Curso: 2018 - 2019

Práctica 1b: APC

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

José Javier Alonso Ramos

dni: deleted

email: deleted

Grupo: 2

Horario prácticas: Miércoles 17:30 - 19:30

UNIVERSIDAD DE GRANADA

Índice

La práctica ha sido realizada en python. Usar python3 como intérprete; si es posible, python3.6.

Aloritmos considerados

- Greedy RELIEF
- Local Search
- 1-NN

Descripción del problema

Dada una **población**, nos encontramos una serie de *n* **elementos** por los que está formada, teniendo cada uno de ellos *x* **características** según las cuales se les **clasifica** en una determinada clase *c* perteneciente a un conjunto de clases *C*.

Nuestro problema comprende la situación de clasificación de un nuevo elemento insertado en la población con la tara que supone no saber el criterio, según el cual, se clasifica.

Para llevar a cabo esta tarea crearemos un sistema clasificador automático que analizará un subconjunto de la población escogido de manera que sea representativo respecto a la población al que llamaremos **muestra**. Aprenderemos a partir de las características de los elementos que conforman la muestra y sus clasificaciones una manera de clasificar el nuevo elemento correctamente con una fiabilidad suficientemente alta.

Es un problema de **aprendizaje supervisado** ya que cuando entrenamos a nuestro sistema clasificador corroboraremos los resultados obtenidos con la clase real a la que pertenece el elemento, pudiendo así variar nuestras decisiones en pos de mejorar nuestro clasificador.
El método empleado para aprender a clasificar será decisivo en la calidad, coste requerido y velocidad de la clasificación.

Elementos comunes a todos algoritmos implementados

Módulos importados y para qué han sido utilizados:

- **arff** (de *scipy.io*): este módulo nos permite leer los archivos con extensión *.arff* de datos de población.
- **numpy**: nos permite trabajar de forma cómoda con vectores y matrices a parte de proporcionar una serie de funciones matemáticas como *valor absoluto*, *media*, *distancia euclídea*, *generación de números aleatorios*, *distribución normal y uniforme*, *truncamiento de valores a un determinado rango*, y un largo etcétera.
- **pandas**: nos permite trabajar de forma cómoda con conjuntos de datos y crear tablas entre otras cosas. En la práctica lo usamos para transformar los datos leídos del fichero arff a una matriz.
- **KDTree** (de *sklearn.neighbors*): creamos un arbol con un determinado conjunto de datos (*conjunto1*) al que, pasándole un segundo conjunto (*conjunto2* - que puede ser el mismo que el primero), nos devuelve *k* elementos del conjunto1 en orden del más cercano al más lejano a cada elemento del conjunto2.
- **StratifiedKFold** (de *sklearn.model_selection*): en la práctica se especifica que particionemos los datos leídos en 5 conjuntos del 20 % de manera que las etiquetas se mantengan proporcionales al conunto original. Este módulo nos permite hacer esto además de manera aleatoria y mezclando los datos en los subconjuntos.
- **MinMaxScaler** (de *sklearn.preprocessing*): nos permite escalar los valores de las carácterísticas de los elementos al rango [1,0]
- **time** (de *time*): nos permite calcular el tiempo de ejecución de los ditintos algoritmos.
- **PrettyTable** (de *prettytable*): para imrimir los resultados obtenidos por los algoritmos de forma ordenada en tablas.
- **matplotlib.pyplot**: nos permite hacer gráficas
- **sys**: nos permite leer parámetros de entrada

Funciones auxiliares:

- **byte2string(x)**: cuando leemos el conjunto de datos *x* transformamos todas las etiquetas (clases) en strings para poder trabajar con ellas de manera uniforme. Esto permite que si leemos un conjunto de datos y sus etiquetas, en un principio, son númericas trabajaremos con él de igual forma que lo haríamos con un conjunto de etiquetas alfabéticas.
- **read_arff(name_of_file)**: lee el contenido de un archivo *.arff* situado en el directorio *../data/* relativo al directorio donde se encuentra nuestro script. Transforma los datos a un formato más amigable.

Devuelve dos objetos: datos y metadatos.
~ python def read_arff(name_of_file): datos, metadatos := read(../data/name_of_file.arff) transform(datos)

return datos, metadatos ~

- **get_tags(data)**: del conjunto de datos *data* devolvemos la última columna, es decir, las etiquetas.
  ~ python def get_tags(data): return last_column(data) ~
- **get_only_data(data)**: del conjunto de datos *data* devolvemos todas las columnas excepto la última, es decir, sólo las características de los datos.
  ~ python def get_only_data(data): return (data - last_column(data)) ~

Elemento solución

Como solución de los algoritmos obtenemos un **vector de pesos** en el intervalo [0,1] de tamaño igual al número de características de los elementos. Este vector pondera la importancia de las características a la hora de clasificar los elementos. Un peso de 1 indica que esa carácterística es clave para saber la clase a la que pertenece el elemento, y un peso de 0 indica que es totalmente irrelevante.
El vector solución se denotará como ***w***.

Clasificador k-NN

Consiste en almacenar una serie de *n* de elementos (que en nuestro caso será la muestra) junto con sus etiquetas de manera que, al incorporar un nuevo elemento *e* a clasificar, se calculará la distancia entre este nuevo elemento y los *n* almacenados, escogiendo los *k* elementos más cercanos y seleccionando como nueva clase para *e* la clase común a la mayoría de los *k* elementos. Por esto es habitual encontrar k-NNs con k impar.
Es importante haber normalizado los datos, tanto *n* como *e*, para no priorizar unos sobre otros.

Si aplicamos el vector de peso que nos han dado como resultado nuestros algoritmos a la muestra y al nuevo elemento su clasificación será más clara y rápida.

En la práctica utilizaremos este clasificador para entrenar al algoritmo de **Búsqueda Local** y para comparar los resultados de clasificar un conjunto **test** con sus etiquetas reales para evaluar la calidad del vector de pesos **w**.
Esta medida de calidad la obtendremos a través de la función:

$$F(x) = \alpha Tasa\_aciertos + (1 - \alpha) Tasa\_reduccion$$

Donde **Tasa_aciertos** será la media de etiquetas acertadas, **Tasa_reducción** será la proporción de características de las cuales podemos prescindir respecto del total y α la importancia que le damos a cada una de las componentes anteriores. En nuestro caso en concreto fijaremos α a 0.5.

```
def k_NN(data_training, tags_training, w, data_test = None, tags_test = None, is_training = True):
    """
    Por defecto el algoritmo se ejecuta para entrenar Local Search,
    lo que significa que el conjunto test será el mismo que el conjunto muestra.
    Si por el contrario queremos pasar a evaluar un conjunto test real,
    pondremos el parámetro 'is_training' a False y los parámetros
    'data_test' y 'tags_test' contendrán las características y las etiquetas
    del conjunuto test respectivamente.
    """
    # data_training * w y nos quedamos sólo con las columnas > 0.2
    ponderar_caracteristicas(data_training)
    # Creamos un arbol de vecinos con los elementos de la muestra
    tree:= KDTree(data_training)
    # Si ejecutamos el algoritmo desde Local Search
    if estamos_entrenando then:
        """
        Obtenemos los 2 vecinos mas cercanos respecto a la muestra.
        Cogemos los dos más cercanos ya que el más cercano sería el propio
        elemento
        """
        vecinos := tree.mas_cercano(data_training, k:=2)
        """
        Nos quedamos con la segunda columna de vecinos.
        La primera es el propio elemento
        """
        vecinos := select_column(vecinos, 2)
        tasa_de_acierto := numero_de_etiquetas_bien_puestas(vecinos,
            data_training) / numero_filas(data_training)
    # Si evaluamos un algoritmo con un conjunto test
    else:
        # data_test * w y nos quedamos sólo con las columnas > 0.2
        ponderar_caracteristicas(data_test)
        """
        Como data_test no pertenece a data_training ningún elemento se
```

```
35          repite y cogemos solo 1 elemento más cercano
36          """
37          vecinos := tree.mas_cercano(data_test, k:=1)
38          tasa_de_acierto := numero_de_etiquetas_bien_puestas(vecinos,
39              data_test) / numero_filas(data_test)
40
41      tasa_de_reduccion := w.n_elementos_menor_que(0.2) /
            numero_elementos(w)
42
43      f := 0.5*tasa_de_acierto + 0.5*tasa_de_reduccion
44
45      return f, tasa_de_acierto, tasa_de_reduccion
```

Local Search

Partimos de un vector de pesos *w* generado aleatoriamente por una **distribución uniforme** entre [0,1] y lo evaluamos con k-NN respecto a la muestra que tenemos. Tras evaluar guardamos esta configuración de *w* y variamos una de sus componentes con una **distribución normal** aleatoria de *media*=0 y *desviación típica*=0.3 también en el intervalo [0,1]. Volvemos a evaluar en k-NN; si el resultado es mejor actualizamos, si no, restauramos el valor de la componente de *w* y cambiamos otra de la misma forma. Repetimos este proceso durante **15000 evaluaciones** o hasta generar **20*n** (n = número de elementos de la muestra) modificaciones erroneas en *w*, lo que suceda antes. Hay que tener en cuenta que cada vez que actualicemos *w* el número de modificaciones erróneas se reseteará.

```
1  def local_search(data, tags):
2      """
3      Generamos w con un tamaño igual al número de características con
           una
4      distribución uniforme aleatoria entre [0,1].
5      La función utilizada para ello ha sido 'numpy.random.uniform' del m
           ódulo
6      numpy
7      """
8      w := distribución_uniforme_aleatoria([0,1])
9      # Establecemos un máximo de evaluaciones para acabar el algoritmo
10     max_eval := 15000
11     # Marcamos el máximo de vecinos erróneos generados permitidos
12     max_neighbors := 20*data.shape[1]
13     # Creamos un contador de evaluaciones
14     n_eval := 0
```

```
    # Creamos un contador de vecinos erróneos
    n_neighbors := 0
    # Evaluamos w inicial
    class_prev = k_NN(data, tags, w)
    # Marcamos la condición de parada del algoritmo
    while n_eval < max_eval and n_neighbors < max_neighbors do:
        # Recorremos los valores del vector w uno a uno
        for w_i in w:
            # Aumentamos el número de evaluaciones
            n_eval := n_eval + 1
            # Guardamos el valor actual del componente de w
            prev := w_i
            # Modificamos la componente de w sumándole la distr. normal
            # Truncamos el resultado para mantenernos en [0,1]
            w_i := truncar( (w_i + distribución_normal_aleatoria(media
                =0, desviación=0.3, [0,1]) ), [0,1] )
            # Evaluamos w trás la modificación
            class_mod := k_NN(data, tags, w)
            # Si w tras la modificación es mejor que antes
            if(class_mod > class_prev):
                # Actualizamos evaluación con la nueva obtenida
                class_prev = class_mod
                # Reseteamos a 0 el número de vecinos erróneos
                n_neighbors := 0
                # Salimos del bucle interno y comenzamos a recorrer w
                    de nuevo
                break
            # Si w tras la modificación es peor que antes
            else:
                # Restauramos el valor de w
                w_i := prev
                # Aumentamos el número de vecinos erróneos generados
                n_neighbors +:= 1
    # Devolvemos w
    return w
```

Algoritmos de Exploración

Greedy RELIEF

Es un método brusco a la par que intuitivo. Iniciamos con un *w* con todas sus componentes a 0. Para cada elemento de la muestra recorremos todos los elementos restantes calculando la distancia entre ellos. Nos quedaremos con el **"amigo"** (elemento con la misma etiqueta) y el **"enemigo"** (elemento con distinta etiqueta) más cercano. Al vector w le sumaremos el valor absoluto de la diferencia entre las componentes del elemento estudiado y su enemigo más cercano, y le restaremos el valor absoluto de la diferencia entre las componentes del elemento estudiado y su amigo más cercano. Dicho de una manera más clara aumenta el peso de las componentes que separan a los enemigos entre sí y disminuye el peso de las componentes que separan los amigos entre sí.

```
1  def relief(data, tags):
2      # Creamos w inicializándolo a 0
3      w := zeros(num_elementos_muestra)
4      """
5       Creamos variables para almacenar el amigo y enemigo más
6       cercano.
7       """
8      closest_enemy := None
9      closest_friend := None
10     """
11     Creamos variables que nos permitan reconocer si hemos alcanzado ya
           un
12     amigo y un enemigo para salir del bucle
13     """
14     ally_found := False
15     enemy_found := False
16     # Creamos un KDTree con la muestra
17     tree := KDTree(data)
18     """
19     Pedimos que os devuelva todos los vecinos para explorarlos y saber
           quién
20     es nuestro amigo y quién nuestro enemigo.
21     """
22     vecinos := tree.mas_cercano(data, k:=num_elementos_muestra)
23     """
24     Quitamos la primera columna pues ese vecino corresponde con el
           propio
25     elemento.
```

```
    """
    vecinos := quit_column(vecinos,1)
    # Recorremos todos los elementos de la muestra
    for elem in numero_de_elementos:
        # Recorremos todos los vecinos generados
        for vec in numero_de_vecinos:
            # Si todavía no hemos encontrado amigo y este lo es
            if not ally_found and este_vecino_es_amigo:
                # Marcamos que ya hemos encontrado amigo
                ally_found := True
                # Guardamos su id
                closest_friend := elem.vecinos(vec)
            # Si todavía no hemos encontrado enemigo y este lo es
            if not enemy_found and tags[i] !:= tags[ vecinos[i,j] ]:
                # Marcamos que ya hemos encontrado enemigo
                enemy_found := True
                # Guardamos su id
                closest_enemy := elem.vecinos(vec)
            # Si ya hemos encontrado al amigo y al enemigo de este
               elemento
            if ally_found and enemy_found:
                # Salimos del bucle interno
                break
        # Restauramos los valores booleanos
        ally_found := enemy_found := False
        # Actualizamos el vector w
        w := w + |elem - closest_enemy| - |elem - closest_friend|

    # Calculamos el máximo de los elementos de w
    w_max := np.max(w)
    # Todos los elementos negativos los convertimos en 0
    w[ w < 0.0] := 0.0
    # Normalizamos los datos
    w /:= w_max
    # Devolvemos el resultado
    return w
```

1-NN

Es igual que el algoritmo *k-NN* explicado anteriormente pero no contamos con un vector de pesos que pondere las características según su importancia. Por lo tanto asignaremos la etiqueta según el vecino más cercano sin modificar los elementos.

```
def _1_NN(data_training, tags_training, data_test, tags_test):
    # Creamos un arbol de vecinos con los elementos de la muestra
    tree:= KDTree(data_training)
    """
    Como data_test no pertenece a data_training ningún elemento se
    repite y cogemos solo 1 elemento más cercano
    """
    vecinos := tree.mas_cercano(data_test, k:=1)
    tasa_de_acierto := numero_de_etiquetas_bien_puestas(vecinos,
        data_test) / numero_filas(data_test)
    """
    Como no hay vector de pesos no hemos reducido el número de
    características imorescindibles
    """
    tasa_de_reduccion := 0.0

    f := 0.5*tasa_de_acierto + 0.5*tasa_de_reduccion

    return f, tasa_de_acierto, tasa_de_reduccion
```

Procedimiento considerado para desarrollar la práctica

La escritura de código se ha llevado a cabo desde cero. El orden seguido durante la creación de la práctica ha sido el marcado por el PDF del seminario y no el del guión de prácticas. Si bien es cierto que para comprender el funcionamiento de los algoritmos he consultado ambos PDFs a la vez que diversos vídeos en YouTube y algunos posts en StackOverflow. Ha sido en estas consultas en internet donde he encontrado algunos de los módulos que uso como son KDTree, StratifiedKFold o MinMaxScaler. Para comenzar debemos cerciorarnos de ser capaces de leer los documentos .arff y transformar los datos, si es necesario, a un formato más amigable. Una vez seamos capaces de leer y manipular los datos podemos comenzar con la implementación de los algoritmos.
Recomiento empezar a programar los algoritmos Greedy o k-NN ya que son independientes de los demás y tienen una estructura clara. Una vez realizados podemos seguir con 1-NN que es tan solo una reducción del k-NN y con Búsqueda Local que requiere del uso de k-NN.

Análisis de resultados

Todos los resultados obtenidos se han tomado con una semilla fija a 1:
np.random.seed(1)
Pero podemos pasar una semilla como parámetro al programa.

Respecto a la diferencia de resultados entre algoritmos, la razón es clara:
1-NN es claramente el peor ya que no optimiza en ningún sentido la clasificación de los elemntos. No cuanta con un vector de pesos que pondere la importancia de las características a la hora de clasificar.
Greedy, al contrario, si genera este vector de pesos, *w*, pero lo hace de una manera muy general aplicando cambios a todas sus componentes a la vez y por lo tanto no siendo muy preciso en qué características son las verdaderamente decisivas.
Local Search sí aplica una variación específica y concreta a cada elemento de *w* permitiendo destacar aquellas variables distintivas y pormenorizar las irrelevantes.

Como vemos la mayor diferencia entre algoritmos es, sobretodo, la capacidad de desechar características inecesarias durante la clasificación. Como en nuestra función objetivo valoramos por igual la capacidad de acertar la etiqueta como la de reducir las características las diferencias se hacen más notables.

Tablas comparativas:

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	76,27	35,48	55,88	0,01	85,92	2,94	44,43	0,02	96,36	2,50	49,43	0,03
Partición 2	68,42	40,32	54,37	0,01	87,14	2,94	45,04	0,02	92,73	7,50	50,11	0,03
Partición 3	70,18	40,32	55,25	0,01	92,86	2,94	47,90	0,02	93,64	15,00	54,32	0,03
Partición 4	68,42	27,42	47,92	0,01	91,43	2,94	47,18	0,02	88,18	20,00	54,09	0,03
Partición 5	71,93	45,16	58,55	0,01	87,14	2,94	45,04	0,02	97,27	5,00	51,14	0,03
Media	71,04	37,74	54,39	0,01	88,90	2,94	45,92	0,02	93,64	10,00	51,82	0,03

Figura1: Greedy

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	76,27	83,87	80,07	4,65	85,92	82,35	84,13	2,35	90,91	82,50	86,70	8,82
Partición 2	70,18	79,03	74,60	5,59	82,86	88,24	85,55	2,04	91,81	85,00	88,41	3,11
Partición 3	70,18	85,48	77,83	8,56	87,14	91,17	89,16	3,04	92,72	82,50	87,61	3,19
Partición 4	70,18	69,35	69,77	4,67	87,14	88,23	87,69	2,55	87,27	85,00	86,14	4,78
Partición 5	73,68	70,96	72,32	5,95	91,42	91,17	91,30	2,02	86,36	82,50	84,43	3,93
Media	72,10	77,74	74,92	5,88	86,89	88,23	87,56	2,40	89,82	83,50	86,66	4,76

Figura2: LocalSearch

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	72,88	0,00	36,44	0,00	83,10	0,00	41,55	0,00	92,73	0,00	46,36	0,00
Partición 2	71,93	0,00	35,96	0,00	87,14	0,00	43,57	0,00	93,64	0,00	46,82	0,00
Partición 3	70,18	0,00	35,09	0,00	85,71	0,00	42,86	0,00	91,82	0,00	45,91	0,00
Partición 4	71,93	0,00	35,96	0,00	91,43	0,00	45,71	0,00	90,91	0,00	45,45	0,00
Partición 5	80,70	0,00	40,35	0,00	85,71	0,00	42,86	0,00	95,45	0,00	47,73	0,00
Media	73,52	0,00	36,76	0,00	86,62	0,00	43,31	0,00	92,91	0,00	46,45	0,00

Figura3: 1-NN

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
1-NN	73,52	0,00	36,76	0	86,62	0,00	43,31	0	92,91	0,00	46,45	0
RELIEF	71,04	37,74	54,39	0,01	88,9	2,94	45,92	0,02	93,64	10,00	51,82	0,03
BL	72,10	77,74	74,92	5,88	86,89	88,23	87,56	2,40	89,82	83,50	86,66	4,76

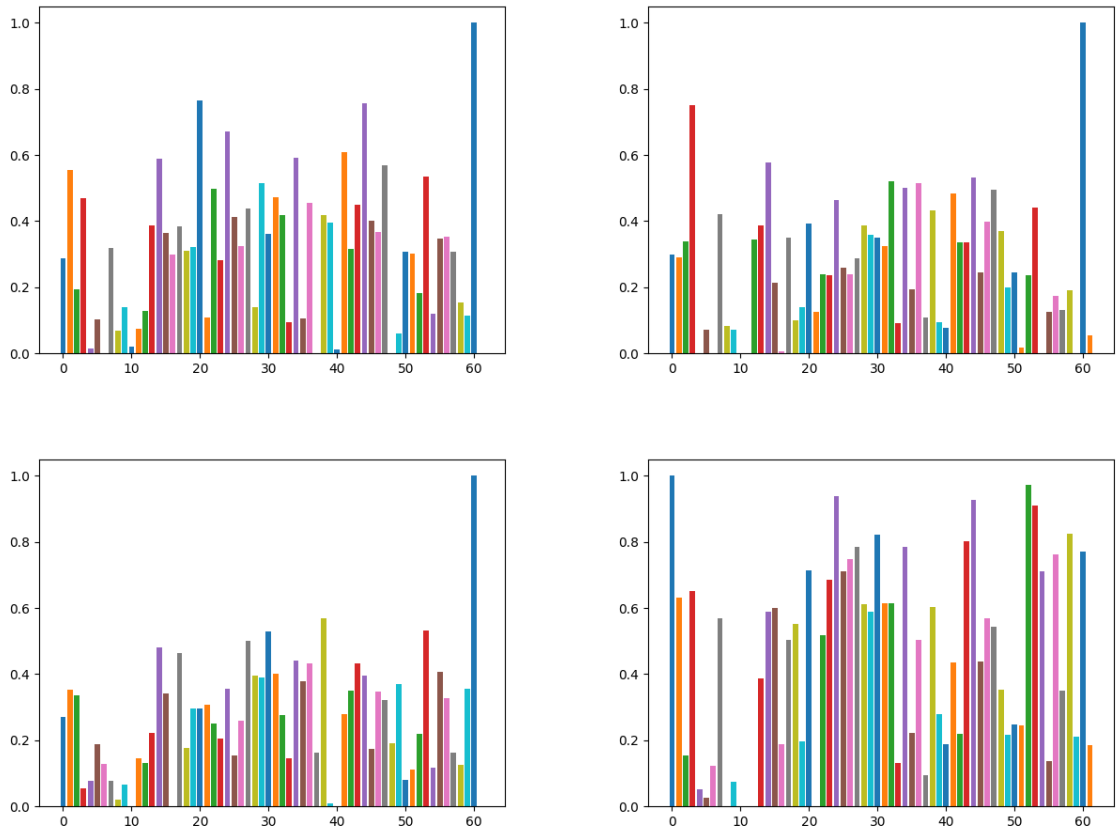
Figura4: Medias

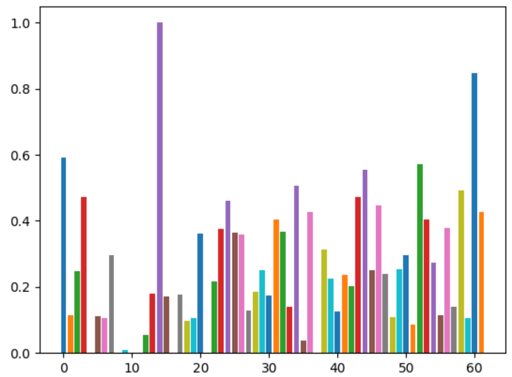
Gráficos comparativos

Cada gráfico corresponde a una de las 5 pariciones diferentes. Representan los valores que toma el vector de pesos *w* para cada característica. Podemos apreciar como en el algoritmo Greedy no hay tanta diferencia de ponderaciones como la hay en Locar Search.

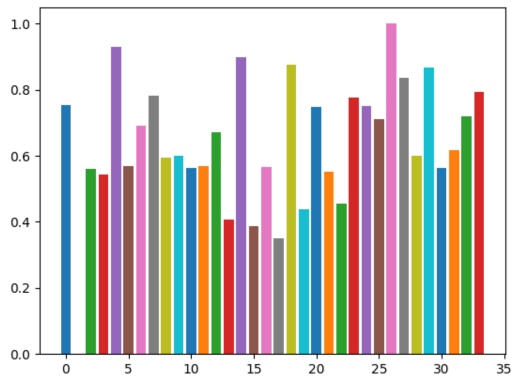
Gráficos Greedy

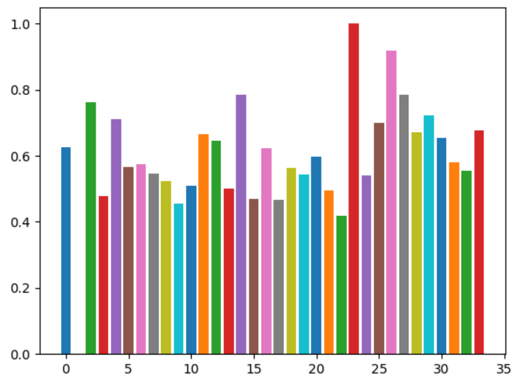
Colposcopy

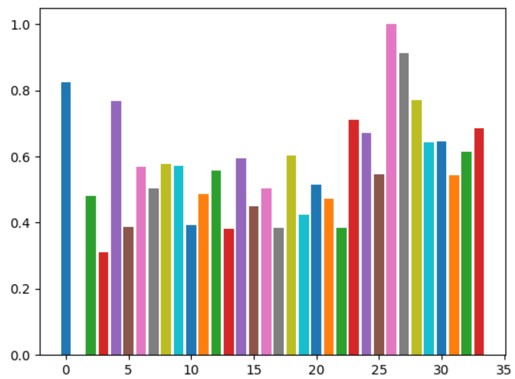


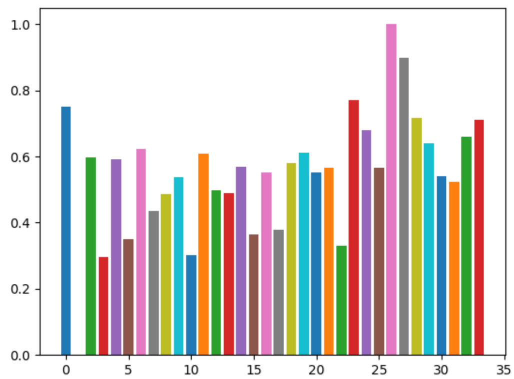

1.0
0.8
0.6
0.4
0.2
0.0
0 10 20 30 40 50 60

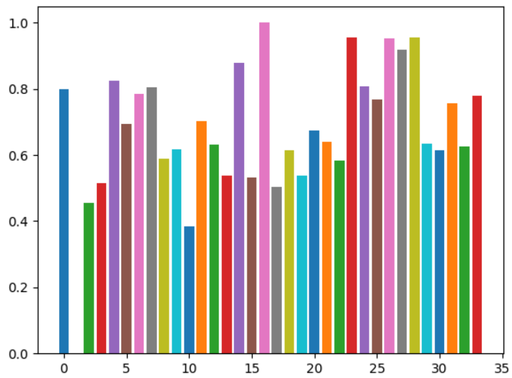
Ionosphere


1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

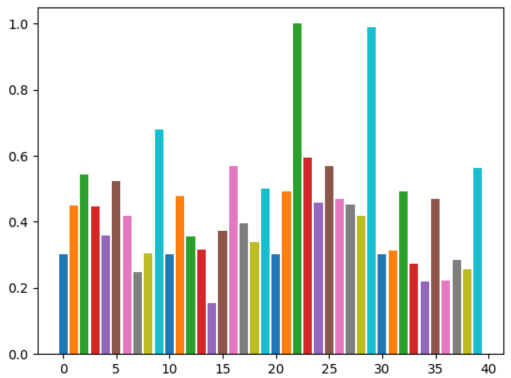

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

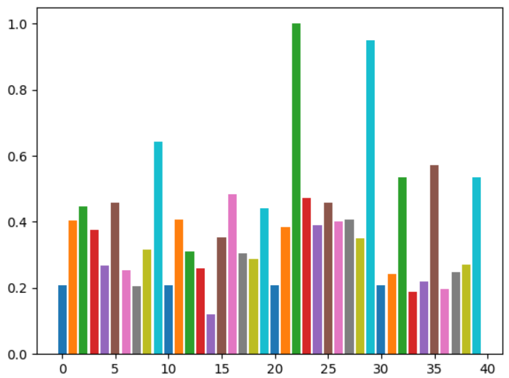

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

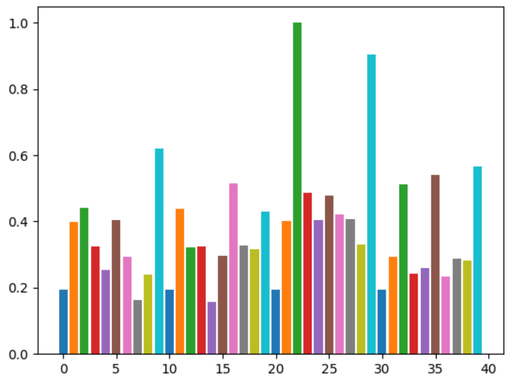

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

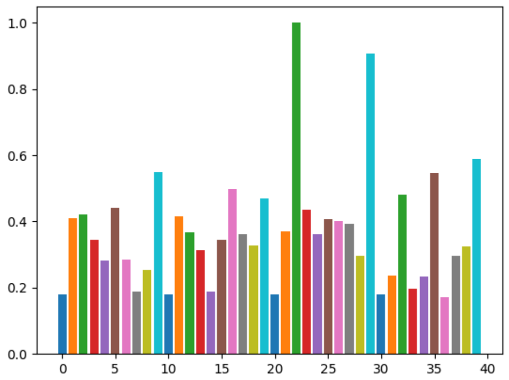


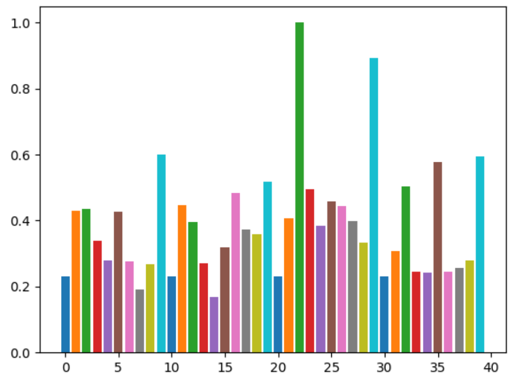
Texture





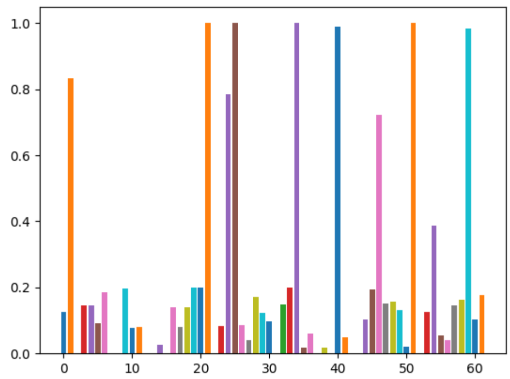


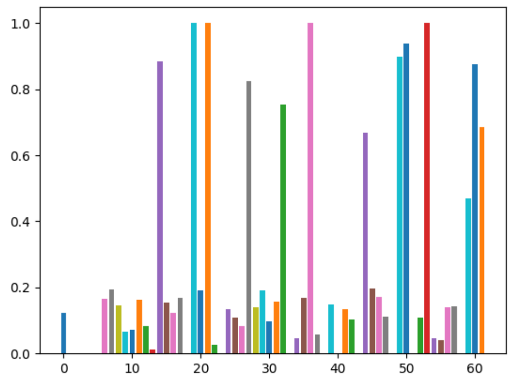


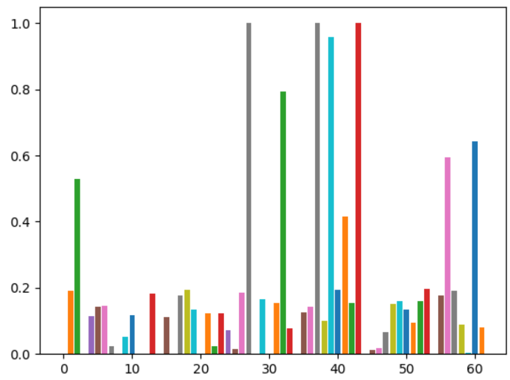


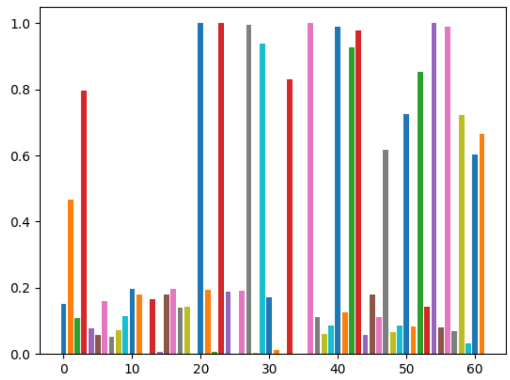
Gráficos Local Search

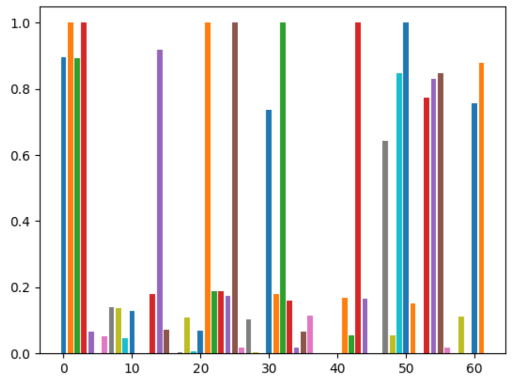
Colposcopy



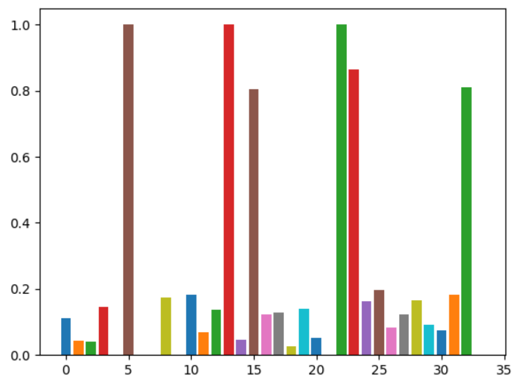


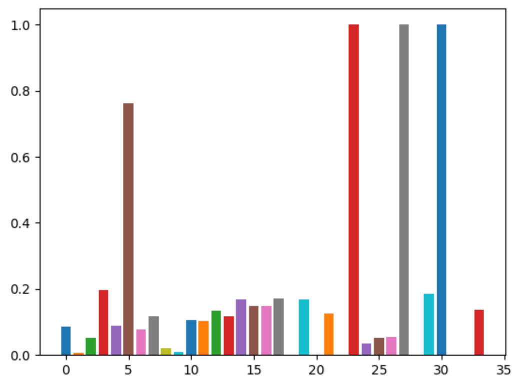


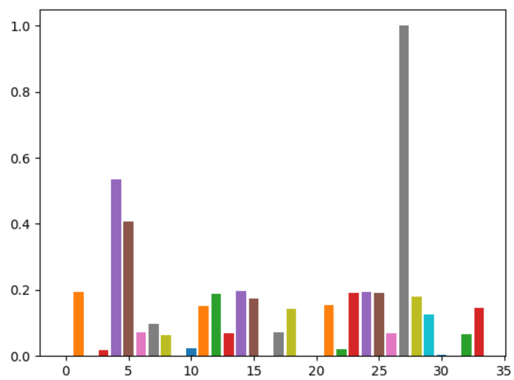


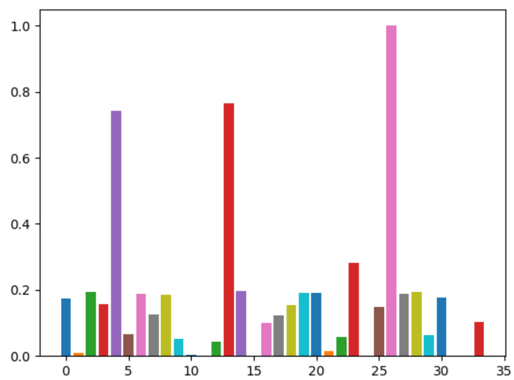

1.0
0.8
0.6
0.4
0.2
0.0
0 10 20 30 40 50 60

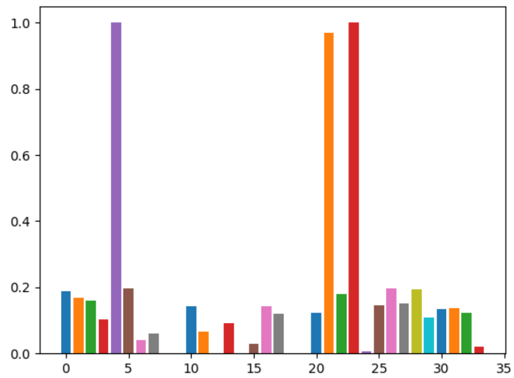
Ionosphere


1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

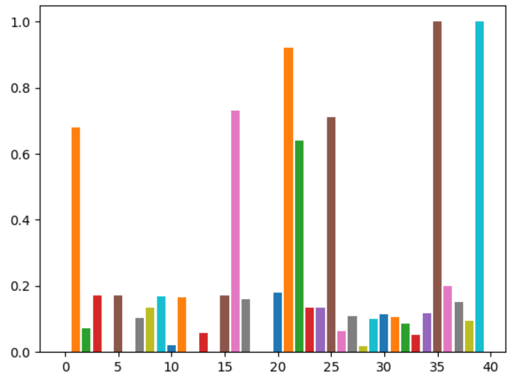

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

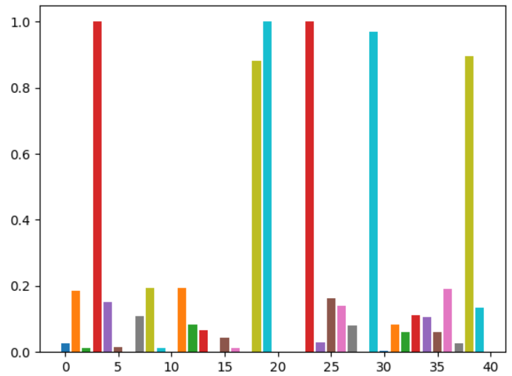

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

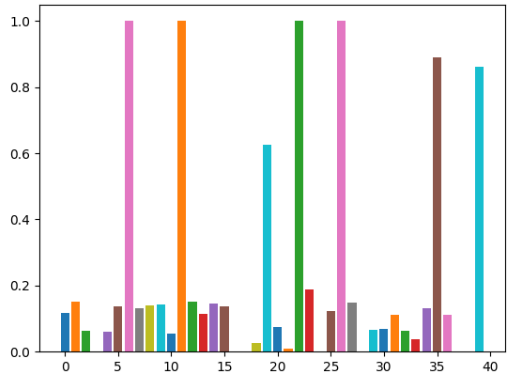

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

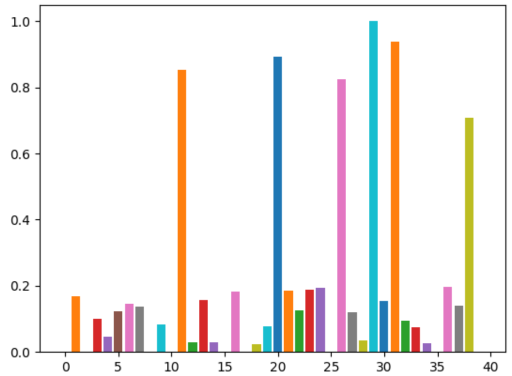

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35

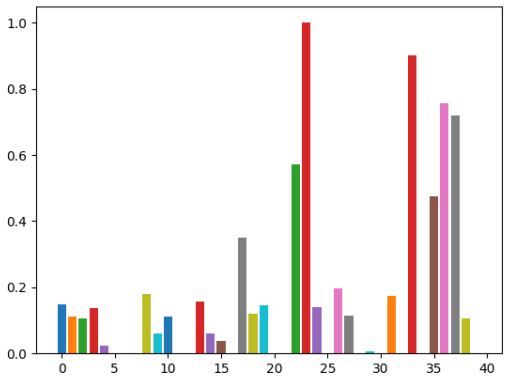
Texture


1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35 40


1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35 40


1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35 40


1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35 40


1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
30
35
40