Facultad de Ingeniería

Universidad de Buenos Aires

Departamento de electrónica

86.51 Teoría de Detección y Estimación

1^{er} cuatrimestre 2019

Trabajo Final:
Métodos de estimación y clasificación no paramétricos

Autora:

Cristina Kuo (97777)
cristinaa.kuo@gmail.com

8 de Julio de 2019

Datos

Se estimarán dos funciones con las siguientes distribuciones: $P(\omega_1) = 0{,}5$ y $P(\omega_2) = 0{,}5$
$F_1 = U[2,4]$ y $F_2 = Gaussiana(2,4)$
En cuanto a la ventana de Parzen, se utilizará una uniforme de largo h.

Ejercicio a)

Para dos distribuciones y probabilidades a priori, genere $N_1 = N_2 = 10^4$ *muestras de cada una.*

Se usaron las funciones de la biblioteca *numpy* de Python para generar muestras aleatorias.

```
# Generate random training samples
N = int(10e4)
x_samples_1 = np.random.uniform(2,10,N)
x_samples_2 = np.random.normal(2,4,N)

# Histograms to check
plt.figure('Histograma clase 1')
plt.hist(x_samples_1,bins='auto')
plt.title('Muestras clase 1: Uniforme (2,10)')
plt.ylabel('Frecuencia')
plt.show()

plt.figure('Histograma clase 2')
plt.hist(x_samples_2,bins='auto')
plt.title('Muestras clase 2: Normal (2,4)')
plt.ylabel('Frecuencia')
plt.show()
```

Figura 1. Histograma de las muestras generadas con distribución uniforme (2,10)

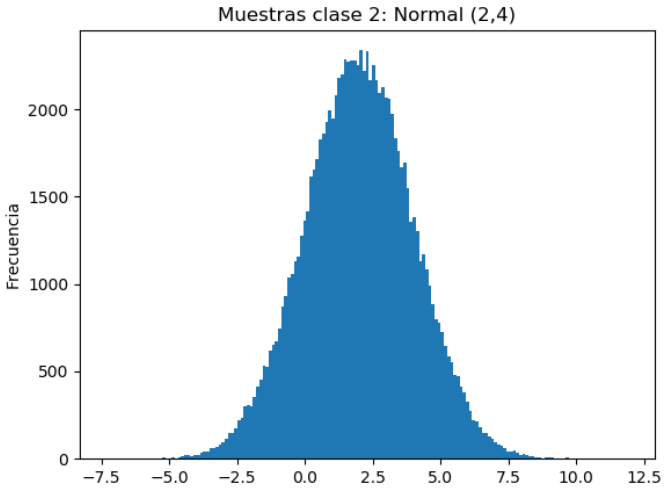

Figura 2. Histograma de las muestras generadas con distribución normal (2,4)

Ejercicio b)

Estimar las diferentes $F_X(x)$ utilizando ventana de Parzen con la ventana según se pide. La ventana debe ser una uniforme de ancho h, el cual se eligirá a continuación. Con un valor de h muy pequeño, se generarían pulsos angostos centradas en las muestras de entrenamiento, resultando en una función muy ruidosa. Mientras que con un h demasiado grande, se estaría imponiendo una distribución que cambia lentamente, pues cada muestra influiría en un rango demasiado grande.

Al elegir h se está adivinando una región en que la densidad sea aproximadamente constante, pero esto requiere conocimiento previo sobre la distribución, lo cual no es el caso en un entrenamiento no paramétrico. Se propone un método para elegir un h que es el siguiente: Generar unas muestras que se usen para validación, es decir, con el único propósito de elegir el parámetro h. Para distintos valores de h, estimar con las muestras de validación usando Parzen y luego obtener el clasificador. Finalmente, elegir el h con menor error en la clasificación.

Se realizó una función llamada `try_several_h`, la cual recibe una lista de parámetros h que se quieren probar, las muestras de validación de cada clase y la cantidad de muestras que se quiere clasificar. Esta realiza el procedimiento descripto en el párrafo anterior y arroja los errores de clasificación para cada valor de h.

```
1  # Receives a list of h and validation samples
2  # Prints out the error rate for each h
3  def try_several_h(h_list, x_validate_1, x_validate_2, N_classify):
4      x_mix,label_real = rand_gen.rand_mix(N_classify)
5
6      for h in h_list:
7          # Estimate densities with parzen
8          px_given_1 = parzen_estimate(x_validate_1, x_mix, h)
9          px_given_2 = parzen_estimate(x_validate_2, x_mix, h)
10
11         # Bayesian classification
```

```
        label_test = by.bayesian_classify(px_given_1, px_given_2)

        # Plot
        myplt.plot_est_vs_theo(x_mix,px_given_1,px_given_2,'o','h='+str(h))

        # Error
        error_rate = err.get_error(label_real,label_test)
        print("h="+str(h)+" error="+str(error_rate))
```

Se usó `N_classify=100` y se probó con distintos valores de h, obteniendo el error como el cociente entre la cantidad de muestras clasificadas incorrectamente y el total.

h=0.03 error=0.23
h=0.1 error=0.23
h=0.3 error=0.21
h=0.9 error=0.21

En cada ejecución el resultado absoluto era diferente dada la aleatoriedad, pero a grandes rasgos se mantenía la relación entre ellos. Se eligió h = 0.3 ya que en varias corridas su error fue algo menor. Puede que este valor no sea el óptimo real pero se tomó la precaución de no elegir ninguno de los extremos (muy grande o muy pequeño).

Luego, usando el h elegido se realiza la estimación para F_1 y F_2.

```
def parzen_estimate(data,X,h):
    N = len(data)
    estimate = []
    for x in X:
        sum = 0
        for x_i in data:
            sum = sum + window_function((x-x_i)/h)
        estimate.append(sum/N/h)
    return estimate

def window_function(x):
    if np.abs(x) <= 0.5:
        return 1
    else:
        return 0
```

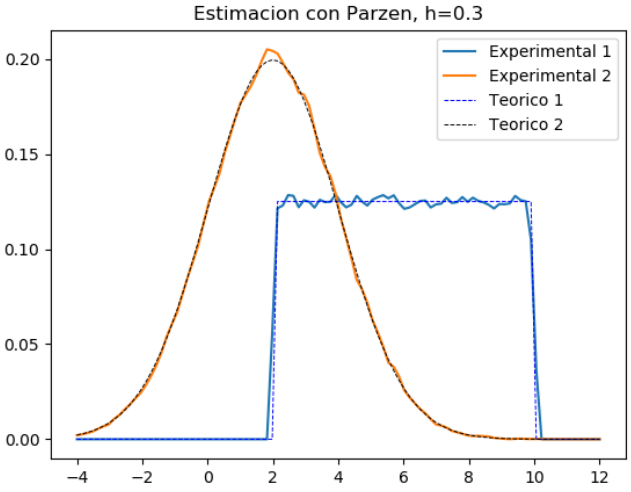

Figura 3. Densidades de probabilidad estimadas con ventana de Parzen, h=0.3.

A modo de comparación, se realizan los gráficos con h=0.01 y h=0.9.

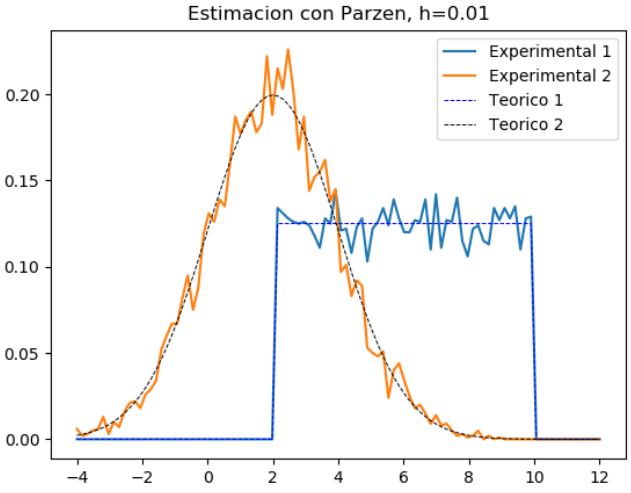

Figura 4. Densidades de probabilidad estimadas con ventana de Parzen, h=0.01.

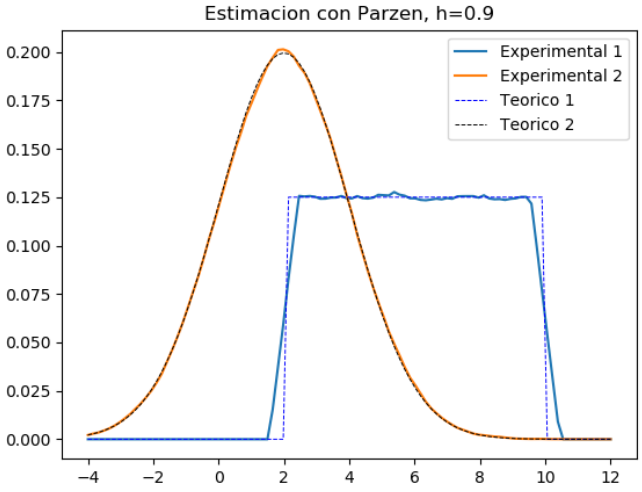

Figura 5. Densidades de probabilidad estimadas con ventana de Parzen, h=0.9.

Como era de esperar, con h=0.01 la densidad estimada es ruidosa, presentando cambios abruptos, mientras que con h=0.9 la función es sumamente suave, causando que en los bordes de la uniforme sean rampas con menor pendiente. Con h=0.3 parece ser un buen punto medio, pues es lo suficientemente suave para seguir a grandes rasgos la forma de las densidades reales, y permitiendo el cambio abrupto en los bordes de la uniforme, aunque igual se observa una pequeña pendiente a cada lado.

Ejercicio c)

Estime las diferentes F_X utilizando los K_n vecinos más cercanos para diferentes valores de k = 1, 10, 50 y 100.

Se realizó una función `knn_search` que busque los k valores más cercanos a un cierto punto dado un vector, y con esta función se realizó `knn_estimate`, la cual estima una función de densidad dada unas muestras.

```
# Returns k nearest neighbors to the value 'ref' found in 'data'.
# Return value type is a list of k tuples: (index,value)
def knn_search(k,data,ref):
    # Add index to data
    data_indexed = enumerate(data)
    # Sort by smaller distance to ref and keep the first k tuples
    return sorted(data_indexed, key=lambda x: abs(x[1]-ref))[:k]

# Returns knn estimated density evaluated in X
def knn_estimate(k,samples_data,X):
    p_estimate = []
    N = len(samples_data)

    if k == 1:
        for x in X:
            k_nearest = knn_search(k, samples_data, x)
            nearest = [x[1] for x in k_nearest] # get the value
```

```
            p_estimate.append(1/(2*abs(x-nearest[0])))
    else:
        for x in X:
            k_nearest = knn_search(k,samples_data,x)
            k_nearest = [x[1] for x in k_nearest] # get the second element (value)
                of each tuple
            max_val = max(k_nearest+[x])
            min_val = min(k_nearest+[x])
            V = abs(max_val-min_val)
            p_estimate.append(k/(N*V))

    return p_estimate
```

Como era de esperar, con k=1 la estimación es muy pobre, ya que la estimación es simplemente la ecuación 1, cuya integral diverge a infinito.

$$p_n(x) = \frac{1}{2|x - x_1|} \tag{1}$$

A medida que k crece, los picos en las funciones estimadas se hacen más pequeñas.

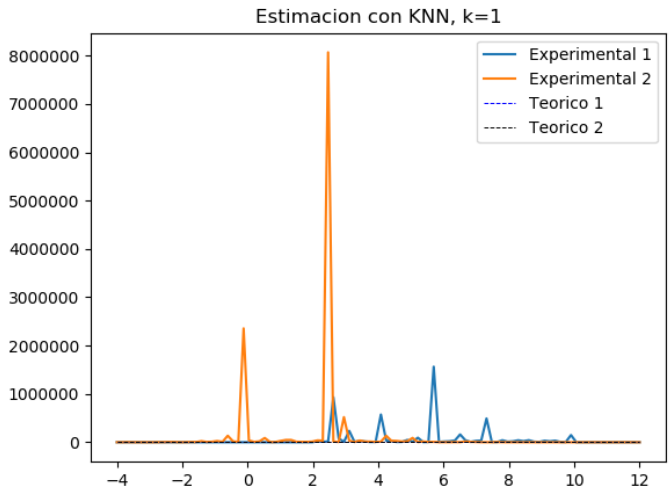

Figura 6. Densidades de probabilidad estimadas con KNN, k=1.

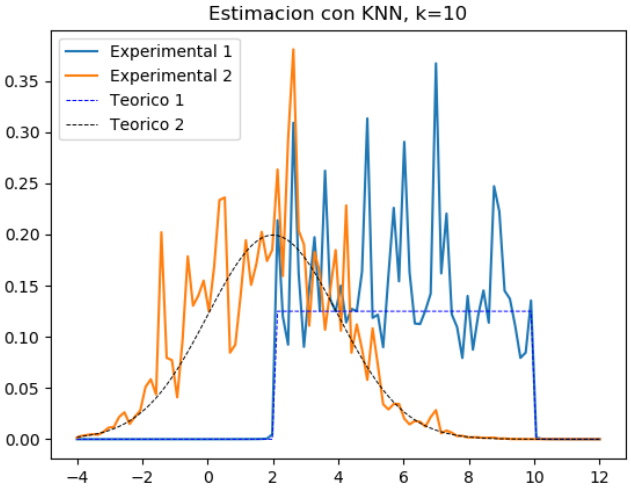

Figura 7. Densidades de probabilidad estimadas con KNN, k=1.

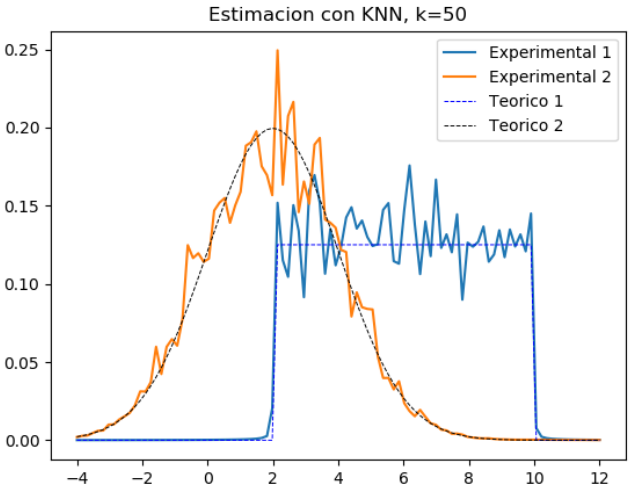

Figura 8. Densidades de probabilidad estimadas con KNN, k=50.

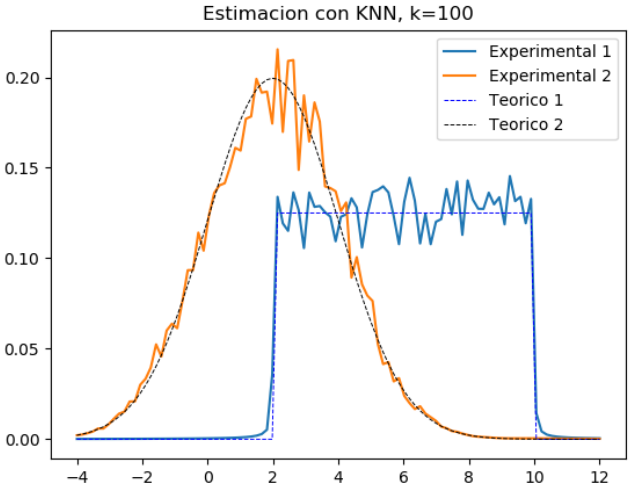

Figura 9. Densidades de probabilidad estimadas con KNN, k=100.

Ejercicio d)

Para b) y c) realice un clasificador y clasifique 10^2 *nuevas muestras. Mida el error obtenido.*

Se realizó un clasificador bayesiano que recibe dos densidades de probabilidad y devuelve la clase a la que pertenece cada punto.

```
CLASS_1 = 1
CLASS_2 = 2
# Returns a list of labels
# NOTE: Assumes a priori probabilities of each class are equal to 1/2.
def bayesian_classify(px_1,px_2):
    label_test = []
    for p1, p2 in zip(px_1, px_2):
        if p1 > p2:
            label_test.append(CLASS_1)
        else:
            label_test.append(CLASS_2)

    return label_test
```

Se generaron unas muestras de `N_test=100` puntos con el fin de poner a prueba las densidades estimadas en los puntos anteriores. La función `rand_mix` devuelve `N_test` muestras de una mezcla entre la uniforme y la gaussiana especificada en el enunciado. Con estas muestras de *test* y a partir de las muestras de entrenamiento del item a), se estimaron los *likelihood* para poder clasificar con el criterio bayesiano. Esto se realizó para Parzen (h=0.3) y para KNN (k=[1,10,50,100]).

```
# Generar una mezcla de 100 puntos
N_test = 100
x_test,label_real = rand_gen.rand_mix(N_test)

```

```
# Clasificador con parzen
h=0.3
px_given_1_parzen = parzen.parzen_estimate(x_samples_1, x_test, h)
px_given_2_parzen = parzen.parzen_estimate(x_samples_2, x_test, h)
label_test_parzen = parzen.bayesian_classify(px_given_1_parzen, px_given_2_parzen)
myplt.plot_with_labels(x_test,label_real,label_test_parzen,'Clasificacion bayesiana
    con estimacion Parzen, h='+str(h))
err_parzen = err.get_error(label_real,label_test_parzen)
print('Error with parzen is: '+str(err_parzen))

# Clasificador con knn
for k in k_list:
    px_given_1_knn = knn.knn_estimate(k,x_samples_1, x_test)
    px_given_2_knn = knn.knn_estimate(k,x_samples_2, x_test)
    label_test_knn = parzen.bayesian_classify(px_given_1_knn, px_given_2_knn)
    myplt.plot_with_labels(x_test,label_real,label_test_knn,'Clasificacion
        bayesiana con estimacion KNN, k='+str(k))
    err_knn = err.get_error(label_real,label_test_knn)
```

Los resultados obtenidos fueron:
Error with parzen is: 0.27
Error with knn is: 0.26 k=1
Error with knn is: 0.15 k=10
Error with knn is: 0.25 k=50
Error with knn is: 0.28 k=100

Otra vez, los valores pueden variar en distintas ejecuciones del programa pero se notó que con KNN (k = 10) el error era siempre menor que con otros valores de k. Esto tiene sentido ya que con un valor muy pequeño de k, la densidad estimada presenta muchos picos, causando un umbral de decisión ruidoso. Mientras que con un k muy grande, se impone una función suave. Es conveniente usar un valor balanceado de k para evitar estos dos extremos, es decir: un k lo suficientemente grande para el error de clasificación sea minimizado, pero también lo suficientemente chico como para que sólo las muestras que están cerca sean incluidas.

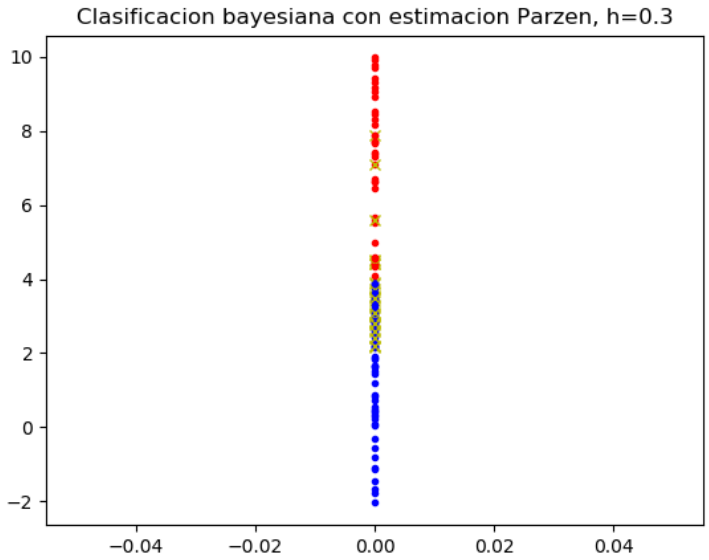

Figura 10. Puntos clasificados: rojo=clase 1, azul=clase 2, cruz=errores, h=0.3.

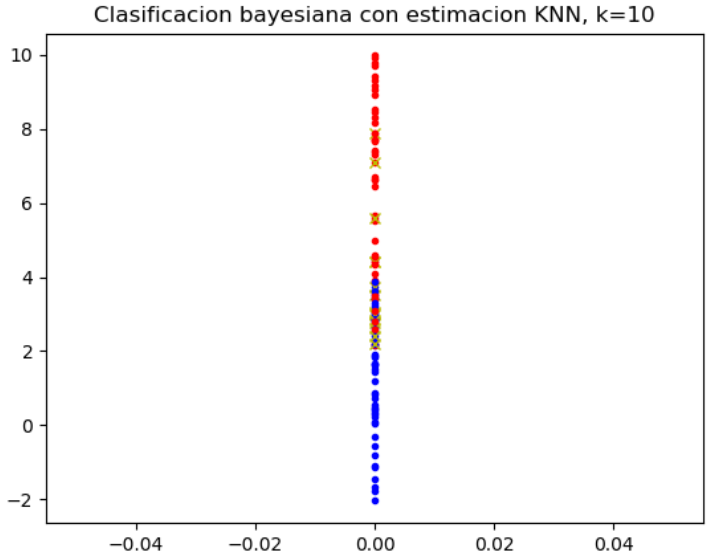

Figura 11. Puntos clasificados: rojo=clase 1, azul=clase 2, cruz=errores, k=10.

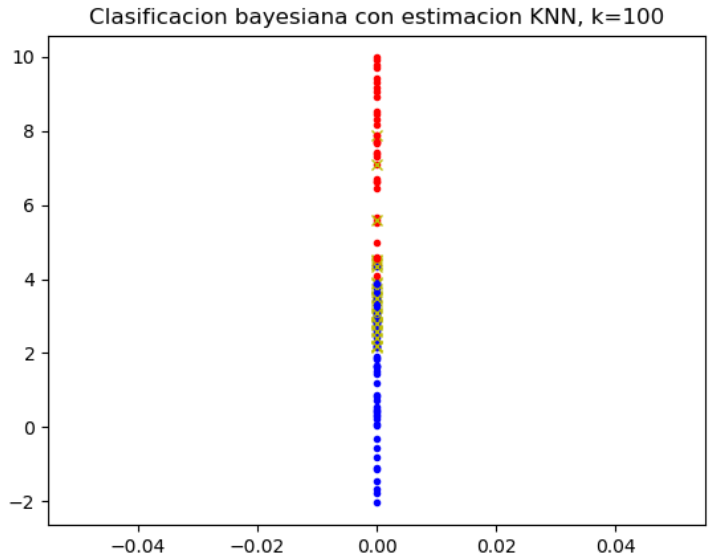

Figura 12. Puntos clasificados: rojo=clase 1, azul=clase 2, cruz=errores, k=100.

Se observa que las muestras clasificadas incorrectamente se encuentran en la región de solapamiento entre las dos funciones de probabilidad.

Ejercicio e)

Implemente la regla de clasificación del K vecino más cercano pra k=1,11 y 51 y calcule el error al clasificar las mismas muestras que en d).

Usando las mismas muestras del item d), se realizó la clasificacion con la regla de los K vecinos más cercanos.

```
# Returns classification labels
def knn_classify(k,samples_class1,samples_class2,X):
    label_real = [CLASS_1] * len(samples_class1) + [CLASS_2] * len(samples_class2)
        # real label of data
    label_test = []
    samples_class1 = samples_class1.tolist()
    samples_class2 = samples_class2.tolist()
    for x in X:
        result = knn_search(k, samples_class1 + samples_class2, x)
        result_index = [a[0] for a in result] # keeps first element of each tuple in
            result
        result_val = [a[1] for a in result] # keeps second element of each tuple in
            result
        label_knn = [label_real[n] for n in result_index] # label of the points
            obtained with knnsearch

        count1 = len([n for n in label_knn if n == CLASS_1])
        count2 = len([n for n in label_knn if n == CLASS_2])

        if count1 > count2:
```

```
            label_test.append(CLASS_1)
        else:
            label_test.append(CLASS_2)

    return label_test
```

Los resultados obtenidos fueron:
Error with knn classify is: 0.26 k=1
Error with knn classify is: 0.21 k=11
Error with knn classify is: 0.24 k=51

Comparando con la sección d), se puede verificar que los errores dados con la clasificación KNN fueron mayores o iguales que la bayesiana, pero no mayor que el doble de ella.

A modo de análisis, se calcula el error teórico:

$$P(error) = P(elegir2|\omega_1)P(\omega_1) + P(elegir2|\omega_2)P(\omega_2)$$

$$P(error) = P(x < x_o|\omega_1)P(\omega_1) + P(x > x_o|\omega_2)P(\omega_2)$$

Siendo $x_o = 3{,}94$ el umbral de decisión, el cual es la intersección entre las dos funciones de *likelihood*.

$$P(x < x_o|\omega_1) = \int_2^{x_o} 1/(10-2)dx = 0{,}2425$$

$$P(x > x_o|\omega_2) = \int_{x_o}^{\infty} \frac{1}{\sqrt{2\pi 4}} e^{-} \frac{(x-2)^2}{2 \cdot 4} dx = 0{,}166$$

$$P(error) = 0{,}2425 \cdot 0{,}5 + 0{,}166 \cdot 0{,}5 = 0{,}2043$$

Este resultado es cercano a los obtenidos experimentalmente en los items anteriores.

Conclusiones

Se logró implementar la estimación de las funciones de probabilidad mediante los métodos Parzen y KNN, en las que los resultados dependían fuertemente del parámetro h o k. Como se supone que en la realidad no se tiene conocimiento sobre el modelo de distribución de los datos, se recurrió a usar una parte de los datos para elegir un valor apropiado de h basándose en los resultados de la clasificación de muestras. Luego, al clasificar con la regla de decisión bayesiana, se obtuvo un valor de error cercano al calculado teóricamente. También se pudo comprobar que en la clasificación mediante la regla de los K vecinos más cercanos, el error no es mayor que el doble del bayesiano.