Regression Housing Pricing Project 3 for Fathers who want to buy a House:
Predicting Housing Prices

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

In this project you will discover how to develop and evaluate neural network models using Keras for a regression problem.

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

- How to load a CSV dataset and make it available to Keras.
- How to create a neural network model with Keras for a regression problem.
- How to use scikit-learn with Keras to evaluate models using cross-validation.
- How to perform data preparation in order to improve skill with Keras models.
- How to tune the network topology of models with Keras.

Step 1. Project Description
The problem that we will look at in this project is the Boston house price dataset (This dataset is also used in chapter 3 of our text book).

We have download the dataset for free and placed it in the project directory with the filename "housing.csv". You can also directly download the dataset:

https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.data

The dataset describes 13 numerical properties of houses in Boston suburbs and is concerned with modeling the price of houses in those suburbs in thousands of dollars. As such, this is a regression predictive modeling problem. Input attributes include things like crime rate, proportion of nonretail business acres, chemical concentrations and more.

This is a well-studied problem in machine learning. It is convenient to work with because all of the input and output attributes are numerical and there are 506 instances to work with.

Reasonable performance for models evaluated using Mean Squared Error (MSE) are around 20 in squared thousands of dollars (or $4,500 if you take the square root). This is a nice target to aim for with our neural network model.

Step 2. Develop a Baseline Neural Network Model
In this step we will create a baseline neural network model for the regression problem.

Let's start off by including all of the functions and objects we will need for this tutorial.

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

We can now load our dataset from a file in the local directory.

The dataset is in fact not in CSV format, the attributes are instead separated by whitespace. We can load this easily using the pandas library. We can then split the input (X) and output (Y) attributes so that they are easier to model with Keras and scikit-learn.

```
# load dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
```

We can create Keras models and evaluate them with scikit-learn by using handy wrapper objects provided by the Keras library. This is desirable, because scikit-learn excels at evaluating models and will allow us to use powerful data preparation and model evaluation schemes with very few lines of code.

The Keras wrappers require a function as an argument. This function that we must define is responsible for creating the neural network model to be evaluated.

Below we define the function to create the baseline model to be evaluated. It is a simple model that has a single fully connected hidden layer with the same number of neurons as input attributes (13). The network uses good practices such as the rectifier activation function for the hidden layer. No activation function is used for the output layer because it is a regression problem and we are interested in predicting numerical values directly without transform.

The efficient ADAM optimization algorithm is used and a mean squared error loss function is optimized. This will be the same metric that we will use to evaluate the performance of the model. It is a desirable metric because by taking the square root gives us an error value we can directly understand in the context of the problem (thousands of dollars).

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


	# Compile model

	return model
```

The Keras wrapper object for use in scikit-learn as a regression estimator is called KerasRegressor. We create an instance and pass it both the name of the function to create the neural network model as well as some parameters to pass along to the fit() function of the model later, such as the number of epochs and batch size. Both of these are set to sensible defaults.

We also initialize the random number generator with a constant random seed, a process we will repeat for each model evaluated in this tutorial. This is an attempt to ensure we compare models consistently.

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# evaluate model with standardized dataset
estimator = KerasRegressor(build_fn=baseline_model, epochs=100, batch_size=5, verbose=0)
```

The final step is to evaluate this baseline model. We will use 10-fold cross validation to evaluate the model.

```
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(estimator, X, Y, cv=kfold)
print("Results: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Running this code gives us an estimate of the model's performance on the problem for unseen data. The result reports the mean squared error including the average and standard deviation (average variance) across all 10 folds of the cross validation evaluation.

```
Baseline: 31.64 (26.82) MSE
```

Step 3: Modeling The Standardized Dataset
An important concern with the Boston house price dataset is that the input attributes all vary in their scales because they measure different quantities.

It is almost always good practice to prepare your data before modeling it using a neural network model.

Continuing on from the above baseline model, we can re-evaluate the same model using a standardized version of the input dataset.

We can use scikit-learn's Pipeline framework to perform the standardization during the model evaluation process, within each fold of the cross validation. This ensures that there is no data leakage from each testset cross validation fold into the training data:

http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

The code below creates a scikit-learn Pipeline that first standardizes the dataset then creates and evaluates the baseline neural network model.

```
# evaluate model with standardized dataset
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=baseline_model, epochs=50, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Standardized: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Running the code provides an improved performance over the baseline model without standardized data, dropping the error.

Standardized: 29.54 (27.87) MSE

Extension of Step 3:
A further extension of this step is to similarly apply a rescaling to the output variable such as normalizing it to the range of 0-1 and use a Sigmoid or similar activation function on the output layer to narrow output predictions to the same range.

Step 4. Tune The Neural Network Topology
There are many concerns that can be optimized for a neural network model.

Perhaps the point of biggest leverage is the structure of the network itself, including the number of layers and the number of neurons in each layer.

In this section we will evaluate two additional network topologies in an effort to further improve the performance of the model. We will look at both a deeper and a wider network topology.

Step 4.1. Evaluate a Deeper Network Topology
One way to improve the performance of a neural network is to add more layers. This might allow the model to extract and recombine higher order features embedded in the data.

In this step we will evaluate the effect of adding one more hidden layer to the model. This is as easy as defining a new function that will create this deeper model, copied from our baseline model above. We can then insert a new line after the first hidden layer. In this case with about half the number of neurons.

```
# define the model
def larger_model():
	# create model


	# Compile model

	return model
```

Our network topology now looks like:

13 inputs -> [13 -> 6] -> 1 output

We can evaluate this network topology in the same way as above, whilst also using the standardization of the dataset that above was shown to improve performance.

```
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=larger_model, epochs=50, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Larger: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Running this model does show a further improvement in performance from 28 down to 24 thousand squared dollars.

Larger: 22.83 (25.33) MSE

Step 4.2. Evaluate a Wider Network Topology

Another approach to increasing the representational capability of the model is to create a wider network.

In this section we evaluate the effect of keeping a shallow network architecture and nearly doubling the number of neurons in the one hidden layer.

Again, all we need to do is define a new function that creates our neural network model. Here, we have increased the number of neurons in the hidden layer compared to the baseline model from 13 to 20.

```
# define wider model
def wider_model():
	# create model

	# Compile model

	return model
```

Our network topology now looks like:

13 inputs -> [20] -> 1 output

We can evaluate the wider network topology using the same scheme as above:

```
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=wider_model, epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Building the model does see a further drop in error to about 21 thousand squared dollars. This is not a bad result for this problem.

Wider: 21.64 (23.75) MSE

It would have been hard to guess that a wider network would outperform a deeper network on this problem. The results demonstrate the importance of empirical testing when it comes to developing neural network models.

Step 5. 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 6. 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 7. Rewriting the code using the Keras Functional API
Now rewrite the code that you have written so far using the Keras Sequential API in Kearas Functional API.

Step 8. 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 9. 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: 10. 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 the Keras deep learning library for modeling regression problems.

Through this project you learned how to develop and evaluate neural network models, including:

- How to load data and develop a baseline model.
- How to lift performance using data preparation techniques like standardization.
- How to design and evaluate networks with different varying topologies on a problem.