Trabajo Red Neuronal Artificial

Jaime García Lozano

10 de abril de 2023

Índice

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

1. Introducción y descripción de los datos.

El *dataset* que trataremos en este trabajo ha sido extraido de la página *kaggle* [1]. Este consiste en los resultados de una encuesta a 400000 personas estadounidenses durante el año 2020. Los datos extraídos contienen información de cada participante (sexo, raza o edad) junto a ciertos hábitos o características que pueden llevar al individuo a sufrir una enfermedad cardíaca (fumar, no hacer deporte, tener sobrepeso, etc.) y una columna indicando si este la ha sufrido alguna vez o no. Por tanto, nuestra variable dependiente será si el sujeto es susceptible de sufrir una enfermedad de este tipo.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

[3]:
```
data = pd.read_csv('/content/drive/MyDrive/TD_Redes_neuronales/
↪heart_2020_cleaned.csv')
```

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

	0	1	2	3	4
HeartDisease	No	No	No	No	No
BMI	16.6	20.34	26.58	24.21	23.71
Smoking	Yes	No	Yes	No	No
AlcoholDrinking	No	No	No	No	No
Stroke	No	Yes	No	No	No
PhysicalHealth	3.0	0.0	20.0	0.0	28.0
MentalHealth	30.0	0.0	30.0	0.0	0.0
DiffWalking	No	No	No	No	Yes
Sex	Female	Female	Male	Female	Female
AgeCategory	55-59	80 or older	65-69	75-79	40-44
Race	White	White	White	White	White
Diabetic	Yes	No	Yes	No	No
PhysicalActivity	Yes	Yes	Yes	No	Yes
GenHealth	Very good	Very good	Fair	Good	Very good
SleepTime	5.0	7.0	8.0	6.0	8.0
Asthma	Yes	No	Yes	No	No
KidneyDisease	No	No	No	No	No
SkinCancer	Yes	No	No	Yes	No

Tabla 1: Primeras filas del conjunto de datos. (Nota: se ha hecho la transpuesta para que cupieran todas las columnas.)

[ ]:
```
data.shape
```

[ ]:
```
(319795, 18)
```

Extraemos las filas con *NaNs*, si es que los hay.

[4]:
```
df=data.dropna()
```

2. Arreglo de las variables categóricas

Tipos de variables categóricas que tenemos:

- **Binarias**: Smoking, AlcoholDrinking, Stroke, DiffWalking, Sex, PhysicalActivity, Asthma, KidneyDisease.
- **Ordinales**: AgeCategory, GenHealth.
- **Multicategóricas no ordinales**: Race, Diabetic.

Tenemos estas variables en formato *string* y las tenemos que pasar a numéricas.

2.1. Binarias

En este caso nos limitamos a asignar un 0 a "No" y un 1 a "Yes".

[5]:
```
dummies=["HeartDisease","Smoking","AlcoholDrinking",␣
↪"Stroke","DiffWalking","Sex","PhysicalActivity","Asthma","KidneyDisease","SkinCancer"]

for dummy in dummies:
  df[dummy]=df[dummy].astype("category").cat.codes
```

[ ]:
```
df[["HeartDisease","Smoking","AlcoholDrinking",␣
↪"Stroke","DiffWalking","Sex","PhysicalActivity","Asthma","KidneyDisease","SkinCancer"]].
↪head()
```

	HeartDisease	Smoking	AlcoholDrinking	Stroke	DiffWalking	Sex	PhysicalActivity	Asthma	KidneyDisease	SkinCancer
0	0	1	0	0	0	0	1	1	0	1
1	0	0	0	1	0	0	1	0	0	0
2	0	1	0	0	0	1	1	1	0	0
3	0	0	0	0	0	0	0	0	0	1
4	0	0	0	0	1	0	1	0	0	0

Tabla 2: Variables *dummy* codificadas.

2.2. Ordinales

Se asigna un número a cada categoría siguiendo un orden establecido.

[6]:
```
df.AgeCategory=df.AgeCategory.astype("category").cat.codes
df.GenHealth=pd.factorize(df.GenHealth.astype("category").cat.
 ↪reorder_categories(["Poor","Fair","Good","Very good","Excellent"]),␣
 ↪sort=True)[0]
```

[ ]:
```
df[["AgeCategory","GenHealth"]].head()
```

	AgeCategory	GenHealth
0	7	3
1	12	3
2	9	1
3	11	2
4	4	3

Tabla 3: Variables categóricas ordinales codificadas.

2.3. Multicategóricas no ordinales

Cada clase se convierte en una variable binaria.

[7]:
```
df=pd.get_dummies(df, columns=["Race", "Diabetic"])
```

[ ]:
```
df.iloc[:,-10:].head()
```

	0	1	2	3	4
Race_American Indian/Alaskan Native	0	0	0	0	0
Race_Asian	0	0	0	0	0
Race_Black	0	0	0	0	0
Race_Hispanic	0	0	0	0	0
Race_Other	0	0	0	0	0
Race_White	1	1	1	1	1
Diabetic_No	0	1	0	1	1
Diabetic_No, borderline diabetes	0	0	0	0	0
Diabetic_Yes	1	0	1	0	0
Diabetic_Yes (during pregnancy)	0	0	0	0	0

Tabla 4: Variables dummy obtenidas a partir de codificar las variables multicategóricas no ordinales. (Nota: se ha hecho la transpuesta para que cupieran todas las columnas.)

3. Estudio de las correlaciones

[8]:
```
import seaborn as sns

plt.figure(figsize=(20,20))
cm = np.corrcoef(df[df.columns].values.T)
sns.set(font_scale=1.5)
hm = sns.heatmap(cm, cmap="vlag" ,cbar=True,annot=True, square=True, fmt='.2f',
annot_kws={'size': 15}, yticklabels=df.columns,xticklabels=df.columns)
plt.show();
```

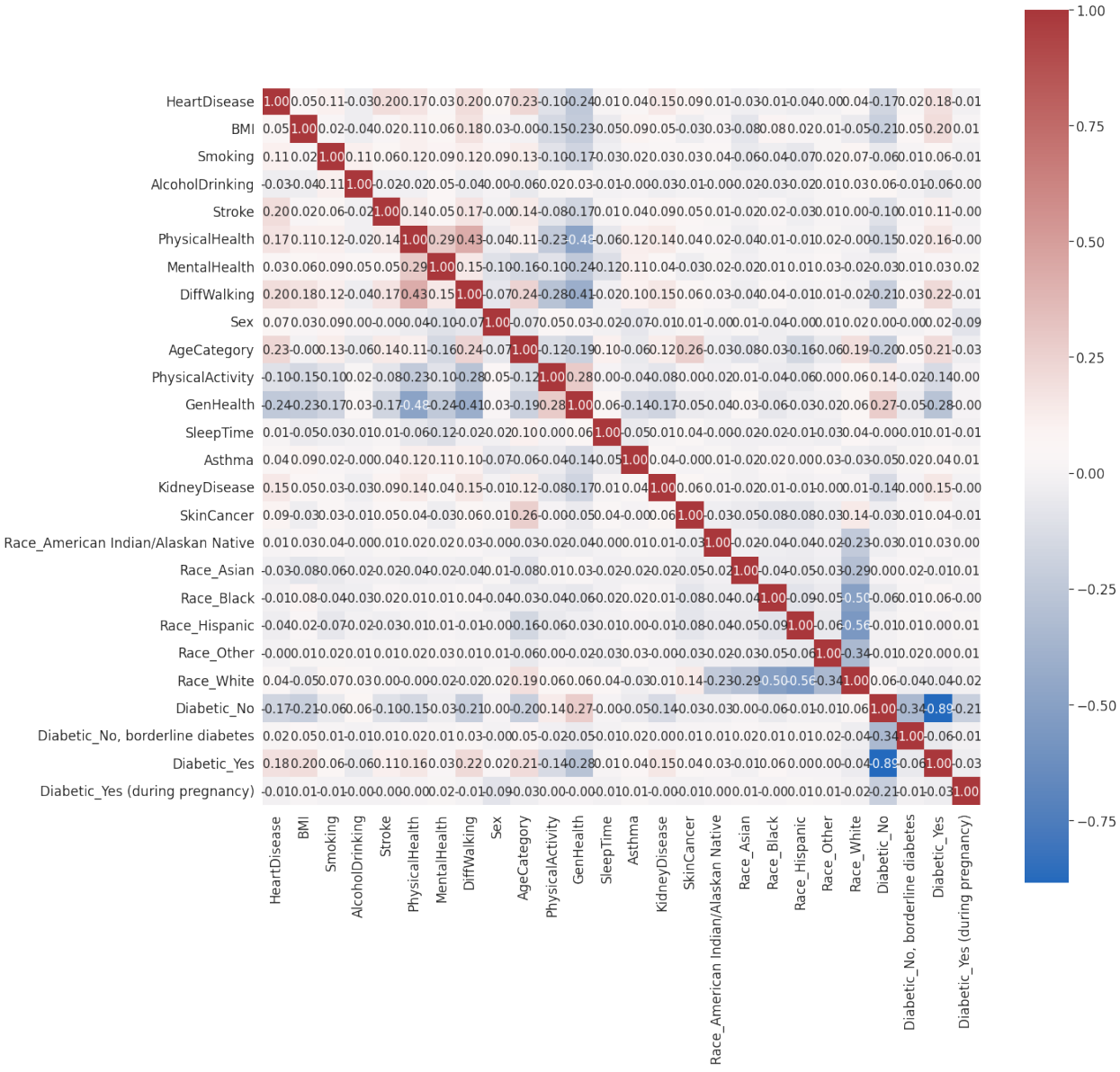

Figura 1: Mapa de calor representando las correlaciones entre las variables.

Eliminamos las variables *Diabetic_No* y *Race_Other* al ser redundantes, además de tener multicolinealidad perfecta en el caso de la primera.

[9]:
```
df=df.drop(["Diabetic_No", "Race_Other"],axis=1, inplace=False)
```

[ ]:
```
df.shape
```

[ ]:
```
(319795, 24)
```

4. Balanceado de los datos

Este conjunto de datos está fuertemente desbalanceado al haber muchos más individuos en la categoría de "no haber pasado nunca una enfermedad cardíaca" que en la otra.

[ ]:
```
df.HeartDisease.value_counts()
```

[ ]:
```
0    292422
1     27373
Name: HeartDisease, dtype: int64
```

El problema que surge es que la Red Nueronal se adaptará demasiado a la primera categroría durante la fase de entrenamiento, siendo incapaz de clasificar correctamente a un individuo de la segunda categoría.

Existen muchas maneras de enfrentarse a esta situación. En nuestro trabajo se ha optado por una combinación de las técnicas *oversampling* y *undersampling* [2]. De manera muy resumida, la primera añade nuevos datos a la clase minoritaria a partir de combinaciones de los datos originales, mientras que la segunda, elimina aleatoriamente observaciones de la categoría dominante.

[10]:
```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
```

[11]:
```
over = SMOTE(sampling_strategy=0.5, random_state=0)
under = RandomUnderSampler(sampling_strategy=0.9, random_state=0)
```

El parámetro *sampling_strategy* afecta en el tamaño que va a tener la clase modificada. No hay una opción mejor que otra, se trata de ir probando valores hasta encontrar los que mejor resultado den.

[12]:
```
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
```

Definimos la variable dependiente y las independientes:

[13]:
```
y=df.HeartDisease
X=df.iloc[:,1:df.shape[1]]
```

Dividimos entre *train* y *test*, siendo este último grupo el 30 % del conjunto de datos:

[14]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪random_state = 0)
```

[15]:
```
y_train.value_counts()
```

[15]:
```
0    204692
1     19164
Name: HeartDisease, dtype: int64
```

Balanceamos los datos de entrenamiento:

[16]:
```
X_train, y_train = pipeline.fit_resample(X_train, y_train)
```

[17]:
```
y_train.value_counts()
```

[17]:
```
0    113717
1    102346
Name: HeartDisease, dtype: int64
```

5. Escalado de las variables

Escalamos las variables para que el diferente orden de magnitud entre ellas no afecte a los resultados.

[18]:
```
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

6. Construcción de la Red Neuronal Artificial (RNA)

[ ]:
```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
```

Nuetra RNA estará compuesta por:

- Capa de entrada (23 entradas) y primera capa oculta (con 12 neuronas). Para elegir este segundo parámetro se suele escoger la media del total de entradas y salidas.
- Regularizador *Dropout* para evitar el *overfitting*. Lo que hace es seleccionar aleatoriamente en cada ciclo una serie de neuronas para que no participen en el entrenamiento. El parámetro 0.2 elegido indica que 1 de cada 5 variables va a ser excluida en cada ciclo.
- Segunda capa oculta (con 12 neuronas).
- Capa de salida (con una salida).

La única salida determinará con cierta probabilidad si el individuo es susceptible de sufrir una enfermedad cardíaca.

```
[ ]: def build_classifier(optimizer):
       # Creación de la red:
       classifier = Sequential()
       # Se añade la capa de entrada y la primera capa oculta:
       classifier.add(Dense(units = 12, kernel_initializer = "uniform",  activation =␣
     ↪"relu", input_dim = 23))
       # Función de regularización:
       classifier.add(Dropout(0.2))
       # Segunda capa oculta:
       classifier.add(Dense(units = 12, kernel_initializer = "uniform",  activation =␣
     ↪"relu"))
       # Capa de salida:
       classifier.add(Dense(units = 1, kernel_initializer = "uniform",  activation =␣
     ↪"sigmoid"))
       # Compilación de la RNA:
       classifier.compile(optimizer = optimizer, loss = "binary_crossentropy",␣
     ↪metrics = [keras.metrics.Recall(name='recall'),keras.metrics.
     ↪Precision(name="precision"),'accuracy'])
       return classifier
```

La función Dense se encarga de generar las conexiones entre capas.

Descripción de los parámetros:

- **input_dim**: Nodos de entrada.
- **units**: Número de nodos o neuronas.
- **kernel_initializer**: distribución que determinará los pesos iniciales. Se ha utilizado la uniforme.
- **activation**: La función de activación. Se ha optado por el Rectificador Lineal Unitario en las dos capas ocultas y la función Sigmoide en la de salida. Esta última nos permite obtener un número entre 0 y 1 interpretable como una probabilidad de activación.
- **optimizer**: lo elegiremos vía una búsqueda exhaustiva.
- **loss**: función de pérdidas que será minimizada. Se ha utilizado la Entropía Cruzada Binaria.
- **metrics**: lista de métricas que queremos que evalúe el modelo para que corrija los errores en cada iteración. Aparte del *accuracy* (proporción de aciertos) nos interesa que el modelo ajuste correctamente la clase de positivos. Por ello hemos añadido las métricas *recall* y *precision* [3]:

$$Recall = \frac{TP}{TP + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

  Donde TP y FP son los verdaderos y falsos positivos respectivamente. *Recall* nos indica la proporción de casos reales positivos que han sido clasificados correctamente, mientras que *precision* nos da la proporción de aciertos en la clase positiva.

Generamos el clasificardor:

[ ]:
```
classifier = KerasClassifier(build_fn = build_classifier)
```

Creamos un diccionario con los hiperparámetros que se utilizarán en la búsqueda exhaustiva:

[ ]:
```
parameters = {
    'batch_size' : [64,128],
    'epochs' : [50,100],
    'optimizer' : ['adam', 'rmsprop']
}
```

Aplicamos la búsqueda exhaustiva. Utilizaremos la métrica *accuracy* para encontrar el mejor conjunto de hiperparámetros. (Nota: Las siguientes tres *chunks* han sido ejecutadas en *Spyder*, ya que la versión gratuita de *Google Colab* no permite un tiempo de ejecución tan largo)

[ ]:
```
grid_search = GridSearchCV(estimator = classifier,
                           param_grid = parameters,
                           scoring = 'precision',
                           cv = 10,
                           verbose=3)
```

[ ]:
```
grid_search=grid_search.fit(X_train, y_train)
```

En esta situación nos interesa detectar correctamente a las personas con enfermedad cardíaca. Por ello, establecemos el umbral de activación de la clase positiva en 0.3, lo que implica que todo aquel que esté en esta clase tendrá una enfermedad cardíaca con probabilidad del 30 % o más.

[ ]:
```
y_pred=(grid_search.predict_proba(X_test)[:,1]>0.3)
```

Una vez ajustado el modelo en *Spyder* y guardado el *y_pred* vía la librería *joblib*, cargamos los resultados:

[19]:
```
import joblib
```

[20]:
```
y_pred=joblib.load("/content/drive/MyDrive/TD_Redes_neuronales/y_pred.pkl")
```

[24]:
```
from sklearn.metrics import confusion_matrix, classification_report
```

[25]:
```
confusion_matrix(y_test, y_pred)
```

[25]:
```
array([[55943, 31787],
       [ 1450,  6759]])
```

[23]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.97      0.64      0.77     87730
           1       0.18      0.82      0.29      8209

    accuracy                           0.65     95939
   macro avg       0.58      0.73      0.53     95939
weighted avg       0.91      0.65      0.73     95939
```

Observamos unos resultados interesantes:

- Se ha conseguido un *recall* de 0.82 en la clase 1, lo que quiere decir que la RNA ha clasificado correctamente el 82 % de los casos reales positivos.
- A pesar de esto, la precisión de esta clase es bastante baja, implicando que hay muchos falsos positivos en proporción a los verdaderos positivos. Cosa que también se ve reflejada en el *recall* de la clase 0, que es bastante menor que la de la clase 1.

7. Conclusión

El problema planteado en este trabajo establece la situación en la que no nos interesa tanto que la RNA tenga un *accuracy* muy alto, ya que este sólo nos indica la proporción de aciertos globales. Con un conjunto de datos tan desbalanceado se podría haber dado el caso en el que esta métrica fuera muy alta a costa de sólo clasificar correctamente la clase mayoritaria.

Partiendo del hecho de que en esta situación es más perjudicial clasificar incorrectamente a un verdadero positivo que a uno negativo, se ha forzado al modelo a evitar lo primero a costa de tener muchos falsos positivos. A pesar de ello, la RNA clasifica correctamente a la mayoría (64 %) de los verdaderos negativos.

El objetivo a partir de este punto sería seguir haciendo pruebas con más modificaciones de la RNA (cambiando el número de capas, las funciones de activación, el número de ciclos, utilizar otras técnicas de balanceado, etc.), para conseguir aumentar el *recall* de la clase 1 y a la vez reducir el número de falsos positivos.

Referencias

[1] *dataset*, https://www.kaggle.com/kamilpytlak/fitting-the-best-logistic-regression-model/datae.

[2] *Over and undersampling*, https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/.

[3] *Recall and Precision*, https://en.wikipedia.org/wiki/Precision_and_recall.