1 Portada

Universidad Autónoma de Querétaro

Maestría en Ciencias en Inteligencia Artificial

Materia - Tópicos Selectos IV - Machine Learning
Examen - K Vecinos más cercanos (KNN)

Profesor - Dr. Marco Aceves

Alumno - Juan Ignacio Ortega Gómez
Expediente - 309236

Quertearo, Queretaro a 05 de Mayo de 2022

2 Introducción

El algoritmo de las K vecinas más cercanas (K-nearest neighbors en inglés y KNN por sus siglas en inglés) es un algoritmo de Machine Learning que pertenece a los algoritmos de aprendizaje supervisado simples y fáciles de aplicar [1].

A pesar de su simplicidad, los vecinos más cercanos han tenido éxito en una gran cantidad de problemas de clasificación y regresión, incluidos dígitos escritos a mano y escenas de imágenes satelitales.

El principio detrás de los métodos de KNN es encontrar un número predefinido de muestras de entrenamiento más cercanas en distancia al nuevo punto y predecir la etiqueta a partir de ellas. El número de muestras (K) puede ser una constante definida por el usuario o variar según la densidad local de puntos [2].

En este proyecto se presenta una metodología para desarrollar modelos de K-vecinos más cercanos (KNN por sus siglas en inglés), en específico, una clase que permite generar un modelo de KNN, el cual es entrenado, tantas veces como se decida, con diferentes K impares en cada iteración con el objetivo de obtener la K que te permita obtener el modelo con la mejor métrica de rendimiento y poder utilizar este como predictor para nuevas instancias de datos.

3 Marco teórico

3.1 Definición de algoritmo de aprendizaje profundo

En el aprendizaje máquina (ML por sus siglas en inglés), la clasificación es una técnica de aprendizaje supervisado en la que la computadora se alimenta con datos etiquetados y aprende de ellos para que, en el futuro, pueda usar este aprendizaje para clasificar nuevos datos. La clasificación puede ser binaria (sólo dos clases) o multiclase (más de dos clases) [3].

3.2 Defínción de K-Vecinos más cercanos

KNN es un algoritmo de clasificación simple propuesto por primera vez por Evelyn Fix y Joseph Hodges en 1951 y desarrollado por Thomas Cover en 1967. Este algoritmo almacena todos los datos de entrada con sus etiquetas correspondientes y clasifica una nueva observación basada en la similitud. Esta clasificación se realiza en base a las etiquetas de sus vecinos más cercanos [3].

3.3 Etapas del algoritmo de KNN

La lógica detrás del algoritmo de KNN es una de las más sencillas de todos los algoritmos de ML supervisados.

Etapa 1: Seleccionar el número de K vecinas.

Etapa 2: Calcular la distancia desde un punto no clasificado a otros puntos. Para esta etapa es posible utilizar la distancia Euclidiana presentada en la Fig. 1.

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Figura 1. Fórmula de la distancia Euclidiana [1].

Etapa 3: tomar las K vecinas más cercanas según la distancia calculada.

Etapa 4: entre las K vecinas, contar el número de puntos en cada categoría. Para entender mejor las etapas hasta esta estapa se puede ver de forma gráfica en la siguiente figura.

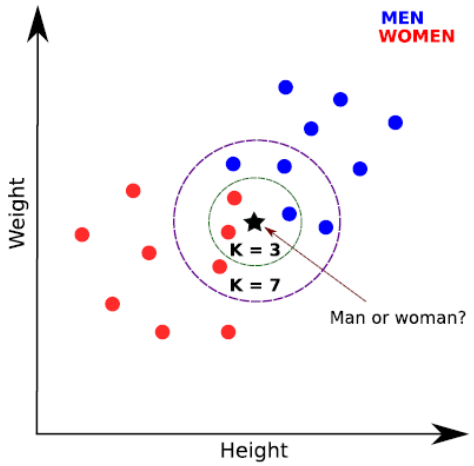


Figura 2. Clasificación basada en diferentes valores de K [3].

Etapa 5: atribuir un nuevo punto a la categoría más presente entre las K vecinas.

Etapa 6: El modelo está listo [1].

En la Figura 2, la nueva observación se clasifica por sus K vecinos más cercanos, donde K tiene dos valores diferentes: K = 3 y K = 7. Para K = 3, los vecinos más cercanos son los que están dentro del círculo verde, y para K = 7, los vecinos más cercanos están dentro del círculo morado.

Se puede observar que, si K = 3, dos de esos tres vecinos son mujeres, por lo que la nueva observación se clasificaría como mujer. Sin embargo, si K = 7, la mayoría de los vecinos más cercanos son hombres, y entonces el nuevo individuo se clasificaría como hombre.

Entonces, para la clasificación de este nuevo individuo, K = 7 es mejor que K = 3 porque lo clasificaba como hombre, que era su género real. Sin embargo, esto no significa que K = 7 sea el mejor valor general para el conjunto de datos; más observaciones nuevas deben clasificarse con diferentes valores de K para encontrar el valor con el mejor rendimiento [3].

3.4 Selección de la mejor cantidad K de vecinos más cercanos.

Es posible considerar una forma de elegir la K que hace que se mejore la clasificación. Una forma de encontrarla consiste en trazar el gráfico del valor K y la tasa de error correspondiente al conjunto de datos, como se muestra en la figura 3, para un ejemplo aleatorio donde la mejor tasa de predicción se obtiene con una K entre 5 y 18 [1].

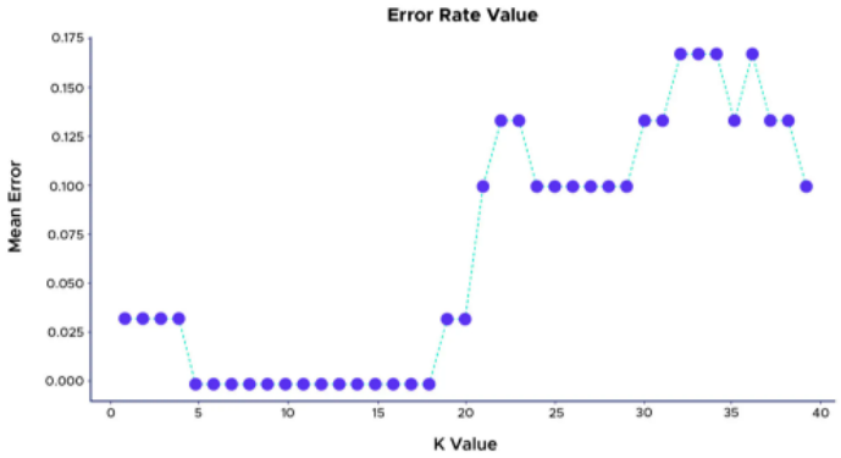


Figura 3. Valor de la tasa de error de un ejemplo aleatorio [1].

3.5 Ventajas y desventajas

1. Ventajas:

El algoritmo es simple y fácil de aplicar. No es necesario crear un modelo, configurar varios parámetros o formular hipótesis suplementarias. El algoritmo es polivalente. Puede ser utilizado para la clasificación o la regresión.

2. Desventajas:

El algoritmo se vuelve más lento a medida que el número de observaciones aumenta y las variables independientes aumentan [1].

3.6 Conceptos en el manejo de datos

3.6.1 Cuartiles

La mediana divide la muestra a la mitad, los cuartiles lo dividen, tanto como sea posible, en cuartos.

$Primercuartil = 0.25 * (n + 1)$

$Segundocuartil = 0.5 * (n + 1)$ –> Idéntico a la mediana

$Tercercuartil = 0.75 * (n + 1)$

El resultado te dice el número del valor que representa el X cuartil, de los datos ordenados de forma ascendente.

Solo si el resultado es un entero, si no, se toma el promedio de los valores de la muestra de cualquier lado de este valor, tomando la muestra de forma ordenada ascendente [5].

3.6.2 Normalización de datos

Algunos algoritmos de inteligencia artificial requieren que todos los datos se centren en un rango específico de valores, normalmente de $-1a1$ o de $0a1$. Incluso si no se requiere que los datos se encuentren dentro de los valores, es buena idea generalmente asegurarse de que los valores se encuentran dentro de un rango específico.

Normalización de valores ordinales Para normalizar un set ordinal, se tiene que preservar el orden.

Normalización de valores cuantitativos Lo primero que se tiene que hacer es observar el rango en el cual se encuentran dichos valores y el intervalo al que se quiere normalizar. No todos los valores requieren ser normalizados.

Es necesario realizar los cálculos de las siguientes variables para encontrar el valor normalizado:

1. $Mximodelosdatos = elvalormsaltodelaobservacinsinnormalizar$
2. $Mnimodelosdatos = elmenoraltodelaobservacinsinnormalizar$
3. $Mximonormalizado = elvalormsaltolimtrofealqueelmximodelosdatossernormalizado$
4. $Mnimonormalizado = Elvalormsbajolimtrofealqueelmnimodelosdatossernormalizado$
5. $Rangodespacedatos = Mximodelosdatos - Mnimodelosdatos$
6. $Rangonormalizado = Mximonormalizado - Mnimonormalizado$
7. $D = Valoranormalizar - Mnimodelosdatos$
8. $DPct = \frac{D}{Rangodedatos}$
9. $dNorm = Rangonormalizado * DPct$
10. $Normalizado = Mnimonormalizado + dNorm$

De esta forma se obtiene el valor normalizado [5].

3.7 Métricas de rendimiento

3.7.1 MSE

Es probablemente el método más comúnmente utilizado para poder predecir el error. Permite evaluar el rendimiento de múltiples modelos en un problema de predicción cuando se trata de datos continuos.

Varía en un rango de $[0, inf]$, siendo menor valor de error cuadrático medio (abreviado como MSE por sus siglas en inglés), un mejor rendimiento del modelo.

Su formulación es la siguiente:

$MSE = \frac{(p_1-a_1)^2+...+(p_n-a_n)^2}{n}$

Donde 'a' es el valor real actual y 'p' es el valor predicho [5].

3.7.2 Tasa de clasificación

La manera más simple de evaluar un modelo con características nominales y discretas es por medio de la tasa de clasificación, cuya formulación es la siguiente [5]:

$Tasadeclasificacin = 1 - \frac{Clasificacionesincorrectas}{Totaldepredicciones realizadas}$

3.7.3 Matriz de confusión binaria

Solo puede haber cuatro diferentes tipos de resultados que arrojen la matriz de confusión, mostrada en la fig. 4:

1. Verdadero positivo (TP) - Se espera que tenga in valor positivo de su característica y se obtiene un valor positivo también.
2. Verdadero negativo (TN) - Se espera un valor negativo de su característica y se predice un valor negativo también.
3. Falso positivo (FP) - Se tiene un valor negativo a pesar de haber predicho un valor positivo.
4. Falso negativo (FN) - Se tiene un valor positivo a pesar de haber predicho un valor negativo [5].

		Predicción	
		Positivos	**Negativos**
Observación	**Positivos**	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Figura 4. Matriz de confusión para cálculo del error [5].

3.7.4 Exactitud

Se puede calcular la tasa de clasificación de una manera más precisa con la matriz de confusión. Se puede calcular como una métrica de exactitud del modelo, cuya formulación es la siguiente [5]:

$Exactitud = \frac{(TP+TN)}{(TP+TN+FP+FN)}$

3.7.5 Precisión

Se puede definir como la tasa de las muestras predichas que son relevantes, se calcula como sigue [5]:

Precisión $= \frac{TP}{TP+FP}$

3.7.6 Sensitividad (Recall)

La sensitividad, también llamado recall, se puede definir como la tasa de las muestras seleccionadas que son relevantes a la prueba. Se obtiene de la siguiente manera:

$Sensitividad = \frac{TP}{TP+FN}$

3.7.7 Puntaje F1

El puntaje F1 (F1 Score en inglés), puntaje F-beta con una beta de 1, se puede definir como la media armónica entre recall y precisión, y se calcula como se muestra en la siguiente ecuación:

$PuntajeF1 = \frac{2^*TP}{2^*TP+FP+FN}$

3.8 K-Fold

K-Fold, también llamado validación cruzada, es un procedimiento que consiste en partir en k veces los datos y realizar pruebas con cada k-partición.

El parámetro K indica cuántas veces se tiene que particionar los datos. Los K más utilizados son 3, 5 y 10 particiones. La K se suele reemplazar por el número de k-particiones, '10-Fold'.

El método es muy popular porque los resultados tienen un menor sesgo y se realiza una estimación menos optimista del rendimiento y la exactitud de un algoritmo.

El algoritmo general es el siguiente: 1. Se desordenan los datos de manera aleatoria. 2. Se separan los datos en k grupos. En este caso, se recomienda crear copias de los datos, cada una con la partición que le corresponde. Por ejemplo, si se quiere hacer un '5-Fold', se hacen cinco copias con los datos. En la primera copia se quita la primera quinta parte, en la segunda copia, la segunda quinta parte y así sucesivamente hasta completar las cinco copias. 3. Las partes que se quitaron serán los bancos de pruebas. La primera prueba se realiza con el modelo de la primera copia y los datos de prueba que se quitaron del mismo y así sucesivamente. 4. Se guardan las métricas de calidad (error, tasa de clasificación, exactitud, precisión, sensitividad y puntaje F-beta). 5. Se desecha el modelo y el banco de pruebas con el que se realizó. 6. Se cambia al siguiente modelo con el siguiente banco de pruebas, se repite el procedimiento desde el paso cuatro hasta que se terminen las pruebas con todas las particiones.

Los datos se dividen lo más homogéneamente posible, es decir, que cada partición contenga aproximadamente la misma cantidad de datos, como forma gráfica de representar esto se puede mostrar el ejemplo de la siguiente figura [5]:

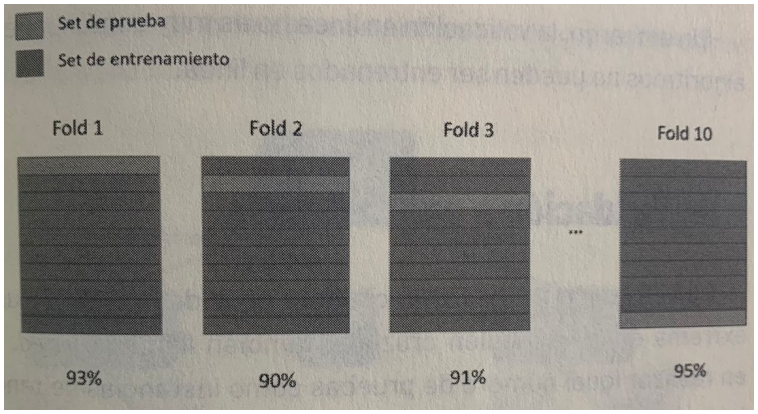


Figura 5. Ejemplo de la distribución de las pruebas para una validación cruzada '10-Fold' [5].

3.9 Base de datos

Se utilizó la base de datos (DB por sus siglas en inglés) Tumour Classification KNN obtenida de Kaggle a partir de [4], la cual proporciona el diagnóstico sobre si el paciente presenta un tumor Maligno o Benigno, además de múltiple información de cada paciente sobre sus análisis.

4 Metodología

4.1 Recolección de datos

Importamos librerías.

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import seaborn as sns
     from collections import Counter
     import random as rd
     from sklearn.model_selection import train_test_split
```

Obtenemos la base de datos.

```
[ ]: from google.colab import drive
     drive.mount('/content/drive')
```

```
[ ]: DB = pd.read_csv('/content/drive/MyDrive/ML_TSIV/KNN/TumorDataset.csv')
```

En este caso especial, eliminamos desde el ahora dos atributos de la base de datos innecesarios, al ser solo de seguimiento.

```
[ ]: DB = DB[DB.columns[1:-1]]
```

```
[ ]: DB.head()
```

```
[ ]:   diagnosis  radius_mean texture_mean  perimeter_mean   area_mean   \
     0         M        17.99         10.38          122.80      1001.0
     1         M        20.57         17.77          132.90      1326.0
     2         M        19.69         21.25          130.00      1203.0
     3         M        11.42         20.38           77.58       386.1
     4         M        20.29         14.34          135.10      1297.0

        smoothness_mean   compactness_mean     ...   smoothness_worst    \
     0          0.11840           0.27760     ...          0.1622
     1          0.08474           0.07864     ...          0.1238
     2          0.10960           0.15990     ...          0.1444
     3          0.14250           0.28390     ...          0.2098
     4          0.10030           0.13280     ...          0.1374

        concave points_worst  symmetry_worst  fractal_dimension_worst
     0                0.2654          0.4601                  0.11890
     1                0.1860          0.2750                  0.08902
     2                0.2430          0.3613                  0.08758
     3                0.2575          0.6638                  0.17300
     4                0.1625          0.2364                  0.07678
```

Desplegamos una muestra de los datos.

[ ]:
```
sns.pairplot(DB, vars = DB.columns[1:6], hue = DB.columns[0])
```

[ ]:

4.2 Preparación de los datos

[ ]:
```
Titles = list(DB.columns)

Titles_str = []
for title in Titles:
  if type(DB[title][0]) == str:
    Titles_str.append(title)

for title in Titles_str:
  types = pd.unique(DB[title])
  i = 0
  for data in types:
    data_loc = np.where(DB[title] == data)[0]
    for pos in data_loc:
      DB[title][pos] = i
    i += 1

DB.head()
```

[ ]:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	\
0	0	17.99	10.38	122.80	1001.0	
1	0	20.57	17.77	132.90	1326.0	
2	0	19.69	21.25	130.00	1203.0	
3	0	11.42	20.38	77.58	386.1	
4	0	20.29	14.34	135.10	1297.0	

	smoothness_mean	compactness_mean	concavity_mean	...	\
0	0.11840	0.27760	0.3001	...	
1	0.08474	0.07864	0.0869	...	
2	0.10960	0.15990	0.1974	...	
3	0.14250	0.28390	0.2414	...	
4	0.10030	0.13280	0.1980	...	

	area_worst	smoothness_worst	compactness_worst	concavity_worst	\
0	2019.0	0.1622	0.6656	0.7119	
1	1956.0	0.1238	0.1866	0.2416	
2	1709.0	0.1444	0.4245	0.4504	
3	567.7	0.2098	0.8663	0.6869	
4	1575.0	0.1374	0.2050	0.4000	

	concave points_worst	symmetry_worst	fractal_dimension_worst
0	0.2654	0.4601	0.11890
1	0.1860	0.2750	0.08902
2	0.2430	0.3613	0.08758
3	0.2575	0.6638	0.17300
4	0.1625	0.2364	0.07678

4.2.1 Analizamos los datos de forma elemental.

Calculamos la cantidad de AtributOs e Instancias.

[ ]:
```
NoAtributos = len(DB.T)
NoInstancias = len(DB)
```

Convertimos la base de datos en un arreglo.

[ ]:
```
DBar = DB.to_numpy()
```

Calculamos el máximo y mínimo de cada Atributo.

[ ]:
```
MaximoDeAtributos = []
MinimoDeAtributos = []
for idx in range(NoAtributos):
  CaractMax = max(DBar.T[idx])
  CaractMin = min(DBar.T[idx])
  MaximoDeAtributos.append(CaractMax)
  MinimoDeAtributos.append(CaractMin)
```

Calculamos el primer y tercer cuartil (Q1 y Q3, respectivamente), de cada atributo.

[ ]:
```
Q1 = []
Q3 = []
for idx in range(NoAtributos):
  if str(type(DBar[0][idx]))[8 : -2] != 'str':
    atrib = DBar.T[idx].tolist()
    atrib.sort()

    NoCuartil1 = 0.25 * (NoInstancias + 1)
    if str(type(NoCuartil1))[8 : -2] != 'int':
      pos1 = round(NoCuartil1)
      if pos1 < NoCuartil1:
        pos2 = pos1 + 1
      else:
        pos2 = pos1 - 1
      NoCuartil1 = round((pos1 + pos2) / 2)
    Cuartil1 = atrib[NoCuartil1 + 1]
    incremento = 1
    while True:
      if str(Cuartil1) == 'nan':
          Cuartil1 = atrib[NoCuartil1]
          NoCuartil1 -= 1
      else:
        break
```

```
    NoCuartil3 = 0.7 * (NoInstancias + 1)
    if str(type(NoCuartil3))[8 : -2] != 'int':
      pos1 = round(NoCuartil3)
      if pos1 < NoCuartil3:
        pos2 = pos1 + 1
      else:
        pos2 = pos1 - 1
      NoCuartil3 = round((pos1 + pos2) / 2)
    Cuartil3 = atrib[NoCuartil3 + 1]
    while True:
      if str(Cuartil3) == 'nan':
          Cuartil3 = atrib[NoCuartil3]
          NoCuartil3 -= 1
      else:
        break

  Q1.append(Cuartil1)
  Q3.append(Cuartil3)
Q1 = np.array(Q1)
Q3 = np.array(Q3)
```

4.2.2 Normalizamos los datos

Elegimos un rango de normalización entre 0 y 1.

[ ]:
```
MaximoNormalizado = 1
MinimoNormalizado = 0
RangoNormalizado = MaximoNormalizado - MinimoNormalizado
```

Normalizamos los valores y obtenemos el nuevo arreglo de valores normalizados 'DBNar'.

[ ]:
```
DBNorm = []
for idx in
range(NoAtributos):

  CaractNorm = []
  ifstr(type(DBar[0][idx]))[8:-2]!='str':

    RangodeDatos = MaximoDeAtributos[idx] - MinimoDeAtributos[idx]
    for idx2 in
    range(NoInstancias):

      if str(DBar[idx2][idx]) != 'nan':
        D = DBar[idx2][idx] - MinimoDeAtributos[idx]
        DPct = D / RangodeDatos
        dNorm = RangoNormalizado * DPct
        Normalizado = MinimoNormalizado + dNorm
        CaractNorm.append(Normalizado)
```

```
      else:
        CaractNorm.append(DBar[idx2][idx])

  else:
    for idx2 in range(NoInstancias):
      CaractNorm.append(DBar[idx2][idx])

  DBNorm.append(CaractNorm)

DBNar = np.array(DBNorm)
```

Visualizamos una parte de la base de datos con los valores normalizados, para garantizar una correcta transformación.

[ ]:
```
DBNarT = DBNar.T
showDB = pd.DataFrame(DBNarT)
showDB.head()
```

[ ]:

	0	1	2	3	4	5	6	7	\
0	0.0	0.521037	0.022658	0.545989	0.363733	0.593753	0.792037	0.703140	
1	0.0	0.643144	0.272574	0.615783	0.501591	0.289880	0.181768	0.203608	
2	0.0	0.601496	0.390260	0.595743	0.449417	0.514309	0.431017	0.462512	
3	0.0	0.210090	0.360839	0.233501	0.102906	0.811321	0.811361	0.565604	
4	0.0	0.629893	0.156578	0.630986	0.489290	0.430351	0.347893	0.463918	

	8	9	...	21	22	23	24	25	\
0	0.731113	0.686364	...	0.620776	0.141525	0.668310	0.450698	0.601136	
1	0.348757	0.379798	...	0.606901	0.303571	0.539818	0.435214	0.347553	
2	0.635686	0.509596	...	0.556386	0.360075	0.508442	0.374508	0.483590	
3	0.522863	0.776263	...	0.248310	0.385928	0.241347	0.094008	0.915472	
4	0.518390	0.378283	...	0.519744	0.123934	0.506948	0.341575	0.437364	

	26	27	28	29	30
0	0.619292	0.568610	0.912027	0.598462	0.418864
1	0.154563	0.192971	0.639175	0.233590	0.222878
2	0.385375	0.359744	0.835052	0.403706	0.213433
3	0.814012	0.548642	0.884880	1.000000	0.773711
4	0.172415	0.319489	0.558419	0.157500	0.142595

4.3 Análisis de los datos

4.3.1 Generación de X e Y

Reordenamos aleatoriamente la lista de la base de datos.

[ ]:
```
RDBNar = rd.sample(DBNar.T.tolist(), k=len(DBNar.T))
RDBNar = np.array(RDBNar).T
```

Seleccionamos el atributo para nuestro valor objetivo 'Y' y el resto 'X' como ejemplos para el entrenamiento.

[ ]:
```
X = DBNar[1:]
X = X.T
Y = DBNar[0]
```

4.3.2 Serie de funciones creadas para obtener las métricas de rendimiento de un modelo.

[ ]:
```
#Función para obtener el error cuadrático medio, recibe las listas de los␣
 ↪valores predichos y los valores reales.
def MSE(MSEpred, MSEreal):
  MSEsize = len(MSEreal)
  MSEt = 0
  for MSEidx in range(MSEsize):
    MSEt += (MSEpred[MSEidx] - MSEreal[MSEidx])**2
  MSEf = MSEt / MSEsize
  return(MSEf)

#Función para calcular la tasa de clasificación, recibe las listas de los␣
 ↪valores predichos y los valores reales.
def TC(TCpred, TCreal):
  TCsize = len(TCpred)
  TCt = 0
  for TCidx in range(TCsize):
    if TCpred[TCidx] != TCreal[TCidx]:
      TCt += 1
  TCf = 1 - TCt / TCsize
  return(TCf)

#Función que obtiene los valores de la matriz de confusión (TP, FN, FP, TN),␣
 ↪recibe las listas de los valores predichos y los valores reales.
#Solo funciona a partir de esta métrica para etiquetas con longitud = 2, es␣
 ↪decir binarias.
#Se decidió utilizar estas métricas en específico para nuestra base de datos ya␣
 ↪que los tipos de sus etiquetas solo son 2, 1 o 0.
#Para proyectos con más categorías en sus etiquetas, se debe
usar directamente la Exactitud, que es con la que se forma la Matriz de
confusión.
```

```
def MatConf(MCpred, MCreal):
  MCsize = len(MCpred)
  MCTP = 0
  MCFN = 0
  MCFP = 0
  MCTN = 0
  for MCidx in range(MCsize):
    if MCreal[MCidx] == 1:
      if MCpred[MCidx] == 1:
        MCTP += 1
      else:
        MCFN += 1
    else:
      if MCpred[MCidx] == 1:
        MCFP += 1
      else:
        MCTN += 1
  return(MCTP, MCFN, MCFP, MCTN)

#Funciones para calcular la tasa de clasificación (TdC), incluyendo la exactitud␣
 ↪(TdCExact), precisión (TdCPre), sensitividad o Recall (TdCSens)
#y puntaje F - beta con beta = 1, es decir, F1 (TdCF1).
def TdC(TdCTP, TdCFN, TdCFP, TdCTN):
  TdCExact = (TdCTP + TdCTN) / (TdCTP + TdCTN + TdCFP + TdCFN)
  TdCPre = TdCTP / (TdCTP + TdCFP)
  TdCSens = TdCTP / (TdCTP + TdCFN)
  TdCF1 = (2 * TdCTP) / (2 * TdCTP + TdCFP + TdCFN)
  return([TdCExact, TdCPre, TdCSens, TdCF1])

#Función para obtener las estadísticas de rendimiento, recibe las listas de los␣
 ↪valores predichos y los valores reales.
def Estadisticas(EYPred, EYreal):
  EMSE = MSE(EYPred, EYreal)
  ETC = TC(EYPred, EYreal)
  ETP, EFN, EFP, ETN = MatConf(EYPred, EYreal)
  EExact, EPrecis, ESens, EF1 = TdC(ETP, EFN, EFP, ETN)
  Estadistica = [EMSE, ETC, EExact, EPrecis, ESens, EF1]
  return(Estadistica)
```

4.3.3 Creación de una clase propia, la cual fungirá como un clasificador KNN.

[ ]:
```
class KNN():
  def __init__(self):
    self.bestK = 3

  #Función para entrenar un modelo de KNN, cuyo entrenamiento consiste en␣
↪encontrar la mejor K de acuerdo a las estadísticas de rendimiento.
  #La cantidad de épocas (fEpocas) consiste en el número de Ks distintas a␣
↪probar, siendo siempre impares.
  #La prioridad asigna a qué medida de rendimiento tomar en cuenta para␣
↪determinar el mejor K.
  def fit(self, fXTrainP, fYTrainP, fXTestP, fYTestP, fEpocasP = 4, fprioritYP =␣
↪'Precision'):
    self.fXtrain = fXTrainP
    self.fYtrain = fYTrainP
    self.fXtest = fXTestP
    self.fYtest = fYTestP
    self.fpriority = fprioritYP
    self.fEpocas = fEpocasP
    self.fKfin = 3 + 2 * (self.fEpocas - 1)
    self.bestK = self.SelectK()

  #Función para predecir un nuevo punto a partir del modelo entrenado.
  def PrednewPoint(self, KNNXnewPoint):
    self.newYpred = self.XpredP(KNNXnewPoint)
    return(self.newYpred)

  #Función que se encarga de calcular la distancia Euclidiana entre dos puntos␣
↪de n dimensionalidad.
  def Euc_Dist(self, EuX1, EuX2):
    EuInSqua = 0
    for EuIdx in range(len(EuX1)):
      EuInSqua += (EuX2[EuIdx] - EuX1[EuIdx])**2
    EucDistance = EuInSqua**0.5
    return(EucDistance)

  #Función que obtiene las distancias entre un nuevo punto y todos los puntos de␣
↪entrenamiento.
  def XDistances(self, XNew):
    XDists = []
    for Xpoint in self.fXtrain:
      XDists.append(self.Euc_Dist(XNew, Xpoint))
    return(XDists)

  #Función dedicada a calcular los K puntos más cercanos respecto a un nuevo␣
↪punto.
```

```
 def KClosest(self, Knew):
   KDists = self.XDistances(Knew)
   KsortDists = sorted(KDists)
   KClosestDist =  KsortDists[:self.fKactual]
   KCposs = []
   for KCDist in KClosestDist:
     KCp = np.where(KDists == KCDist)
     KCposs.append(KCp[0][0])
   self.KClosestPoints = []
   for KCpos in KCposs:
     self.KClosestPoints.append(self.fXtrain[KCpos])
   return(self.KClosestPoints)

 #Función que busca determinar entre los K puntos más cercanos a un nuevo␣
↪punto, el número de puntos de cada categoría.
 def NumPointsxCat(self, Nnew):
   NClosestPoints = self.KClosest(Nnew)
   Npos = []
   for Npoint in NClosestPoints:
     NW = np.where(self.fXtrain == Npoint)
     Npos.append(NW[0][0])
   CatxPoint = []
   for Nelement in Npos:
     CatxPoint.append(self.fYtrain[Nelement])
   return(CatxPoint)

 #Función para determinar el valor predicho de un nuevo punto.
 def XpredP(self, XnewP):
   XCatP = self.NumPointsxCat(XnewP)
   XCounterP = Counter(XCatP)
   first = XCounterP.most_common(1)
   XYpredP = first[0][0]
   return(XYpredP)

 #Función que se encarga de calcular los valores predichos de todos los punto␣
↪de prueba.
 def PredAllP(self):
   PAYpred = []
   for PApoint in self.fXtest:
     PAYpred.append(self.XpredP(PApoint))
   return(PAYpred)
```

```
 #Función que obtiene las estadísticas de rendimiento con diferentes Ks.
 def fstatistics(self):
   self.fKactual = 3
   fEstadisticas = []
   print(str(self.fpriority), 'con', str(self.fEpocas), 'K distintas:')
   while self.fKactual <= self.fKfin:
     fYpred = self.PredAllP()
     fEstadistica = Estadisticas(fYpred, self.fYtest)
     print(str(self.fpriority), 'con k =', str(self.fKactual) + ':',␣
↪str(fEstadistica[3]) + '.')
     fEstadisticas.append(fEstadistica)
     self.fKactual += 2
   return(fEstadisticas)

 #Función que selecciona la mejor K, de acuerdo a su rendimiento.
 def SelectK(self):
   SEstadisticas = self.fstatistics()
   SMSE = []
   STC = []
   SExact = []
   SPrecis = []
   SSens = []
   SF1 = []
   for Selement in SEstadisticas:
     SMSE.append(Selement[0])
     STC.append(Selement[1])
     SExact.append(Selement[2])
     SPrecis.append(Selement[3])
     SSens.append(Selement[4])
     SF1.append(Selement[5])

   self.MSE = min(SMSE)
   self.TC = max(STC)
   self.Exact = max(SExact)
   self.Precis = max(SPrecis)
   self.Rec = max(SSens)
   self.F1 = max(SF1)

   if self.fpriority == 'MSE':
     SW = np.where(SMSE == self.MSE)
   elif self.fpriority == 'TdC':
     SW = np.where(STC == self.TC)
   elif self.fpriority == 'Exactitud':
     SW = np.where(SExact == self.Exact)
   elif self.fpriority == 'Precision':
     SW = np.where(SPrecis == self.Precis)
   elif self.fpriority == 'Recall':
```

```
      SW = np.where(SSens == self.Rec)
    elif self.fpriority == 'F1':
      SW = np.where(SF1 == self.F1)
    else:
      SW = np.where(SPrecis == self.Precis)

    SBestPos = SW[0]
    BestK = 3 + 2 * (SBestPos - 1)
    return(BestK)

  #Función que permite obetener la precisión final del modelo, tomado con la␣
↪mejor K.
  def get_precision(self):
    return(self.Precis)

  #Función que permite obetener la estadística final seleccionada por 'GSName'␣
↪del cualqueir modelo generado.
  def get_stadistict(self, GSName):
    if GSName == 'MSE':
      GSResult = self.MSE
    elif GSName == 'TdC':
      GSResult = self.TC
    elif GSName == 'Exactitud':
      GSResult = self.Exact
    elif GSName == 'Precision':
      GSResult = self.Precis
    elif GSName == 'Recall':
      GSResult = self.Rec
    elif GSName == 'F1':
      GSResult = self.F1
    else:
      GSResult = self.Precis
    return(GSResult)
```

4.4 Entrenamos un modelo con el algoritmo de KNN generado.

4.4.1 Separamos a X e Y en conjuntos para el entrenamiento y para las pruebas.

Primeramente apartamos una sección pequeña de datos de validación para las pruebas finales (Xval, Yval) y dejamos el resto para el entrenamiento (Xres, Yres).

Nota:

Los datos de X y Y ya están re-ordenados aleatoriamente al haber sido formados por una variación de la base de datos re-ordenada aleatoriamente.

[ ]:
```
Xres, Xval, Yres, Yval = train_test_split(X, Y, test_size = 0.10)
```

4.4.2 Entrenamos el modelo

Separaremos los valores en cada iteración de acuerdo al método de validación cruzada (K-FOLD), tomando un K = 5, es decir un 5-FOLD.

[ ]:
```
DBsize = len(Xres)
DBp = DBsize / 5
VDB = int(DBp)
CDB = int(DBp * 2)
SDB = int(DBp * 3)
ODB = int(DBp * 4)

Xtest1, Xtrain1, Ytest1, Ytrain1 = train_test_split(Xres, Yres, test_size = 0.
 ↪80, shuffle = False)

Xtrain2, Ytrain2 = Xres[:VDB].tolist() + Xres[CDB:].tolist(), Yres[:VDB].
 ↪tolist() + Yres[CDB:].tolist()
Xtrain2, Ytrain2 = np.array(Xtrain2), np.array(Ytrain2)
Xtest2, Ytest2 = Xres[VDB:CDB], Yres[VDB:CDB]

Xtrain3, Ytrain3 = Xres[:CDB].tolist() + Xres[SDB:].tolist(), Yres[:CDB].
 ↪tolist() + Yres[SDB:].tolist()
Xtrain3, Ytrain3 = np.array(Xtrain3), np.array(Ytrain3)
Xtest3, Ytest3 = Xres[CDB:SDB], Yres[CDB:SDB]

Xtrain4, Ytrain4 = Xres[:SDB].tolist() + Xres[ODB:].tolist(), Yres[:SDB].
 ↪tolist() + Yres[ODB:].tolist()
Xtrain4, Ytrain4 = np.array(Xtrain4), np.array(Ytrain4)
Xtest4, Ytest4 = Xres[SDB:ODB], Yres[SDB:ODB]

Xtrain5, Xtest5, Ytrain5, Ytest5 = train_test_split(Xres, Yres, test_size = 0.
 ↪20, shuffle = False)
```

Entrenamos con KNN cada uno de los 5 grupos de datos Nota: Es posible elegir qué métrica ('MSE', 'TdC', 'Exactitud', 'Precision', 'Recall' o 'F1') tomar como referencia, en este caso se elige 'Precision'.

[ ]:
```
model1 = KNN()
model1.fit(Xtrain1, Ytrain1, Xtest1, Ytest1, fEpocasP = 5, fprioritYP =␣
 ↪'Precision')
print('')
model2 = KNN()
model2.fit(Xtrain2, Ytrain2, Xtest2, Ytest2, fEpocasP = 5, fprioritYP =␣
 ↪'Precision')
print('')
model3 = KNN()
model3.fit(Xtrain3, Ytrain3, Xtest3, Ytest3, fEpocasP = 5, fprioritYP =␣
 ↪'Precision')
print('')
model4 = KNN()
model4.fit(Xtrain4, Ytrain4, Xtest4, Ytest4, fEpocasP = 5, fprioritYP =␣
 ↪'Precision')
print('')
model5 = KNN()
model5.fit(Xtrain5, Ytrain5, Xtest5, Ytest5, fEpocasP = 5, fprioritYP =␣
 ↪'Precision')
```

```
Precision con 5 K distintas:
Precision con k = 3: 0.8428571428571429.
Precision con k = 5: 0.8378378378378378.
Precision con k = 7: 0.8271604938271605.
Precision con k = 9: 0.8192771084337349.
Precision con k = 11: 0.
 ↪8292682926829268.

Precision con 5 K distintas:
Precision con k = 3: 0.7564102564102564.
Precision con k = 5: 0.7804878048780488.
Precision con k = 7: 0.7948717948717948.
Precision con k = 9: 0.7875.
Precision con k = 11: 0.8.

Precision con 5 K distintas:
Precision con k = 3: 0.7682926829268293.
Precision con k = 5: 0.8.
Precision con k = 7: 0.8101265822784810.
Precision con k = 9: 0.84.
Precision con k = 11: 0.
 ↪7974683544303798.

Precision con 5 K distintas:
Precision con k = 3: 0.7571428571428571.
Precision con k = 5: 0.7971014492753623.
Precision con k = 7: 0.7866666666666666.
Precision con k = 9: 0.7922077922077922.
Precision con k = 11: 0.
 ↪7894736842105263.

Precision con 5 K distintas:
Precision con k = 3: 0.717948717948718.
Precision con k = 5: 0.7564102564102564.
Precision con k = 7: 0.7564102564102564.
Precision con k = 9: 0.7792207792207793.
Precision con k = 11: 0.
 ↪7792207792207793.
```

Determinamos la precisión individual de cada modelo y la precisión final del análisis K-Fold.

```
[ ]: Stadistic1 = model1.get_stadistict('Precision')
     Stadistic2 = model2.get_stadistict('Precision')
     Stadistic3 = model3.get_stadistict('Precision')
     Stadistic4 = model4.get_stadistict('Precision')
     Stadistic5 = model5.get_stadistict('Precision')

     StadisticFin = (Stadistic1 + Stadistic2 + Stadistic3 + Stadistic4 + Stadistic5) /
      ↪ 5
```

Desplegamos las precisiones determinadas.

```
[ ]: print('Precisión 1 =', str(Stadistic1) + '.')
     print('Precisión 2 =', str(Stadistic2) + '.')
     print('Precisión 3 =', str(Stadistic3) + '.')
     print('Precisión 4 =', str(Stadistic4) + '.')
     print('Precisión 5 =', str(Stadistic5) + '.')
     print('Precisión final =', str(StadisticFin) + '.')
```

```
Precisión 1 = 0.8428571428571429.
Precisión 2 = 0.8.
Precisión 3 = 0.84.
Precisión 4 = 0.7971014492753623.
Precisión 5 = 0.7792207792207793.
Precisión final = 0.8118358742706568.
```

4.5 Evaluamos el modelo

Se evaluará con los datos de validación anteriormente seleccionados de los datos globales, los cuales no se utilizaron durante el entrenamiento.

4.5.1 Obtenemos el vector de predicciones de cada modelo.

Nota: Se elige el modelo con la mayor precisión, de acuerdo a las obtenidas en el paso anterior, en este caso del modelo 1.

```
[ ]: YvalPred = []
     for element in Xval:
       YvalPred.append(model1.PrednewPoint(element))
```

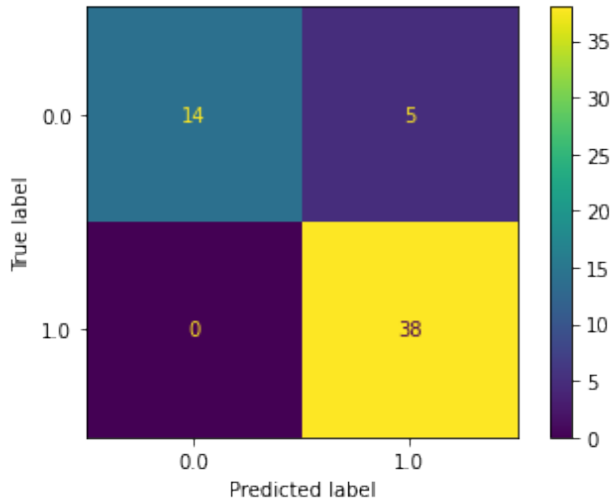
4.5.2 Analizamos métricas de evaluación

[ ]:
```
from tabulate import tabulate

ModelStatistics = Estadisticas(YvalPred, Yval)

STable = [ ['MSE', str(ModelStatistics[0])],
     ['Tasa de clasificación', str(ModelStatistics[1])],
     ['Exactitud', str(ModelStatistics[2])],
     ['Precisión', str(ModelStatistics[3])],
     ['Recall', str(ModelStatistics[4])],
     ['F1', str(ModelStatistics[5])] ]

print(tabulate(STable, headers = ['Métrica', 'Valor'], tablefmt="presto"))
```

```
 Métrica               |     Valor
-----------------------+-----------
 MSE                   | 0.0877193
 Tasa de clasificación | 0.912281
 Exactitud             | 0.912281
 Precisión             | 0.883721
 Recall                | 1
 F1                    | 0.938272
```

Obtenemos la matriz de confusión de los datos de validación.

[ ]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(Yval, YvalPred, labels = np.unique(Yval))
disp = ConfusionMatrixDisplay(confusion_matrix = cm, display_labels = np.
 ↪unique(Yval))
disp.plot()
plt.show()
```



5 Concluciones

1. La creación de una clase que permite generar modelos de KNN, facilitó el entendimiento del funcionamiento de estos a profundidad. Además, dicha creación generó que sea más sencillo pensar en su utilización, una vez analizada una nueva base de datos, según los objetivos deseados.
2. La métrica de rendimiento que utilices, no influye en las decisiones finales sobre cual K otorga mejores resultados.
3. Enfocando el análisis sobre la base de datos que estamos utilizando, el modelo desarrollado es el deseado a pesar de contar con falsos positivos debido a que resulta con cero falsos negativos, decuardo a la métrica de la matriz de confusión de los datos de validación. Lo anterior, es preferible debido a que es mejor preveer resultados negativos para desmentir posteriormente con mayores estudios, que permitir un resultado donde de verdad exista un tumor maligno y no se hagan mayores estudios al seguir la predicción de lo contrario.

6 Referencias

[1] A. Mike, ¿Qué es el algoritmo KNN?, Formación en ciencia de datos | DataScientest.com, 28-dic-2021.

[2] 1.6. Nearest Neighbors, scikit-learn. [En línea]. Disponible en: https://scikit-learn/stable/modules/neighbors.html. [Accedido: 04-may-2022].

[3] D. Lopez-Bernal, D. Balderas, P. Ponce, y A. Molina, Education 4.0: Teaching the basics of KNN, LDA and Simple Perceptron algorithms for binary classification problems, Future internet, vol. 13, n.o 8, p. 193, 2021.

[4] Shubhankitsirvaiya, Tumour Classification KNN, Kaggle.com, 16-feb-2021. [En línea]. Disponible en: https://kaggle.com/shubhankitsirvaiya06/tumour-classification-knn. [Accedido: 04-may-2022].

[5] M. A. Aceves Fernández, Inteligencia Artificial para programadores con prisa. UNIVERSO de LETRAS, 2021.