UNIVERSIDAD DE GRANADA

Práctica 1.b:
Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Aprendizaje de Pesos en Características (APC)

Brao Aissaoni, Isaac
Grupo de prácticas 3 Jueves(17:30-19:30)

Índice

1. Descripción del problema: Problema del Aprendizaje de Pesos en Características (APC)..... 3
2. Funciones comunes a ambos algoritmos..... 3
2.1 Codificación de los datos del problema.....3
2.1.1 Representación del problema..... 3
2.1.2 Representación de la solución..... 3
2.2 Función objetivo..... 4
2.3 Cálculo de distancias..... 5
2.4 Cálculo de la tasa de clasificación..... 6
2.5 Lectura, preprocesamiento de datos y Five Cross Validation..... 7
2.5.1 Five Cross Validation.....8
2.5.2 Normalización de los datos con un escalado MinMax..... 9
3. Algoritmo Greedy.....9
3.1 Obtención de pesos mediante el método Relief.....9
3.2 Obtención de los amigos y enemigos..... 10
3.2.1 Obtención de amigos..... 10
3.2.2 Obtención de enemigos..... 10
4. Algoritmo de Búsqueda Local..... 11
4.1 Obtención de los pesos mediante la Búsqueda Local.....11
4.2 Generación de vecinos.....12
5. Procedimiento de desarrollo de la práctica.....12
5.1 Software utilizado para el desarrollo del código fuente.....12
5.2 Manual de usuario.....12
5.3 Consideraciones a tener en cuenta.....13
6. Experimentos y análisis de los resultados.....14
6.1 Experimentos y resultados.....14
6.1.1 1NN..... 14
6.1.1.1 Ecoli.....14
6.1.1.2 Parkinsons.....14
6.1.1.3 Breast Cancer.....15
6.1.2 Greedy Relief..... 15
6.1.2.1 Ecoli.....15
6.1.2.2 Parkinsons.....16
6.1.2.3 Breast Cancer.....16
6.1.3 BL..... 17
6.1.3.1 Ecoli.....17
6.1.3.2 Parkinsons.....17
6.1.3.3 Breast Cancer.....18
6.1.4 Medias.....18
6.2 Análisis de resultados..... 19

1. Descripción del problema: Problema del Aprendizaje de Pesos en Características (APC)

El problema consiste en la realización de la clasificación de datos en clases que ya son conocidas. La pertenencia de cada dato a una clase ya se conoce a priori.

El objetivo es que el modelo de clasificación sea capaz de clasificar los datos correctamente con un error mínimo, además de aprender la importancia de cada característica que describe a los datos. Para ello debe asignar distinta importancia a cada característica a la que le es asignada un peso distinto, el cual se tendrá en cuenta al momento de clasificar.

2. Funciones comunes a ambos algoritmos

2.1 Codificación de los datos del problema

2.1.1 Representación del problema

El problema se representa con 5 matrices $n_i$xm donde n_i es el número de datos de cada partición del problema y m es el número de característica.

$$X_i = \begin{pmatrix} x_{i11} & \cdots & x_{i1m} \\ \cdots & \cdots & \\ x_{in_i1} & \cdots & x_{in_im} \end{pmatrix}$$

Donde i corresponde al número de la partición.

2.1.2 Representación de la solución

Los pesos están representados en un vector de tamaño m donde m es el número de características

$$w_i = \begin{pmatrix} w_{i1} & \cdots & w_{im} \end{pmatrix}$$

Donde i corresponde al número de la partición.

2.2 Función objetivo

La función fitness tendrá en cuenta la tasa de acierto sobre el conjunto de entrenamiento y la tasa de reducción de los pesos. Buscamos maximizar el fitness.

$$fitness = \alpha \cdot \%aciertos + (1 - \alpha) \cdot \%reducción \quad donde\, \alpha = 0.75\, en\, los\, problemas$$

```python
def fitness(X_train, y_train, X_test, y_test, pesos, alfa=0.75):
    predicciones = clasificador1NN(X_train, y_train, X_test, pesos)
    aciertos = accuracy(y_test, predicciones)
    tasa_red = tasa_reduccion(pesos)
    return alfa*aciertos + (1-alfa)*tasa_red
```

```python
def tasa_reduccion(pesos):
    cont = 0
    for p in pesos:
        if p >= 0.1:
            cont += 1

    return 100*(cont / len(pesos))
```

```python
def accuracy(y_true, y_pred):
    correctos = 0

    for i in range(len(y_true)):
        if y_true[i] == y_pred[i]:
            correctos += 1

    return correctos / len(y_true)
```

```python
def clasificador1NN(X_train, y_test, X_test, pesos=None, k=1):
    # Inicializar pesos a 1 si no se especifican simulando un KNN normal
    if pesos is None:
        pesos = []
        for i in range(n_columnas(X_train)):
            pesos.append(1)

    predictions = np.array(n_columnas(X_test), dtype=str)

    # Calcular matriz de distancias
    for i in range(n_filas(X_test)):
        distancias = np.zeros(n_filas(X_train))
        for j in range(n_filas(X_train)):
            distancias[j] = distanciaEuclidiana(X_test[i], X_train[j], pesos)

        indices_ord = ordenarPorDistancia(distancias, eje='fila')

        # Si elegimos el numero 1, el vecino más cercano es el segundo porque el primero es el mismo
        indice_cercano = indices_ord[1]

        # Asignar la clase del vecino más cercano
        clase = y_test[indice_cercano]

        # Asignar la clase al conjunto de predicciones
        predictions[i] = clase

    return predictions
```

2.3 Cálculo de distancias

Usamos la distancia euclídea con pesos en el BL y Greedy. Para el clasificador 1NN estándar ignoramos los pesos, es decir ponemos todos los pesos a 1.

$$d(A, B) = \sqrt{\sum_i w_i (A_i - B_i)^2},$$

```python
def distanciaEuclidiana(x1, x2, pesos):
    suma = 0
    for i in range(n_columnas(x1)):
        # Si el peso es menor o igual a 0.1, no se tiene en cuenta
        if pesos[i] <= 0.1:
            continue

        diff = x1[i] - x2[i]
        diff = diff ** 2
        suma += diff * pesos[i]

    return sqrt(suma)

def matrizCuadradaDistancias(X, pesos):
    distancias = np.zeros((n_filas(X), n_filas(X)))

    for i in range(n_filas(X)):
        for j in range(n_filas(X)):
            distancias[i][j] = distanciaEuclidiana(X[i], X[j], pesos)

    return distancias
```

2.4 Cálculo de la tasa de clasificación

```python
def tasa_clasificacion(X, y, pesos):
    # Calcular matriz de distancias
    distancias = matrizCuadradaDistancias(X, pesos)
    # Ordenar indices de distancias
    indices_ord = ordenarPorDistancia(distancias, eje='fila')

    # Inicializar tasa de clasificacion
    tasa_clasificacion = 0

    for i in range(n_filas(X)):
        # Como los indices_ord estan ordenados, el mas cercano es el primero
        # Hay que ignorar el indice 0 porque es el mismo
        mas_cercano = indices_ord[i][1]

        if y[i] == y[mas_cercano]:
            tasa_clasificacion += 1

    return 100*(tasa_clasificacion / n_filas(X))
```

2.5 Lectura, preprocesamiento de datos y Five Cross Validation

```python
def main():
    print('Introduce el nombre del conjunto de datos:')
    cadena = input()

    todos_los_modelos = ['KNN', 'Relief', 'BL', 'ALL']
    print('Introduce el modelo a utilizar [KNN, Relief, BL, ALL]:')
    model_type = input().upper()

    if model_type == 'KNN' or model_type == 'ALL':
        print('Introduce el valor de k:')
        k = int(input())

    if model_type == 'BL' or model_type == 'ALL':
        print('Introduce el valor de la semilla [DEFAULT=7]:')
        seed_i = input()
        if seed_i == '':
            seed = 7
        else:
            seed = int(seed_i)

    # Cargar los 5 conjuntos de datos
    data1 = leer_arff('./data/'+cadena+'_1.arff')
    data2 = leer_arff('./data/'+cadena+'_2.arff')
    data3 = leer_arff('./data/'+cadena+'_3.arff')
    data4 = leer_arff('./data/'+cadena+'_4.arff')
    data5 = leer_arff('./data/'+cadena+'_5.arff')

    # Separar los datos en características y etiquetas
    X1 = np.array(data1[:, :-1], dtype=float)
    y1 = data1[:, -1]

    X2 = np.array(data2[:, :-1], dtype=float)
    y2 = data2[:, -1]

    X3 = np.array(data3[:, :-1], dtype=float)
    y3 = data3[:, -1]

    X4 = np.array(data4[:, :-1], dtype=float)
    y4 = data4[:, -1]

    X5 = np.array(data5[:, :-1], dtype=float)
    y5 = data5[:, -1]

    # Normalizar los datos

    X = np.concatenate((X1, X2, X3, X4, X5), axis=0)

    X = escaladoMinmax(X)

    X1 = X[:n_filas(X1)]
    X2 = X[n_filas(X1):n_filas(X1)+n_filas(X2)]
    X3 = X[n_filas(X1)+n_filas(X2):n_filas(X1)+n_filas(X2)+n_filas(X3)]
    X4 = X[n_filas(X1)+n_filas(X2)+n_filas(X3):n_filas(X1)+n_filas(X2)+n_filas(X3)+n_filas(X4)]
    X5 = X[n_filas(X1)+n_filas(X2)+n_filas(X3)+n_filas(X4):]

    # Five cross validation
    modelos_a_utilizar = [model_type] if model_type != 'ALL' else todos_los_modelos

    for modelo in modelos_a_utilizar:
        resultados = fiveCrossValidation(X1, X2, X3, X4, X5, y1, y2, y3, y4, y5, modelo, seed, k)

        print(f'Modelo: {modelo}')

        for res in resultados:
            print(res)
```

2.5.1 Five Cross Validation

```python
def fiveCrossValidation(X1, X2, X3, X4, X5, y1, y2, y3, y4, y5, model_type, seed=7, k=1):
    resultados = []
    np.random.seed(seed)
    for i in range(5):
        # Unir los conjuntos de datos
        if(i == 0):
            X_train = np.concatenate((X2, X3, X4, X5))
            y_train = np.concatenate((y2, y3, y4, y5))
            X_test = X1
            y_test = y1
        elif(i == 1):
            X_train = np.concatenate((X1, X3, X4, X5))
            y_train = np.concatenate((y1, y3, y4, y5))
            X_test = X2
            y_test = y2
        elif(i == 2):
            X_train = np.concatenate((X1, X2, X4, X5))
            y_train = np.concatenate((y1, y2, y4, y5))
            X_test = X3
            y_test = y3
        elif(i == 3):
            X_train = np.concatenate((X1, X2, X3, X5))
            y_train = np.concatenate((y1, y2, y3, y5))
            X_test = X4
            y_test = y4
        elif(i == 4):
            X_train = np.concatenate((X1, X2, X3, X4))
            y_train = np.concatenate((y1, y2, y3, y4))
            X_test = X5
            y_test = y5

        # Entrenar el modelo

        time_start = time.time()

        pesos = np.array(n_columnas(X_train))

        if model_type == 'KNN':
            pesos = np.ones(n_columnas(X_train))
        elif model_type == 'Relief':
            pesos = fit_Relief(X_train, y_train)
        elif model_type == 'BL':
            pesos = fit_BL(X_train, y_train, max_evaluaciones=15000, semilla=np.random.randint(0, 1000))
        else:
            raise ValueError("El modelo no es válido.")

        # Evaluar el modelo

        tasa_red = tasa_reduccion(pesos)

        tasa_clas = tasa_clasificacion(X_train, y_train, pesos)

        fitness_train = fitness(X_train, y_train, pesos, 0.75)

        acierto = accuracy(y_test, clasificador1NN(X_train, y_test, X_test, pesos, k))

        fitness_test = fitness2(acierto, tasa_red, 0.75)

        time_end = time.time()

        total_time = time_end - time_start

        resultados.append((tasa_red, tasa_clas, fitness_train, acierto, fitness_test, total_time))

    return resultados
```

2.5.2 Normalización de los datos con un escalado MinMax

```
def escaladoMinmax(X):
    minimo = np.min(X, axis='col')
    maximo = np.max(X, axis='col')

    return (X - minimo) / (maximo - minimo)
```

3. Algoritmo Greedy

3.1 Obtención de pesos mediante el método Relief

```
def fit_Relief(X, y):
    # Inicializar pesos a 0 con la cantidad de columnas de X
    pesos = []

    for i in range(n_columnas(X)):
        pesos.append(0)

    # Calcular matriz de distancias
    distancias = matrizCuadradaDistancias(X, pesos)
    distancias[distancias == 0] = np.inf
    # Ordenar indices de distancias
    indices_ord = ordenarPorDistancia(distancias, eje='fila')

    for i in range(n_filas(X)):
        amigo_cercano = -1

        # Filtrar amigos y quitamos los que tengan distancia 0
        amigos = filtrarAmigos(i, indices_ord, y)

        # Si hay amigos, el amigo cercano es el primero
        if len(amigos) > 0:
            amigo_cercano = amigos[0]

        # Filtrar enemigos
        enemigo_cercano = filtrarEnemigos(i, indices_ord, y)

        # Si hay amigos, actualizamos los pesos
        if amigo_cercano != -1 and distancias[i][amigo_cercano] != np.inf:
            pesos += abs(X[i] - X[amigo_cercano]) - abs(X[i] - X[enemigo_cercano])

    # Normalizar pesos y ponerlos en el rango [0, 1]
    maximo  = valor_maximo(pesos)

    for p in pesos:
        p = max(p/maximo, 0)

    return pesos
```

Para actualizar los pesos se utiliza la siguiente fórmula

$$\text{pesos} = \text{pesos} + \left(\left|X[i] - X[\text{amigo\_cercano}]\right| - \left|X[i] - X[\text{enemigo\_cercano}]\right|\right)$$

3.2 Obtención de los amigos y enemigos

3.2.1 Obtención de amigos

```
def filtrarAmigos(i, indices_ord, y):
    amigos = []

    for j in range(n_filas(indices_ord)):
        if y[indices_ord[i]] == y[indices_ord[j]]:
            amigos.append(indices_ord[j])

    return amigos
```

3.2.2 Obtención de enemigos

```
def filtrarEnemigos(i, indices_ord, y):
    enemigos = []

    for j in range(n_filas(indices_ord)):
        if y[indices_ord[i]] != y[indices_ord[j]]:
            enemigos.append(indices_ord[j])

    return enemigos
```

4. Algoritmo de Búsqueda Local

4.1 Obtención de los pesos mediante la Búsqueda Local

```python
def fit_BL(X, y, max_evaluaciones, semilla=7):
    # Semilla para inicializar pesos aleatorios
    np.random.seed(semilla)

    # Incializar pesos con una distribución uniforme con el tamaño de columnas de X
    pesos = np.random.uniform(0, 1, n_columnas(X))

    # Maximo de iteraciones de veces en la que toleramos no mejorar
    MAX_ITER = 20*n_columnas(X)

    # Alfa para el fitness
    ALFA = 0.75

    # Evaluacion actual
    fitness_actual = fitness(X, y, pesos, ALFA)

    n_evaluaciones = 0
    iteraciones_sin_mejora = 0

    while n_evaluaciones < max_evaluaciones and iteraciones_sin_mejora < MAX_ITER:
        # Obtener un orden de mutacion aleatorio
        orden_mutacion = np.random.permutation(len(pesos))

        for mut in orden_mutacion:
            # Obtener un vecino
            vecino = obtenerVecino(pesos, mut)

            # Calcular fitness del vecino
            fitness_vecino = fitness(X, y, vecino, ALFA)
            n_evaluaciones += 1

            # Si el vecino es mejor, actualizamos pesos
            if fitness_vecino > fitness_actual:
                pesos = vecino
                fitness_actual = fitness_vecino
                iteraciones_sin_mejora = 0
            else: # Si no, aumentamos el contador de iteraciones sin mejora
                iteraciones_sin_mejora += 1

    return pesos
```

4.2 Generación de vecinos

```python
def obtenerVecino(pesos, i):
    #Obtener una mutacion aleatoria de la distribucion normal de media 0 y varianza 0.3
    mutacion = np.random.normal(0, np.sqrt(0.3))
    vecino = pesos.copy()

    # Aplicar la mutacion
    vecino[i] += mutacion

    # Normalizar el peso cambiado
    vecino[i] = max(vecino[i], 0)
    vecino[i] = min(vecino[i], 1)

    return vecino
```

5. Procedimiento de desarrollo de la práctica

5.1 Software utilizado para el desarrollo del código fuente

La práctica se ha realizado en el lenguaje de programación python. Se ha usado **numpy** para las operaciones vectoriales y matriciales; **scipy** para el cálculo de distancias euclidianas, con peso y sin pesos para el KNN; **scikit learn** para la normalización y escalado de los datos; **pandas** para las tablas, estadísticas y guardar la salida en formato csv y el paquete **liac-arff** para la lectura de los datos.

5.2 Manual de usuario

1. **Inicio del Programa:**

   Al iniciar el programa, verás el siguiente mensaje:

   Introduce la base de datos a utilizar [Opciones: BreastCancer[DEFAULT][1], Ecoli[2], Parkinson[3]]:

   Debes introducir el nombre de la base de datos que deseas utilizar. Las opciones son BreastCancer, Ecoli y Parkinson. También puedes seleccionar simplemente ingresando el número correspondiente a la base de datos deseada.

2. **Selección del Modelo:**

   Después de seleccionar la base de datos, el programa te pedirá que elijas el modelo a utilizar con el siguiente mensaje:

   Elige el modelo a utilizar [Opciones: KNN[DEFAULT][1], Relief[2], BL[3], ALL[4]]:

Puedes seleccionar el modelo escribiendo su nombre completo o simplemente ingresando el número correspondiente al modelo deseado. Las opciones disponibles son KNN, Relief, BL y ALL. Si eliges ALL, se utilizarán todos los modelos disponibles.

3. **Configuración Adicional según el Modelo:**

   Dependiendo del modelo seleccionado, es posible que se soliciten configuraciones adicionales:
   - **Para KNN:**
     - Se te pedirá que introduzcas el valor de k, que representa el número de vecinos a considerar. Por defecto es 1, representando el 1NN.

   - **Para BL:**
     - Se te pedirá que introduzcas el valor de la semilla que se utilizará en el proceso. Por defecto es 7.

4. **Guardar Resultados en Archivo CSV:**

   Después de seleccionar el modelo y configurar las opciones adicionales (si es necesario), se te preguntará si deseas guardar los resultados en un archivo CSV con el mensaje:

   Guardar resultados en archivo CSV [S/N][DEFAULT=N]:

   Puedes seleccionar S para guardar los resultados en un archivo CSV o N para no guardarlos.

5. **Requisitos del Programa:**

   Este programa requiere los siguientes paquetes de Python instalados en tu sistema:
   - **numpy**
   - **scipy**
   - **pandas**
   - **liac-arff**
   - **scikit-learn**

6. **Errores y Salida del Programa:**

   Si introduces una base de datos o modelo no válido, el programa mostrará un mensaje de error correspondiente y se cerrará.

5.3 Consideraciones a tener en cuenta

1. Si tiene instalado el paquete arff de google tendrá que desinstalarlo e instalar a continuación liac-arff porque generan incompatibilidad entre ellos.
2. Los valores utilizados para el análisis de los resultados han sido k 1 y semilla 7.

6. Experimentos y análisis de los resultados

6.1 Experimentos y resultados

Los experimentos han sido realizados por el parámetro k = 1 (KNN) y con la semilla igual a 7 (BL)

6.1.1 1NN

6.1.1.1 Ecoli

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	81.20300752	0	60.90225564	60	80	1.40E-03
2	83.45864662	0	62.59398496	54.64285714	72.85714286	5.43E-04
3	80.97014925	0	60.72761194	61.76470588	82.35294118	5.56E-04
4	77.98507463	0	58.48880597	62.86764706	83.82352941	5.04E-04
5	80.79710145	0	60.59782609	63.75	85	6.55E-04
Media	80.88279589	0	60.66209692	60.60504202	80.80672269	7.32E-04
Desviación típica	1.946105413	0	1.45957906	3.614854315	4.819805754	3.80E-04

6.1.1.2 Parkinsons

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	95.48387097	0	71.61290323	73.125	97.5	7.89E-04
2	94.19354839	0	70.64516129	65.625	87.5	4.14E-04
3	96.12903226	0	72.09677419	73.125	97.5	3.94E-04
4	94.19354839	0	70.64516129	75	100	3.76E-04
5	96.875	0	72.65625	68.571428	91.428571	3.92E-04

				57	43	
Media	95.375	0	71.53125	71.08928571	94.78571429	4.73E-04
Desviación típica	1.185545588	0	0.889159190 7	3.865875531	5.154500708	1.77E-04

6.1.1.3 Breast Cancer

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	95.15418502	0	71.36563877	73.04347826	97.39130435	7.51E-03
2	94.9339207	0	71.20044053	72.39130435	96.52173913	4.86E-03
3	96.91629956	0	72.68722467	69.13043478	92.17391304	3.69E-03
4	93.83259912	0	70.37444934	73.69565217	98.26086957	5.78E-03
5	95.43478261	0	71.57608696	70.18348624	93.57798165	4.04E-03
Media	95.2543574	0	71.44076805	71.68887116	95.58516155	5.18E-03
Desviación típica	1.110035561	0	0.832526670 9	1.947262391	2.596349855	1.53E-03

6.1.2 Greedy Relief

6.1.2.1 Ecoli

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	79.32330827	28.57142857	66.63533835	65	77.14285714	9.94E-03
2	82.33082707	28.57142857	68.89097744	62.85714286	74.28571429	8.43E-03
3	79.47761194	28.57142857	66.751066 1	66.701680 67	79.411764 71	9.19E-03
4	78.35820896	28.57142857	65.911513 86	67.804621 85	80.882352 94	9.58E-03

5	79.34782609	28.57142857	66.65372671	72.14285714	86.66666667	8.56E-03
Media	79.76755646	28.57142857	66.96852449	66.9012605	79.67787115	9.14E-03
Desviación típica	1.501184071	0	1.125888053	3.474906768	4.633209025	6.50E-04

6.1.2.2 Parkinsons

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	96.12903226	0	72.09677419	73.125	97.5	4.06E-03
2	95.48387097	0	71.61290323	67.5	90	3.57E-03
3	96.77419355	0	72.58064516	71.25	95	3.39E-03
4	94.19354839	0	70.64516129	75	100	3.77E-03
5	96.875	0	72.65625	68.57142857	91.42857143	3.92E-03
Media	95.89112903	0	71.91834677	71.08928571	94.78571429	3.74E-03
Desviación típica	1.101367575	0	0.826025681 5	3.109912559	4.146550078	2.66E-04

6.1.2.3 Breast Cancer

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	94.71365639	3.333333333	71.86857562	73.87681159	97.39130435	2.75E-02
2	94.27312775	3.333333333	71.53817915	73.87681159	97.39130435	2.45E-02
3	98.23788546	13.33333333	77.01174743	70.50724638	89.56521739	2.40E-02
4	94.05286344	0	70.53964758	73.04347826	97.39130435	2.52E-02
5	96.08695652	0	72.065217	70.871559	94.495412	2.49E-02

			39	63	84	
Media	95.47289791	4	72.60467343	72.43518149	95.24690866	2.52E-02
Desviación típica	1.735988059	5.477225575	2.532631826	1.634661034	3.414734991	1.35E-03

6.1.3 BL

6.1.3.1 Ecoli

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	80.82706767	57.14285714	74.90601504	72.14285714	77.14285714	9.46E-02
2	79.32330827	57.14285714	73.77819549	70	74.28571429	6.58E-02
3	73.88059701	71.42857143	73.26759062	67.4894958	66.17647059	8.53E-02
4	77.98507463	57.14285714	72.77452026	73.84453782	79.41176471	8.32E-02
5	73.91304348	71.42857143	73.29192547	71.60714286	71.66666667	4.51E-02
Media	77.18581821	62.85714286	73.60364937	71.01680672	73.73669468	7.48E-02
Desviación típica	3.166303594	7.824607964	0.809966827 2	2.4025178 3	5.136920024	1.96E-02

6.1.3.2 Parkinsons

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	94.19354839	81.81818182	91.09970674	89.82954545	92.5	2.57E-01
2	96.12903226	77.27272727	91.41495601	90.56818182	95	1.81E-01
3	97.41935484	77.27272727	92.38269795	90.56818182	95	1.70E-01
4	98.70967742	77.27272727	93.350439	86.818181	90	2.37E-01

			88	82		
5	98.125	77.27272727	92.91193182	83.6038961	85.71428571	1.92E-01
Media	96.91532258	78.18181818	92.23194648	88.2775974	91.64285714	2.08E-01
Desviación típica	1.800235019	2.03278907	0.9598924158	3.03625898	3.90904236	3.76E-02

6.1.3.3 Breast Cancer

Partición	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1	96.03524229	86.66666667	93.69309838	92.10144928	93.91304348	1.62E+00
2	96.03524229	86.66666667	93.69309838	92.10144928	93.91304348	2.44E+00
3	98.01762115	76.66666667	92.67988253	86.99275362	90.43478261	1.35E+00
4	96.47577093	90	94.85682819	90.97826087	91.30434783	2.57E+00
5	96.73913043	80	92.55434783	90.87155963	94.49541284	1.57E+00
Media	96.66060142	84	93.49545106	90.60909454	92.81212605	1.91E+00
Desviación típica	0.8160851264	5.477225575	0.9330466175	2.10578596	1.81539549	5.55E-01

6.1.4 Medias

	Ecoli					
	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1NN	80.88	0.00	60.66	60.61	80.81	7.32E-04
Relief	79.77	28.57	66.97	66.90	79.68	9.14E-03
BL	77.19	62.86	73.60	71.02	73.74	7.48E-02
	Parkinsons					
	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1NN	95.38	0.00	71.53	71.09	94.79	4.73E-04
Relief	95.89	0.00	71.92	71.09	94.79	3.74E-03
BL	96.92	78.18	92.23	88.28	91.64	2.08E-01
	Breast Cancer					
	Tasa de clasificación	Tasa de reducción	Fitness train	Fitness test	Accuracy	Tiempo de ejecución
1NN	95.25	0.00	71.44	71.69	95.59	5.18E-03
Relief	95.47	4.00	72.60	72.44	95.25	2.52E-02
BL	96.66	84.00	93.50	90.61	92.81	1.91E+00

6.2 Análisis de resultados

El 1NN por lo general es el que más porcentaje de acierto tiene y el que menor tiempo de ejecución tiene. Pero tiene un problema, que no reduce nada por lo que el fitness es el más bajo de los tres.

El método greedy también es bueno en acierto, pero tampoco reduce lo suficiente, por lo que el fitness aumenta ligeramente al haber reducido algo y aumentar ligeramente el acierto.

La búsqueda local reduce mucho, pero es el que más tarda con diferencia, aprox 2s en breast cancer frente a los milisegundos que tarda el greedy. En cuanto a fitness, es el mejor porque es el que más reduce sin comprometer tanto la tasa de acierto y tasa de clasificación. Pero parece tener un problema de sobreajuste al haber tanta diferencia entre la tasa de clasificación y la de acierto.

En conclusión la búsqueda local es la mejor al aprender pesos de características al tener mejor fitness y mayor reducción, pero a cambio tarda más y parece tener un ligero problema de sobreajuste a los datos de entrenamiento.