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.

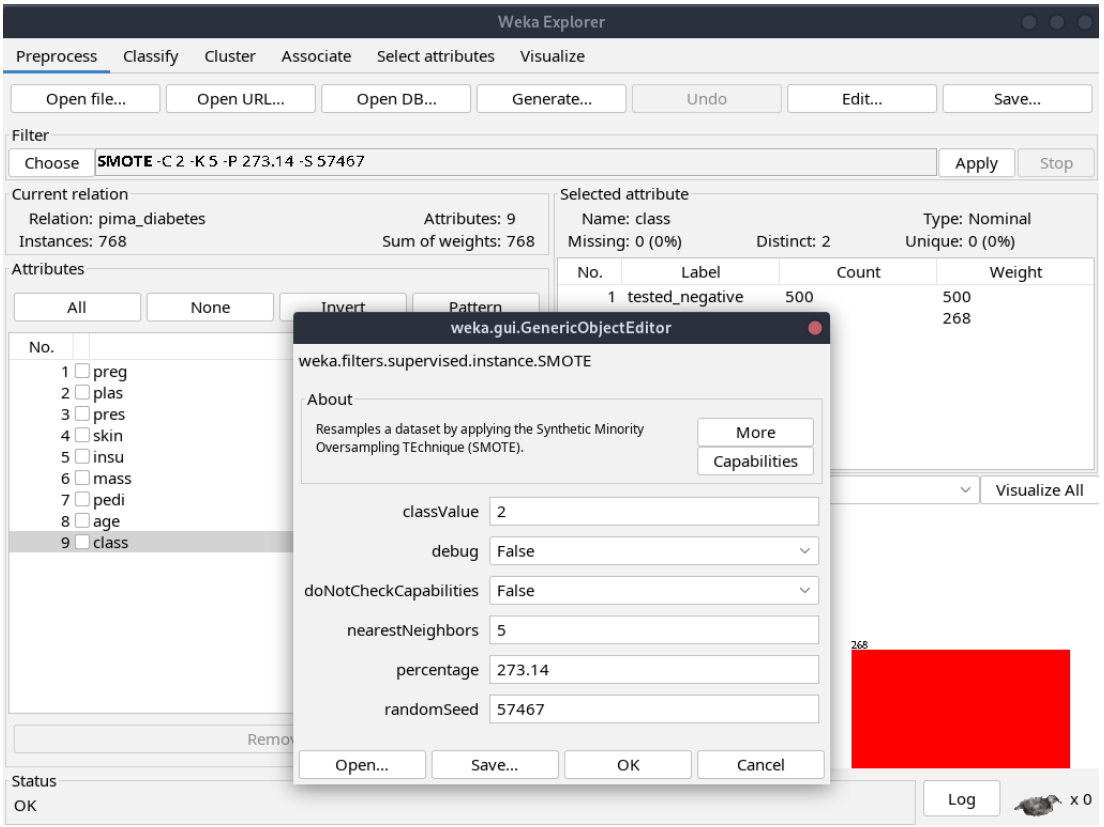


Fig. 1. Parámetros con los que se aplicó el filtro SMOTE sobre la clase tested_positive

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.

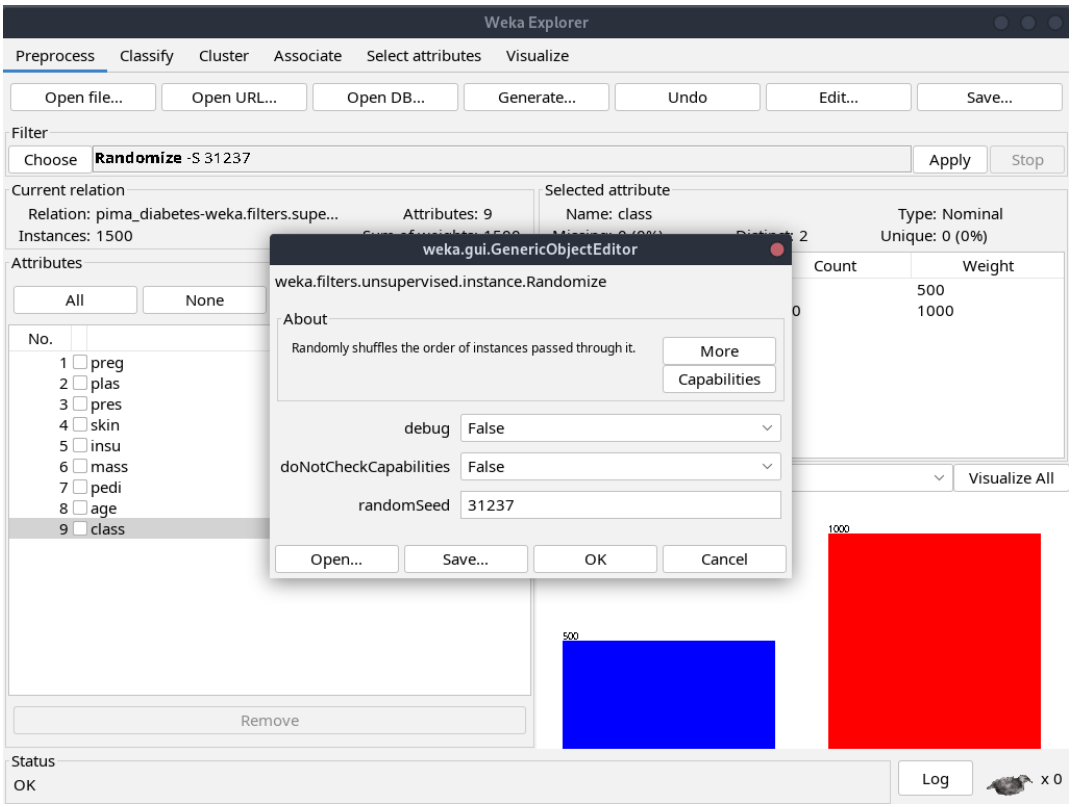


Fig. 2. Parámetros con los que se aplicó el filtro Randomize sobre todos los datos

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.

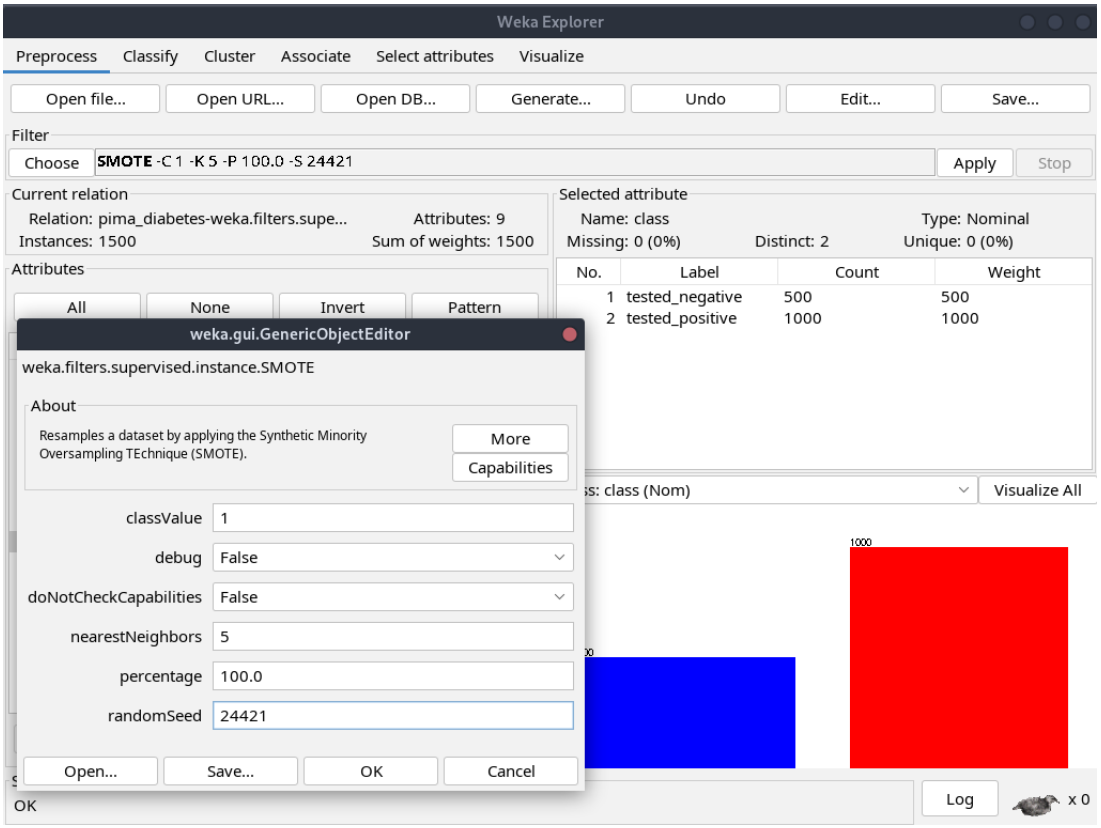


Fig. 3. Parámetros con los que se aplicó el filtro SMOTE sobre la clase tested_negative

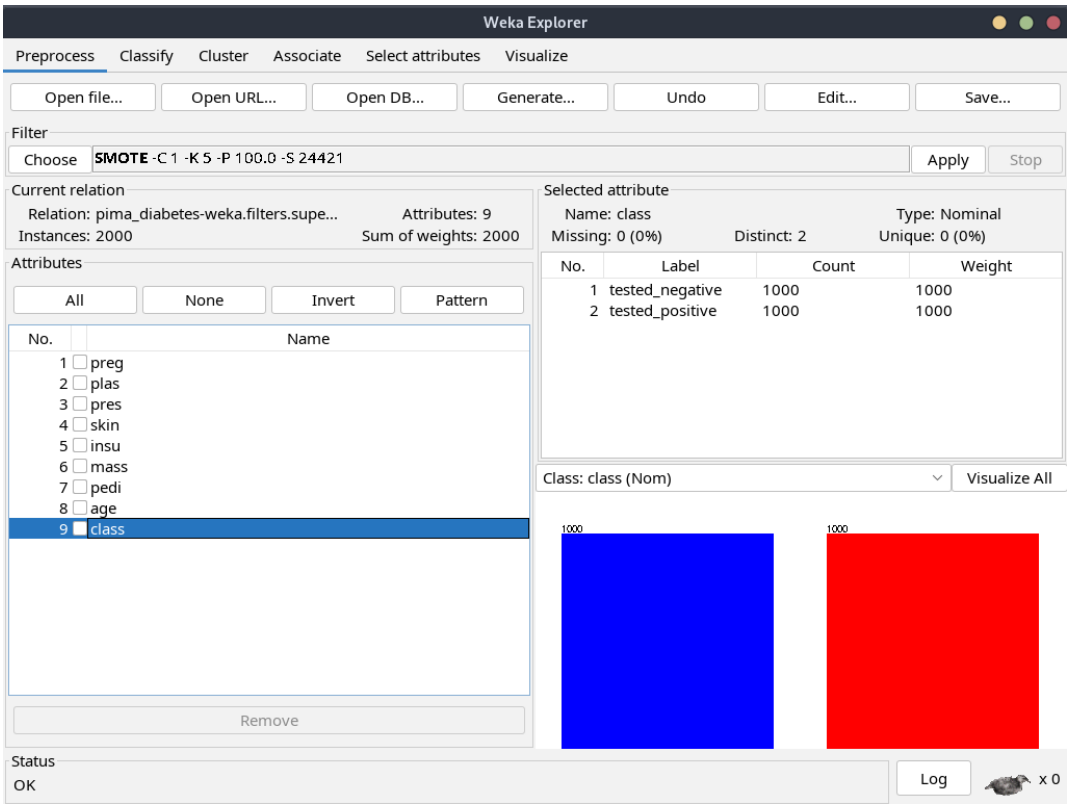


Fig. 4. Captura que muestra las clases balanceadas a 1000 instancias

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.

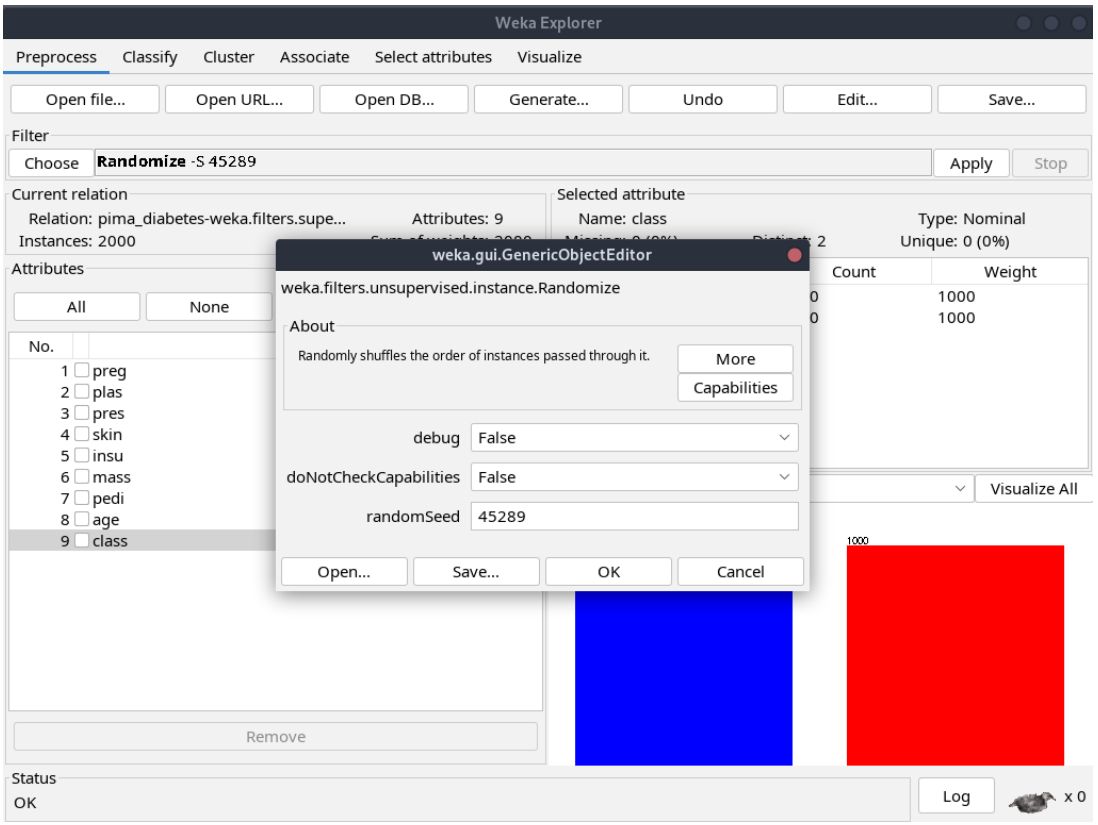


Fig. 5. Parámetros con los que se aplicó el filtro Randomize sobre todos los datos

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

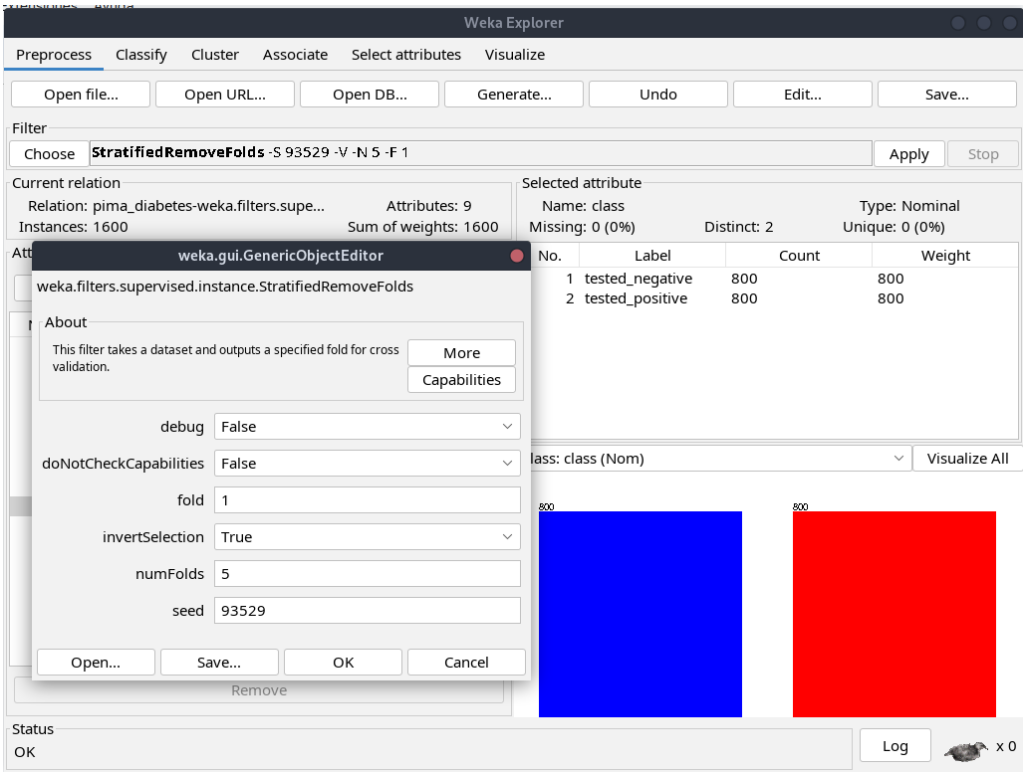


Fig. 6. Parámetros con los que se aplicó el filtro StratifiedRemoveFolds sobre todos los datos para obtener el conjunto training global.

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.

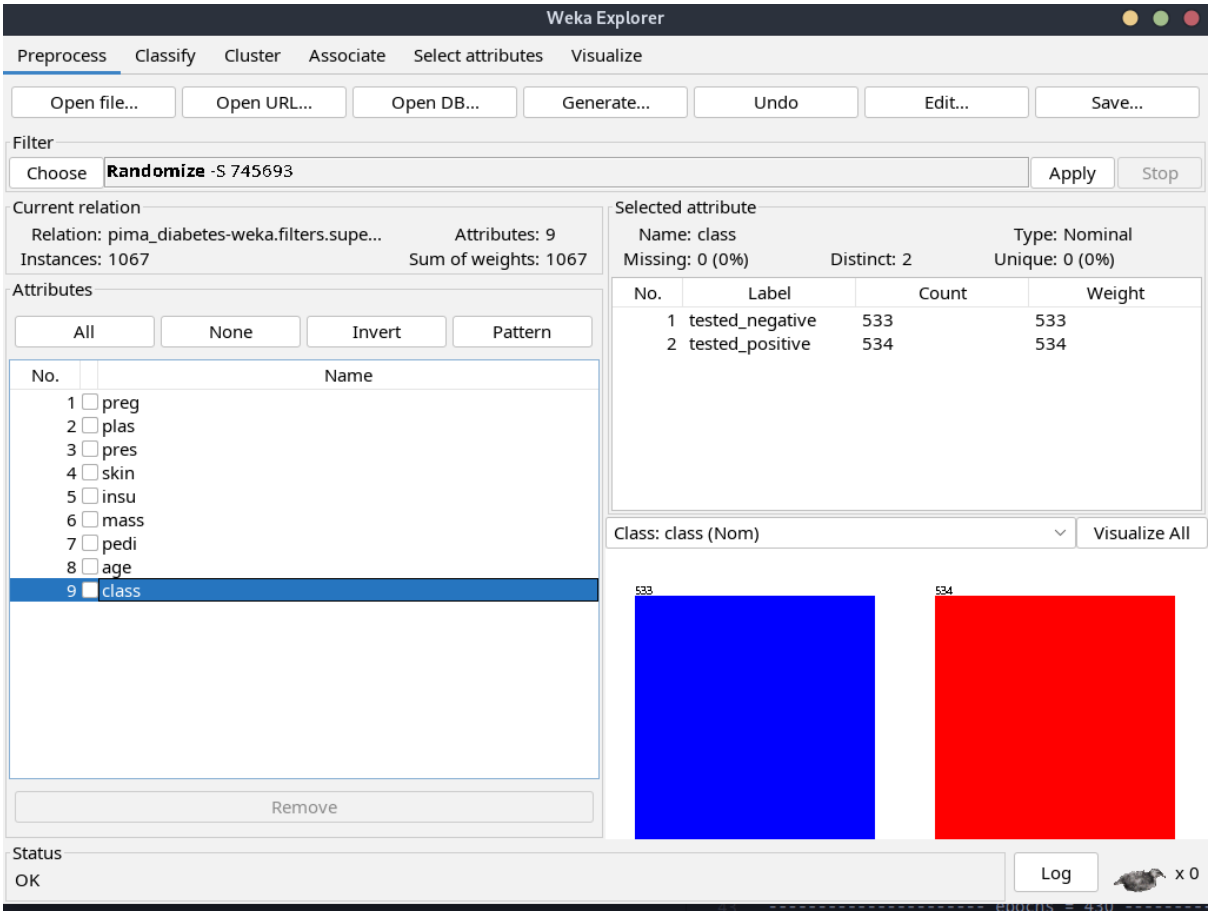


Fig. 6. Estado de los datos para el fold 3, en el conjunto training. Nótse que el último filtro en aplicarse antes de guardar el archivo fue Randomize

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

### *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.

```
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:

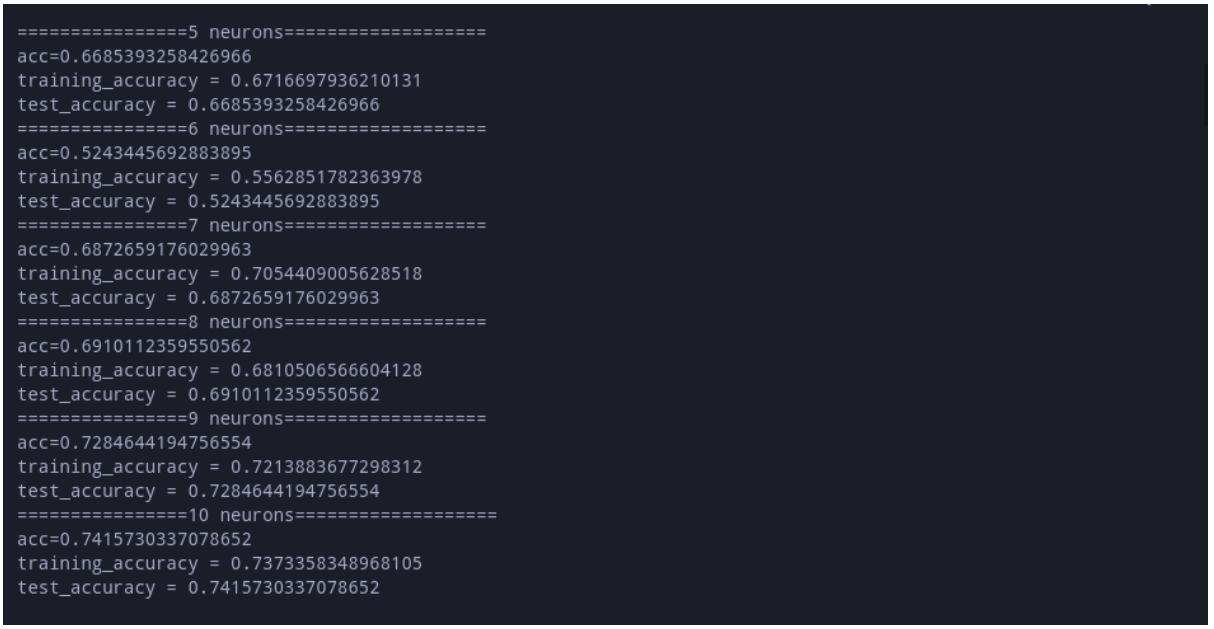


Fig. 8. Output generado por el experimento sore el número de neuronas.

De estos resultados, nos podemos percatar que con 9 neuronas se obtiene el mejor resultado puesto a que nos da el mejor accuracy sin que haya una disparidad muy grande entre el accuracy con respecto al test y el acc con respecto al training. No obstante, con 10 neuronasobtenemos un mejor accuracy e incluso el accuracy sobre el conjunto test es mejor que el de sobre el conjunto training. Por esto, para experimentos subsecuentes se utilizarán 9 y 10como el número de neuronas fijo para los siguientes experimentos.

***Número de capas ocultas***

***Número de épocas***

**Learning rate y momentum**

Utilizando la clase MLPCLassifier

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.

6. 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.