Practica 1: APC con 1-NN, RELIEF y BL

Metahurística

José Manuel Pérez Lendínez

jmplz14@correo.ugr.es
Grupo 1 curso 18/19

Contents

1 Descripción del problema **1**

2 Aplicación de los algoritmos **2**
2.1 Descripción de la representación y calculo de distancias 2
2.2 Calculo de la tasa de reducción 3
2.3 Calculo de la tasa de clase 3
2.4 Función de evaluación 3

3 Explicación de los algoritmos **4**
3.1 Algoritmo del vecino mas cercanos 1-NN 4
3.2 Algoritmo RELIEF 4
3.3 Algoritmo de Búsqueda Local 7

4 Algoritmo de comparación **9**

5 Explicación de desarrollo de la práctica **10**

6 Análisis de resultados **10**
6.0.1 Semilla 10
6.0.2 Valores utilizados 10
6.0.3 Tablas de resultado 10
6.0.4 Análisis de resultados 11
6.0.5 Tiempo de ejecución 12

7 Bibliografía **12**

1 Descripción del problema

Vamos a estudiar tres métodos para afrontar el problema de aprendizaje de pesos con características. El problema se basa en optimizar la clasificación de nuevos elementos a partir de dos particiones. La particiones de entrenamiento y la partición de test.

Tendremos un conjunto de datos que dividiremos en 5 particiones, 4 para entrenamiento y una para test. La partición de test ira rotando hasta que todas pasen por entrenamiento una vez.

Los conjuntos de datos vendrán dados por una clase y un conjunto de características. Se representarán como un vector con la siguiente estructura.

$$(x_1, x_2, ..., x_n, y)$$

Donde las x corresponde a las características y la y a la clase que pertenece ese elemento.

El objetivo sera ser capaces de aceptar con el vector de características a la clase que pertenece. Para ello entrenaremos nuestro modelo con los datos de entrenamiento y usaremos los datos de test para ver si el modelo da resultados mas o menos acertados.

El clasificador tendrá que ser capaz de adivinar la clase utilizando la distancia a los datos de entrenamiento mas cercanos a la muestra elegida de los datos de test. Esta técnica se conoce como k-vecinos mas cercanos y sera la utilizada en todos nuestros algoritmos. Nosotros usaremos una k=1 por lo que solo buscaremos el vecino mas cercano para clasificar sin tener en cuenta a el mismo como vecino.

Al trabajar con datos reales sera muy difícil llegar a la solución exacta por lo que se buscaran aproximaciones.

Para esto utilizaremos tres algoritmos que explicaremos a continuación y que son:

1. 1-nn
2. RELIEF
3. Búsqueda Local

2 Aplicación de los algoritmos

En esta parte se explicaran la representación del problema de una forma mas completa y las características comunes que tienen todos los algoritmos.

2.1 Descripción de la representación y calculo de distancias

Para representar los datos usaremos matrices de numpy para los datos, que tendrán la siguiente estructura después de cargarlos desde los fichero.

$$datos = \begin{bmatrix} x_{1,1} & x_{1,2} & ... & x_{1,j} & y_1 \\ . & . & . & . & . \\ . & . & . & . & . \\ x_{i,1} & x_{i,2} & ... & x_{i,j} & y_i \end{bmatrix}$$

Esta matriz tendrá que ser normalizada la parte de los datos (las x) entre [0,1] para realizar con ella los cálculos de distancias para ver cual es el vecino mas cercano.

Para los algoritmos RELIEF y para la búsqueda local necesitaremos un vector de pesos que se representara de la siguiente manera:

$$(w_1, w_2, ..., w_j)$$

Este vector sera utilizado para ponderar la importancia de las distintas características de nuestros problemas. El vector tendrá unos valores acotados entre [0,1] y tendremos en cuenta que los valores menores que 0.2 no se utilizara para calcular las distancias en la clasificación. Esto restara importancia a las variables que tengan una ponderación muy baja y que podrían introducir ruido.

Para calculo de distancias usaremos la distancia euclídea. Multiplicando los pesos por las características de los datos.

$$d_e(e_1, e_2) = \sqrt{\sum_{n=0}^{\mathrm{J}} w_j(e_1^i - e_2^i)}$$

En nuestros datos de trabajo no hay ninguna característica que sea nominales por lo que no utilizo la distancia de Hamming para calcular la con las características nominales.

2.2 Calculo de la tasa de reducción

La tasa de reducción vendrá dada por la cantidad de pesos de nuestro vector w que sean inferior a 0.2 de forma que usaremos el siguiente pseudónimo para calcularla. La formula de la tasa de reducción es la siguiente:

$$tasa_reduccion = 100\frac{n^o\ de\ w_j < 0.2}{w.size}$$

El pseudocodigo seria el siguiente:

```
num_reducciones = 0
for i in pesos
        if pesos < 0.2
                num_reducciones++
end
tasa_reduccion = 100*(num_reducciones / pesos.size)
```

2.3 Calculo de la tasa de clase

La tasa de clase nos dará el porcentaje de acierto a la hora de entrenar nuestro modelo. Para ello utilizamos la siguiente formula:

$$tasa_clase = 100\frac{n^o\ de\ istancias\ bien\ clasificadas}{n^o\ instancias\ totales\ en\ test}$$

Para saber si una instancia esta bien clasificada se usara el algoritmos de 1-nn vecino mas cercano que explicaremos mas adelante. El pseudocodigo para calcular la tasa de clase seria:

```
num_aciertos = 0
for i in test
        clase_clasificado = clasificador_1NN(i,pesos)
        if clase(i) == clase_casificado
                num_aciertos += 0
end
tasa_clasificacion = 100*(num_aciertos / Y_test.size)
```

2.4 Función de evaluación

La función de evaluación es la encargada dar un valor numérico que representara lo bueno que es nuestro clasificador. Para ello utiliza tanto la tasa de reducción como la tasa de clase y un valor α.

La formula es la siguiente:

$$F(pesos) = tasa_clase(pesos) * \alpha + (1 - \alpha) * tasa_reduccion(pesos)$$

En este caso nosotros le daremos a α un valor de 0.5 para la practica. Esto hará que le demos la misma importancia la tasa de reducción y a la tasa de clase a la hora de ver como ajustan los datos. No voy a poner el pseudocódigo porque simplemente es implementar es función sin ninguna complicación mas.

3 Explicación de los algoritmos

En esta sección explicaremos los tres algoritmos que hemos utilizado para obtener los datos de clasificación y sus mediciones.

3.1 Algoritmo del vecino mas cercanos 1-NN

El algoritmos 1-NN o vecino mas cercano se entrena con las particiones de entrenamiento para poder clasificar las entradas de la partición de test. Se basa en el buscar el vecino mas cercano de las muestras de entrenamiento para una muestra de test. De este modo calcula la distancia euclídea de cada elemento de entrenamiento con respecto a un elemento de test. Selecciona el que mas se acerca y mira la clase de este. Si la clase coincide con la clase del elemento de entrenamiento se da como acertado.

En mi caso he implementado dos uno para obtener los datos del 1-NN en el que no necesitamos pasarle vector de pesos y que utilizo KNeighborsClassifier de la librería de python sklearn como clasificador.

El pseudocódigo es el siguiente:

```
        def K-NN(datos_train,datos_test){

                X_train = datos(datos_train)
                X_test = datos(datos_test)
                Y_train = datos(datos_train)
                Y_test = clase(datos_test)

                num_aciertos = 0
                #esto indica al clasificador que buscara uun unico vecion
                     cercano
                clasificador = crearClasificador(n=1)
                clasificador = clasificador.entreno(X_train,Y_train)

                num_elementos = getNumeroElementos(X_test)

                for i in [0,num_elementos]
                        clase = clasificador.predice_clase(Y_test[i])
                        if clase == Y_test[i]
                                num_aciertos += 1
                        end
                end
                tasa_acierto = 100 * (num_aciertos / Y_test.size)

                #como no trabaja con pesos el valor de tasa_reduccion sera 0
                return tasa_acierto,funcionObjetivo(tasa_acierto,0)
        }
```

3.2 Algoritmo RELIEF

El algoritmo RELIEF es una solución greedy que se basa en buscar el enemigo y vecino mas cercano para para mejorar el vector de pesos. El vector de pesos se inicia a 0.

La formula utilizada para calcular el nuevo vector de pesos es la siguiente:

$$pesos = pesos + |e_i - e_e| - |e_i - e_a|$$

Esto se realiza para cada elemento del conjunto de entrenamiento.

Una vez terminado se calcula la función objetivo clasificando mediante uno-NN pero antes hay que normalizar el vector de pesos por si alguno ha sobrepasado el uno o es menor que 0. De esta manera se coloca a 0 los menos que este y nos quedamos con el valor mayor del vector para normalizar respecto a este todos los valores.

```
        def RELIEF(X_train,Y_train,X_test, Y_test)
                num_elementos = getNumeroElementos(X_test)
                pesos = zeros(num_elementos)

                #creo una matriz de distancia para no tener que
                #estar calculando consantemente distancias para calcular
                #y repetir calculos
                distancias = disntacia_euclidea(X_train)

                #Recorremos la matriz de distancia comparando
                for i in [0,num_elementos]
                        mejor_enemigo = int()
                        valor_enemigo = max_float
                        mejor_amigo = int()
                        valor_amigo = max_float

                        for j in [0,num_elementos]

                                if Y_train[i] == Y_train[j]

                                        if valor_amigo < distancias[i][j]
                                                #Se evita que se el mimso el
                                                     mejor amigo
                                                if i != j
                                                        mejor_amigo = j
                                                        valor_amigo =
                                                             distancias[i][j]
                                                end

                                        end

                                else

                                        if valor_enemigo < distancias[i][j]
                                                mejor_enemigo = j
                                                valor_amigo = distancias[i][j
                                                     ]
                                        end

                                end

                        end

                        pesos = pesos + |X_train[i]-X_train[mejor_enemigo]| -
                        |X_train[i]-X_train[mejor_amigo]|
                end

                #normalizamos el vector de pesos si es menor

                max = obtenerMayor(pesos)

                for i in peso
                        if i < 0
                                i = 0
                        else
                                i = i / max
                end

                return uno-nn(X_train,Y_train,X_test,Y_test,pesos)

        end
```

3.3 Algoritmo de Búsqueda Local

En la búsqueda local iremos generando vecinos por mutación hasta un máximo de 15000. El pseudocódigo para la mutación es el siguiente. En mi caso he usado la librería randon de numpy. Si el peso supera 0 o 1 se trunca el peso.

```
        def mutacion(posicion,pesos)
                Z = np.random.normal(0.0, 0.3, None)
                pesos[posicion] += z
                if pesos[posicion] > 1
                        peso[posicion] = 1
                end
                if pesos[posicion] < 0
                        peso[posicion] = 0
                end

                return pesos
        end
```

El valor 0.3 es la varianza que utilizaremos para la mutación. Para generar el primer vector de pesos se tiene que realizar de forma aleatoria con valores entre [0,1]. Para eso utilizo también la librería numpy.random de python con la opción rand que genera valores entre 0 y 1

```
        def generar_vector(num_valores)
                pesos = np.random.rand(num_valores)
                return pesos
        end
```

La búsqueda local ira mutando un componente del vector y quedara este si mejora los valores que el anterior. Si no se da la mejora volverá al anterior y mutara el siguiente valor. Por cada fallo en la mutación se contara un erro de mejora. Si se obtiene 20*n errores en la mejora de forma consecutiva se para la búsqueda. El valor n corresponde al numero de características que tienen un elemento de nuestro dataset.

De esta forma ya tenemos dos criterios de parada, el generar 15000 vecionos o no mejorar en 20*n mutaciones de forma consecutiva.

Para mejorar un poco los tiempos antes de realizar una comprobación de si los pesos son mejores que los anteriores miro dos cosas.

La primera es que si antes de mutar una posición del vector de pesos esa posición era menor que 0.2 y al mutar sigue siendo menor que 0.2, sabemos que no tendrá mejora, por lo que no realizamos la comprobación.

La segunda opción que nos asegura que no se da mejora es cuando al mutar un gen el valor anterior era igual que el valor mutado. Esto parece poco probable pero cuando el valor de ese peso antes de la mutación ya era 1 si nos sale en la mutación un valor positivo no mejorara ese 1 puesto que es el valor máximo. En ese caso tampoco hago la comprobación de mejora.

El pseudocódigo es el siguiente;

```
        def BL(X_train,Y_train,X_test, Y_test)
                w = generar_vector(getNumeroCaracteristicas(X_train))
                pos_w = 0
                num_veciones = 0
                sin_mejora = 0

                #Se comprueba el valor de la solucion
                tasa_clase, tasa_reduccion = uno_nn(train_datos, train_clases
                     , test_datos, test_clases, w)

                funcion_mejora = funcionObjetivo(tasa_clase,tasa_reducion)

                mejor_valor_w = funcion_mejora
                mejor_w = w
                mejor_tasa_clase = tasa_clase
                mejor_tasa_reduccion = tasa_reduccion

                while continua_ejecutando(num_vecions,sin_mejora)
                        num_veciones += 1
                        anterior_peso = w[pos_w]
                        pesos = mutacion(pos_w,pesos)

                        #comprobamos si hay que calcular la mejora
                        if comprobarSiCalculamosMejora(anterior_peso,w[pos_w
                             ]):
                                w[pos_w] = anterior_peso
                                sin_mejora += 1

                        else
                                tasa_clase, tasa_reduccion = uno_nn(
                                     train_datos, train_clases, test_datos,
                                     test_clases, w)
                                funcion_mejora = funcionObjetivo(tasa_clase,
                                     tasa_reducion)


                                if mejor_valor_w < funcion_mejora:

                                        #Si mejora actualizamos los mejores
                                             valores
                                        mejor_w = w
                                        mejor_valor_w = funcion_mejora
                                        mejor_tasa_clase = tasa_clase
                                        mejor_tasa_reduccion = tasa_reduccion
                                        sin_mejora = 0


                                else:
                                        #volvemos al valor anterior y
                                             contamos una pasada sin mejora
                                        w[pos_w] = anterior_peso
                                        sin_mejora += 1
                                end

                        end


                        pos_w = obtenemosSigientePosicion(pos_w)

                end

                return uno-nn(X_train,Y_train,X_test,Y_test,pesos)
        end
```

4 Algoritmo de comparación

Para La comparación he usado el algoritmo uno-nn que nos compara las clases con el elemento de entrenamiento mas cercano a nuestro elemento de test. Nos clasificara la clase con la clase del elemento mas cercano que no sea el mismo.Tiene en cuenta que el mas cercano no puede ser el mismo.

La diferencia con el k-nn que use para el apartado anterior es que a este si le paso los datos y etiquetas de train y test ya separados para ahorrarle ese computo y un vector de pesos que utilizare para ponderar los datos. Esta función es la que utilizo en la búsqueda local para obtener si una solución es mejor que la anterior. También ahorro cálculos al quitar de los datos las columnas que tienen valores para los pesos menores que 0.2. Esto la hace un poco mas rápida que la anterior. También la uso en el algoritmos de RELIEF al trabajar con un vector de pesos.

Las tengo por separado simplemente porque una trabaja con vector de pesos y otra no, usando así la que mas me conviene en cada momento.

El pseudocódigo es el siguiente:

```
def uno-NN(X_train, Y_train, X_test, Y_test, pesos){

#Elimino las columnas estan en la mima posicion que un peso
#menor a 0.2
X_train = eliminarColumnas(Xtrain,pesos)
Y_train = eliminarColumnas(Xtrain,pesos)

#Elimino los pesos que tienen valor menor que 0.2
pesos_sin_minimos = eliminarInferiores(pesos)

X_train = X_train * pesos_sin_minimos
Y_test = Y_test * pesos_sin_minimos



num_aciertos = 0

#esto indica al clasificador que buscara uun unico vecion cercano
clasificador = crearClasificador(n=1)
clasificador = clasificador.entreno(X_train,Y_train)

num_elementos = getNumeroElementos(X_test)

for i in [0,num_elementos]
clase = clasificador.predice_clase(Y_test[i])
if clase == Y_test[i]
num_aciertos += 1
end
end
tasa_acierto = 100 * (num_aciertos / Y_test.size)
tasa_reduccion = tasaReduccion(pesos)

return tasa_acierto,tasa_reducion
}
```

5 Explicación de desarrollo de la práctica

La he realizado en python por la gran cantidad de librerías y código ya aportado por estas para el tratamiento y clasificación de este. Aparte de ser un leguaje que te permite implementar mas rápido los proyectos. Al ser python no hace falta hacer make ni ningún tipo de constructor para ejecutarla simplemente se ejecuta el fichero en el interprete de comandos.

En el main he puesto un for que cicla tres veces para calcular los datos de cada uno de los algoritmos con cada uno de los ficheros de datos dado. Los datos se muestran cuando se han ejecutado las 5 particiones distintas que podremos tener con todos los tres algoritmos de esta practica. Puede tardar en mostrar los datos debido a que los tiempos para la ejecución para la búsqueda local tardan mas que los demás. Esto retrasa la muestra de datos.

Si se quiere mostrar los datos de un solo algoritmo de búsqueda solo se tiene que comentar la llamada a este. Al tener las matrices donde almacenado los datos al inicio a 0 no dara error solamente mostrara los datos del algoritmos comentado como todo 0.

6 Análisis de resultados

6.0.1 Semilla

La semilla utilizada ha sido 14 y tengo un parámetro definido al principio del scrip llamado semilla para poder cambiarlo.

6.0.2 Valores utilizados

Los valores utilizados han sido los especificados en la practica, no cambien ninguno mas que para hacer algunas pruebas que no he tenido en cuenta para la practica. Los valores los tengo definidos al principio del script mediante variables para que se fácil la modificación sin tener que estar tocando todo el código.

6.0.3 Tablas de resultado

Tabla 5.1: 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	74.576271	32.258065	53.417168	0.077334	85.915493	2.941176	44.428335	0.122969	94.545455	5.0	49.772727	0.278389
Partición 2	82.456140	27.419355	54.937748	0.076381	87.142857	2.941176	45.042017	0.104845	92.727273	2.5	47.613636	0.238462
Partición 3	78.947368	24.193548	51.570458	0.078437	87.142857	2.941176	45.042017	0.105410	88.181818	20.0	54.090909	0.231540
Partición 4	70.175439	35.483871	52.829655	0.076579	87.142857	2.941176	45.042017	0.105282	94.545455	2.5	48.522727	0.225342
Partición 5	73.684211	24.193548	48.938879	0.078767	94.285714	2.941176	48.613445	0.105393	92.727273	12.5	52.613636	0.220764
Media	75.967886	28.709677	52.338782	0.077500	88.325956	2.941176	45.633566	0.108780	92.545455	8.5	50.522727	0.238899

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	71.186441	0.0	35.593220	0.025666	84.507042	0.0	42.253521	0.030654	91.818182	0.0	45.909091	0.043423
Partición 2	78.947368	0.0	39.473684	0.024994	88.571429	0.0	44.285714	0.030145	91.818182	0.0	45.909091	0.042935
Partición 3	77.192982	0.0	38.596491	0.025259	84.285714	0.0	42.142857	0.030164	87.272727	0.0	43.636364	0.041187
Partición 4	70.175439	0.0	35.087719	0.025259	85.714286	0.0	42.857143	0.030123	92.727273	0.0	46.363636	0.041029
Partición 5	73.684211	0.0	36.842105	0.024933	90.000000	0.0	45.000000	0.030081	93.636364	0.0	46.818182	0.042407
Media	74.237288	0.0	37.118644	0.025222	86.615694	0.0	43.307847	0.030233	91.454545	0.0	45.727273	0.042196

Tabla 5.1: 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	81.355932	53.225806	67.290869	38.819485	94.366197	82.352941	88.359569	27.224013	98.181818	80.0	89.090909	79.786123
Partición 2	89.473684	69.354839	79.414261	125.77766	95.714286	79.411765	87.563025	37.040083	95.454545	77.5	86.477273	38.796238
Partición 3	82.456140	58.064516	70.260328	50.393600	95.714286	82.352941	89.033613	23.723314	88.181818	77.5	82.840909	42.283510
Partición 4	80.701754	62.903226	71.802490	62.182601	97.142857	88.235294	89.747899	41.164473	92.727273	75.0	83.863636	80.049610
Partición 5	84.210526	69.354839	76.782683	94.267057	98.571429	79.411765	88.991597	42.310930	98.181818	75.0	86.590909	37.313678
Media	83.639607	62.580645	73.110126	74.288081	96.301811	81.176471	88.739141	34.292563	94.545455	77.0	85.772727	55.645832

Tabla 5.2: Resultados globales en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
RELIEF	75.967886	28.709677	52.338782	0.077500	88.325956	2.941176	45.633566	0.108780	92.545455	8.5	50.522727	0.238899
1-NN	74.237288	0.0	37.118644	0.025222	86.615694	0.0	43.307847	0.030233	91.454545	0.0	45.727273	0.042196
BL	83.639607	62.580645	73.110126	74.288081	96.301811	81.176471	88.739141	34.292563	94.545455	77.0	85.772727	55.645832

Figure 1:

6.0.4 Análisis de resultados

Los resultados obtenidos nos muestra que la búsqueda local es muy superior a las otras en cuanto a obtener mejores resultados y la que mayor tasa de reducción. Esto se debe a que es la que mas opciones comprueba y siempre se va quedando con la mejor. Con esto evita perder buenas soluciones. Ademas al solo aceptar la mutación cuando mejora tiende a mejorar rápido hasta que se cumplen las 20*n iteraciones sin mejora. Estas iteraciones sin mejora son siempre la causa de que termine el algoritmo sin llegar nunca a el máximo de vecinos generado. Suele generar unos 7000 vecinos la vez que mas y normalmente ronda entre 2000 y 4000 en textura por ejemplo.

El siguiente que mejor resultados me suele dar es el RELIEF aunque el 1-nn no se diferencia tanto de el. Esto se trata a que el RELIEF si utiliza los datos de entrenamiento para intentar mejorar mediante la ponderación de los pesos. El 1-nn depende de lo parecidos que sean las particiones de entrenamiento y test puesto que no obtienen ningún tipo de mejora para aproximar sus datos a los de entrenamiento.

Algo curioso es que en los ficheros de IONOSPHERE RELIEF siempre me reduce el mismo numero de pesos. Imagino que podrá ser por pesos que son muy parecidos para todos los elementos de la muestra. Y por eso los tiende a

reducir al no ser significativos.

El mejor conjunto de datos para el ajuste es textura. Llegando a obtener mas de 90% de media para todos los datos. El mas difícil de clasificar correctamente es el colposcopy. Aunque en la función de evaluación mejora mucho colopscopy porque suele ser tener una buena tasa de reducción. Y al darle la misma importancia a la clasificación y la reducido eso hace que se la que mejor función objetivo tenga. Esto no se cumple por ejemplo en el 1-nn al no tener reducción es la que peor función objetivo tiene

En la búsqueda local me sorprende que se capaz de ajustar tanto teniendo tasa reducción tan altas. Entiendo que con eso evitara mucho ruido que dan los datos eliminados

6.0.5 Tiempo de ejecución

El tiempo de ejecución de menor a mayor como era obvio el que menos tarda es el 1-nn puesto que los otros dos utilizan a este para la clasificación. Es prácticamente instantáneo

Lo sigue el RELIEF que si el anterior tarda como el doble triple que esta pero tampoco es un tiempo considerable.

La búsqueda local es mucho mas lenta. Conseguí reducirla mucho al tener en cuenta que hay ocasiones en las que no hacia falta volver a valorar un peso porque no es posible que este mejore. Llegue a bajar de tardar unos 10 minutos de media partición a 1.25 minutos la que mas me puede tardar. También es verdad que depende de la semilla en algunas condiciones. Siendo los datos en lo que mas tarda los que mas numero de características por elemento tienen.

7 Bibliografía

No he usado nada fuera de las propias paginas de información de python o de las librerías utilizadas como podría ser la de numpy.