Práctica 3.b.
Técnicas de Búsqueda basadas en Trayectorias para el Problema del Aprendizaje de Pesos en Características

UNIVERSIDAD
DE GRANADA

Curso 2022/23

Jesús Miguel Rojas Gómez. 31015253-Y.
jesusjrg1400@correo.ugr.es
Subgrupo A2 — Martes 17:30-19:30

Contents

1 Descripción del problema

El problema de Aprendizaje de Pesos en Características (APC) se refiere a la selección y ponderación de características relevantes para mejorar la precisión de los modelos de aprendizaje automático. El objetivo es encontrar una combinación óptima de pesos para cada característica que maximice la precisión del modelo en la tarea de predicción. El problema APC está relacionado con el aprendizaje supervisado, en el que se utiliza un conjunto de datos etiquetados para entrenar un modelo y luego hacer predicciones sobre nuevos datos.

Utilizaremos como clasificador la técnica del vecino más cercano. El clasificador 1-NN, o más general, k-NN (del inglés, *k-Nearest Neighbors*) busca determinar la etiqueta de un objeto desconocido basándose en las etiquetas de los objetos más cercanos a él en el espacio de características (sin tener en cuenta al propio objeto). En un clasificador k-NN, el valor de k representa el número de vecinos más cercanos que se tomarán en cuenta para la clasificación del objeto desconocido. Para medir la similitud entre los objetos, el clasificador k-NN utiliza una función de distancia. En nuestro caso, utilizaremos la función distancia euclídea ponderada, que tiene la siguiente expresión:

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

donde e_j^i corresponde a la característica i del dato j, y w_i corresponde al valor del peso que pondera la característica i.

Evaluaremos el modelo combinando dos criterios: precisión y simplicidad:

- Para medir la precisión, utilizaremos el porcentaje de instancias correctamente clasificadas:

$$tasa\_clas = 100 \cdot \frac{\text{nº de instancias bien clasificadas}}{\text{nº total de instancias}}$$

- Para medir la simplicidad, utilizaremos el porcentaje de características descartadas:

$$tasa\_red = 100 \cdot \frac{\text{nº valores } w_i < 0.1}{\text{nº características}}$$

Para combinar ambos criterios, utilizaremos la siguiente función objetivo:

$$tasa\_fitness = \alpha \cdot tasa\_clas + (1 - \alpha) \cdot tasa\_red,$$

donde α es el parámetro que pondera precisión y reducción. En nuestro caso, utilizaremos $\alpha = 0.8$.

A lo largo de las distintas prácticas de la asignatura, estudiaremos e implementaremos diferentes algoritmos para obtener tales combinaciones de pesos, para después realizar una comparación entre ellos.

2 Descripción de la aplicación y algoritmos comúnes

Para realizar la aplicación y la implementación de los algoritmos utilizaremos Python, debido al manejo que hace este lenguaje de arrays y listas.

2.1 Representación de la solución

Utilizaremos arrays de la biblioteca numpy para representar la solución de APC. Concretamente, al ser un peso un elemento que pondera cada característica del conjunto de datos de entrada, la solución será representada por un array de pesos del tamaño del número de características de los datos, siendo la componente i del vector de pesos la ponderación correspondiente a la característica i de los datos de entrada. Cada componente del vector de pesos pertenecerá al intervalo $[0, 1]$.

2.2 Representación de los datos

Utilizaremos de nuevo arrays de numpy para representar los datos de entrada del problema APC. Para cada base de datos, disponemos de cinco ficheros con formato .arff. Dentro de cada fichero, hemos realizado tres divisiones: nombres de las características, datos, siendo estos las características, y clase. Ambos elementos del fichero serán almacenados en arrays de numpy, por separado. Estos, a su vez, se encontrarán en un array junto con los elementos del resto de ficheros, de nuevo por separado. He decidido realizar esta jerarquía para agilizar el proceso de ejecución y el número de llamadas al fichero main; de este modo, solo necesitaremos realizar una llamada por *dataset*. Por lo tanto, tenemos la siguiente estructura:

```
datosi = [ [...], [...], ..., [...] ]
clasesi = [ tipo1, tipo2, ..., tipo1 ]
...
datos_todos = [datos1, datos2, datos3, datos4, datos5]
clases_todos = [clases1, clases2, clases3, clases4, clases5]
```

Para leer los ficheros, he definido una función `leer_datos`, que operaría de la siguiente manera:

```
def leer_datos(ruta_fichero):
    abrimos ruta_fichero
    por cada linea en fichero:
        si no contiene @data la saltamos
        si contiene @atribute, guardamos el nombre del atributo
        si contiene @data: por cada linea sucesiva:
            dividimos la linea en datos y clase, y la almacenamos

    return nombres, datos, clases
```

Esta función toma como parámetros la ruta del fichero de datos y devuelve los nombres, los datos y las clases del fichero, atendiendo a su formato.

Una vez leídos todos los ficheros, se normalizan los datos de los mismos, de modo que toda característica de cualquier dato se encuentra en el intervalo $[0, 1]$. Para ello, he definido otra función `normalizar_datos`, que opera de la siguiente manera:

```
def normalizar_datos(datos):
    min = valor minimo de todas las caracteristicas
    max = valor maximo de todas las caracteristicas
    por cada dato en datos:
        por cada caracteristica en dato:
            caracteristica = (caracteristica-min) / (max-min)

    return datos_normalizados
```

Esta función toma como parámetro todos los datos y devuelve los datos normalizados. Destacar que esta normalización se realiza teniendo en cuenta todos los ficheros de un mismo *dataset*.

2.3 Funciones comunes

La implementación de esta práctica está organizada en tres archivos: `faux.py`, `P1.py` y `P3.py`. El archivo `faux.py` contiene las funciones comunes a los algoritmos implementados, el archivo `P1.py` contiene los algoritmos implementados y el main de la práctica 1, y el archivo `P3.py` contiene los algoritmos implementados y el main de la práctica 3.

Dentro del fichero `faux.py`, encontramos varias funciones. Concretamente, están las funciones ya mencionadas `leer_datos` y `normalizar_datos`, y las funciones necesarias para el calculo de la distancia, del clasificador 1-NN y la función objetivo.

Clasificador 1-NN

Para definir el clasificador necesitaríamos definir una función distancia ponderada, pero gracias al lenguaje que estamos utilizando y al manejo de arrays que realiza, no necesitamos explícitamente definir dicha función. Por lo tanto, tenemos el siguiente pseudocódigo:

```
def clasificador1nn(pesos, dato, datos, clases, indice=-1):
    distancias_sin_sumar = pesos x (datos-dato)^2
    # en este punto tenemos todas las caracteristicas de los datos
        con el resultado de la operacion
    distancias_sumadas = suma por filas de distancias_sin_sumar
    distancias = raiz cuadrada de distancias_sumadas
    si estamos utilizando el conjunto train para predecir la clase
        , implementamos leave-one-out y la distancia en la
        posicion indice la sustituimos por mas infinito

    return de la clase en clases relativa al dato_elegido en datos
        que hace minima la distancia entre dato (parametro) y
        dato_elegido
```

Esta función toma como parámetros un vector de pesos, el dato que queremos clasificar, y los datos y las clases del conjunto donde estamos realizando la comparación. Calcula la distancia ponderada al cuadrado del dato respecto a todos los datos, y devuelve la clase relativa al dato que hace mínima la distancia

anterior.

Esta función ha sido modificada respecto de la práctica 1 ya que presentaba errores. Concretamente, cuando sustituíamos las distancias que eran cero por infinito intentando implementar leave-one-out, sustituíamos más de un valor por infinito ya que había varias distancias que eran cero. Leave-one-out realmente solo se debe implementar cuando estamos entrenando un modelo, por tanto, he añadido un parámetro `indice` inicializado por defecto a -1 para distinguir cuando estamos entrenando y cuando testeando el modelo. Si el parámetro es distinto de su inicialización por defecto (es decir, -1), significa que estamos entrenando el modelo y por tanto debemos aplicar leave-one-out; esto lo hacemos sustituyendo la distancia relativa al dato que estamos comparando por infinito (utilizando el parámetro índice). De este modo, la implementación de esta función ya sería correcta.

Función objetivo

Como he mencionado en la introducción, la función objetivo es la función `tasa_fitness`. Para ello, necesitamos definir antes las funciones relativas a precisión y simplicidad:

```
def tasa_clas(pesos, datos_clasificar, clases_clasificar,
    datos_train, clases_train):
    si estamos entrenando:
        por cada dato,clase en datos_clasificar,clases_clasificar:
            si clasificador1nn(pesos,dato,datos_train,clases_train
                ,indice_dato_a_clasificar) == clase,
                contabilizamos un acierto
    si estamos testeando:
        por cada dato,clase en datos_clasificar,clases_clasificar:
            si clasificador1nn(pesos,dato,datos_train,clases_train
                ) == clase, contabilizamos un acierto

    return 100 x aciertos / no datos clasificar


def tasa_red(pesos):
    return 100 x no pesos con valor menor que 0.1 / n pesos
```

La función `tasa_clas` aplica el `clasificador1nn` que hemos definido previamente a cada dato a clasificar, y compara la clase obtenida con la clase verdadera del propio dato. Una vez hecho esto para cada dato, devuelve el porcentaje de aciertos. La función `tasa_red` devuelve el porcentaje del número de pesos con valor menor que 0.1.

La función `tasa_clas` por consiguiente también ha sufrido cambios de cara a la práctica 2. Como debemos distinguir entre entrenar y testear un modelo, debemos distinguir a la hora de realizar una llamada al clasificador. En este caso, he implementado esta nueva casuística comparando el tamaño del conjunto de datos a clasificar y a entrenar; si son del mismo tamaño, significa que estamos entrenando al modelo, si no, estamos testeando. De este modo la función ya estaría totalmente correcta.

Una vez definidas ambas funciones, podemos definir la función objetivo, realizando una ponderación según un valor alpha, que será fijado en el main a 0.8:

```
def tasa_fitness(alpha, pesos, datos_clasificar, clases_clasificar
    , datos_train, clases_train):
    return alpha x tasa_clas(pesos, datos_clasificar,
        clases_clasificar, datos_train, clases_train) + (1-alpha)
        x tasa_red(pesos)
```

Main

Todo lo relativo a la ejecución de los algoritmos y obtención de los resultados se encuentra en el main del archivo `P3.py`.

En primer lugar, leo los datos de los cincos ficheros, los junto en un mismo array, los normalizo y los vuelvo a dividir. En cuanto a las clases, dependiendo del *dataset* las clases tenian un formato diferente, por lo que decidí transformarlas a un valor numérico, ayudándome en la transformación de un diccionario. Una vez hecho esto, fijamos la semilla y con ayuda del módulo tabulate construimos la tabla con los resultados. Por cada algoritmo, realizamos cinco ejecuciones, cada una con un fichero de test distinto, cronometramos la ejecución y calculamos las métricas respecto al fichero test correspondiente. Una vez realizadas las cinco ejecuciones, imprimimos los resultados y pasamos al siguiente algoritmo.

Para cronometrar cada ejecución hemos utilizado la librería timeit.

3 Descripción de los algoritmos implementados

En total, para esta práctica, hemos implementado cinco algoritmos nuevos: ES, BMB, ILS, ES-ILS y VNS.

En esta práctica, como clases y funciones comunes generales, hemos reciclado el operador de mutación y la búsqueda local de la práctica 1:

```
def mov(pesos, indice, delta):
    pesos[indice] = pesos[indice] + distribucion_normal(0,delta)
    si pesos[indice] > 1 entonces pesos[indice] = 1
    si pesos[indice] < 0 entonces pesos[indice] = 0
    return pesos
```

Esta función toma como parámetro el vector de pesos, el índice a modificar y el valor de delta, y realiza tal mutación.

```
def busqueda_local(n_max_vecinos, n_max_evaluaciones, delta, datos
    , clases, pesos=[]):
    si tamano(pesos) == 0:
        pesos = vector inicializado segun una distribucion
            uniforme en [0,1]
    vecinos_consecutivos = no de mutaciones consecutivas
        realizadas = 0
    evaluaciones = no de evaluaciones de la funcion objetivo = 0
```

```
tasa_ultima_mejora = tasa_fitness de los pesos inicializados =
     tasa_fitness(0.8, pesos, datos, clases, datos, clases)

mientras vecinos_consecutivos < n_max_vecinos y evaluaciones <
     n_max_evaluaciones:
    hacemos una copia de los pesos := copia
    elegimos un indice aleatorio := indice
    mov(copia,indice,delta)
    tasa_actual = tasa_fitness con el nuevo vector de pesos
    si se produce mejora:
        actualizamos el valor de los pesos, la tasa de la
            ultima mejora, y el numero de vecinos consecutivos
    vecinos_consecutivos = vecinos_consecutivos+1
    evaluaciones = evaluaciones+1

return pesos
```

Esta función toma como parámetro las condiciones de parada del algoritmo de búsqueda local, el valor δ de la mutación, los datos y clases a clasificar, y el vector de pesos sobre el que aplicar la búsqueda local, y devuelve el vector de pesos modificado. Esta función ha sido modificada para esta práctica. En concreto, se le ha añadido un nuevo parámetro, pesos, inicializado por defecto a una lista vacía. En el caso de proporcionar este parámetro, se aplica la búsqueda local sobre dicho vector de pesos.

Una vez definidas las clases y funciones necesarias, podemos describir los principales algoritmos de la práctica.

3.1 ES. Enfriamiento Simulado.

Este algoritmo toma como parámetros los datos de train y de test a utilizar, un vector de pesos inicializado por defecto a una lista vacía y el número máximo de iteraciones, para así aplicar el algoritmo de enfriamiento simulado. Este algoritmo sigue la misma estructura que una búsqueda local, solo que aceptamos empeorar soluciones de acuerdo a una baja probabilidad. Consta de varias etapas:

- Esquema de enfriamiento: se empleará el esquema de Cauchy modificado, en el que:
$$T_{k+1} = \frac{T_k}{1+\beta T_k}, \qquad \beta = \frac{T_0 - T_f}{M T_0 T_f},$$
donde M es el número de enfriamientos a realizar, T_0 es la temperatura inicial y T_f es la temperatura final. La temperatura inicial y final serán inicializadas a:
$$T_0 = \frac{\mu C(S_0)}{-ln(\phi)}, \qquad T_f = 10^{-4},$$
donde $C(S_0)$ es el coste de la solución inicial, $\phi = 0.2$ y $\mu = 0.3$.

- Vecino y exploración del entorno: utilizaremos el operador de mutación definido, y lo aplicaremos a una característica aleatoria del vector de pesos.

- Condición de enfriamiento: se enfriará la temperatura cuando se haya generado un número máximo de vecinos o cuando se haya aceptado un

número máximo de vecinos. En este caso, el número máximo de vecinos es $10n$ y el número máximo de aceptaciones es n.

- Condición de parada: cuando la temperatura sea 0 o se haya alcanzado el número máximo de iteraciones. En nuestro caso, el número máximo de iteraciones es 15000.

A continuación vemos el pseudocódigo del algoritmo:

```
def ES(datos_train, clases_train, datos_test, clases_test, pesos =
    [], max_iters=MAX_ITERS_ES):
    si pesos == []: inicializamos pesos seg n una distribuci n
        uniforme en [0,1]

    fitness_actual = tasa_fitness usando pesos
    mejor_fitness = fitness_actual
    mejor_pesos = copia(pesos)

    # valores iniciales del algoritmo
    max_vecinos = 10 * len(pesos)
    max_exitos = 0.1 * max_vecinos
    M = redondear(max_iters / max_vecinos)

    T0 = (0.3*fitness_actual) / (-ln(0.2))
    Tf = 10^-4

    mientras Tf > T0:
        Tf = Tf * 10^-4

    beta = (T0-Tf)/(M*T0*Tf)

    T_actual = T0
    n_exitos = 1
    it = 1

    mientras haya exitos y it <= max_iters y T_actual > Tf:
        n_exitos = 0
        n_vecinos = 0
        mientras n_exitos < max_exitos y n_vecinos < max_vecinos
            # mutamos
            w = copia(pesos)
            w = mov(w, indice_aleatorio, SIGMA=0.3)
            nuevo_fitness = tasa_fitness usando pesos
            it++

            # sustitucion
            diferencia = nuevo_fitness - fitness_actual
            si diferencia > 0 o se da la probabilidad exp(
                diferencia/T_actual):
                fitness_actual = nuevo_fitness
                pesos = copia(w)
                n_exitos++
                si fitness_actual > mejor_fitness:
                    mejor_fitness = fitness_actual
                    mejor_pesos = pesos

            n_vecinos++

        # actualizamos la temperatura
        T_actual = T_actual (1+beta*T_actual)

    si es ES:
```

```
        # resultados
        tclas = tasa_clasificacion de test con mejor_pesos
        tred = tasa_reducci n de test con mejor_pesos
        tfit = tasa_fitness de test con mejor_pesos

        return tclas,tred,tfit
    si es ILS-ES:
        # pesos del ES
        return mejor_pesos
```

Destacar de este algoritmo que los datos de test no se utilizan hasta que ha acabado el número máximo de iteraciones. Para el cuerpo del bucle, únicamente se utilizan los datos de train.

3.2 BMB. Búsqueda Multiarranque Básica

Este algoritmo toma como parámetros los datos y clases de train, y los datos y clases de test, y aplica una búsqueda multiarranque básica. Este algoritmo simplemente genera un número determinado de soluciones aleatorias y aplica búsqueda local a cada una de ellas, para devolver la mejor solución. En concreto se generarán 15 soluciones aleatorias.

Su descripción en pseudocódigo es la siguiente:

```
def BMB(datos_train, clases_train, datos_test, clases_test):
    pesos_actual = segun una distribucion uniforme en [0,1]
    mejor_fitness = tasa_fitness usando pesos_actual
    mejor_pesos = copia(pesos_actual)

    para cada i en rango(ITERS_BMB):
        # busqeuda local a la solucion inicial
        pesos_actual = busqueda_local(len(mejor_pesos),
            MAX_EVALUACIONES_BMB, SIGMA_MOV, datos_train,
            clases_train, pesos_actual)
        fitness_actual = tasa_fitness usando pesos_actual

        # nos quedamos con el mejor
        si fitness_actual > mejor_fitness:
            mejor_pesos = copia(pesos_actual)
            mejor_fitness = fitness_actual

        # nueva solucion
        pesos_actual = nuevo vector segun una distribucion
            uniforme en [0,1]

    # resultados
    tclas = tasa_clasificacion de test usando mejor_pesos
    tred = tasa_reducci n de test usando mejor_pesos
    tfit = tasa_fitness de test usando mejor_pesos

    return tclas, tred, tfit
```

De nuevo, destacar de este algoritmo que los datos de test no se utilizan hasta que ha acabado el número máximo de iteraciones. Para el cuerpo del bucle, únicamente se utilizan los datos de train.

3.3 ILS. Búsqueda Local Reiterada.

Este algoritmo toma como parámetros los datos y clases de train, y datos y clases test, y aplica una búsqueda local reiterada. Este algoritmo en esencia es igual que el anterior, solo que en lugar de generar una nueva solución cada vez, elegimos la mejor solución actual, aplicamos una mutación brusca, y volvemos a aplicar la búsqueda local sobre dicha solución. En este caso el algortimo también se repetirá un total de 15 veces, y la mutación brusca consistirá en cambiar el 10% de las características según una distribución uniforme.

Su descripción en pseudocódigo es la siguiente:

```
def ILS(datos_train, clases_train, datos_test, clases_test):
    pesos = segun una distribucion uniforme en [0,1]

    # busqueda local a la solucion inicial
    pesos = busqueda_local(len(pesos), MAX_EVALUACIONES_BMB,
        SIGMA_MOV, datos_train, clases_train, pesos)
    mejor_fitness = tasa_fitness usando pesos
    mejor_pesos = copia(pesos)

    mutaciones = redondear(0.1 * len(pesos))
    si mutaciones < 2: mutaciones = 2

    para cada i en rango(ITERS_BMB-1):
        para cada _ en rango(mutaciones):
            # mutacion brusca
            indice = indice_aleatorio
            pesos[indice] = distribucion uniforme en [0,1]

        pesos = busqueda_local(len(pesos), MAX_EVALUACIONES_BMB,
            SIGMA_MOV, datos_train, clases_train, pesos)
        fitness_actual = tasa_fitness usando pesos

        # sustitucion
        si fitness_actual > mejor_fitness:
            mejor_fitness = fitness_actual
            mejor_pesos = copia(pesos)

        pesos = copia(mejor_pesos)

    # resultados
    tclas = tasa_clasificacion de test con mejor_pesos
    tred = tasa_reducci n de test con mejor_pesos
    tfit = tasa_fitness de test con mejor_pesos

    return tclas, tred, tfit
```

De nuevo, destacar de este algoritmo que los datos de test no se utilizan hasta que ha acabado el número máximo de iteraciones. Para el cuerpo del bucle, únicamente se utilizan los datos de train.

3.4 ILS-ES. Búsqueda Local Reiterada con Enfriamiento Simulado

Este algoritmo toma como parámetros los datos y clases de train, y datos y clases test, y aplica una búsqueda local reiterada utilizando enfriamiento simu-

lado. Este algoritmo es exactamente igual que el anterior, solo que sustituyendo la búsqueda local que se aplica por un enfriamiento simulado.

El pseudocódigo del algoritmo es el siguiente:

```
def ILS_ES(datos_train, clases_train, datos_test, clases_test):
    pesos = segun una distribucion uniforme en [0,1]

    # busqueda local a la solucion inicial
    pesos = ES(datos_train, clases_train, datos_train,
        clases_train, pesos, MAX_EVALUACIONS_BMB)
    mejor_fitness = tasa_fitness usando pesos
    mejor_pesos = copia(pesos)

    mutaciones = redondear(0.1 * len(pesos))
    si mutaciones < 2: mutaciones = 2

    para cada i en rango(ITERS_BMB-1):
        para cada _ en rango(mutaciones):
            # mutacion brusca
            indice = indice_aleatorio
            pesos[indice] = distribucion uniforme en [0,1]

        pesos = ES(datos_train, clases_train, datos_train,
            clases_train, pesos, MAX_EVALUACIONS_BMB)
        fitness_actual = tasa_fitness usando pesos

        # sustitucion
        si fitness_actual > mejor_fitness:
            mejor_fitness = fitness_actual
            mejor_pesos = copia(pesos)

        pesos = copia(mejor_pesos)

    # resultados
    tclas = tasa_clasificacion de test con mejor_pesos
    tred = tasa_reducci n de test con mejor_pesos
    tfit = tasa_fitness de test con mejor_pesos

    return tclas, tred, tfit
```

De nuevo destacar que los datos de test unicamente son utilizados al final del algoritmo.

3.5 VNS. Búsqueda de Vecindario Variable.

Este algoritmo toma como parámetros los datos y clases de train, datos y clases test, y un valor k, y aplica una búsqueda de vecindario variable. Este algoritmo es exactamente igual que ILS, solo que al realizar la búsqueda local, actualizamos un número de características igual a un valor variable en función de k. En nuestro caso, k será 3, y en concreto irá aplicando búsqueda local modificando en la mutación 1,2 y 3 características, siguiendo un esquema circular.

Para ello, hemos tenido que modificar la búsqueda local de la siguiente forma, para que acepte mutaciones de varias componentes a la vez:

```
def busqueda_local_modificada(n_max_vecinos, n_max_evaluaciones,
    delta, datos, clases, pesos=[], k=KMAX_VNS):
```

```
    si len(pesos) == 0:
        pesos = pesos segun una distribucion uniforme

    tasa_ultima_mejora = tasa_fitness usando pesos

    vecinos_consecutivos = 0
    evaluaciones = 0
    mientras vecinos_consecutivos < n_max_vecinos y evaluaciones <
         n_max_evaluaciones:
        # mutamos k caracteristicas
        indice = indice_aleatorio
        copia = copia(pesos)
        copia = mov(copia, indice, delta=SIGMA_MOV)
        vecinos_consecutivos = vecinos_consecutivos + 1
        para cada i en rango(k-1):
            indice = indice_aleatorio
            copia = mov(copia, indice, delta=SIGMA_MOV)
        tasa_actual = tasa_fitness usando copia
        evaluaciones = evaluaciones + 1
        si tasa_actual > tasa_ultima_mejora:
            pesos = copia
            tasa_ultima_mejora = tasa_actual
            vecinos_consecutivos = 0

    return pesos
```

Una vez definida esta función, podemos presentar el pseudocódigo del algoritmo:

```
def VNS(datos_train, clases_train, datos_test, clases_test, k_max=
    KMAX_VNS):
    # inicializamos pesos
    pesos = inicializar_pesos(datos_train)
    k = 0

    # busqueda local a la solucion inicial
    pesos = busqueda_local_modificada(len(pesos),
        MAX_EVALUACIONES_BMB, SIGMA_MOV, datos_train, clases_train
        , pesos=pesos, k=(k % k_max)+1)
    mejor_fitness = tasa_fitness usando pesos
    mejor_pesos = copia(pesos)

    mutaciones = redondear(0.1 * len(pesos))
    si mutaciones < 2: mutaciones = 2

    para cada i en rango(ITERS_BMB-1):
        # mutacion brusca
        para cada _ en rango(mutaciones):
            indice = indice_aleatorio
            pesos[indice] = inicializar_pesos(datos_train)

        # busqueda local a pesos actuales
        pesos = busqueda_local_modificada(len(pesos),
            MAX_EVALUACIONES_BMB, SIGMA_MOV, datos_train,
            clases_train, pesos=pesos, k=(k % k_max)+1)
        fitness_actual = tasa_fitness usando pesos

        # actualizacion
        si fitness_actual > mejor_fitness:
            mejor_fitness = fitness_actual
            mejor_pesos = copia(pesos)
            k = 0
```

```
        sino:
            k = k + 1

        # repetimos sobre los mejores pesos
        pesos = copia(mejor_pesos)

    # resultados
    tclas = tasa_clasificacion de test usando mejor_pesos
    tred = tasa_reduccion de test usando mejor_pesos
    tfit = tasa_fitness de test usando mejor_pesos

    return tclas, tred, tfit
```

De nuevo, los datos de test son utilizados únicamente para devolver las métricas.

4 Desarrollo de la práctica y manual de usuario

En mi caso, he desarrollado la práctica desde cero apoyándome en lo entregado para la práctica 1, con las modificaciones pertinentes.

Para obtener correctamente los resultados, realizaremos la siguiente ejecución:

```
<interprete-python> P3.py <fichero-1> <fichero-2> <fichero-3> <
    fichero-4> <fichero-5> <semilla>
```

El orden de los ficheros determina el orden en el que aparecen los resultados de cada algoritmo. Si el orden es $[1, 2, 3, 4, 5]$, es decir, el que aparece en el recuadro, entonces en cada tabla, la primera linea corresponderá a la partición 1, la segunda a la partición 2, etc.

Para realizar tal ejecución deberemos tener instalado las librerías numpy, tabulate y timeit.

Para ejecutar los algoritmos, ya sean de la práctica 1 o de la práctica 3, debemos hacer una llamada con cada dataset.

5 Experimentos y análisis de resultados

Del análisis realizado de la práctica 1 concluimos lo siguiente:

- 1-NN es más rápido que el resto debido a su sencillez, pero tiene una tasa de reducción de 0 y esto hace que el fitness sea más bajo en comparación con el resto. La tasa de clasificación es similar al resto.
- Relief es más rápido que Búsqueda Local, pero aproximadamente diez veces más lento que 1-NN. Su tasa de reducción es baja y su tasa de clasificación es comparable a otros algoritmos.

- Búsqueda Local es significativamente más lenta en comparación con los otros algoritmos debido a su constante exploración de entornos de soluciones. Su enfoque en mejorar la función objetivo resulta en una mejor tasa fitness.

- El tiempo de ejecución en 1-NN y Relief es mayor en el dataset diabetes debido al mayor número de datos. El tiempo de ejecución en Búsqueda Local depende del número máximo de vecinos a explorar, que a su vez depende del número de características del dataset, por lo tanto, diabetes es el dataset más rápido, seguido por heart y ozone.

- La variabilidad de los resultados es baja en 1-NN y Relief, ya que solo dependen de la partición de los datos. Sin embargo, en Búsqueda Local hay más variabilidad debido a los fenómenos aleatorios involucrados, como la inicialización de la solución y la elección de mutación.

- En Búsqueda Local, el porcentaje de reducción siempre es mayor, lo cual mejora la tasa fitness. En el dataset diabetes, el porcentaje de reducción es especialmente alto debido a la estabilización de los pesos a valores menores que 0.1, lo que maximiza la función objetivo. En los otros datasets, esto no ocurre debido al mayor número de características.

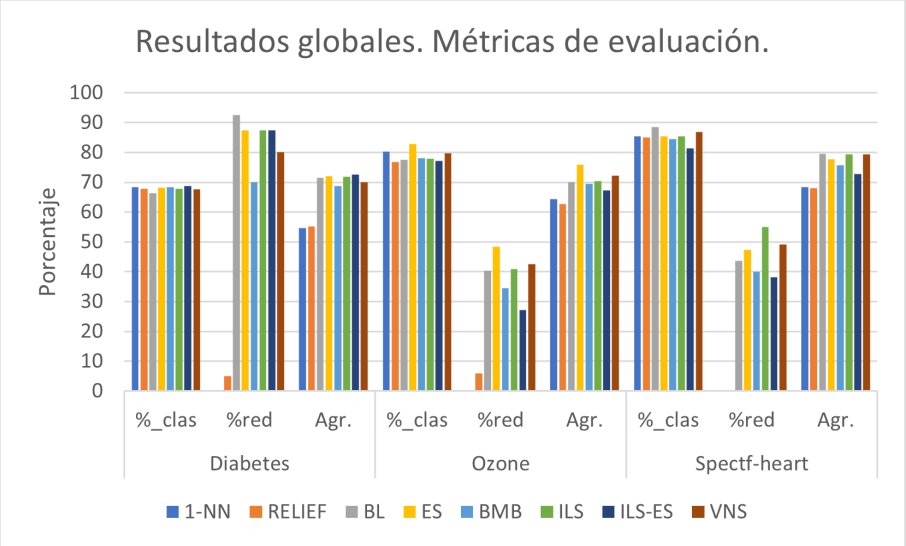
En esta práctica, al igual que en la anterior, el único hiperparámetro utilizado, que no está indicado en el guión, es la semilla, inicializada a 42. Una vez resumido el análisis de la práctica 1, realizamos el análisis de la práctica 3. A continuación vemos una tabla (Tabla 1) y dos gráficas con los resultados globales de los algoritmos (Figura 2).

	Diabetes			Diabetes	Ozone			Ozone	Spectf-			Spectf-
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
1-NN	68,36	0,00	54,69	0,00	80,31	0,00	64,25	0,00	85,38	0,00	68,31	0,00
RELIEF	67,84	5,00	55,27	0,03	76,88	5,83	62,67	0,01	85,08	0,00	68,07	0,01
BL	66,28	92,50	71,53	53,51	77,50	40,28	70,06	108,89	88,54	43,64	79,56	79,83
ES	68,09	87,50	71,98	116,77	82,81	48,33	75,92	22,16	85,39	47,27	77,77	24,47
BMB	68,35	70,00	68,68	5,94	78,13	34,44	69,39	32,19	84,53	40,00	75,62	19,80
ILS	67,82	87,50	71,76	4,08	77,81	40,83	70,42	21,56	85,39	55,00	79,31	11,71
ILS-ES	68,75	87,50	72,50	84,55	77,19	27,22	67,19	10,16	81,38	38,18	72,74	15,03
VNS	67,57	80,00	70,06	4,53	79,69	42,50	72,25	20,18	86,83	49,09	79,28	11,64

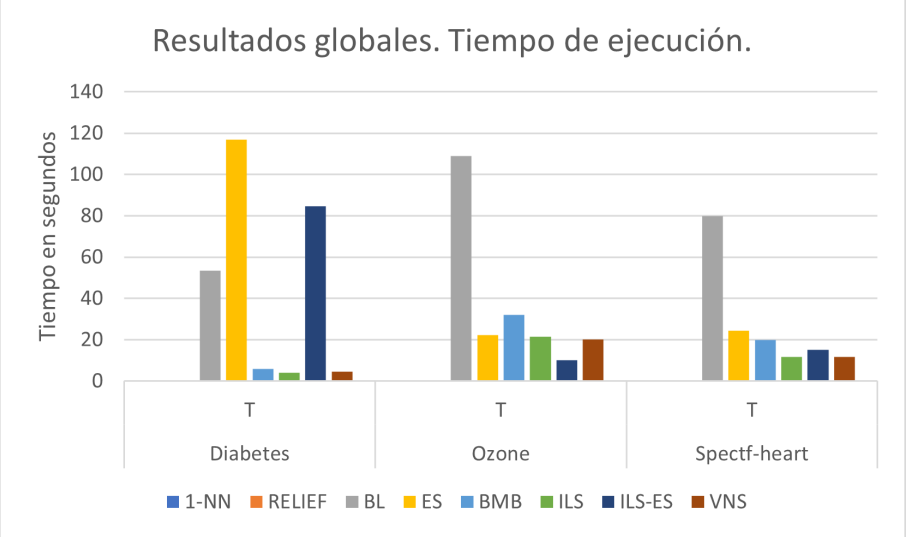
Figure 1: Tabla de los resultados globales obtenidos

En primer lugar, destacar que los tiempos de ejecución de esta práctica son más altos que los de la práctica 1 (exceptuando búsqueda local), pero más bajos que los de la práctica 2. Esto lo podemos ver en el gráfico de la Figura 2b. Búsqueda Local toma más tiempo de ejecución en general, menos en el dataset diabetes, en el que Enfriamiento Simulado tarda más (y por tanto, Búsqueda Local Iterativa con Enfriamiento Simulado). Esto es debido al bajo número de características de diabetes, ya que llegados a cierto punto en el que el fitness es relativamente alto, debe cambiar mucho el valor de la mutación de una característica para mejorar el fitness. En un dataset con mayor número de características hay más probabilidad de que haya características del dataset que no estén optimizadas.

En cuanto a las métricas evaluadas (Figura 2a), vemos que, en general, en diabetes la tasa de clasificación es bastante similar entre algoritmos, en ozone



(a) Tasas.



(b) Tiempo.

Figure 2: Gráficas de los resultados globales obtenidos, divididos en tasas de evaluación y tiempo.

destacan algo más 1-NN, Enfriamiento Simulado y Búsqueda de Vecindario Variable, y en heart destacan Búsqueda Local y Búsqueda de Vecindario Variable. Al haber obtenido resultados similares o mejores Búsqueda de Vecindario Variable en los tres datasets, podemos concluir que este es el mejor algoritmo, en cuanto a tasa de clasificación. Respecto a la tasa de reducción, en diabetes destaca en el algoritmo Búsqueda Local, en ozone destaca Enfriamiento Simulado y en heart destaca Búsqueda Local Iterativa, por tanto, no podemos concluir que un

algoritmo sea mejor que otro en cuanto a esta métrica ya que el comportamiento que presenta es muy distinto en cada dataset. Por último, en cuanto a la tasa fitness, en diabetes están en cabeza Búsqueda Local, Enfriamiento Simulado, Búsqueda Local Iterativa y Búsqueda Local Iterativa con Enfriamiento Simulado, en ozone, Enfriamiento Simulado, y en heart, Búsqueda Local, Búsqueda Local Iterativa y Búsqueda de Vecindario Variable, por tanto, en cuanto a esta métrica, los mejores algoritmos son Búsqueda Local, Enfriamiento Simulado y Búsqueda Local Iterativa.

De estos tres algoritmos, en general, el que tarda menos tiempo es Búsqueda Local Iterativa, por lo que podemos concluir que este es el mejor algoritmo. Por otro lado, los que peores resultados obtienen, en general en los tres datasets, en cuanto a fitness, siempre son 1-NN y Relief, seguidos de Búsqueda Local Iterativa con Enfriamiento Simulado y Búsqueda Multiarranque Básica. Es lógico pensar que Búsqueda Multiarranque Básica sea de los peores de esta práctica ya que realiza optimizaciones sobre soluciones nuevas inicializadas aleatoriamente, mientras que el resto cada iteración parte de la mejor solución.

Ahora realizaremos más en profundidad dos comparaciones naturales: Búsqueda Local vs Enfriamiento Simulado, y Búsqueda Multiarranque Básica vs Búsqueda Local Iterativa vs Búsqueda Local Iterativa con Enfriamiento Simulado vs Búsqueda de Vecindario Variable. He seleccionado estas dos comparaciones ya que en cuanto a código son más similares entre sí.

Empecemos con Búsqueda Local vs Enfriamiento Simulado. En la Tabla 3 podemos ver los resultados obtenidos en las cinco particiones y en la Figura 4 la comparación gráfica:

	Diabetes			Diabetes	Ozone			Ozone	Spectf-heart			Spectf-
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	67,53	87,50	71,53	40,78	79,69	48,61	73,47	108,94	85,71	47,73	78,12	73,27
Partición 2	66,88	87,50	71,01	47,82	76,56	31,94	67,64	108,95	88,57	43,18	79,49	78,24
Partición 3	61,04	87,50	66,33	41,29	85,94	36,11	75,97	108,72	91,43	38,64	80,87	78,11
Partición 4	67,53	100,00	74,03	76,95	62,50	27,78	55,56	108,85	88,57	45,45	79,95	78,29
Partición 5	68,42	100,00	74,74	60,74	82,81	56,94	77,64	109,00	88,41	43,18	79,36	91,25
Media	66,28	92,50	71,53	53,51	77,50	40,28	70,06	108,89	88,54	43,64	79,56	79,83

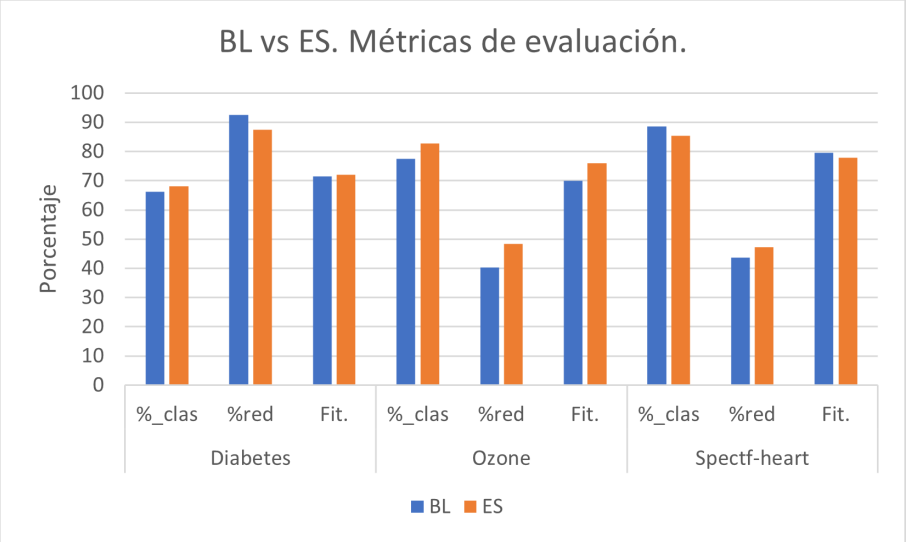
(a) Tabla de los resultados de BL.

	Diabetes			Diabetes	Ozone			Ozone	Spectf-heart			Spectf-
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	71,43	87,50	74,64	135,87	84,38	44,44	76,39	22,72	84,29	52,27	77,88	23,84
Partición 2	66,23	87,50	70,49	103,22	84,38	51,39	77,78	22,02	84,29	43,18	76,06	19,44
Partición 3	68,83	87,50	72,56	99,12	76,56	54,17	72,08	19,29	88,57	50,00	80,86	22,75
Partición 4	67,53	87,50	71,53	103,22	76,56	40,28	69,31	27,15	82,86	50,00	76,29	29,04
Partición 5	66,45	87,50	70,66	142,42	92,19	51,39	84,03	19,64	86,96	40,91	77,75	27,28
Media	68,09	87,50	71,98	116,77	82,81	48,33	75,92	22,16	85,39	47,27	77,77	24,47

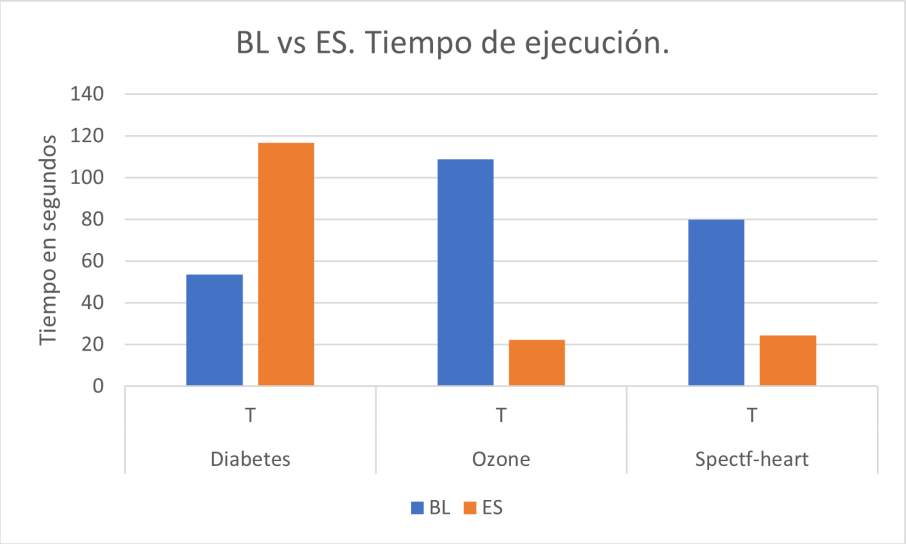
(b) Tabla de los resultados de ES.

Figure 3: Resultados de BL vs ES.

En diabetes, en cuanto a fitness el resultado es empate (Enfriamiento Simulado está por encima, pero muy poco), en ozone supera Enfriamiento Simulado y en heart supera Búsqueda Local, por lo tanto, podríamos decir que es un empate. Sin embargo, en cuanto a tiempo, exceptuando diabetes, Enfriamiento Simulado es mucho más rápido. Por lo tanto, podemos concluir que Enfriamiento Simualado es mejor algoritmo que Búsqueda Local. Esto se debe a que



(a) Tasas.



(b) Tiempo.

Figure 4: Gráficas de BL vs ES.

acepta soluciones peores con cierta probabilidad, y al estar optimizando en un espacio no convexo, a veces, al optimizar sobre peores soluciones puede conducir a obtener mejores resultados.

Veamos que ocurre con la otra comparación. En la Tabla 5 podemos ver los resultados obtenidos por los algoritmos en las cinco particiones, y en la Figura 6 las gráficas extraídas de dichos resultados.

	Diabetes			Diabetes	Ozone			Ozone	Spectf-heart			Spectf-
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	69,48	75,00	70,58	6,18	73,44	44,44	67,64	29,67	77,14	40,91	69,90	16,83
Partición 2	67,53	62,50	66,53	5,81	82,81	37,50	73,75	28,37	88,57	38,64	78,58	20,39
Partición 3	70,13	62,50	68,60	6,22	81,25	33,33	71,67	35,91	85,71	43,18	77,21	19,67
Partición 4	68,83	75,00	70,06	5,16	70,31	30,56	62,36	35,65	87,14	38,64	77,44	18,82
Partición 5	65,79	75,00	67,63	6,35	82,81	26,39	71,53	31,37	84,06	38,64	74,97	23,29
Media	68,35	70,00	68,68	5,94	78,13	34,44	69,39	32,19	84,53	40,00	75,62	19,80

(a) Tabla de los resultados de BMB.

	Diabetes			Diabetes	Ozone			Ozone	Spectf-heart			Spectf-
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	71,43	75,00	72,14	3,84	76,56	51,39	71,53	20,12	78,57	50,00	72,86	10,63
Partición 2	68,83	100,00	75,06	4,49	78,13	40,28	70,56	23,07	90,00	45,45	81,09	9,63
Partición 3	69,48	87,50	73,08	4,11	78,13	40,28	70,56	20,90	90,00	59,09	83,82	12,79
Partición 4	66,88	87,50	71,01	3,80	75,00	44,44	68,89	21,67	82,86	65,91	79,47	13,40
Partición 5	62,50	87,50	67,50	4,15	81,25	27,78	70,56	22,03	85,51	54,55	79,31	12,12
Media	67,82	87,50	71,76	4,08	77,81	40,83	70,42	21,56	85,39	55,00	79,31	11,71

(b) Tabla de los resultados de ILS.

	Diabetes			Diabetes	Ozone			Ozone	Spectf-heart			Spectf-
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	68,18	87,50	72,05	87,96	73,44	36,11	65,97	11,25	75,71	31,82	66,94	14,59
Partición 2	66,23	87,50	70,49	102,42	81,25	22,22	69,44	9,31	78,57	47,73	72,40	16,07
Partición 3	72,08	87,50	75,16	78,25	78,13	26,39	67,78	11,09	91,43	31,82	79,51	14,28
Partición 4	70,13	87,50	73,60	77,67	73,44	23,61	63,47	10,09	78,57	36,36	70,13	15,62
Partición 5	67,11	87,50	71,18	76,47	79,69	27,78	69,31	9,07	82,61	43,18	74,72	14,58
Media	68,75	87,50	72,50	84,55	77,19	27,22	67,19	10,16	81,38	38,18	72,74	15,03

(c) Tabla de los resultados de ILS-ES.

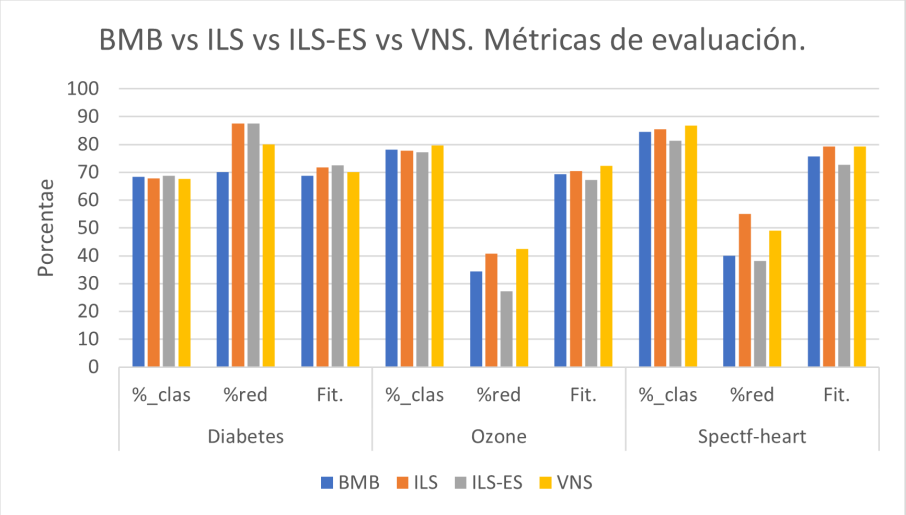
	Diabetes			Diabetes	Ozone			Ozone	Spectf-heart			Spectf-
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
Partición 1	68,18	62,50	67,05	4,96	78,13	43,06	71,11	20,82	84,29	43,18	76,06	10,92
Partición 2	69,48	75,00	70,58	4,31	79,69	41,67	72,08	21,64	85,71	47,73	78,12	11,34
Partición 3	67,53	87,50	71,53	4,54	84,38	38,89	75,28	20,80	92,86	45,45	83,38	12,02
Partición 4	66,88	87,50	71,01	4,59	70,31	44,44	65,14	18,15	81,43	54,55	76,05	11,57
Partición 5	65,79	87,50	70,13	4,23	85,94	44,44	77,64	19,47	89,86	54,55	82,79	12,33
Media	67,57	80,00	70,06	4,53	79,69	42,50	72,25	20,18	86,83	49,09	79,28	11,64

(d) Tabla de los resultados de VNS.

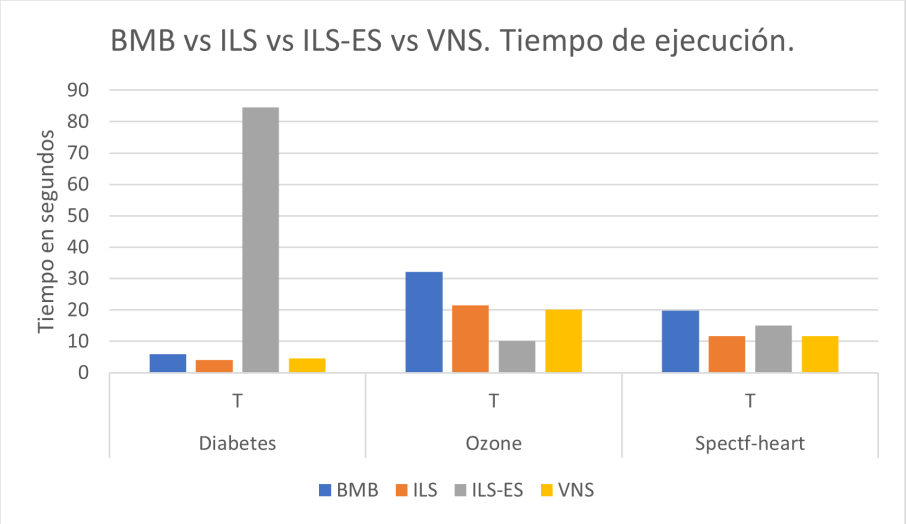
Figure 5: Resultados de BMB vs ILS vs ILS-ES vs VNS.

Respecto a fitness no hay una tendencia clara en los datasets en cuanto a mejor algoritmo. En cuanto a peor algoritmo, destacar de nuevo que Búsqueda Multiarranque Básica se queda por detrás en cuanto a resultados. Destacar también que el siguiente peor algoritmo es Búsqueda Local Iterativa con Enfriamiento Simulado, aunque en diabetes haya conseguido el valor más alto. Por tanto, quedan como mejores algoritmos, empatados, Búsqueda Local Iterativa y Búsqueda de Vecindario Variable. En cuanto a tiempo de ejecución, ambos tienen un tiempo similar, luego no es concluyente para poder decidir un ganador. Destacar que Búsqueda de Vecindario Variable obtiene tan buenos resultados porque en la mutación a veces se mutan a la vez más de una característica, lo que aceleraría el proceso de búsqueda local.

Por otro lado, de esta misma gráfica podemos comparar las dos variantes de Búsqueda Local Iterativa presentadas. Destacar que aunque Enfriamiento Simulado sea mejor que una simple Búsqueda Local, esto no ocurre cuando el procedimiento es iterativo. En base a los resultados podemos concluir que cuando el número de iteraciones es menor, es mejor utilizar una Búsqueda Local simple, ya que cuando es relativamente fácil mejorar el fitness de nada sirve aceptar soluciones peores, pero cuando se trata de un número de iteraciones mayor, es mejor enfriamiento simulado ya que cuando el valor de fitness está rozando el óptimo, aceptar esporádicamente soluciones peores puede ayudar a



(a) Tasas.



(b) Tiempo.

Figure 6: Gráficas de los Algoritmos Genéticos Estacionarios.

mejorar el fitness final.

Como conclusión de esta práctica, en cuanto a la práctica 1, hemos mejorado resultados. En la práctica 1, el algoritmo ganador era búsqueda local, aunque tardaba bastante más que el resto. En esta práctica, hemos conseguido resultados mejores y empleando menos tiempo de ejecución. Luego podemos concluir que los algoritmos presentados en esta práctica son mejores.