Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Puebla

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 301)
TC3002B.301

Actividad 2.1: Entrenamiento de una RNA

Uziel Humberto López Meneses A01733922

Fecha:
24 de mayo del 2023

1. (15 pts) A partir de la base de datos de diabetes, implementar un proceso de balanceo en la BD, de tal forma que queden igual número de instancias por clase (se recomienda 1000 instancias, pero puede manejarse otro número).

   Este punto se realizó con el programa Weka ya que su itnerfaz gráfica permite no solamente balancear las instancias con pocos clicks, sino que las visualizaciones permiten comprobar que los filtros aplicados sobre los datos nos den el resultado que esperamos.

   Para balancear las cargas, se utilizó el filtro de Synthetic Minority Oversampling Technique (SMOTE). Los pasos para realizar el balanceo con este filtro fueron los siguentes:

   I. Con el atributo clase seleccionado, se aplicó el filtro SMOTE con sus valores default, exceptuando:
      A. classValue = 2, para aplicarlo sobre las clase tested_positive.
      B. percentage = 273.14, valor necesario para generar las instancias que le faltaban a la clase 2 para llegar a 1000 instancias.
      C. randomSeed = 57467, para tener una semilla más grande que la default y que sea un número primo.

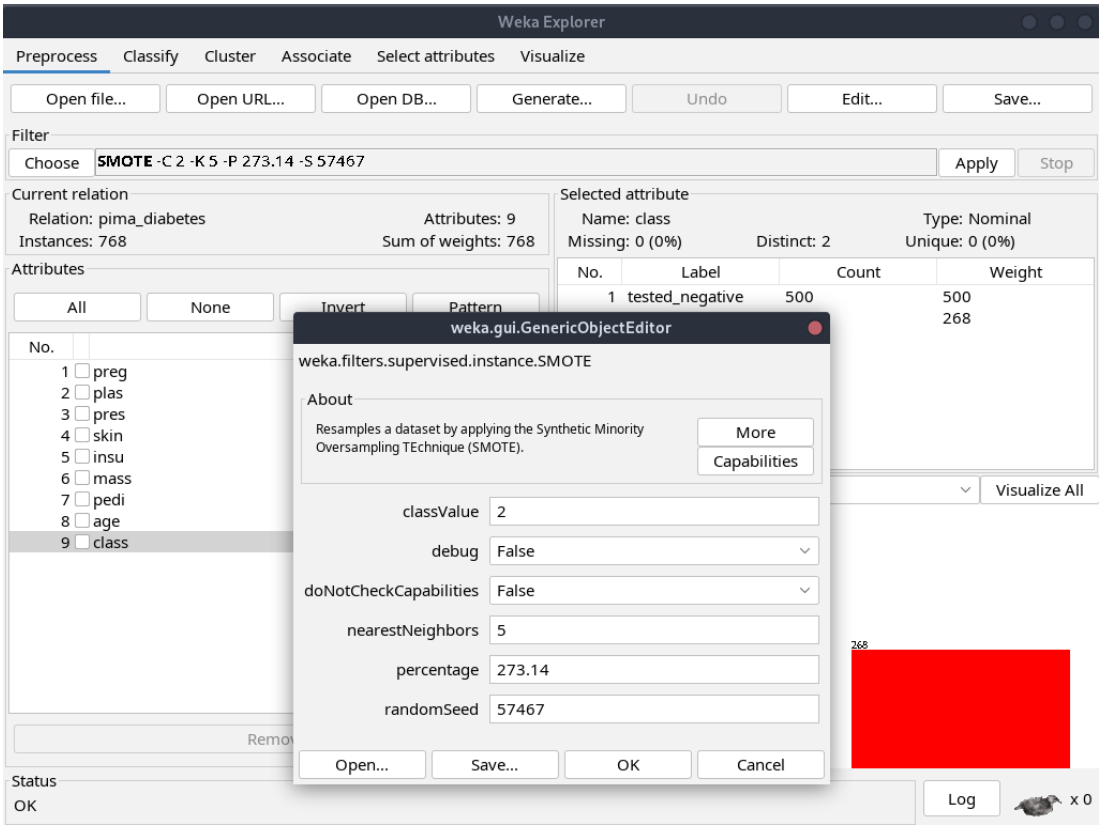

Fig. 1. Parámetros con los que se aplicó el filtro SMOTE sobre la clase tested_positive

II. Se aplicó el filtro de Randomize para cambiar mezclar las instancias recién generadas con el resto. Se usó una semilla con valor de un número primo relativamente grande.

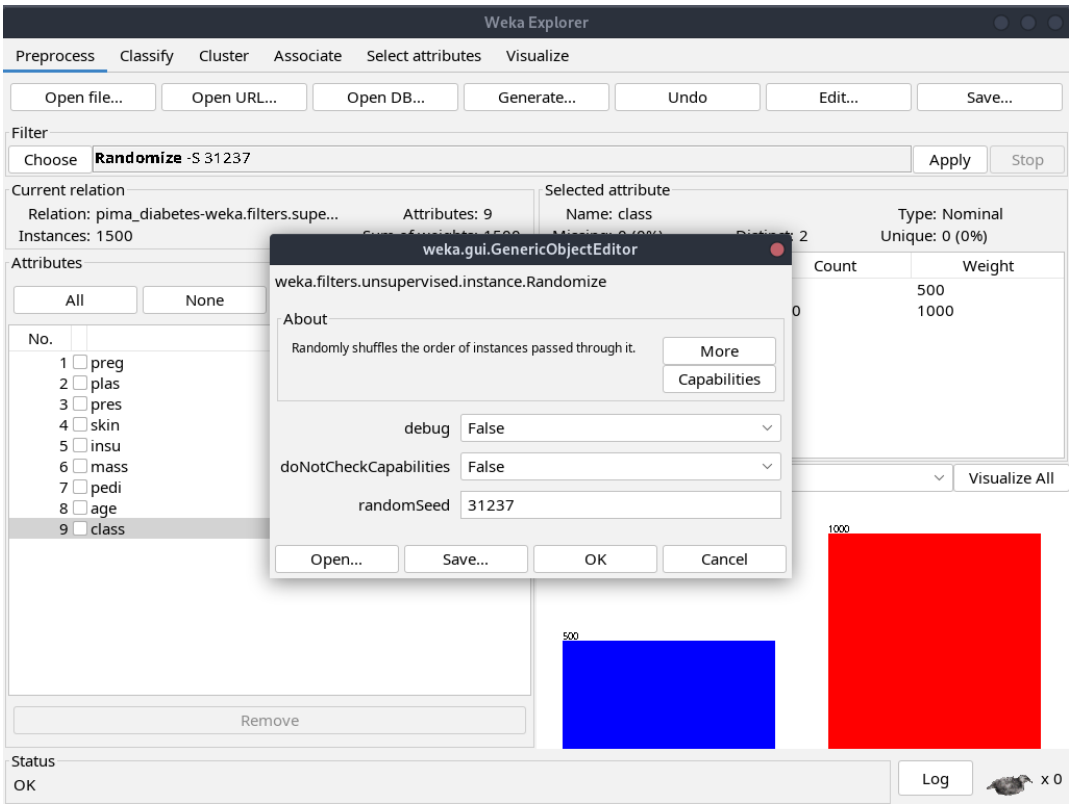

Fig. 2. Parámetros con los que se aplicó el filtro Randomize sobre todos los datos

III. Se aplicó el filtro de Randomize para cambiar mezclar las instancias recién generadas con el resto. Se usó una semilla con valor de un número primo relativamente grande.

IV. Con el atributo clase seleccionado, se aplicó el filtro SMOTE con sus valores default, exceptuando:

A. classValue = 1, para aplicarlo sobre las clase tested_positive.
B. percentage = 100, valor necesario para generar las instancias que le faltaban a la clase 2 para llegar a 1000 instancias.
C. randomSeed = 24421, para tener una semilla más grande que la default y que sea un número primo, diferente a la anterior.

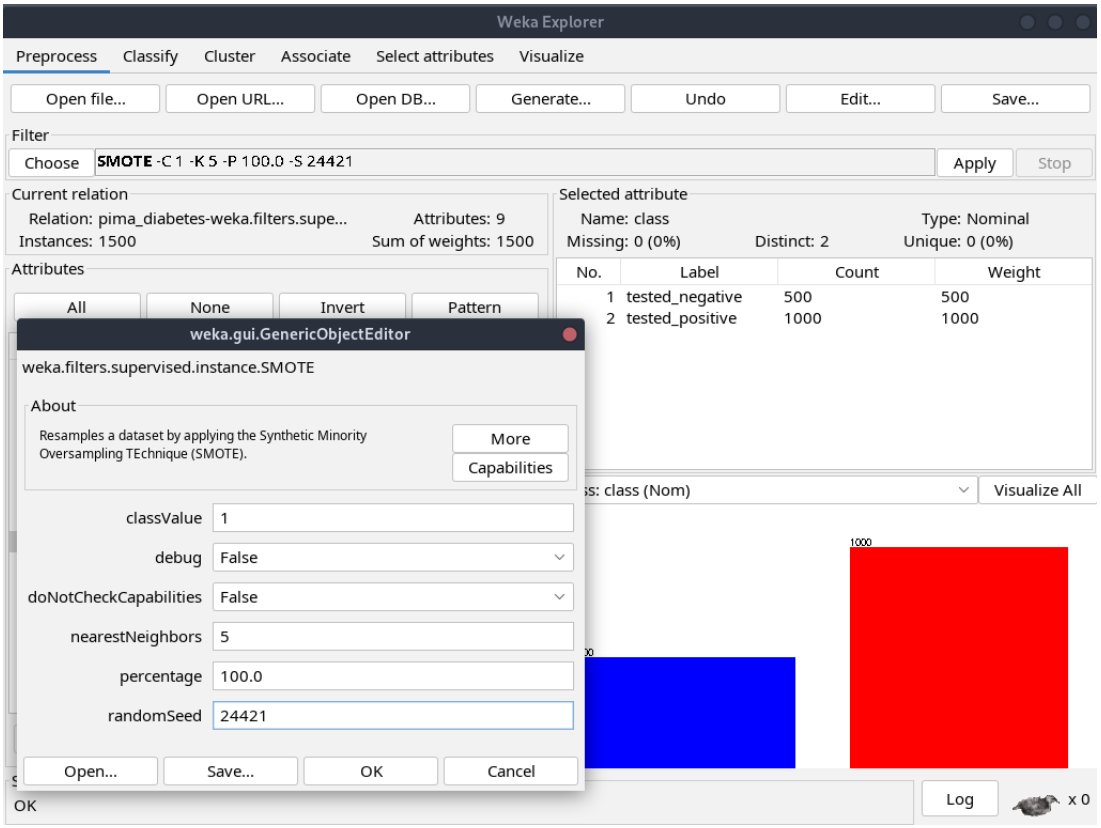

Fig. 3. Parámetros con los que se aplicó el filtro SMOTE sobre la clase tested_negative

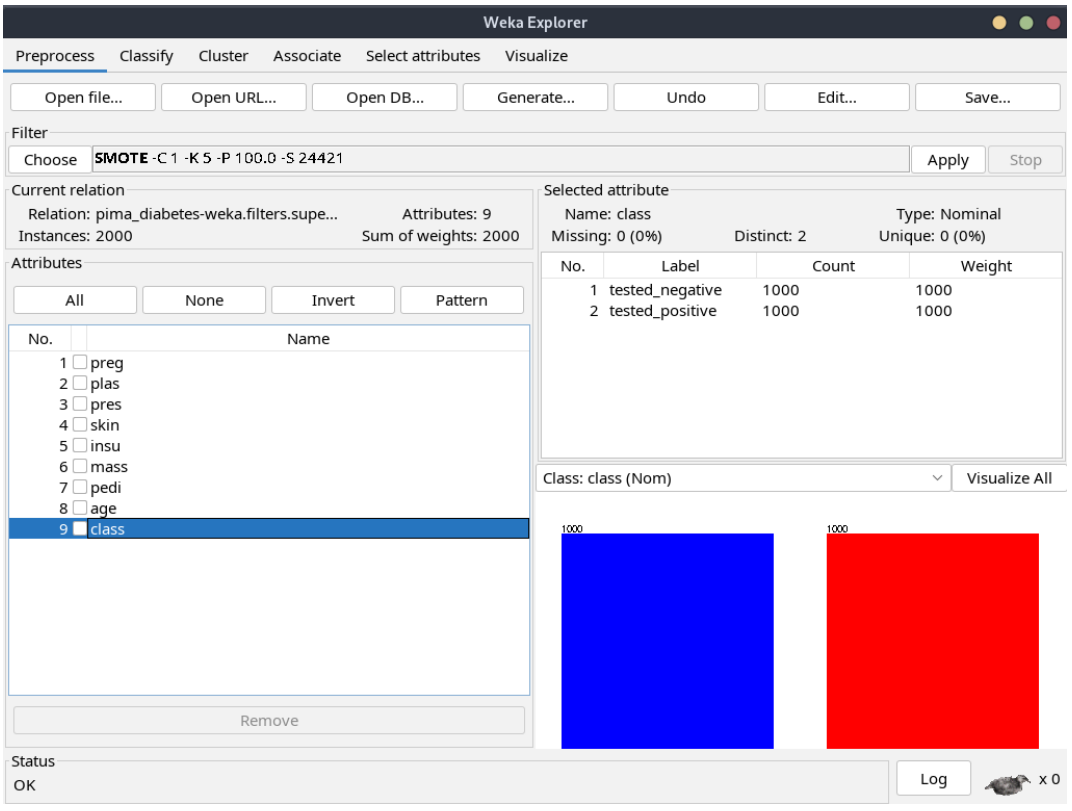

Fig. 4. Captura que muestra las clases balanceadas a 1000 instancias

V. Se aplicó nuievamente el filtro de Randomize para cambiar mezclar las instancias recién generadas con el resto. Se usó una semilla con valor de un número primo relativamente grande.

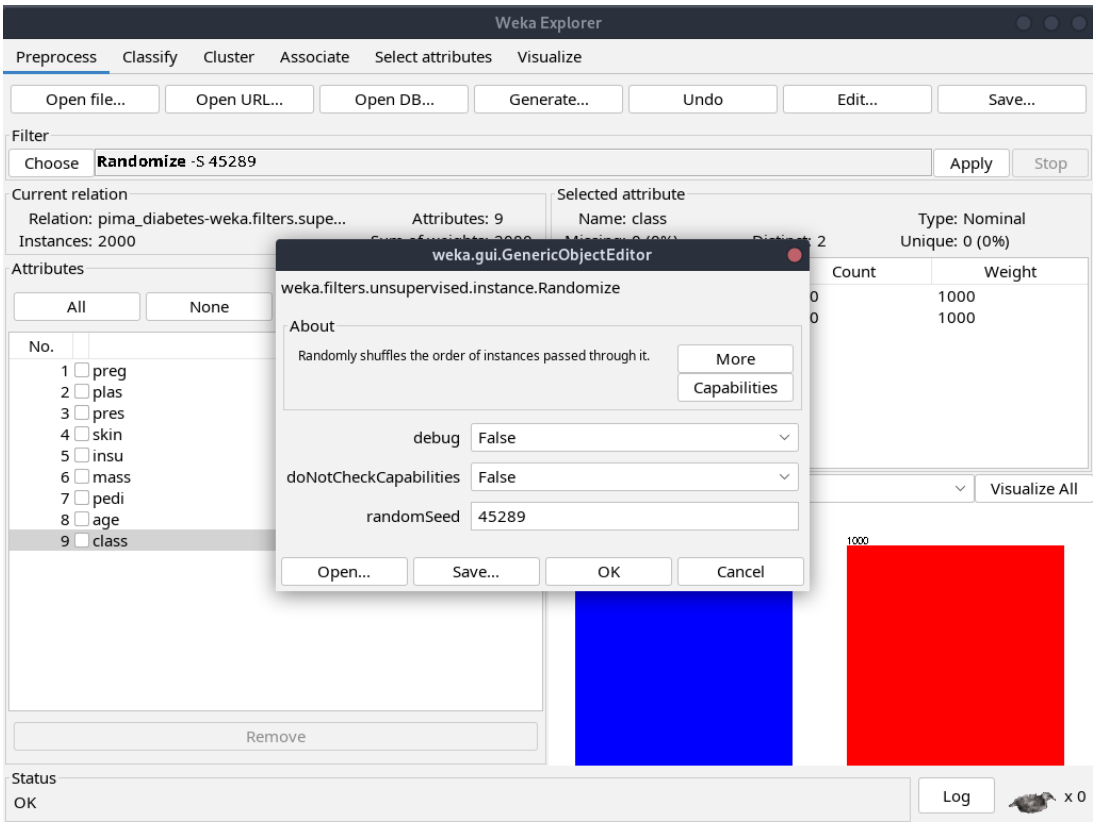

Fig. 5. Parámetros con los que se aplicó el filtro Randomize sobre todos los datos

2. (15 pts) A partir de la base de datos balanceada, particionar la misma de tal forma que se divida la base de datos en un conjunto training y un conjunto test global. Se puede manejar una partición 80 - 20, o bien una partición 70 - 30, por ejemplo. Generar dos archivos en los cuales se almacenen las particiones generadas. Se debe cuidar que las particiones se construyan con un proceso estratificado - estocástico, cuidando el balanceo por clases en cada partición.

   Nuevamente, se optó por utilizar Weka para realizar este punto ya que cuenta con un filtro para crear particiones de forma estratificada, el filtro StratifiedRemoveFolds.

   Para generar los conjuntos de datos traning y test globales, se siguieron los siguientes pasos:

   I. Se aplicó el filtro de StratifiedRemoveFolds, cambiando los valores:
      A. numFolds = 5, para seccionar los datos en 5 grupos que contengan cada uno el 20% de los datos.
      B. fold = 1, para seleccionar el único fold en el que vamos a hacer

C. invertSelection = True, para apartar un 80% de los datos, es decir, los datos del conjunto train global.
D. seed = 93529

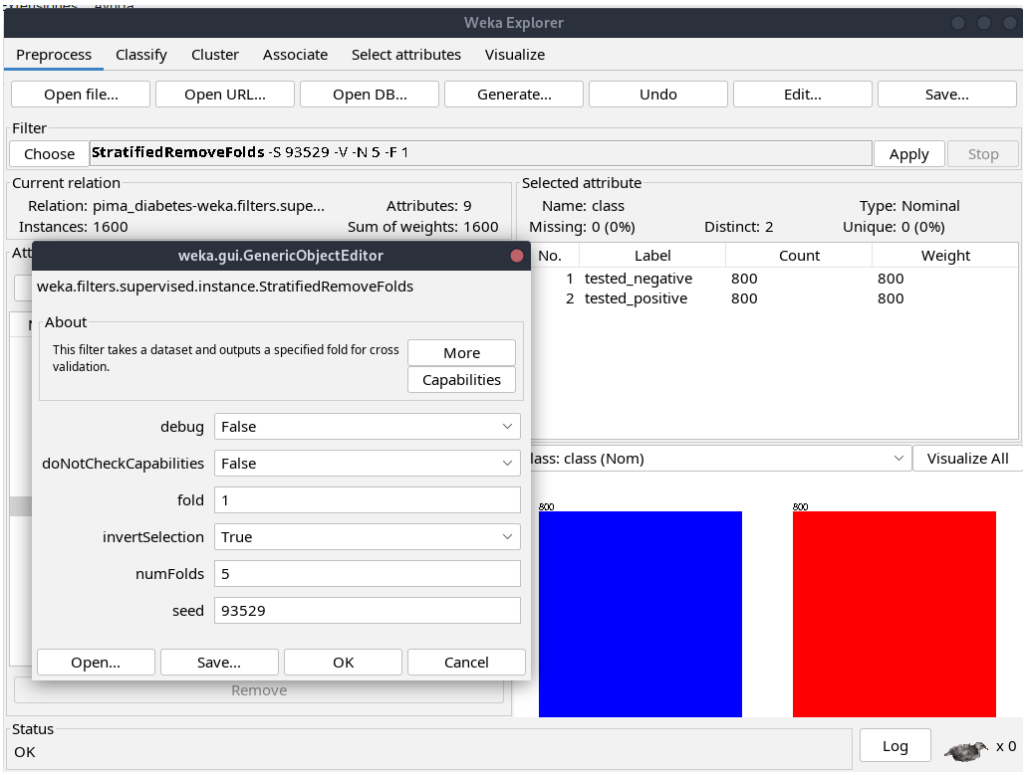

Fig. 6. Parámetros con los que se aplicó el filtro StratifiedRemoveFolds sobre todos los datos para obtener el conjunto training global.

II. Se aplicó el filtro Randomize con seed = 4019.
III. Se guardó el conjunto de datos con la opción Save de Weka en un archivo aparte, llamado diabetes_balanced_train.
IV. Se aplicó el filtro de StratifiedRemoveFolds, cambiando los valores:
    A. numFolds = 5, para seccionar los datos en 5 grupos que contengan cada uno el 20% de los datos.
    B. fold = 1, para seleccionar el único fold en el que vamos a hacer
    C. invertSelection = Faslse, para apartar un 20% de los datos, es decir, los datos del conjunto test global.35621
    D. seed = 93529
V. Se aplicó el filtro Randomize con seed = 35621.
VI. Se guardó el conjunto de datos con la opción Save de Weka en un archivo aparte, llamado diabetes_balanced_test.

3. (20 pts) Sobre el conjunto test global, aplicar un proceso de validación curzada K = 3. Cuidar que la representatividad de los patrones se mantenga lo más posible en cada

partición (generar un conjunto training y un conjunto validation por cada pliegue, guardando los archivos generados).

Sobre el conjunto training global, se aplicó el mismo filtro StratifiedRemoveFolds que en el paso anterior, con la diferencia de que ahora se usó un valor de numFolds = 3 y por cada fold se realizó el proceso de separación de conjunto train y conjunto test, guardando en carpetas llamadas fold_1, fold_2 y fold_3.

Para generar cada fold, se fue cambiando el valor fold dependiendo del fold que quisieramos generar, aplicando el filtro de Randomize cada que se generaba una partición test o train para garantizar aleatoriedad en los datos.

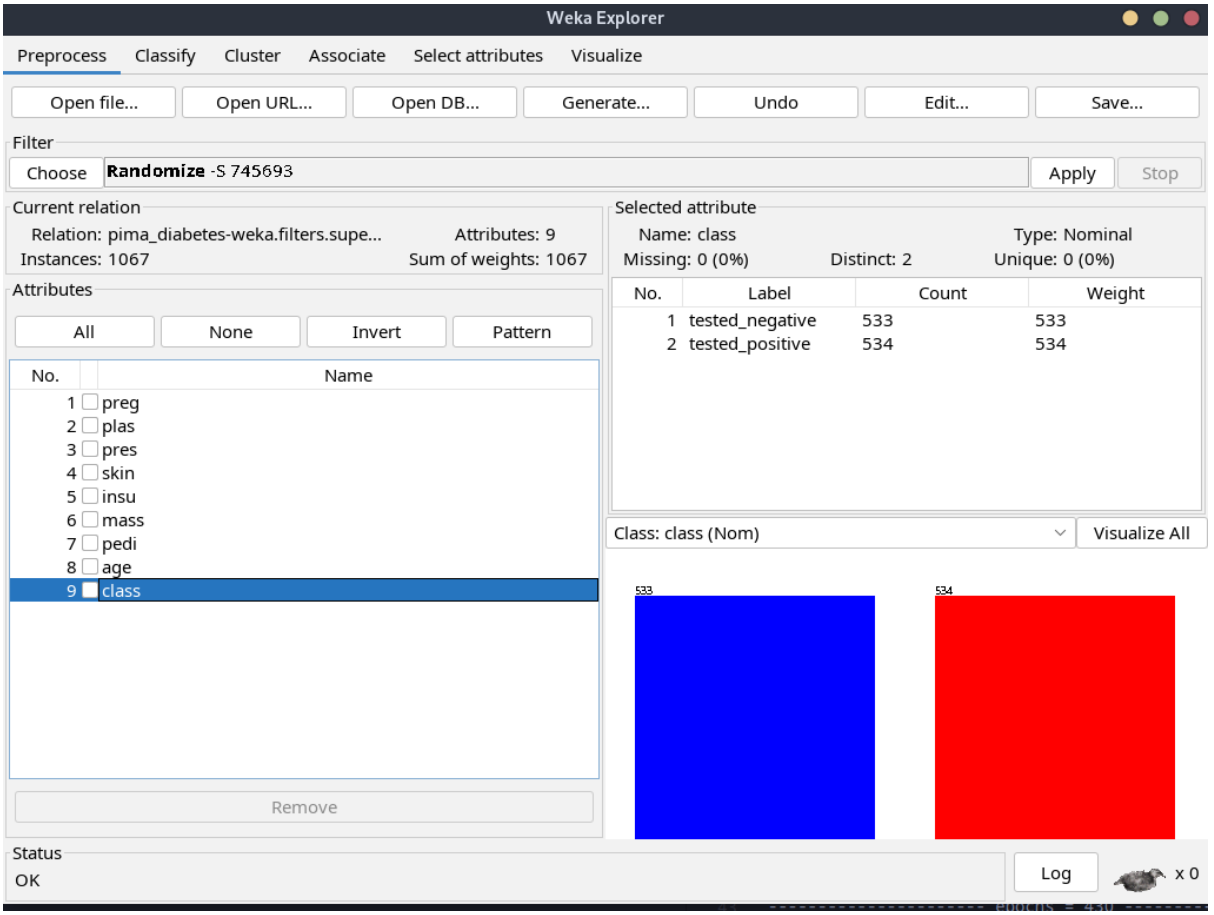

Fig. 6. Estado de los datos para el fold 3, en el conjunto training. Nótse que el último filtro en aplicarse antes de guardar el archivo fue Randomize

Mediante este proceso, se logró generar tres pares archivos .arff que contenían los conjuntos train y test de cada fold. Para poderlos utilizar en los pasos siguientes, cada uno de estos archivos fue copiado a un archivo del mismo nombre pero de extensión .csv, en el cual se quitó todos los metadatos presentes en el .arff original para dejar unicamente los headers de las columnas y los valores para cada atributo separados por comas.

4. (25 pts) Conducir un proceso experimental para construir un modelo de aprendizaje basado en una RNA sobre cada pliegue de la validación cruzada. Aplicar la técnica de la rejilla para la exploración de los hiper – parámetros de ajuste de la RNA. Se deberá cuidar qué para cada experimento, el resultado que se obtenga no este sobre – ajustado (overfitting), o bien quede en sub – ajuste (underfitting). Generar las gráficas correspondientes para mostrar en que momento el entrenamiento entra en overfitting.

   El código de Python y los archivos utilizados para esta sección están en el archivo .zip que se entregó junto con este reporte en Canvas. Para evitar desorden en este documento, solo se colocarán capturas de pantalla de secciones relevantes del código.

   Para cada fold, se experimentó con los siguientes metaparámetros, en ese orden:

   - Número de neuronas
   - Número de capas ocultas
   - Numero de épocas

   La exploración de valores de cada uno de estos metaparámetros se dio a través de la técnica de la rejilla, utilizando estos rangos:

   - Número de neuronas: De (número de atributos + número_de clases) / 2 a número de atributos + número_de clases. Es decir, para nuestros datos [5, 10]
   - Número de capas ocultas: de 1 a 5. Más de 5 sería excesivo dada la cantidad de atributos de nuestros datos.
   - Numero de épocas: De 0 550

   La experimentación por folds está contenida cada una en libretas de Jupyter distintas y al inicio de cada una de ellas se abren con pandas el archivos .csv correspondientes a cada fold, guardándolos en variables test_set_x y train_set_x donde x es el número de fold.

   De igual manera, por cada fold se inicializan algunas variables utilizadas en los distintos experimentos.

Fig. 7. Sección del código compartida en cada una de las libretas de cada experimento

Fold 1

El código de este fold se encuentra en la libreta entrenamiento_fold_1.ipynb.

Número de neuronas

Dentro de un for loop en el rango range(init_neurons, (init_neurons*2) + 1, 1), por cada valor de iteración se creó una instancia de la clase MLPClassifier del módulo sklearn.neural_network, la cual se creó dando los parámetros:

- solver='lbfgs'
- hidden_layer_sizes=hidden_layer_sizes
- random_state=98041

Instanciado el clasificador, se separan desde los archivos .csv tanto de training como de test las columnas que contienen unicamente atributos de la columna que contiene las clases de cada instancia y se guardan en DataFrames distintos.

Una vez separados los datos en DataFrames, se llama a la función .fit() del clasificador, dando como argumentos posicionales el DataFrame que contiene los valores de las columnas de atributos del conjunto training, seguido del DataFrame que contiene los valores de las clases del mismo conjunto training. Con esto, se entrena el modelo.

Cuando el proceso de entrenamiento ha concluido, se regresa un modelo que guardamos en una variable llamada model y posteriormente llamamos su método predict() con el DataFrame que contiene los valores de los atributos del conjunto test para probar qué tan bueno terminó siendo el modelo recién entrenado.

A partir de dicha predicción, se generan dos reportes, uno en forma de diccionario y otro como una clase especial que genera un reporte formateado. A partir de los datos contenidos en el diccionario del reporte, se calculan el error del modelo con respecto al conjunto training y elerror del modelo con respecto al conjunto test.

Este proceso de entrenamiento y medición del error se repite en cada uno de los experimentos con ligeras variaciones, como lo son cambios en los parámetros con los que se instancia la clase MLPCLassifier o manipulación especial de los datos del diccionario para obtener insights especificos por experimento, pero el procedimiento es esencialmente el mismo y se omitirá en secciones siguientes para evitar redundancia de información.

```python
for neurons_count in range(init_neurons, (init_neurons*2) + 1, 1):
    print(f"================{neurons_count} neurons==================")
    hidden_layer_sizes = (neurons_count)
    clasificador = MLPClassifier(solver='lbfgs',
                                 hidden_layer_sizes=hidden_layer_sizes,
                                 random_state=98041)


    #A PARTIR DE AQUÍ DE INICIA CON LA SEPARACIÓN Y CLASIFICACIÓN
    train_attribute_values = training_set_1[attributes]
    train_class_values = training_set_1[class_attribute]

    test_attribute_values = test_set_1[attributes]
    test_class_values = test_set_1[class_attribute]

    ################# Modelo ##########################
    model = clasificador.fit(train_attribute_values, train_class_values)
    ################# Clasificar  #################
    predict = model.predict(test_attribute_values)
    ############ Evaluar ############################
    report_dict = classification_report(test_class_values, predict, labels=class_attribute_name, output_dic
    report = classification_report(test_class_values, predict, labels=class_attribute_name)
    # record training set accuracy and error
    training_accuracy = (clasificador.score(train_attribute_values, train_class_values))
    training_error = (1.0 - clasificador.score(train_attribute_values, train_class_values))
    # record generalization accuracy and error
    test_accuracy = (clasificador.score(test_attribute_values, test_class_values))
    test_error = (1.0 - clasificador.score(test_attribute_values, test_class_values))
    #print(report)
    print(f"acc={report_dict['accuracy']}")
    print(f"training_accuracy = {training_accuracy}")
    print(f"test_accuracy = {test_accuracy}")
    #last_expermient_no += 1
    #results.append([])
```

Fig. 7. Sección del código que genera los experimentos para cada número de neuronas. Nótese que este es esencialmente el *blueprint* utilizado en cada uno de los experimentos realizados para este trabajo.

Cuando las iteraciones de este experimento concluyen, vemos el siguiente output:

```
================5 neurons==================
acc=0.7771535580524345
training_accuracy = 0.7560975609756098
test_accuracy = 0.7771535580524345
================6 neurons==================
acc=0.5224719101123596
training_accuracy = 0.5581613508442776
test_accuracy = 0.5224719101123596
================7 neurons==================
acc=0.6891385767790262
training_accuracy = 0.7035647279549718
test_accuracy = 0.6891385767790262
================8 neurons==================
acc=0.7808988764044944
training_accuracy = 0.7589118198874296
test_accuracy = 0.7808988764044944
================9 neurons==================
acc=0.7247191011235955
training_accuracy = 0.7176360225140713
test_accuracy = 0.7247191011235955
================10 neurons==================
acc=0.7771535580524345
training_accuracy = 0.7654784240150094
test_accuracy = 0.7771535580524345
```

Fig. 8. Output generado por el experimento sore el número de neuronas.

Podemos observar que el mejor accuracy se obtuvo con 8 neuronas. Sin embargo, el accuracy con 5 y 10 neuronas no están muy lejos de el accuracy con 8 neuronas, además de que sus valores de error se encuentran relativamente cercanos.

Por esto, decidí no descartar por el momento el usar 5 o 10 neuronas, aunque conceptualmente tenga más sentido tener tantas neuronas como atributos tengan nuestros datos. En el siguiente experimento se esperaba ver que las topologías con 8 neuronas tuvieran un mejor desempeño, pero buscaba encontrar algo de interés en las topologías con números distintos de neuronas.

Número de capas ocultas

Para probar rapidamente el número de capas ocultas por cada uno de los números de neuronas que son de interés, se implementó un ciclo for que recorre una tupla que contiene la cantidad de neuronas de interés.

Después por cada uno de los elementos de esta tupla, en un ciclo, se inicializa un arreglo vacío hidden_layer_sizes que representa la topología, donde cada índice es una capa oculta y el valor en dicho índice representa el número de neuronas en dicha capa. A este arreglo se le añade un elemento con valor igual al número de neuronas que se esté probando en una iteración dada, para así probar topologías con una a cinco capaz ocultas, con valores posibles de neuronas de 5, 8 y 10.

```
for n in (5, 8, 10):
    hidden_layer_sizes = []
    for _ in range(1, 6, 1):
        hidden_layer_sizes.append(n)
        print(f"================layers = {hidden_layer_sizes}==================")
        clasificador = MLPClassifier(solver='lbfgs',
                                     hidden_layer_sizes=hidden_layer_sizes,
                                     random_state=98041)
```

Fig. 9. Ciclos anidados que prueban las distintas combinaciones de número de neuronas y número de capas.

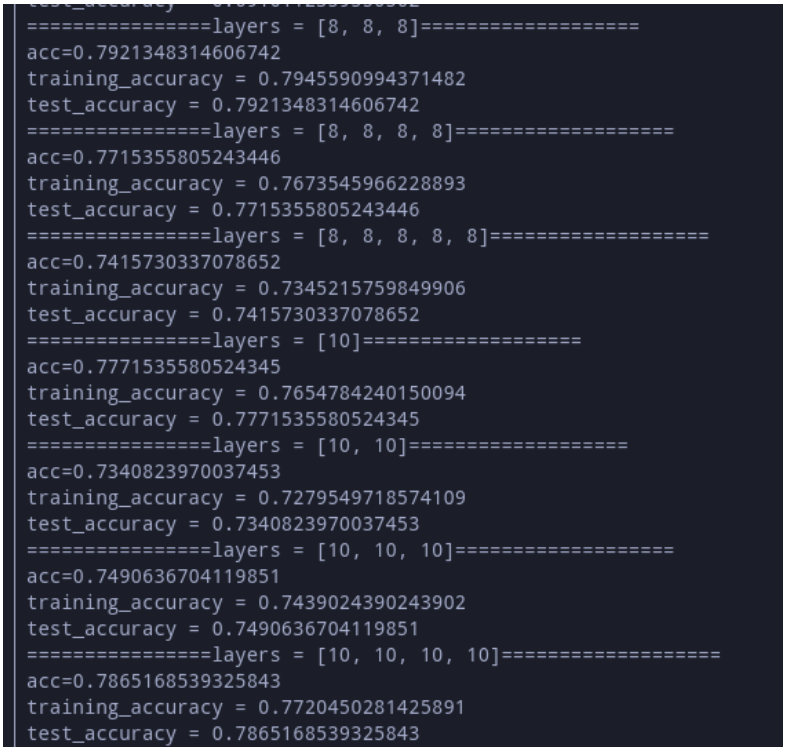

Fig. 10. Fragmento del output del experimento de capas ocultas. Output completo en la libreta correspondiente.

Como se esperaba desde el experimento anterior, las topologías con 8 neuronas presentan un buen rendimiento, tomando en cuenta que tienen un tamaño mas compacto que aquellas con 10 neuronas y la diferencia en los errores no es tan grande.

De todas las combinaciones probadas, la topología de configuración [8,8,8] (es decir, de tres capas con ocho neuronas cada una), presenta el segundo accuracy más alto, manteniendo un tamaño más compacto que la topología [10,10,10,10] y con una disparidad entre el error sobre el conjunto training y el conjunto test muy pequeña.

Número de épocas

La intención de esta experimentación es encontrar el número de épocas que nos de el mejor accuracy antes de que se entre en overfitting. Para ello, se declaro una variable epochs que contiene el rango [10, 350] en pasos de 5, que nos permitirá probar entrenamientos una gran variedad de épocas y permitiría ver rapidamente el punto de overfitting.

Con el rango epochs definido, se realiza un ciclo anidado en el que se prueban cada una de las topologías candidatas (en este caso solo es una) en combinación con cada una de los números de épocas en el rango. Los valores de las medidas de error de cada número de épocas se acumulan en un arreglo con el que posteriormente, con ayuda de la librería matplotlib, se genera una gráfica que muestra la comparación del comportamiento de las dos medidas de error conforme aumenta el número de épocas.

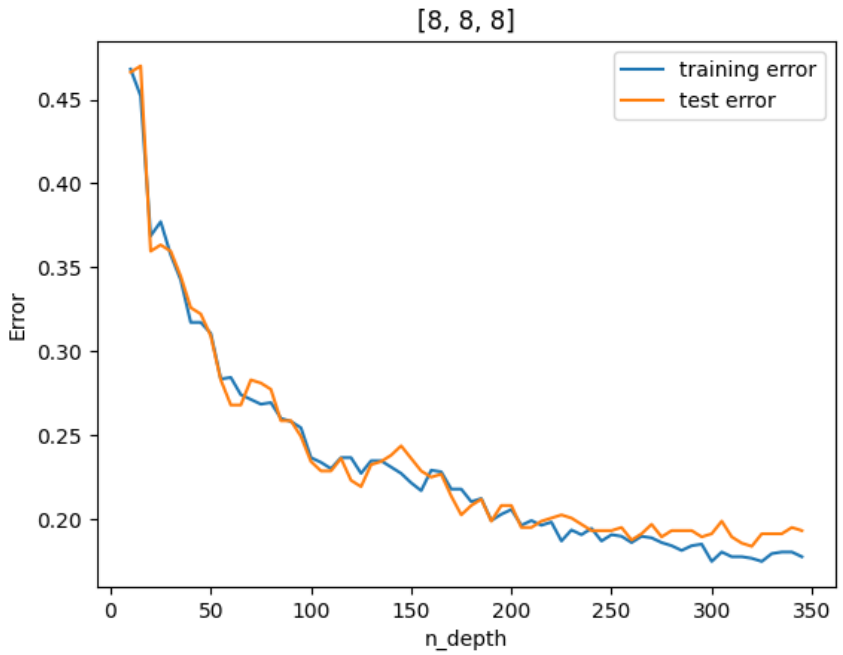

Fig. 11. Gráfica de errores de los conjuntos para la topología [8,8,8] en el rango [10, 350]

Con base en la gráfica 11 y el output de la celda en la libreta de Jupyter, podemos delimitar que el overfitting se da al rededor de 285 capas.

Learning rate y momentum

Utilizando la clase MLPCLassifier, conforme a la documentación de sklearn, no se pueden ajustar los valores de rate y momentum cuando se ocupa el solver 'lbfgs'. Previo al inicio de la experimentación aquí documentada, se probó la función 'sgd' con los valores default para el clasificador y mostró un rendimiento mucho más pobre que con el solver 'lbfgs', por lo que los experimentos utilizan exclusivamente esta última función.

Sin embargo, para comprobar que efectivamente se obtenía un peor modelo utilizando la función 'sgd', corrí un experimento que prueba las combinaciones de valores de learning rate dentro del rango [1,6] en pasos de dos con variaciones en el momentum en el rango [3,9] con saltos de 3, a su vez probando múltiples valores de épocas.

```
epochs = range(10, 225, 5) # añado un pequeño margen para comprobar que si se entra en
learning_rate_range = range(1,7,2)
momentum_range = range(3,11,3)

for _lr in learning_rate_range:
    lr = _lr / 10.0
    for _momentum in momentum_range:
        best_acc = -1
        momentum = _momentum / 10.0 if _momentum != 0 else 0
        print(f"================LR: {lr} mom={momentum}======================")
        training_accuracy = []
        test_accuracy = []
        training_error = []
        test_error = []
        for epoch in epochs:
            clasificador = MLPClassifier(solver='sgd',
                                hidden_layer_sizes=[8,8,8],
                                max_iter = epoch,
                                random_state=98041,
                                l      rate_init=lr,
                                m      momentum
                                )
```

Fig. 12. Ciclos anidados que prueban las combinaciones de learning rate, momentum y épocas dados los rangos vistos en la imágen.

Como se esperaba, el rendimiento del modelo entramado de esta forma fue muy pobre, teniendo un error de entre .44 y .50

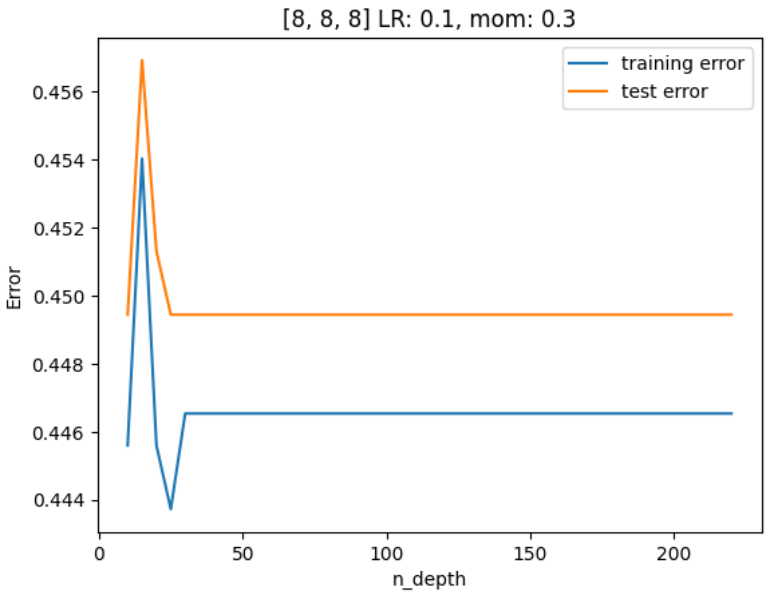

Fig. 13. Gráfica de errores para la topología cuando el LR=0.1 y momentum= 0.3, usando la función sgd.

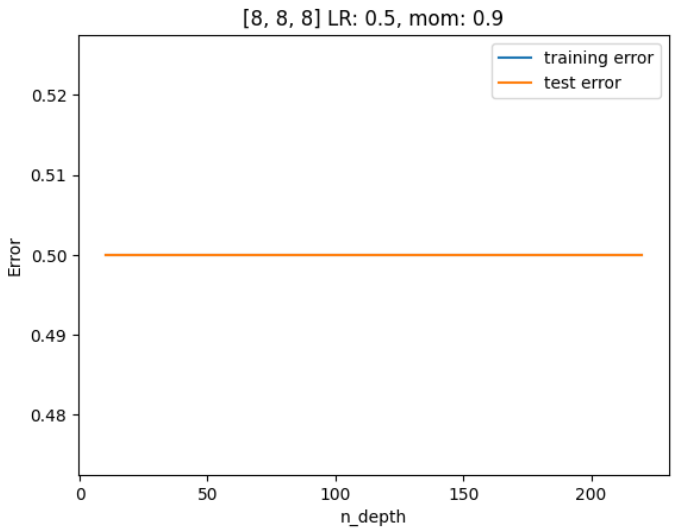

Fig. 14. Gráfica de errores para la topología cuando el LR=0.5 y momentum= 0.9, usando la función sgd.

Por esta razón, se optó por omitir los resultados de este experimento en los experimentos de los folds subsecuentes.

Resultado del entrenamiento

Finalmente, estos son los metaparámetros óptimos para este fold:

- Solver = 'lbfgs'
- Número de neuronas = 8
- hidden_layer_sizes (topología) = [8,8,8]
- Épocas = 260
- Learning Rate = N/A
- Momentum = N/A

Para concluir, se corrió una vez más el clasificador con los metaparámetros anteriores y se obtuvieron estos resultados:

Accuracy = 0.80711

	precision	recall	f1-score	support
tested_positive	0.81	0.80	0.81	267
tested_negative	0.80	0.82	0.81	267
accuracy			0.81	534
macro avg	0.81	0.81	0.81	534
weighted avg	0.81	0.81	0.81	534

Fold 2

El código de este fold se encuentra en la libreta entrenamiento_fold_2.ipynb.

Para este y el siguiente fold, se siguió la misma metodología y se utilizó el mismo código que en el fold 1, por lo que para evitar redundancia se omitirá la explicación detallada del código y se discutiran unicamente los resultados de cada experimento.

Número de neuronas

```
================5 neurons==================
acc=0.6435272045028143
training_accuracy = 0.6607310215557638
test_accuracy = 0.6435272045028143
================6 neurons==================
acc=0.5290806754221389
training_accuracy = 0.5304592314901593
test_accuracy = 0.5290806754221389
================7 neurons==================
acc=0.7035647279549718
training_accuracy = 0.7029053420805998
test_accuracy = 0.7035647279549718
================8 neurons==================
acc=0.7091932457786116
training_accuracy = 0.7357075913776945
test_accuracy = 0.7091932457786116
================9 neurons==================
acc=0.7073170731707317
training_accuracy = 0.7160262417994376
test_accuracy = 0.7073170731707317
================10 neurons==================
acc=0.7467166979362101
training_accuracy = 0.7647610121836926
test_accuracy = 0.7467166979362101
```

Fig. 15. Output generado por el experimento sore el número de neuronas en el fold 2.

Al ejecutar este experimento, a diferencia del entrenamiento con el fold 1, la ventaja de la topología con 8 neuronas no era tan evidente, ya que la topología con 10 neuronas presenta un accuracy superior y la diferencia entre los accuracies podría entrar dentro de un rango aceptable, por lo que en el siguiente experimento también se probaron topologías con 10 neuronas para ver si vale la pena el costo cómputo extra.

Número de épocas

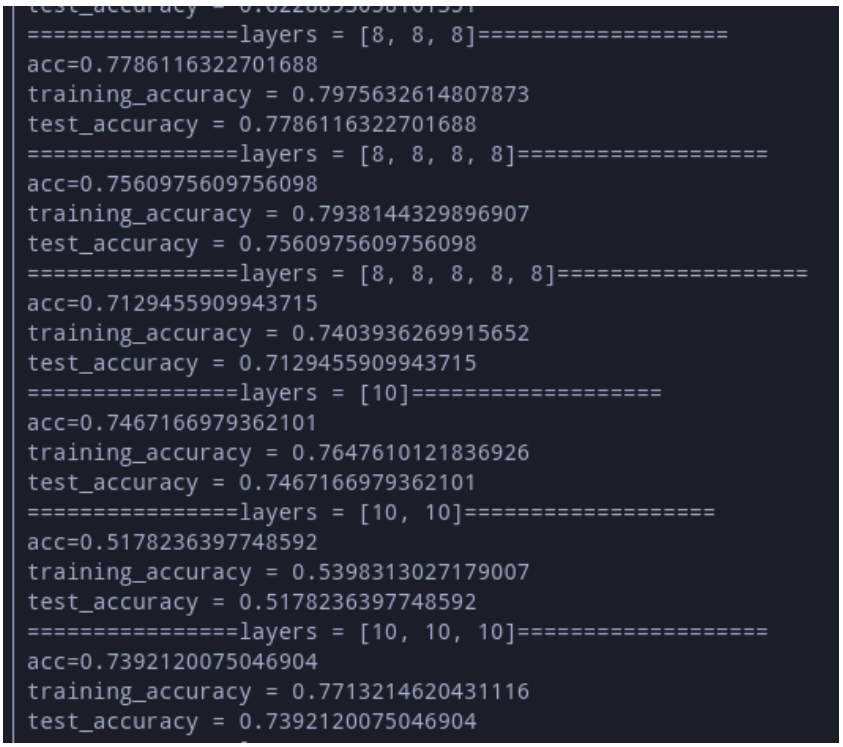

Fig. 16. Fragmento del output del experimento de capas ocultas para el fold 2. Output completo en la libreta correspondiente.

De nueva cuenta, no es claro cuál es el número óptimo de capaz ocultas. Por un lado, con una sola capa de 10 neuronas se obtiene la mejor relación entre el acc sobre el test y sobre el training pero por el otro quizás una sola neurona no nos de tanta expresividad como topologías con más capas. También están las topologías [10,10,10,10] y [10,10,10,10,10] que si bien son muy complejas, los accuracies de 0.7842 y 0.7899 parecen intereseantes.

Para descartar dudas a través de experimentación, se tomaron las mejores topologías con 8 ([8,8,8]) y 10 ([10,10,10, 10]) neuronas para el experimento de épocas. Se descartó la topología con 5 capas de 10 neuronas debido a su alto costo computacional.

Número de épocas

Para este fold se realizaron dos experimentos. El primero con un rango epochs = range(10, 351, 5) y el segundo con un rango epochs = range(100, 551, 10).

El primero de estos sirvió para descartar la topología [10,10,10, 10] ya que antes presenta un error mayor que la otra topología en el mismo número de épocas, además de ser la topología más compleja de las dos.

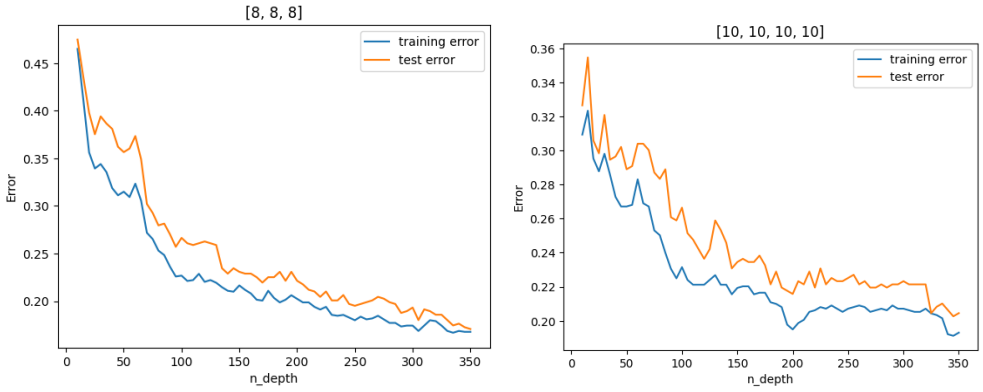

Fig. 17. Gráficas de error para las topologías [8,8,8] y [10,10,10,10] en el rango de 10 a 350 épocas.

El segundo experimento sirvió para identificar en qué punto la topología [8,8,8] entra en overfitting, del cual en conjunto con el output en la libreta se pudo identificar que a partir de aproximadamente las 350 épocas el modelo no mejora y entra en overfitting poco después.

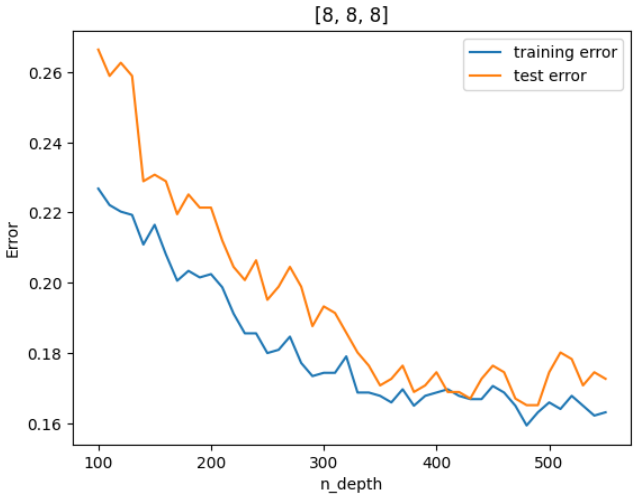

Fig. 18. Gráfica de error para la topología [8,8,8] en el rango de 100 a 550 épocas.

Resultado del entrenamiento

Finalmente, estos son los metaparámetros óptimos para este fold:

- Solver = 'lbfgs'
- Número de neuronas = 8
- hidden_layer_sizes (topología) = [8,8,8]
- Épocas = 350
- Learning Rate = N/A
- Momentum = N/A

Para concluir, se corrió una vez más el clasificador con los metaparámetros anteriores y se obtuvieron estos resultados:

Accuracy = 0.82926

	precision	recall	f1-score	support
tested_positive	0.85	0.80	0.82	267
tested_negative	0.81	0.86	0.83	266
accuracy			0.83	533
macro avg	0.83	0.83	0.83	533
weighted avg	0.83	0.83	0.83	533

Fold 3

El código de este fold se encuentra en la libreta entrenamiento_fold_3.ipynb.

Número de neuronas

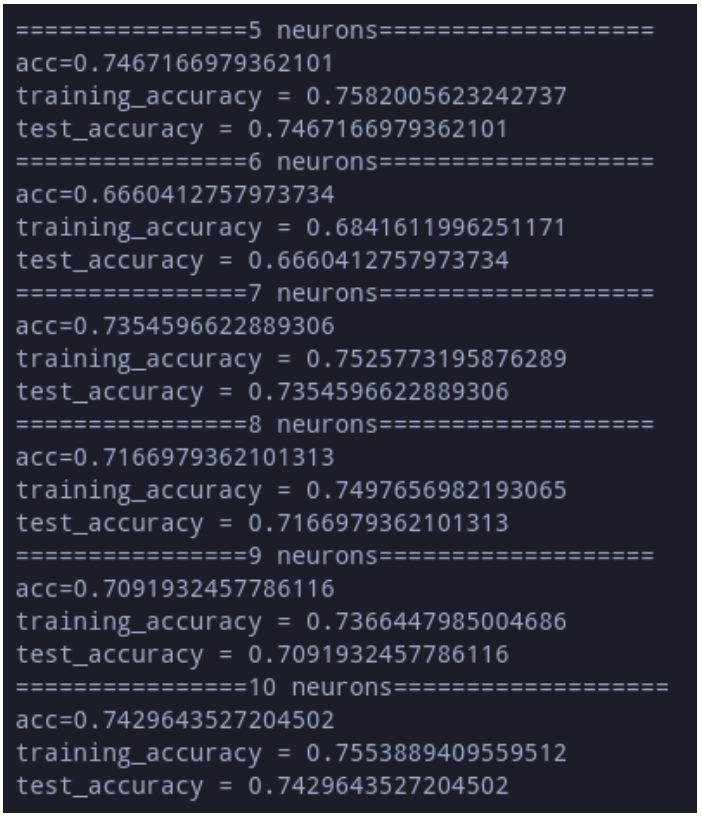

Fig. 19. Output generado por el experimento sore el número de neuronas en el fold 3.

Para este fold, se observó un comportamiento parecido al fold 2. Si bien las 8 neuronas sería la opción lógica a escoger basándonos en lo que hemos aprendido de nuestros datos con los experimentos sobre los folds anteriores, el accuracy de esta

cuenta de neuronas es menor al de 5, 7 y 10 neuronas. Por esto, y habiendo visto el comportamiento de los datos en el fold anterior, se tomó la decisión de realizar el experimento correspondiente a las capas ocultas utilizando 5, 8 y 10 neuronas en cada capa. Se descartó la cuenta de 7 neuronas ya que basándonos en el comportamiento observado hasta este momento la diferencia con los resultados al probar con 8 neuronas es considerable.

Número de capas ocultas

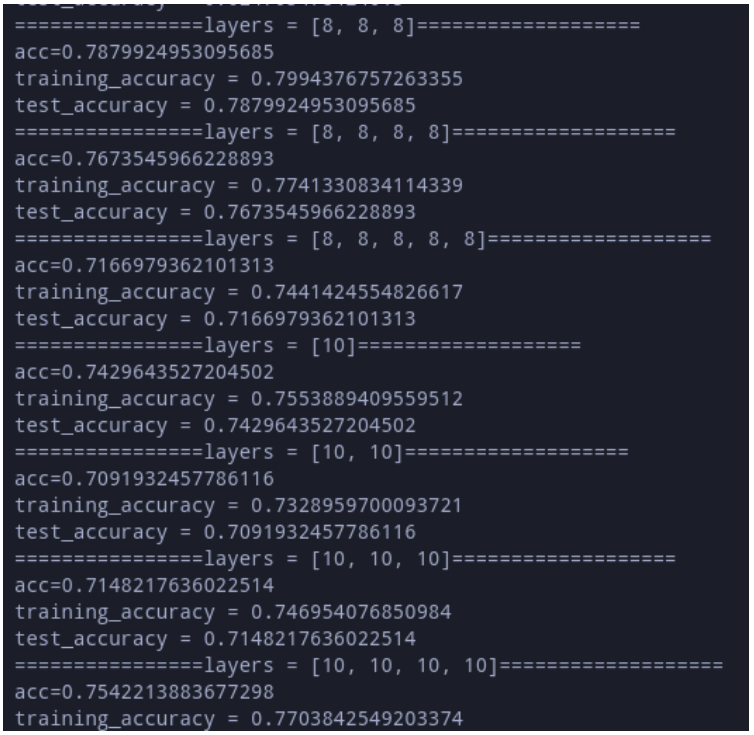

Fig. 20. Fragmento del output del experimento de capas ocultas para el fold 3. Output completo en la libreta correspondiente.

Al igual como sucedió en el fold 2, con este experimento ya se nota la superioridad que tienen las topologías con 8 neuoranas sobre las otras configuraciones. De nueva cuenta, el mejor accuracy fue logrado con una configuración [8,8,8] y de nueva cuenta el desempeño fue claramente superior al del resto de modelos en términos de error, cercanía de las medidas de error y compactez de la topología.

Por esto, se escogió la topología [8,8,8] como la única topología sobre la que se realizaria el siguiente experimento.

Número de épocas

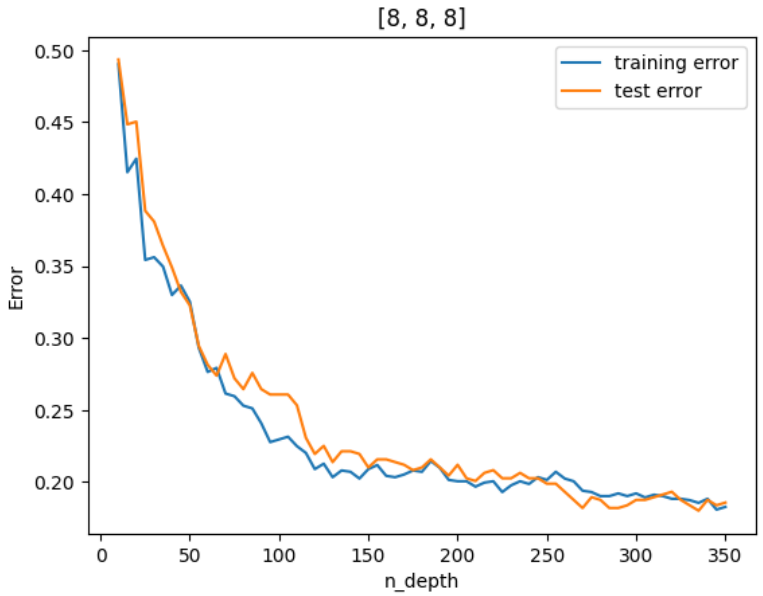

Fig. 21. Gráfica de error para la topología [8,8,8] en el rango de 10 a 350 épocas.

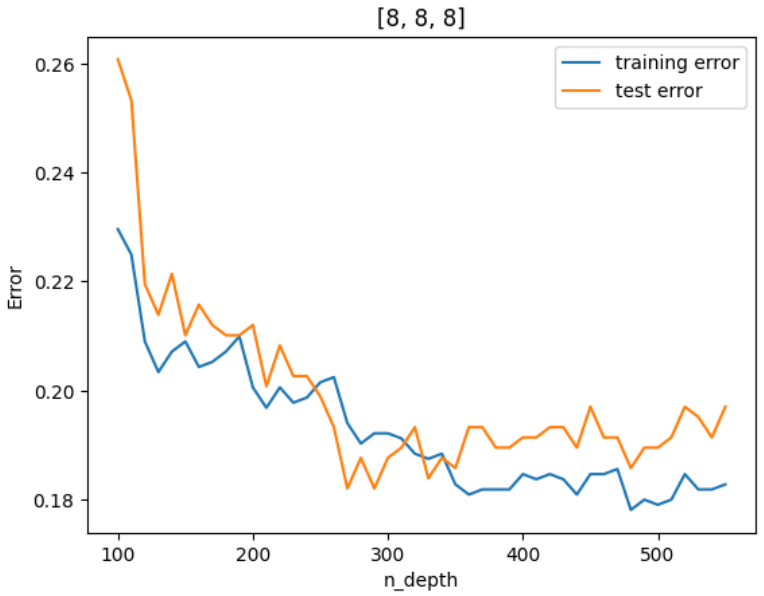

Fig. 22. Gráfica de error para la topología [8,8,8] en el rango de 100 a 550 épocas.

Podemos observar que para esta topología el error se reduce sin overfitting hasta aproximadamente las 350 épocas, punto donde el error del sobre el conjunto tres aumenta mientras que el error sobre el conjunto training continúa reduciéndose.

Por ello, el metaparámetro de épocas se fija en 350 épocas.

Resultado del entrenamiento

Finalmente, estos son los metaparámetros óptimos para este fold:

- Solver = 'lbfgs'
- Número de neuronas = 8
- hidden_layer_sizes (topología) = [8,8,8]
- Épocas = 350
- Learning Rate = N/A
- Momentum = N/A

Para concluir, se corrió una vez más el clasificador con los metaparámetros anteriores y se obtuvieron estos resultados:

Accuracy = 0.81425

	precision	recall	f1-score	support
tested_positive	0.82	0.81	0.81	266
tested_negative	0.81	0.82	0.82	267
accuracy			0.81	533
macro avg	0.81	0.81	0.81	533
weighted avg	0.81	0.81	0.81	533

Análisis de los resultados

En cada uno de los folds, el mejor modelo fue aquel que contiene tres capas de ocho neuronas, utilizando el solver lbfgs y con un número de épocas de entre 250 y 350.

El promedio del accuracy medido a lo largo de los tres folds es de 0.81687. La desviación estándar, calculada con la fórmula para población, es de σ: 0.0092.

Al analizar las tablas de resultados de cada uno de los folds, podemos observar que los falsos negativos son menos comunes que los falsos positivos, comportamiento que sería deseable al ser la diabetes una enfermedad que de no ser detectada a tiempo puede comprometer la salud o incluso la vida de los pacientes.

Con esto, podemos decir que, al menos con tres folds, se puede crear un modelo que prediga con precisión si un paciente va a dar positivo a diabetes usando los ocho atributos que vienen en la base de datos.

Aún hay pequeño margen de mejora, por lo que quizás un futuro experimento con un número mayor de folds y con más instancias por clase pueda servir para construir un mejor modelo.

5. (25 pts) Realizar un experimento final en el cual se entrene una RNA (con búsqueda de hyper - parámetros) sobre el conjunto training y test global, cuidando que los resultados no queden en underfitting ni en overfitting.

   Hacer un análisis comparativo con respecto a los resultados obtenidos en los pliegues, de tal forma que se discuta que tan robustos son los resultados obtenidos.

   El código de este fold se encuentra en la libreta entrenamiento_global.ipynb.

   Para este experimento, se utilizaron los archivos diabetes_balanced_test.csv y diabetes_balanced_train.csv.

   Número de neuronas

   Para este experimento, se cambió el rango de las neuronas presentes a [5,12] con la intención de descubrir si al tener una cantidad mayor de datos con la que trabajar es necesaria más expresividad en la red para crear un modelo preciso.

```
================6 neurons==================
acc=0.5525
training_accuracy = 0.55375
test_accuracy = 0.5525
================7 neurons==================
acc=0.715
training_accuracy = 0.706875
test_accuracy = 0.715
================8 neurons==================
acc=0.7575
training_accuracy = 0.7375
test_accuracy = 0.7575
================9 neurons==================
acc=0.725
training_accuracy = 0.72875
test_accuracy = 0.725
================10 neurons==================
acc=0.7625
training_accuracy = 0.75375
test_accuracy = 0.7625
================11 neurons==================
acc=0.745
training_accuracy = 0.74875
test_accuracy = 0.745
================12 neurons==================
acc=0.78
training_accuracy = 0.761875
test_accuracy = 0.78
```

Fig. 23. Output parcial generado por el experimento sobre el número de neuronas en los conjuntos globales. Output completo en la libreta correspondiente.

Al correr esta celda, de nueva cuenta no es claro cuál podría ser el número óptimo de neuronas. Sin embargo, sí es claro que los resultados con 6, 7, 9 y 11 ya que el resto de configuraciones presentan un accuracy mayor.

En el experimento siguiente, se probarán topologías construidas con 5, 8, 10 y 12 neuronas en las capas ocultas.

Número de capas ocultas

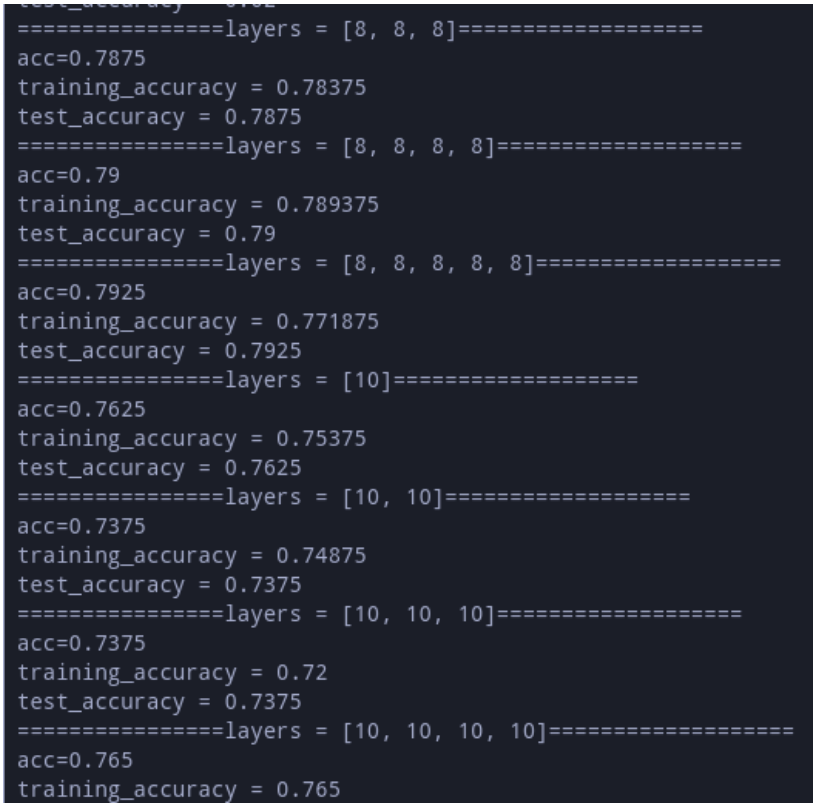

Fig. 24. Output parcial generado por el experimento sobre el número de capas ocultas en los conjuntos globales. Output completo en la libreta correspondiente.

Al análizar cada uno de los resultados para las distintas topologías, nuevamente no queda claro cuál es la indicada. Sin embargo, se pueden descartar las topologías de 5 capas ocultas, porque a pesar de tener un buen accuracy hay otras topologías menos complejas que alcanzan resultados similares.

Por otro lado, se descartan todas las topologías que tengan un accuracy menor a 0.77 y la topología [12,12,12,12] ya que hay varias topologías que cumplen con este sí superan este threshold y son estas las que se usarán en el siguiente experimento.

Entonces, nos quedamos con las topologías:

- [5]
- [5,5,5,5]
- [8,8,8]
- [8,8,8,8]
- [12]
- [12,12]

Número de épocas

Para determinar este metaparámetro, se realizaron dos experimentos. El primero de ellos se hizo sobre las topologías previamente enlistadas en un rango de [10, 550] en pasos de 10.

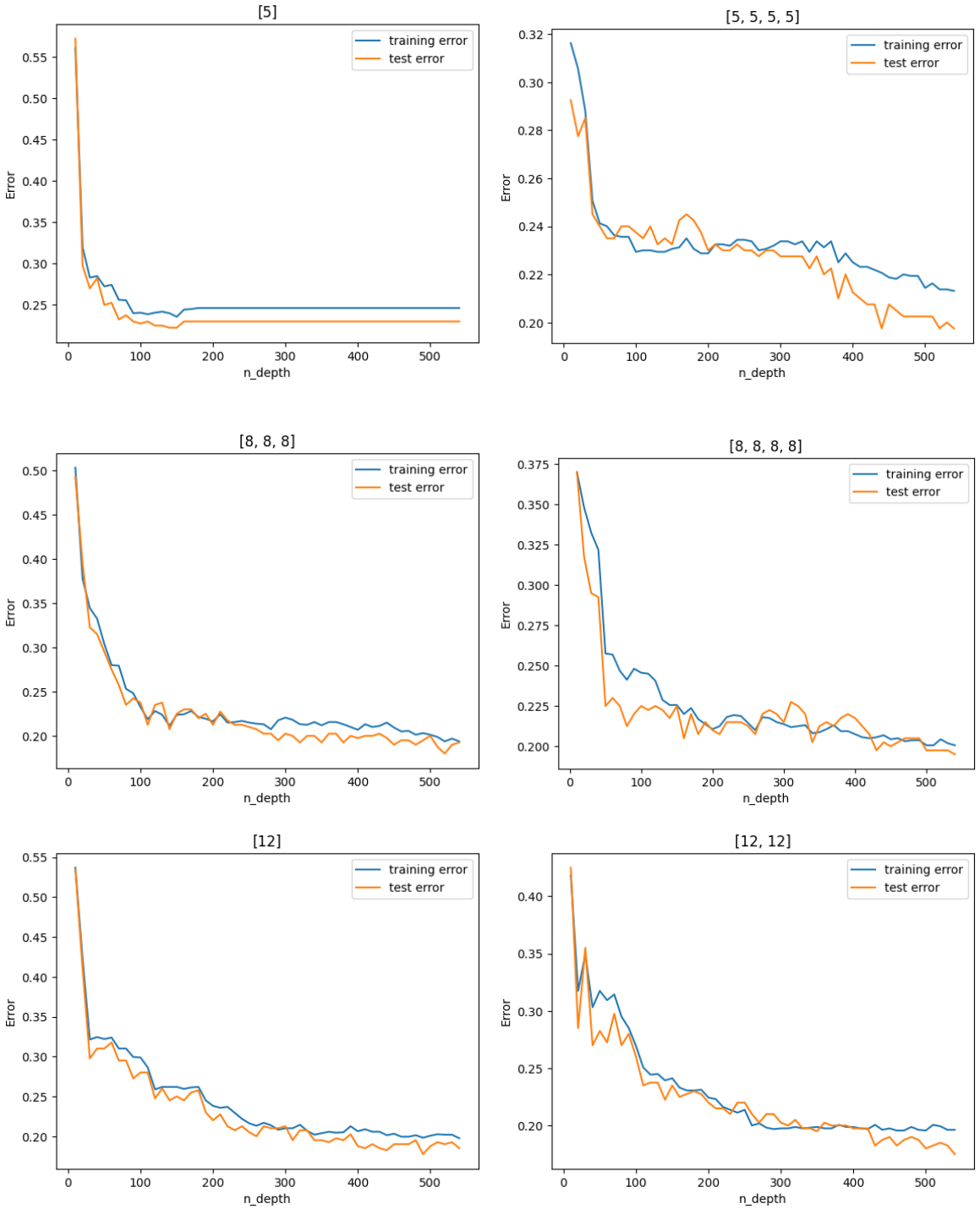

Fig. 25. Gráficas de los errores de las topologías de interés en el rango de 10 a 550 épocas.

Como podemos ver, en prácticamente todas las gráficas es complicado decir el punto de overfitting ya que varias presentan la característica de que el error sobre el conjunto training es superior al error sobre el conjunto test.

Tras revisar detenidamente las gráficas y el output de la celda correspondiente, se descartaron las topologías [5,5,5,5], [8,8,8,8], [12,12] por su error relativamente alto si se le compara con otras topologías que llegaron a un mejor error en menos épocas.

Este experimento no fue suficiente para determinar cuál es la topología más adecuada, por lo que se hizo uno más en el rango [100, 650] con las topologías que no fueron descartadas. El aumento en el número máximo de épocas se hizo ya que en varias gráficas parece que el punto de overfitting está un poco más allá de las 550 épocas.

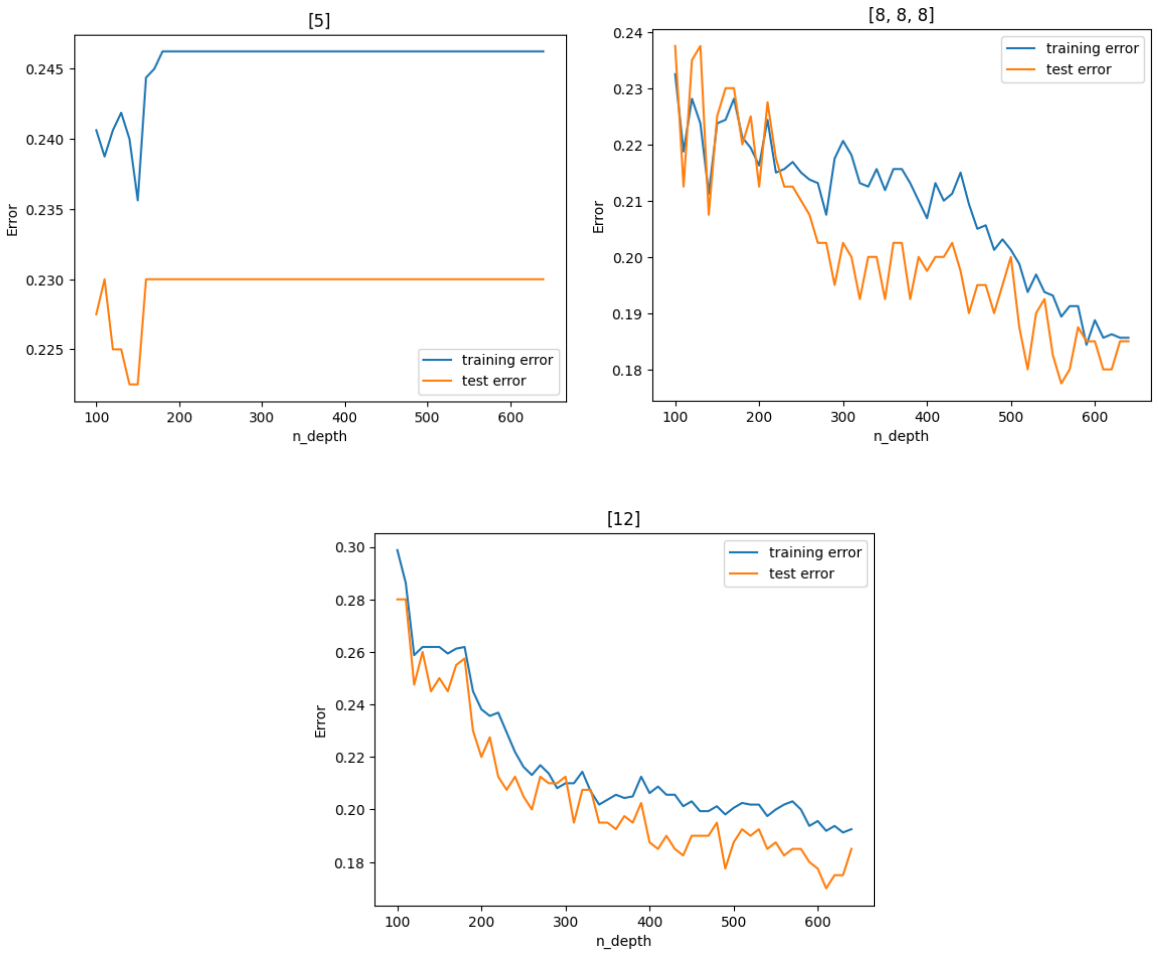

Fig. 26. Gráficas de los errores de las topologías de interés en el rango de 100 a 650 épocas.

De la segunda ronda, se descarta la topología [5] por tener el accuracy más bajo y estancarse rapidamente en su aprendizaje. Se descartaró también la topología [12] por requerir muchas épocas para llegar a un error que otras topologías llegan más rápido, conforme lo observado en el output de la celda correspondiente.

Esto nos deja, una vez más, con las topología [8,8,8] con un número de épocas igual a 550.

Learning rate y momentum

Como se mencionó en la sección del fold 1, utilizando una configuración que use el solver sgd para entrenar el modelo y que pueda así aceptar variaciones en el learning rate, se obtenían resultados peores que aquellos obtenidos con lbfgs. No obstante, por complitud, se corrió el mismo test descrito en el fold 1 ahora sobre los conjuntos de datos globales.

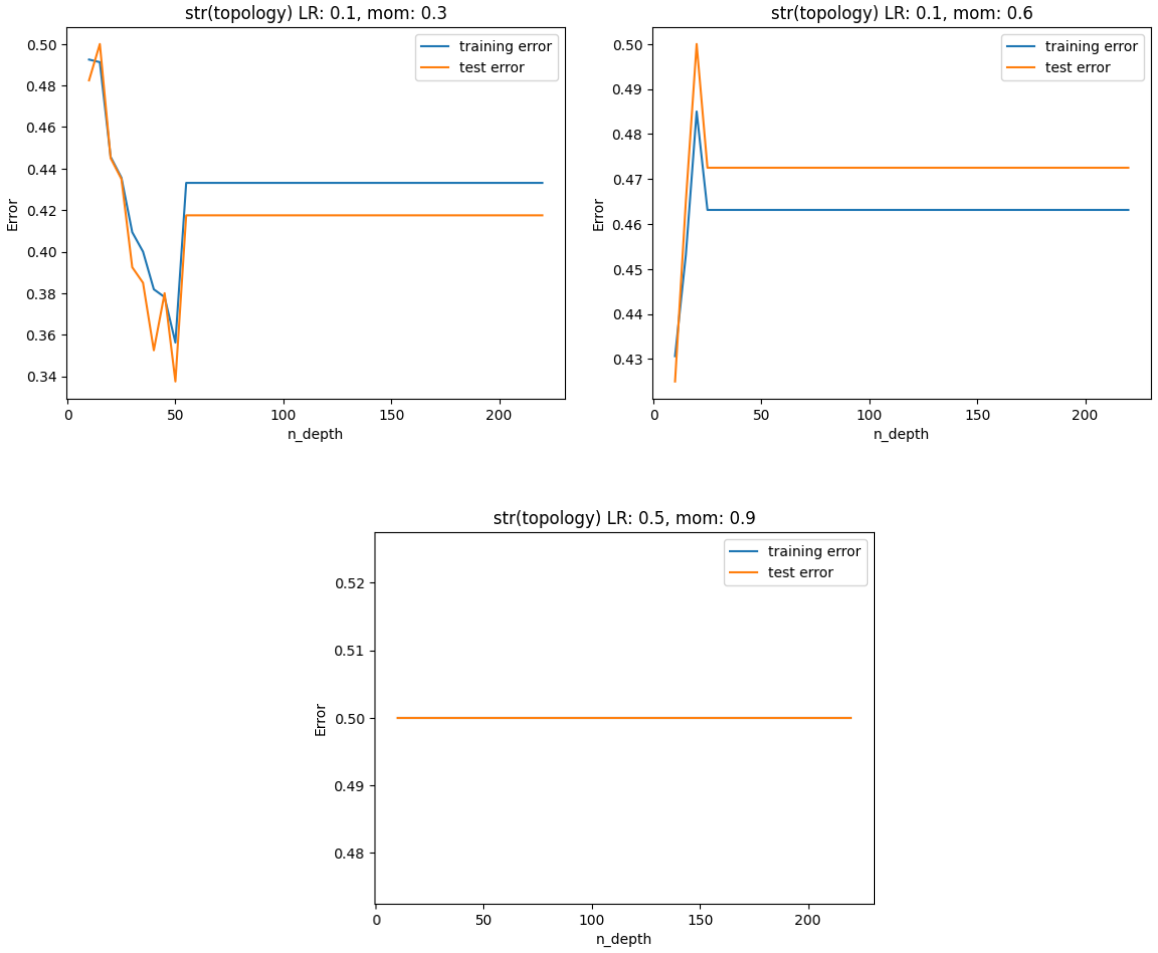

Fig. 27. Tres de las gráficas de los errores de la topología [8,8,8] en el rango de 100 a 200 épocas, variando el learnig rate y el momentum.

Como era de esperar, el error medido con respecto a ambos conjuntos es peor que cuando se utiliza el solver 'lbfgs'. Por tanto, ignoramos por completo estos resultados.

Resultado del entrenamiento

Finalmente, estos son los metaparámetros óptimos para este fold:

- Solver = 'lbfgs'

- Número de neuronas = 8
- hidden_layer_sizes (topología) = [8,8,8]
- Épocas = 550
- Learning Rate = N/A
- Momentum = N/A

Para concluir, se corrió una vez más el clasificador con los metaparámetros anteriores y se obtuvieron estos resultados:

Accuracy = 0.81425

	precision	recall	f1-score	support
tested_positive	0.84	0.78	0.81	200
tested_negative	0.80	0.85	0.82	200
accuracy			0.82	400
macro avg	0.82	0.82	0.82	400
weighted avg	0.82	0.82	0.82	400

Análisis de los resultados

En los resultados para el entrenamiento utilizando los conjuntos globales, vemos que se sigue la tendencia de tener menos falsos negativos que falsos positivos, lo que es ideal para el dominio de nuestros datos.

Si tomamos en cuenta el resultado de este experimento como parte de la población de experimentos, calculando el promedio de accuracy obtenemos un valor de 0.8162 con una desviación estándar de $\sigma = 0.008$.

El accuracy entra dentro del rango que vimos para los folds, pero en esta ocasión nos tomó 200 épocas más que las épocas que necesitaron los folds con los valores para las épocas más grandes. Esto puede ser debido a que teniendo un conjunto de datos más grande llegar a una convergencia es más difícil para el modelo ya que los patrones de los datos podrían parecer más complejos o dispersos.

De hecho, esta dificultad para encontrar un buen modelo para el conjunto global se vio reflejado en la cantidad de topologías y en los distintos números de neuronas que se tuvieron que probar para crear este modelo, siendo que para los folds se ocuparon una o un par de topologías y se probaron al rededor de uno a tres números distintos de neuronas, lo que contrasta con las seis topologías y cinco números de neuronas que se probaron en este último experimento.

No obstante, el hecho de que para todos los experimentos se llegara a la misma topología puede significar que los patrones presentes en estos datos, si bien complejos, están bien definidos y tanto el proceso de estratificación como el de sobremuestreo preservaron las características de los datos.

Concluimos que se encontró una topología capaz de predecir con, en promedio, un 81.42% de precisión si un paciente es positivo para diabetes con base en los ocho atributos de la base datos.