Solve a Classification Problem With Keras

In this tutorial we are going to use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years. It is a binary classification problem (onset of diabetes as 1 or not as 0). The input variables that describe each patient are numerical and have varying scales. Below lists the eight features for the dataset + the target:

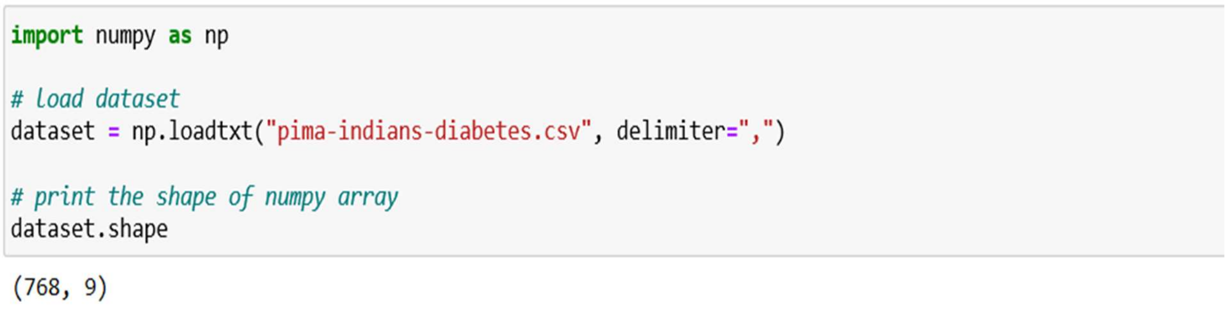
1. Number of times pregnant.
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skin fold thickness (mm).
5. 2-Hour serum insulin (mu U/ml).
6. Body mass index.
7. Diabetes pedigree function.
8. Age (years).
9. Class, onset of diabetes within five years.

Given that all attributes are numerical makes it easy to use directly with neural networks that expect numerical inputs and output values, and ideal for Keras. Below is a sample of the dataset showing the first 5 rows of the 768 instances:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

The baseline accuracy if all predictions are made as *no onset of diabetes* is 65.1%. A neural network should perform better than this baseline accuracy.

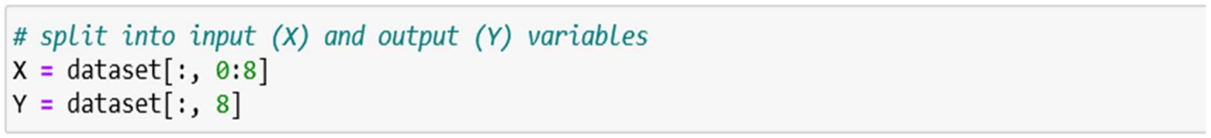
Now we can load our Pima Indians dataset directly using the NumPy function loadtxt(). There are eight input variables and one output variable (the last column).



```python
import numpy as np

# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")

# print the shape of numpy array
dataset.shape
```

```
(768, 9)
```

Once loaded we can split the dataset into input variables X and the output variable Y.



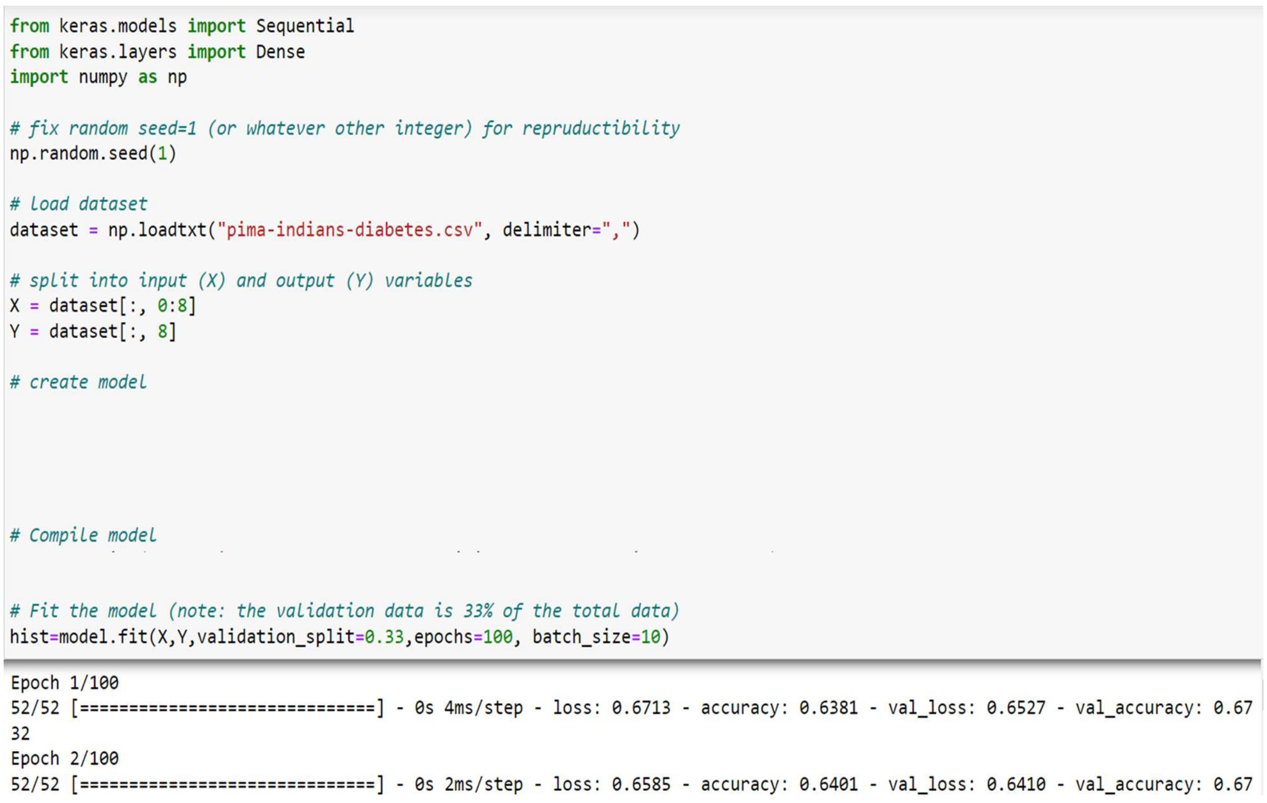
```
# split into input (X) and output (Y) variables
X = dataset[:, 0:8]
Y = dataset[:, 8]
```

1. Split training and validation data using Keras

Keras can separate a portion of your training data into a validation dataset and evaluate the performance of your model on that validation dataset each epoch. You can do this by setting the validation split argument on the fit() function to a percentage of the size of your training dataset. For example, a reasonable value might be 0.2 or 0.33 for 20% or 33% of your training data held back for validation. The example below demonstrates the use of using an automatic validation dataset on the Pima Indians onset of diabetes dataset. The output of the network is a scalar belonging to [0,1] since it should represents the probability to have diabetes within five years. To make sure the model outputs is a scalar, the number of neurons of the output layer must be equal to one. To ensure this number is between 0 and 1, we use a sigmoid function as output activation function. Lastly, the loss function that is minimized to find the parameters of the network is the "binary cross entropy" since we only have two classes (diabetes or no diabetes). As a reminder, the binary cross entropy takes the form:

$$C = \frac{-1}{N}\sum_{i=1}^{n} y_i\, ln(\hat{y}_i) + (1 - y_i) ln(1 - \hat{y}_i)$$

where *N* is the number of samples, $\hat{y}_i$ is the output of the neural network for an input x_i and y_i is the exact output value. Here is the code that you need to complete:



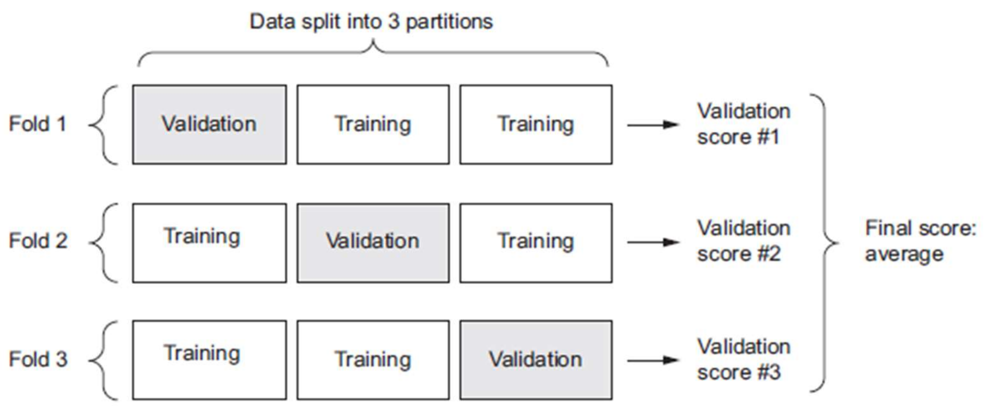
```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# fix random seed=1 (or whatever other integer) for repruductibility
np.random.seed(1)

# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")

# split into input (X) and output (Y) variables
X = dataset[:, 0:8]
Y = dataset[:, 8]

# create model





# Compile model


# Fit the model (note: the validation data is 33% of the total data)
hist=model.fit(X,Y,validation_split=0.33,epochs=100, batch_size=10)
```

```
Epoch 1/100
52/52 [==============================] - 0s 4ms/step - loss: 0.6713 - accuracy: 0.6381 - val_loss: 0.6527 - val_accuracy: 0.67
32
Epoch 2/100
52/52 [==============================] - 0s 2ms/step - loss: 0.6585 - accuracy: 0.6401 - val_loss: 0.6410 - val_accuracy: 0.67
```

This network achieves an accuracy of about 72%, which is not a great achievement over the baseline (65.1%). Due to the small size of the dataset the validation set is also small. As a consequence, the validation score might change a lot depending on which data points you chose for validation and which you chose for training. This would prevent you from reliably evaluating your model. The best practice in such situation is to use K-fold cross-validation as described in the next section.

2. K-Fold validation

The gold standard for machine learning model evaluation is k-fold cross validation. It provides a robust estimate of the performance of a model on unseen data. It does this by splitting the training dataset into k subsets and takes turns training models on all subsets except one which is held out, and evaluating model performance on the held out validation dataset. The process is repeated until all subsets are given an opportunity to be the held out validation set. The performance measure is then averaged across all models that are created. The Figure below shows an example of 3-fold cross-validation.



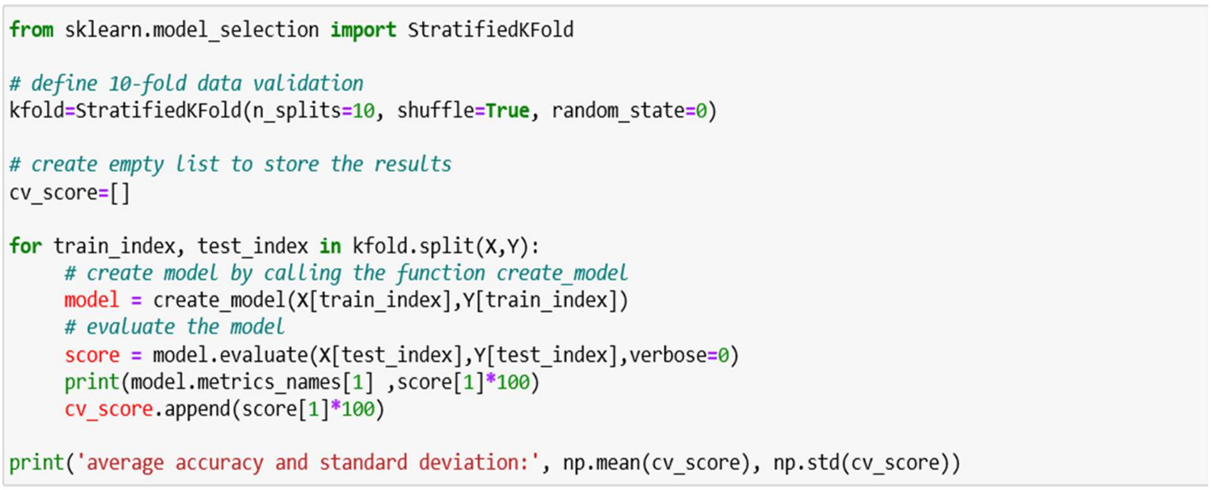
In practice, k-fold cross validation is often used with 5 or 10 folds. As such, 5 or 10 models must be constructed and evaluated, greatly adding to the evaluation time of a model. Nevertheless, when the problem is small enough or if you have sufficient compute resources, k-fold cross validation can give you a less biased estimate of the performance of your model.

3. Implementation of K-Fold validation

In the example below we use the handy StratifiedKFold class from the scikit-learn Python machine learning library to split up the training dataset into 10 folds. The folds are stratified, meaning that the algorithm attempts to balance the number of instances of each class in each fold. The example creates and evaluates 10 models using the 10 splits of the data and collects all of the scores. The verbose output for each epoch is turned of by passing verbose=0 to the fit() and evaluate() functions on the model. The performance is printed for each model and it is stored. The average and standard deviation of the model performance is then printed at the end of the run to provide a robust estimate of model accuracy. First, it is always a good idea to define a function that creates the model, given input/output (X,Y).

```python
def create_model(X,Y):


    # Your model here


    model.compile(loss='binary_crossentropy', optimizer='adam',metrics='accuracy')
    model.fit(X,Y,epochs=100, batch_size=10, verbose=0,validation_split=0)

    return model
```

Then, this function can be called with each data (X,Y) as argument. Note that *kfold.split(X,Y)* returns the indices of the training data and validation data to be selected in one of the partitions. You can now test the K-Fold validation (beforehand, do not forget to add the libraries, import the data and split input X and output Y variables).



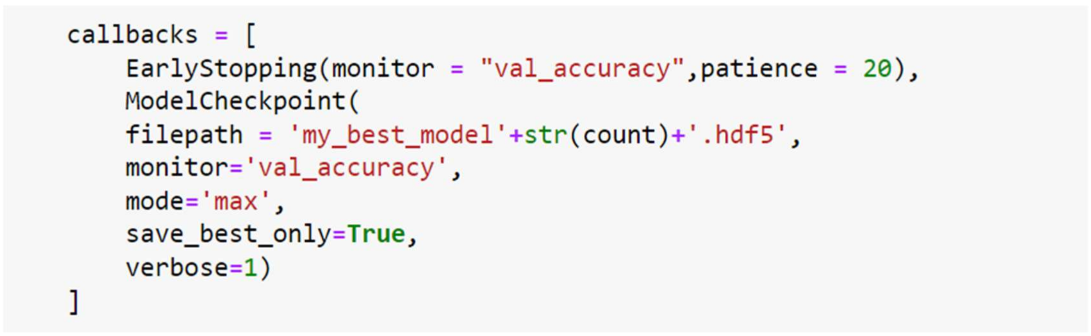
```
from sklearn.model_selection import StratifiedKFold

# define 10-fold data validation
kfold=StratifiedKFold(n_splits=10, shuffle=True, random_state=0)

# create empty list to store the results
cv_score=[]

for train_index, test_index in kfold.split(X,Y):
    # create model by calling the function create_model
    model = create_model(X[train_index],Y[train_index])
    # evaluate the model
    score = model.evaluate(X[test_index],Y[test_index],verbose=0)
    print(model.metrics_names[1] ,score[1]*100)
    cv_score.append(score[1]*100)

print('average accuracy and standard deviation:', np.mean(cv_score), np.std(cv_score))
```

4. Improve the Model

With K-Fold cross validation, we have a reliable way of measuring the performance of our model, however the model does not perform great. It is because we have made a major error when dealing with neural networks. **The input data has not been normalized**. To improve your model, you should implement two standard normalization methods: data standardization already seen before (so that the mean of input data is 0 and its standard deviation is 1) and rescaling of the data so that all values are within the range of 0 and 1. This last normalization can be performed using the following transformation:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Then, you should observe an increase in the performance of your model (around 80%).
To tune the number of epochs, you should define and use the *callback* capability of Keras. The callback specifies when you want to store your best model and it must be defined before training the model. The following callback is defined such as a best new model is stored each time the accuracy computed with the validation data improves. If it does not improve for 20 epochs, the training stops. The variable *count* specifies which model you are training (from 1 to 10) and is required to store the best model under the name '*my_best_model1.hdf5*', '*my_best_model2.hdf5*'…



```
callbacks = [
    EarlyStopping(monitor = "val_accuracy",patience = 20),
    ModelCheckpoint(
    filepath = 'my_best_model'+str(count)+'.hdf5',
    monitor='val_accuracy',
    mode='max',
    save_best_only=True,
    verbose=1)
]
```

Since a data validation set is required by the callback, the function *create_model* must be modified as follows (X_t, Y_t represent the training data and X_v, Y_v the validation data):

```
def create_model(X_t,Y_t,X_v,Y_v):


    # Your model here



    model.compile(loss='binary_crossentropy', optimizer='adam',metrics='accuracy')

    model.fit(X_t,Y_t,epochs=100,batch_size=10,verbose=0,validation_data=(X_v,Y_v),callbacks=callbacks )

    return model
```

Once your 10 best model are built and store on the hard disk, it's time to test the final model which can be the average of all models:

$$Final\_model = \frac{1}{10}\sum_{i=1}^{10} my\_best\_model\_i$$

The *Final_model* must be tested on data that has never been seen before by any models. This must be anticipated at the beginning of the code by splitting the data into two parts: the data for the kfold procedure (80%) and the data for the final testing (20%). For that, we use the *train_test_split* function of the *sklearn* library:

```
from sklearn.model_selection import train_test_split

X_kfold, X_test, y_kfold, y_test = train_test_split(X, Y, test_size=0.2, shuffle=True, random_state=3)
```

Then, the 10 best models can be retrieved from the hard disk and stored into a list *all_model*, and ready to be used:

```
from tensorflow.keras.models import load_model

all_model=[]

for i in range(n_folds):
    all_model.append(load_model('my_best_model'+str(i+1)+'.hdf5'))
```

Each model of the list is then asked to make a prediction for the test dataset and all the results are averaged. The averaged values are rounded to the nearest integer and compared against the true expected value to compute the accuracy.

```
model_predict_moyenne = all_model[0].predict(X_test)

for i in range(1,10):
    model_predict_moyenne = model_predict_moyenne + all_model[i].predict(X_test)

model_predict_moyenne = np.round(model_predict_moyenne/10)

# Accyracy of the final model:
print('accuracy:',1-(np.sum(np.abs(model_predict_moyenne[:,0]-y_test))/len(y_test)))
```

```
5/5 [==============================] - 0s 1ms/step
5/5 [==============================] - 0s 1ms/step
5/5 [==============================] - 0s 998us/step
5/5 [==============================] - 0s 752us/step
5/5 [==============================] - 0s 1ms/step
5/5 [==============================] - 0s 998us/step
5/5 [==============================] - 0s 998us/step
5/5 [==============================] - 0s 1ms/step
5/5 [==============================] - 0s 748us/step
5/5 [==============================] - 0s 750us/step
accuracy: 0.7467532467532467
```

Exercise :

For the final decision of the classification, the average of the 10 models is considered. Instead of that, we could have taken the decision based on a system by vote. In that case, to avoid the possibility to have 5 votes in one class and 5 votes in the other class, it is preferable to use an odd number of models (for example 9 models instead of 10).
Modify the code to implement a 9-Fold method coupled with a vote system. Compare the accuracy with the previous method.