Laboratorio 1 - SaludAlpes

Integrantes:
Sebastian Beltran - KNN
David Dominguez - Regresión Logística
Angelo Valero - Árbol Decisión.

Objetivo

SaludAlpes como entidad de salud colombiana, especializada en atender pacientes diagnosticados con diabetes, busca implementar un sistema que le permita generar un rápido diagnóstico para poder agilizar el proceso de análisis resultados clínicos. De tal forma que reduzca los tiempos de confirmación de diabetes en pacientes de la entidad. Auxiliando y agilizando el inicio del tratamiento de estos pacientes con diabetes confirmada.

Carga y exploración de los datos

La importación de los datos fue realizada desde un archivo *csv*. El cual contenía 768 registros y 11 columnas con información relevante para el diagnóstico de un paciente.

Lo primero que se hizo fue una visualización de las primeras cinco filas para tener una idea de los datos con los que se iba a trabajar.

	Hair color	Pregnancies	Glucose	City	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	Red	6	148	New York	72	35	0	336	627	50	1
1	Black	1	85	New York	66	29	0	266	351	31	0
2	Red	8	183	New York	64	0	0	233	672	32	1
3	Black	1	89	New York	66	23	94	281	167	21	0
4	Black	0	137	New York	40	35	168	431	2288	33	1

Luego visualizamos el tipo de datos que se manejaba por cada columna.

```
Hair color                 object
Pregnancies                object
Glucose                    object
City                       object
BloodPressure              object
SkinThickness              object
Insulin                    object
BMI                         int64
DiabetesPedigreeFunction   object
Age                         int64
Outcome                    object
dtype: object
```

Buscamos si existen valores nulos.

```
Hair color                  0
Pregnancies                 0
Glucose                     0
City                        0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Limpieza y preparación.

En este punto queremos asignar el valor adecuado de cada columna, pues muchos de sus valores son enteros: 'Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin' , 'DiabetesPedigreeFunction' y 'Outcome'. Pandas los reconoce como *object*.

```
Hair color                  object
Pregnancies                 object
Glucose                     object
City                        object
BloodPressure               object
SkinThickness               object
Insulin                     object
BMI                          int64
DiabetesPedigreeFunction    object
Age                          int64
Outcome                     object
dtype: object
```

Al intentar cambiar el tipo de cada columna a entero, nos arroja un error: *ValueError: invalid literal for int() with base 10: '-'* Esto es debido a que existen datos o caracteres que no corresponden en la columna a cambiar, en este caso existe el carácter *'-'* el cual no nos permite cambiar el tipo de la columna.

```
     Hair color Pregnancies Glucose City BloodPressure SkinThickness Insulin  \
583      Black           -       -    -             -             -       -

      BMI DiabetesPedigreeFunction  Age Outcome
583   387                       -   42       -
```

Como podemos observar la fila 583 contiene varias columnas con este valor, lo cual no aporta información relevante para los modelos de ML y será eliminada.

Ahora convertimos cada columna en su correspondiente tipo de datos

```
Hair color                  object
Pregnancies                  int64
Glucose                      int64
City                        object
BloodPressure                int64
SkinThickness                int64
Insulin                      int64
BMI                          int64
DiabetesPedigreeFunction     int64
Age                          int64
Outcome                      int64
dtype: object
```

Selección de atributos.

En este punto deseamos conocer cuáles son las columnas más importantes para nuestro modelo, aquellas que aporten valor y sirvan para crear modelos más precisos y exactos. Para este punto necesitamos conocimiento sobre cada atributo y como este podría impactar en la predicción del modelo.

Primero convertimos los datos categóricos en datos numéricos para conocer si realizan algún aporte significativo en el diagnóstico de diabetes.

En este caso las columnas de Hair color y City son categóricas.

```
Black    685
Red       50
Blue      32
Name: Hair color, dtype: int64

New York    767
Name: City, dtype: int64
```

Convertimos los datos categóricos en enteros para poder manejarlos mejor.

```
#mapping para la columna de Hair color
cat_mapping_hair = {'Black':0,'Red':1,'Blue':2}
```

```
#mapping para la columna de City
cat_mapping_city = {'New York':0}
```

Resumen de todos los datos

	Hair color	Pregnancies	Glucose	City	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	767.000000	767.000000	767.000000	767.0	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000
mean	0.148631	3.839635	120.921773	0.0	69.096480	20.563233	79.903520	289.670143	432.395046	38.006519	0.349413
std	0.458537	3.368429	31.984561	0.0	19.366833	15.945349	115.283105	116.780873	336.144934	117.902397	0.477096
min	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.000000	1.000000	21.000000	0.000000
25%	0.000000	1.000000	99.000000	0.0	62.000000	0.000000	0.000000	251.500000	205.500000	24.000000	0.000000
50%	0.000000	3.000000	117.000000	0.0	72.000000	23.000000	32.000000	309.000000	337.000000	29.000000	0.000000
75%	0.000000	6.000000	140.500000	0.0	80.000000	32.000000	127.500000	359.000000	592.000000	41.000000	1.000000
max	2.000000	17.000000	199.000000	0.0	122.000000	99.000000	846.000000	671.000000	2329.000000	3256.000000	1.000000

Podemos observar que el atributo de **City** solo tiene el valor de cero para toda la columna. En este caso **City** no realizará ningún aporte para los modelos de ML. Entonces procedemos a eliminar esta columna.

```
del df_diagnostics_t['City']
```

Ahora. El diccionario nos indica que la presión arterial debe contener valores mayores a 0, entonces comprobamos si cumple con esta condición.

```
df_diagnostics_t.BloodPressure.count
```

```
<bound method Series.count of 0      72
```

Podemos observar que existen 72 filas que no cumplen con esta condición, por tal motivo, determinamos cambiar los valores de 0 con la media de la columna para no perder información valiosa. También se puede deducir con esta información que una persona que muestre una presión sanguínea igual a 0, es que no estaría viva.

```
mediaPresion = df_diagnostics_t.BloodPressure.mean()
print(mediaPresion)
```

```
69.09647979139504
```

Ahora también el diccionario nos afirma que la glucosa debe tener valores mayores a 0, entonces comprobamos si cumple con esta condición.

```
df_diagnostics_t.Glucose.count
```

```
<bound method Series.count of 0      148
```

Podemos observar que existen 148 registros con el valor de glucosa igual a 0, entonces al tener esta cantidad considerable de registros no podemos prescindir de ellos. Por tal motivo decidimos conservarlos y cambiar este valor por la media de la columna de glucosa.

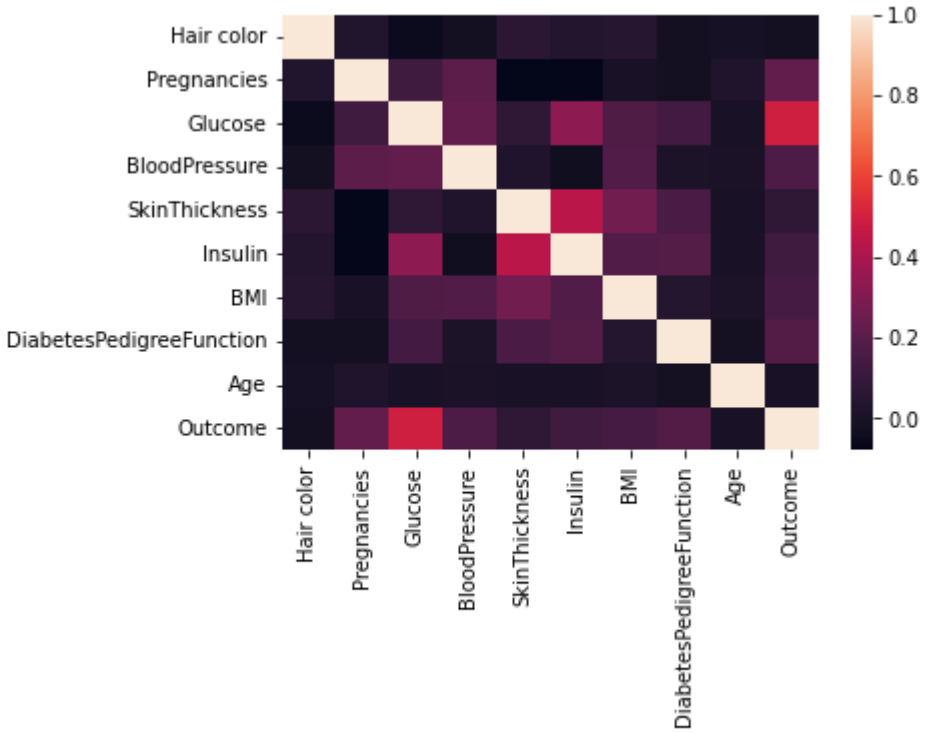
```
mediaGlucosa = df_diagnostics_t.Glucose.mean()
print(mediaGlucosa)
```

```
120.92177314211213
```

Resumen de los datos corregidos.

	Hair color	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000
mean	0.148631	3.839635	121.710051	72.249513	20.563233	79.903520	289.670143	432.395046	38.006519	0.349413
std	0.458537	3.368429	30.445781	12.123188	15.945349	115.283105	116.780873	336.144934	117.902397	0.477096
min	0.000000	0.000000	44.000000	24.000000	0.000000	0.000000	0.000000	1.000000	21.000000	0.000000
25%	0.000000	1.000000	99.500000	64.000000	0.000000	0.000000	251.500000	205.500000	24.000000	0.000000
50%	0.000000	3.000000	117.000000	72.000000	23.000000	32.000000	309.000000	337.000000	29.000000	0.000000
75%	0.000000	6.000000	140.500000	80.000000	32.000000	127.500000	359.000000	592.000000	41.000000	1.000000
max	2.000000	17.000000	199.000000	122.000000	99.000000	846.000000	671.000000	2329.000000	3256.000000	1.000000

A continuación se realizará una gráfica para visualizar la correlación que existe entre los datos.



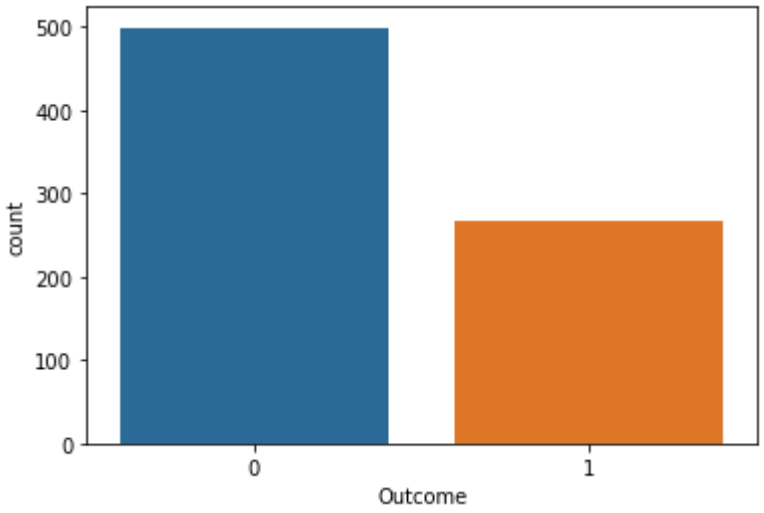
Se puede apreciar los atributos que tienen correlación con la variable de la etiqueta. Entre más brillante mayor será la correlación, en este caso vemos que las variables de **Age** y **Hair color** no tienen ninguna correlación con la variable de **Outcome** o con el diagnóstico de diabetes. Por tal motivo podemos prescindir de ellas.

Atributos que se usarán en los modelos.

```
Data columns (total 8 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               767 non-null    int64
 1   Glucose                   767 non-null    float64
 2   BloodPressure             767 non-null    float64
 3   SkinThickness             767 non-null    int64
 4   Insulin                   767 non-null    int64
 5   BMI                       767 non-null    int64
 6   DiabetesPedigreeFunction  767 non-null    int64
 7   Outcome                   767 non-null    int64
dtypes: float64(2), int64(6)
memory usage: 53.9 KB
```

Cantidad de datos de cada clase.

```
0    499
1    268
```



Modelos.

Árbol de decisión - Angelo Valero

Definimos la variable objetivo.

```
# Seleccionamos la variable objetivo - 'Outcome'
Y = df_diagnostics_t['Outcome']
# Eliminamos la variable 'Outcome' del conjunto de datos
X = df_diagnostics_t.drop(['Outcome'], axis=1)
```

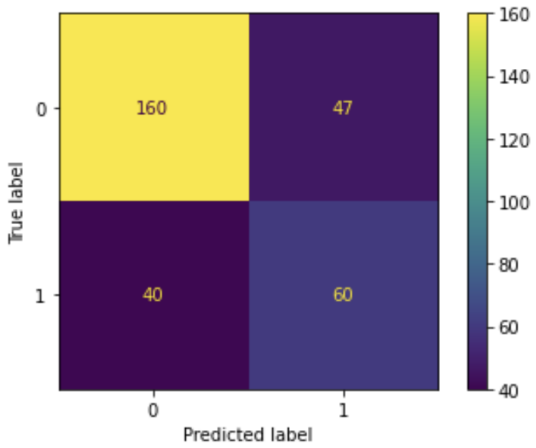
Divididos los datos de entrenamiento y de prueba. Seleccionamos el 40% de nuestros datos como el conjunto de prueba, debido a que tenemos poca cantidad y determinamos que esta cantidad es considerable y acertada para el entrenamiento del árbol. Caso contrario si se tuviera una cantidad enorme de datos, lo cual seleccionaremos una cantidad mínima de entrenamiento que sería entre el 10% u 1%.

```
# Dividir los datos en entrenamiento y test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4, random_state=0)
```

Escogimos como criterio la entropía, pues buscamos la pureza de los nodos y la división adecuada para que el árbol tenga una correcta ramificación.

```
# Crear arbol de decisión. El criterio de pureza sera la entropía
arbol = DecisionTreeClassifier(criterion='entropy', random_state=0)
```

Métricas de rendimiento.




```
Exactitud: 0.72
Recall: 0.6
Precisión: 0.5607476635514018
Puntuación F1: 0.5797101449275363
```

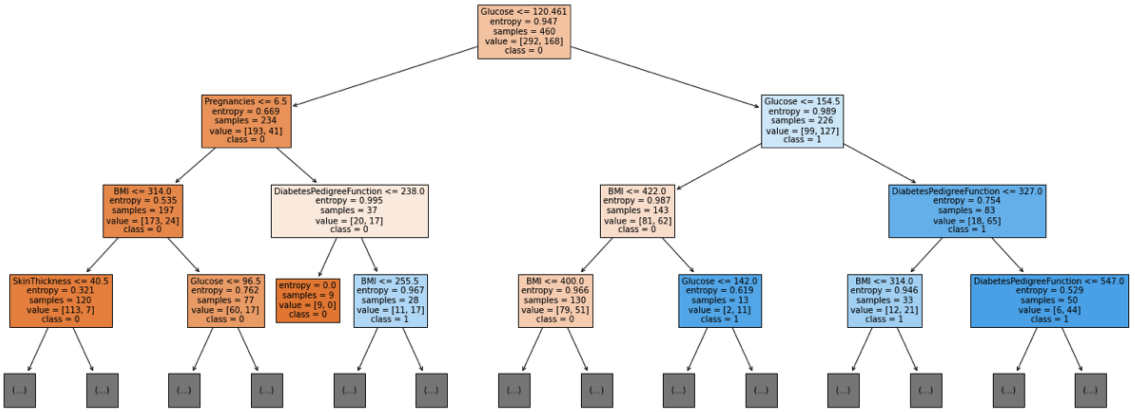
Encontramos unas métricas adecuadas. Esperamos tener una exactitud alta debido a que la exactitud se relaciona con todas las predicciones correctas, lo cual es deseado en este escenario donde un diagnóstico bueno o malo puede ser determinante.

Interpretación del modelo.

En esta tabla podemos ver *peso para cada atributo, entre mayor sea este más importante sera para la variable asociada.*

	Atributo	Importancia
0	Glucose	0.270700
1	DiabetesPedigreeFunction	0.184008
2	BMI	0.182408
3	Pregnancies	0.128118
4	SkinThickness	0.112983
5	Insulin	0.073249
6	BloodPressure	0.048534

Podemos observar que el atributo con mayor importancia es la Glucosa, que era de esperarse por nuestra tabla de correlaciones y pues que es usada para un acertado diagnóstico médico.



Finalmente nuestro árbol es determinado por la glucosa en la raíz y de aquí deriva en una segmentación hacia los nodos puros para una correcta predicción.

KNN - Sebastian Beltran

Primero definimos la variable objetivo Outcome ya que con este podremos predecir si una persona sufre de diabetes.
Luego dividimos los datos de entrenamiento y test. en este caso tomaremos 20% de los datos para el test.
de igual manera usaremos n=3 para realizar una prueba y mirar las metricas

```
# Se selecciona la variable objetivo, en este caso "Popularity_label".
Y = df_diagnostics_t['Outcome']
# Del conjunto de datos se elimina la variable "Popularity_label"
X = df_diagnostics_t.drop(['Outcome'], axis=1)
```

```
# Dividir los datos en entrenamiento y test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)
```

```
neigh = KNeighborsClassifier(n_neighbors=3)
 neigh = neigh.fit(X_train, Y_train)
```

```
y_pred = neigh.predict(X_test)
```

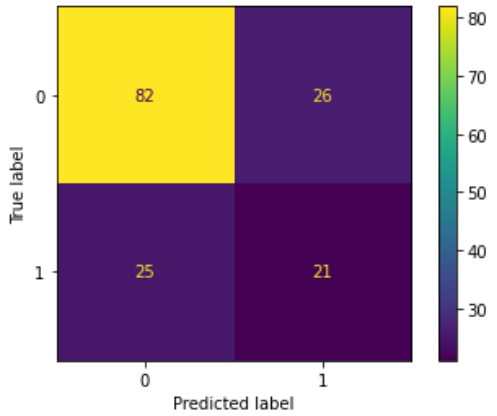
Métricas de Rendimiento

Al realizar una prueba con los datos de entrenamiento, nuestras métricas arrojan resultados no muy buenos en cuanto a exactitud, recall, precision y F1

```
# Se genera la matriz de confusión
confusion_matrix(Y_test, y_pred)
```

```
array([[82, 26],
       [25, 21]])
```

```
# Se puede visualizar la matriz de confusión
plot_confusion_matrix(neigh, X_test, Y_test)
plt.show()
```



```
# Mostrar reporte de clasificación
print(classification_report(Y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.77      0.76      0.76       108
           1       0.45      0.46      0.45        46

    accuracy                           0.67       154
   macro avg       0.61      0.61      0.61       154
weighted avg       0.67      0.67      0.67       154
```

```
print('Exactitud: %.2f' % accuracy_score(Y_test, y_pred))
print("Recall: {}".format(recall_score(Y_test,y_pred)))
print("Precisión: {}".format(precision_score(Y_test,y_pred)))
print("Puntuación F1: {}".format(f1_score(Y_test,y_pred)))
```

```
Exactitud: 0.67
Recall: 0.45652173913043476
Precisión: 0.44680851063829785
Puntuación F1: 0.45161290322580644
```

Normalizar

con el propósito de aumentar las métricas decidimos normalizar los datos que tenemos

```
df_diagnostics_pru = df_diagnostics_t.copy()
df_diagnostics_pru
normalized_df=(df_diagnostics_pru-df_diagnostics_pru.min())/(df_diagnostics_pru.max()-df_diagnostics_pru.min())

print(normalized_df)
```

```
     Pregnancies   Glucose  BloodPressure  SkinThickness   Insulin       BMI  \
0       0.352941  0.670968       0.489796       0.353535  0.000000  0.500745
1       0.058824  0.264516       0.428571       0.292929  0.000000  0.396423
2       0.470588  0.896774       0.408163       0.000000  0.000000  0.347243
3       0.058824  0.290323       0.428571       0.232323  0.111111  0.418778
4       0.000000  0.600000       0.163265       0.353535  0.198582  0.642325
..           ...       ...            ...            ...       ...       ...
763     0.588235  0.367742       0.530612       0.484848  0.212766  0.490313
764     0.117647  0.503226       0.469388       0.272727  0.000000  0.548435
765     0.294118  0.496774       0.489796       0.232323  0.132388  0.390462
766     0.058824  0.529032       0.367347       0.000000  0.000000  0.448584
767     0.058824  0.316129       0.469388       0.313131  0.000000  0.453055

     DiabetesPedigreeFunction  Outcome
0                    0.268900      1.0
1                    0.150344      0.0
2                    0.288230      1.0
3                    0.071306      0.0
4                    0.982388      1.0
..                        ...      ...
763                  0.073024      0.0
764                  0.014175      0.0
765                  0.104811      0.0
766                  0.149485      1.0
767                  0.134880      0.0
```

Train

ya con los datos normalizados realizamos el entrenamiento del modelo knn con n=3 y el 80% de los datos

```
# Se selecciona la variable objetivo, en este caso "Popularity_label".
Y = normalized_df['Outcome']
# Del conjunto de datos se elimina la variable "Popularity_label"
X = normalized_df.drop(['Outcome'], axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)
neigh = KNeighborsClassifier(n_neighbors=3) # arbol = DecisionTreeClassifier(criterion='entropy', random_s
neigh = neigh.fit(X_train, Y_train) # arbol = arbol.fit(X_train,Y_train)
y_pred = neigh.predict(X_test) # y_pred = arbol.predict(X_test)
print('Exactitud: %.2f' % accuracy_score(Y_test, y_pred))
print("Recall: {}".format(recall_score(Y_test,y_pred)))
print("Precisión: {}".format(precision_score(Y_test,y_pred)))
print("Puntuación F1: {}".format(f1_score(Y_test,y_pred)))
```

```
Exactitud: 0.71
Recall: 0.5217391304347826
Precisión: 0.5217391304347826
Puntuación F1: 0.5217391304347826
```

```
#Estadisticas con el conjunto de entrenamiento
y_pred = neigh.predict(X_train) # y_pred = arbol.predict(X_train)
print('Exactitud: %.2f' % accuracy_score(Y_train, y_pred))
print("Recall: {}".format(recall_score(Y_train,y_pred)))
print("Precisión: {}".format(precision_score(Y_train,y_pred)))
print("Puntuación F1: {}".format(f1_score(Y_train,y_pred)))
```

```
Exactitud: 0.84
Recall: 0.7072072072072072
Precisión: 0.8263157894736842
Puntuación F1: 0.762135922330097
```

Al entrenar el modelo podemos obtener los mejores hiperparametros para nuestro modelo, los cuales son p=1 y n=6

```
# https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClas

# Lista de Hiperparámetros a afinar
n_neighbors = list(range(1,10))
n_odd_neighbors = list(filter(lambda x: (x % 2 != 0), n_neighbors))
p=[1,2] #Función de distancia 1: manhattan, 2: euclidean, otro valor: minkowski

#Convert to dictionary
hyperparameters = dict(n_neighbors=n_neighbors, p=p)

#Create new KNN object
knn_2 = KNeighborsClassifier()

#Use GridSearch
mejor_modelo_knn = GridSearchCV(knn_2, hyperparameters, cv=10)

#Fit the model
mejor_modelo_knn.fit(X_train, Y_train)

#Print The value of best Hyperparameters
print('Best p:', mejor_modelo_knn.best_estimator_.get_params()['p'])
print('Best n_neighbors:', mejor_modelo_knn.best_estimator_.get_params()['n_neighbor
```

```
Best p: 1
Best n_neighbors: 6
```

Test

Finalmente hacemos el test del modelo knn y obtenemos unas métricas mucho mejores que las del comienzo

```
# Obtener el mejor modelo.
neigh_final = mejor_modelo_knn.best_estimator_
# Probemos ahora este modelo sobre test.
y_pred_train = neigh_final.predict(X_train)
y_pred_test = neigh_final.predict(X_test)
print('Exactitud sobre entrenamiento: %.2f' % accuracy_score(Y_train, y_pred_train))
print('Exactitud sobre test: %.2f' % accuracy_score(Y_test, y_pred_test))
```

```
Exactitud sobre entrenamiento: 0.78
Exactitud sobre test: 0.78
```

```
print(classification_report(Y_test, y_pred_test))
```

```
              precision    recall  f1-score   support

         0.0       0.79      0.93      0.85       108
         1.0       0.71      0.43      0.54        46

    accuracy                           0.78       154
   macro avg       0.75      0.68      0.70       154
weighted avg       0.77      0.78      0.76       154
```

Regresión Logística - David Dominguez

Decidimos usar una regresión logística para intentar resolver la tarea de clasificación, ya que este tipo de modelo da como resultado una distribución de probabilidad sobre las opciones, las cuales, aproximándose de forma conjunta con el uso de un umbral determinado, permite predecir si un elemento pertenece o no a un grupo dado.

Primero definimos los conjuntos de training y testing, para luego declarar el modelo a usar y finalmente entrenarlo con los datos previamente preprocesados.

```
# Seleccionamos la variable objetivo - 'Outcome'
Y = df_diagnostics_t['Outcome']
# Eliminamos la variable 'Outcome' del conjunto de datos
X = df_diagnostics_t.drop(['Outcome'], axis=1)
```

```
# Dividir los datos en entrenamiento y test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=0)
```

```
# Importamos la clase de scikit para realizar regresiones logísticas
from sklearn.linear_model import LogisticRegression
```

```
# Instanciamos la clase con un número de iteraciones de 1000, lo que permitirá que el algoritmo tenga convergencia
logisticRegr = LogisticRegression(solver='lbfgs', max_iter=1000)
# Entrenamos el modelo con los datos previamente separados
logisticRegr.fit(X_train, Y_train)
```

Tuvimos en cuenta hiper-parámetros como el tamaño del conjunto de testing, basándonos en diferentes pruebas realizadas. Particularmente probamos tamaños del 30% y 45%, sin embargo, 40% dio los mejores resultados.

Métricas de Rendimiento

Probamos el rendimiento del modelo comparando las predicciones hechas en el set de testing contra los resultados reales.
Al hacer esto, obtuvimos la matriz de confusión, las estadísticas básicas sobre el modelo, su precisión, recall y puntuación F1, como se observa en la imagen de abajo.

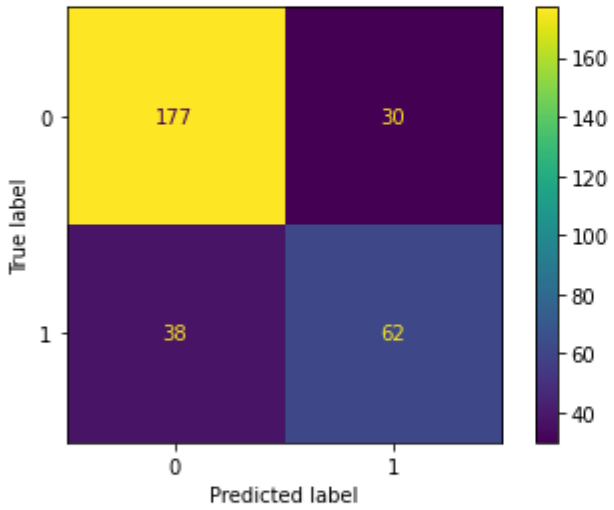
```python
# Comparamos las predicciones del modelo con los valores reales
score = logisticRegr.score(X_test, Y_test)
print(score)
```

```
0.7835497835497836
```

```python
# Generamos la matriz de confusión
confusion_matrix(Y_test, predictions)
```

```
array([[193,  14],
       [ 50,  50]], dtype=int64)
```

```python
# Se puede visualizar la matriz de confusión
plot_confusion_matrix(neigh, X_test, Y_test)
plt.show()
```




```python
# Mostrar reporte de clasificación
print(classification_report(Y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.79      0.93      0.86       207
           1       0.78      0.50      0.61       100

    accuracy                           0.79       307
   macro avg       0.79      0.72      0.73       307
weighted avg       0.79      0.79      0.78       307
```

Interpretación del modelo

Finalmente, de los coeficientes del modelo mostrados abajo, podemos deducir que los atributos con mayor peso para el modelo son el estado de embarazo y la glucosa.

```
# Mostramos una tabla con los coeficientes de la regresión realizada, ordenada por importancia
import_atributo = pd.DataFrame( data={ "Atributo": X_train.columns,"Importancia": logisticRegr.coef_[0] } )
import_atributo = import_atributo.sort_values(by='Importancia',ascending=False).reset_index(drop=True)
import_atributo
```

	Atributo	Importancia
0	Pregnancies	0.100424
1	Glucose	0.037739
2	SkinThickness	0.008230
3	BMI	0.000942
4	DiabetesPedigreeFunction	0.000771
5	Insulin	-0.001454
6	BloodPressure	-0.002071

Esto va de acuerdo a nuestras hipótesis, excepto por el embarazo. Sin embargo, sí esperábamos ver una fuerte relación entre la glucosa y tener o no diabetes.

Elección del mejor modelo

A partir de los resultados, vemos que en general el modelo basado en regresión logística es el que tiene mejor rendimiento en todas las métricas. A partir de esto, consideramos que es el modelo que finalmente se debe implementar.

El hecho de que regresión logística haya tenido mejores resultados se debe a que, como se observó en los diagramas de dispersión del tablero de control, los datos tienen formas definidas al graficarlos, lo cual equivale al tipo de dato ideal para la técnica. Adicionalmente, este modelo genera una “nube de probabilidad”, o distribución, alrededor de los posibles resultados, lo que, al final, con una función de activación permite producir un output binario, totalmente basado en probabilidades.

Otra cosa a tener en cuenta en la elección del modelo es que en este campo particular, clasificar un dato como un falso positivo es tan grave como clasificarlo como un falso negativo, debido a esto, debemos guiarnos principalmente por los promedios y no por la precisión de la clasificación de uno de los dos tipos de datos en particular.