Open in Colab

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 8 Material

- Part 8.1: Introduction to Kaggle [Video] [Notebook]
- **Part 8.2: Building Ensembles with Scikit-Learn and Keras** [Video] [Notebook]
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [Video] [Notebook]
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [Video] [Notebook]
- Part 8.5: Current Semester's Kaggle [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow. Running the following code will map your GDrive to `/content/drive`.

In [1]:
```
try:
    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
```

```
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)
```

```
Mounted at /content/drive
Note: using Google CoLab
```

Part 8.2: Building Ensembles with Scikit-Learn and Keras

Evaluating Feature Importance

Feature importance tells us how important each feature (from the feature/import vector) is to predicting a neural network or another model. There are many different ways to evaluate the feature importance of neural networks. The following paper presents an excellent (and readable) overview of the various means of assessing the significance of neural network inputs/features.

- An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data [Cite:olden2004accurate]. *Ecological Modelling*, 178(3), 389-397.

In summary, the following methods are available to neural networks:

- Connection Weights Algorithm
- Partial Derivatives
- Input Perturbation
- Sensitivity Analysis
- Forward Stepwise Addition
- Improved Stepwise Selection 1
- Backward Stepwise Elimination
- Improved Stepwise Selection

For this chapter, we will use the input Perturbation feature ranking algorithm. This algorithm will work with any regression or classification network. In the next section, I provide an implementation of the input perturbation algorithm for scikit-learn. This code implements a function below that will work with any scikit-learn model.

Leo Breiman provided this algorithm in his seminal paper on random forests. [Citebreiman2001random:] Although he presented this algorithm in conjunction with random forests, it is model-independent and appropriate for any supervised learning model. This algorithm, known as the input perturbation algorithm, works by evaluating a trained model's accuracy with each input individually shuffled from a data set. Shuffling an input causes it to become useless—effectively removing it from the model. More important inputs will produce a less accurate score when they are removed by shuffling them. This process makes sense because important features will contribute to the model's accuracy. I first presented the TensorFlow implementation of this algorithm in the following paper.

- Early stabilizing feature importance for TensorFlow deep neural networks[Cite:heaton2017early]

This algorithm will use log loss to evaluate a classification problem and RMSE for regression.

In [2]:
```
from sklearn import metrics
import scipy as sp
import numpy as np
import math
from sklearn import metrics

def perturbation_rank(model, x, y, names, regression):
    errors = []

    for i in range(x.shape[1]):
        hold = np.array(x[:, i])
        np.random.shuffle(x[:, i])

        if regression:
            pred = model.predict(x)
            error = metrics.mean_squared_error(y, pred)
        else:
            pred = model.predict(x)
            error = metrics.log_loss(y, pred)

        errors.append(error)
        x[:, i] = hold

    max_error = np.max(errors)
    importance = [e/max_error for e in errors]

    data = {'name':names,'error':errors,'importance':importance}
    result = pd.DataFrame(data, columns = ['name','error','importance'])
    result.sort_values(by=['importance'], ascending=[0], inplace=True)
    result.reset_index(inplace=True, drop=True)
    return result
```

Classification and Input Perturbation Ranking

We now look at the code to perform perturbation ranking for a classification neural network. The implementation technique is slightly different for classification vs. regression, so I must provide two different implementations. The primary difference between classification and regression is how we evaluate the accuracy of the neural network in each of these two network types. We will use the Root Mean Square (RMSE) error calculation, whereas we will use log loss for classification.

The code presented below creates a classification neural network that will predict the classic iris dataset.

In [3]:
```
# HIDE OUTPUT
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output
model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x_train,y_train,verbose=2,epochs=100)
```

```
Epoch 1/100
4/4 - 1s - loss: 2.0814 - 1s/epoch - 292ms/step
Epoch 2/100
4/4 - 0s - loss: 1.6125 - 14ms/epoch - 4ms/step
Epoch 3/100
4/4 - 0s - loss: 1.3316 - 26ms/epoch - 7ms/step
Epoch 4/100
4/4 - 0s - loss: 1.2246 - 13ms/epoch - 3ms/step
Epoch 5/100
4/4 - 0s - loss: 1.1989 - 13ms/epoch - 3ms/step
Epoch 6/100
4/4 - 0s - loss: 1.1349 - 14ms/epoch - 4ms/step
Epoch 7/100
4/4 - 0s - loss: 1.0543 - 21ms/epoch - 5ms/step
Epoch 8/100
4/4 - 0s - loss: 0.9987 - 25ms/epoch - 6ms/step
Epoch 9/100
4/4 - 0s - loss: 0.9449 - 20ms/epoch - 5ms/step
Epoch 10/100
4/4 - 0s - loss: 0.9032 - 16ms/epoch - 4ms/step
Epoch 11/100
4/4 - 0s - loss: 0.8623 - 20ms/epoch - 5ms/step
Epoch 12/100
4/4 - 0s - loss: 0.8274 - 12ms/epoch - 3ms/step
Epoch 13/100
4/4 - 0s - loss: 0.8013 - 18ms/epoch - 4ms/step
Epoch 14/100
4/4 - 0s - loss: 0.7718 - 18ms/epoch - 5ms/step
Epoch 15/100
4/4 - 0s - loss: 0.7426 - 19ms/epoch - 5ms/step
Epoch 16/100
4/4 - 0s - loss: 0.7163 - 13ms/epoch - 3ms/step
Epoch 17/100
4/4 - 0s - loss: 0.6933 - 13ms/epoch - 3ms/step
Epoch 18/100
4/4 - 0s - loss: 0.6689 - 14ms/epoch - 3ms/step
Epoch 19/100
4/4 - 0s - loss: 0.6488 - 11ms/epoch - 3ms/step
Epoch 20/100
4/4 - 0s - loss: 0.6294 - 11ms/epoch - 3ms/step
Epoch 21/100
4/4 - 0s - loss: 0.6094 - 20ms/epoch - 5ms/step
Epoch 22/100
4/4 - 0s - loss: 0.5911 - 18ms/epoch - 4ms/step
Epoch 23/100
4/4 - 0s - loss: 0.5725 - 16ms/epoch - 4ms/step
Epoch 24/100
4/4 - 0s - loss: 0.5550 - 13ms/epoch - 3ms/step
Epoch 25/100
4/4 - 0s - loss: 0.5389 - 14ms/epoch - 3ms/step
Epoch 26/100
4/4 - 0s - loss: 0.5207 - 15ms/epoch - 4ms/step
Epoch 27/100
4/4 - 0s - loss: 0.5041 - 14ms/epoch - 4ms/step
Epoch 28/100
4/4 - 0s - loss: 0.4901 - 14ms/epoch - 3ms/step
```

```
Epoch 29/100
4/4 - 0s - loss: 0.4765 - 14ms/epoch - 4ms/step
Epoch 30/100
4/4 - 0s - loss: 0.4619 - 16ms/epoch - 4ms/step
Epoch 31/100
4/4 - 0s - loss: 0.4489 - 16ms/epoch - 4ms/step
Epoch 32/100
4/4 - 0s - loss: 0.4366 - 13ms/epoch - 3ms/step
Epoch 33/100
4/4 - 0s - loss: 0.4243 - 13ms/epoch - 3ms/step
Epoch 34/100
4/4 - 0s - loss: 0.4124 - 14ms/epoch - 3ms/step
Epoch 35/100
4/4 - 0s - loss: 0.4015 - 14ms/epoch - 3ms/step
Epoch 36/100
4/4 - 0s - loss: 0.3917 - 21ms/epoch - 5ms/step
Epoch 37/100
4/4 - 0s - loss: 0.3826 - 30ms/epoch - 7ms/step
Epoch 38/100
4/4 - 0s - loss: 0.3713 - 18ms/epoch - 4ms/step
Epoch 39/100
4/4 - 0s - loss: 0.3621 - 16ms/epoch - 4ms/step
Epoch 40/100
4/4 - 0s - loss: 0.3543 - 14ms/epoch - 4ms/step
Epoch 41/100
4/4 - 0s - loss: 0.3460 - 14ms/epoch - 4ms/step
Epoch 42/100
4/4 - 0s - loss: 0.3385 - 28ms/epoch - 7ms/step
Epoch 43/100
4/4 - 0s - loss: 0.3280 - 31ms/epoch - 8ms/step
Epoch 44/100
4/4 - 0s - loss: 0.3211 - 15ms/epoch - 4ms/step
Epoch 45/100
4/4 - 0s - loss: 0.3144 - 14ms/epoch - 3ms/step
Epoch 46/100
4/4 - 0s - loss: 0.3068 - 15ms/epoch - 4ms/step
Epoch 47/100
4/4 - 0s - loss: 0.2992 - 19ms/epoch - 5ms/step
Epoch 48/100
4/4 - 0s - loss: 0.2922 - 18ms/epoch - 4ms/step
Epoch 49/100
4/4 - 0s - loss: 0.2847 - 37ms/epoch - 9ms/step
Epoch 50/100
4/4 - 0s - loss: 0.2803 - 13ms/epoch - 3ms/step
Epoch 51/100
4/4 - 0s - loss: 0.2756 - 13ms/epoch - 3ms/step
Epoch 52/100
4/4 - 0s - loss: 0.2665 - 18ms/epoch - 4ms/step
Epoch 53/100
4/4 - 0s - loss: 0.2632 - 16ms/epoch - 4ms/step
Epoch 54/100
4/4 - 0s - loss: 0.2571 - 20ms/epoch - 5ms/step
Epoch 55/100
4/4 - 0s - loss: 0.2499 - 17ms/epoch - 4ms/step
Epoch 56/100
4/4 - 0s - loss: 0.2458 - 17ms/epoch - 4ms/step
```

```
Epoch 57/100
4/4 - 0s - loss: 0.2399 - 23ms/epoch - 6ms/step
Epoch 58/100
4/4 - 0s - loss: 0.2340 - 16ms/epoch - 4ms/step
Epoch 59/100
4/4 - 0s - loss: 0.2318 - 16ms/epoch - 4ms/step
Epoch 60/100
4/4 - 0s - loss: 0.2225 - 12ms/epoch - 3ms/step
Epoch 61/100
4/4 - 0s - loss: 0.2266 - 15ms/epoch - 4ms/step
Epoch 62/100
4/4 - 0s - loss: 0.2178 - 12ms/epoch - 3ms/step
Epoch 63/100
4/4 - 0s - loss: 0.2116 - 15ms/epoch - 4ms/step
Epoch 64/100
4/4 - 0s - loss: 0.2137 - 21ms/epoch - 5ms/step
Epoch 65/100
4/4 - 0s - loss: 0.2030 - 17ms/epoch - 4ms/step
Epoch 66/100
4/4 - 0s - loss: 0.2041 - 14ms/epoch - 3ms/step
Epoch 67/100
4/4 - 0s - loss: 0.2001 - 15ms/epoch - 4ms/step
Epoch 68/100
4/4 - 0s - loss: 0.1919 - 25ms/epoch - 6ms/step
Epoch 69/100
4/4 - 0s - loss: 0.1894 - 23ms/epoch - 6ms/step
Epoch 70/100
4/4 - 0s - loss: 0.1863 - 17ms/epoch - 4ms/step
Epoch 71/100
4/4 - 0s - loss: 0.1823 - 16ms/epoch - 4ms/step
Epoch 72/100
4/4 - 0s - loss: 0.1790 - 24ms/epoch - 6ms/step
Epoch 73/100
4/4 - 0s - loss: 0.1780 - 16ms/epoch - 4ms/step
Epoch 74/100
4/4 - 0s - loss: 0.1755 - 15ms/epoch - 4ms/step
Epoch 75/100
4/4 - 0s - loss: 0.1719 - 31ms/epoch - 8ms/step
Epoch 76/100
4/4 - 0s - loss: 0.1767 - 21ms/epoch - 5ms/step
Epoch 77/100
4/4 - 0s - loss: 0.1694 - 17ms/epoch - 4ms/step
Epoch 78/100
4/4 - 0s - loss: 0.1655 - 27ms/epoch - 7ms/step
Epoch 79/100
4/4 - 0s - loss: 0.1634 - 23ms/epoch - 6ms/step
Epoch 80/100
4/4 - 0s - loss: 0.1566 - 17ms/epoch - 4ms/step
Epoch 81/100
4/4 - 0s - loss: 0.1563 - 17ms/epoch - 4ms/step
Epoch 82/100
4/4 - 0s - loss: 0.1536 - 15ms/epoch - 4ms/step
Epoch 83/100
4/4 - 0s - loss: 0.1504 - 18ms/epoch - 4ms/step
Epoch 84/100
4/4 - 0s - loss: 0.1502 - 16ms/epoch - 4ms/step
```

```
Epoch 85/100
4/4 - 0s - loss: 0.1469 - 17ms/epoch - 4ms/step
Epoch 86/100
4/4 - 0s - loss: 0.1448 - 28ms/epoch - 7ms/step
Epoch 87/100
4/4 - 0s - loss: 0.1424 - 23ms/epoch - 6ms/step
Epoch 88/100
4/4 - 0s - loss: 0.1401 - 25ms/epoch - 6ms/step
Epoch 89/100
4/4 - 0s - loss: 0.1386 - 47ms/epoch - 12ms/step
Epoch 90/100
4/4 - 0s - loss: 0.1365 - 30ms/epoch - 7ms/step
Epoch 91/100
4/4 - 0s - loss: 0.1383 - 41ms/epoch - 10ms/step
Epoch 92/100
4/4 - 0s - loss: 0.1332 - 12ms/epoch - 3ms/step
Epoch 93/100
4/4 - 0s - loss: 0.1311 - 20ms/epoch - 5ms/step
Epoch 94/100
4/4 - 0s - loss: 0.1320 - 20ms/epoch - 5ms/step
Epoch 95/100
4/4 - 0s - loss: 0.1302 - 17ms/epoch - 4ms/step
Epoch 96/100
4/4 - 0s - loss: 0.1311 - 19ms/epoch - 5ms/step
Epoch 97/100
4/4 - 0s - loss: 0.1248 - 14ms/epoch - 3ms/step
Epoch 98/100
4/4 - 0s - loss: 0.1254 - 12ms/epoch - 3ms/step
Epoch 99/100
4/4 - 0s - loss: 0.1275 - 13ms/epoch - 3ms/step
Epoch 100/100
4/4 - 0s - loss: 0.1225 - 41ms/epoch - 10ms/step
```

```
Out[3]: <keras.callbacks.History at 0x7fc869fd2950>
```

Next, we evaluate the accuracy of the trained model. Here we see that the neural network performs great, with an accuracy of 1.0. We might fear overfitting with such high accuracy for a more complex dataset. However, for this example, we are more interested in determining the importance of each column.

```
In [4]: from sklearn.metrics import accuracy_score

pred = model.predict(x_test)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y_test,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

```
Accuracy: 1.0
```

We are now ready to call the input perturbation algorithm. First, we extract the column names and remove the target column. The target column is not important, as it is the objective, not one of the inputs. In supervised learning, the target is of the utmost importance.

We can see the importance displayed in the following table. The most important column is always 1.0, and lessor columns will continue in a downward trend. The least important column will have the lowest rank.

In [5]:
```
# Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("species") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank)
```

	name	error	importance
0	petal_l	2.609378	1.000000
1	petal_w	0.480387	0.184100
2	sepal_l	0.223239	0.085553
3	sepal_w	0.128518	0.049252

Regression and Input Perturbation Ranking

We now see how to use input perturbation ranking for a regression neural network. We will use the MPG dataset as a demonstration. The code below loads the MPG dataset and creates a regression neural network for this dataset. The code trains the neural network and calculates an RMSE evaluation.

In [6]:
```
# HIDE OUTPUT
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

save_path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
```

```
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x_train,y_train,verbose=2,epochs=100)

# Predict
pred = model.predict(x)
```

```
Epoch 1/100
10/10 - 1s - loss: 328433.8125 - 898ms/epoch - 90ms/step
Epoch 2/100
10/10 - 0s - loss: 78914.6406 - 26ms/epoch - 3ms/step
Epoch 3/100
10/10 - 0s - loss: 5371.1025 - 50ms/epoch - 5ms/step
Epoch 4/100
10/10 - 0s - loss: 4021.9753 - 34ms/epoch - 3ms/step
Epoch 5/100
10/10 - 0s - loss: 4438.5728 - 33ms/epoch - 3ms/step
Epoch 6/100
10/10 - 0s - loss: 1030.3115 - 34ms/epoch - 3ms/step
Epoch 7/100
10/10 - 0s - loss: 594.9177 - 31ms/epoch - 3ms/step
Epoch 8/100
10/10 - 0s - loss: 655.3908 - 31ms/epoch - 3ms/step
Epoch 9/100
10/10 - 0s - loss: 465.0457 - 25ms/epoch - 2ms/step
Epoch 10/100
10/10 - 0s - loss: 458.7520 - 30ms/epoch - 3ms/step
Epoch 11/100
10/10 - 0s - loss: 452.4102 - 22ms/epoch - 2ms/step
Epoch 12/100
10/10 - 0s - loss: 439.8730 - 25ms/epoch - 3ms/step
Epoch 13/100
10/10 - 0s - loss: 434.8245 - 27ms/epoch - 3ms/step
Epoch 14/100
10/10 - 0s - loss: 433.7303 - 25ms/epoch - 3ms/step
Epoch 15/100
10/10 - 0s - loss: 427.2859 - 46ms/epoch - 5ms/step
Epoch 16/100
10/10 - 0s - loss: 424.1164 - 50ms/epoch - 5ms/step
Epoch 17/100
10/10 - 0s - loss: 422.3007 - 42ms/epoch - 4ms/step
Epoch 18/100
10/10 - 0s - loss: 418.4877 - 31ms/epoch - 3ms/step
Epoch 19/100
10/10 - 0s - loss: 414.2283 - 23ms/epoch - 2ms/step
Epoch 20/100
10/10 - 0s - loss: 410.2691 - 34ms/epoch - 3ms/step
Epoch 21/100
10/10 - 0s - loss: 407.0490 - 29ms/epoch - 3ms/step
Epoch 22/100
10/10 - 0s - loss: 406.2433 - 46ms/epoch - 5ms/step
Epoch 23/100
10/10 - 0s - loss: 399.7404 - 37ms/epoch - 4ms/step
Epoch 24/100
10/10 - 0s - loss: 396.3280 - 66ms/epoch - 7ms/step
Epoch 25/100
10/10 - 0s - loss: 391.0629 - 28ms/epoch - 3ms/step
Epoch 26/100
10/10 - 0s - loss: 387.3203 - 26ms/epoch - 3ms/step
Epoch 27/100
10/10 - 0s - loss: 382.7670 - 54ms/epoch - 5ms/step
Epoch 28/100
10/10 - 0s - loss: 380.6316 - 21ms/epoch - 2ms/step
```

```
Epoch 29/100
10/10 - 0s - loss: 375.9518 - 30ms/epoch - 3ms/step
Epoch 30/100
10/10 - 0s - loss: 372.7001 - 24ms/epoch - 2ms/step
Epoch 31/100
10/10 - 0s - loss: 366.7871 - 24ms/epoch - 2ms/step
Epoch 32/100
10/10 - 0s - loss: 363.4180 - 42ms/epoch - 4ms/step
Epoch 33/100
10/10 - 0s - loss: 359.6006 - 47ms/epoch - 5ms/step
Epoch 34/100
10/10 - 0s - loss: 359.4055 - 46ms/epoch - 5ms/step
Epoch 35/100
10/10 - 0s - loss: 350.7181 - 29ms/epoch - 3ms/step
Epoch 36/100
10/10 - 0s - loss: 348.6260 - 42ms/epoch - 4ms/step
Epoch 37/100
10/10 - 0s - loss: 343.6122 - 28ms/epoch - 3ms/step
Epoch 38/100
10/10 - 0s - loss: 339.6165 - 32ms/epoch - 3ms/step
Epoch 39/100
10/10 - 0s - loss: 334.5634 - 32ms/epoch - 3ms/step
Epoch 40/100
10/10 - 0s - loss: 332.6061 - 34ms/epoch - 3ms/step
Epoch 41/100
10/10 - 0s - loss: 326.7434 - 22ms/epoch - 2ms/step
Epoch 42/100
10/10 - 0s - loss: 323.8063 - 40ms/epoch - 4ms/step
Epoch 43/100
10/10 - 0s - loss: 320.2585 - 29ms/epoch - 3ms/step
Epoch 44/100
10/10 - 0s - loss: 315.3609 - 23ms/epoch - 2ms/step
Epoch 45/100
10/10 - 0s - loss: 311.4920 - 23ms/epoch - 2ms/step
Epoch 46/100
10/10 - 0s - loss: 308.9212 - 29ms/epoch - 3ms/step
Epoch 47/100
10/10 - 0s - loss: 303.1410 - 24ms/epoch - 2ms/step
Epoch 48/100
10/10 - 0s - loss: 299.9317 - 24ms/epoch - 2ms/step
Epoch 49/100
10/10 - 0s - loss: 294.4305 - 23ms/epoch - 2ms/step
Epoch 50/100
10/10 - 0s - loss: 291.4469 - 24ms/epoch - 2ms/step
Epoch 51/100
10/10 - 0s - loss: 287.3263 - 41ms/epoch - 4ms/step
Epoch 52/100
10/10 - 0s - loss: 284.3096 - 49ms/epoch - 5ms/step
Epoch 53/100
10/10 - 0s - loss: 280.5522 - 30ms/epoch - 3ms/step
Epoch 54/100
10/10 - 0s - loss: 276.1487 - 26ms/epoch - 3ms/step
Epoch 55/100
10/10 - 0s - loss: 271.3444 - 42ms/epoch - 4ms/step
Epoch 56/100
10/10 - 0s - loss: 280.0936 - 33ms/epoch - 3ms/step
```

```
Epoch 57/100
10/10 - 0s - loss: 263.7166 - 40ms/epoch - 4ms/step
Epoch 58/100
10/10 - 0s - loss: 261.6750 - 56ms/epoch - 6ms/step
Epoch 59/100
10/10 - 0s - loss: 258.5714 - 45ms/epoch - 4ms/step
Epoch 60/100
10/10 - 0s - loss: 252.6791 - 31ms/epoch - 3ms/step
Epoch 61/100
10/10 - 0s - loss: 250.1348 - 53ms/epoch - 5ms/step
Epoch 62/100
10/10 - 0s - loss: 246.4157 - 72ms/epoch - 7ms/step
Epoch 63/100
10/10 - 0s - loss: 242.3768 - 46ms/epoch - 5ms/step
Epoch 64/100
10/10 - 0s - loss: 238.7874 - 28ms/epoch - 3ms/step
Epoch 65/100
10/10 - 0s - loss: 235.8578 - 42ms/epoch - 4ms/step
Epoch 66/100
10/10 - 0s - loss: 233.7492 - 24ms/epoch - 2ms/step
Epoch 67/100
10/10 - 0s - loss: 229.0066 - 26ms/epoch - 3ms/step
Epoch 68/100
10/10 - 0s - loss: 225.7449 - 25ms/epoch - 3ms/step
Epoch 69/100
10/10 - 0s - loss: 223.5038 - 25ms/epoch - 2ms/step
Epoch 70/100
10/10 - 0s - loss: 219.9561 - 39ms/epoch - 4ms/step
Epoch 71/100
10/10 - 0s - loss: 215.1055 - 58ms/epoch - 6ms/step
Epoch 72/100
10/10 - 0s - loss: 211.9364 - 39ms/epoch - 4ms/step
Epoch 73/100
10/10 - 0s - loss: 208.1019 - 55ms/epoch - 5ms/step
Epoch 74/100
10/10 - 0s - loss: 207.4119 - 34ms/epoch - 3ms/step
Epoch 75/100
10/10 - 0s - loss: 206.8693 - 40ms/epoch - 4ms/step
Epoch 76/100
10/10 - 0s - loss: 197.9749 - 49ms/epoch - 5ms/step
Epoch 77/100
10/10 - 0s - loss: 196.9090 - 34ms/epoch - 3ms/step
Epoch 78/100
10/10 - 0s - loss: 192.6349 - 45ms/epoch - 4ms/step
Epoch 79/100
10/10 - 0s - loss: 189.6783 - 31ms/epoch - 3ms/step
Epoch 80/100
10/10 - 0s - loss: 186.6584 - 25ms/epoch - 2ms/step
Epoch 81/100
10/10 - 0s - loss: 186.1920 - 29ms/epoch - 3ms/step
Epoch 82/100
10/10 - 0s - loss: 181.1735 - 31ms/epoch - 3ms/step
Epoch 83/100
10/10 - 0s - loss: 177.9338 - 51ms/epoch - 5ms/step
Epoch 84/100
10/10 - 0s - loss: 174.6662 - 87ms/epoch - 9ms/step
```

```
Epoch 85/100
10/10 - 0s - loss: 172.9421 - 90ms/epoch - 9ms/step
Epoch 86/100
10/10 - 0s - loss: 169.1906 - 58ms/epoch - 6ms/step
Epoch 87/100
10/10 - 0s - loss: 166.4181 - 57ms/epoch - 6ms/step
Epoch 88/100
10/10 - 0s - loss: 163.7466 - 36ms/epoch - 4ms/step
Epoch 89/100
10/10 - 0s - loss: 161.4653 - 29ms/epoch - 3ms/step
Epoch 90/100
10/10 - 0s - loss: 158.6274 - 30ms/epoch - 3ms/step
Epoch 91/100
10/10 - 0s - loss: 159.4237 - 32ms/epoch - 3ms/step
Epoch 92/100
10/10 - 0s - loss: 159.2035 - 31ms/epoch - 3ms/step
Epoch 93/100
10/10 - 0s - loss: 150.2793 - 38ms/epoch - 4ms/step
Epoch 94/100
10/10 - 0s - loss: 148.9276 - 36ms/epoch - 4ms/step
Epoch 95/100
10/10 - 0s - loss: 146.7706 - 34ms/epoch - 3ms/step
Epoch 96/100
10/10 - 0s - loss: 144.4946 - 29ms/epoch - 3ms/step
Epoch 97/100
10/10 - 0s - loss: 141.5782 - 28ms/epoch - 3ms/step
Epoch 98/100
10/10 - 0s - loss: 139.3355 - 27ms/epoch - 3ms/step
Epoch 99/100
10/10 - 0s - loss: 136.9762 - 56ms/epoch - 6ms/step
Epoch 100/100
10/10 - 0s - loss: 135.6660 - 23ms/epoch - 2ms/step
```

Just as before, we extract the column names and discard the target. We can now create a ranking of the importance of each of the input features. The feature with a ranking of 1.0 is the most important.

In [7]:

```python
# Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("name")
names.remove("mpg") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, True)
display(rank)
```

	name	error	importance
0	displacement	139.657598	1.000000
1	acceleration	139.261508	0.997164
2	origin	134.637690	0.964056
3	year	134.177126	0.960758
4	cylinders	132.747246	0.950519
5	horsepower	121.501102	0.869993
6	weight	75.244610	0.538779

Biological Response with Neural Network

The following sections will demonstrate how to use feature importance ranking and ensembling with a more complex dataset. Ensembling is the process where you combine multiple models for greater accuracy. Kaggle competition winners frequently make use of ensembling for high-ranking solutions.

We will use the biological response dataset, a Kaggle dataset, where there is an unusually high number of columns. Because of the large number of columns, it is essential to use feature ranking to determine the importance of these columns. We begin by loading the dataset and preprocessing. This Kaggle dataset is a binary classification problem. You must predict if certain conditions will cause a biological response.

- Predicting a Biological Response

In [8]:
```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import KFold
from IPython.display import HTML, display

URL = "https://data.heatonresearch.com/data/t81-558/kaggle/"

df_train = pd.read_csv(
    URL+"bio_train.csv",
    na_values=['NA', '?'])

df_test = pd.read_csv(
    URL+"bio_test.csv",
    na_values=['NA', '?'])

activity_classes = df_train['Activity']
```

A large number of columns is evident when we display the shape of the dataset.

In [9]:
```
print(df_train.shape)
```

```
(3751, 1777)
```

The following code constructs a classification neural network and trains it for the biological response dataset. Once trained, the accuracy is measured.

In [10]:
```
import os
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np
import sklearn

# Encode feature vector
# Convert to numpy - Classification
x_columns = df_train.columns.drop('Activity')
x = df_train[x_columns].values
y = df_train['Activity'].values # Classification
x_submit = df_test[x_columns].values.astype(np.float32)


# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

print("Fitting/Training...")
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu'))
model.add(Dense(10))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto')
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)
print("Fitting done...")

# Predict
pred = model.predict(x_test).flatten()


# Clip so that min is never exactly 0, max never 1
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))
print("Validation logloss: {}".format(
    sklearn.metrics.log_loss(y_test,pred)))

# Evaluate success using accuracy
pred = pred>0.5 # If greater than 0.5 probability, then true
score = metrics.accuracy_score(y_test, pred)
```

```
print("Validation accuracy score: {}".format(score))

# Build real submit file
pred_submit = model.predict(x_submit)

# Clip so that min is never exactly 0, max never 1 (would be a NaN score)
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))
submit_df = pd.DataFrame({'MoleculeId':[x+1 for x \
        in range(len(pred_submit))],'PredictedProbability':\
                                  pred_submit.flatten()})
submit_df.to_csv("submit.csv", index=False)
```

```
Fitting/Training...
Epoch 7: early stopping
Fitting done...
Validation logloss: 0.5564708781752792
Validation accuracy score: 0.7515991471215352
```

What Features/Columns are Important

The following uses perturbation ranking to evaluate the neural network.

In [11]:
```
# Rank the features
from IPython.display import display, HTML

names = list(df_train.columns) # x+y column names
names.remove("Activity") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank[0:10])
```

	name	error	importance
0	D27	0.603974	1.000000
1	D1049	0.565997	0.937122
2	D51	0.565883	0.936934
3	D998	0.563872	0.933604
4	D1059	0.563745	0.933394
5	D961	0.563723	0.933357
6	D1407	0.563532	0.933041
7	D1309	0.562244	0.930908
8	D1100	0.561902	0.930341
9	D1275	0.561659	0.929940

Neural Network Ensemble

A neural network ensemble combines neural network predictions with other models. The program determines the exact blend of these models by logistic regression. The following code performs this blend for a classification. If you present the final predictions from the ensemble to Kaggle, you will see that the result is very accurate.

In [12]:
```
# HIDE OUTPUT
import numpy as np
import os
import pandas as pd
import math
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression

SHUFFLE = False
FOLDS = 10

def build_ann(input_size,classes,neurons):
    model = Sequential()
    model.add(Dense(neurons, input_dim=input_size, activation='relu'))
    model.add(Dense(1))
    model.add(Dense(classes,activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    return model

def mlogloss(y_test, preds):
    epsilon = 1e-15
    sum = 0
    for row in zip(preds,y_test):
        x = row[0][row[1]]
        x = max(epsilon,x)
        x = min(1-epsilon,x)
        sum+=math.log(x)
    return( (-1/len(preds))*sum)

def stretch(y):
    return (y - y.min()) / (y.max() - y.min())


def blend_ensemble(x, y, x_submit):
    kf = StratifiedKFold(FOLDS)
    folds = list(kf.split(x,y))

    models = [
        KerasClassifier(build_fn=build_ann,neurons=20,
                        input_size=x.shape[1],classes=2),
        KNeighborsClassifier(n_neighbors=3),
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='gini'),
```

```
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='entropy'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='gini'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='entropy'),
        GradientBoostingClassifier(learning_rate=0.05,
                subsample=0.5, max_depth=6, n_estimators=50)]

    dataset_blend_train = np.zeros((x.shape[0], len(models)))
    dataset_blend_test = np.zeros((x_submit.shape[0], len(models)))

    for j, model in enumerate(models):
        print("Model: {} : {}".format(j, model) )
        fold_sums = np.zeros((x_submit.shape[0], len(folds)))
        total_loss = 0
        for i, (train, test) in enumerate(folds):
            x_train = x[train]
            y_train = y[train]
            x_test = x[test]
            y_test = y[test]
            model.fit(x_train, y_train)
            pred = np.array(model.predict_proba(x_test))
            dataset_blend_train[test, j] = pred[:, 1]
            pred2 = np.array(model.predict_proba(x_submit))
            fold_sums[:, i] = pred2[:, 1]
            loss = mlogloss(y_test, pred)
            total_loss+=loss
            print("Fold #{}: loss={}".format(i,loss))
        print("{}: Mean loss={}".format(model.__class__.__name__,
                                        total_loss/len(folds)))
        dataset_blend_test[:, j] = fold_sums.mean(1)

    print()
    print("Blending models.")
    blend = LogisticRegression(solver='lbfgs')
    blend.fit(dataset_blend_train, y)
    return blend.predict_proba(dataset_blend_test)

if __name__ == '__main__':

    np.random.seed(42)  # seed to shuffle the train set

    print("Loading data...")
    URL = "https://data.heatonresearch.com/data/t81-558/kaggle/"

    df_train = pd.read_csv(
        URL+"bio_train.csv",
        na_values=['NA', '?'])

    df_submit = pd.read_csv(
        URL+"bio_test.csv",
        na_values=['NA', '?'])

    predictors = list(df_train.columns.values)
    predictors.remove('Activity')
```

```
    x = df_train[predictors].values
    y = df_train['Activity']
    x_submit = df_submit.values

    if SHUFFLE:
        idx = np.random.permutation(y.size)
        x = x[idx]
        y = y[idx]

    submit_data = blend_ensemble(x, y, x_submit)
    submit_data = stretch(submit_data)

    ####################
    # Build submit file
    ####################
    ids = [id+1 for id in range(submit_data.shape[0])]
    submit_df = pd.DataFrame({'MoleculeId': ids,
                              'PredictedProbability':
                              submit_data[:, 1]},
                             columns=['MoleculeId',
                             'PredictedProbability'])
    submit_df.to_csv("submit.csv", index=False)
```

```
Loading data...
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:44: Deprecation
Warning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/ad
riangb/scikeras) instead. See https://www.adriangb.com/scikeras/stable/migra
tion.html for help migrating.
```

```
Model: 0 : <keras.wrappers.scikit_learn.KerasClassifier object at 0x7fc86980
9610>
106/106 [==============================] - 1s 2ms/step - loss: 0.6048
Fold #0: loss=0.5544745638322883
106/106 [==============================] - 1s 2ms/step - loss: 0.6046
Fold #1: loss=0.5684765604955473
106/106 [==============================] - 1s 2ms/step - loss: 0.5943
Fold #2: loss=0.5214491621944897
106/106 [==============================] - 1s 2ms/step - loss: 0.6301
Fold #3: loss=0.5264746750391351
106/106 [==============================] - 1s 2ms/step - loss: 0.5905
Fold #4: loss=0.5327822461352748
106/106 [==============================] - 1s 2ms/step - loss: 0.5993
Fold #5: loss=0.5800157462831582
106/106 [==============================] - 1s 2ms/step - loss: 0.5877
Fold #6: loss=0.5189563830365144
106/106 [==============================] - 1s 2ms/step - loss: 0.6038
Fold #7: loss=0.5625417655617023
106/106 [==============================] - 1s 2ms/step - loss: 0.5935
Fold #8: loss=0.5238374326475557
106/106 [==============================] - 1s 2ms/step - loss: 0.5991
Fold #9: loss=0.5322226787930878
KerasClassifier: Mean loss=0.5421231214018752
Model: 1 : KNeighborsClassifier(n_neighbors=3)
Fold #0: loss=3.606678388314123
Fold #1: loss=2.2256421551487593
Fold #2: loss=3.6815437059542186
Fold #3: loss=2.416161292225968
Fold #4: loss=4.442472310149748
Fold #5: loss=4.321350530738247
Fold #6: loss=3.400455469543658
Fold #7: loss=3.1724147110842513
Fold #8: loss=2.117356283193681
Fold #9: loss=3.0532135963322586
KNeighborsClassifier: Mean loss=3.243728844268491
Model: 2 : RandomForestClassifier(n_jobs=-1)
Fold #0: loss=0.4657177982691548
Fold #1: loss=0.4346825805694879
Fold #2: loss=0.4593868993445528
Fold #3: loss=0.41674899522216713
Fold #4: loss=0.4851849131056564
Fold #5: loss=0.48473291073937
Fold #6: loss=0.41274608628217674
Fold #7: loss=0.47405291219252377
Fold #8: loss=0.44974230059938286
Fold #9: loss=0.46340159258241087
RandomForestClassifier: Mean loss=0.45463969889068834
Model: 3 : RandomForestClassifier(criterion='entropy', n_jobs=-1)
Fold #0: loss=0.4511847247326708
Fold #1: loss=0.42707704254926593
Fold #2: loss=0.5550335199035183
Fold #3: loss=0.42186970733328516
Fold #4: loss=0.4794331756190797
Fold #5: loss=0.4730559509802762
Fold #6: loss=0.41116235817215196
Fold #7: loss=0.46835919493314265
```

```
Fold #8: loss=0.4496144890690015
Fold #9: loss=0.4625902934553457
RandomForestClassifier: Mean loss=0.4599380456747738
Model: 4 : ExtraTreesClassifier(n_jobs=-1)
Fold #0: loss=0.45496751079363495
Fold #1: loss=0.5013051157905043
Fold #2: loss=0.5886179891724027
Fold #3: loss=0.41646902160044674
Fold #4: loss=0.4957910697444236
Fold #5: loss=0.4773401028797005
Fold #6: loss=0.41935061504547827
Fold #7: loss=0.5757908399174205
Fold #8: loss=0.4585195863412778
Fold #9: loss=0.6210675972963805
ExtraTreesClassifier: Mean loss=0.500921944858167
Model: 5 : ExtraTreesClassifier(criterion='entropy', n_jobs=-1)
Fold #0: loss=0.44825346440152214
Fold #1: loss=0.40764412171784686
Fold #2: loss=0.5819367378417363
Fold #3: loss=0.4140589874942631
Fold #4: loss=0.4923489720481471
Fold #5: loss=0.5744429921555051
Fold #6: loss=0.42334390524742155
Fold #7: loss=0.6409291880353659
Fold #8: loss=0.45627884947155956
Fold #9: loss=0.466653395317917
ExtraTreesClassifier: Mean loss=0.49058906137312847
Model: 6 : GradientBoostingClassifier(learning_rate=0.05, max_depth=6, n_est
imators=50,
                           subsample=0.5)
Fold #0: loss=0.4789324034433162
Fold #1: loss=0.4565636914381977
Fold #2: loss=0.47057741836357014
Fold #3: loss=0.4436328438944843
Fold #4: loss=0.4883293501002484
Fold #5: loss=0.4843521206311074
Fold #6: loss=0.4436043855503229
Fold #7: loss=0.45977393398397765
Fold #8: loss=0.46632256794136323
Fold #9: loss=0.4703354072414907
GradientBoostingClassifier: Mean loss=0.4662424122588078

Blending models.
```

In [12]: