Multi-Class Classification Iris Flowers Project 2 for Mothers who love Gardening and Flowers: Identifying Flower Types

Total Marks: 100 points
Project Supervisor: Mr. Inam Ul Haq, CTO Zaavia

In this project, you will discover how you can use Keras to develop and evaluate neural network models for multi-class classification projects.

After completing this step-by-step project, you will know:

- How to load data from CSV and make it available to Keras.
- How to prepare multi-class classification data for modeling with neural networks.
- How to evaluate Keras neural network models with scikit-learn.

Step 1. Project Description

In this project, we will use the standard machine-learning problem called the iris flowers dataset:

http://archive.ics.uci.edu/ml/datasets/Iris

This dataset is well studied and is a good problem for practicing on neural networks because all of the 4 input variables are numeric and have the same scale in centimeters. Each instance describes the properties of an observed flower measurements and the output variable is specific iris species.

This is a multi-class classification problem, meaning that there are more than two classes to be predicted, in fact there are three flower species. This is an important type of problem on which to practice with neural networks because the three class values require specialized handling.

The iris flower dataset is a well-studied problem and as such we can expect to achieve model accuracy in the range of 95% to 97%. This provides a good target to aim for when developing our models in this project.

We have download the iris flowers dataset for free and place it in the project directory with the filename "iris.csv". You can also directly download the dataset:
http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data

Step 2. Making Preparations
We will start off by importing all of the classes and functions we will need. This includes both the functionality we require from Keras, but also data loading from pandas as well as data preparation and model evaluation from scikit-learn:

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
```

Next, we need to initialize the random number generator to a constant value (7).

This is important to ensure that the results we achieve from this model can be achieved again precisely. It ensures that the stochastic process of training a neural network model can be reproduced:

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

The dataset can be loaded directly. Because the output variable contains strings, it is easiest to load the data using pandas. We can then split the attributes (columns) into input variables (X) and output variables (Y).

```
# load dataset
dataframe = pandas.read_csv("iris.csv", header=None)
dataset = dataframe.values
X = dataset[:,0:4].astype(float)
Y = dataset[:,4]
```

The output variable contains three different string values.

When modeling multi-class classification problems using neural networks, it is good practice to reshape the output attribute from a vector that contains values for each class value to be a matrix with a boolean for each class value and whether or not a given instance has that class value or not.

This is called one hot encoding or creating dummy variables from a categorical variable:
https://en.wikipedia.org/wiki/One-hot

For example, in this problem three class values are Iris-setosa, Iris-versicolor and Iris-virginica. If we had the observations:

Iris-setosa
Iris-versicolor
Iris-virginica

We can turn this into a one-hot encoded binary matrix for each data instance that would look as follows:

Iris-setosa,	*Iris-versicolor,*	*Iris-virginica*
1,	*0,*	*0*
0,	*1,*	*0*
0,	*0,*	*1*

We can do this by first encoding the strings consistently to integers using the scikit-learn class LabelEncoder. Then convert the vector of integers to a one hot encoding using the Keras function to_categorical().

```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```

Step 3: Define the Neural Network Baseline Model
The Keras library provides wrapper classes to allow you to use neural network models developed with Keras in scikit-learn.

There is a KerasClassifier class in Keras that can be used as an Estimator in scikit-learn, the base type of model in the library. The KerasClassifier takes the name of a function as an argument. This function must return the constructed neural network model, ready for training.

Below is a function that will create a baseline neural network for the iris classification problem. It creates a simple fully connected network with one hidden layer that contains 8 neurons.

The hidden layer uses a rectifier activation function, which is a good practice. Because we used a one-hot encoding for our iris dataset, the output layer must create 3 output values, one for each class. The output value with the largest value will be taken as the class predicted by the model.

The network topology of this simple one-layer neural network can be summarized as:

4 inputs -> [8 hidden nodes] -> 3 outputs

Note that we use a “softmax” activation function in the output layer. This is to ensure the output values are in the range of 0 and 1 and may be used as predicted probabilities.

Finally, the network uses the efficient Adam gradient descent optimization algorithm with a logarithmic loss function, which is called “categorical_crossentropy” in Keras.

```
# define baseline model
def baseline_model():
	# create model


	# Compile model

	return model
```

We can now create our KerasClassifier for use in scikit-learn.

We can also pass arguments in the construction of the KerasClassifier class that will be passed on to the fit() function internally used to train the neural network. Here, we pass the number of epochs as 200 and batch size as 5 to use when training the model. Debugging is also turned off when training by setting verbose to 0.

```
estimator = KerasClassifier(build_fn=baseline_model, epochs=200, batch_size=5, verbose=0)
```

Step 4. Evaluate The Model with k-Fold Cross Validation
We can now evaluate the neural network model on our training data.

The scikit-learn has excellent capability to evaluate models using a suite of techniques. The gold standard for evaluating machine learning models is k-fold cross validation.

First we can define the model evaluation procedure. Here, we set the number of folds to be 10 (an excellent default) and to shuffle the data before partitioning it.

```
kfold = KFold(n_splits=10, shuffle=True, random_state=seed)
```

Now we can evaluate our model (estimator) on our dataset (X and dummy_y) using a 10-fold cross-validation procedure (kfold).

Evaluating the model only takes approximately 10 seconds and returns an object that describes the evaluation of the 10 constructed models for each of the splits of the dataset.

```
results = cross_val_score(estimator, X, dummy_y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

The results are summarized as both the mean and standard deviation of the model accuracy on the dataset. This is a reasonable estimation of the performance of the model on unseen data. It is also within the realm of known top results for this problem.

Accuracy: 97.33% (4.42%)

We with these good results we plan to go further.

Step 5. Tuning Layers and Number of Neurons in The Model
There are many things to tune on a neural network, such as the weight initialization, activation functions, optimization procedure and so on.

One aspect that may have an outsized effect is the structure of the network itself called the network topology. In this section, we take a look at two experiments on the structure of the network: making it smaller and making it larger.

These are good experiments to perform when tuning a neural network on your problem.

Step 5.1. Evaluate a Smaller Network

Step 5.2. Evaluate a Larger Network
A neural network topology with more layers offers more opportunity for the network to extract key features and recombine them in useful nonlinear ways.

We can evaluate whether adding more layers to the network improves the performance easily by making another small tweak to the function used to create our model.

The idea here is that the network is given the opportunity to model all input variables before being bottlenecked and forced to halve the representational capacity, much like we did in the experiment above with the smaller network.

Instead of squeezing the representation of the inputs themselves, we add hidden layers to aid in the process.

Step 6. Really Scaling up: developing a model that overfits
Once you've obtained a model that has statistical power, the question becomes, is your model sufficiently powerful? Does it have enough layers and parameters to properly model the problem at hand?

Remember that the universal tension in machine learning is between optimization and generalization; the ideal model is one that stands right at the border between underfitting and overfitting; between undercapacity and overcapacity. To figure out where this border lies, first you must cross it.

To figure out how big a model you'll need, you must develop a model that overfits.
This is fairly easy:

1. Add layers.
2. Make the layers bigger.

3. Train for more epochs.

Always monitor the training loss and validation loss, as well as the training and validation values for any metrics you care about. When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.

The next step is to start regularizing and tuning the model, to get as close as possible to the ideal model that neither underfits nor overfits.

Step 7. Tuning the Model
With further tuning of aspects like the optimization algorithm etc. and the number of training epochs, it is expected that further improvements are possible. What is the best score that you can achieve on this dataset?

Step 8. Rewriting the code using the Keras Functional API
Review the April 9, 2018 presentation done by Chollet contained in the project file:

Francois_Chollet_March9.pdf

Now rewrite the code that you have written so far using the Keras Sequential API in Kearas Functional API.

Step 9. Rewriting the code by doing Model Subclassing
Now rewrite the code that you have written so far using the Keras Model Subclassing as mentioned in the Chollet April 9, 2018 presentation.

Reference:
https://www.tensorflow.org/api_docs/python/tf/keras/Model
Please note you will have to use TensorFlow 1.7+ with built-in Keras.

Step 10. Rewriting the code without using scikit-learn
Once you have written the model in all three API style you are required to do k-fold cross validation without using scikit-learn library.

Step: 11. Present your Model and appear in Viva
After you have completed your project come to Sir Syed University (Lab 2) on Sunday at 4:00 pm to present your project to the project supervisor Mr. Inam ul Haq, he will also conduct a Viva. Once review of the project is complete you will be awarded marks out of 100 points.

Note: You can also consult the project supervisor on any Sunday at 4:00 pm.

What we achieved doing this Project:
In this project you discovered how to develop and evaluate a neural network using the Keras Python library for deep learning.

By completing this project, you learned:

- How to load data and make it available to Keras.
- How to prepare multi-class classification data for modeling using one hot encoding.
- How to use Keras neural network models with scikit-learn.
- How to define a neural network using Keras for multi-class classification.
- How to evaluate a Keras neural network model using scikit-learn with k-fold cross validation