Práctica 1.b:

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

Curso: 2018 - 2019
Alumno: Christian Vigil Zamora
DNI: [redacted]
Correo: [redacted]
Grupo: A2: Miércoles 17:30 - 19:30

Índice

1. DESCRIPCIÓN DEL PROBLEMA **2**

2. APLICACIÓN DE LOS ALGORITMOS EMPLEADOS AL PROBLEMA **3**

3. DESCRIPCIÓN EN PSEUDOCÓDIGO DE LA ESTRUCTURA DEL MÉTODO DE BÚSQUEDA **6**

4. DESCRIPCIÓN EN PSEUDOCÓDIGO DE LOS ALGORITMOS DE COMPARACIÓN **8**

5. BREVE EXPLICACIÓN DEL PROCEDIMIENTO CONSIDERADO PARA DESARROLLAR LA PRÁCTICA **11**

6. EXPERIMENTOS Y ANÁLISIS DE RESULTADOS **12**

7. REFERENCIAS BIBLIOGRÁFICAS **17**

1. DESCRIPCIÓN DEL PROBLEMA

El problema del Aprendizaje de Pesos en Características tiene como objetivo ajustar un conjunto de pesos asociados al conjunto total de características, con la finalidad de obtener un clasificador que de mejores resultados. Los datos de nuestro problema estarán formados por un conjunto de elementos, los cuáles tienen asociados a su vez un conjunto de características.
Para entrenar a nuestro clasificador recurrimos a la técnica de validación cruzada **5-fold cross validation**, con la que el conjunto de datos total se divide en 5 particiones manteniendo la distribución de clases equilibrada. Nuestro clasificador se entrenará con con la concatenación de 4 particiones, y será evaluado con la partición restante. En nuestro caso, vamos a usar un clasificador muy simple, el 1-NN cuyo funcionamiento está basado en la Regla del vecino más próximo, por lo que vamos a tener que calcular la distancia entre dos elementos e ir comparando el valor obtenido. Para realizar el cálculo de la distancia entre dos elementos recurrimos a la **distancia euclídea**, que viene dada por la fórmula:

$$d_e(e_1, e_2) = \sqrt{\sum_i w_i \cdot (e_1^i - e_2^i)^2 + \sum_j w_j \cdot d_h(e_1^j, e_2^j)}$$

Siendo Wi un elemento del vector de pesos asociado a cada características, representando un valor real en [0, 1].
Para analizar el rendimiento del clasificador utilizamos una **función de evaluación**:

$$F(w) = \alpha \cdot tasa - clas(w) + (1 - \alpha) \cdot tasa - red(w)$$

que viene a ser una combinación de la **tasa_clas** (porcentaje de instancias correctamente clasificadas pertenecientes a un conjunto de datos, T):

$$tasa - clas = 100 \cdot \frac{n^o \text{ instancias bien clasificadas en } T}{n^o \text{ instancias en } T}$$

de la **tasa_red** (porcentaje de características descartadas):

$$tasa - red = 100 \cdot \frac{n^o \text{ valores } wi < 0.2}{n^o \text{ características}}$$

y de $\alpha \varepsilon [0, 1]$ (pondera la importancia entre el acierto y la reducción de la solución encontrada), teniendo como objetivo obtener el conjunto de pesos W que maximiza esta función.

2. APLICACIÓN DE LOS ALGORITMOS EMPLEADOS AL PROBLEMA

Operadores comunes y Función Objetivo

- **euclidean_distance** : Función que calcula la distancia euclídea entre 2 elementos mediante la sumatoria de la resta de las características de un elemento con otro, al cuadrado. Para reducir el tiempo de cálculo, he suprimido el cálculo de la raíz cuadrada. "np.sum((elem1 - elem2)**2)" devuelve la sumatoria de lo anterior comentado, directamente. Pseudocódigo:

```
Function euclidean_distance (elem1,elem2):
    distance = 0.0
    distance = np.sum((elem1 - elem2)**2)
    Return distance
```

- **closest_examples** : Función que dado un ejemplo, devuelve su amigo más cercano (distancia mínima entre el ejemplo y él, y ambos comparten clase) y su enemigo más cercano (distancia mínima entre el ejemplo y él, y ambos con clase diferente). "**np.inf**" otorga a una variable el valor infinito. Su funcionamiento se basa en que por cada elemento de los datos, se calcula la distancia entre ese elemento y el ejemplo dado como argumento. Si esa distancia es la mínima posible, se comprueba la etiqueta de ambos elementos, si coinciden significa que es el amigo más cercano, sino, el enemigo. Pseudocódigo:

```
Function closest_examples (datos, ejemplo, etiquetas)
    closest_enemy = 0
    closest_friend = 0
    etiqueta_ejemplo = etiquetas[ejemplo]
    enemy_dist = np.inf(∞)
    friend_dist = np.inf(∞)

    for i = 0 until n°datos
        if i != ejemplo
            dist = euclidean_distance(ejemplo,datos[i])
            if dist < enemy_dist and etiqueta_ejemplo != etiquetas[i]
                enemy_dist = dist
                closest_enemy = i
            if dist < enemy_dist and etiqueta_ejemplo == etiquetas[i]
                friend_dist = dist
                closest_friend = i

    Return datos[closest_enemy], datos[closest_friend]
```

- **evaluation_function** : Función de evaluación en la que se mide el rendimiento del clasificador. Recibe las etiquetas de los valores con los que se va a testear, la predicción dada por el clasificador, el vector de pesos asociado a las características y alpha). "**np.count.nonzero(condicion)**" devuelve el resultado de contabilizar cuántas veces se cumple una condición dada. En éste caso, para la tasa_clas devuelve el número de veces que la predicción se corresponde con el valor de la etiqueta asociado a cada elemento (success), y para calcular el porcentaje de tasa_clas, se divide el valor de 'success' entre el número total de etiquetas. Para la tasa red, devuelve el número de elementos del vector de pesos que tienen un valor por debajo de 0.2 (under_value), y para calcular el porcentaje de tasa_red, se divide el valor de 'under_value' entre el número total de etiquetas. Una vez obtenidos éstos valores, es posible calcular la función objetivo, cuyo resultado se almacena como Agregado. Por último, se devuelve el valor de la tasa_clas, la tasa_red y el Agregado para esos datos. Pseudocódigo:

Function evaluation_function (etiquetas_test, prediccion, W, α)
    success = 0
    under_value = 0

    success = np.count_nonzero(prediccion == etiquetas_test)
    tasa_clas = (100 * (success / nºetiquetas_test)

    under_value = np.count.nonzero(W < 0.2)
    tasa_red = (100 * (under_value / nºcaracteristicas(W))

    agr = α * **tasa_clas** + (1 - α) * **tasa_red**

    Return tasa_clas, tasa_red, agr

- **mean_calculus** : Función que calcula la media de los resultados obtenidos por la tasa_clas, tasa_red, agregaciones y tiempos en las 5 particiónes de cada ejecución. "**np.sum(valores)**" devuelve la sumatoria de los valores que se le pasan. Pseudocódigo:

Function mean_calculus(valores)
    summ = np.sum(valores)
    Return summ / 5.0

- **display_table** : Función que lleva a cabo la representación de las soluciones. Recibe la tasa_clas, la tasa_red, las agregaciones y los tiempos de ejecución de cada partición tras la ejecución de un algoritmo y las representa. Por último, se muestra la media de los resultados obtenidos en las 5 particiones. Pseudocódigo:

Function display_table (tasa_clas, tasa_red, agregaciones, times)

```
print(tasa_clas[0], tasa_red[0], agregaciones[0], times[0])
.
.
print(tasa_clas[4], tasa_red[4], agregaciones[4], times[4])
print(mean_calculus(tasa_clas),mean_calculus(tasa_red),
        mean_calculus(agregaciones),mean_calculus(times))
```

- Procedimiento común a todos los Algoritmos:

Para realizar la división del conjunto de datos en 5 particiones manteniendo las proporciones de cada clase, utilizo la función:

```
skf = StratifiedKFold(n_splits=5,shuffle=True, random_state=1)
```

Dicha función divide los datos mediante la técnica de validación cruzada. Los datos son tomados aleatoriamente. Devuelve los índices de la división de los datos para separar en datos de entrenamiento y datos de testeo:

```
for indice_train, indice_test until skf.split(datos,etiquetas)
    datos_train= datos[indice_train]
    etiquetas_train = etiquetas[indice_train]
    datos_test = datos[indice_test]
    etiquetas_test = etiquetas[indice_test]
```

Para construir el clasificador K-NN, utilizo:

```
neigh = KNeighborsClassifier(n_neighbors = 1)
```

El entrenamiento del clasificador se realiza con los datos y etiquetas de entrenamiento, utilizo:

```
neigh.fit(datos_train,etiquetas_train)
```

Una vez entrenado el clasificador, predice la etiqueta de los datos de testeo, utilizo:

```
prediction = neigh.predict(datos_test)
```

3. DESCRIPCIÓN EN PSEUDOCÓDIGO DE LA ESTRUCTURA DEL MÉTODO DE BÚSQUEDA

Function local_search (datos, etiquetas):

```
tasa_clas = []
tasa_red = []
times = []
agregacion = []

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
for train_index, test_index in skf.split(datos, etiquetas):
        start = time.time()
        x_train, y_train = datos[train_index], etiquetas[train_index]
        x_test, y_test = datos[test_index], etiquetas[test_index]

        w = np.random.uniform[0,1]
        n = 1
        iteraciones = 1
        neigh = KNeighborsClassifier(n_neighbors=1)
        neigh.fit(x_train*w, y_train)
        vecinos = neigh.kneighbors(x_test*w,n_neighbors=2,return_distance=False)
        y_test = y_train[vecinos[:, 1]]
        prediccion = neigh.predict(x_test*w)
        last_clas, last_red, last_agr = evaluation_function(y_test,prediccion,w,0.5)

        while n < 20*nºcaracteristicas and iteraciones < 15000
                for i=0  until nºcaracteristicas
                        w_copy = w.copy()
                        z = np.random.normal[0,0.3]
                        w[i] += z

                        if w[i] < 0.2
                                w[i] = 0.0
                        else if w[i] > 1.0
                                w[i] = 1.0


                        neigh = KNeighborsClassifier(n_neighbors=1)
                        neigh.fit(x_train*w,y_train)
                        idx =
                neigh.kneighbors(x_test*w,n_neighbors=2,return_distance=False)

                        y_test = y_train[idx[:,1]]
                        prediccion = neigh.predict(x_test*w)

                        new_clas, new_red, new_agr =
                        evaluation_function(y_test,prediccion,w,0.5)
                        iteraciones += 1
```

```
                    if new_agr > last_agr
                        last_agr = new_agr
                        break
                    else
                        w = w_copy
                        n += 1

        end = time.time()
        final_clas = new_clas
        final_red = new_red
        final_agr = new_agr

        tasa_clas.append(final_clas)
        tasa_red.append(final_red)
        agregacion.append(final_agr)
        times.append(end-start)

    Return tasa_clas, tasa_red, agregacion, times
```

El algoritmo de la Búsqueda Local tiene dos condiciones de salida: generar 20*nºcaracteristicas vecinos o evaluar la función objetivo 15.000 veces. Mientras no se cumpla alguna de las condiciones, el algoritmo va mutando el vector de pesos en busca de una solución que mejore los resultados anteriores. Su ejecución es tan sencilla como que en cada iteración va mutando un peso, si la solución mejora los resultados, se pasa a mutar el siguiente peso dejando el peso anterior con el valor de la mutación, si no mejora, se pasa a mutar el siguiente peso sin considerar el valor de la mutación anterior. Finalmente, el algoritmo devuelve los valores tasa_clas, tasa_red, agregacion y times obtenido en cada iteración considerando la mejor solución posible, es decir, el vector de pesos que maximiza la función de evaluación. Las funciones que no explico en éste apartado es porque han sido explicadas en el apartado **2.)**, ya que son comunes a todos los algoritmos. **Funciones a explicar:**
"**np.random.uniform[0,1]**" genera el vector de pesos inicial de forma aleatoria, mediante una distribución uniforme en el intervalo [0, 1].
"**neigh.kneighbors**" devuelve el índice del vecino más cercano dados unos datos. El hecho de recibir como argumento que el número de vecinos sea 2 es para cumplir la técnica de validación **leave-one-out**, es decir, como el vecino más próximo vamos a ser nosotros mismos, obtengo 2 vecinos y me quedo con el segundo. "**np.random.normal[0,0.3]**" genera el valor para llevar a cabo la mutación de un peso y es generado mediante una distribución normal de media 0 y desviación típica 0.3. "**copy()**" es usada para obtener una copia del vector de pesos. "**append**" es la función usada para insertar los resultados obtenido en cada partición.

4. DESCRIPCIÓN EN PSEUDOCÓDIGO DE LOS ALGORITMOS DE COMPARACIÓN

* ALGORITMO RELIEF *

Function relief (datos, etiquetas):

```
tasa_clas = []
tasa_red = []
agregacion = []
times = []

skf = StratifiedKFold(n_splits=5,shuffle=True, random_state=1)

for train_index, test_index until skf.split(datos, etiquetas)
        start = time.time()

        x_train, y_train = datos[train_index], etiquetas[train_index]
        x_test, y_test = datos[test_index], etiquetas[test_index]

        w = np.zeros(datos.shape[1])

        for i=0 until nºdatos_train
                enemigo, amigo = closest_examples(x_train,i,y_train)

                w = w + np.abs(x_train[i] - enemigo) - np.abs(x_train[i] - amigo)

        wm = np.max(w)
        for i=0 until nºcaracteristicas
                if w[i] < 0.0
                        w[i] = 0.0
                else
                        w[i] = w[i] / wm

        x_train = x_train * w
        x_test = x_test * w

        neigh = KNeighborsClassifier(n_neighbors=1)
        neigh.fit(x_train,y_train)
        prediccion = neigh.predict(x_test)
        clas, red, agr = evaluation_function(y_test,prediccion,w,0.5)

        end = time.time()

        tasa_clas.append(clas)
        tasa_red.append(red)
        agregacion.append(agr)
        times.append(end-start)

Return tasa_clas, tasa_red, agregacion, times
```

El algoritmo greedy Relief parte de un vector de pesos inicial, el cual trata de mejorar para obtener una solución mejor. Su funcionamiento para mejorarlo es el siguiente: Por cada valor en los datos de entrenamiento, obtiene su amigo y enemigo más cercano. Una vez obtenidos ambos, modifica el vector de pesos de forma que si la distancia entre el enemigo más cercano es mayor que la distancia con respecto al amigo más cercano, se incrementa el valor de los pesos, mientras que cuando ocurre al revés, se decrementa. Finalmente, el algoritmo devuelve los valores tasa_clas, tasa_red, agregacion y times obtenido en cada iteración considerando la mejor solución posible, es decir, el vector de pesos que maximiza la función de evaluación. Las funciones que no explico en éste apartado es porque han sido explicadas en el apartado **2.)**, ya que son comunes a todos los algoritmos. **Funciones a explicar:**
"**np.zeros**" es una función que inicializa el vector de pesos con el tamaño que se le indique a valor 0. "**np.abs**" devuelve el resultado de sus argumentos en valor absoluto. "**np.max**" devuelve el valor máximo hallado en el vector de pesos.

* ALGORITMO 1-NN *

Function k-nn (datos, etiquetas):

```
tasa_clas = []
tasa_red = []
times = []
agregacion = []

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)

for train_index, test_index until skf.split(x,y)

        start = time.time()
        x_train, y_train = datos[train_index], etiquetas[train_index]
        x_test, y_test = datos[test_index], etiquetas[test_index]

        w = np.ones(x.shape[1])

        neigh = KNeighborsClassifier(n_neighbors=1)
        neigh.fit(x_train,y_train)
        prediccion = neigh.predict(x_test)
        clas, red, agr = evaluation_function(y_test,prediccion,w,0.5)
        end = time.time()

tasa_clas.append(clas)
tasa_red.append(red)
agregacion.append(agr)
times.append(end-start)

Return tasa_clas, tasa_red, agregacion, times
```

Las funciones que usa éste algoritmo ya han sido explicadas anteriormente. En éste caso, todos los valores del vector de pesos se inicializan a 1 para que todos los elementos tengan la misma importancia.

Funciones a explicar:

"**np.ones**" es la función que inicializa el vector de pesos a valor uno.

5. BREVE EXPLICACIÓN DEL PROCEDIMIENTO CONSIDERADO PARA DESARROLLAR LA PRÁCTICA

- Para desarrollar la práctica he usado el lenguaje de programación Python 3.7.1, siendo necesario tenerlo instalado.

- El desarrollo lo he llevado a cabo en el IDE Spyder.

- Para ejecutar la práctica es necesario tener instalados los siguientes paquetes:

- Scikit-learn
- numpy (para realizar todo tipo de operaciones)
- pandas (para leer los ficheros en formato CSV)
- StratifiedKFold (para realizar las 5 particiones, train, test...)
- KneighborsClassifier (para obtener vecinos, y métodos derivados...)
- time (para contabilizar el inicio y fin de ejecución)
- MinMaxScaler (para normalizar los datos extraídos de los ficheros CSV)

- Los ficheros de datos están dentro de un directorio llamado 'datos'. En el propio fichero ya he dejado escrita la ruta de lectura, por lo que sería necesario que el fichero y la carpeta 'datos' estuvieran en el mismo nivel antes de ejecutar.

- Personalmente, recomiendo instalar Anaconda, puesto que ya trae por defecto la mayoría de los paquetes que he usado.

- Práctica realizada en el SO: Windows 10.

- Para ejecutar el fichero, bastaría con correrlo si se usa un IDE, o en CMD(también en otras terminales) teclear: '**python P1.py'**.

6. EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

○ CASOS DEL PROBLEMA Y VALORES DE LOS PARÁMETROS EMPLEADOS EN LAS EJECUCIONES DE CADA ALGORITMO

- Los datasets han sido convertidos a CSV previamente a su lectura. En la lectura de ellos, almaceno en arrays de Numpy los datos por un lado y las etiquetas por otro. Una vez leídos los datos, son normalizados. Posteriormente, se van llamando a los diferentes algoritmos y se van mostrando sus resultados.

- En cuanto a la pseudoaleatoriedad, la semilla está fijada al principio del fichero a 1. "**np.random.seed(1)**"

- En los 3 algoritmos, a la hora de dividir el conjunto en 5 particiones, he utilizado 'shuffle' en los datos. El hecho de haber considerado 'shuffle' conlleva que los datos se barajan con cierto índice de aleatoriedad, por lo que el valor de "**random_state**" ha sido puesto a 1 también durante la inicialización para que concuerde con el valor de la semilla.

- Para obtener el valor Agregado, es decir la función objetivo, he usado un valor de $\alpha = 0.5$.

- En el algoritmo de Búsqueda Local, el valor Z que provoca la mutación del vector de pesos ha sido generado mediante una distribución normal de media 0 y desviación típica = 0.3 para todas las ejecuciones.

○ RESULTADOS OBTENIDOS

Tabla 5.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	72,8810	0,0000	36,4406	0.001	83,0985	0,0000	41,5492	0,0029	92,7272	0,0000	46,3636	0,0029
Partición 2	71,9290	0,0000	35,9649	0.001	87,1428	0,0000	43,5714	0,0019	93,6363	0,0000	46,8181	0,0029
Partición 3	70,1750	0,0000	35,0877	0.001	85,7142	0,0000	42,8571	0,0019	91,8181	0,0000	45,9090	0,0019
Partición 4	71,9290	0,0000	35,9649	0.001	91,4285	0,0000	45,7142	0,0020	90,9090	0,0000	45,4545	0,0019
Partición 5	80,7010	0,0000	40,3508	0.001	85,7142	0,0000	42,8571	0,0019	95,4545	0,0000	47,7272	0,0029
Media	73,5230	0,0000	36,7618	0.001	86,6197	0,0000	43,3098	0,0021	92,9090	0,0000	46,4545	0,0025

Tabla 5.2: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	76,2711	35,4838	55,8775	0,2573	85,9154	2,9411	44,4283	0,3739	96,3636	2,5000	49,4318	0,9175
Partición 2	68,4210	40,3225	54,3718	0,2503	87,1428	2,9411	45,0420	0,3700	93,6363	7,5000	50,5681	0,9025
Partición 3	71,9298	40,3225	56,1262	0,2483	91,4285	2,9411	47,1848	0,3700	93,6363	15,0000	54,3181	0,9005
Partición 4	68,4210	27,4193	47,9202	0,2413	91,4285	2,9411	47,1848	0,3630	89,0909	20,0000	54,5454	0,8866
Partición 5	70,1754	45,1612	57,6683	0,2503	87,1428	2,9411	45,0420	0,3670	97,2727	5,0000	51,1363	0,8906
Media	71,0437	37,7419	54,3928	0,2495	88,6116	2,9411	45,7764	0,3688	94,0000	10,0000	52,0000	0,8995

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

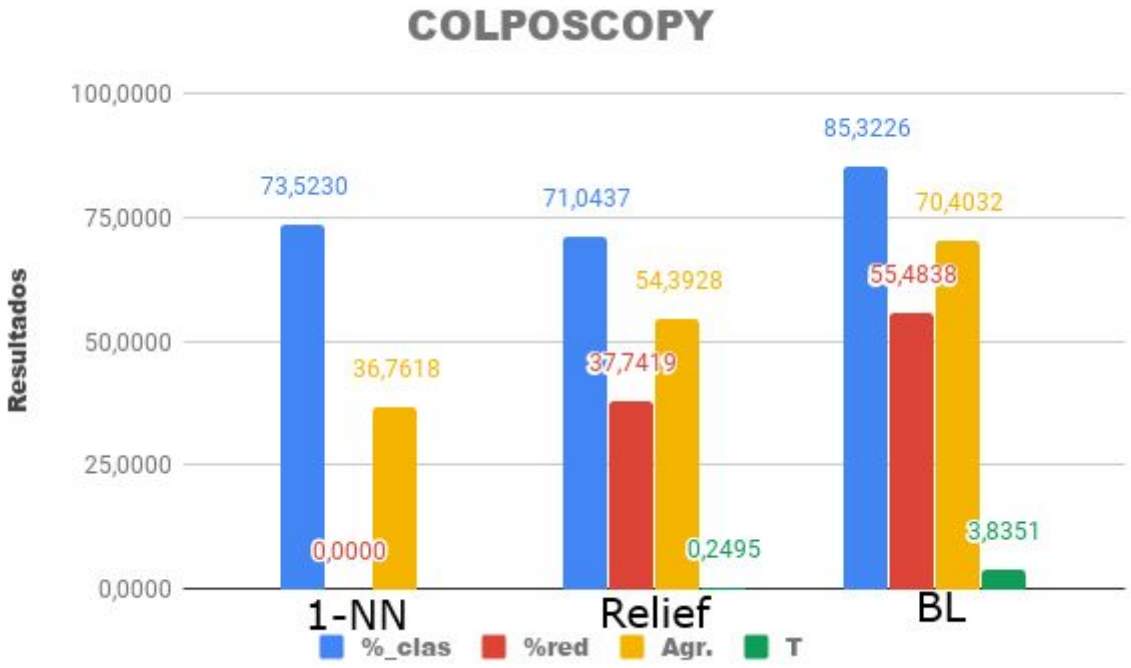
	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
Partición 1	91,5254	59,6774	75,6014	3,8716	98,5915	67,6470	83,1193	1,7293	92,7272	65,0000	78,8636	2,9681
Partición 2	82,4561	54,8387	68,6474	3,8766	94,2857	61,7647	78,0252	1,8460	89,0909	65,0000	77,0454	3,5375
Partición 3	82,4561	46,7741	64,6151	3,8975	95,7142	61,7647	78,7394	1,8161	93,6363	62,5000	78,0681	3,1735
Partición 4	87,7192	61,2903	74,5048	3,6741	98,5714	61,7647	80,1680	1,7503	93,6363	70,0000	81,8181	3,4597
Partición 5	82,4561	54,8387	68,6474	3,8556	97,1428	79,4117	88,2773	1,8480	90,0000	77,5000	77,5000	3,2722
Media	85,3226	55,4838	70,4032	3,8351	96,8611	66,4705	81,6658	1,7979	91,8181	65,5000	78,6590	3,2822

Tabla 5.4: Resultados globales en el problema del APC

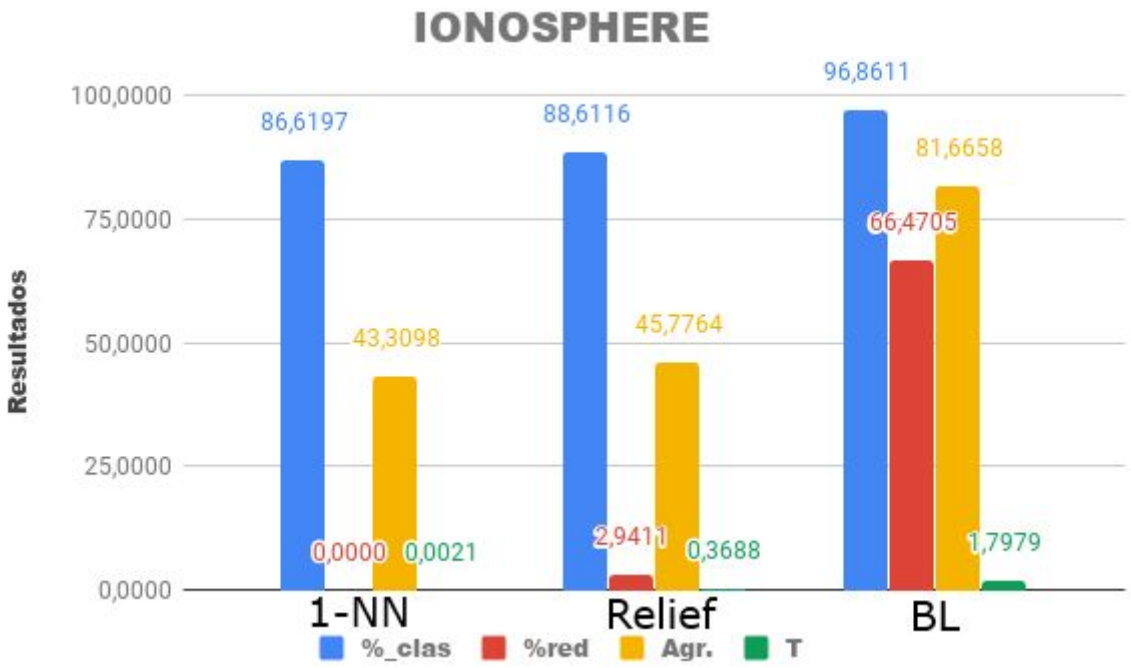
	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
1-NN	73,5230	0,0000	36,7618	0.001	86,6197	0,0000	43,3098	0,0021	92,9090	0,0000	46,4545	0,0025
RELIEF	71,0437	37,7419	54,3928	0,2495	88,6116	2,9411	45,7764	0,3688	94,0000	10,0000	52,0000	0,8995
BL	85,3226	55,4838	70,4032	3,8351	96,8611	66,4705	81,6658	1,7979	91,8181	65,5000	78,6590	3,2822

- **ANÁLISIS DE RESULTADOS**

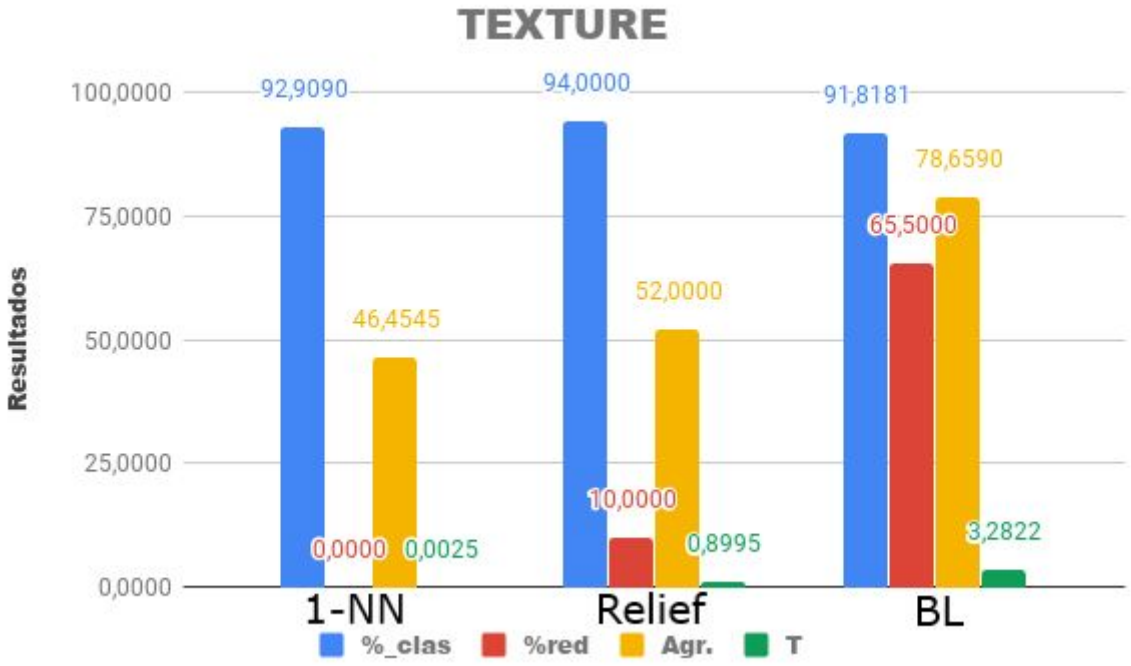
Para llevar a cabo el análisis de forma más ilustrativa, me voy a apoyar en 3 gráficas que he creado, en las que muestro una comparativa de los resultados obtenidos por cada algoritmo de forma individual en cada dataset.



En éste primer dataset, vemos como se evidencia que el algoritmo 1-NN es el más simple de todos y el que peores resultados obtiene. Si bien tiene una tasa de acierto aceptable, no consigue reducir a la hora de elegir al subconjunto de pesos, cosa que sí consigue Relief. Como vemos también, el valor de la función objetivo es mayor con el Relief, denotando así que obtiene mejores resultados que 1-NN. Si hablamos del algoritmo de Búsqueda Local, vemos que la tasa de aciertos se ve incrementada con respecto a los algoritmos de comparación, es decir que clasifica de forma más precisa ante unos datos de test. Por otra parte, consigue una reducción mayor y es el algoritmo que mejor maximiza la función objetivo. La conclusión general que se saca de éste gráfico es que la Búsqueda Local es el mejor algoritmo de los 3, con la única pega de que a cambio de obtener mejores resultados, el tiempo de ejecución es mayor.



En éste dataset de nuevo, la Búsqueda Local es muy superior con respecto a su comparativa. Obtiene una tasa de acierto rozando el 100%, lo que da una credibilidad a éste algoritmo a un mayor pues el fallo de predicción no llega al 5%. En cuanto a la reducción, vemos que al Relief le cuesta particularmente con estos datos, por lo que obtiene un valor de función objetivo similar al 1-NN en el que no es posible la reducción.



Para el último dataset, vemos que se repite todo lo comentado en los anteriores, la Búsqueda Local obtiene una mayor tasa de reducción, maximiza la función objetivo... pero en éste caso hay una particularidad, y es que para obtener esos valores de reducción y función objetivo, estamos sacrificando un pequeño porcentaje de tasa de acierto, dando lugar a que tanto Relief como 1-NN posean una tasa de acierto mayor. ¿Sería peor entonces la Búsqueda Local en éste caso? La respuesta es NO. Si bien es cierto que su tasa de acierto es inferior, la diferencia es mínima y no por ello lo hace peor, ya que estamos reduciendo los datos un 55% más que en el Relief y obteniendo un resultado de función objetivo que casi duplica al 1-NN y supera holgadamente al Relief.

Conclusión General

A la vista de los resultados, el algoritmo de Búsqueda Local es muy superior tanto al 1-NN como al Relief. Afronta con mayor facilidad la diversidad del dataset y siempre reduce considerablemente más que el Relief, manteniendo un nivel de tasa de aciertos más que aceptable que hace que sea el algoritmo que más maximiza la función objetivo en todos los casos. En cuánto al tiempo de ejecución, considero que he realizado una implementación óptima que permite que el algoritmo no tarde mucho más que los de comparación.

7. REFERENCIAS BIBLIOGRÁFICAS

- Consultas de funciones de Numpy: https://docs.scipy.org/doc/numpy/reference/index.html

- Consultas de funciones de Scikit-Learn: https://scikit-learn.org/stable/

- Consulta referente a la lectura de datos: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

- Material de clase.