Adrián Donate Aguilar, Javier Carrizosa Bermejo, Alba Calvo Herrero, Nicolae Alexandru Molnar

PRÁCTICA 2 INTELIGENCIA ARTIFICIAL

1. OBJETIVO

Esta práctica consiste en aplicar los conceptos vistos relativos a la construcción de modelos no paramétricos, en particular k-vecinos próximos, árboles de clasificación y Naïve Bayes en un problema de predicción en un contexto médico de forma que el alumno desarrolle su capacidad crítica a la hora de emplear estos modelos.

2. PLANTEAMIENTO

Se trata de un problema de detección de la benignidad o malignidad de un tumor a partir de imágenes obtenidas por punción de aguja fina. Los datos y su descripción pueden descargarse aquí: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

3. TAREAS A REALIZAR Y PREGUNTAS

1. Describir brevemente en qué consiste el problema ampliando la información dada más arriba (1 punto, 10 líneas de texto máximo)

Queremos predecir la malignidad o benignidad de un tumor de mamas en función de los parámetros contenidos en la base de datos "breast_cancer" de scikit-learn. Para llevar a cabo esta predicción utilizaremos tres modelos: **K-vecinos más próximos**, **Árboles de clasificación** y **Naïve Bayes**. Modelos que compararemos en busca del que ofrezca una mejor precisión.

Para cada modelo, se realizarán las siguientes tareas: Obtener, si lo tuviera, el valor óptimo de su hiper-parámetro respectivo; crear el modelo con el hiper-parámetro obtenido y entrenarlo con el conjunto de entrenamiento; predecir la clasificación de los datos de test y calcular la precisión del modelo.

Una vez obtenida la precisión de cada modelo, los compararemos según el ratio de falsos positivos (FP) obtenido en el conjunto de test. El objetivo de este criterio es minimizar el número de tumores malignos clasificados como benignos, ya que consideramos este caso como el más peligroso.

2. Dividir la muestra según un criterio razonable (explicar por qué esa división) (1 puntos, 10 líneas máximo)

```
[ ]  # Split data into train and test
     from sklearn.model_selection import train_test_split
     X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, random_state=0)

     print("Size of training set:",len(X_train))
     print("Size of testing set:",len(X_test))
```

```
Size of training set: 455
Size of testing set: 114
```

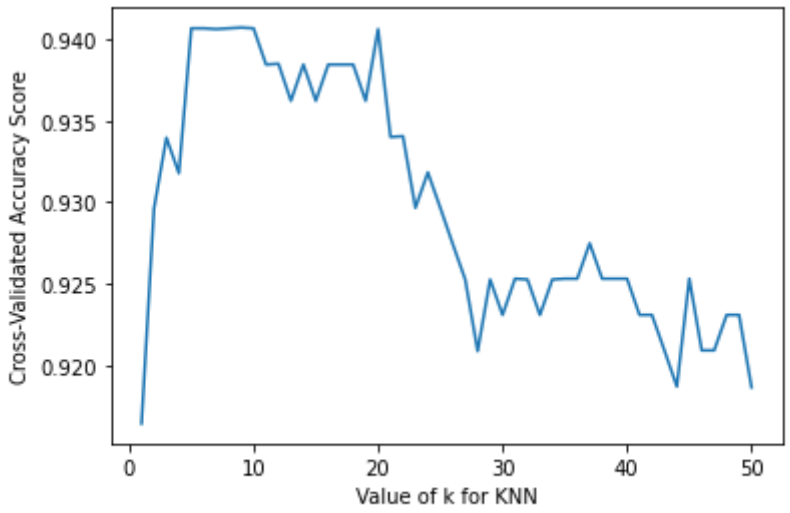
Se están dividiendo los datos en datos de **test** y de **entrenamiento**. Un 80% de los datos se guardan como datos de entrenamiento y el 20% como datos de test. La división por defecto de la función "train_test_split" es de 75% entrenamiento y 25% test.

La relación Training/Test set depende de la disponibilidad de los datos. Por tanto, como no tenemos una disponibilidad inmensa de datos no podemos adoptar divisiones populares como 90% entrenamiento y

10% test. Sin embargo, la disponibilidad tampoco es escasa, siendo innecesario priorizar el conjunto de test. Por lo tanto, se opta por una división de 80% entrenamiento y 20% test.

3. Emplear un modelo de k-nn seleccionando los hiper-parámetros óptimos mediante algún procedimiento (explicar por que ese procedimiento) (1 punto, 10 líneas máximo).

```
[ ]  # Select the best model using cross-validation
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.model_selection import cross_val_score

     def best_knn(X,Y, k_max=10, detailed = False):
         k_scores = []
         k_range = range(1, k_max+1)
         for k in k_range:
             knn = KNeighborsClassifier(n_neighbors=k)
             scores = cross_val_score(knn, X, Y, cv=10, scoring='accuracy')
             if detailed: print(f"Accuracy score for k={k}: {scores.mean()}")
             k_scores.append(scores.mean())

         # Plot k_scores with k_range
         plt.plot(k_range, k_scores)
         plt.xlabel('Value of k for KNN')
         plt.ylabel('Cross-Validated Accuracy Score')
         plt.show()

         return k_scores.index(max(k_scores)) + 1, max(k_scores)
     n_neighbors, best_score = best_knn(X_train, Y_train, 50)
     print(f"The best model has k={n_neighbors} with a score of {round(best_score,3)}")
```




```
The best model has k=9 with a score of 0.941
```

En el algoritmo de vecinos próximos, el **único hiper-parámetro** es **K**. Conocemos dos alternativas para encontrar un valor de K óptimo: la regla heurística $\sqrt{n}$ y la búsqueda por validación cruzada.

En este apartado se han probado varios valores diferentes para K (de K=1 hasta K=50). Para cada K se ha empleado **validación cruzada** sobre los datos y se ha puntuado cada modelo generado guardándonos la media de las puntuaciones de cada iteración. La k que genere la mayor puntuación es la K óptima. En este caso es k=8. En la evaluación por validación cruzada podemos especificar un criterio de evaluación concreto, entre ellos hemos decidido utilizar '*accuracy*' porque lo consideramos el más apropiado para el modelo.

Representando la puntuación del modelo con cada K, podemos observar como a partir de K=8 la puntuación se va degradando. Por tanto, no tiene sentido continuar la evaluación para valores de K mayores a 50.

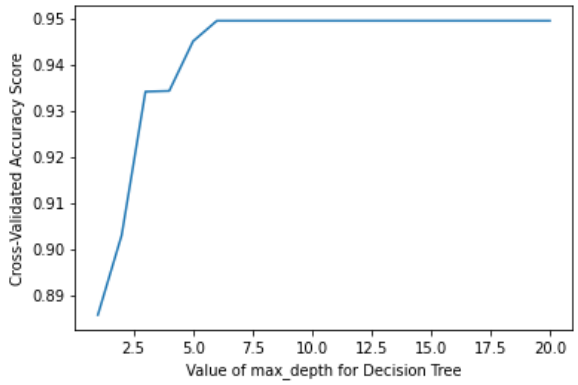
```
[ ]  # Create the optimum model
     knn = KNeighborsClassifier(n_neighbors=n_neighbors)

     # Train the model
     knn.fit(X_train, Y_train)

     # Test the model
     predictions = knn.predict(X_test)

     # Evaluate the model
     from sklearn.metrics import confusion_matrix,precision_score

     print("Confusion matrix")
     cm_knn = confusion_matrix(Y_test, predictions)
     print(cm_knn)
     print(f"Accuracy: {round(knn.score(X_test, Y_test),3)}")
     knn_precision = round(precision_score(Y_test,predictions),3)
     print(f"Precision: {knn_precision}")
```

```
Confusion matrix
[[36  6]
 [ 5 67]]
Accuracy: 0.904
Precision: 0.918
```

Una vez creado el modelo k-nn óptimo, se realiza la predicción de los datos de test y se calculan la precisión y exactitud del modelo en función de la matriz de confusión generada.

4. Hacer lo mismo para un modelo de árboles (1 punto, 10 líneas máximo).

```
[ ]  # Select the best tree-classifier using cross-validation
     from sklearn.tree import DecisionTreeClassifier

     def best_tree(X, Y, depth_max=10, detailed=False):
         tree_scores = []
         depth_range = range(1, depth_max+1)
         for max_depth in depth_range:
             # Random_state = 0 to ensure reproducibility
             tree = DecisionTreeClassifier(max_depth=max_depth, criterion='entropy', random_state=0)
             scores = cross_val_score(tree, X, Y, cv=10, scoring='accuracy')
             if detailed: print(f"Accuracy score for max_depth={max_depth}: {scores.mean()}")
             tree_scores.append(scores.mean())

         # Plot f1 with depth_range
         plt.plot(depth_range, tree_scores)
         plt.xlabel('Value of max_depth for Decision Tree')
         plt.ylabel('Cross-Validated Accuracy Score')
         plt.show()

         return tree_scores.index(max(tree_scores)) + 1, max(tree_scores)

     best_depth, max_score = best_tree(X_train, Y_train, depth_max=20)
     print(f"The best model has max_depth={best_depth} with a score of {round(max_score,3)}")
```




```
The best model has max_depth=6 with a score of 0.949
```

Para obtener el árbol de decisión óptimo se ha seguido un método similar al apartado anterior. En este caso se han construido diferentes modelos para distintas profundidades (desde profundidad = 1 hasta profundidad = 20). Para cada profundidad se ha realizado validación cruzada sobre los datos guardándonos la puntuación media de cada iteración siguiendo el criterio *'accuracy'* en consonancia con el caso de k-nn.

Tras comparar todas las puntuaciones obtenidas obtenemos que la profundidad óptima es 4. Hemos utilizado este método ya que la función "DecisionTreeClassifier", si no se especifica una profundidad, genera nodos hasta que todas las hojas sean puras (0 fallos de clasificación); creemos que el modelo generado bajo este criterio no es el más óptimo y necesitaría de poda en el futuro.

```
[ ]  # Create the tree with the best depth
     tree=DecisionTreeClassifier(max_depth=best_depth)

     # Train the model
     tree.fit(X_train, Y_train)

     # Test the model
     predictions = tree.predict(X_test)

     # Evaluate the model
     cm_tree = confusion_matrix(Y_test, predictions)
     print(cm_tree)
     print(f"Accuracy: {round(tree.score(X_test, Y_test),3)}")
     tree_precision = round(precision_score(Y_test,predictions),3)
     print(f"Precision: {tree_precision}")
```

```
[[39  3]
 [ 5 67]]
Accuracy: 0.93
Precision: 0.957
```

Una vez creado el módelo árbol de decisión óptimo, se realiza la predicción de los datos de test y se calculan la precisión y exactitud del modelo en función de la matriz de confusión generada.

5. Hacer lo mismo para un modelo de naïve Bayes (1 punto, 10 líneas máximo).

```
[ ]  # Select the best bayes-classifier using cross-validation
     from sklearn.naive_bayes import GaussianNB

     # Create the bayes with the best var
     bayes=GaussianNB()

     # Train the model
     bayes.fit(X_train, Y_train) # Si no existe prior definido, lo genera en función de Y_train

     # Test the model
     predictions = bayes.predict(X_test)

     # Evaluate the model
     cm_bayes = confusion_matrix(Y_test, predictions)
     print(cm_bayes)
     print(f"Accuracy: {round(bayes.score(X_test, Y_test),3)}")
     bayes_precision = round(precision_score(Y_test,predictions),3)
     print(f"Precision: {bayes_precision}")
```

```
[[38  4]
 [ 5 67]]
Accuracy: 0.921
Precision: 0.944
```

A diferencia de los modelos anteriores **no existe un hiper-parámetro a optimizar**. Lo único que puede variar la precisión del modelo es la distribución de probabilidades prior. Por tanto, no es necesario aplicar ningún proceso de optimización.

Para el modelo de Bayes hemos utilizado directamente la función "GaussianNB". En un principio utilizamos "CategoricalNB" pero tras varias pruebas llegamos a la conclusión de que la función actual genera un modelo mejor y es más apropiada para el problema en cuestión. Como no pasamos como parámetro ninguna prior al modelo, la prior se genera de forma automática al recibir los datos de entrenamiento en la función "fit".

Por último, utilizamos el modelo entrenado para realizar las predicciones y calcular la precisión y exactitud del modelo en función de la matriz de confusión generada.

6. Comparar, con algún criterio (explicando por qué) los resultados obtenidos con los tres modelos (2 puntos, 20 líneas máximo)

```
[ ]  # Compare the models by their precision
     print(f"Precision of KNN: {knn_precision}")
     print(f"Precision of Decision Tree: {tree_precision}")
     print(f"Precision of Naïve Bayes: {bayes_precision}\n")

     # Check false positives ratio in each model
     knn_fp = cm_knn[0][1] / (cm_knn[0][1] + cm_knn[1][1]) # fp/(fp+tp)
     print(f"False positives ratio for KNN: {round(knn_fp,3)}")

     tree_fp = cm_tree[0][1] / (cm_tree[0][1] + cm_tree[1][1])
     print(f"False positives ratio for Decision Tree: {round(tree_fp,3)}")

     bayes_fp = cm_bayes[0][1] / (cm_bayes[0][1] + cm_bayes[1][1])
     print(f"False positives ratio for Naive Bayes: {round(bayes_fp,3)}")
```

```
Precision of KNN: 0.918
Precision of Decision Tree: 0.957
Precision of Naïve Bayes: 0.944

False positives ratio for KNN: 0.082
False positives ratio for Decision Tree: 0.043
False positives ratio for Naive Bayes: 0.056
```

Entre los criterios disponibles para evaluar los modelos, se ha utilizado la matriz de confusion. En esta matriz se puede observar que el modelo de k-nn obtiene una precisión de 0.918, mientras que el modelo de árboles obtiene una puntuación de 0.943 y el modelo de naïve Bayes obtiene una puntuación de 0.944. En este caso, el modelo de Naïve Bayes es el que obtiene la mayor puntuación.

Naive bayes se considera el mejor clasificador siempre y cuando las características sean condicionalmente independientes. En este caso, un punto a favor del modelo de Naïve Bayes ha sido el reducido coste de creación del modelo puesto que, a diferencia de los otros modelos, no hemos necesitado optimizar ningún hiper-parámetro.

Otra manera de evualuar los modelos es según su capacidad para evitar falsos positivos, es decir, calificar como benignos tumores malignos. Consideramos este el peor escenario posible ya que una persona enferma se califica como sana. Creemos que en este caso es menos costoso clasificar a una persona como enferma y que sea mentira que el caso contrario. Este criterio recibe el nombre de error tipo I en estadística y cada modelo tiene una puntuación de:

- **KNN** (Error tipo I): 0.082
- **DECISION TREE** (Error tipo I): 0.057
- **NAÏVE BAYES** (Error tipo I): 0.056

En función de los valores obtenidos podemos observar que el modelo de Naïve Bayes obtiene la menor tasa de falsos positivos. Es decir, este modelo es el que menos probabilidad tiene de clasificar un tumor maligno como benigno.

El modelo de **Naïve Bayes** ha demostrado ser el **mejor modelo** obtenido por ambos criterios de evaluación. Sin embargo, cabe destacar que la diferencia con el árbol de decisión es mínima en ambos criterios, lo que coloca a Naïve Bayes en primer lugar es su ausencia de optimización.

7. Plantear posibles críticas o mejoras de cada uno de los modelos o del procedimiento seguido, tratando de evitar la repetición de lo mencionado en clase (1 punto, 10 líneas máximo)

Llegados a este punto hemos presentado una solución al problema planteado. Sin embargo, la implementación de los modelos pedidos deja margen a algunas mejoras y críticas.

En general podemos seleccionar, del conjunto de datos, las características (columnas) más relevantes eliminando las menos relevantes o aplicando cierta ponderación. Se puede realizar una comprobación sobre la independencia de las características y quedarnos con aquellas independientes entre sí (Necesario para Naïve Bayes). Por último, podemos analizar los datos en busca de diferencias notables de escala, que pueden ser problemáticas en k-nn.

En el caso de árboles de decisión, utilizamos el criterio de partición de 'entropía', pero podemos analizar el rendimiento del modelo usando el criterio 'gini' y quedarnos con el mejor. Además, el modelo base le da la misma importancia a todas las características, sin embargo, algunas pueden ser más relevantes que otras y podemos ponderar las particiones en función de dicha importancia. En el caso de k-nn esta ponderación también se debe tener en cuenta.

En el caso de Naïve Bayes, podemos calcular una distribución prior basada en los datos o alguna distribución que nos permita ajustar el modelo mejor. Es imperativo comprobar la independencia condicional de las características para el correcto funcionamiento del modelo.

8. Replicar todos estos resultados, empleando obligatoriamente un código similar al utilizado en los puntos anteriores, sobre otro problema médico de características similares (clasificación binaria). Pueden consultarse bases de datos libres como las contenidas en el repositorio kaggle.com. (2 puntos, 40 líneas máximo)

Para este ejercicio hemos decidido utilizar la base de datos Cardiovascular Disease Database que contiene los datos que usaremos para predecir si una persona de x características padece una enfermedad cardiovascular.

Primero, cargamos los datos para su tratamiento. Debemos tener en cuenta que "famhist" puede tener los valores "Absent" o "Present" que son de tipo String, por lo que los cambiamos a 0 ó 1 respectivamente para que el modelo pueda ser entrenado.

```
[ ]  # Read csv with filename "cardiovascular.txt"
     df = pd.read_csv("cardiovascular.csv", sep=";")

     # Check if '?' is present in the dataset
     if True in df.isnull().any():
         print("There are missing values in the dataset")
     else:
         print("There are no missing values in the dataset")

     # Enumerate the values of 'famhist' to 0 if "Present" and 1 if "Absent"
     for pos in range(len(df['famhist'])):
         df['famhist'][pos] = '1' if df['famhist'][pos] == 'Present' else '0'

     print(df.head())

     X = df.iloc[:, :-1]
     Y = df.iloc[:, -1]
```

```
There are no missing values in the dataset
   ind  sbp  tobacco   ldl  adiposity famhist  typea  obesity  alcohol  age  \
0    1  160    12.00  5.73      23.11       1     49    25.30    97.20   52
1    2  144     0.01  4.41      28.61       0     55    28.87     2.06   63
2    3  118     0.08  3.48      32.28       1     52    29.14     3.81   46
3    4  170     7.50  6.41      38.03       1     51    31.99    24.26   58
4    5  134    13.60  3.50      27.78       1     60    25.99    57.34   49

   chd
0    1
1    1
2    0
3    1
4    1
```

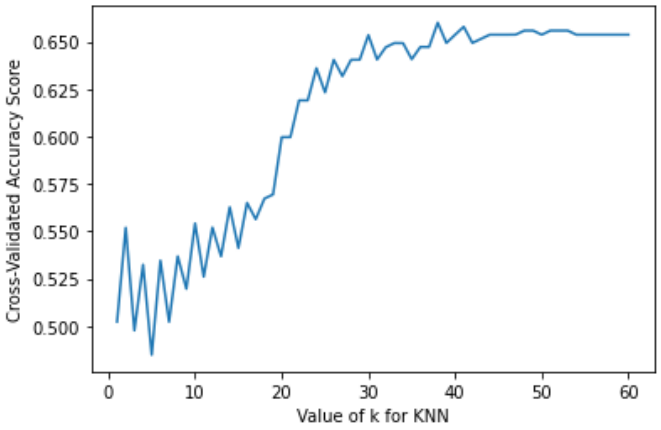
A continuación, dividimos los datos en datos de test y de entrenamiento. Al igual que en la base de datos anterior, el 80% de datos se guardan como datos de entrenamiento y 20% como datos de test, ya que la disponibilidad de los datos de esta base de datos es muy parecida a la de "breast_cancer".

```
[ ]  # Split data into train and test
     from sklearn.model_selection import train_test_split
     X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=1, stratify=Y)

     print("Size of training set:", len(X_train), "samples")
     print("Size of testing set:", len(X_test), "samples")
```

```
Size of training set: 369 samples
Size of testing set: 93 samples
```

Empleamos un modelo k-nn. En este apartado se ha usado la función ***best_knn()*** utilizada anteriormente. La k que genera la mayor puntuación es la k óptima. En este caso es k=21. Una vez encontrado el hiper-parámetro óptimo, calculamos la precisión del modelo y la almacenamos para compararla al finalizar.

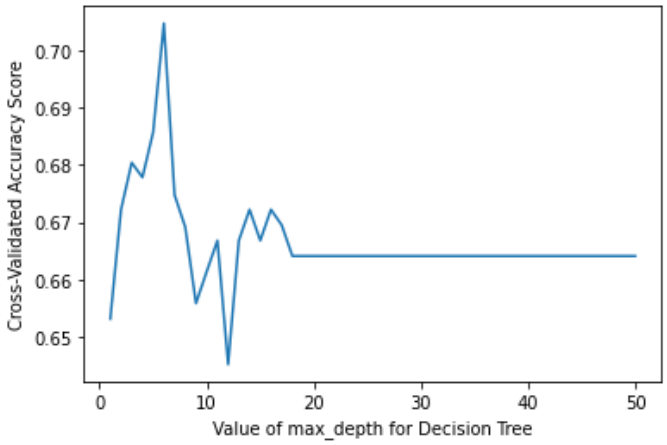
```
[ ]  ############## KNN Classifier ##############
     # Select the best k using cross-validation
     n_neighbors, knn_score = best_knn(X, Y, 60)
     print(f"The best model has k={n_neighbors} with a score of {round(knn_score,3)}")

     # Create the optimum model
     knn = KNeighborsClassifier(n_neighbors=n_neighbors)

     # Train the model
     knn.fit(X_train,Y_train)

     # Test the model
     predictions = knn.predict(X_test)

     #Evaluate the model
     print ("Confusion matrix")
     cm_knn= confusion_matrix(Y_test, predictions)

     print(cm_knn)
     print(f"Accuracy: {round(knn.score(X_test, Y_test),3)}")
     knn_precision = round(precision_score(Y_test,predictions),3)
     print(f"Precision: {knn_precision}")
```




```
The best model has k=38 with a score of 0.66
Confusion matrix
[[58  3]
 [30  2]]
Accuracy: 0.645
Precision: 0.4
```

Empleamos un modelo de árbol de decisión. En este apartado se ha usado la función best_tree() utilizada anteriormente. Tras comparar todas las puntuaciones obtenidas, concluimos que la profundidad óptima es 3. De nuevo calculamos la precisión y la guardamos para compararla al finalizar.

```
[ ]  ############## Decision Tree Classifier ##############

     # Select the best depth using cross-validation
     best_depth, tree_score = best_tree(X_train, Y_train, 50)
     print(f"The best model has max_depth={best_depth} with a score of {round(tree_score,3)}")

     # Create the optimum model
     tree=DecisionTreeClassifier(max_depth=best_depth)

     # Train the model
     tree.fit(X_train, Y_train)

     # Test the model
     predictions = tree.predict(X_test)

     # Evaluate the model
     cm_tree = confusion_matrix(Y_test, predictions)
     print(cm_tree)
     print(f"Accuracy: {round(tree.score(X_test, Y_test),3)}")
     tree_precision = round(precision_score(Y_test,predictions),3)
     print(f"Precision: {tree_precision}")
```




```
The best model has max_depth=6 with a score of 0.705
[[54  7]
 [25  7]]
Accuracy: 0.656
Precision: 0.5
```

Seguidamente empleamos un modelo **Naïve Bayes**, utilizamos la función "GaussianNB" al igual que en la base de datos anterior.

Por último, calculamos la precisión y exactitud del modelo en función de la matriz de confusión generada para comparar el modelo.

```
[ ]  ############## Naïve Bayes Classifier ##############

     #Generate a prior for the model using Y
     counts = [0,0]

     values = Y_train.values
     for i in range(len(values)):
         counts[values[i]] += 1

     # Create the bayes with the calculated prior
     bayes=GaussianNB(priors=[counts[0]/len(Y_train), counts[1]/len(Y_train)])

     # Train the model
     bayes.fit(X_train, Y_train) # If the prior does not exist, it is generated in the fit function

     # Test the model
     predictions = bayes.predict(X_test)

     # Evaluate the model
     cm_bayes = confusion_matrix(Y_test, predictions)
     print(cm_bayes)
     print(f"Accuracy: {round(bayes.score(X_test, Y_test),3)}")
     bayes_precision = round(precision_score(Y_test,predictions),3)
     print(f"Precision: {bayes_precision}")
```

```
[[48 13]
 [10 22]]
Accuracy: 0.753
Precision: 0.629
```

```
[ ]  # Compare the models by their precision
     print(f"Precision of KNN: {knn_precision}")
     print(f"Precision of Decision Tree: {tree_precision}")
     print(f"Precision of Naïve Bayes: {bayes_precision}\n")

     # Check false positives ratio in each model
     knn_fp = cm_knn[0][1] / (cm_knn[0][1] + cm_knn[1][1]) # fp/(fp+tp)
     print(f"False positives ratio for KNN: {round(knn_fp,3)}")

     tree_fp = cm_tree[0][1] / (cm_tree[0][1] + cm_tree[1][1])
     print(f"False positives ratio for Decision Tree: {round(tree_fp,3)}")

     bayes_fp = cm_bayes[0][1] / (cm_bayes[0][1] + cm_bayes[1][1])
     print(f"False positives ratio for Naive Bayes: {round(bayes_fp,3)}")
```

```
Precision of KNN: 0.4
Precision of Decision Tree: 0.5
Precision of Naïve Bayes: 0.629

False positives ratio for KNN: 0.6
False positives ratio for Decision Tree: 0.5
False positives ratio for Naive Bayes: 0.371
```

Para finalizar, comparamos los modelos según las precisiones calculadas. Observamos que el modelo Naïve Bayes es el modelo con mayor precisión (**0.629**) y con un menor porcentaje de falsos positivos (**0.371**).

Podemos concluir que la precisión de los tres modelos usados es baja. Esto se debe a que las características son poco representativas y por eso la cantidad de datos que tenemos no es suficiente para entrenar el modelo correctamente.

Además, a diferencia de los otros modelos, **Naïve Bayes funciona de forma adecuada** con pocos datos mientras que knn no se entrena, sino que usa los datos para hacer predicciones, y el árbol de decisión necesita una cantidad moderada de datos para hacer las particiones necesarias y funcionar de forma precisa.