Inteligencia artificial. Práctica 3.

Programación lógica.

Abril, 2021

Introducción

Esta práctica te ayudará a dominar la programacion lógica. De paso, se proponen funcionalidades muy guiadas de aprendizaje automático para que aprendas programando y la siguiente práctica sea mas sencilla. Resuelve los siguientes ejercicios. No te olvides de hacer una exhaustiva batería de tests para comprobar que los ejercicios están bien. Recuerda que el comando `trace/0` de Prolog te ayuda a depurar el código de Prolog con facilidad.

Entrenamiento Os recomendamos también echar un vistazo a la colección de 99 problemas de Prolog publicados en `https://www.ic.unicamp.br/~meidanis/courses/mc336/2009s2/prolog/problemas/`. Ahí puedes encontrar una gran serie de problemas con código que te pueden ayudar como entrenamiento para solucionar los ejercicios propuestos en esta práctica si al principio te resultan complicados.

Entrega Se debe entregar un único fichero comprimido llamado:
code_XXXX_YY_name1_surname1_name2_surname2.zip.
Donde **XXXX** es el identificador del grupo y **YY** es el de la pareja. Por favor respetad estrictamente este formato. Ese fichero .zip contendrá el código **code_XXXX_YY_name1_surname1_name2_surname2.pl** y la memoria relativa al apartado 7. **Tienes hasta el 21-23 de Abril a las 09:00 horas, dependiendo de si tienes clase el Miércoles, Jueves o Viernes.** Se deben respetar las cabeceras suministradas en `cabecera.pl` ya que se usa un corrector automático que asume que los predicados tienen los nombres y parámetros que os proporcionamos en la cabecera.

Gestión de errores Este corrector automático también verifica la correcta gestión de errores pedida en cada ejercicio. Las trazas de error que vuestros programas imprimen deben coincidir con la que os damos. Para escribir en el fichero log de errores se proporciona en el código de `cabecera.pl` el predicado `escribir_log/1` que imprime en el fichero `error_logs.txt` las trazas que se le pasan como argumento.

1 Suma de los productos elevados a una potencia. (1 punto)

Sean `X` e `Y` dos vectores cuyos elementos son numeros reales. Haz un predicado en Prolog `sum_pot_prod/4` de argumentos `sum_pot_prod(X, Y, Potencia, Resultado)` que devuelva el sumatorio `Resultado` del producto de los elementos de los vectores `X` e `Y` elevados a la potencia dada por el valor `Potencia` que se pasa por argumento. Es decir, se debe calcular:

$$z = \sum_{i=1}^{l} (x_i y_i)^p \, .$$

Donde p es la potencia, z es el resultado final de la operación, l es la longitud de los vectores $\mathbf{x}$ e $\mathbf{y}$ y x_i e y_i son los elementos referenciados por el índice i de los vectores $\mathbf{x}$ e $\mathbf{y}$. Esta potencia debe ser positiva. Los vectores deben tener la misma longitud. Si se da el valor 1 a la potencia, esta operación es equivalente al producto escalar. Ejemplos:

```
sum_pot_prod([1,2,3],[3,4,5], 1, X). X = 26.
sum_pot_prod([1,2,3],[3,4,5], 2, X). X = 298.
sum_pot_prod([1,2,3],[3,4,5], -1, _).  false.   Imprime: ERROR 1.1 Potencia.
sum_pot_prod([1,2,3],[3,4,5,6], 3, _).  false.   Imprime: ERROR 1.2 Longitud.
```

2 Segundo y penúltimo. (1 punto)

Haz un predicado `segundo_penultimo/3` de argumentos `segundo_penultimo(L, X, Y)` que devuelva el segundo elemento `X` y el penúltimo elemento `Y` de una lista `L`. La lista debe tener longitud mayor que 1. El intérprete debe devolver una única solución. Ejemplos:

```
segundo_penultimo([1,2], X, Y). X = 2.  Y = 1.
segundo_penultimo([1,2,3], X, Y). X = 2.  Y = 2.
segundo_penultimo([1,2,3,4], X, Y). X = 2.  Y = 3.
segundo_penultimo([1], X, Y). false. Imprime: ERROR 2.1 Longitud.
```

3 Sublista que contiene un elemento. (1.5 puntos)

Haz un predicado `sublista/5` de argumentos `sublista(L, Menor, Mayor, E, Sublista)` que devuelva la sublista `Sublista` a partir de la lista `L` dada por dos indices `Menor` y `Mayor`. Esta sublista debera ademas contener el elemento `E` dado como cuarto elemento. Sino, fallara. El indice empieza por 1. La sublista incluye los dos elementos referenciados por los indices `Menor` y `Mayor`. Comprueba que los indices son correctos. Es decir, `Menor <= Mayor` y `Mayor <= longitud_lista(L)`. Ejemplos:

```
sublista(['a','b','c','d','e'], 2, 4, 'b', X). X = ['b','c','d'].
sublista(['a','b','c','d','e'], 2, 4, 'f', X). false. Imprime: ERROR 3.1 Elemento.
sublista(['a','b','c','d','e'], 5, 4, 'b', X). false. Imprime: ERROR 3.2 Indices.
```

4 Espacio lineal. (1.5 puntos)

Haz un predicado `espacio_lineal/4` de argumentos `espacio_lineal(Menor, Mayor, Numero_elementos, Rejilla)` que genere una rejilla lineal (lista de numeros reales) `Rejilla` de un numero de puntos `Numero_elementos` en un intervalo dado `[Menor, Mayor]`. Una rejilla lineal es un vector de elementos entre los numeros `Menor` y `Mayor` (incluídos) tales que la distancia entre cada elemento es igual y cada elemento es mayor que el anterior. `Menor` y `Mayor` deben contenerse en la rejilla lineal `Rejilla`. Comprueba que los indices son correctos. Los ejemplos ilustran con mayor claridad la funcionalidad que se pide. Ejemplos:

```
espacio_lineal(0, 1, 5, L).
L = [0, 0.25, 0.5, 0.75, 1.0].

espacio_lineal(0, 1, 100, L).
L=[0, 0.01010101, 0.02020202, 0.03030303, 0.04040404, 0.05050505, 0.06060606, 0.07070707,
0.08080808, 0.09090909, 0.1010101 , 0.11111111, 0.12121212, 0.13131313, 0.14141414, 0.15151515,
0.16161616, 0.17171717, 0.18181818, 0.19191919, 0.2020202 , 0.21212121, 0.22222222, 0.23232323,
0.24242424, 0.25252525, 0.26262626, 0.27272727, 0.28282828, 0.29292929, 0.3030303 , 0.31313131,
0.32323232, 0.33333333, 0.34343434, 0.35353535, 0.36363636, 0.37373737, 0.38383838, 0.39393939,
0.4040404 , 0.41414141, 0.42424242, 0.43434343, 0.44444444, 0.45454545, 0.46464646, 0.47474747,
0.48484848, 0.49494949, 0.50505051, 0.51515152, 0.52525253, 0.53535354, 0.54545455, 0.55555556,
0.56565657, 0.57575758, 0.58585859, 0.5959596 , 0.60606061, 0.61616162, 0.62626263, 0.63636364,
0.64646465, 0.65656566, 0.66666667, 0.67676768, 0.68686869, 0.6969697 , 0.70707071, 0.71717172,
0.72727273, 0.73737374, 0.74747475, 0.75757576, 0.76767677, 0.77777778, 0.78787879, 0.7979798
, 0.80808081, 0.81818182, 0.82828283, 0.83838384, 0.84848485, 0.85858586, 0.86868687, 0.87878788,
0.88888889, 0.8989899 , 0.90909091, 0.91919192, 0.92929293, 0.93939394, 0.94949495, 0.95959596,
0.96969697, 0.97979798, 0.98989899, 1]

espacio_lineal(2,0,3,L). false. Imprime: ERROR 4.1 Indices.
```

5 Normalizar distribuciones de probabilidad. (1 punto)

Haz un predicado `normalizar/2` de argumentos `normalizar(Distribucion_sin_normalizar, Distribucion)` que normalize la distribucion univariante sin normalizar `Distribucion_sin_normalizar` en la variable `Distribucion`. Para normalizar una distribucion univariante, debes obtener la constante de normalizacion z de la misma. Esta viene dada por la integral de la distribucion en su soporte (suma de todos los elementos de la distribucion univariante).

$$z = \int p(x)dx\,.$$

Donde la integral se calcula en el soporte (rango del espacio) donde la distribución está definida y la x representa a cada elemento de la integral. No te asustes por la integral del enunciado. Recuerda que, en esencia, la integral es una suma. Si tenemos, como es el caso, la distribución univariante representada por una lista $p(\mathbf{x})$, la integral anterior se puede aproximar por la suma de los elementos de esa lista.

$$z \approx \sum_{i=1}^{l} p(x_i)\,.$$

Donde $p(x_i)$ es el elemento de la lista referenciado por el índice i y l es el tamaño de la lista. En otras palabras, para estimar la constante de normalización de una distribución representada por una lista basta con sumar los elementos de esa lista. Todos los elementos de la distribucion sin normalizar deben ser positivos.

Para normalizar la distribución dada como argumento `Distribucion_sin_normalizar` debes dividir cada uno de los elementos de la distribucion `Distribucion_sin_normalizar` por la constante de normalizacion z. Una buena forma de testear que has hecho bien este ejercicio es verificar que el vector de salida de tu programa representa a una funcion de densidad de probabilidad de una distribucion, es decir, su integral (aproximada por la suma de todos los elementos) debe ser igual a 1. Ejemplos:

```
normalizar([3,4,5], X).
X = [0.25, 0.3333333333333333, 0.4166666666666667].
normalizar([1,2,3,4,5], X)
X = [0.06666666666666667, 0.13333333333333333, 0.2, 0.26666666666666666, 0.3333333333333333].
normalizar([-1,2,3,4,5], X). false. Imprime: ERROR 5.1.  Negativos.
```

6 Divergencia de Kullback-Leibler. (1.5 punto)

Al igual que sabemos mediante el valor absoluto de la resta la distancia entre dos números reales o empleamos la distancia euclídea para mostrar una distancia entre dos números que pertenecen a $\mathbb{R}^d$ donde d es un número entero mayor que 1 también podemos calcular una noción de distancia o una discrepancia (no exactamente una distancia desde el punto de vista formal) entre dos distribuciones de probabilidad. Esta divergencia nos indica como de semejantes son dos distribuciones de probabilidad. Por ejemplo, dos distribuciones idénticas tienen divergencia 0. Al igual que existen varias métricas de distancia entre puntos que pertenecen a $\mathbb{R}^d$ como la Manhattan o la euclídea, existen varias discrepancias entre distribuciones. Un ejemplo de divergencia es la divergencia de Kullback Leibler. Haz un predicado `divergencia_kl/3` de argumentos `divergencia_kl(D1, D2, KL)` que calcule la divergencia de Kullback Leibler `KL` de dos distribuciones `D1` y `D2`. La divergencia KL entre dos distribuciones de probabilidad discretas univariantes $p(x)$ y $q(x)$ se calcula mediante la siguiente expresión (utilizado el truco de aproximar una integral por un sumatorio comentado en el ejercicio anterior):

$$D_{KL}(p(x)||q(x)) = \int p(x)\ln\frac{p(x)}{q(x)}dx \approx \sum_{x=1}^{N} p(x_i)\ln\frac{p(x_i)}{q(x_i)}\,.$$

Es decir, para cada elemento de los dos vectores $p(\mathbf{x})$ y $q(\mathbf{x})$ que tienes como entrada tienes que calcular $p(x_i)\ln\frac{p(x_i)}{q(x_i)}$ e ir sumando estas cantidades. Notifica si algun elemento tiene valor cero o menor de cero. La divergencia no esta definida para distribuciones que tienen valores iguales a 0 en su soporte. Notifica al usuario si alguno de los vectores pasados como argumento no se trata de una funcion de densidad de probabilidad de una distribución. Para ello, te puede venir bien usar codigo del ejercicio anterior. Ejemplos:

```
divergencia_kl([0.2, 0.3, 0.5], [0.2, 0.3, 0.5], D). D = 0.0
```
Las distribuciones son iguales, la divergencia KL es 0.
```
divergencia_kl([0.5, 0.3, 0.2], [0.2, 0.3, 0.5], D). D = 0.27488721956224654.
divergencia_kl([0.98, 0.01, 0.01], [0.01, 0.01, 0.98], D). D = 4.447418454310455.
```
Distribuciones muy diferentes tienen alta divergencia KL.
```
divergencia_kl([0.99, 0.0, 0.01], [0.01, 0.01, 0.98], D). false.
```
Imprime: `ERROR 6.1.  Divergencia KL no definida.`
```
divergencia_kl([0.2,0.3,0.6], [0.1,0.5,0.4], D). false.
```
Imprime: `ERROR 6.2.  Divergencia KL definida solo para distribuciones.`

7 Comentario del problema de las 8 damas. (2.5 puntos)

Acude a la batería de ejercicios de Prolog `https://www.ic.unicamp.br/~meidanis/courses/mc336/2009s2/prolog/problemas/`. Lee el enunciado y el código de la solución del problema 90: Eight queens problem. Usa `trace/0` para comprender el funcionamiento del código. Redacta un comentario del funcionamiento de la implementación de Prolog de este problema en un documento **memoria.pdf** que puedes entregar junto al resto del código. El comentario debe incluir: Descripción del problema de las 8 reinas. Aporta alguna referencia. Explicación del algoritmo que soluciona el problema (no simplemente una descripción sino aporta un razonamiento de los pasos efectuados). Descripción detallada de los predicados de Prolog usados en la solución. Capturas de pantalla comentadas de la ejecución del código. Comentario informal sobre la solución: ¿Te parece la adecuada? ¿Se te habría ocurrido otra forma de resolverlo?

8 OPCIONAL: Matriz por bloques dada por el producto de Kronecker.

Del mismo modo que hay definidas varias distancias entre elementos de un mismo espacio, también hay definidos varios productos entre, en este caso, matrices. Imagina que tienes dos matrices cuadradas 2x2 de números reales **A** y **B**:

$$\mathbf{A} = \begin{pmatrix} 2 & 5 \\ 3 & 4 \end{pmatrix}\,, \quad \mathbf{B} = \begin{pmatrix} 1 & 7 \\ 9 & 3 \end{pmatrix}\,.$$

Imagina que quieres obtener como resultado de estas matrices una matriz $\mathbf{C}$ que contenga tantos elementos C_{ij} como posibles multiplicaciones podamos calcular sobre los elementos A_{ij} y B_{ij} de las matrices $\mathbf{A}$ y $\mathbf{B}$. Es decir, la matriz por bloques:

$$\mathbf{C} = \begin{pmatrix} \begin{pmatrix} 2*1 & 2*7 \\ 2*9 & 2*3 \end{pmatrix} & \begin{pmatrix} 5*1 & 5*7 \\ 5*9 & 5*3 \end{pmatrix} \\ \begin{pmatrix} 3*1 & 3*7 \\ 3*9 & 3*3 \end{pmatrix} & \begin{pmatrix} 4*1 & 4*7 \\ 4*9 & 4*3 \end{pmatrix} \end{pmatrix}.$$

Este producto puede ser útil para no perder información de ningún elemento y ganar información del producto de los elementos de dos matrices. Antes de exponer la generalización de este producto, llamemos a esta operación producto de Kronecker sobre dos matrices $\mathbf{A}$ y $\mathbf{B}$ y denotémosla como $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$. El producto de kronecker no tiene por qué requerir que las matrices sean cuadradas, como en el caso anterior. Puede darse entre matrices de distintas dimensiones como en este ejemplo:

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix}, \quad \mathbf{Y} = \begin{pmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \end{pmatrix}.$$

$$\mathbf{Z} = \begin{pmatrix} \begin{pmatrix} x_{11}y_{11} & x_{11}y_{12} & x_{11}y_{13} \\ x_{11}y_{21} & x_{11}y_{22} & x_{11}y_{23} \end{pmatrix} & \begin{pmatrix} x_{12}y_{11} & x_{12}y_{12} & x_{12}y_{13} \\ x_{12}y_{21} & x_{12}y_{22} & x_{12}y_{23} \end{pmatrix} \\ \begin{pmatrix} x_{21}y_{11} & x_{21}y_{12} & x_{21}y_{13} \\ x_{21}y_{21} & x_{21}y_{22} & x_{21}y_{23} \end{pmatrix} & \begin{pmatrix} x_{22}y_{11} & x_{22}y_{12} & x_{22}y_{13} \\ x_{22}y_{21} & x_{22}y_{22} & x_{22}y_{23} \end{pmatrix} \\ \begin{pmatrix} x_{31}y_{11} & x_{31}y_{12} & x_{31}y_{13} \\ x_{31}y_{21} & x_{31}y_{22} & x_{31}y_{23} \end{pmatrix} & \begin{pmatrix} x_{32}y_{11} & x_{32}y_{12} & x_{32}y_{13} \\ x_{32}y_{21} & x_{32}y_{22} & x_{32}y_{23} \end{pmatrix} \end{pmatrix}.$$

A partir de esta definición, haz un predicado `producto_kronecker/3` de parametros `producto_kronecker(Matriz_A, Matriz_B, Matriz_bloques)` que calcule el producto de Kronecker `Matriz_bloques` de dos matrices `Matriz_A` y `Matriz_B`. Si cualquiera de los elementos de la matriz resultante es menor que cero, entonces, se debe interrumpir el calculo y advertir al usuario de la causa del fallo. Se debe obtener una matriz a bloques donde, si te fijas en los ejemplos y a modo de pista, cada bloque es la multiplicacion de un elemento de la primera matriz por la otra matriz. Otra pista: Enfoca este ejercicio siguiendo una metodologia de abajo a arriba. Es decir, de funcionalidad mas simple a mas compleja. No dudes en usar varios niveles de recursión para solucionar este ejercicio. De eso se trata. Ejemplos:

`producto_kronecker([[1,2],[3,4]], [[0,5], [6,7]], R).`

$$\mathbf{R} = \begin{pmatrix} \begin{pmatrix} 0 & 5 \\ 6 & 7 \end{pmatrix} & \begin{pmatrix} 0 & 10 \\ 12 & 14 \end{pmatrix} \\ \begin{pmatrix} 0 & 15 \\ 18 & 21 \end{pmatrix} & \begin{pmatrix} 0 & 20 \\ 24 & 28 \end{pmatrix} \end{pmatrix}.$$

```
R = [[[[0, 5], [6, 7]], [[0, 10], [12, 14]]], [[[0, 15], [18, 21]], [[0, 20], [24, 28]]]].
producto_kronecker([[-1,2],[3,4]], [[0,5], [6,7]], R). false
```

Imprime: `ERROR 7.1.  Elemento menor que cero.`

9 OPCIONAL: K vecinos próximos.

El algoritmo K vecinos próximos es un algoritmo de aprendizaje automático. Sin embargo, no es necesario disponer de conocimientos de aprendizaje automático para resolver este ejercicio, ya que está guíado. No obstante, realizar la implementación de este algoritmo en Prolog para dominar la programación lógica y observar los resultados que da este algoritmo sirve como un excelente prólogo para aprender aprendizaje automático, sobre lo que versa la última práctica de la asignatura. Veamos por tanto para aquellos alumnos interesados un pequeñísimo avance de aprendizaje automático que servirá para obtener la intuición de su funcionamiento y del tipo de problemas que resuelven estos algoritmos.

El algoritmo de aprendizaje automático K vecinos próximos es capaz de, por un mecanismo sencillo, predecir a qué clase de entre un conjunto de clases pertenece una serie de datos. Este algoritmo es capaz

de predecir esta información ya que ha sido entrenado con anterioridad sobre un conjunto de datos en el que se suministraba a cada serie de datos a qué clase pertenecían. Por ejemplo, una serie de datos, que llamaremos instancia, contiene las calificaciones de un alumno en su último curso de bachillerato. La clase que le suministramos es la carrera que ha elegido. De este modo, por ejemplo, una instancia podría contener que un alumno ha sacado muy buena nota en física y matemáticas en comparación con el resto de asignaturas y ha decidido estudiar Ingeniería Informática en la UAM. Si muchas instancias contienen un patrón similar, el algoritmo K vecinos próximos, cuando se le presenta una nueva instancia con un patrón similar de la cuál desconocemos la clase, en este caso la carrera que estudiará el alumno, predecirá que el alumno estudiará Ingeniería Informática. Los algoritmos de aprendizaje automático para clasificación por aprendizaje supervisado, que es como se define el problema descrito, funcionan por tanto en 2 etapas. Entrenamiento, en el que se les presenta un conjunto de datos llamado conjunto de entrenamiento donde cada instancia viene anotada con la clase y el algoritmo calcula una configuración para minimizar el error de predicción. La otra etapa es la predicción, donde el algoritmo toma una instancia de datos sin etiquetar y debe determinar su clase.

Implementa un predicado `k_vecinos_proximos/5` de parámetros `k_vecinos_proximos(X_entrenamiento, Y_entrenamiento, K, X_test, Y_test)` que codifique el algoritmo k vecinos próximos. Iremos resolviendo la implementación de este algoritmo de forma guiada, en sucesivos apartados. El k vecinos próximos usará la distancia euclídea para su funcionamiento. Se recibe una matriz `X_entrenamiento`. Esta matriz tendrá N instancias, donde cada instancia es una lista de números reales. Cada instancia tiene M caracteristicas, es decir la lista que la representa tiene longitud M. Por otro lado, se recibe una lista `Y_entrenamiento` de N elementos o etiquetas que representan una variable categórica. Cada elemento de `Y_entrenamiento` está asociado respectivamente a una instancia de `X_entrenamiento`. Es decir, la primera etiqueta de `Y_entrenamiento` corresponde a la etiqueta de la primera instancia de `X_entrenamiento` y así sucesivamente. El algoritmo `k_vecinos_proximos/5` determinará la etiqueta de cada instancia del conjunto de test `X_test`, que al igual que `X_entrenamiento` es una matriz. Sin embargo, en el caso de `X_test` desconocemos sus etiquetas. Por ello `k_vecinos_proximos/5` predecirá las etiquetas de `X_test` y las guardará en `Y_test`. Como veremos mas adelante, la etiqueta de cada elemento sera la determinada por la mayoria de las etiquetas guardadas en un vector de K posiciones. Este vector de K posiciones correspondiente a cada instancia de `X_test` (de ahí el nombre `k_vecinos_proximos/5`) será el correspondiente a las K instancias de `X_entrenamiento` cuya distancia euclídea sobre la distancia de `X_test` sea menor. A no ser que se diga lo contrario, este ejercicio asume que errores verificados anteriormente no deben ser verificados ahora, es decir, que los datos de entrada son correctos. Para implementar este clasificador, sigue los siguientes pasos:

9.1 Distancia euclídea.

Escribe un predicado `distancia_euclidea/3` con parametros `distancia_euclidea(X1, X2, D)` que devuelva la distancia euclidea `D` entre las instancias `X1` y `X2`. Recordar que si tenemos dos vectores $\mathbf{x}$ e $\mathbf{y}$ tales que $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$ donde k es un número entero mayor que 0, entonces, la distancia euclidea $d(\mathbf{x}, \mathbf{y})$ entre ambos vectores es igual a la siguiente expresión.

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_i - y_i)^2 + ... + (x_k - y_k)^2}\,.$$

Ejemplos:

```
distancia_euclidea([1,4,3], [1,2,1], D).
D = 2.8284271247461903.
distancia_euclidea([1,2,1], [1,2,1], D).
D = 0.0.
```

9.2 Cálculo de las distancias entre puntos.

Haz un predicado `calcular_distancias/3` con parametros `calcular_distancias(X_entrenamiento, X_test, Matriz_resultados)` que obtenga una matriz de distancias `Matriz_resultados` donde cada fila es la distancia de un punto de test que esta en `X_test` con respecto a todos los puntos del conjunto de entrenamiento `X_entrenamiento`. Pista: piensa que deberás usar un doble procedimiento de recursión para construir esta

matriz. Ejemplos:

```
calcular_distancias([[1,2,3],[1,2,4],[2,3,1]], [[1,2,1], [1,2,5]], D).
D = [[2.0, 3.0, 1.4142135623730951], [2.0, 1.0, 4.242640687119285]].
calcular_distancias([[1,2,3,4],[1,2,4,5],[2,3,1,5]], [[1,2,1,2], [1,2,5,2], [1,2,3,4], [5,6,7,8],
[9,10,11,12]], D).
D = [[2.8284271247461903, 4.242640687119285, 3.3166247903554], [2.8284271247461903,
3.1622776601683795, 5.196152422706632], [0.0, 1.4142135623730951, 2.6457513110645907], [8.0,
7.0710678118654755, 7.937253933193772], [16.0, 15.033296378372908, 15.716233645501712]].
```

9.3 Predicción de las clases de puntos de test.

Haz un predicado `predecir_etiquetas/4` de parametros `predecir_etiquetas(Y_entrenamiento, K, Matriz_resultados, Y_test)` que calcule las predicciones `Y_test` a partir de la matriz de distancias `Matriz_resultados` calculada en el apartado anterior. Si te fijas, en este punto ya no necesitamos las instancias de entrenamiento ni las de test, puesto que la información útil de ellos ya está codificada en `Matriz_resultados`. Recuerda que en cada fila de `Matriz_resultados` tenemos la distancia de un punto de test con respecto a todos los puntos de entrenamiento. La etiqueta de cada punto de test corresponde a su fila en `Matriz_resultados`. Ejemplos:

```
predecir_etiquetas(['a','b','a','a','a','a','d'], 3, [[12,32,11,3,44,32,0], [1,2,3,4,5,6,7]],
Etiquetas).
Etiquetas = ['a','a'].
predecir_etiquetas(['c','b','a','a','a','a','d'], 1, [[12,32,11,3,44,32,0], [1,2,3,4,5,6,7]],
Etiquetas).
Etiquetas = ['d','c']
```

Para ello, antes de implementar el código de este predicado, se aconseja implementar el siguiente predicado:

9.3.1 Predicción de la clase del punto de test.

Haz un predicado `predecir_etiqueta/4` de parametros `predecir_etiqueta(Y_entrenamiento, K, Vec_distancias, Etiqueta)` que calcule en `Etiqueta` la etiqueta mas cercana segun `Vec_distancias` de cada punto de test cuya información útil reside en `Vec_distancias`. Las etiquetas de cada punto estan en `Y_entrenamiento`. `Vec_distancias` hace referencia a cada fila de `Matriz_resultados`. Ejemplos:

```
predecir_etiqueta(['a','b','a','a','a','a','d'], 3, [12,32,11,3,44,32,0], Etiqueta).
Etiqueta = 'a'
predecir_etiqueta(['a','b','a','a','a','a','d'], 1, [12,32,11,3,44,32,0], Etiqueta).
Etiqueta = 'd'
```

Para ello, antes de implementar el código de este predicado, se aconseja implementar los siguientes predicados:

Cálculo de las etiquetas mas relevantes Haz un predicado `calcular_K_etiquetas_mas_relevantes/4` de parametros `calcular_K_etiquetas_mas_relevantes(Y_entrenamiento, K, Vec_distancias, K_etiquetas)` que construya el vector de etiquetas `K_etiquetas` de tamaño `K` del punto de test (omitido en el código) pero cuya información útil para la implementación de este predicado se encuentra en el vector de distancias `Vec_distancias`. Pista: hay varias formas de resolver este ejercicio. Pista: para hacer este ejercicio, se aconseja implementar funciones de soporte como insertar en orden en una lista y eliminar el primer elemento de una lista. Otra pista: Es posible que tengas que hacer una lista de tuplas donde cada tupla consista en la distancia de un punto y su etiqueta. Ejemplos:

```
   calcular_K_etiquetas_mas_relevantes(['a','b','a','b','c','a','d'], 1, [12,32,11,3,1,2,0],
Etiquetas).
   Etiquetas = ['d'].
   calcular_K_etiquetas_mas_relevantes(['a','b','a','a','a','a','d'], 3, [12,32,11,3,44,32,0],
Etiquetas).
   Etiquetas = ['a','a','d'].
```

Para ello, antes de implementar el código de este predicado, se aconseja implementar el siguiente predicado:

Cálculo de la etiqueta mas relevante Haz un predicado `calcular_etiqueta_mas_relevante/2` de parametros `calcular_etiqueta_mas_relevante(K_etiquetas, Etiqueta)` que a partir de una lista de etiquetas `K_etiquetas` saque la mas comun en `Etiqueta`. Es decir, la etiqueta que aparece un número mayor de veces en la lista `K_etiquetas`. Pista: es posible que tengas que implementar funciones de soporte como si un elemento es miembro de una lista. Ejemplos:

```
   calcular_etiqueta_mas_relevante(['a','c','c'], E).
   E = 'c'.
   calcular_etiqueta_mas_relevante(['a','b','a','c','d','b','a','c','a'], E).
   E = 'a'.
```

Pista: es posible que quieras en este ejercicio convertir la lista ['a','b','a','c','d','b','a','c','a'] a una lista de tuplas del estilo: [[1,'d'],[2,'c'],[2,'b'],[4,'a']]. Puedes hacerlo con una función auxiliar `calcular_contadores/2`. Ejemplos:

```
   calcular_contadores(['a','b','a','c','c','c'], [[3, 'c'], [1, 'b'], [2, 'a']]).
   L = [[3, 'c'], [1, 'b'], [2, 'a']].
```

9.4 Ensamblando los predicados.

Haz un predicado `k_vecinos_proximos/5` de parametros `k_vecinos_proximos/5(X_entrenamiento, Y_entrenamiento, K, X_test, Y_test)` cuya implementación contenga los predicados `calcular_distancias/3` y `predecir_etiquetas/4`. Con ello: ¡ya habrías terminado la implementación del clasificador!. Ejemplos:

```
   k_vecinos_proximos([[2,3,1], [3,4,5], [2,3,2], [4,5,6], [6,5,8], [1,0,2], [3,4,3]],
['a','b','a','c','c','a','b'], 1, [[10,7,5], [2,3,1]], Etiquetas).
   Etiquetas = ['c','a'].
   k_vecinos_proximos([[2,3,1], [3,4,5], [2,3,2], [4,5,6], [6,5,8], [1,0,2], [3,4,3]],
['a','b','a','c','c','a','b'], 7, [[3,4,3], [10,10,10]], Etiquetas).
   Etiquetas = ['a','a'].
```