DEEP LEARNING AND REINFORCEMENT LEARNING - ASSIGNMENT

PEER-GRADED ASSIGNMENT

0. Introduction

As instructed, this report is designed to be presented to the Chief Data Officer or the Head of Analytics at a fictional company (you).

The Jupyter book is given as an attachment but is not important in this assignment.

This assignment focuses to answer as instructed by IBM course creators and bring calculated-based conclusions and constructive analytical analysis to a senior audience, i.e. the Chief Data Officer or the Head of Analytics, that will broken into the 7 major topics;

1.Main objective of the analysis that also specifies whether your model will be focused on a specific type of Deep Learning or Reinforcement Learning algorithm and the benefits that your analysis brings to the business or stakeholders of this data.

2.Brief description of the data set you chose, a summary of its attributes, and an outline of what you are trying to accomplish with this analysis.

3.Brief summary of data exploration and actions taken for data cleaning or feature engineering.

4.Summary of training at least three variations of the Deep Learning model you selected. For example, you can use different clustering techniques or different hyperparameters.

5.A paragraph explaining which of your Deep Learning models you recommend as a final model that best fits your needs in terms of accuracy or explainability.

6.Summary Key Findings and Insights, which walks your reader through the main findings of your modeling exercise.

7.Suggestions for next steps in analyzing this data, which may include suggesting revisiting this model or adding specific data features to achieve a better model.

FULL JUPYTER NOTEBOOK WITH ALL ANALYTICAL DETAILS AT END OF THIS REPORT

1. Main Objectives

Two years ago I did a proof of concept in predicting whether a fan was running, stopped or imbalanced (dust on blades). I went further to predict two different types of problems - imbalanced fan and airflow blocked. In this proof of concept I used and accelerometer placed on a fan and thingspeak and matlab for the data storage and predictive analysis.

I now want to revisit this proof of concept with the original data and apply the knowledge we learnt here to see what results I can get from the original data gathered.

In matlab, we used a supervised neural network model was implemented with great success. The original training success rate is not available anymore, but from memory it was over 80%.

We now want to do the real work and test different neural network models step-by-step to see if we can do this in python with equal or better success than the "black box" models available to matlab.

A successful outcome of this exercise will be to;

1. Clearly show the conclusions to each stage of creating our model for future review for possible improvements on our data cleaning, feature engineering, model selection.
2. Choose which neural network model can best predict the outcome
3. Analyse whether our neural network models can achieve the same success as the matlab models (> 80%).

Within these objectives we will choose the following neural network techniques which suit this type of machine learning data analysis;

1. Simple One hidden layer
2. Two hidden layers
3. RNN
4. LTSM

Even though the proof of concept worked well and produced reliable results, this was done with a "black-box" matlab option. With the python step-by-step we feel this exercise should produce good results, though there are some concerns we have in implementation;

1. Originally we did not do data cleaning or feature engineering. The performance was good with just raw data, so it will be interesting to see the data with new insights into the data cleaning/feature engineering. We don't think much will be required beyond normalising the data, but we will have to analyse what we have learnt in this course while implementing this.
2. The data is large (10x the samples used in this course). I am not sure if my computing power will be enough for this particular exercise as I'm only using CPU. If it is we can reduce the data and try training/testing on a subset of the data. For that we will use stratified K-fold to ensure our outcomes are balanced.

2. Brief Description and Summary of the Data Set

We have chosen our own data. The data collection was performed as follows;

- The data was gain from a accelerometer placed on a fan
- Data was collected in the x, y and z axis every 10 milliseconds
- Due to limitations of our subscription we could only push 60 values per second to thingspeak databases, so only the first 0.6 seconds of data was recorded of every second.
- The data was pushed once per second in two columns per axis, thus 6 columns in total. Each column contained 30 data points in comma delimited string. This will need to be conversion in the data cleaning section.
- We have three operating modes (clusters) recorded - OFF, ON, FAULTY (ROTOR IMBALANCE).

The dataset consists of 251,100 observations with the following features;

- Time of recording data set (60 - 80 data points)
- Entry ID
- Operation mode (Off, On, Rotor Imbalance)

- Accelerometer - X Axis
- Accelerometer - Y Axis
- Accelerometer - Z Axis

A full analysis of the raw data with graphs and analytics is given in the Data Exploration Stage in the following chapters.

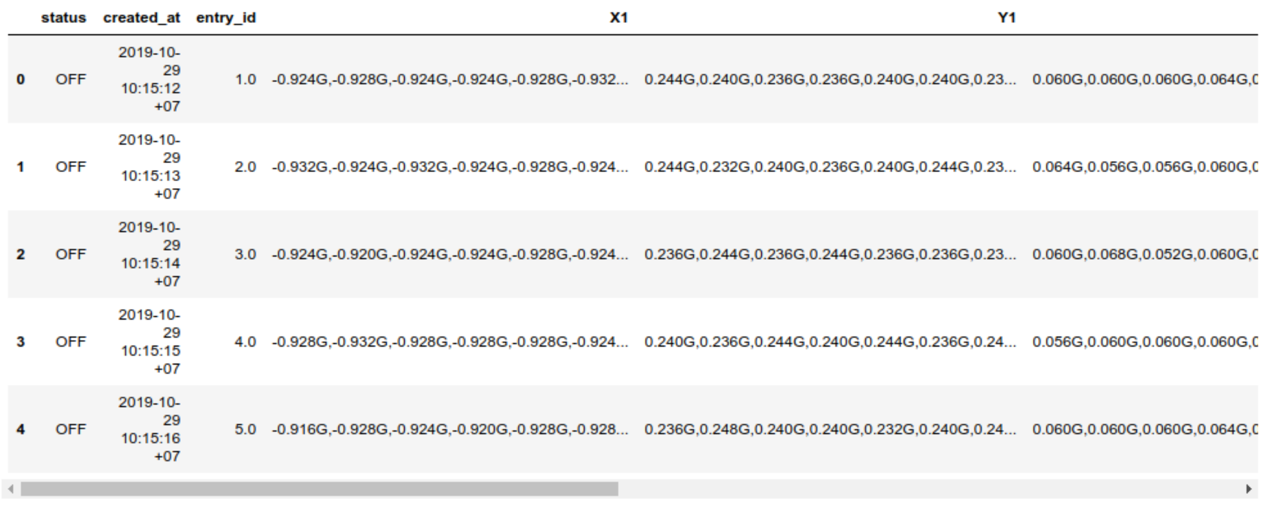
3. Data Exploration (Data Cleaning and Feature Engineering)

List of explorations we performed

In this section we looked at all the standard features;

- Look at the types and how the data has been collected
- Split and reconstruct the data into a dataframe
- Delete any irrelevant columns that are not features
- Check for missing data and quality of the data for each feature
- Delete any features with no value variations throughout all observations
- Checked the correlation of the data features
- Check for skewered data and to decide what to do in this section we looked at all the standard features;

Looking at the raw data;

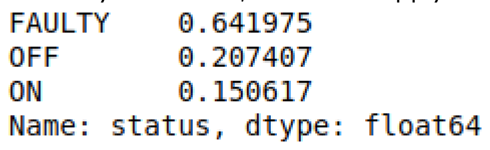


	status	created_at	entry_id	X1	Y1	
0	OFF	2019-10-29 10:15:12 +07	1.0	-0.924G,-0.928G,-0.924G,-0.924G,-0.928G,-0.932...	0.244G,0.240G,0.236G,0.236G,0.240G,0.240G,0.23...	0.060G,0.060G,0.060G,0.064G,
1	OFF	2019-10-29 10:15:13 +07	2.0	-0.932G,-0.924G,-0.932G,-0.924G,-0.928G,-0.924...	0.244G,0.232G,0.240G,0.236G,0.240G,0.244G,0.23...	0.064G,0.056G,0.056G,0.060G,
2	OFF	2019-10-29 10:15:14 +07	3.0	-0.924G,-0.920G,-0.924G,-0.924G,-0.928G,-0.924...	0.236G,0.244G,0.236G,0.244G,0.236G,0.236G,0.23...	0.060G,0.068G,0.052G,0.060G,
3	OFF	2019-10-29 10:15:15 +07	4.0	-0.928G,-0.932G,-0.928G,-0.928G,-0.928G,-0.924...	0.240G,0.236G,0.244G,0.240G,0.244G,0.236G,0.24...	0.056G,0.060G,0.060G,0.060G,
4	OFF	2019-10-29 10:15:16 +07	5.0	-0.916G,-0.928G,-0.924G,-0.920G,-0.928G,-0.928...	0.236G,0.248G,0.240G,0.240G,0.232G,0.240G,0.24...	0.060G,0.060G,0.060G,0.064G,

The data downloaded from the thingspeak website is quite dirty with difficult formatting. We needed to do quite some cleaning to get to the real data with status labels. This included;

- Splitting out the 30 values combined together strings in the X1, Y1, Z1, X2, Y2, Z2 columns in lists
- Create new dataframes from the lists
- Expand the status column to reflect the expansion of the 60 points per observation into 60 observations.
- Recombine the data and status into one long dataframe of observations with labels

- The data labels are not evenly distributed, so we will apply stratified kfold where required.



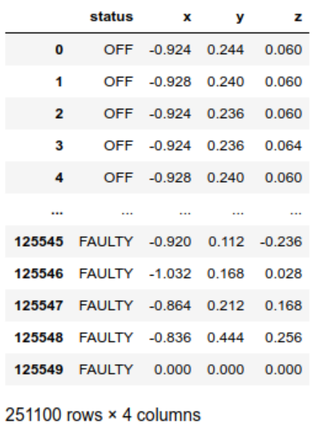
```
FAULTY    0.641975
OFF       0.207407
ON        0.150617
Name: status, dtype: float64
```

During this cleaning we will also delete the unwanted columns created_at and entry_id which has no value in an unsupervised classification model.

We will also remove all the unwanted error observations at the same time.

At the end of this we will have an organized dataset with all relevant data with the following columns;

- status
- x
- y
- x

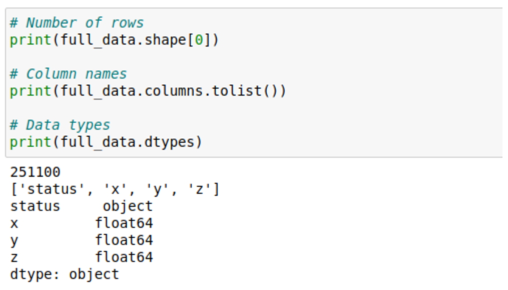


	status	x	y	z
0	OFF	-0.924	0.244	0.060
1	OFF	-0.928	0.240	0.060
2	OFF	-0.924	0.236	0.060
3	OFF	-0.924	0.236	0.064
4	OFF	-0.928	0.240	0.060
...	...	...	...	...
125545	FAULTY	-0.920	0.112	-0.236
125546	FAULTY	-1.032	0.168	0.028
125547	FAULTY	-0.864	0.212	0.168
125548	FAULTY	-0.836	0.444	0.256
125549	FAULTY	0.000	0.000	0.000

251100 rows × 4 columns

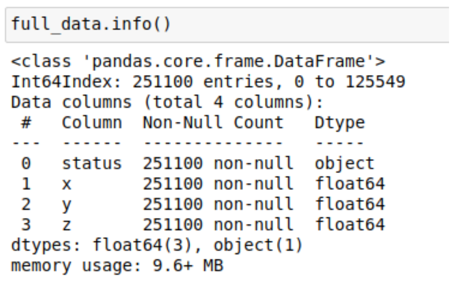
With a better layout of the data in a Pandas DataFrame, we proceed to do some initial analysis on the data and also do some feature engineering. We performed the following on the data;

1. The data was complete with no missing values
2. Two columns (features) was deleted as it did not contain valid feature information - the unique identifier entry_id and created_at timestamp.
3. The DataFrame now looks good and doesn't need further cleaning;



```
# Number of rows
print(full_data.shape[0])

# Column names
print(full_data.columns.tolist())

# Data types
print(full_data.dtypes)
```

```
251100
['status', 'x', 'y', 'z']
status     object
x          float64
y          float64
z          float64
dtype: object
```



```
full_data.info()
```

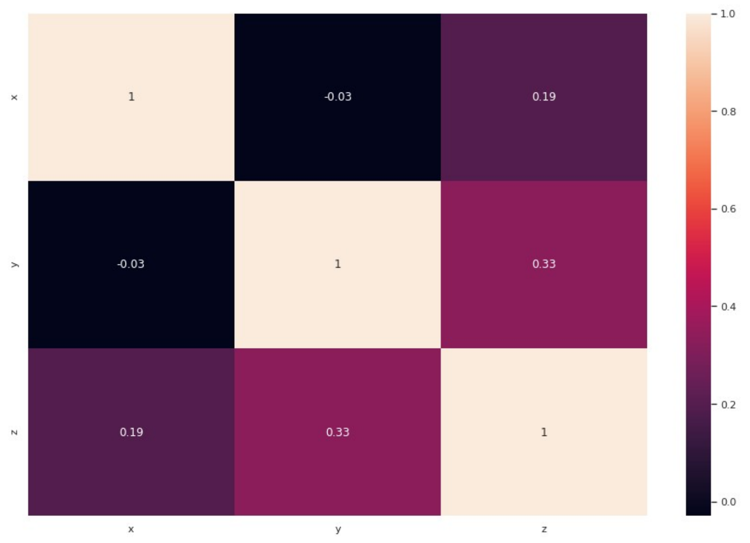
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 251100 entries, 0 to 125549
Data columns (total 4 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   status  251100 non-null  object 
 1   x       251100 non-null  float64
 2   y       251100 non-null  float64
 3   z       251100 non-null  float64
dtypes: float64(3), object(1)
memory usage: 9.6+ MB
```

4. We did not do any scaling as the accelerometer used is all recorded in the same scale already and so all observations are already in the same scale range.

5. Statistical analysis (mean, 25%, range, etc.) was done on all features remaining. All the data looks clean and contains reasonable information as a feature.

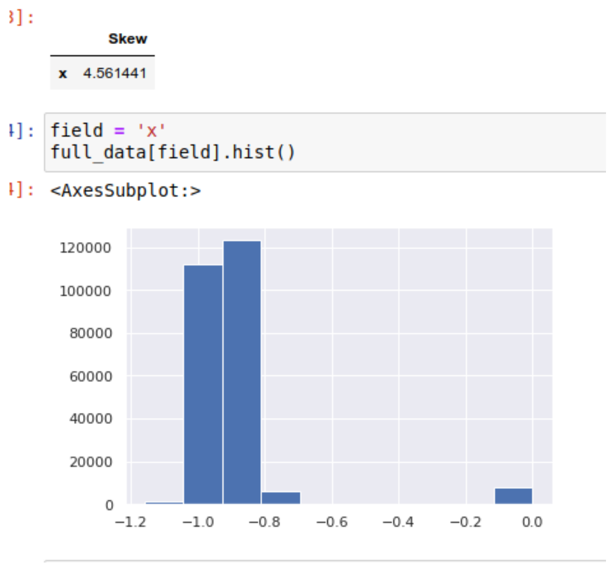
	x	y	z
mean	-0.890780	0.245250	0.039450
25%	-0.948000	0.208000	-0.008000
median	-0.924000	0.240000	0.056000
75%	-0.884000	0.288000	0.084000
min	-1.160000	-0.128000	-0.372000
max	0.000000	0.736000	0.404000
range	1.160000	0.864000	0.776000
std	0.170415	0.091097	0.095839

6. The classes are unbalanced, so we will also use stratified shuffle split in areas that may be useful to use it or when we want to use a smaller data set.



7. The correlation heat map shows a heavy negative correlation between x / y and a positive correlation between y / z .

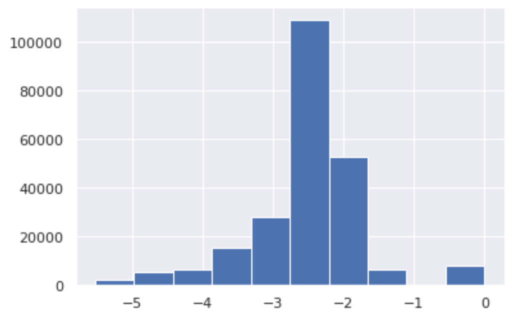
8. The data was heavily skewered in distribution of our x feature. With a log transformation it became a normally distributed data set.



```
]:
     Skew
x  4.561441
```

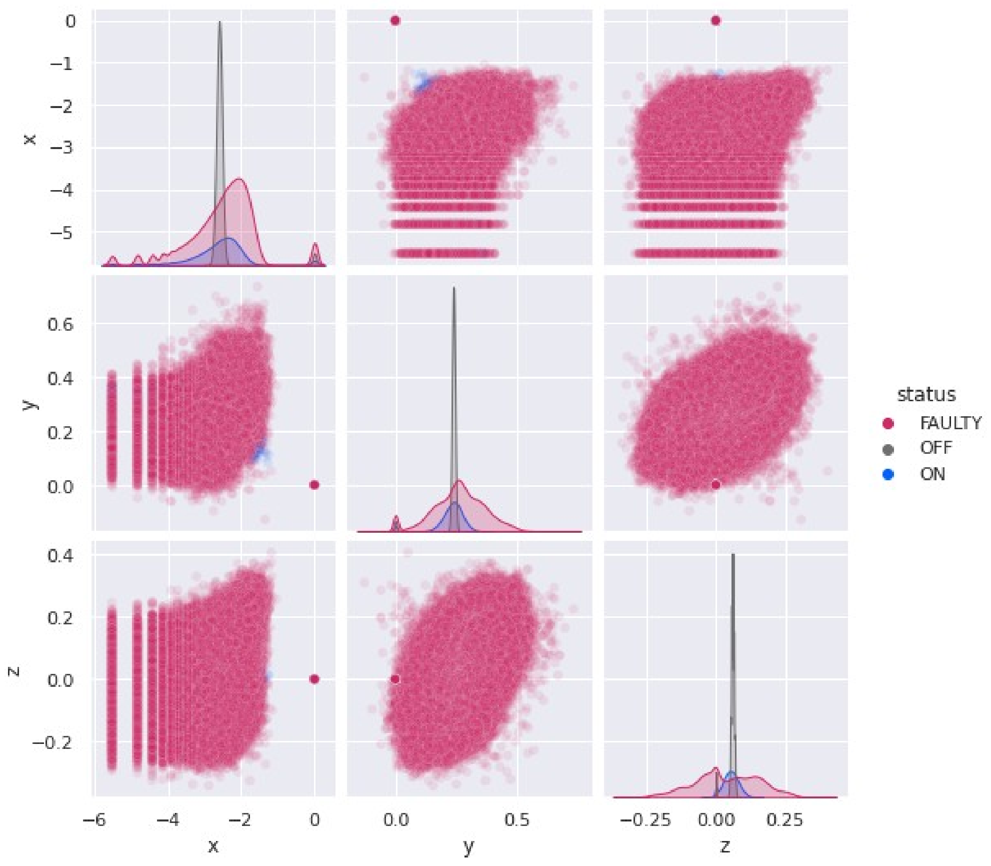
```
]: field = 'x'
   full_data[field].hist()
]: <AxesSubplot:>
```

Before Log Transformation



After Log Transformation

9. The seaborn pairplot already shows a lack of definition between the different clusters. Our optimism to get any good results from unsupervised models is limited.



4. Testing Neural Network Machine Learning - 4 models

We will now perform a number of neural network machine learning models on the data to find out which one performs the best. We will try some hyperparameters as well.

We will do the following;

1. Single Layer Hidden Network
2. Two layer Hidden Network
3. "vanilla" RNN
4. LSTM model

Basic Neural Network with two hidden layers and 6 nodes

If we find some good results, then will will revisit to setup some error parameters to compare against different performing models such as;

1. Graph the trajectory of the loss functions, accuracy on both train and test set
2. Plot the roc curve for the predictions
3. Experiment with different learning rates, numbers of epochs, and network structures

5. Summary of Tests and Final Model

It was a very interesting exercise and we can say we reached our goals as we had two models reach an accuracy of 80% ... just.

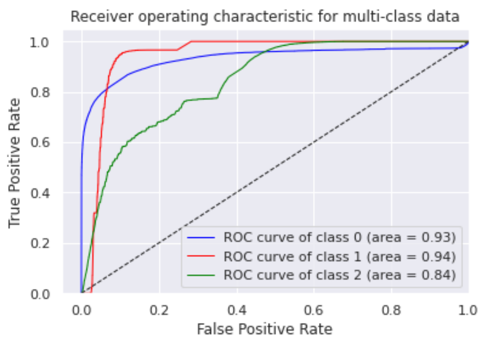
Basic Neural Network with 1 hidden layer each with 12 nodes

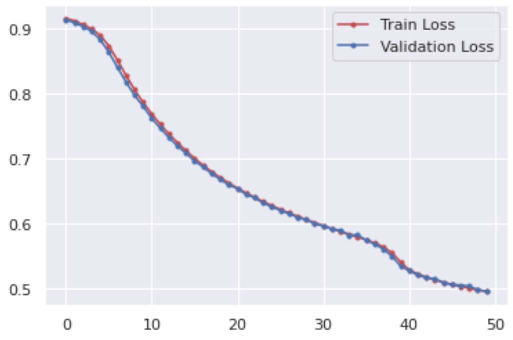
On the first run we found that the significantly large dataset we had of over 250,000 observations was not a problem for this computer. Most likely the small number of features has a significant influence on the speed on these models. Clocking it at 8 seconds for 1 epoch, we decided to try 50 epochs on all models as a base comparison.

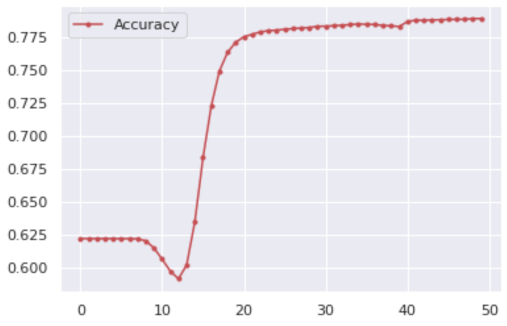
This model came out at a respectable max 79% accuracy with an AOC 0.93/0.94/0.84 over the three classes.

After 50 epoch training span, the loss was still significantly improving thus we would be interested to spend more time training to a higher epoch.

However the accuracy plateaus around 20 epochs and only improves 1 % from 20 to 50 epochs.





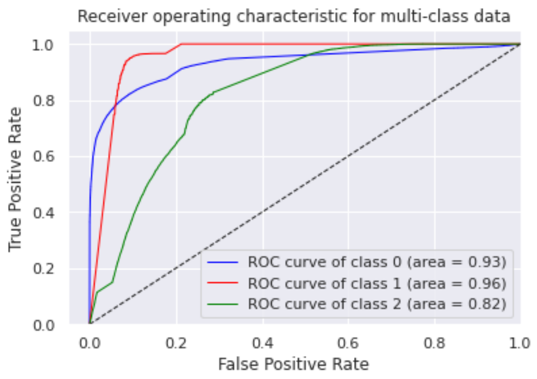


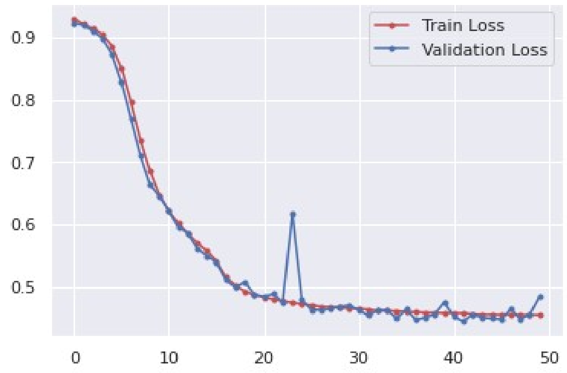
Basic Neural Network with 2 hidden layer each with 6 nodes

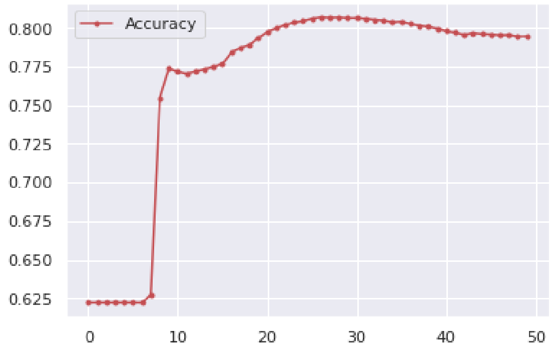
This model reach our goal at just under 81% accuracy with an AOC 0.93/0.96/0.82 over the three classes.

After 40 epoch training span, the loss had plateaued and has very little improvements up to 50 epochs.

However the accuracy plateaus around 25 epochs and actually starts to decrease afterwards.





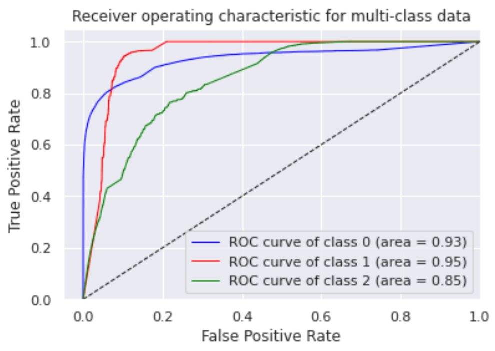


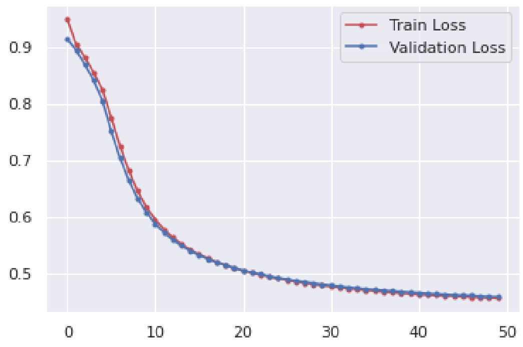
SimpleRNN Model

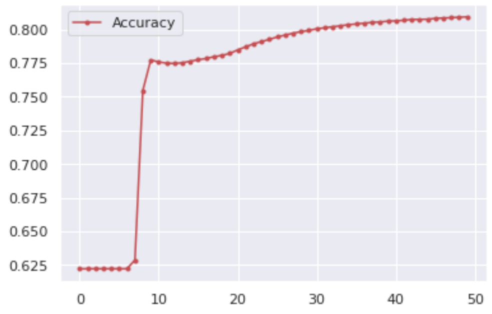
This model reach our goal and was the best of the four at just under 81% accuracy with an AOC 0.93/0.95/0.85 over the three classes.

After 50 epoch training span, the loss seems to have plateaued.

However the accuracy is still improving after 50 epochs. It would be interesting here to increase the number of epochs and sees how much we can improve the accuracy.







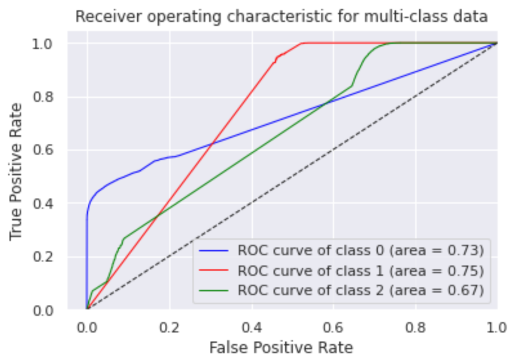
LSTM Model

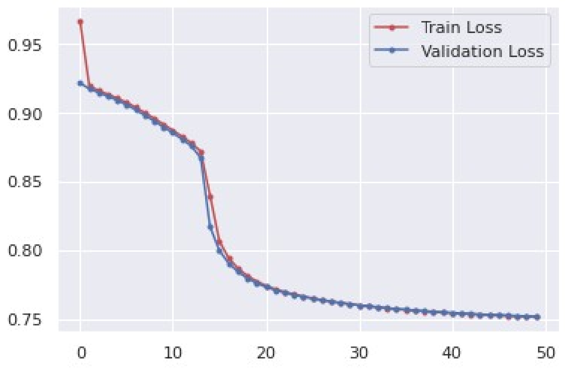
This model was surprisingly the worst and performed very poorly at a flat 62% accuracy with an AOC 0.73/0.75/0.67 over the three classes.

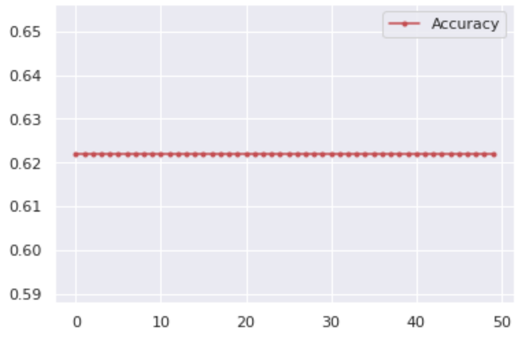
After 40 epoch training span, the loss seems to have plateaued.

However the accuracy never improved starting at around 62% and ending around 62%.

This is very surprising as we felt this is the most promising model which we wanted to work further mainly with the idea to had a self-adjusting historical analysis as the fan motor quality (or bearings) start to wear out and produces some unwanted noise in the system.







RECOMMENDED MODEL : SimpleRNN

6. Summary Key Findings and Insights

The technical findings are;

- The data received was in good condition and only needs to be cleaned of unwanted features and a left skewered x data feature.
- Three models performed within the targets we set at the beginning with two just passing our 80% target.
- The poor performance of the LTSM was very surprising. This could be because of a bad implementation of the model, or wrongly set parameters, so definitely we want to go back and research more on tuning this model.

Some concerns in the model and data would be;

- The data feature engineering, we have only normalised one vector (x). We are not sure if this is correct feature engineering for neural networks and because all three are recorded on the same scale, we should do a normalisation of all data together to keep the aspect ratios equivalent to each other.
- As mentioned above, we suspect that we have given wrong parameters to LTSM. It may not be performing because we haven't set the historical parameters correctly.

Other observations

- We feel there is still a lot of improvements that can be done as hyperparameters where set and not adjusted and many other parameters were not touched from their defaults.

7. Next steps in analyzing this data

With the current models, as we mentioned a lot of work and extra tests can be done with different hyperparameter settings and playing with the many options available.

With the standard neural networks we only went to a 2 hidden / 6 nodes per layer. We could experiment with more complex models to see if that improves the accuracy and aoc.

SimpleRNN is promising and we would like to research further into this model and see if we can improve it with hyperparameters and other settings available.

We also want to investigate why LTSM failed to perform and see if we can get this working or use the GRU as an alternative.

Beyond this, we want to try and move this to an unsupervised model where we can use probabilistic to monitor when a machine deviates from the current sensor readings and thus be considered an anomaly. Over time anomalies can be recorded as ok or identify fault to improve the model. This is why LTSM or GRU would be interesting as part of a final solution.

DEEP LEARNING AND REINFORCEMENT LEARNING - ASSIGNMENT

FULL JUPYTER NOTEBOOK

Peer Group Assignment

June 8, 2021

1 Instructions

Required

Once you have selected a data set, you will produce the deliverables listed below and submit them to one of your peers for review. Treat this exercise as an opportunity to produce analysis that are ready to highlight your analytical skills for a senior audience, for example, the Chief Data Officer, or the Head of Analytics at your company.

Sections required in your report:

- Main objective of the analysis that also specifies whether your model will be focused on a specific type of Deep Learning or Reinforcement Learning algorithm and the benefits that your analysis brings to the business or stakeholders of this data.
- Brief description of the data set you chose, a summary of its attributes, and an outline of what you are trying to accomplish with this analysis.
- Brief summary of data exploration and actions taken for data cleaning or feature engineering.
- Summary of training at least three variations of the Deep Learning model you selected. For example, you can use different clustering techniques or different hyperparameters.
- A paragraph explaining which of your Deep Learning models you recommend as a final model that best fits your needs in terms of accuracy or explainability.
- Summary Key Findings and Insights, which walks your reader through the main findings of your modeling exercise.
- Suggestions for next steps in analyzing this data, which may include suggesting revisiting this model or adding specific data features to achieve a better model.

2 Introduction

As instructed, this report is designed to be presented to the Chief Data Officer or the Head of Analytics at a fictional company (you).

The Jupyter book is given as an attachment but is not important in this assignment.

This assignment focuses to answer as instructed by IBM course creators and bring calculated-based conclusions and constructive analytical analysis to a senior audience, i.e. the Chief Data Officer or the Head of Analytics, that will broken into the 7 major topics;

1. Main objective of the analysis that also specifies whether your model will be focused on a specific type of Deep Learning or Reinforcement Learning algorithm and the benefits that your analysis brings to the business or stakeholders of this data.

2. Brief description of the data set you chose, a summary of its attributes, and an outline of what you are trying to accomplish with this analysis.
3. Brief summary of data exploration and actions taken for data cleaning or feature engineering.
4. Summary of training at least three variations of the Deep Learning model you selected. For example, you can use different clustering techniques or different hyperparameters.
5. A paragraph explaining which of your Deep Learning models you recommend as a final model that best fits your needs in terms of accuracy or explainability.
6. Summary Key Findings and Insights, which walks your reader through the main findings of your modeling exercise.
7. Suggestions for next steps in analyzing this data, which may include suggesting revisiting this model or adding specific data features to achieve a better model.

3 1. Main Objectives

Two years ago I did a proof of concept in predicting whether a fan was running, stopped or imbalanced (dust on blades). I went further to predict two different types of problems - imbalanced fan and airflow blocked. In this proof of concept I used and accelerometer placed on a fan and thingspeak and matlab for the data storage and predictive analysis.

I now want to revisit this proof of concept with the original data and apply the knowledge we learnt here to see what results I can get from the original data gathered.

In matlab, we used a supervised nueral network model was implemented with great success. The original training success rate is not available anymore, but from memory it was over 80%.

We now want to do the real work and test different neural network models step-by-step to see if we can do this in python with equal or better success than the "black box" models available to matlab.

A successful outcome of this exercise will be to;

1. Clearly show the conclusions to each stage of creating our model for future review for possible improvements on our data cleaning, feature engineering, model selection.
2. Choose which neural network model can best predict the outcome
3. Analyse whether out neural network models can achieve the same success as the matlab models ($> 80\%$).

Within these objectives we will choose the following neural network techniques;

1. Simple One hidden layer
2. Two hidden layers
3. RNN
4. LTSM

Even though the proof of concept worked well and produced reliable results, this was done with a "black-box" matlab option. With the python step-by-step we feel this exercise should produce good results, though there are some concerns we have in implementation;

1. Originally we did not do data cleaning or feature engineering. The performance was good with just raw data, so it will be interesting to see the data with new insights into the data cleaning/feature engineering. We don't think much will be required beyong normalising the data, but we will have to analyse what we have learnt in this course while implementing this.

2. The data is large (10x the samples used in this course). I am not sure if my computing power will be enough for this particular exercise as I'm only using CPU. If it is we can reduce the data and try training/tesing ona subset of the data. For that we will use stratified K-fold to ensure our outcomes are balanced.

4 2. Brief Description and Summary of the Data Set

We have chosen our own data. The data collection was performed as follows;

- The data was gain from a accelerometer placed on a fan
- Data was collected in the x, y and z axis every 10 milliseconds
- Due to limitations of our subscription we could only push 60 values per second to thingspeak databases, so only the first 0.6 seconds of data was recorded of every second.
- The data was pushed once per second in two columns per axis, thus 6 columns in total. Each column contained 30 data points in comma delimited string. This will need to be conversion in the data cleaning section.
- We have three operating modes (clusters) recorded - OFF, ON, FAULTY (ROTOR IMBALANCE).

The dataset consists of 251,100 observations with the following features;

- Time of recording data set (60 - 80 data points)
- Entry ID
- Operation mode (Off, On, Rotor Imbalance)
- Accelerometer - X Axis
- Accelerometer - Y Axis
- Accelerometer - Z Axis

A full analysis of the raw data with graphs and analytics is given in the Data Exploration Stage in the following chapters.

5 3. Data Exploration (Data Cleaning and Feature Engineering)

List of explorations we performed In this section we looked at all the standard features;

- Look at the types and how the data has been collected
- Split and reconstruct the data into a dataframe
- Delete any irrelevent columns that are not features
- Check for missing data and quality of the data for each feature
- Delete any faetures with no value variations throughout all observations
- Checked the correlation of the data features
- Check for skewered data and to decide what to do

[1]:
```
# Import all required Libraries

# Operating System Related Libraries
import os
import time
from io import StringIO
import re
```

```
# Standard Data Management Libraries
import numpy as np
import pandas as pd

# Data analytics and manipulation libraries
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.preprocessing import MinMaxScaler
from scipy.stats.mstats import normaltest
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import confusion_matrix, precision_recall_curve,␣
 ↪roc_auc_score, roc_curve, accuracy_score
from sklearn.metrics import roc_curve, auc

# Graphical UI
#%pylab inline
#%config InlineBackend.figure_formats = ['retina']
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
from IPython.display import Image
from sklearn.tree import export_graphviz
from sklearn.preprocessing import label_binarize
from itertools import cycle

# Neural Networks Models
from keras.models  import Sequential
from keras.layers import Input, Dense, Flatten, Dropout, BatchNormalization,␣
 ↪Embedding, LSTM
from keras.optimizers import Adam, SGD, RMSprop
from keras.layers import SimpleRNN
from keras import initializers
import keras
```

```
[2]: filepath = "data/DatasetforAssignment.csv"
     data = pd.read_csv(filepath)
     data_original = data
     os.chdir('data')
     data.head()
```

```
[2]:   status                created_at  entry_id  \
     0    OFF  2019-10-29 10:15:12 +07       1.0
     1    OFF  2019-10-29 10:15:13 +07       2.0
     2    OFF  2019-10-29 10:15:14 +07       3.0
     3    OFF  2019-10-29 10:15:15 +07       4.0
     4    OFF  2019-10-29 10:15:16 +07       5.0
```

```
                                                  X1  \
0  -0.924G,-0.928G,-0.924G,-0.924G,-0.928G,-0.932…
1  -0.932G,-0.924G,-0.932G,-0.924G,-0.928G,-0.924…
2  -0.924G,-0.920G,-0.924G,-0.924G,-0.928G,-0.924…
3  -0.928G,-0.932G,-0.928G,-0.928G,-0.928G,-0.924…
4  -0.916G,-0.928G,-0.924G,-0.920G,-0.928G,-0.928…

                                                  Y1  \
0  0.244G,0.240G,0.236G,0.236G,0.240G,0.240G,0.23…
1  0.244G,0.232G,0.240G,0.236G,0.240G,0.244G,0.23…
2  0.236G,0.244G,0.236G,0.244G,0.236G,0.236G,0.23…
3  0.240G,0.236G,0.244G,0.240G,0.244G,0.236G,0.24…
4  0.236G,0.248G,0.240G,0.240G,0.232G,0.240G,0.24…

                                                  Z1  \
0  0.060G,0.060G,0.060G,0.064G,0.060G,0.060G,0.06…
1  0.064G,0.056G,0.056G,0.060G,0.064G,0.052G,0.05…
2  0.060G,0.068G,0.052G,0.060G,0.064G,0.064G,0.06…
3  0.056G,0.060G,0.060G,0.060G,0.060G,0.064G,0.06…
4  0.060G,0.060G,0.060G,0.064G,0.056G,0.064G,0.06…

                                                  X2  \
0  -0.924G,-0.924G,-0.928G,-0.928G,-0.924G,-0.920…
1  -0.928G,-0.928G,-0.932G,-0.932G,-0.928G,-0.928…
2  -0.928G,-0.932G,-0.924G,-0.924G,-0.936G,-0.928…
3  -0.924G,-0.924G,-0.920G,-0.924G,-0.924G,-0.924…
4  -0.924G,-0.920G,-0.920G,-0.928G,-0.932G,-0.928…

                                                  Y2  \
0  0.240G,0.244G,0.248G,0.240G,0.248G,0.232G,0.23…
1  0.236G,0.244G,0.244G,0.236G,0.228G,0.240G,0.24…
2  0.240G,0.240G,0.236G,0.236G,0.252G,0.236G,0.23…
3  0.240G,0.240G,0.240G,0.240G,0.240G,0.236G,0.24…
4  0.236G,0.232G,0.236G,0.236G,0.240G,0.240G,0.24…

                                                  Z2
0  0.068G,0.056G,0.064G,0.056G,0.064G,0.060G,0.06…
1  0.064G,0.052G,0.056G,0.060G,0.048G,0.052G,0.06…
2  0.056G,0.056G,0.064G,0.060G,0.060G,0.052G,0.06…
3  0.060G,0.048G,0.072G,0.052G,0.056G,0.068G,0.05…
4  0.056G,0.056G,0.052G,0.056G,0.060G,0.060G,0.07…
```

Create a clean copy of the data The data downloaded from the thingspeak website is quite dirty with difficult formatting. We need to do quite some cleaning to get to the real data with status labels.

During this cleaning we will also delete the unwanted columns created_at and entry_id which has no value in an unsupervised classification model.

we will also remove all the unwanted error observations at the same time.

At the end of this we will have an organised dataset with all relevant data with the following columns;

- status
- x
- y
- x

[3]:
```
# Let's convert CSV strings into lists
colX1 = "X1"
colY1 = "Y1"
colZ1 = "Z1"
colX2 = "X2"
colY2 = "Y2"
colZ2 = "Z2"

df = data.assign(**{colX1:data[colX1].str.split(',')})
df = df.assign(**{colY1:df[colY1].str.split(',')})
df = df.assign(**{colZ1:df[colZ1].str.split(',')})
df = df.assign(**{colX2:df[colX2].str.split(',')})
df = df.assign(**{colY2:df[colY2].str.split(',')})
df = df.assign(**{colZ2:df[colZ2].str.split(',')})

df
```

[3]:
```
      status                created_at  entry_id  \
0        OFF  2019-10-29 10:15:12 +07       1.0
1        OFF  2019-10-29 10:15:13 +07       2.0
2        OFF  2019-10-29 10:15:14 +07       3.0
3        OFF  2019-10-29 10:15:15 +07       4.0
4        OFF  2019-10-29 10:15:16 +07       5.0
…        …                        …         …
4049  FAULTY  2019-10-29 12:31:20 +07    4050.0
4050     NaN                       NaN       NaN
4051     OFF                       NaN       NaN
4052      ON                     #REF!       NaN
4053  FAULTY                       NaN       NaN

                                                        X1  \
0     [-0.924G, -0.928G, -0.924G, -0.924G, -0.928G, …
1     [-0.932G, -0.924G, -0.932G, -0.924G, -0.928G, …
2     [-0.924G, -0.920G, -0.924G, -0.924G, -0.928G, …
3     [-0.928G, -0.932G, -0.928G, -0.928G, -0.928G, …
4     [-0.916G, -0.928G, -0.924G, -0.920G, -0.928G, …
…                                                    …
```

```
4049  [-0.916G, -0.784G, -0.872G, -0.876G, -0.992G, …
4050                                                NaN
4051                                                NaN
4052                                                NaN
4053                                                NaN

                                                     Y1  \
0     [0.244G, 0.240G, 0.236G, 0.236G, 0.240G, 0.240…
1     [0.244G, 0.232G, 0.240G, 0.236G, 0.240G, 0.244…
2     [0.236G, 0.244G, 0.236G, 0.244G, 0.236G, 0.236…
3     [0.240G, 0.236G, 0.244G, 0.240G, 0.244G, 0.236…
4     [0.236G, 0.248G, 0.240G, 0.240G, 0.232G, 0.240…
…                                                 …
4049  [0.272G, 0.428G, 0.504G, 0.336G, 0.204G, 0.164…
4050                                                NaN
4051                                                NaN
4052                                                NaN
4053                                                NaN

                                                     Z1  \
0     [0.060G, 0.060G, 0.060G, 0.064G, 0.060G, 0.060…
1     [0.064G, 0.056G, 0.056G, 0.060G, 0.064G, 0.052…
2     [0.060G, 0.068G, 0.052G, 0.060G, 0.064G, 0.064…
3     [0.056G, 0.060G, 0.060G, 0.060G, 0.060G, 0.064…
4     [0.060G, 0.060G, 0.060G, 0.064G, 0.056G, 0.064…
…                                                 …
4049  [0.228G, 0.208G, 0.092G, -0.148G, -0.156G, -0…
4050                                                NaN
4051                                                NaN
4052                                                NaN
4053                                                NaN

                                                     X2  \
0     [-0.924G, -0.924G, -0.928G, -0.928G, -0.924G, …
1     [-0.928G, -0.928G, -0.932G, -0.932G, -0.928G, …
2     [-0.928G, -0.932G, -0.924G, -0.924G, -0.936G, …
3     [-0.924G, -0.924G, -0.920G, -0.924G, -0.924G, …
4     [-0.924G, -0.920G, -0.920G, -0.928G, -0.932G, …
…                                                 …
4049  [-0.964G, -1.020G, -0.904G, -0.816G, -0.856G, …
4050                                                NaN
4051                                                NaN
4052                                                NaN
4053                                                NaN

                                                     Y2  \
0     [0.240G, 0.244G, 0.248G, 0.240G, 0.248G, 0.232…
```

```
1     [0.236G, 0.244G, 0.244G, 0.236G, 0.228G, 0.240…
2     [0.240G, 0.240G, 0.236G, 0.236G, 0.252G, 0.236…
3     [0.240G, 0.240G, 0.240G, 0.240G, 0.240G, 0.236…
4     [0.236G, 0.232G, 0.236G, 0.236G, 0.240G, 0.240…
…                                                 …
4049  [0.156G, 0.104G, 0.200G, 0.436G, 0.484G, 0.364…
4050                                                NaN
4051                                                NaN
4052                                                NaN
4053                                                NaN

                                                     Z2
0     [0.068G, 0.056G, 0.064G, 0.056G, 0.064G, 0.060…
1     [0.064G, 0.052G, 0.056G, 0.060G, 0.048G, 0.052…
2     [0.056G, 0.056G, 0.064G, 0.060G, 0.060G, 0.052…
3     [0.060G, 0.048G, 0.072G, 0.052G, 0.056G, 0.068…
4     [0.056G, 0.056G, 0.052G, 0.056G, 0.060G, 0.060…
…                                                 …
4049  [-0.212G, 0.008G, 0.168G, 0.276G, 0.100G, -0.0…
4050                                                NaN
4051                                                NaN
4052                                                NaN
4053                                                NaN

[4054 rows x 9 columns]
```

[4]:
```
# Delete last supurfluous rows which are just noise from the dataset saving␣
↪format

df.drop(df.tail(4).index,inplace=True)
```

[5]:
```
# Split the data
x1_new = df.X1.apply(pd.Series).stack().reset_index(drop=True)
y1_new = df.Y1.apply(pd.Series).stack().reset_index(drop=True)
z1_new = df.Z1.apply(pd.Series).stack().reset_index(drop=True)
x2_new = df.X2.apply(pd.Series).stack().reset_index(drop=True)
y2_new = df.Y2.apply(pd.Series).stack().reset_index(drop=True)
z2_new = df.Z2.apply(pd.Series).stack().reset_index(drop=True)

# Combine the data
result = pd.concat([x1_new, y1_new, z1_new], axis=1)
result.columns = ["x", "y", "z"]

result_2 = pd.concat([x2_new, y2_new, z2_new], axis=1)
result_2.columns = ["x", "y", "z"]
```

```
#Convert to floats and delete all 0 rows (this are the emptys 31's on the␣
↪original data)
result = result.applymap(lambda x: re.sub(r'[^0-9^\-\.]+', '', x)).replace('',␣
↪np.float64(0)).astype('float64')
result_2 = result_2.applymap(lambda x: re.sub(r'[^0-9^\-\.]+', '', x)).
↪replace('', np.float64(0)).astype('float64')

# create the labels for referencing accuracy later
labels = df["status"]
labels = pd.DataFrame(np.repeat(labels.values,31,axis=0))

# Combine the data
data_1 = pd.concat([labels.reset_index(drop=True), result.
↪reset_index(drop=True)], axis=1)
data_2 = pd.concat([labels.reset_index(drop=True), result_2.
↪reset_index(drop=True)], axis=1)
full_data = pd.concat([data_1.reset_index(drop=True), data_2.
↪reset_index(drop=True)], axis=0)

full_data.columns = ["status", "x", "y", "z"]

#result = result[(result.T != 0).any()]
full_data
```

[5]:
```
        status      x      y      z
0          OFF -0.924  0.244  0.060
1          OFF -0.928  0.240  0.060
2          OFF -0.924  0.236  0.060
3          OFF -0.924  0.236  0.064
4          OFF -0.928  0.240  0.060
…          …      …      …      …
125545  FAULTY -0.920  0.112 -0.236
125546  FAULTY -1.032  0.168  0.028
125547  FAULTY -0.864  0.212  0.168
125548  FAULTY -0.836  0.444  0.256
125549  FAULTY  0.000  0.000  0.000

[251100 rows x 4 columns]
```

[6]:
```
# Number of rows
print(full_data.shape[0])

# Column names
print(full_data.columns.tolist())

# Data types
```

```
print(full_data.dtypes)
```

```
251100
['status', 'x', 'y', 'z']
status     object
x         float64
y         float64
z         float64
dtype: object
```

Check to ensure no missing data Check for missing data to see if any columns are unusable in missing data. Last time we only checked for any rows with only 0 in them.

```
[7]: full_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 251100 entries, 0 to 125549
Data columns (total 4 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   status  251100 non-null  object
 1   x       251100 non-null  float64
 2   y       251100 non-null  float64
 3   z       251100 non-null  float64
dtypes: float64(3), object(1)
memory usage: 9.6+ MB
```

Check scaling Even though not so important, we will check for scaling

```
[8]: data.min().value_counts()
```

```
[8]: 1.0    1
     dtype: int64
```

```
[9]: data.max().value_counts()
```

```
[9]: 4050.0    1
     dtype: int64
```

Check unbalanced classes From this we see that the data is unbalanced, definitely this already shows that some models will work and some will struggle.

```
[10]: full_data['status'].value_counts(normalize=True).sort_index()
```

```
[10]: FAULTY    0.641975
      OFF       0.207407
      ON        0.150617
```

```
Name: status, dtype: float64
```

Check the statistics of the data Check the mean, ranges,max/min and other statistical information

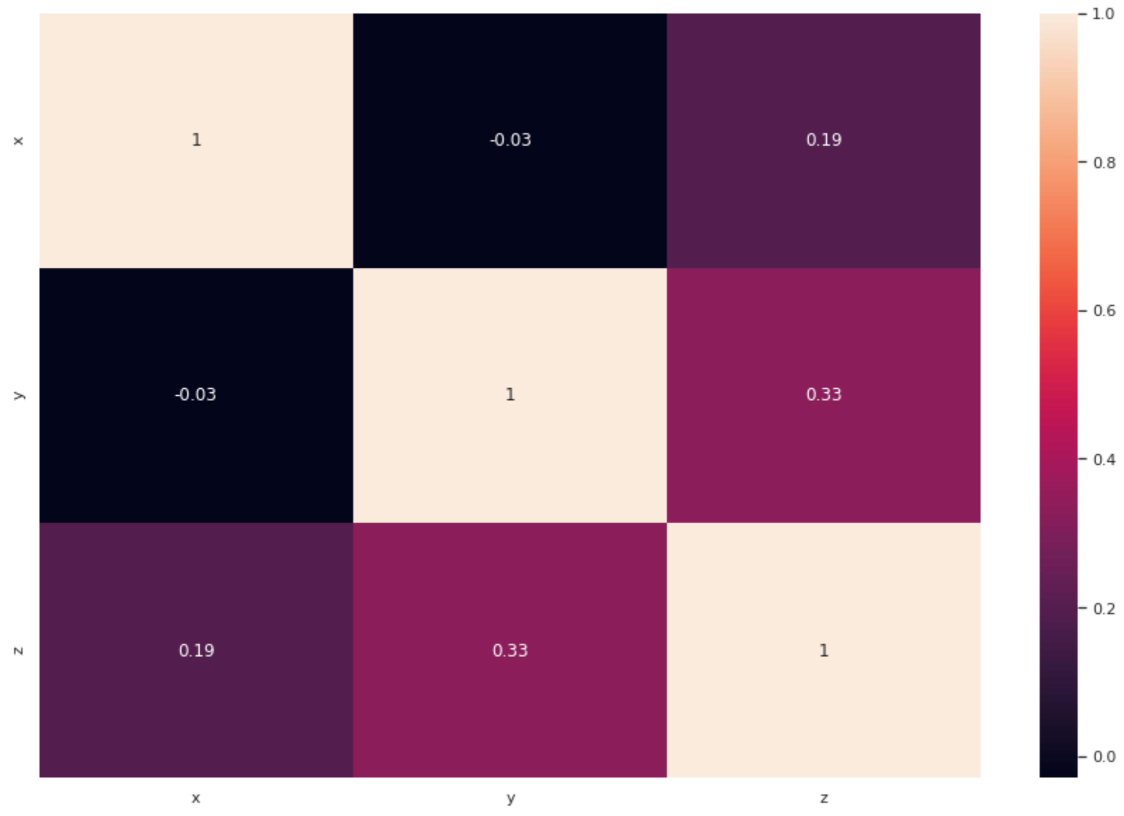
[11]:
```
stats_df = full_data.describe()
stats_df.loc['range'] = stats_df.loc['max'] - stats_df.loc['min']
out_fields = ['mean','25%','50%','75%','min','max','range','std']
stats_df = stats_df.loc[out_fields]
stats_df.rename({'50%': 'median'}, inplace=True)
stats_df
```

[11]:
```
               x         y         z
mean   -0.890780  0.245250  0.039450
25%    -0.948000  0.208000 -0.008000
median -0.924000  0.240000  0.056000
75%    -0.884000  0.288000  0.084000
min    -1.160000 -0.128000 -0.372000
max     0.000000  0.736000  0.404000
range   1.160000  0.864000  0.776000
std     0.170415  0.091097  0.095839
```

Check the correlation between the features For ease of viewing we will use a heat map

[68]:
```
fig, ax = plt.subplots(figsize=(15,10))
sns.heatmap(full_data.corr(),annot=True)
```

[68]:
```
<AxesSubplot:>
```



There is heavily correlation between x and y, while y and z have some correlation. x and z don't seem to have any correlation.

Check for skewered data

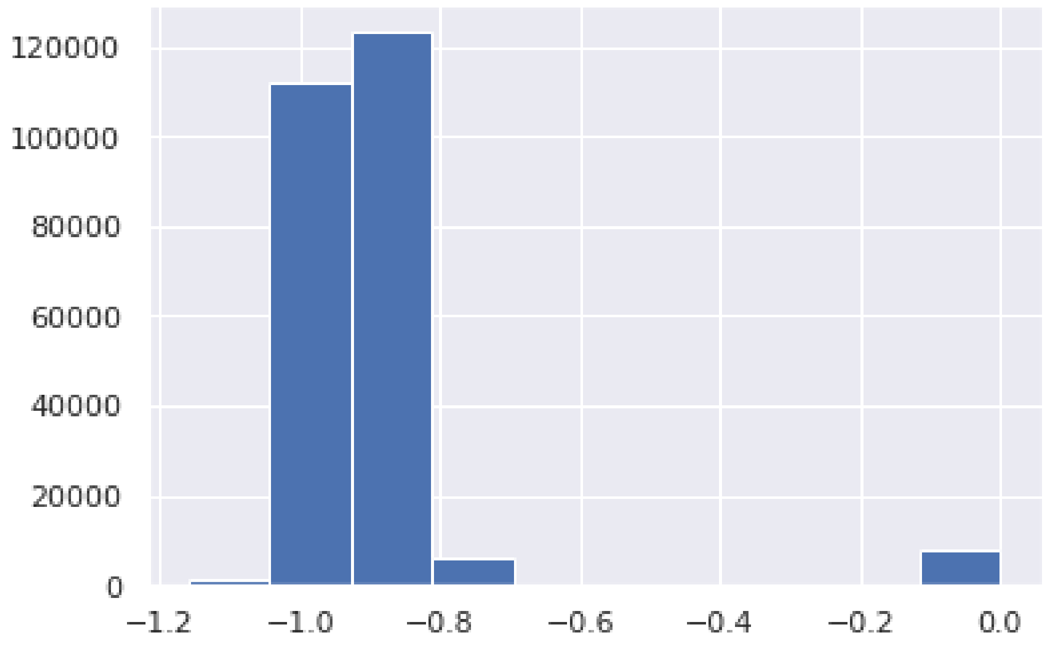
[13]:
```
# Create a list of float colums to check for skewing
skew_check = full_data

mask = skew_check.dtypes == float
mask['x'] = True
mask['y'] = True
mask['z'] = True
float_cols = skew_check.columns[mask]

skew_limit = 0.75 # define a limit above which we will log transform
skew_vals = skew_check[float_cols].skew()

# Showing the skewed columns
skew_cols = (skew_vals
             .sort_values(ascending=False)
             .to_frame()
             .rename(columns={0:'Skew'})
```

```
              .query('abs(Skew) > {}'.format(skew_limit))

skew_cols
```

[13]:
```
       Skew
x  4.561441
```

[14]:
```
field = 'x'
full_data[field].hist()
```

[14]: `<AxesSubplot:>`



[15]:
```
full_data
```

[15]:
```
        status      x      y      z
0          OFF -0.924  0.244  0.060
1          OFF -0.928  0.240  0.060
2          OFF -0.924  0.236  0.060
3          OFF -0.924  0.236  0.064
4          OFF -0.928  0.240  0.060
…         …     …     …     …
125545  FAULTY -0.920  0.112 -0.236
125546  FAULTY -1.032  0.168  0.028
125547  FAULTY -0.864  0.212  0.168
125548  FAULTY -0.836  0.444  0.256
```

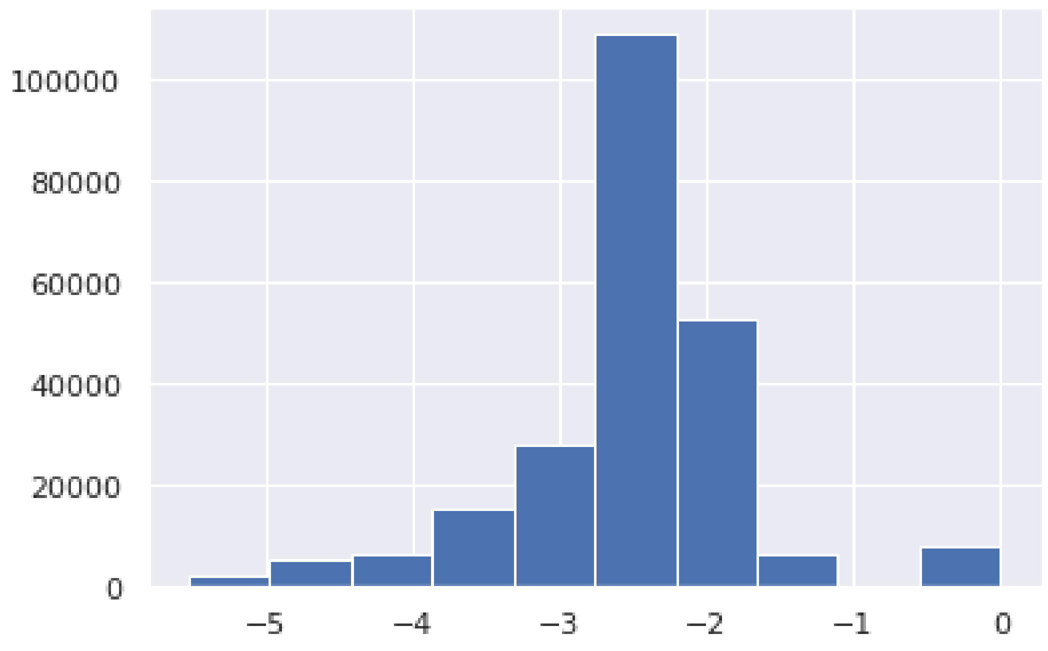
```
125549  FAULTY  0.000  0.000  0.000

[251100 rows x 4 columns]
```

The box cox transformation and square root can not be tested as there are negative numbers, so we can only try to lop1p transformation. SO let's look at the log transformation only

[16]:
```
log1p_field = np.log1p(full_data[field])
log1p_field[np.isfinite(log1p_field)].hist()
```

```
/opt/conda/lib/python3.9/site-packages/pandas/core/arraylike.py:358:
RuntimeWarning: divide by zero encountered in log1p
  result = getattr(ufunc, method)(*inputs, **kwargs)
/opt/conda/lib/python3.9/site-packages/pandas/core/arraylike.py:358:
RuntimeWarning: invalid value encountered in log1p
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

[16]:
```
<AxesSubplot:>
```



[17]:
```
normaltest(log1p_field[np.isfinite(log1p_field)])
```

[17]:
```
NormaltestResult(statistic=18699.981510360078, pvalue=0.0)
```

It works well and the null hypothesis is accepted, so lets change the x field. we also delete the rows that have a NaN in the X after the log transform. This deleted about 16,000 observations.

```
[18]: full_data[field] = log1p_field
```

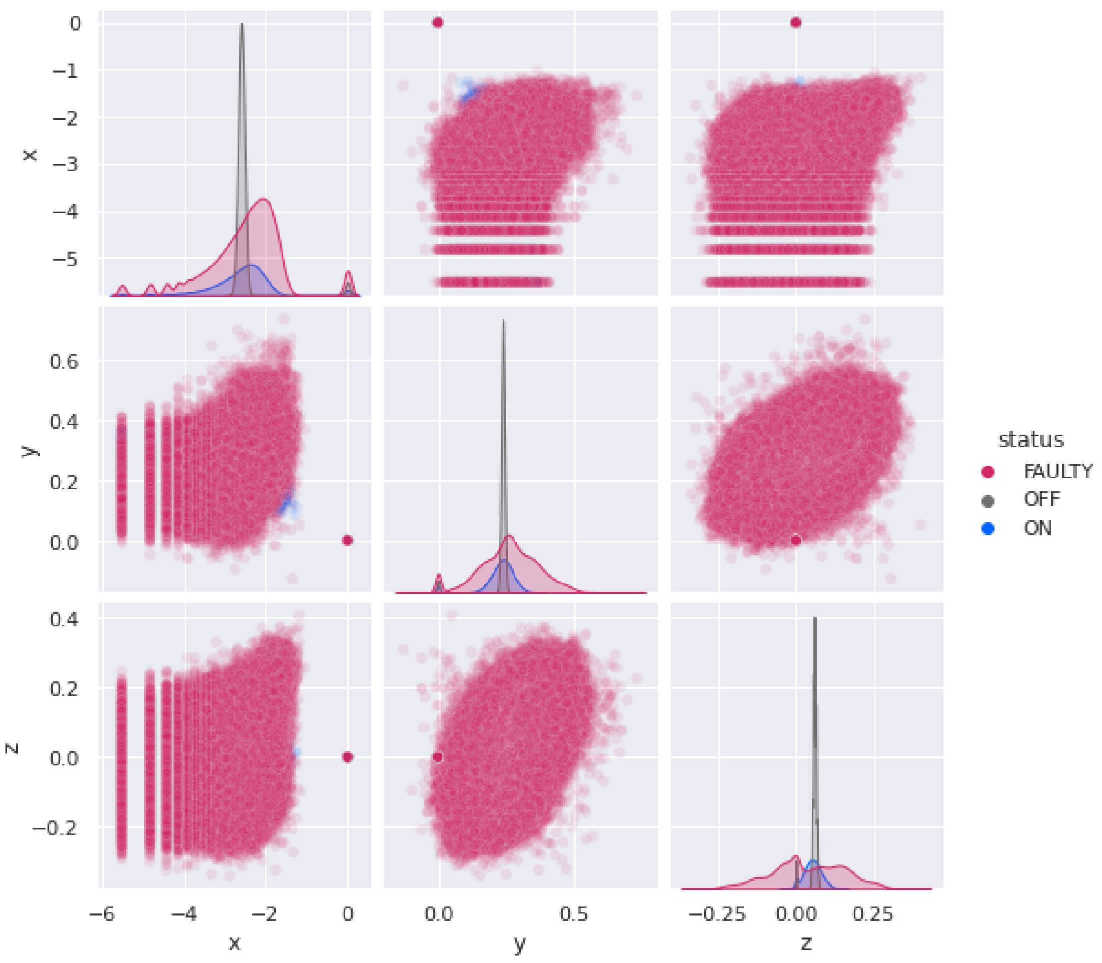
```
[19]: full_data
```

```
[19]:         status         x      y      z
      0          OFF -2.577022  0.244  0.060
      1          OFF -2.631089  0.240  0.060
      2          OFF -2.577022  0.236  0.060
      3          OFF -2.577022  0.236  0.064
      4          OFF -2.631089  0.240  0.060
      …          …        …      …      …
      125545  FAULTY -2.525729  0.112 -0.236
      125546  FAULTY       NaN  0.168  0.028
      125547  FAULTY -1.995100  0.212  0.168
      125548  FAULTY -1.807889  0.444  0.256
      125549  FAULTY  0.000000  0.000  0.000

      [251100 rows x 4 columns]
```

```
[20]: full_data = full_data[np.isfinite(full_data['x'])]
```

Make a pair plot to see everything

```
[21]: from colorsetup import colors, palette

      sns.set_palette(palette)

      # custom colors
      OFF = sns.color_palette()[4]
      ON = sns.color_palette()[0]
      FAULTY = sns.color_palette()[2]

      float_columns = [x for x in full_data.columns if x not in ['status']]

      sns.set_context('notebook')
      sns.pairplot(full_data[float_columns + ['status']],
                   hue='status',
                   hue_order=['FAULTY', 'OFF', 'ON'],
                   palette={'FAULTY':FAULTY, 'OFF':OFF, 'ON':ON}, plot_kws={'alpha':0.
       ↪1});
```



One hot encode the y variable

```
[22]: strat_data = full_data.reset_index(drop=True)

      y = strat_data['status']

      onehotencoder = OneHotEncoder(categories='auto')
      y_onehot = onehotencoder.fit_transform(y.values.reshape(-1,1)).toarray()

      type(y_onehot)
```

```
[22]: numpy.ndarray
```

Split the data with the Machine Failure values being stratified

```
[23]: feature_cols = [x for x in strat_data.columns if x != 'status']

      # Split the data into two parts with 30% in the test data
      # This creates a generator
      strat_shuff_split = StratifiedShuffleSplit(n_splits=1, test_size=0.3,␣
      ↪random_state=42)

      # Get the index values from the generator
      train_idx, test_idx = next(strat_shuff_split.split(strat_data[feature_cols],␣
      ↪y_onehot))

      # Create the data sets
      X_train = strat_data.loc[train_idx, feature_cols]
      y_train = y_onehot[train_idx, :]
      #data_train = pd.concat([y_train.reset_index(drop=True), X_train.
      ↪reset_index(drop=True)], axis=1)

      X_test = strat_data.loc[test_idx, feature_cols]
      y_test = y_onehot[test_idx, :]
      #data_test = pd.concat([y_test.reset_index(drop=True), X_test.
      ↪reset_index(drop=True)], axis=1)
```

Save this cleaned and feature engineered data

[24]:
```
# Split the labels out for comparison only
feature_cols = [x for x in full_data.columns if x != 'status']
labels_data = full_data['status']
unsupervised_data = full_data[feature_cols]

# Save unsupervised data for reuse (labels are only comparisons and don't need)
data_cleaned = unsupervised_data
```

Results of the data cleaning and feature engineering we performed on the data

- The data was complete with no missing values
- The data was downloaded from the thingspeak website and was not ina a readily usable form for running the models. So we had to do a lot of data cleaning to get it ready. This included;
  - Splitting out the 30 values combined together strings in the X1, Y1, Z1, X2, Y2, Z2 columns in lists
  - Create new dataframes from the lists
  - Expand the status column to reflect the expansion of the 60 points per observation into 60 observations.
  - Recombine the data and staus into one long dataframe of observations with labels
  - The data labels are not evenly distributed, so we will apply stratesfied kfold where required.
- We also did some feature engineering on the data to prepare it for the models
  - The x value was left skewered. y and z were normally distributed
  - The x value was applied with a logarithmic filter which made it normally distributed
  - The seborn plot shows some clustering but because of the amount of data we can't see if there will be easy separation of the data.

– x/y and y/z show corrrelation while x/z does not.

6 4. Testing Neural Network Machine Learning - 4 models

We will now perform a number of neural network machine learning models on the data to find out which one performs the best. We will try some hyperparameters as well.

We will do the following;

1. Single Layer Hidden Network
2. Two layer Hidden Network
3. "vanilla" RNN
4. LSTM model

Basic Neural Network with two hidden layers and 6 nodes

If we find some good results, then will will revisit to setup some error parameters to compare against different performing models such as;

1. Graph the trajectory of the loss functions, accuracy on both train and test set
2. Plot the roc curve for the predictions
3. Experiment with different learning rates, numbers of epochs, and network structures

6.0.1 Declarations and Prepare

[25]:
```
# Refresh the data
data = data_cleaned.reset_index(drop=True)
data_with_labels = full_data.reset_index(drop=True)

# Prepare saving the output
results = pd.DataFrame()
```

Function for plotting ROC and AUC

[26]:
```
def plot_roc(y_test, y_pred, model_name):
    fpr, tpr, thr = roc_curve(y_test, y_pred)
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.plot(fpr, tpr, 'k-')
    ax.plot([0, 1], [0, 1], 'k--', linewidth=.5)  # roc curve for random model
    ax.grid(True)
    ax.set(title='ROC Curve for {} on PIMA diabetes problem'.format(model_name),
           xlim=[-0.01, 1.01], ylim=[-0.01, 1.01])
```

6.1 Basic Neural Network with 1 hidden layer each with 12 nodes

We will run a standard relu activation neural network and see how the initial results are with 1 epoch, especially the run time so we can start to feel how long it will take on the more complex models.

[27]:
```
model_1 = Sequential()
model_1.add(Dense(12,input_shape = (3,),activation = 'relu'))
model_1.add(Dense(3,activation='softmax'))
```

[28]:
```
model_1.summary()
```

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 12)                48

_________________________________________________________________
dense_1 (Dense)              (None, 3)                 39
=================================================================
Total params: 87
Trainable params: 87
Non-trainable params: 0

_________________________________________________________________
```

[29]:
```
model_1.compile(SGD(lr = .003), "categorical_crossentropy",␣
 ↪metrics=["accuracy"])
run_hist_1 = model_1.fit(X_train, y_train, validation_data=(X_test, y_test),␣
 ↪epochs=1)
```

```
5131/5131 [==============================] - 9s 2ms/step - loss: 0.9456 -
accuracy: 0.5968 - val_loss: 0.9186 - val_accuracy: 0.6221
```

Ok, was 8 seconds, so this is well within acceptable means for a deep dive here. Let continue with 50 epochs which will take about 6 minutes.

[30]:
```
run_hist_1 = model_1.fit(X_train, y_train, validation_data=(X_test, y_test),␣
 ↪epochs=50)
```

```
Epoch 1/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.9159 -
accuracy: 0.6221 - val_loss: 0.9141 - val_accuracy: 0.6221
Epoch 2/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.9122 -
accuracy: 0.6221 - val_loss: 0.9090 - val_accuracy: 0.6221
Epoch 3/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.9064 -
accuracy: 0.6221 - val_loss: 0.9036 - val_accuracy: 0.6221
Epoch 4/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8999 -
accuracy: 0.6221 - val_loss: 0.8960 - val_accuracy: 0.6221
Epoch 5/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8898 -
accuracy: 0.6221 - val_loss: 0.8832 - val_accuracy: 0.6221
```

```
Epoch 6/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8739 -
accuracy: 0.6221 - val_loss: 0.8641 - val_accuracy: 0.6221
Epoch 7/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8520 -
accuracy: 0.6221 - val_loss: 0.8409 - val_accuracy: 0.6221
Epoch 8/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8285 -
accuracy: 0.6219 - val_loss: 0.8181 - val_accuracy: 0.6218
Epoch 9/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8064 -
accuracy: 0.6205 - val_loss: 0.7977 - val_accuracy: 0.6179
Epoch 10/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7867 -
accuracy: 0.6150 - val_loss: 0.7807 - val_accuracy: 0.6030
Epoch 11/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7691 -
accuracy: 0.6065 - val_loss: 0.7626 - val_accuracy: 0.6064
Epoch 12/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7530 -
accuracy: 0.5974 - val_loss: 0.7469 - val_accuracy: 0.5928
Epoch 13/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7384 -
accuracy: 0.5916 - val_loss: 0.7327 - val_accuracy: 0.5878
Epoch 14/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7248 -
accuracy: 0.6021 - val_loss: 0.7196 - val_accuracy: 0.5877
Epoch 15/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7123 -
accuracy: 0.6349 - val_loss: 0.7085 - val_accuracy: 0.7476
Epoch 16/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7007 -
accuracy: 0.6833 - val_loss: 0.6964 - val_accuracy: 0.7401
Epoch 17/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6899 -
accuracy: 0.7223 - val_loss: 0.6873 - val_accuracy: 0.7178
Epoch 18/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6801 -
accuracy: 0.7490 - val_loss: 0.6767 - val_accuracy: 0.7435
Epoch 19/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6710 -
accuracy: 0.7631 - val_loss: 0.6681 - val_accuracy: 0.7742
Epoch 20/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6625 -
accuracy: 0.7708 - val_loss: 0.6596 - val_accuracy: 0.7755
Epoch 21/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6548 -
accuracy: 0.7750 - val_loss: 0.6541 - val_accuracy: 0.7675
```

```
Epoch 22/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6474 -
accuracy: 0.7768 - val_loss: 0.6450 - val_accuracy: 0.7797
Epoch 23/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6406 -
accuracy: 0.7786 - val_loss: 0.6402 - val_accuracy: 0.7766
Epoch 24/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6341 -
accuracy: 0.7796 - val_loss: 0.6318 - val_accuracy: 0.7797
Epoch 25/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6278 -
accuracy: 0.7800 - val_loss: 0.6261 - val_accuracy: 0.7809
Epoch 26/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6221 -
accuracy: 0.7806 - val_loss: 0.6205 - val_accuracy: 0.7826
Epoch 27/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6167 -
accuracy: 0.7811 - val_loss: 0.6157 - val_accuracy: 0.7856
Epoch 28/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6115 -
accuracy: 0.7817 - val_loss: 0.6097 - val_accuracy: 0.7812
Epoch 29/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6065 -
accuracy: 0.7820 - val_loss: 0.6062 - val_accuracy: 0.7867
Epoch 30/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6017 -
accuracy: 0.7830 - val_loss: 0.6000 - val_accuracy: 0.7819
Epoch 31/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5970 -
accuracy: 0.7830 - val_loss: 0.5963 - val_accuracy: 0.7763
Epoch 32/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5926 -
accuracy: 0.7834 - val_loss: 0.5919 - val_accuracy: 0.7833
Epoch 33/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5881 -
accuracy: 0.7837 - val_loss: 0.5897 - val_accuracy: 0.7894
Epoch 34/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5838 -
accuracy: 0.7842 - val_loss: 0.5824 - val_accuracy: 0.7828
Epoch 35/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5795 -
accuracy: 0.7847 - val_loss: 0.5831 - val_accuracy: 0.7910
Epoch 36/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5752 -
accuracy: 0.7844 - val_loss: 0.5743 - val_accuracy: 0.7885
Epoch 37/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5704 -
accuracy: 0.7843 - val_loss: 0.5686 - val_accuracy: 0.7876
```

```
Epoch 38/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5644 -
accuracy: 0.7835 - val_loss: 0.5610 - val_accuracy: 0.7854
Epoch 39/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5553 -
accuracy: 0.7833 - val_loss: 0.5500 - val_accuracy: 0.7841
Epoch 40/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5414 -
accuracy: 0.7827 - val_loss: 0.5351 - val_accuracy: 0.7877
Epoch 41/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5290 -
accuracy: 0.7865 - val_loss: 0.5277 - val_accuracy: 0.7900
Epoch 42/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5229 -
accuracy: 0.7875 - val_loss: 0.5211 - val_accuracy: 0.7860
Epoch 43/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5178 -
accuracy: 0.7875 - val_loss: 0.5170 - val_accuracy: 0.7879
Epoch 44/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5137 -
accuracy: 0.7878 - val_loss: 0.5153 - val_accuracy: 0.7883
Epoch 45/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5099 -
accuracy: 0.7879 - val_loss: 0.5094 - val_accuracy: 0.7851
Epoch 46/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5066 -
accuracy: 0.7881 - val_loss: 0.5064 - val_accuracy: 0.7888
Epoch 47/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5035 -
accuracy: 0.7882 - val_loss: 0.5056 - val_accuracy: 0.7888
Epoch 48/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5008 -
accuracy: 0.7882 - val_loss: 0.5042 - val_accuracy: 0.7808
Epoch 49/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4981 -
accuracy: 0.7887 - val_loss: 0.4984 - val_accuracy: 0.7862
Epoch 50/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4956 -
accuracy: 0.7888 - val_loss: 0.4965 - val_accuracy: 0.7865
```

[32]:
```
type(run_hist_1)
```

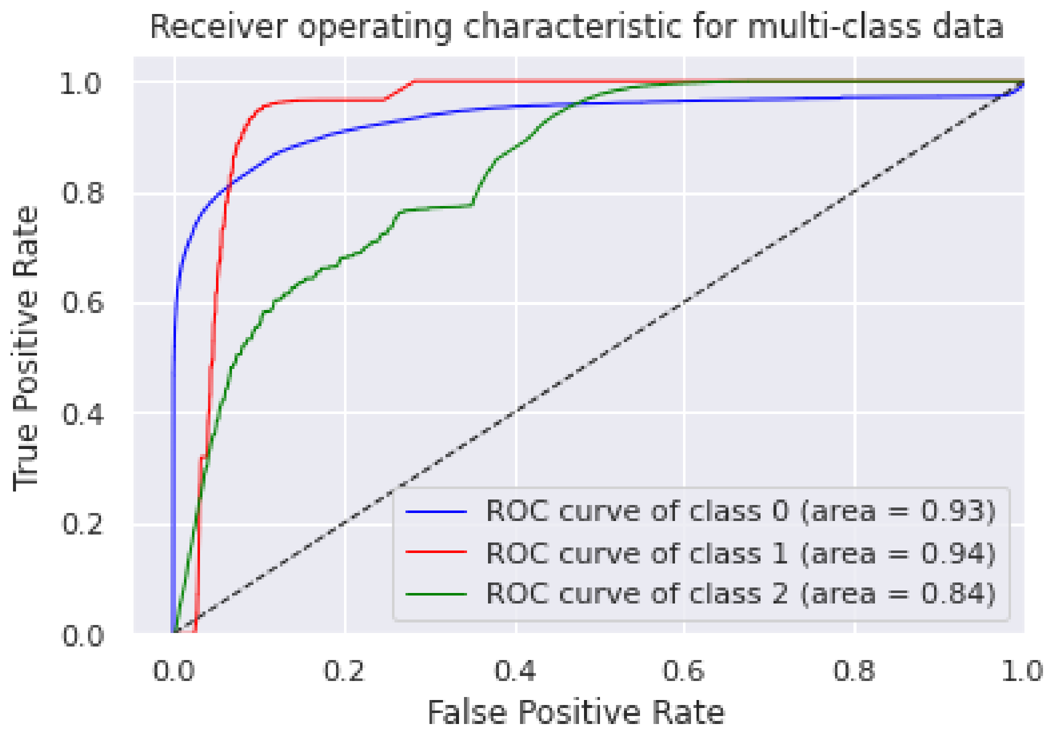
[32]:
```
tensorflow.python.keras.callbacks.History
```

[33]:
```
y_pred_prob_nn_1 = model_1.predict(X_test)
```

[34]:
```
y_pred_prob_nn_1
```

```
[34]: array([[9.99533534e-01, 1.54896483e-07, 4.66364989e-04],
              [9.94818747e-01, 1.10948080e-04, 5.07022999e-03],
              [8.64666045e-01, 1.58792902e-02, 1.19454585e-01],
              …,
              [9.99989867e-01, 6.22858223e-11, 1.01535716e-05],
              [1.22212380e-01, 6.22818768e-01, 2.54968882e-01],
              [1.26315385e-01, 6.16571367e-01, 2.57113338e-01]], dtype=float32)
```

```
[37]: n_classes = y_pred_prob_nn_1.shape[1]

      fpr = dict()
      tpr = dict()
      roc_auc = dict()
      for i in range(y_pred_prob_nn_1.shape[1]):
          fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred_prob_nn_1[:, i])
          roc_auc[i] = auc(fpr[i], tpr[i])
      colors = cycle(['blue', 'red', 'green'])
      for i, color in zip(range(n_classes), colors):
          plt.plot(fpr[i], tpr[i], color=color, lw=1,
                   label='ROC curve of class {0} (area = {1:0.2f})'
                   ''.format(i, roc_auc[i]))
      plt.plot([0, 1], [0, 1], 'k--', lw=1)
      plt.xlim([-0.05, 1.0])
      plt.ylim([0.0, 1.05])
      plt.xlabel('False Positive Rate')
      plt.ylabel('True Positive Rate')
      plt.title('Receiver operating characteristic for multi-class data')
      plt.legend(loc="lower right")
      plt.show()
```

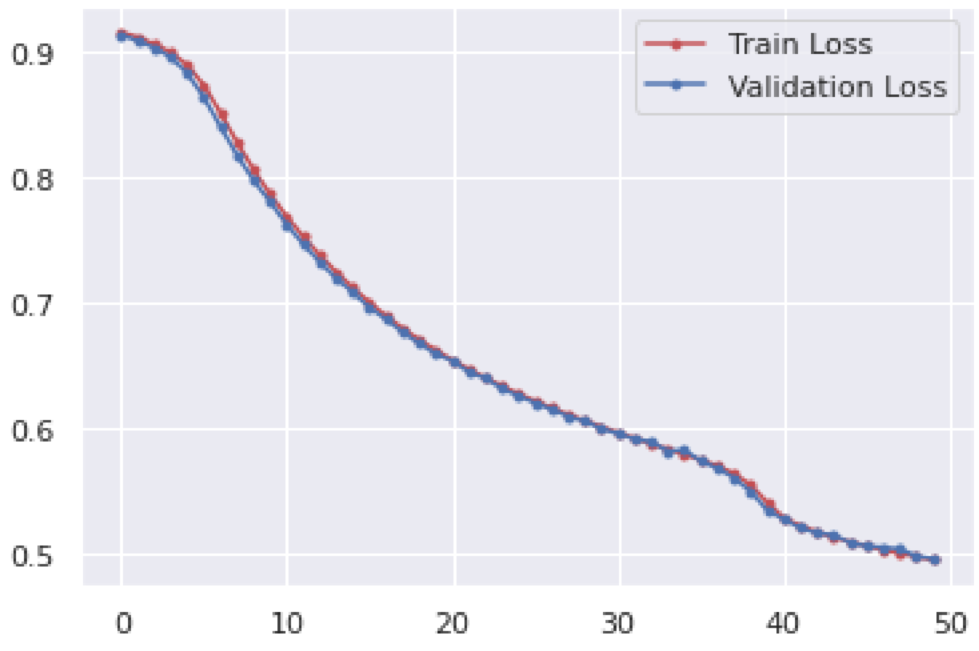


```
[38]: run_hist_1.history.keys()
```

```
[38]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

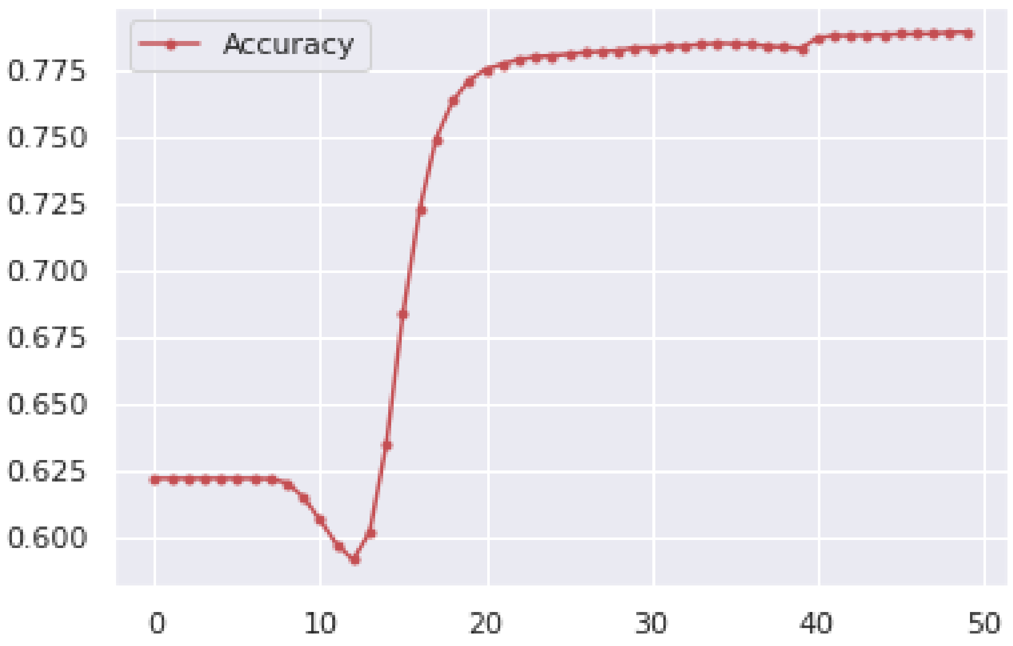
```
[39]: fig, ax = plt.subplots()
      ax.plot(run_hist_1.history["loss"],'r', marker='.', label="Train Loss")
      ax.plot(run_hist_1.history["val_loss"],'b', marker='.', label="Validation Loss")
      ax.legend()
```

```
[39]: <matplotlib.legend.Legend at 0x7f79d46b8160>
```



```
[59]: fig, ax = plt.subplots()
      ax.plot(run_hist_1.history["accuracy"],'r', marker='.', label="Accuracy")
      ax.legend()
```

```
[59]: <matplotlib.legend.Legend at 0x7f79e6807e50>
```



```
[67]: max(run_hist_1.history["accuracy"])
```

```
[67]: 0.788811981678009
```

6.2 Basic Neural Network with 2 hidden layer each with 6 nodes

We will run a standard relu activation neural network and see how the initial results are with 1 epoch, especially the run time so we can start to feel how long it will take on the more complex models.

```
[40]: model_2 = Sequential()
      model_2.add(Dense(6, input_shape=(3,), activation="relu"))
      model_2.add(Dense(6, activation="relu"))
      model_2.add(Dense(3, activation="softmax"))

      model_2.compile(SGD(lr = .003), "categorical_crossentropy",␣
       ↪metrics=["accuracy"])
      run_hist_2 = model_2.fit(X_train, y_train, validation_data=(X_test, y_test),␣
       ↪epochs=50)
```

```
Epoch 1/50
5131/5131 [==============================] - 9s 2ms/step - loss: 0.9337 -
accuracy: 0.6247 - val_loss: 0.9230 - val_accuracy: 0.6221
Epoch 2/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.9209 -
accuracy: 0.6229 - val_loss: 0.9191 - val_accuracy: 0.6221
Epoch 3/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.9166 -
accuracy: 0.6221 - val_loss: 0.9102 - val_accuracy: 0.6221
Epoch 4/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.9057 -
accuracy: 0.6237 - val_loss: 0.8980 - val_accuracy: 0.6221
Epoch 5/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8920 -
accuracy: 0.6224 - val_loss: 0.8730 - val_accuracy: 0.6221
Epoch 6/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8626 -
accuracy: 0.6216 - val_loss: 0.8278 - val_accuracy: 0.6221
Epoch 7/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.8145 -
accuracy: 0.6201 - val_loss: 0.7692 - val_accuracy: 0.6218
Epoch 8/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.7487 -
accuracy: 0.6188 - val_loss: 0.7109 - val_accuracy: 0.6861
Epoch 9/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6983 -
```

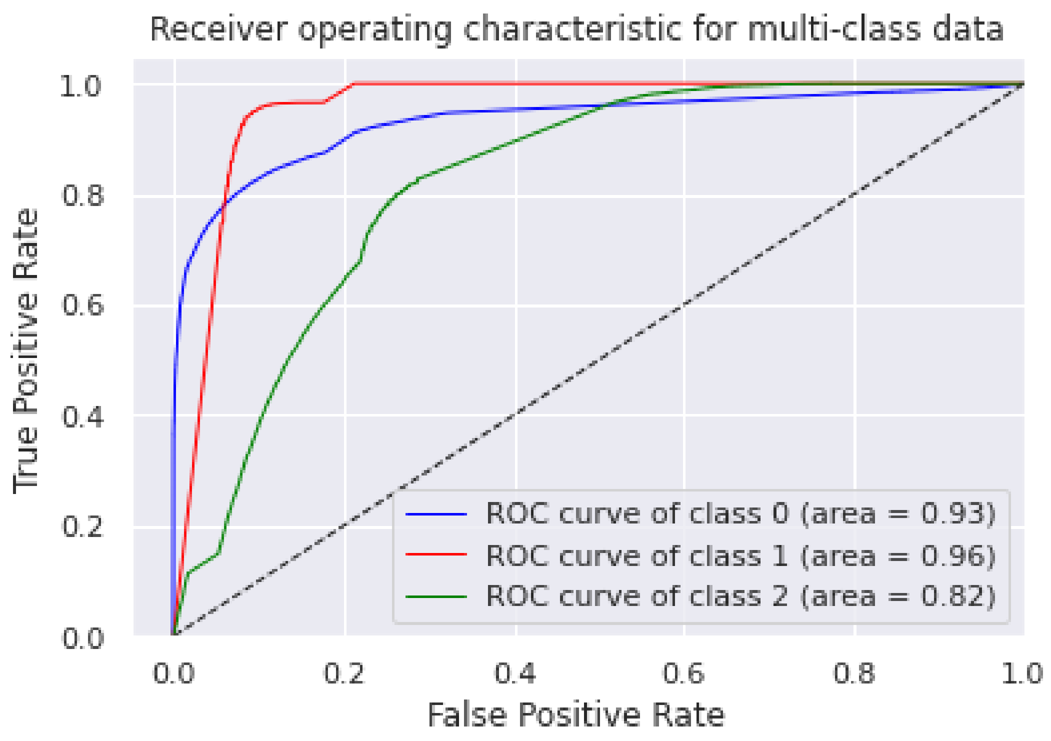
```
accuracy: 0.7333 - val_loss: 0.6635 - val_accuracy: 0.7758
Epoch 10/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6536 -
accuracy: 0.7754 - val_loss: 0.6448 - val_accuracy: 0.7465
Epoch 11/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6278 -
accuracy: 0.7711 - val_loss: 0.6211 - val_accuracy: 0.7607
Epoch 12/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.6062 -
accuracy: 0.7716 - val_loss: 0.5952 - val_accuracy: 0.7636
Epoch 13/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5861 -
accuracy: 0.7736 - val_loss: 0.5854 - val_accuracy: 0.7592
Epoch 14/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5737 -
accuracy: 0.7736 - val_loss: 0.5610 - val_accuracy: 0.7731
Epoch 15/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5604 -
accuracy: 0.7750 - val_loss: 0.5490 - val_accuracy: 0.7733
Epoch 16/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5465 -
accuracy: 0.7770 - val_loss: 0.5388 - val_accuracy: 0.7882
Epoch 17/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5212 -
accuracy: 0.7819 - val_loss: 0.5104 - val_accuracy: 0.7911
Epoch 18/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.5049 -
accuracy: 0.7873 - val_loss: 0.4981 - val_accuracy: 0.7882
Epoch 19/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4966 -
accuracy: 0.7860 - val_loss: 0.5070 - val_accuracy: 0.8004
Epoch 20/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4890 -
accuracy: 0.7920 - val_loss: 0.4872 - val_accuracy: 0.7958
Epoch 21/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4831 -
accuracy: 0.7973 - val_loss: 0.4835 - val_accuracy: 0.8062
Epoch 22/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4757 -
accuracy: 0.8021 - val_loss: 0.4880 - val_accuracy: 0.8085
Epoch 23/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4777 -
accuracy: 0.8019 - val_loss: 0.4744 - val_accuracy: 0.7999
Epoch 24/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4747 -
accuracy: 0.8038 - val_loss: 0.6181 - val_accuracy: 0.6326
Epoch 25/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4721 -
```

```
accuracy: 0.8041 - val_loss: 0.4784 - val_accuracy: 0.7983
Epoch 26/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4710 -
accuracy: 0.8051 - val_loss: 0.4630 - val_accuracy: 0.8050
Epoch 27/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4659 -
accuracy: 0.8077 - val_loss: 0.4621 - val_accuracy: 0.8069
Epoch 28/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4706 -
accuracy: 0.8051 - val_loss: 0.4642 - val_accuracy: 0.8038
Epoch 29/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4646 -
accuracy: 0.8077 - val_loss: 0.4677 - val_accuracy: 0.8017
Epoch 30/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4663 -
accuracy: 0.8064 - val_loss: 0.4695 - val_accuracy: 0.8120
Epoch 31/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4655 -
accuracy: 0.8060 - val_loss: 0.4619 - val_accuracy: 0.8057
Epoch 32/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4624 -
accuracy: 0.8063 - val_loss: 0.4534 - val_accuracy: 0.8141
Epoch 33/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4627 -
accuracy: 0.8052 - val_loss: 0.4621 - val_accuracy: 0.7879
Epoch 34/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4642 -
accuracy: 0.7997 - val_loss: 0.4615 - val_accuracy: 0.8055
Epoch 35/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4600 -
accuracy: 0.8041 - val_loss: 0.4488 - val_accuracy: 0.8115
Epoch 36/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4595 -
accuracy: 0.8050 - val_loss: 0.4633 - val_accuracy: 0.8061
Epoch 37/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4608 -
accuracy: 0.8039 - val_loss: 0.4462 - val_accuracy: 0.8133
Epoch 38/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4572 -
accuracy: 0.8019 - val_loss: 0.4499 - val_accuracy: 0.8144
Epoch 39/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4583 -
accuracy: 0.8020 - val_loss: 0.4556 - val_accuracy: 0.7928
Epoch 40/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4567 -
accuracy: 0.8012 - val_loss: 0.4747 - val_accuracy: 0.7838
Epoch 41/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4567 -
```

```
accuracy: 0.7973 - val_loss: 0.4512 - val_accuracy: 0.8110
Epoch 42/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4584 -
accuracy: 0.7966 - val_loss: 0.4436 - val_accuracy: 0.7977
Epoch 43/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4538 -
accuracy: 0.7972 - val_loss: 0.4553 - val_accuracy: 0.7925
Epoch 44/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4573 -
accuracy: 0.7954 - val_loss: 0.4497 - val_accuracy: 0.7960
Epoch 45/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4556 -
accuracy: 0.7951 - val_loss: 0.4484 - val_accuracy: 0.7965
Epoch 46/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4523 -
accuracy: 0.7975 - val_loss: 0.4474 - val_accuracy: 0.7969
Epoch 47/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4536 -
accuracy: 0.7958 - val_loss: 0.4639 - val_accuracy: 0.7889
Epoch 48/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4544 -
accuracy: 0.7953 - val_loss: 0.4474 - val_accuracy: 0.7963
Epoch 49/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4570 -
accuracy: 0.7930 - val_loss: 0.4549 - val_accuracy: 0.7916
Epoch 50/50
5131/5131 [==============================] - 8s 2ms/step - loss: 0.4559 -
accuracy: 0.7945 - val_loss: 0.4835 - val_accuracy: 0.7814
```

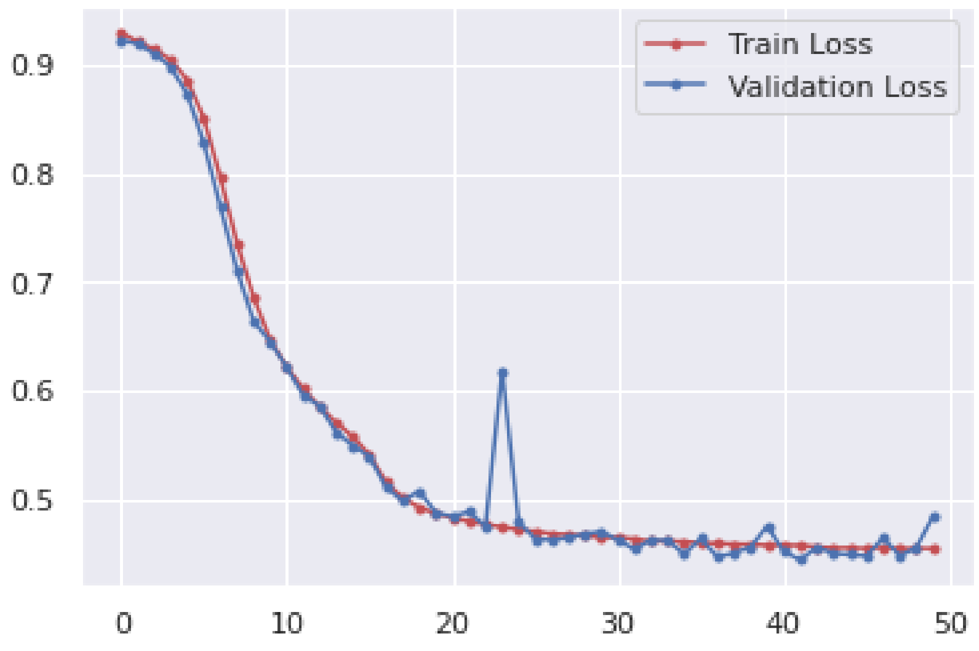
[41]:
```
y_pred_prob_nn_2 = model_2.predict(X_test)
```

[42]:
```
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(y_pred_prob_nn_2.shape[1]):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred_prob_nn_2[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
colors = cycle(['blue', 'red', 'green'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=1,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()
```



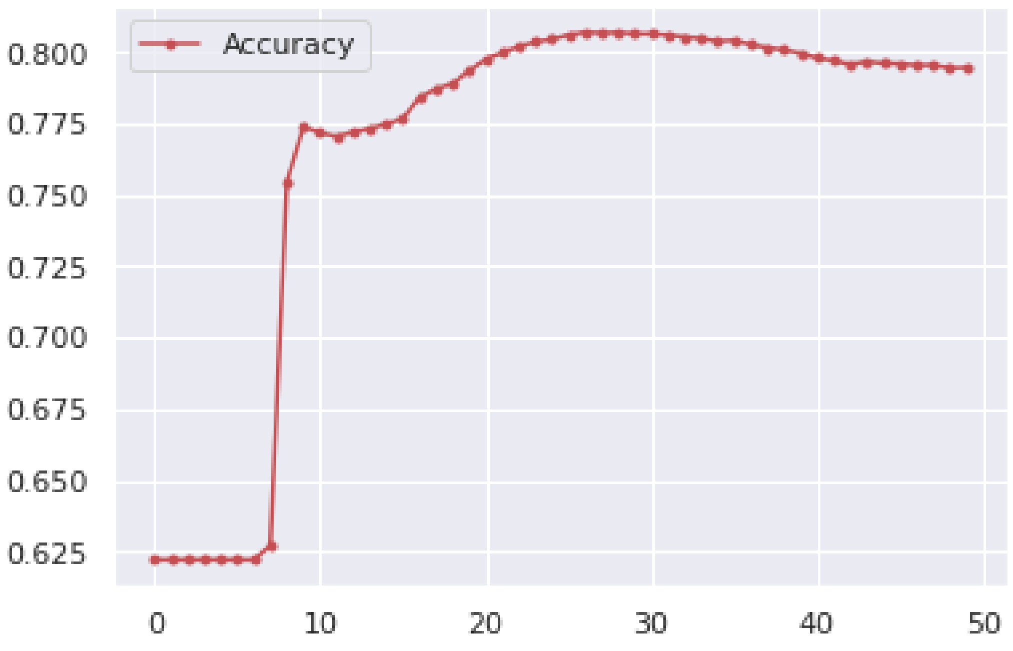
[43]:
```
fig, ax = plt.subplots()
ax.plot(run_hist_2.history["loss"],'r', marker='.', label="Train Loss")
ax.plot(run_hist_2.history["val_loss"],'b', marker='.', label="Validation Loss")
ax.legend()
```

[43]:
```
<matplotlib.legend.Legend at 0x7f79d45ef040>
```



```
[60]: fig, ax = plt.subplots()
      ax.plot(run_hist_2.history["accuracy"],'r', marker='.', label="Accuracy")
      ax.legend()
```

```
[60]: <matplotlib.legend.Legend at 0x7f79d4e20430>
```



```
[66]: max(run_hist_2.history["accuracy"])
```

```
[66]: 0.8071094155311584
```

6.3 SimpleRNN Model

We will train a “vanilla” RNN.

We will need to reshape the X train data first to ensure it fits in the SimpleRNN model.

```
[44]: X_train_reshape=X_train.values.reshape(X_train.shape[0],1,X_train.shape[1])
      X_train_reshape.shape[1:]
```

```
[44]: (1, 3)
```

```
[45]: X_test_reshape=X_test.values.reshape(X_test.shape[0],1,X_test.shape[1])
      X_test_reshape.shape[1:]
```

```
[45]: (1, 3)
```

```
[46]: rnn_hidden_dim = 5
      model_rnn = Sequential()
      model_rnn.add(SimpleRNN(rnn_hidden_dim,
                          kernel_initializer=initializers.RandomNormal(stddev=0.001),
                          recurrent_initializer=initializers.Identity(gain=1.0),
                          activation='relu',
                          input_shape=X_train_reshape.shape[1:]))

      model_rnn.add(Dense(3, activation='softmax'))
```

```
[ ]:
```

```
[47]: rmsprop = keras.optimizers.RMSprop(lr = .0001)
      model_rnn.compile(loss='categorical_crossentropy',
                    optimizer=rmsprop,
                    metrics=['accuracy'])
      model_rnn.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 5)                 45
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 18
=================================================================
Total params: 63
```

```
Trainable params: 63
Non-trainable params: 0
_________________________________________________________________
```

[48]:
```
batch_size = 32
run_hist_3 = model_rnn.fit(X_train_reshape, y_train,
                           batch_size=batch_size,
                           epochs=50,
                           validation_data=(X_test_reshape, y_test))
```

```
Epoch 1/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9968 -
accuracy: 0.6199 - val_loss: 0.9143 - val_accuracy: 0.6221
Epoch 2/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.9111 -
accuracy: 0.6208 - val_loss: 0.8945 - val_accuracy: 0.6221
Epoch 3/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.8878 -
accuracy: 0.6227 - val_loss: 0.8691 - val_accuracy: 0.6221
Epoch 4/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.8612 -
accuracy: 0.6225 - val_loss: 0.8408 - val_accuracy: 0.6221
Epoch 5/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.8334 -
accuracy: 0.6213 - val_loss: 0.8053 - val_accuracy: 0.6221
Epoch 6/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.7890 -
accuracy: 0.6226 - val_loss: 0.7507 - val_accuracy: 0.6221
Epoch 7/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.7377 -
accuracy: 0.6215 - val_loss: 0.7040 - val_accuracy: 0.6221
Epoch 8/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.6913 -
accuracy: 0.6217 - val_loss: 0.6642 - val_accuracy: 0.6700
Epoch 9/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.6561 -
accuracy: 0.7314 - val_loss: 0.6324 - val_accuracy: 0.7758
Epoch 10/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.6241 -
accuracy: 0.7764 - val_loss: 0.6073 - val_accuracy: 0.7755
Epoch 11/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5985 -
accuracy: 0.7778 - val_loss: 0.5878 - val_accuracy: 0.7736
Epoch 12/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5792 -
accuracy: 0.7763 - val_loss: 0.5721 - val_accuracy: 0.7725
Epoch 13/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5668 -
```

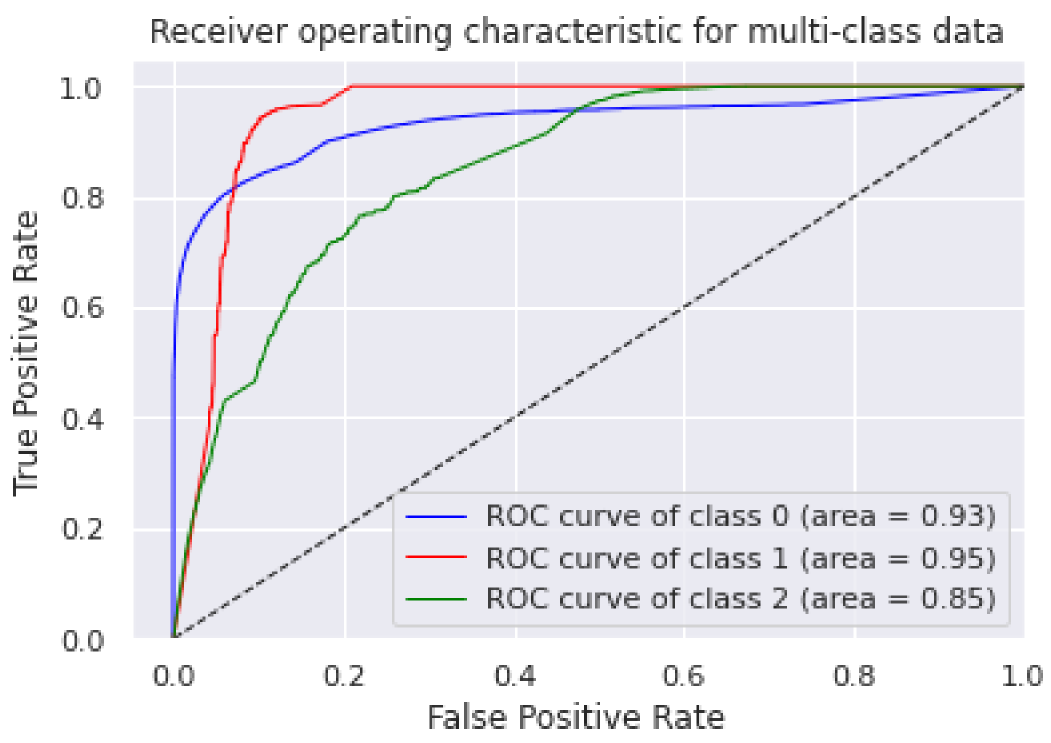
```
accuracy: 0.7741 - val_loss: 0.5590 - val_accuracy: 0.7735
Epoch 14/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5546 -
accuracy: 0.7758 - val_loss: 0.5484 - val_accuracy: 0.7749
Epoch 15/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5434 -
accuracy: 0.7759 - val_loss: 0.5393 - val_accuracy: 0.7749
Epoch 16/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5346 -
accuracy: 0.7769 - val_loss: 0.5316 - val_accuracy: 0.7752
Epoch 17/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5275 -
accuracy: 0.7786 - val_loss: 0.5249 - val_accuracy: 0.7780
Epoch 18/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5228 -
accuracy: 0.7784 - val_loss: 0.5190 - val_accuracy: 0.7777
Epoch 19/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5180 -
accuracy: 0.7790 - val_loss: 0.5139 - val_accuracy: 0.7783
Epoch 20/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5110 -
accuracy: 0.7810 - val_loss: 0.5092 - val_accuracy: 0.7821
Epoch 21/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.5052 -
accuracy: 0.7846 - val_loss: 0.5050 - val_accuracy: 0.7847
Epoch 22/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4997 -
accuracy: 0.7882 - val_loss: 0.5012 - val_accuracy: 0.7872
Epoch 23/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4990 -
accuracy: 0.7879 - val_loss: 0.4979 - val_accuracy: 0.7875
Epoch 24/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4949 -
accuracy: 0.7902 - val_loss: 0.4946 - val_accuracy: 0.7900
Epoch 25/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4918 -
accuracy: 0.7926 - val_loss: 0.4917 - val_accuracy: 0.7915
Epoch 26/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4895 -
accuracy: 0.7928 - val_loss: 0.4892 - val_accuracy: 0.7922
Epoch 27/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4834 -
accuracy: 0.7961 - val_loss: 0.4866 - val_accuracy: 0.7940
Epoch 28/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4833 -
accuracy: 0.7965 - val_loss: 0.4842 - val_accuracy: 0.7953
Epoch 29/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4798 -
```

```
accuracy: 0.7991 - val_loss: 0.4822 - val_accuracy: 0.7969
Epoch 30/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4765 -
accuracy: 0.8011 - val_loss: 0.4801 - val_accuracy: 0.7979
Epoch 31/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4735 -
accuracy: 0.8017 - val_loss: 0.4785 - val_accuracy: 0.7982
Epoch 32/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4751 -
accuracy: 0.8009 - val_loss: 0.4766 - val_accuracy: 0.8001
Epoch 33/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4701 -
accuracy: 0.8043 - val_loss: 0.4750 - val_accuracy: 0.8001
Epoch 34/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4717 -
accuracy: 0.8025 - val_loss: 0.4735 - val_accuracy: 0.8010
Epoch 35/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4695 -
accuracy: 0.8028 - val_loss: 0.4720 - val_accuracy: 0.8019
Epoch 36/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4673 -
accuracy: 0.8045 - val_loss: 0.4707 - val_accuracy: 0.8022
Epoch 37/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4662 -
accuracy: 0.8052 - val_loss: 0.4694 - val_accuracy: 0.8029
Epoch 38/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4658 -
accuracy: 0.8050 - val_loss: 0.4683 - val_accuracy: 0.8031
Epoch 39/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4630 -
accuracy: 0.8062 - val_loss: 0.4672 - val_accuracy: 0.8035
Epoch 40/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4627 -
accuracy: 0.8066 - val_loss: 0.4660 - val_accuracy: 0.8049
Epoch 41/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4634 -
accuracy: 0.8064 - val_loss: 0.4653 - val_accuracy: 0.8045
Epoch 42/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4621 -
accuracy: 0.8066 - val_loss: 0.4641 - val_accuracy: 0.8054
Epoch 43/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4607 -
accuracy: 0.8081 - val_loss: 0.4632 - val_accuracy: 0.8058
Epoch 44/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4607 -
accuracy: 0.8060 - val_loss: 0.4625 - val_accuracy: 0.8058
Epoch 45/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4620 -
```

```
accuracy: 0.8053 - val_loss: 0.4616 - val_accuracy: 0.8070
Epoch 46/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4595 -
accuracy: 0.8076 - val_loss: 0.4609 - val_accuracy: 0.8067
Epoch 47/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4587 -
accuracy: 0.8068 - val_loss: 0.4606 - val_accuracy: 0.8061
Epoch 48/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4556 -
accuracy: 0.8087 - val_loss: 0.4595 - val_accuracy: 0.8080
Epoch 49/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4570 -
accuracy: 0.8085 - val_loss: 0.4588 - val_accuracy: 0.8078
Epoch 50/50
5131/5131 [==============================] - 10s 2ms/step - loss: 0.4574 -
accuracy: 0.8082 - val_loss: 0.4583 - val_accuracy: 0.8088
```

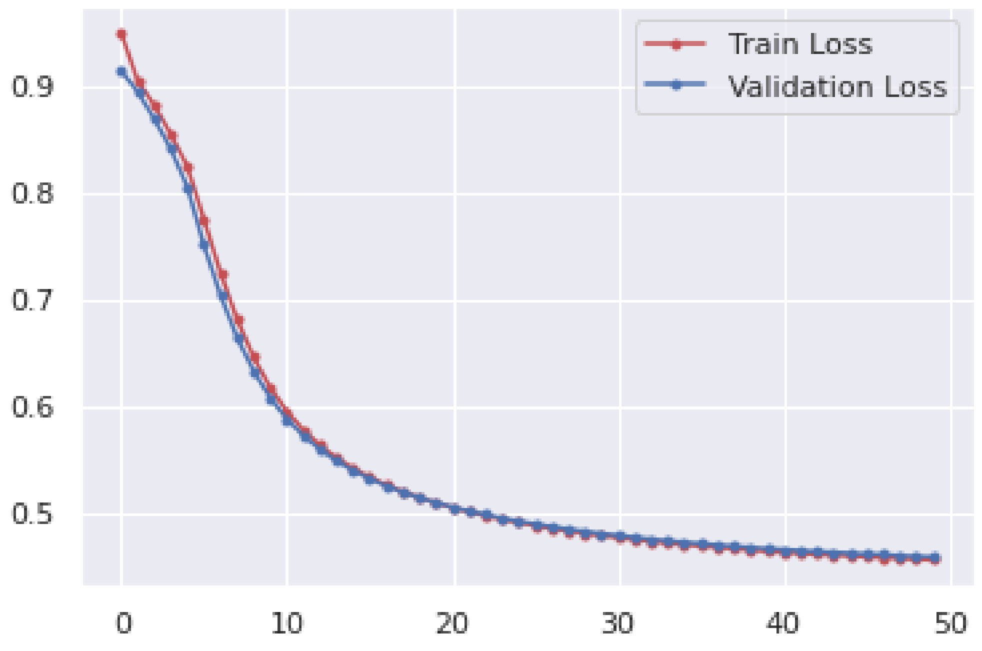
[49]:
```
y_pred_prob_rnn = model_rnn.predict(X_test_reshape)
```

[50]:
```
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(y_pred_prob_rnn.shape[1]):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred_prob_rnn[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
colors = cycle(['blue', 'red', 'green'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=1,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()
```



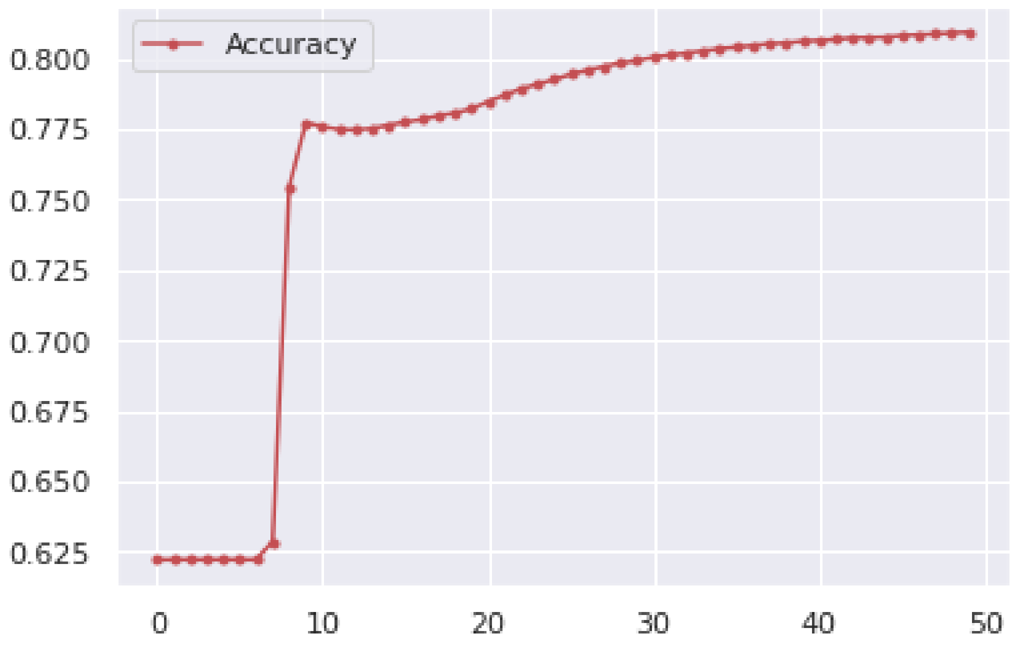
[51]:
```
fig, ax = plt.subplots()
ax.plot(run_hist_3.history["loss"],'r', marker='.', label="Train Loss")
ax.plot(run_hist_3.history["val_loss"],'b', marker='.', label="Validation Loss")
ax.legend()
```

[51]:
```
<matplotlib.legend.Legend at 0x7f79e0926f70>
```



```
[61]: fig, ax = plt.subplots()
      ax.plot(run_hist_3.history["accuracy"],'r', marker='.', label="Accuracy")
      ax.legend()
```

```
[61]: <matplotlib.legend.Legend at 0x7f79d4d8d3d0>
```



```
[65]: max(run_hist_3.history["accuracy"])
```

```
[65]: 0.8094727396965027
```

6.4 LSTM Model

```
[52]: lstm_hidden_dim = 5
      model_lstm = Sequential()
      model_lstm.add(LSTM(lstm_hidden_dim,
                          kernel_initializer=initializers.RandomNormal(stddev=0.001),
                          recurrent_initializer=initializers.Identity(gain=1.0),
                          activation='relu',
                          input_shape=X_train_reshape.shape[1:]))
      model_lstm.add(Dense(3, activation = 'softmax'))
```

```
[53]: model_lstm.summary()
```

```
Model: "sequential_3"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 5)                 180

_________________________________________________________________
dense_6 (Dense)              (None, 3)                 18
=================================================================
Total params: 198
Trainable params: 198
Non-trainable params: 0

_________________________________________________________________
```

```
[54]: rmsprop = keras.optimizers.RMSprop(lr = .0001)
      model_lstm.compile(loss='categorical_crossentropy',
                    optimizer=rmsprop,
                    metrics=['accuracy'])
```

```
[55]: run_hist_4 = model_lstm.fit(X_train_reshape, y_train,
                                  batch_size=batch_size,
                                  epochs=50,
                                  validation_data=(X_test_reshape, y_test))
```

```
Epoch 1/50
5131/5131 [==============================] - 12s 2ms/step - loss: 1.0174 -
accuracy: 0.6223 - val_loss: 0.9217 - val_accuracy: 0.6221
Epoch 2/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9211 -
accuracy: 0.6216 - val_loss: 0.9176 - val_accuracy: 0.6221
```

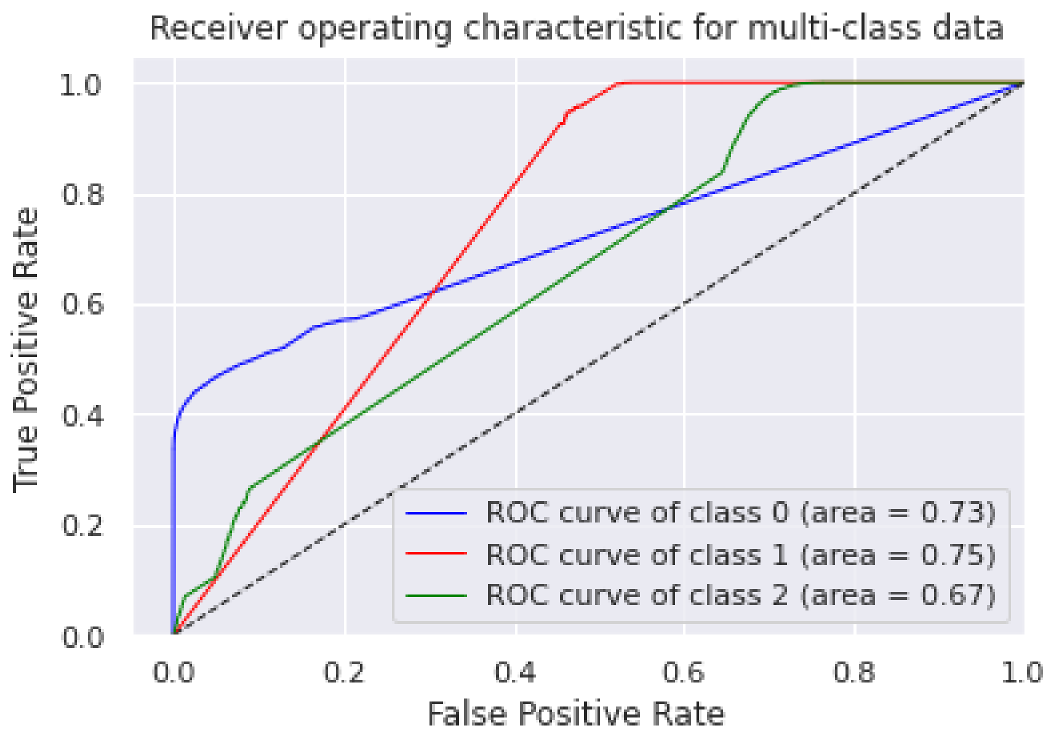
```
Epoch 3/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9190 -
accuracy: 0.6205 - val_loss: 0.9148 - val_accuracy: 0.6221
Epoch 4/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9152 -
accuracy: 0.6214 - val_loss: 0.9121 - val_accuracy: 0.6221
Epoch 5/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9119 -
accuracy: 0.6221 - val_loss: 0.9092 - val_accuracy: 0.6221
Epoch 6/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9079 -
accuracy: 0.6221 - val_loss: 0.9059 - val_accuracy: 0.6221
Epoch 7/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9021 -
accuracy: 0.6243 - val_loss: 0.9023 - val_accuracy: 0.6221
Epoch 8/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.9048 -
accuracy: 0.6193 - val_loss: 0.8983 - val_accuracy: 0.6221
Epoch 9/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8936 -
accuracy: 0.6253 - val_loss: 0.8942 - val_accuracy: 0.6221
Epoch 10/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8922 -
accuracy: 0.6228 - val_loss: 0.8899 - val_accuracy: 0.6221
Epoch 11/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8876 -
accuracy: 0.6228 - val_loss: 0.8856 - val_accuracy: 0.6221
Epoch 12/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8850 -
accuracy: 0.6214 - val_loss: 0.8809 - val_accuracy: 0.6221
Epoch 13/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8792 -
accuracy: 0.6222 - val_loss: 0.8760 - val_accuracy: 0.6221
Epoch 14/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8746 -
accuracy: 0.6218 - val_loss: 0.8674 - val_accuracy: 0.6221
Epoch 15/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8547 -
accuracy: 0.6222 - val_loss: 0.8171 - val_accuracy: 0.6221
Epoch 16/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.8087 -
accuracy: 0.6240 - val_loss: 0.7997 - val_accuracy: 0.6221
Epoch 17/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7965 -
accuracy: 0.6224 - val_loss: 0.7905 - val_accuracy: 0.6221
Epoch 18/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7859 -
accuracy: 0.6234 - val_loss: 0.7843 - val_accuracy: 0.6221
```

```
Epoch 19/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7848 -
accuracy: 0.6203 - val_loss: 0.7796 - val_accuracy: 0.6221
Epoch 20/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7770 -
accuracy: 0.6232 - val_loss: 0.7760 - val_accuracy: 0.6221
Epoch 21/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7770 -
accuracy: 0.6215 - val_loss: 0.7733 - val_accuracy: 0.6221
Epoch 22/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7735 -
accuracy: 0.6211 - val_loss: 0.7710 - val_accuracy: 0.6221
Epoch 23/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7708 -
accuracy: 0.6211 - val_loss: 0.7692 - val_accuracy: 0.6221
Epoch 24/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7692 -
accuracy: 0.6202 - val_loss: 0.7676 - val_accuracy: 0.6221
Epoch 25/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7667 -
accuracy: 0.6227 - val_loss: 0.7662 - val_accuracy: 0.6221
Epoch 26/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7663 -
accuracy: 0.6208 - val_loss: 0.7649 - val_accuracy: 0.6221
Epoch 27/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7633 -
accuracy: 0.6231 - val_loss: 0.7637 - val_accuracy: 0.6221
Epoch 28/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7645 -
accuracy: 0.6210 - val_loss: 0.7627 - val_accuracy: 0.6221
Epoch 29/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7613 -
accuracy: 0.6216 - val_loss: 0.7618 - val_accuracy: 0.6221
Epoch 30/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7606 -
accuracy: 0.6217 - val_loss: 0.7609 - val_accuracy: 0.6221
Epoch 31/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7614 -
accuracy: 0.6208 - val_loss: 0.7601 - val_accuracy: 0.6221
Epoch 32/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7613 -
accuracy: 0.6198 - val_loss: 0.7595 - val_accuracy: 0.6221
Epoch 33/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7555 -
accuracy: 0.6228 - val_loss: 0.7587 - val_accuracy: 0.6221
Epoch 34/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7556 -
accuracy: 0.6235 - val_loss: 0.7580 - val_accuracy: 0.6221
```

```
Epoch 35/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7579 -
accuracy: 0.6217 - val_loss: 0.7574 - val_accuracy: 0.6221
Epoch 36/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7580 -
accuracy: 0.6214 - val_loss: 0.7569 - val_accuracy: 0.6221
Epoch 37/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7571 -
accuracy: 0.6218 - val_loss: 0.7564 - val_accuracy: 0.6221
Epoch 38/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7555 -
accuracy: 0.6220 - val_loss: 0.7559 - val_accuracy: 0.6221
Epoch 39/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7542 -
accuracy: 0.6226 - val_loss: 0.7555 - val_accuracy: 0.6221
Epoch 40/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7557 -
accuracy: 0.6224 - val_loss: 0.7550 - val_accuracy: 0.6221
Epoch 41/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7554 -
accuracy: 0.6222 - val_loss: 0.7546 - val_accuracy: 0.6221
Epoch 42/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7558 -
accuracy: 0.6196 - val_loss: 0.7542 - val_accuracy: 0.6221
Epoch 43/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7542 -
accuracy: 0.6227 - val_loss: 0.7539 - val_accuracy: 0.6221
Epoch 44/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7500 -
accuracy: 0.6254 - val_loss: 0.7535 - val_accuracy: 0.6221
Epoch 45/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7551 -
accuracy: 0.6202 - val_loss: 0.7532 - val_accuracy: 0.6221
Epoch 46/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7527 -
accuracy: 0.6220 - val_loss: 0.7530 - val_accuracy: 0.6221
Epoch 47/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7522 -
accuracy: 0.6217 - val_loss: 0.7527 - val_accuracy: 0.6221
Epoch 48/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7533 -
accuracy: 0.6200 - val_loss: 0.7524 - val_accuracy: 0.6221
Epoch 49/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7499 -
accuracy: 0.6232 - val_loss: 0.7521 - val_accuracy: 0.6221
Epoch 50/50
5131/5131 [==============================] - 11s 2ms/step - loss: 0.7527 -
accuracy: 0.6216 - val_loss: 0.7519 - val_accuracy: 0.6221
```

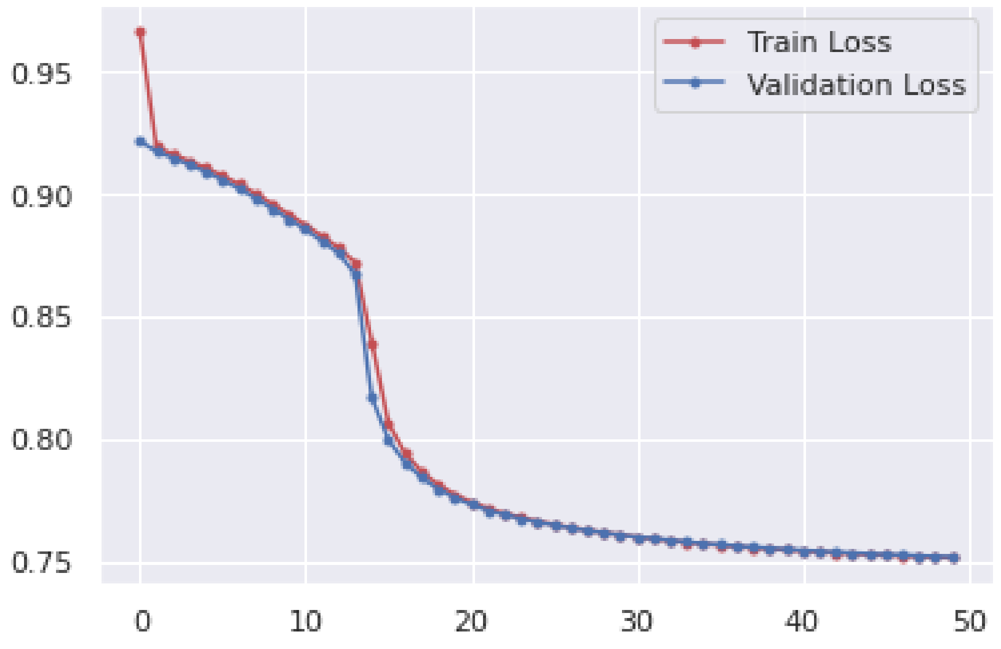
[56]:
```
y_pred_prob_ltsm = model_lstm.predict(X_test_reshape)
```

[57]:
```
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(y_pred_prob_ltsm.shape[1]):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred_prob_ltsm[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
colors = cycle(['blue', 'red', 'green'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=1,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()
```



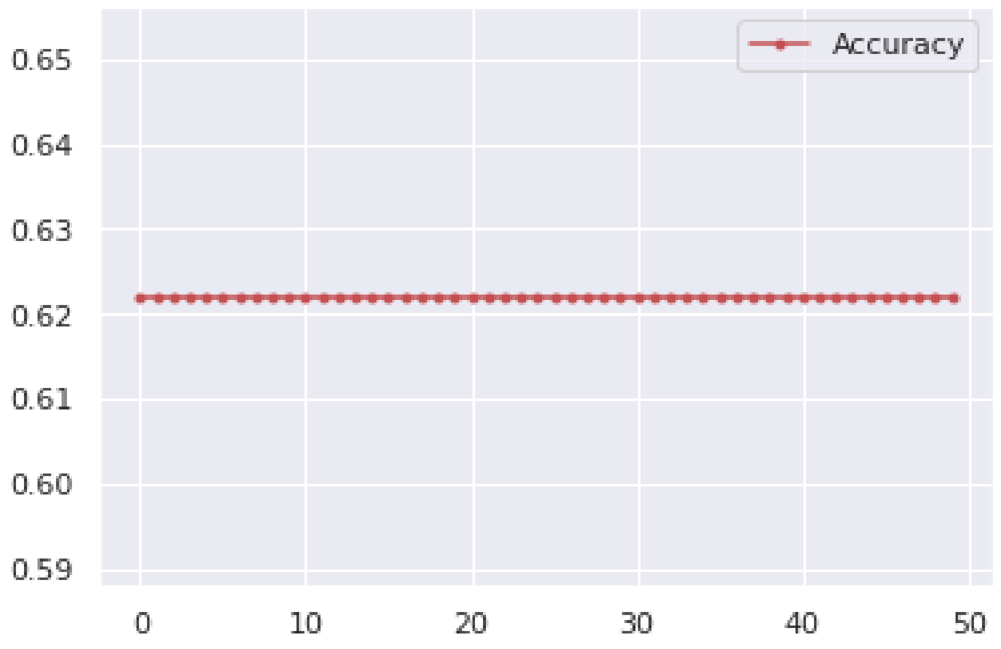
```
[58]: fig, ax = plt.subplots()
      ax.plot(run_hist_4.history["loss"],'r', marker='.', label="Train Loss")
      ax.plot(run_hist_4.history["val_loss"],'b', marker='.', label="Validation Loss")
      ax.legend()
```

```
[58]: <matplotlib.legend.Legend at 0x7f79e1c8f6a0>
```



```
[62]: fig, ax = plt.subplots()
      ax.plot(run_hist_4.history["accuracy"],'r', marker='.', label="Accuracy")
      ax.legend()
```

```
[62]: <matplotlib.legend.Legend at 0x7f79d4d795b0>
```



```
[64]: max(run_hist_4.history["accuracy"])
```

```
[64]: 0.6220945715904236
```

7 Summary of all the models

8 5. Summary of Neural Network Machine Learning Models

It was a very interesting exercise and we can say we reached our goals as we had two models reach an accuracy of 80% … just.

Basic Neural Network with 1 hidden layer each with 12 nodes On the first run we found that the significantly large dataset we had of over 250,000 observations was not a problem for this computer. Mostb likely the small number of features has a significant influence on the speed on these models. Clocking it at 8 seconds for 1 epoch, we decided to try 50 epochs on all models as a base comparison.

This model came out at a respectable max 79% accuracy with an AOC 0.93/0.94/0.84 over the three classes.

After 50 epoch training span, the loss was still significantly improving thus we would be interested to spend more time training to a higher epoch.

However the accuracy plateaus around 20 epochs and only improves 1 % from 20 to 50 epochs.

Basic Neural Network with 2 hidden layer each with 6 nodes This model reach our goal at just under 81% accuracy with an AOC 0.93/0.96/0.82 over the three classes.

After 40 epoch training span, the loss had plateaued and has very little improvements up to 50 epochs.

However the accuracy plateaus around 25 epochs and actually starts to decrease afterwards.

SimpleRNN Model This model reach our goal and was the best of the four at just under 81% accuracy with an AOC 0.93/0.95/0.85 over the three classes.

After 50 epoch training span, the loss seems to have plateaued.

However the accuracy is still improving after 50 epochs. It would be interesting here to increase the number of epochs and sees how muchnwe can improve the acccuracy.

LSTM Model This model was surprisingly the worst and performed very poorly at a flat 62% accuracy with an AOC 0.73/0.75/0.67 over the three classes.

After 40 epoch training span, the loss seems to have plateaued.

However the accuracy never improved starting at around 62% and ending around 62%.

This is very surprising as we felt this is the most promising model which we wanted to work further mainly with the idea to had a self-adjusting historical analysis as the fan motor quality (or bearings) start to wear out and produces some unwanted noise in the system.

`<b>RECOMMENDED MODEL : SimpleRNN</b>`

9 6. Summary Key Findings and Insights

The technical findings are;

- The data received was in good condition and only needs to be cleaned of unwanted features and a left skewered x data feature.
- Three models performed within the targets we set at the beginning with two just passing our 80% target.
- The poor performance of the LTSM was very suprising. This could be because of a bad implementation of the model, or wrongly set pasrameters, so definitely we want to go back and research more on tuning this model.

Some concerns in the model and data would be;

- The data feature engineering, we have only normalised one vextor (x). We are not sure if this is correct feature engineering for neural networks and because all three are recoded on the same scale, we should do a normalisation of all data together to keep the aspect ratios equivalent to each other.
- As mentioned above, we suspect that we have given wrong parameters to LTSM. It may not be performing becuase we haven't set the historical parameters correctly.

Other observations

- We feel there is still a lot of improvements that can be done as hyperparameters where set and not adjusted and many other parameters were not touched fromt their defaults.

10 7. Next steps in analyzing this data

With the current models, as we mentioned a lot of work and extra tests can be done with different hyperparameter settings and playing with the many options available.

With the standard neural networks we only went to a 2 hidden / 6 nodes per layer. We could experiement with more complex models to see if that improves the accuracy and aoc.

SimpleRNN is promising and we would like to research further into this model and see if we can improve it with hyperparameters and other settings available.

We also want to investigate why LTSM failed to perform and se if we can get this working or use the GRU as an alternative.

Beyond this, we want to try and move this to an unsupervised model where we can use probablistics to monitor when a machine deviates from the current sensor readings and thus be considered an amonoly. Over time anomolies can be recorded as ok or identify fault to improve the model. This is why LTSM or GRU would be interesting as part of a final solution.

```
[ ]:
```