# ****M Owais Tahir | Binary Classification Sonar Project 1 for the Navy: Mines vs. Rocks****

**Step 1. Description of the Dataset**

The dataset we will use in this tutorial is the Sonar dataset.

This is a dataset that describes sonar chirp returns bouncing off different services. The 60 input variables are the strength of the returns at different angles. It is a binary classification problem that requires a model to differentiate rocks from metal cylinders

**Step 2. Baseline Neural Network Model Performance**

Let’s create a baseline model and result for this project.

We will start off by importing all of the classes and functions we will need:

**import numpy as np**

**import pandas as pd**

**from keras.models import Sequential**

**from keras.layers import Dense**

**from keras.wrappers.scikit\_learn import KerasClassifier**

**from sklearn.model\_selection import cross\_val\_score**

**from sklearn.preprocessing import LabelEncoder**

**from sklearn.model\_selection import StratifiedKFold**

**from sklearn.preprocessing import StandardScaler**

**from sklearn.pipeline import Pipeline**

Next, we can initialize the random number generator to ensure that we always get the same results when executing this code. This will help if we are debugging:

**seed = 7**

**np.random.seed(seed)**

Now we can load the dataset using pandas and split the columns into 60 input variables (X) and 1 output variable (Y). We use pandas to load the data because it easily handles strings (the output variable), whereas attempting to load the data directly using NumPy would be more difficult

**from google.colab import files**

**uploaded = files.upload()**

**for fn in uploaded.keys():**

**print('User uploaded file "{name}" with length {length} bytes'.format(**

**name=fn, length=len(uploaded[fn])))**

**fn = pd.read\_csv("sonar.csv", header=None)**

**# limit to categorical data using df.select\_dtypes()**

**y=fn.select\_dtypes(include=[object])**

**x=fn.select\_dtypes(include=[np.number])**

**le = LabelEncoder()**

**# 2/3. FIT AND TRANSFORM**

**# use df.apply() to apply le.fit\_transform to all columns**

**y = y.apply(le.fit\_transform)**

**In [0]:**

Finally, we are using the logarithmic loss function (binary\_crossentropy) during training, the preferred loss function for binary classification problems. The model also uses the efficient Adam optimization algorithm for gradient descent and accuracy metrics will be collected when the model is trained.

**def create\_baseline():**

**model =Sequential()**

**model.add(Dense(60, activation='relu', input\_shape=(60,)))**

**model.add(Dense(1, activation='sigmoid'))**

**model.compile(optimizer='Adam',loss='binary\_crossentropy',metrics=['accuracy'])**

**return model**

**Step 3. Re-Run The Baseline Model With Data Preparation**

It is a good practice to prepare your data before modeling.

Neural network models are especially suitable to having consistent input values, both in scale and distribution.

**# evaluate model with standardized dataset**

**estimator = KerasClassifier(build\_fn=create\_baseline, epochs=100, batch\_size=5, verbose=0)**

**kfold = StratifiedKFold(n\_splits=10, shuffle=True, random\_state=seed)**

**results = cross\_val\_score(estimator, x, y, cv=kfold)**

**print("Results: %.2f%% (%.2f%%)" % (results.mean()\*100, results.std()\*100))**

**Results: 83.71% (6.13%)**

**Step 4. 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.

**np.random.seed(seed)**

**estimators = []**

**estimators.append(('standardize', StandardScaler()))**

**estimators.append(('mlp', KerasClassifier(build\_fn=create\_baseline, epochs=100, batch\_size=5, verbose=0)))**

**pipeline = Pipeline(estimators)**

**kfold = StratifiedKFold(n\_splits=10, shuffle=True, random\_state=seed)**

**results = cross\_val\_score(pipeline, x, y, cv=kfold)**

**print("Standardized: %.2f%% (%.2f%%)" % (results.mean()\*100, results.std()\*100))**

**Standardized: 85.59% (7.46%)**

**In [0]:**

**# baseline model**

**def create\_larger():**

**model =Sequential()**

**model.add(Dense(60, activation='relu', input\_shape=(60,)))**

**model.add(Dense(30, activation='relu', input\_shape=(60,)))**

**#model.add(Dense(60, activation='relu', input\_shape=(30,)))**

**model.add(Dense(1, activation='sigmoid'))**

**# Compile model, write code below**

**model.compile(optimizer='Adam',loss='binary\_crossentropy',metrics=['accuracy'])**

**return model**

**# evaluate model with standardized dataset**

**estimator = KerasClassifier(build\_fn=create\_baseline, epochs=100, batch\_size=5, verbose=0)**

**kfold = StratifiedKFold(n\_splits=10, shuffle=True, random\_state=seed)**

**results = cross\_val\_score(estimator, x, y, cv=kfold)**

**print("Results: %.2f%% (%.2f%%)" % (results.mean()\*100, results.std()\*100))**

**np.random.seed(seed)**

**estimators = []**

**estimators.append(('standardize', StandardScaler()))**

**estimators.append(('mlp', KerasClassifier(build\_fn=create\_baseline, epochs=100, batch\_size=5, verbose=0)))**

**pipeline = Pipeline(estimators)**

**kfold = StratifiedKFold(n\_splits=10, shuffle=True, random\_state=seed)**

**results = cross\_val\_score(pipeline, x, y, cv=kfold)**

**print("Larger: %.2f%% (%.2f%%)" % (results.mean()\*100, results.std()\*100))**

**Results: 84.13% (3.83%)**

**Larger: 85.59% (7.46%)**