ANN comprehension

Notebook:	2. Building an ANN!		
Created:	11/24/2018 8:06 PM	**Updated:**	5/11/2020 8:31 PM
Author:	yousef.kafif@outlook.com		

Our classification problem:

The skills learnt in this section can be applied to any customer-centric organisation!

- Can also be applied to any situation where you have lots of independent variables but only binary dependent variable outcomes.

Our Problem:

- Bank has an unusual churn rate, and wants you to figure out why using their customer data.
- We have IVs (features of customers) that affect the churn rate.
- We want a DV binary value which tells us if the customer will leave (1) or not (0)
    - We have a dataset that contains customer info (IVs we will use) and whether they have left or not (DV - Binary value), we will split this dataset into a training set to facilitate learning and a test set for evaluating the model.

Step 1. Preprocess Data #always
Step 2. Creating our ANN i.e. classifier
Step 3. Compiling it via classifier.compile() and training it via classifier.fit(X_train, y_train, batch_size, epoch)
Step 4. Predicting using the IV_test and evaluating models performance.

1. PREPROCESSING :-

1. Import and extract your data into the X and y - IV and DV variables.
2. Encode any categorical data or fix any missing data if needed.
3. Split the data into the test and training set.
4. SCALE IT! - Scales all values to a suitable range, prevents over valued IVs = more accurate, faster processing etc MUST BE DONE.

2. CREATING OUR ANN :-

1. Need to create an object of the 'sequential' class which declares and initializes a SEQUENCE OF LAYERS. This object is our classifier / ANN
2. Create the layers by using the classifier.dense() function to create our individual layers, assigning the # of nodes, activation function, kernel initializer ***#random weight initializer to numbers close to 0*** and our # of inputs to receive ***#only for the first layer, this creates the input layer.***
3. Set our model 'settings' with the classifier.compile() function to set the optimizing - gradient descent algorithm, the loss-cost function, and the metric-criterion used to judge the results.

4. Now run the model through the test set with the classifier.fit() function. In which you input the training set by arguments, and set the batch size ***#the ANN will optimize the weights after every batch*** and epochs to be completed in the session.

3. TESTING/PREDICTING and EVALUATING THE MODEL :-

1. You test the model by creating a new DV_predict object and assigning it to the classifier.predict() function which returns a set of predicted values based on the IV_test array given to it in the argument.
2. You can evaluate the model with the confusion matrix which return a [correct, wrong] [wrong, correct] result of which you can determine the accuracy. #I will learn new methods in the future.

- You can test the model on any set of data by converting the data into an array with the np.array() #remember each observation is a row. [[row1] , [row2]] and scale it with the same scaling object used on the training set. And finally use the predict method to see the result.

PART 1 :- DATA PRE-PROCESSING

1. **Importing the dataset & exporting the IVs and DVs to our object arrays.**

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
dataset = pd.read_csv('Data.csv')
```

- .read_csv(' mydatafile.csv ')
- Imports our dataset.

```
X = dataset.iloc[:, :-1].values # X is always INDEPENDANT VARIABLES
y = dataset.iloc[:,-1].values # y is always DEPENDANT VARIABLES
```

- Extracting our values. Designating which columns to extract from our data set according to our IVs and DV.

- **IF you need to encode categorical data, OR take care of missing data, YOU DO SO BEFORE SPLITTING!** *Look in preprocessing template for these shenanigans*

2. **NOW YOU CAN SPLIT IT INTO/Create OUR TRAINING & TEST SET.**

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 0) # Creates our X_train, y_train & X_test, y_test sets.
```

- train_test_split(<IV-object> , <DV-object> , test_size = ??)

3. **FEATURE SCALING!** ***# Always required! Scales all values to be within a suitable range. FASTER PROCESSING + NO OVERVALUED I.V***

   - Do not scale DV (y) if it is binary!

```
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
```

- Creating our StandardScaler object

```
X_train = sc_X.fit_transform(X_train)
```

- fitting our sc_X object to X_train and scaling X_train.
- We must ALWAYS fit_transform() our training sets!
- fit_transform(<array>) #transform = scale basically

```
X_test = sc_X.transform(X_test)
```

- Scaling X_test, sc_X has already been fitted to X_train and so is transforming on the same fit basis.

PART 2 :- MAKING THE CLASSIFIER / ANN and TRAINING IT!

1. **Creating the classifier - layers! # NOTE: Not all programs need deep networks i.e. layers.**

```
import keras # or whatever class you will use. e.g. LogisticRegression from
sklearn.linear_model <- No layers for this
from keras.models import Sequential
from keras.layers import Dense
classifier = Sequential( ) # Creating & Initializing
```

- Initializes / Creates our classifier - Defining it as a *SEQUENCE OF LAYERS.*

```
classifier.add(Dense(units = 6, activation = 'relu', kernel_initializer =
'uniform', input_shape=(11,) )) # Input + 1st Hidden Layer
```

- *Units* = # of Nodes = 6 -> We have 6 nodes (neurons) in this layer.
- *Activation* = 'relu' -> using the Rectifier activation function
- *kernel_initializer* = 'uniform' -> randomly initializes weights to small numbers close to 0
- *input_shape* = (# of inputs,) or *input_dim* = 11 -> Declares inputs ***#COMPULSORY FOR THE FIRST HIDDEN LAYER - IT CREATES THE INPUT LAYER***

```
classifier.add(Dense(units = 6, activation = 'relu', kernel_initializer =
'uniform')) # 2nd Hidden Layer
```

- The same as above, but does not need to know inputs! It has the hidden layer before it to know.

```
classifier.add(Dense(units = 1, activation = 'sigmoid', kernel_initializer =
'uniform')) # Output Layer
```

- Contains only 1 node because we have a categorical DV in binary. Just 2 categories, a 1 or 0.
  - What if I have multiple categories?
  - If we have 3 or more categories in our DV, that means we are using the **OneHotEncoded** method (meaning we transformed the 3

category DV column into 3 DummyVs -> 3 columns of binary output) and **thus our # of output nodes = # of categories (DV columns - DummyVs).**
  - We would also replace the 'Sigmoid' function with its multi-categorical version, the 'Softmax' function.
- We use the sigmoid function to achieve a probability output from our model!

2. **Compiling the ANN! - Basically applying Stochastic Gradient Descent on the whole network!**

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

- *optimizer* -> The algorithm applied to find/optimize the weights based on the gradient of the loss function. The Stochastic Gradient Descent algorithm!
  - There are many optimizers, 'adam' is just a good efficient one.
- *loss* -> The cost-error-loss function to be optimized to find the optimal weights.
  - Remember, applying the logistic regression loss function to a sigmoid function yields a logistic regression model from which we can see clear predictions.
    - This is normally applied to a binary DV, but can still be applied to a categorical DV with 2 or more classes. 'crossentropy'.
  - 'binary_crossentropy' is the logistic loss function applied to a binary output.
- *metrics* -> The criterion used to evaluate the model. The optimizer also uses this criterion to improve the models performance after every observation/batch.
  - The 'accuracy' metric is most typically used. Note that you can use multiple metrics so that is why the input is in array form.

- **Fitting classifier to the Training Set i.e. Connecting the ANN to the set and training it, it learns the correlation between IVs and DV in training set! //Skip to this part if an ANN wasn't needed.**

```
classifier.fit(X_train, y_train, batch_size=10, epochs = 100) #for an ANN
```

- *batch_size* -> # of observations before the ANN updates the weights. I.E 'trains' itself through gradient descent.
  - This is up to you, the A.I. artist.
- *epochs* -> The # of epochs to be completed. An epoch is a whole round through a training set.
  - Also up to you.

PART 3: PREDICTIONS & EVALUATIONS

- **Predicting the Test Set results**

```
y_pred = classifier.predict(X_test)
```

- Predicting the test set, using the predict() function. Remember Y is our DV (result), thus y_pred is our predicted DVs
- Outputs a probability as our ANN is outputting the probability.

```
y_pred = (y_pred > 0.5)
```

    - Needed to transform y_pred into a boolean value of true or false. 1 > .5 && 0 <.5.
    - Needed for confusion matrix, since our true DV results are in boolean, the two arrays to compare must be of the same value type!
- **Evaluating performance w/ confusion matrix #NEW METHODS WILL BE LEARNED**

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

    - Returns a 4x4 matrix. 1st Row = [Correct, Wrong] 2nd Row = [Wrong, Correct] -> (Correct + Correct)/ (Dataset count) = accuracy.

Testing ONE NEW observation on the model!

    - You have to :

1.Convert your observation into an array object w/ a row for each observation. i.e. A horizontal array.
2. Scale it!
3.Plug it in to predict()

        - Can be done in one line of code! #go python

```
New_ObservationPrediction = classifier.predict(sc.transform( np.array( [ [ 1st obs IVs - Features here ] , [ 2nd row obs] ,  [etc] ] ) ) )
```

        - Again, if you want a boolean - binary prediction use:

```
New_ObservationPrediction = (New_ObservationPrediction>0.5)
```