,tasa_red_);
            contador_evaluaciones++;

            //Calculamos el incremento
            incremento=fitness_s_prima-fitness_s;

            //Condiciones de éxito
            Si (incremento>0 o dado<=exp(incremento/(T_actual))){
                //si se dan contamos un éxito
                contador_exitos++;
                //s pasa a ser s_prima
                s=s';
                //actualizamos el fitness
                fitness_s=fitness_s';

                //Si mejora el fitness máximo pues tenemos nueva solución
                if(fitness_s > fitness_max){
                    fitness_max=fitness_s;
                    w=s;
                }
            }
        }
        //enfriamos
        T_actual=Enfriamiento(T_actual,T_inicial,T_final,M,generator);
    }
}
```

El cálculo de la temperatura inicial se realiza mediante la siguiente función:

```
double CalculaTempInicial(double coste, double mu, double phi){
    return (mu * coste)/(-log(phi));
}
```

En nuestro problema usaremos siempre $\mu = 0.3 = \phi$.

El enfriamiento por su parte se realizará mediante la siguiente función:

```
double Enfriamiento (double T, double T_inicial, double T_final, double M, std::mt19937 &generator){
    double beta=(T_inicial-T_final)/(M*T_inicial*T_final);
    return T/(1+beta*T);
}
```

Que pretende implementar el sistema de Cauchy modificado, en el cual la temperatura se actualiza de acuerdo a:

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

El resto de funciones empleadas ya se han explicado con anterioridad en otros algoritmos.

Búsqueda Multiarranque Básica (BMB)

Es un algoritmo de búsqueda **Global** que tiene dos etapas entre las que itera:

1. Se genera una solución aleatoria inicial.
2. Se aplica Búsqueda Local.

Hasta que se cumple el criterio de parada y se devuelve como resultado la mejor de entre todas las soluciones obtenidas. Con esto se pretende evitar caer en extremos locales con al Búsqueda Local, así se realiza una búsqueda desde distintas soluciones iniciales ubicadas en distintos puntos del espacio de búsqueda con la esperanza de alcanzar el óptimo o un extremeo local cercano a este.

En nuestro caso, vamos a generar un total de $T = 15$ soluciones iniciales aleatorias a las que aplicaremos Búsqueda Local un total de $15000/T = 1000$ iteraciones para cada solución generada.

Finalmente tomaremos como solución final la que mejor *fitness* tenga de las 15.

El pseudocódigo del algoritmo es:

```
void BusquedaMultiarranqueBasica(matriz datos,matriz validacion,vector w,
 generador_num_aleatorios generator, int tam_vector,
 int max_eval, int T){

    for(int i=0; i<T; i++)
      //Generamos solución actual
      solucion_actual=inicializacionBL(dim,generator);
      //Aplicamos búsqueda local max_eval/T iteraciones
      BusquedaLocal(datos,solucion_actual,generator,dim,max_eval/T)
      //Calculamos Fitness
          fitness=funcionEvaluacion()

      //Si es mejor que el máximo actualizamos solucion
      Si(fitness>fitness_max)
        fitness_max=fitness
        w=solucion_actual
}
```

Búsqueda local Reiterativa (ILS)

El algoritmo ILS se basa en la aplicación repetida de un algoritmo de Búsqueda Local a una solución inicial que se obtiene por medio de la mutación de un óptimo local previamente encontrado.

El algoritmo se compone de:

1. Una solución inicial (que generamos aleatoriamente)
2. Un procedimiento de mutación que usaremos para generar un cambio brusco sobre la solución actual para obtener otra intermedia. Para esto usaremos el operador de mutación usado en Búsqueda Local: Mov(W,α).
3. Procedimiento de Búsqueda Local.
4. Criterio de aceptación que nos indica a qué solución aplicar la próxima modificación. En nuestro caso será a la mejor solución que tengamos en ese momento.

El pseudocódigo será el siguiente:

```
MetodoILS()
  //Solucion aleatoria inicial
  w=InicializacionBL()
  //Aplicamos BL a w
  BusquedaLocal()
  //Calculamos su fitness
  fitness_max=funcionEvaluacion()

  Mientras(contador < T)
    solucion_actual=w
    //Mutamos el 10%
    for (int i=0; i<0.1*tam_w; i++){
        Mov(solucion_actual,0.4,i,generator);
    }
    //Aplicamos Búsqueda local
    BusquedaLocal()
    //Calculamos el Fitness
    fitness=funcionEvaluacion()

    //Si es mejor solucion la reemplazamos
    Si(fitness>fitness_max)
      fitness_max=fitness
      w=solucion_actual

    contador++
```

Por otro lado, realizaremos también una modificación del algoritmo para ejecutarlo con Enfriamiento Simulado en vez de con Búsqueda Local, el pseudocódigo sería el siguiente:

```
MetodoILS_ES()
  //Solucion aleatoria inicial
  w=InicializacionBL()
  //Aplicamos ES a w
  EfriamientoSimulado()
  //Calculamos su fitness
  fitness_max=funcionEvaluacion()

  Mientras(contador < T)
    solucion_actual=w
    //Mutamos el 10%
    for (int i=0; i<0.1*tam_w; i++){
        Mov(solucion_actual,0.4,i,generator);
    }
    //Aplicamos Búsqueda local
    EnfriamientoSimulado()
```

```
//Calculamos el Fitness
fitness=funcionEvaluacion()

//Si es mejor solucion la reemplazamos
Si(fitness>fitness_max)
  fitness_max=fitness
  w=solucion_actual

contador++
```

Chapter 4

Procedimiento

Para el desarrollo de la práctica se ha optado por implementar todo en el lenguaje c++ y a partir de código propio, únicamente haciendo uso de funciones y estructuras de datos de la STL y de la librería random para los procesos aleatorios.

Se ha estructurado todo en una carpeta denominada Software que contiene a su vez una carpeta includes con los ficheros de cabecera empleados: utilidades.h, RELIEF.h, BL.h, etc…

El fichero utilidades.h tiene funciones utilizadas por todos los algoritmos como el clasificador 1NN, Leave One Out u otras. Mientras que las otras dos contienen las funciones específicas de los otros dos algoritmos.

También se dispone de una carpeta src dónde se encientran las implementaciones de los ficheros anteriores así como el programa principal.

Por otro lado encontramos la carpeta instancias_APC con las bases de datos.

Finalmente, para compilarlo todo se dispone de un Makefile que además de la compilación contiene las reglas run y clean, para ejecutar y limpiar los directorios respectivamente.

Manual de uso

Para compilar y ejecutar el proyecto se tiene un makefile con reglas para compilar (make) y ejecutar (make run_parkinsons, make run_ionosphere y make run_heart) no es necesario recompilar para cambiar la base de datos pues se pasa como argumento al ejecutable según la orden del makefile que usemos.

Chapter 5

Experimentos y análisis de resultados

Casos del problema empleados y parámetros utilizados.

En la resolución del problema hemos optado por mezclar los datos y normalizarlos sea cual sea la base de datos empleada, así nos aseguramos la efectividad del clasificador y que se reduzca el problema de clases desbalanceadas como ocurre en la base de datos de **Parkinsons**. Por otro lado, para que los procesos aleatorios, como la generación de vectores aleatorios que siguen una distribución uniforme al comienzo de la Búsqueda Local, sean reproducibles en cualquier ordenador, hemos establecido una semilla al llamar a cada una de estas funciones que coincidía con la iteración de 5-fold Cross Validation en la que se encuentre el programa en ese momento, por lo que las semillas usadas son 1,2,3,4,5. Estas se usan también para inicializar el generador de números pseudoaleatorios mt19937 cuando lo usemos en las mutaciones de la Búsqueda local, solo que en lugar de utilizar las iteraciones de Cross Validation usaremos el índice del bucle for que recorre los atributos de w realizando las mutaciones aleatorias.

Resultados Obtenidos

Obtenemos los resultados para el algoritmo **RELIEF**:

Table 5.1: Resultados en el Dataset Ionosphere para RELIEF

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	85.714	2.941	44.327	3.394
Partición 2	90	2.941	46.470	3.272
Partición 3	90	2.941	46.470	4.259
Partición 4	84.285	2.941	43.613	3.316
Partición 5	88.732	2.941	45.836	3.485
Media	87.746	2.941	45.343	3.545

Table 5.2: Resultados en el Dataset Parkinsons para RELIEF

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	94.871	0	47.435	1.199
Partición 2	94.871	0	47.435	1.017

Particiones	Parkinsons			
Partición 3	97.435	0	48.717	1.213
Partición 4	92.307	0	46.153	1.328
Partición 5	100	0	50	1.005
Media	95.897	0	47.948	1.152

Table 5.3: Resultados en el Dataset Spectf_heart para RELIEF

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	79.71	0	39.855	4.122
Partición 2	82.608	0	41.304	4.498
Partición 3	76.811	0	38.405	4.172
Partición 4	85.507	0	42.753	4.781
Partición 5	90.411	0	45.2055	3.952
Media	83.009	0	41.504	4.303

Obtenemos los resultados para el algoritmo de **Búsqueda Local**:

Table 5.4: Resultados en el Dataset Ionosphere para BL

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	85.714	58.823	72.268	8825.46
Partición 2	87.142	55.882	71.512	3509.74
Partición 3	87.142	79.411	83.277	8396.57
Partición 4	91.428	58.823	75.126	3881.64
Partición 5	87.323	64.705	76.014	8485.97
Media	87.750	63.529	75.64	6619.88

Table 5.5: Resultados en el Dataset Parkinsons para BL

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	100	50	75	788.805
Partición 2	76.923	77.272	77.097	1018.39
Partición 3	94.871	72.727	83.799	965.224
Partición 4	97.435	68.181	82.808	426.535
Partición 5	97.435	68.181	82.808	804.335
Media	93.333	67.272	80.303	800.658

Table 5.6: Resultados en el Dataset Spectf_heart para BL

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	79.71	72.727	76.218	5768.42
Partición 2	78.260	54.545	66.403	5768.42
Partición 3	81.159	65.909	73.534	9876.57
Partición 4	86.956	59.090	73.023	6979.56
Partición 5	86.301	56.818	71.559	11173.2
Media	82.477	61.818	72.147	9528.56

Obtenemos los resultados para el algoritmo de **Enfriamiento Simulado**:

Table 5.7: Resultados en el Dataset Ionosphere para ES

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	88.5714	35.2941	61.9328	3542.24
Partición 2	84.2857	35.2941	59.7899	3526.17
Partición 3	95.7143	32.3529	64.0336	3495.62
Partición 4	85.7143	35.2941	60.5042	3505.82
Partición 5	88.7324	35.2941	62.0133	3470.94
Media	88.6036	34.7059	61.6548	3508.16

Table 5.8: Resultados en el Dataset Parkinsons para ES

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	100	50	75	755.589
Partición 2	89.7436	45.4545	67.5991	763.403
Partición 3	97.4359	36.3636	66.8998	752.234
Partición 4	97.4359	40.9091	69.1725	776.462
Partición 5	97.4359	50	73.7179	773.082
Media	96.41	44.54	70.47	764.154

Table 5.9: Resultados en el Dataset Spectf_heart para ES

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	76.8116	34.0909	55.4513	4532.26
Partición 2	86.9565	36.3636	61.6601	4381.53
Partición 3	81.1594	31.8182	56.4888	4370.6
Partición 4	79.7101	29.5455	54.6278	4371.91
Partición 5	84.9315	36.3636	60.6476	4406.46
Media	81.9138	33.6364	57.7751	4412.55

Obtenemos los resultados para el algoritmo de **Búsqueda Local Multiarranque**:

Table 5.10: Resultados en el Dataset Ionosphere para BMB

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	88.5714	88.2353	88.4034	35906.8
Partición 2	91.4286	91.1765	91.3025	35842.3
Partición 3	94.2857	85.2941	89.7899	36004.2
Partición 4	85.7143	85.2941	85.5042	36576.6
Partición 5	87.3239	88.2353	87.7796	35714.9
Media	89.4648	87.6471	88.5559	36009

Table 5.11: Resultados en el Dataset Parkinsons para BMB

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	87.1795	100	93.5897	7483.76
Partición 2	89.7436	95.4545	92.5991	7784.57
Partición 3	94.8718	86.3636	90.6177	7421.57
Partición 4	94.8718	100	97.4359	7917.07
Partición 5	100	100	100	7679.42
Media	93.3333	96.3636	94.8485	7657.28

Table 5.12: Resultados en el Dataset Spectf_heart para BMB

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	72.4638	79.5455	76.0046	45002.1
Partición 2	78.2609	79.5455	78.9032	45477.1
Partición 3	81.1594	77.2727	79.2161	45015.7
Partición 4	79.7101	79.5455	79.6278	44832.9
Partición 5	91.7808	86.3636	89.0722	43937.5
Media	80.675	80.4545	80.5648	44853.1

Obtenemos los resultados para el algoritmo de **ILS con Búsqueda Local**:

Table 5.13: Resultados en el Dataset Ionosphere para ILS-BL

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	90	97.0588	93.5294	35726.6
Partición 2	95.7143	100	97.8571	34967.6
Partición 3	88.5714	100	94.2857	36600.3
Partición 4	87.1429	100	93.5714	37243.4
Partición 5	85.9155	100	92.9577	35765.7

Particiones	Ionosphere			
Media	89.4688	99.4118	94.4403	36060.7

Table 5.14: Resultados en el Dataset Parkinsons para ILS_BL

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	94.8718	100	97.4359	5797.23
Partición 2	92.3077	100	96.1538	5754.64
Partición 3	97.4359	100	98.7179	6073.34
Partición 4	100	100	100	5801.88
Partición 5	100	100	100	5979.39
Media	96.9231	100	98.4615	5881.3

Table 5.15: Resultados en el Dataset Spectf_heart para ILS_BL

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	78.2609	95.4545	86.8577	47870.7
Partición 2	84.058	97.7273	90.8926	48201.4
Partición 3	86.9565	97.7273	92.3419	47747.2
Partición 4	86.9565	100	93.4783	48165.6
Partición 5	86.3014	97.7273	92.0143	47410.9
Media	84.5067	97.7273	91.117	47879.2

Obtenemos los resultados para el algoritmo de **ILS con Enfriamiento Simulado**:

Table 5.16: Resultados en el Dataset Ionosphere para ILS-ES

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	87.1429	44.1176	65.6303	57601
Partición 2	87.1429	55.8824	71.5126	56993.4
Partición 3	90	47.0588	68.5294	56825.8
Partición 4	87.1429	41.1765	64.1597	57177
Partición 5	84.507	47.0588	65.7829	56768.2
Media	87.1871	47.0588	67.123	57073.1

Table 5.17: Resultados en el Dataset Parkinsons para ILS-ES

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	94.8718	45.4545	70.1632	12286.8
Partición 2	87.1795	50	68.5897	12409.1
Partición 3	92.3077	54.5455	73.4266	12279.7

Particiones	Parkinsons			
Partición 4	94.8718	50	72.4359	12188.4
Partición 5	92.3077	59.0909	75.6993	12121
Media	92.3077	51.8182	72.0629	12257

Table 5.18: Resultados en el Dataset Spectf_heart para ILS-ES

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	76.8116	45.4545	61.1331	71913.4
Partición 2	79.7101	45.4545	62.5823	71790.8
Partición 3	85.5072	40.9091	63.2082	71684.5
Partición 4	86.9565	40.9091	63.9328	71732.4
Partición 5	89.0411	43.1818	66.1115	69315.8
Media	83.6053	43.1818	63.3936	71287.4

Los resultados anteriores se han obtenido en cada iteración del proceso de 5-fold Cross Validation y como podemos observar, por regla general el método **RELIEF** consigue unas tasas muy elevadas de Precisión sobre el conjunto de Test en todas las bases de datos utilizadas, **pero en cambio este método no consigue reducir atributos** o al menos no sirve para este cometido (a excepción de la base de datos de **Ionosphere** dónde se reduce en un 3% aproximadamente), por lo que no consigue una evaluación muy alta de la función objetivo, ya que nuestro propósito era reducir el mayor número de atributos posible a la vez que intentar mantener una elevada precisión al clasificar.

Sin embargo, el método de **Búsqueda local** si que tiene un comportamiento que se ajusta mejor a lo que queríamos obtener, pues como podemos observar, aunque las tasas de clasificación son siempre menores o iguales que las obtenidas con el método RELIEF, obtenemos unas tasas de reducción muy elevadas, por encima del 60% en la mayoría de los casos, y logrando un aumento muy considerable en los valores de la función objetivo en comparación con los obtenidos con RELIEF (normalmente por encima de los 70). Este hecho puede deberse a al naturaleza del algoritmo, pues las mutaciones se realizan en cada paso sobre un único atributo en lugar de todos a la vez como en RELIEF, por lo que es más fácil encontrar atributos relevantes y no relevantes.

Por otro lado, comparando los tiempos empleados por ambos algoritmos podemos ver que el método RELIEF es mucho más rápido que el de Búsqueda Local, pues en media tarda unos 2-3ms en entrenar, en cambio el de Búsqueda Local está muy condicionado a si las mutaciones mejoran más o menos la función objetivo lo que en un caso extremo podría llevar a ejecutar 15000 evaluaciones de la función objetivo, es por ello que los tiempos son mayores, aunque muy razonables para la notable mejora conseguida con respecto a RELIEF (menos de 10 segundos normalmente).

Si nos fijamos en el modelo basado en *trayectorias simples* (**Enfriamiento Simulado**) vemos que el comportamiento en todos los datasets es pobre, sobre todo en lo referente a la tasa de reducción (siempre por debajo del 50%), comparado con la **Búsqueda Local**, que consigue mejores resultados en todos los datasets. Esto puede deberse a que la función de enfriamiento que se emplea (**método de cauchy modificado**) enfría demasiado rápido, ya que en pocas iteraciones se alcanzan temperaturas muy próximas a la final, lo que se traduce en una mayor dificultad para aceptar soluciones por parte del criterio probabilístico (pues $e^{ f/T} será muy pequeño desde muy pronto) lo que puede suponer que el algoritmo acepte de primeras soluciones malas, que luego por la baja probabilidad no va a volver a cambiar. No obstante, las tasas de clasificación son elevadas, logrando buenos resultados en todas las bases de datos.Además el algoritmo es muy rápido, logrando ejecutarse en menos de 5 segundos en todas las bases de datos.

Por otro lado, si analizamos los modelos basados en Trayectorias múltiples observamos:

- El algoritmo **BMB** obtiene muy buenos resultados en todos los datasets, superando el valor 80 en la función

objetivo en todos los datasets, y superando el 90 en el dataset de Parkinsons, debido a las altas tasas de reducción y clasificación que obtiene. El motivo por el que esto ocurres es porque se beneficia del hecho de que la búsqueda local por si sola solía finalizar tras poco más de 1000 iteraciones en todos los datasets, luego el hecho de seleccionar $T = 15$ soluciones en distintos puntos del espacio de búsqueda y realizar 1000 iteraciones con cada una de ellas en el algorimto de **Búsqueda Local** obtenemos resultados muy próximos a los 15 extremos locales que hemos aproximado con cada vector inicial (recordemos que **BL** es muy buen explotador) y además, como la **BL** de por si sola apenas tarda más de 1000 iteraciones las soluciones obtenidas por los 15 vectores iniciales son muy parecidas a las que se habrían obtenido si hubiese finalizado el algoritmo de **BL**.

Por otro lado, esta diversidad de explorar distintos puntos iniciales nos permite quedarnos con el mejor extremo local obtenido, por eso mejora el comportamiento de la **BL** simple como veremos a continuación en la tabla resumen.

- El algoritmo **ILS-BL** como podemos observar obtiene excelentes resultados en todos los datasets, superando los 90 puntos de función objetivo en todos los datasets, llegando a casi el 100 en Parkinsons (98.46), con tasa de reducción y clasificación muy elevadas. Esto lo consigue porque desde mi punto de vista, la estrategia seguida por el algorimto es mejor (aunque parecida) a la de **BMB**, por ello consigue mejores resultados que este último en todos los datasets. El hecho de partir de una solución inicial muy próxima al óptimo local (gracias a la **BL**) permite que las mutaciones que le realicemos provoquen soluciones previsiblemente próximas a otros óptimos locales que junto con el proceso de búsqueda local siguiente consiguen aproximar muy bien. Por eso, este hecho de cambiar de buscar trayectorias desde puntos iniciales aleatorios (**BMB**) a puntos inciales aleatorios pero "próximos" en teoría a extremos locales permiten un mayor aprovechamiento del algoritmo de **BL** y como se demuestra, se obtienen muy buenos resultados.

  En cambio, con la variante que usa el algorimto **ES** en vez de **BL** obtiene resultados pobres (aunque mejores que el algoritmo **ES** por si solo) ya que como comentábamos antes, el algoritmo **ES** tiene este problema de enfriamiento muy rápido que no permite aproximar bien extremos locales.

Todos estos hechos comentados se pueden observar mejor en las siguientes tablas resumen:

Table 5.19: Resumen resultados en el Dataset Ionosphere para todos los algoritmos

Algoritmos	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
1-NN	86.599	0	43.299	2.653
RELIEF	87.746	2.941	45.343	3.545
Búsqueda Local	87.750	63.529	75.64	6619.88
Enfriamiento Simulado	88.6036	34.7059	61.6548	3508.16
BMB	89.4648	87.6471	88.5559	36009
ILS-BL	89.4688	99.4118	94.4403	36060.7
ILS-ES	87.1871	47.0588	67.123	57073.1

Table 5.20: Resumen resultados en el Dataset Parkinsons para todos los algoritmos

Algoritmos	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
1-NN	93.333	67.272	80.303	800.658
RELIEF	95.897	0	47.948	1.152
Búsqueda Local	93.333	67.272	80.303	800.658
Enfriamiento simulado	96.41	44.54	70.47	764.154

Algoritmos	Parkinsons			
BMB	93.3333	96.3636	94.8485	7657.28
ILS-BL	96.9231	100	98.4615	5881.3
ILS-ES	92.3077	51.8182	72.0629	12257

Table 5.21: Resumen resultados en el Dataset Spectf_heart para todos los algoritmos

Algoritmos	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
1-NN	96.923	0	48.461	0.58
RELIEF	83.009	0	41.504	4.303
Búsqueda Local	82.477	61.818	72.147	9528.56
Enfriamiento Simulado	81.9138	33.6364	57.7751	4412.55
BMB 80.675	80.4545	80.5648	44853.1	
ILS-BL	84.5067	97.7273	91.117	47879.2
ILS-ES	83.6053	43.1818	63.3936	71287.4

En estas tablas podemos ver mejor todo lo comentado anteriormente, y en vista de los resultados obtenidos podemos concluir:

- No todos los atributos recogidos son necesarios para obtener una alta tasa de clasificación, pues eliminando más de la mitad en cada base de datos se obtienen valores de clasificación muy elevados también, y además son **excluyentes**.
- El tiempo que tarda la búsqueda local es muy razonable, por lo que con apenas unos segundos más que en RELIEF obtenemos resultados mucho mejores.
- El tiempo de ejecución de los modelos basados en **trayectorias múltiples** son mucho mayores a los basados en **trayectorias simples**.
- Los modelos basados en **trayectorias múltiples** generalmente obtienen mejores resultados que los modelos basados en **trayectorias simples**.
- No obstante, no siempre los modelos basados en **trayectorias múltiples** son mejores que los modelos basados en **trayectorias simples**.

Estos hechos los desarrollaremos a continuación:

El hecho de que existan atributos innecesarios lo observamos en el hecho de que se pueden obtener altas tasas de clasificación eliminando muchos atributos. Por ejemplo, en el caso de la base de datos **Parkinsons**, obtenemos las siguientes soluciones por pantalla al ejecutar el algoritmo de Búsqueda Local:

```
Particion: 1

Solucion obtenida:
0.781141, 0, 0.0552385, 0.999041, 0.0200457, 0, 0,
0.0950612, 0.935539, 0.846311, 0.0972302, 0.524548, 0,
0.00326139, 0, 0.913962, 0.752749, 0.841574, 0.939128,
0.0387805, 0.715971, 1,

Particion: 2

Solucion obtenida:
0.0635937, 0, 1, 0, 0, 0.0329383, 0.698863, 0, 0,
```

```
0.0580529, 0.0398286, 0.998241, 0, 0.0121121, 0.719754, 0,
0.0819994, 0, 0.0600942, 1, 0.080975, 0.0718255,

Particion: 3

Solucion obtenida:
0.0707249, 0.839949, 0.0552385, 0, 0, 0.018748, 0.0406307,
0.0100419, 0.0935515, 1, 0, 0, 0, 0, 0, 0.913301, 0.696564,
0.398963, 0, 0, 0.0903173, 0.863963,

Particion: 4

Solucion obtenida:
0.900621, 0, 0.855621, 0.0343507, 0.0228713, 0, 0.0595821,
0.0869721, 0.903179, 0.0582782, 0.00515915, 0.572356, 0,
0.0294471, 0.728605, 0.811948, 0.0979347, 0.0504305,
0.0692294, 0.961353, 0.0190247, 0.0397804,

Particion: 5

Solucion obtenida:
0.0551801, 0.831328, 0, 0.979445, 0.089821, 0, 0, 0,
0.990821, 0.808282, 0.074774, 0.819473, 0.0264971, 0,
0.0580757, 0, 0.0875768, 0.877903, 0.0697856, 0.673025, 0,
0.0698192,
```

Si nos fijamos, los atributos ponderados por encima de 0.7 (valor a partir del cual considero que un atributo es muy relevante) en la primera iteración serían: 1,4,9,10,17,18,19,21,22

Sin embargo en la segunda iteración son: 3,12,15,20

Como vemos, niguno de los ponderados en la primera iteración por encima de 0.7 se repite en la segunda iteración.

Si seguimos ahora con la tercera iteración: 2, 10, 16, 22

Como vemos en este caso se ponderan algunos atributos comunes con la primera iteración y otros no ponderados hasta ahora por encima de 0.7.

En cambio, los valores de clasificación son todos muy similares y elevados.

Si nos fijamos ahora en los tiempos de ejecución, los algoritmos basados en **trayectorias simples** yel **RELIEF**, al aproximar solamente una solución tardan mucho menos que los algoritmos basados en **trayectorias múltipes**, que aproximan diversas soluciones distintas, lo que se traduce en un mayor tiempo de ejecución.

Por otro lado, observamos que los modelos basados en **trayectorias múltiples** obtienen (en general) mejores resultados que los modelos basados en **trayectorias simples**, así la **BMB** o el algorimto **ILS** obtienen mejores resultados que la **Búsqueda local** simple o el algoritmo **ILS-ES** obtiene mejores resultados que **ES**. Esto se debe, como ya se ha mencionado antes, a que los modelos basados en **trayectorias múltiples** añaden a los algoritmos basados en **trayectorias simples** esa capacidad de mejorar el aspecto que peor realizan (la exploración de soluciones) sacrificando en cierto modo parte de aquello que mejor realizan (la explotación de soluciones) logrando un compromiso que en la mayoría de casos mejora resultados.

No obstante, no siempre son mejores los algoritmos basados en **trayectorias múltiples** son mejores que los algoritmos basados en **trayectorias simples**. De hecho, en el caso en el que el espacio de búsqueda sea **Convexo** posiblemente un algorimto basado en **trayectorias simples** como la **BL** tendría un mejor rendimiento que sus versiones modificadas para **trayectorias múltiples** (**BMB** o **ILS**), ya que al existir únicamente un extremo local

que aproximar, lo harán mejor y con más exactitud (debido a su alta capacidad de explotación) que si partimos de diversos puntos y no explotamos tanto las soluciones.

Chapter 6

Referencias Bibliográficas

En esta práctica el material utilizado ha sido por regla general el proporcionado en el Seminario 2 y 3, así como de las transparencias de teoría.

Otros enlaces utilizados:

- Generador mt19937: https://www.cplusplus.com/reference/random/mt19937/
- Distribución Uniforme: https://www.cplusplus.com/reference/random/uniform_real_distribution/
- Distribución normal: https://www.cplusplus.com/reference/random/normal_distribution/