Predictive maintenance of Aircraft engine using Long Short Term Memory Neural Networks

TABLE OF CONTENTS

AIM 1
PROBLEM DESCRIPTION 1
DATASET 1
RECURRENT NEURAL NETWORK 2
LONG SHORT TERM MEMORY (LSTM) NETWORKS 3
GATED RECURRENT UNIT (GRU) 6
CUSTOM MODEL 6
IMPLEMENTATION 7
IMPORTING NECESSARY LIBRARIES 7
IMPORTING THE DATASET 7
DATA PREPROCESSING AND EXPLORATORY DATA ANALYSIS 8
MODELLING 14
CONCLUSION 23
FUTURE SCOPE 23
REFERENCES 24

AIM:

To model an analytical framework to implement predictive maintenance on aircraft engines using Long Short Term Memory (LSTM) neural networks.

PROBLEM DESCRIPTION:

The objective of this project is to build a LSTM network in order to implement predictive maintenance capabilities to aircraft engines, such that it will help to increase the efficiency of usage of the engines and also prevent any misfortunes. Predictive maintenance encompasses a variety of topics, including but not limited to, failure prediction, fault diagnosis (root cause analysis), fault detection, failure type classification and recommendation of mitigation or maintenance actions after failure. This predictive maintenance system focuses on the technique to predict Remaining Useful Life (RUL) or time to failure of aircraft engines. The network uses simulated aircraft sensor values to predict when an aircraft engine will fail in the future, so that maintenance can be planned in advance. The columns correspond to unit number, time (in cycles) and measurements from 21 sensors.

METHODOLOGY:

DATASET:

The dataset is collected from Prognostics CoE at NASA Ames. Data sets consist of multiple multivariate time series. There are three operational settings that have a substantial effect on engine performance. The dataset consists of 26 columns of numbers, separated by spaces. Each row is a snapshot of data taken during a single operational cycle; each column is a different variable.

Given the data about aircraft engine operation and failure events history, we need to predict when an in-service engine will fail. We re-formulate the problem into two closely related sub-problems and solve them using two different types of machine learning models.

1) Is the engine going to fail within w1 cycles? **- Binary Classification**
2) How many cycles will an in-service engine last before it fails? - **Regression**

Sample training data

~20k rows,
100 unique engine id

id	cycle	setting1	setting2	setting3	s1	s2	s3	...	s19	s20	s21
1	1	-0.0007	-0.0004	100	518.67	641.82	1589.7		100	39.06	23.419
1	2	0.0019	-0.0003	100	518.67	642.15	1591.82		100	39	23.4236
1	3	-0.0043	0.0003	100	518.67	642.35	1587.99		100	38.95	23.3442
...	...										
1	191	0	-0.0004	100	518.67	643.34	1602.36		100	38.45	23.1295
1	192	0.0009	0	100	518.67	643.54	1601.41		100	38.48	22.9649
2	1	-0.0018	0.0006	100	518.67	641.89	1583.84		100	38.94	23.4585
2	2	0.0043	-0.0003	100	518.67	641.82	1587.05		100	39.06	23.4085
2	3	0.0018	0.0003	100	518.67	641.55	1588.32		100	39.11	23.425
...	...										
2	286	-0.001	-0.0003	100	518.67	643.44	1603.63		100	38.33	23.0169
2	287	-0.0005	0.0006	100	518.67	643.85	1608.5		100	38.43	23.0848

Sample testing data

~13k rows,
100 unique engine id

id	cycle	setting1	setting2	setting3	s1	s2	s3	...	s19	s20	s21
1	1	0.0023	0.0003	100	518.67	643.02	1585.29		100	38.86	23.3735
1	2	-0.0027	-0.0003	100	518.67	641.71	1588.45		100	39.02	23.3916
1	3	0.0003	0.0001	100	518.67	642.46	1586.94		100	39.08	23.4166
...	...										
1	30	-0.0025	0.0004	100	518.67	642.79	1585.72		100	39.09	23.4069
1	31	-0.0006	0.0004	100	518.67	642.58	1581.22		100	38.81	23.3552
2	1	-0.0009	0.0004	100	518.67	642.66	1589.3		100	39	23.3923
2	2	-0.0011	0.0002	100	518.67	642.51	1588.43		100	38.84	23.2902
2	3	0.0002	0.0003	100	518.67	642.58	1595.6		100	39.02	23.4064
...	...										
2	48	0.0011	-0.0001	100	518.67	642.64	1587.71		100	38.99	23.2918
2	49	0.0018	-0.0001	100	518.67	642.55	1586.59		100	38.81	23.2618
3	1	-0.0001	0.0001	100	518.67	642.03	1589.92		100	38.99	23.296
3	2	0.0039	-0.0003	100	518.67	642.23	1597.31		100	38.84	23.3191
3	3	0.0006	0.0003	100	518.67	642.98	1586.77		100	38.69	23.3774
...	...										
3	125	0.0014	0.0002	100	518.67	643.24	1588.64		100	38.56	23.227
3	126	-0.0016	0.0004	100	518.67	642.88	1589.75		100	38.93	23.274

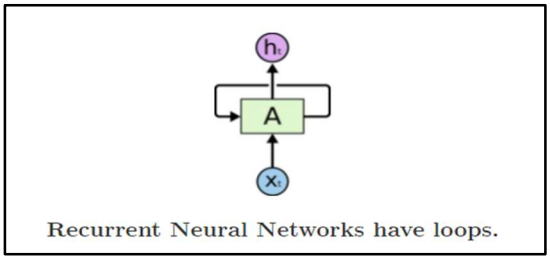
Sample ground truth data

100 rows

RUL
112
98
69
82
91

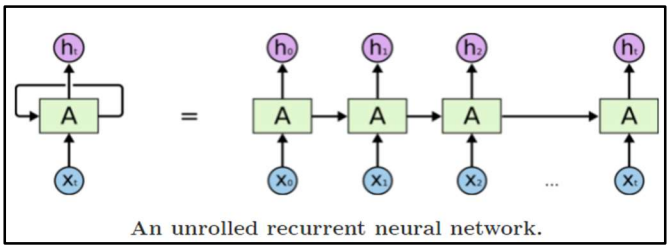
RECURRENT NEURAL NETWORK:

A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data. The traditional neural networks do not look back into previous outputs before predicting the new ones. Recurrent Neural networks address this issue.

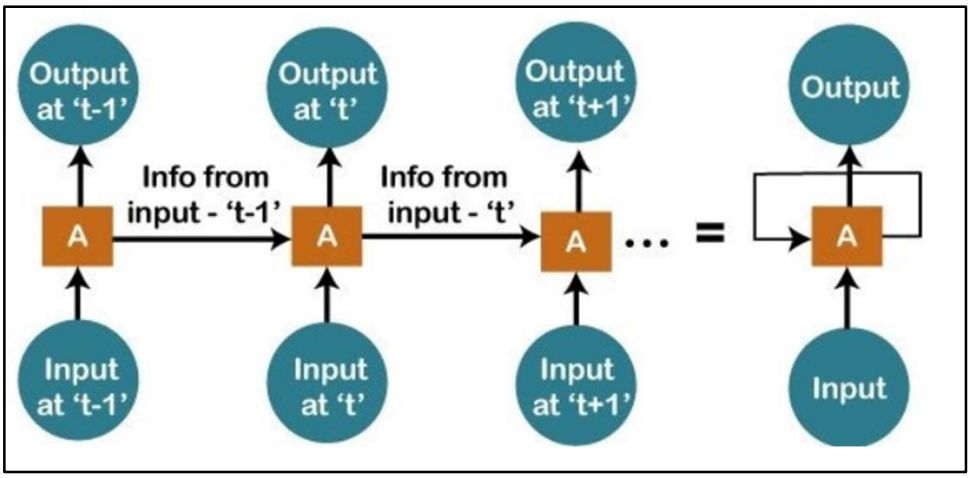


Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, A, looks at some input x_t and outputs a value ht. A loop allows information to be passed from one step of the network to the next. If we unroll the loop:



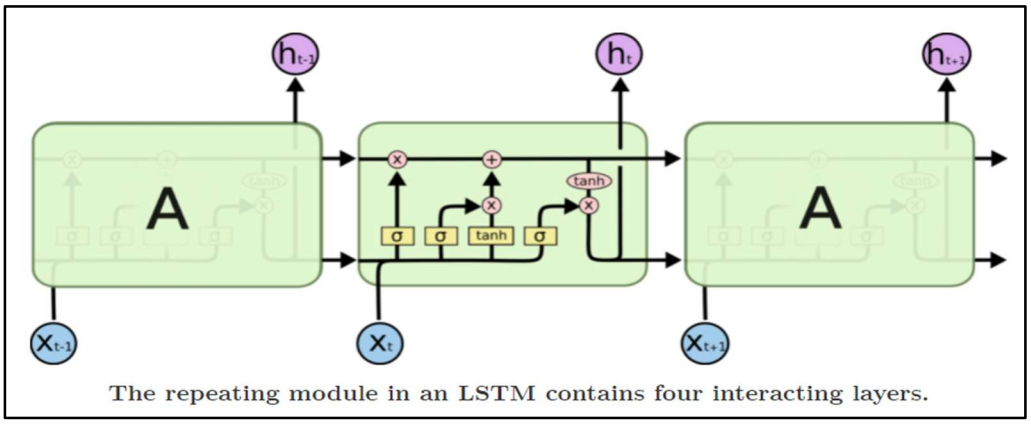
An unrolled recurrent neural network.



A recurrent neural network uses a backpropagation algorithm for training, but backpropagation happens for every timestamp, which is why it is commonly called as backpropagation through time. With backpropagation, there are certain issues, namely vanishing and exploding gradients. Also, because of vanishing gradients, RNN can not hold the past data in memory over a long duration. These shortcomings of RNN are rectified by Long Short Term Memory (LSTM).

LONG SHORT TERM MEMORY (LSTM) NETWORKS:

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. LSTMs have feedback connections, which enables LSTMs to process entire sequences of data (e.g. time series) without treating each point in the sequence independently, but rather, retaining useful information about previous data in the sequence to help with the processing of new data points.

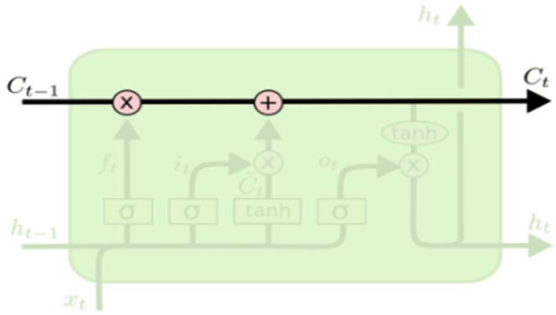


The repeating module in an LSTM contains four interacting layers.

LSTMs use a series of 'gates' which control how the information in a sequence of data comes into, is stored in and leaves the network. There are three gates in a typical LSTM, which are forget gate, input gate and output gate. These gates can be thought of as filters and are each their own neural network.

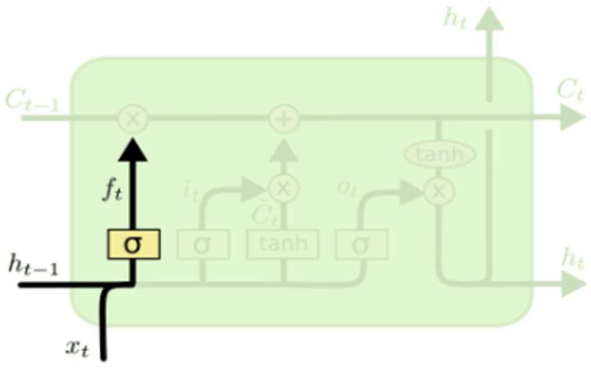
Cell State:

The cell state is like a conveyor belt, carrying the past state, with minor interactions in between. LSTM has the ability to add or remove information from the cell state.



Forget Gate:

The first step in the LSTM is to identify that information which is not required and will be thrown away from the cell state. This decision is made by a sigmoid layer, which is called the forget gate layer. The sigmoid output values between 0 and 1, determining how much of each component has to pass through.

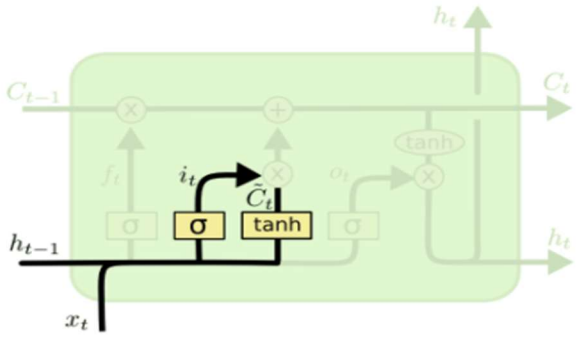


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

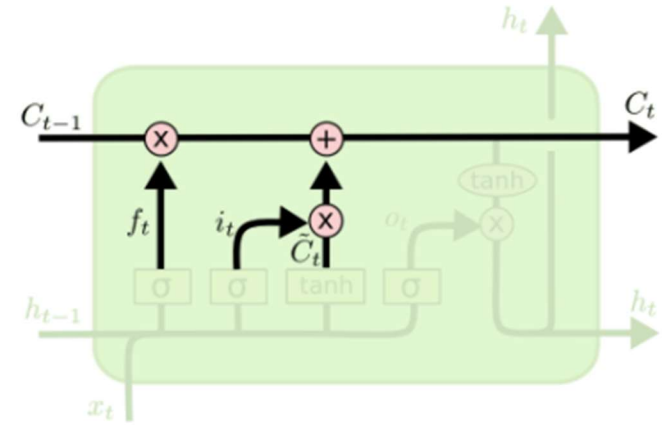
Input Gate:

The next step is to determine which new information has to be added to the cell state. This has two steps:

1) Sigmoid layer, called the input gate layer, determines which values we shall update
2) Tanh layer creates a vector of new candidate values that could be added to the state.



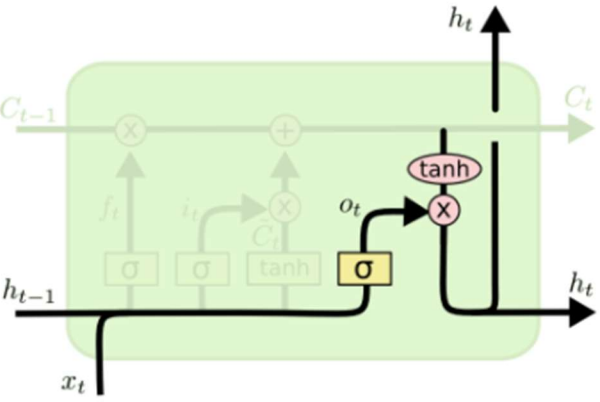
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output Gate:

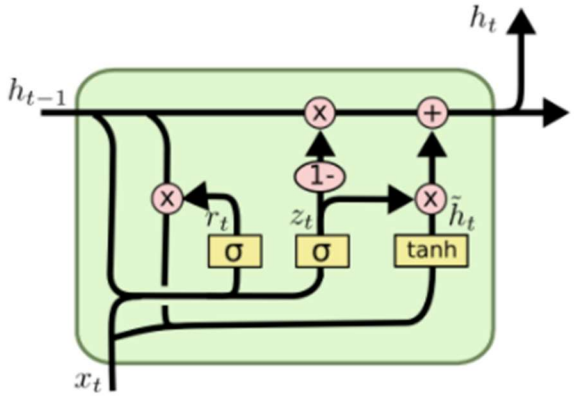
Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

GATED RECURRENT UNIT (GRU):

Gated Recurrent units (GRUs) are similar to LSTMs, as it solves the vanishing gradient problem of RNN like LSTM. But unlike LSTM, which has 3 gates, GRU has only 2 gates. It combines the forget and input gates into a single "Update gate". It also merges the cell state and hidden state and makes some other changes. The resulting model is simpler than standard LSTM models.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Update gate:

The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future.

Reset Gate:

This gate is used from the model to decide how much of the past information to forget. The formula is the same as the one for the update gate. The difference comes in the weights and the gate's usage.

CUSTOM MODEL:

We plan to compare the performances of LSTM and GRU models separately and then create a model combining the best of both the models to create a custom model and analyse its performance for the classification and regression problems. The idea of this custom model is to combine the learnings of LSTM and GRU models and analyse how the combined model performs.

IMPLEMENTATION:

1) IMPORTING NECESSARY LIBRARIES:

```
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
import klib
%matplotlib inline
```

Pandas and numpy libraries are used to load, handle and perform mathematical analysis on the dataset. Matplotlib and Seaborn libraries are used for visualization of data. Klib library is used for data preprocessing.

```
import keras
from keras.layers.core import Activation
from keras.models import Sequential, load_model, Model
from keras.layers import Dense, Dropout, LSTM, GRU,Input,Concatenate
from sklearn.preprocessing import MinMaxScaler
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

Keras is a powerful and easy-to-use free open source Python library for developing and evaluating deep learning models. It wraps the efficient numerical computation libraries Theano and TensorFlow and allows you to define and train neural network models in just a few lines of code.

2) IMPORTING THE DATASET:

```
1 train_data = pd.read_csv("PM_train.txt", sep=" ", header=None)
2 train_data
```

id	cycle	setting1	setting2	setting3	s1	s2	s3	s4	s5	...	s12	s13	s14	s15	s16	s17	s18	s19	s20	s21
1	1	-0.0007	-0.0004	100.0	518.67	641.82	1589.70	1400.60	14.62	...	521.66	2388.02	8138.62	8.4195	0.03	392	2388	100.0	39.06	23.4190
1	2	0.0019	-0.0003	100.0	518.67	642.15	1591.82	1403.14	14.62	...	522.28	2388.07	8131.49	8.4318	0.03	392	2388	100.0	39.00	23.4236
1	3	-0.0043	0.0003	100.0	518.67	642.35	1587.99	1404.20	14.62	...	522.42	2388.03	8133.23	8.4178	0.03	390	2388	100.0	38.95	23.3442
1	4	0.0007	0.0000	100.0	518.67	642.35	1582.79	1401.87	14.62	...	522.86	2388.08	8133.83	8.3682	0.03	392	2388	100.0	38.88	23.3739
1	5	-0.0019	-0.0002	100.0	518.67	642.37	1582.85	1406.22	14.62	...	522.19	2388.04	8133.80	8.4294	0.03	393	2388	100.0	38.90	23.4044
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
100	196	-0.0004	-0.0003	100.0	518.67	643.49	1597.98	1428.63	14.62	...	519.49	2388.26	8137.60	8.4956	0.03	397	2388	100.0	38.49	22.9735
100	197	-0.0016	-0.0005	100.0	518.67	643.54	1604.50	1433.58	14.62	...	519.68	2388.22	8136.50	8.5139	0.03	395	2388	100.0	38.30	23.1594
100	198	0.0004	0.0000	100.0	518.67	643.42	1602.46	1428.18	14.62	...	520.01	2388.24	8141.05	8.5646	0.03	398	2388	100.0	38.44	22.9333
100	199	-0.0011	0.0003	100.0	518.67	643.23	1605.26	1426.53	14.62	...	519.67	2388.23	8139.29	8.5389	0.03	395	2388	100.0	38.29	23.0640
100	200	-0.0032	-0.0005	100.0	518.67	643.85	1600.38	1432.14	14.62	...	519.30	2388.26	8137.33	8.5036	0.03	396	2388	100.0	38.37	23.0522

3) DATA PREPROCESSING AND EXPLORATORY DATA ANALYSIS:

Data preprocessing was done using Klib library. As the data is multivariate time series data, it was space separated and was not properly indexed. Initial preprocessing involved changing the text file(.txt) to CSV file by properly denoting the column names.

The dataset contains failure data of 100 engines denoted by their ID numbers, with 3 operational settings and sensor measurements from 21 sensors for several time cycles before eventual failure. Using feature engineering we created a few other features namely, Remaining Useful Life (RUL), label1 and label2.

The RUL feature is created by subtracting the current cycle from maximum cycles for a given engine ID, as we know that the engine failed at the maximum cycle.

```
rul = pd.DataFrame(train_data.groupby('id')['cycle'].max()).reset_index()
```

```
train_data = train_data.merge(rul, on=['id'], how='left')
```

```
train_data['RUL'] = train_data['max'] - train_data['cycle']
train_data.drop('max', axis=1, inplace=True)
```

“label1” denotes whether the engine will fail or not within a given window w1 and “label2” denotes whether the engine will fail with the period of (1,w0) or period of (w0+1,w1) or will not fail. Thus, label1 is used for binary classification with values 1 or 0, denoting that the engine will fail or not. label2 is used for multi-class classification with 0 denoting that the engine will not fail, 1 denoting it will fail within the window (w0+1, w1) and 2 denoting it will fail within window (1, w0). Here, we have taken the window w0 and w1 as 15 and 30 cycles. It is used to alert the users for forthcoming maintenance.

```
w1 = 30
w0 = 15
train_data['label1'] = np.where(train_data['RUL']<=w1, 1,0 )
train_data['label2'] = train_data['label1']
train_data.loc[train_data['RUL']<=w0,'label2'] =2
```

Normalization of data:

Normalization refers to rescaling real valued numeric attributes into the range 0 and 1. It is done to make the data fit into the model efficiently and it can reduce the training time significantly.

```
min_max_scaler = MinMaxScaler()
train_data['norm_cycle'] = train_data['cycle']
norm_cols = train_data.columns.difference(['id','cycle','RUL','label1','label2'])
norm_train_data = pd.DataFrame(min_max_scaler.fit_transform(train_data[norm_cols]), columns=norm_cols,
                               index=train_data.index)
```

id	cycle	setting1	setting2	setting3	s1	s2	s3	s4	s5	...	s16	s17	s18	s19	s20	s21	RUL	label1	label2	norm_cycle
1	1	0.459770	0.166667	0.0	0.0	0.183735	0.406802	0.309757	0.0	...	0.0	0.333333	0.0	0.0	0.713178	0.724662	191	0	0	0.00000
1	2	0.609195	0.250000	0.0	0.0	0.283133	0.453019	0.352633	0.0	...	0.0	0.333333	0.0	0.0	0.666667	0.731014	190	0	0	0.00277
1	3	0.252874	0.750000	0.0	0.0	0.343373	0.369523	0.370527	0.0	...	0.0	0.166667	0.0	0.0	0.627907	0.621375	189	0	0	0.00554
1	4	0.540230	0.500000	0.0	0.0	0.343373	0.256159	0.331195	0.0	...	0.0	0.333333	0.0	0.0	0.573643	0.662386	188	0	0	0.00831
1	5	0.390805	0.333333	0.0	0.0	0.349398	0.257467	0.404625	0.0	...	0.0	0.416667	0.0	0.0	0.589147	0.704502	187	0	0	0.01108

Removing unwanted data:

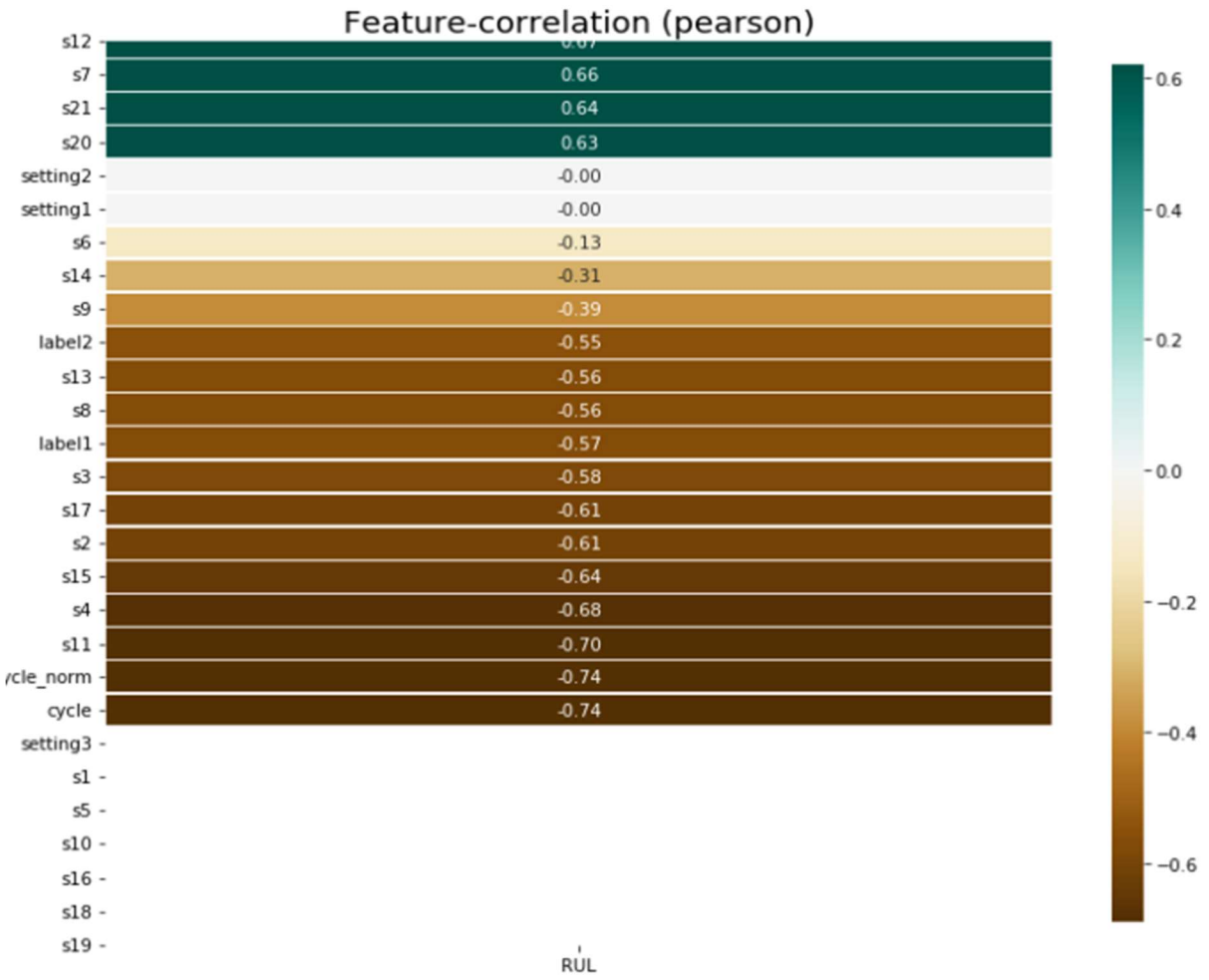
Using the Klib library, we clean the data. From the below figure, it is evident that there are no duplicates or missing values in our data, but few columns have single values, meaning that their values do not change over time. So, those columns will not affect our prediction, thus we can remove them. By doing this, we reduced our dataset's memory by nearly 60% meaning our model will train faster.

```
klib.data_cleaning(data)
```

```
Shape of cleaned data: (20631, 22)Remaining NAs: 0

Changes:
Dropped rows: 0
     of which 0 duplicates. (Rows: [])
Dropped columns: 7
     of which 7 single valued.     Columns: ['setting3', 's1', 's5', 's10', 's16', 's18', 's19']
Dropped missing values: 0
Reduced memory by at least: 2.67 MB (-58.55%)
```

Checking correlation of RUL:

```
klib.corr_plot(data, target='RUL')
```

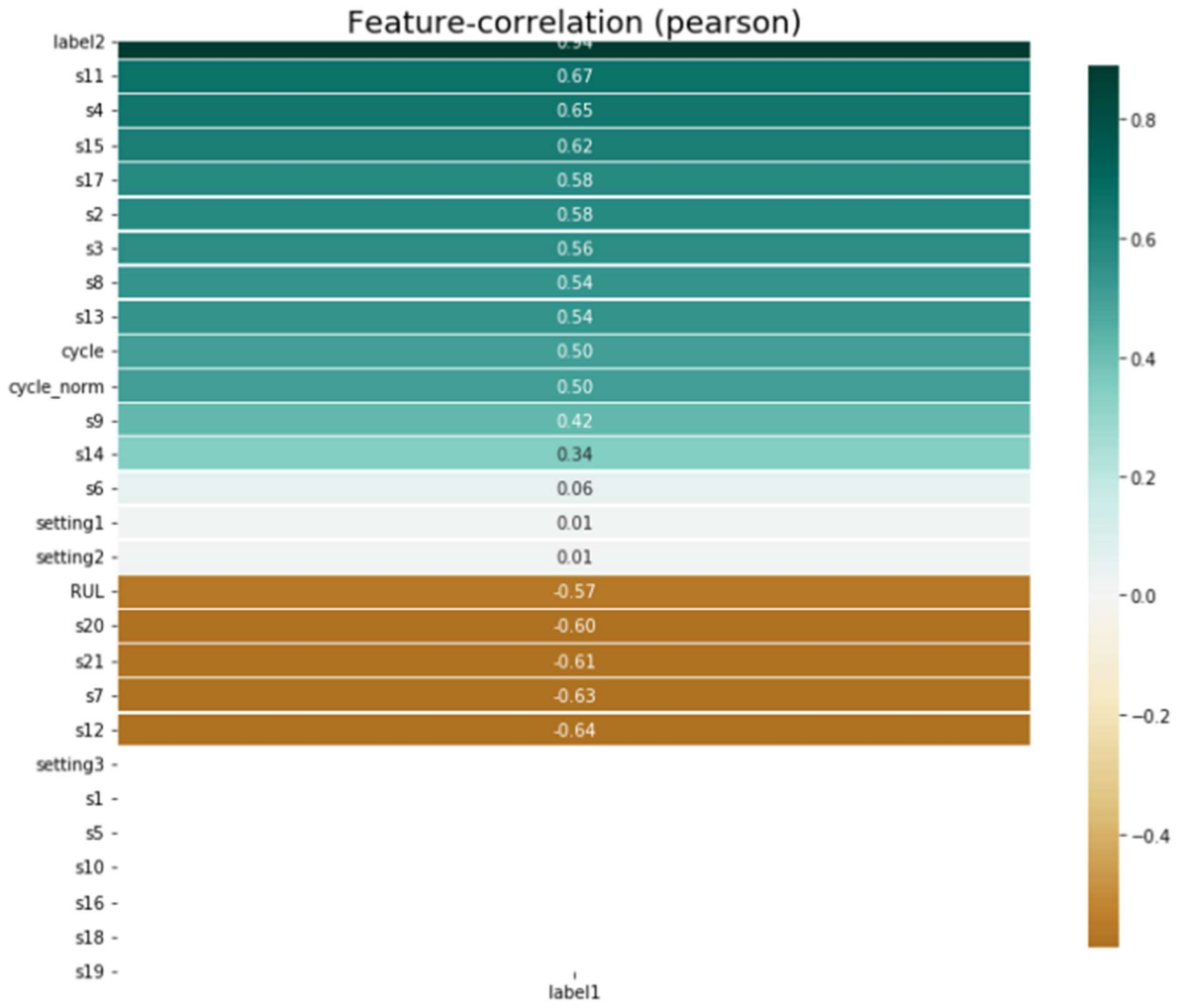


The above figure correlates RUL value with all other features. RUL gives the insight about our engines remaining lifespan and is our prediction variable for our regression problem.

Checking correlation of label1:

```
klib.corr_plot(data, target='label1')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b720634388>
```



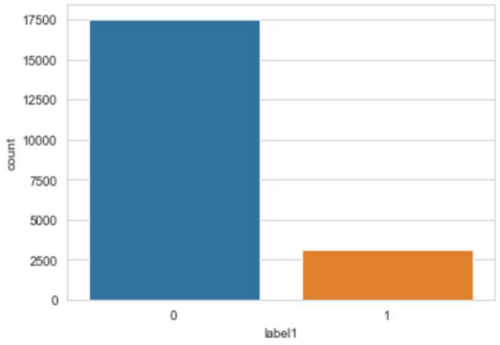
The above figure correlates label1 with all the other features. “label1” is the prediction variable for the binary classification problem.

Data Visualizations:

Distribution of values in label1 and label2:

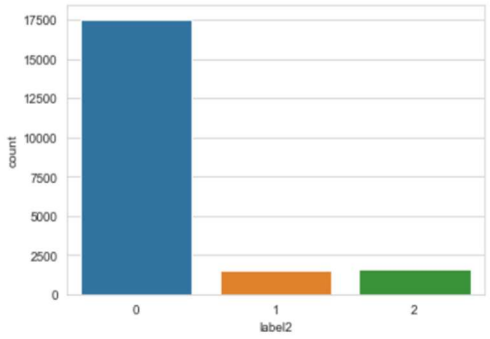
```
sns.set_style("whitegrid")
sns.countplot(x='label1', data=data)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x23293462e88>
```




```
sns.set_style("whitegrid")
sns.countplot(x='label2', data=data)
```

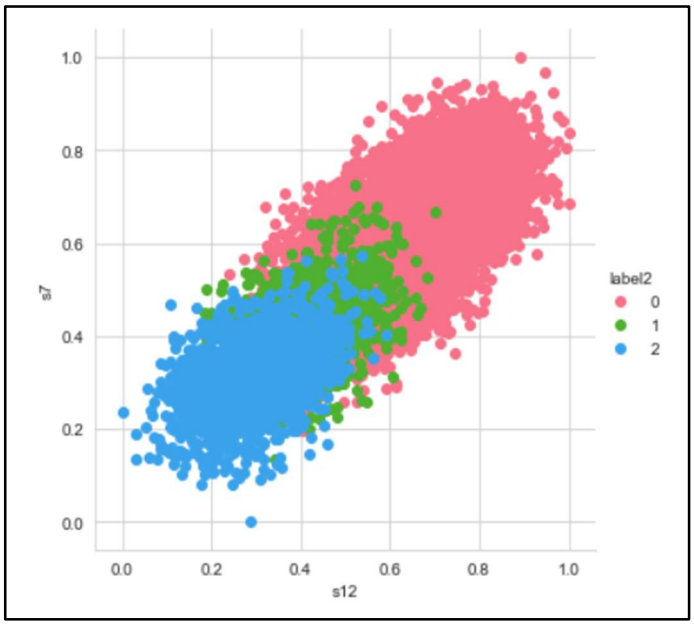
```
<matplotlib.axes._subplots.AxesSubplot at 0x232934d0788>
```



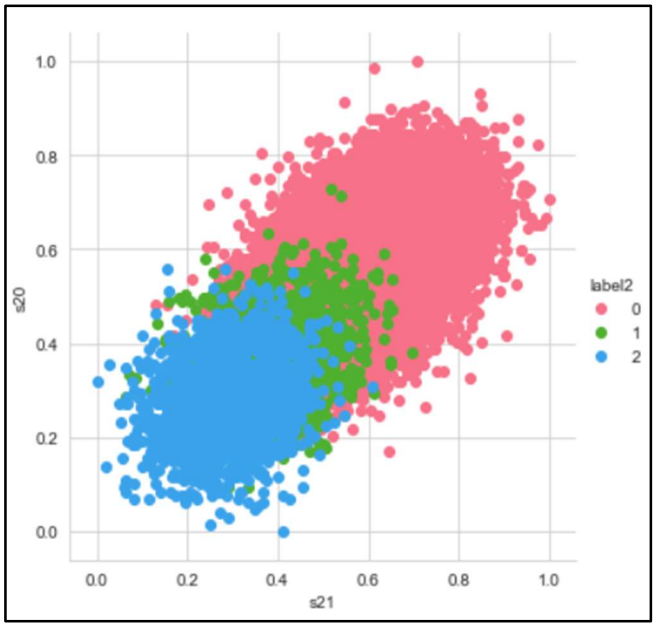
The above figures show the distribution of values in label1 and label2.

Distribution of labels based on sensors :

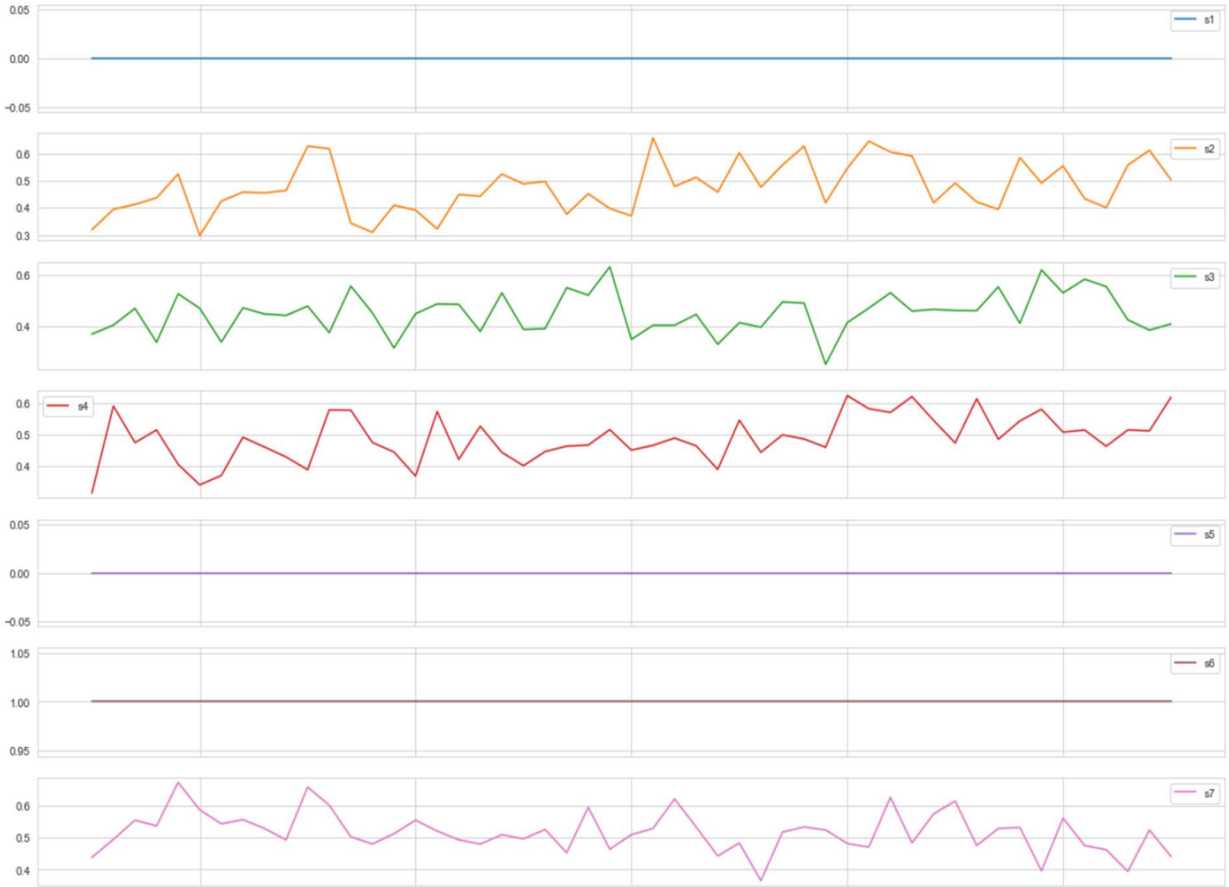
Sensor-7 vs sensor-12:



Sensor-21 vs sensor-20:



Variation of sensor measurements over time:



4) MODELLING:

Binary Classification:

In this problem, our aim is to predict whether our engine will fail within a given window or not. So, we will use label1 values as the prediction variable for this problem. The LSTM network requires input to be given in a certain format, so we will generate the input sequence and label sequence for fitting the data into our LSTM network.

```
def gen_sequence(id_df,seq_length,seq_cols):
#     print(id_df)
    data_array = id_df[seq_cols].values
#     print(data_array)
#     input()
    num_elements = data_array.shape[0]
#     print(data_array.shape[0])
    for start,stop in zip(range(0,num_elements-seq_length), range(seq_
length,num_elements)):
        yield data_array[start:stop,:]
```

```
# generator for the sequences
seq_gen = (list(gen_sequence(train_df[train_df['id']==id], sequence_le
ngth, sequence_cols))
           for id in train_df['id'].unique())
# seq_gen
```

```
# generate sequences and convert to numpy array
seq_array = np.concatenate(list(seq_gen)).astype(np.float32)
seq_array.shape
```

```
(15631, 50, 25)
```

```
# function to generate labels
def gen_labels(id_df, seq_length, label):
    data_array = id_df[label].values
    num_elements = data_array.shape[0]
    return data_array[seq_length:num_elements, :]
```

```
# generate labels
label_gen = [gen_labels(train_df[train_df['id']==id], sequence_length,
['label1'])
             for id in train_df['id'].unique()]
label_array = np.concatenate(label_gen).astype(np.float32)
label_array.shape
```

```
(15631, 1)
```

LSTM Model:

```
# build the network
nb_features = seq_array.shape[2]
nb_out = label_array.shape[1]

model = Sequential()

model.add(LSTM(
         input_shape=(sequence_length, nb_features),
         units=100,
         return_sequences=True))
model.add(Dropout(0.2))

model.add(LSTM(
          units=100,
          return_sequences=True))
# model.add(Dropout(0.2))
model.add(Dropout(0.2))

model.add(LSTM(
          units=50,
          return_sequences=True))
model.add(Dropout(0.2))

model.add(LSTM(
          units=50,
          return_sequences=False))
model.add(Dropout(0.2))

model.add(Dense(units=nb_out, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())
```

We are using 20% dropout to reduce the overfitting of the model.

```
# training metrics
scores = model.evaluate(seq_array, label_array, verbose=1, batch_size=200)
print('Accurracy: {}'.format(scores[1]))
```

```
15631/15631 [==============================] - 12s 780us/step
Accurracy: 0.9760732054710388
```

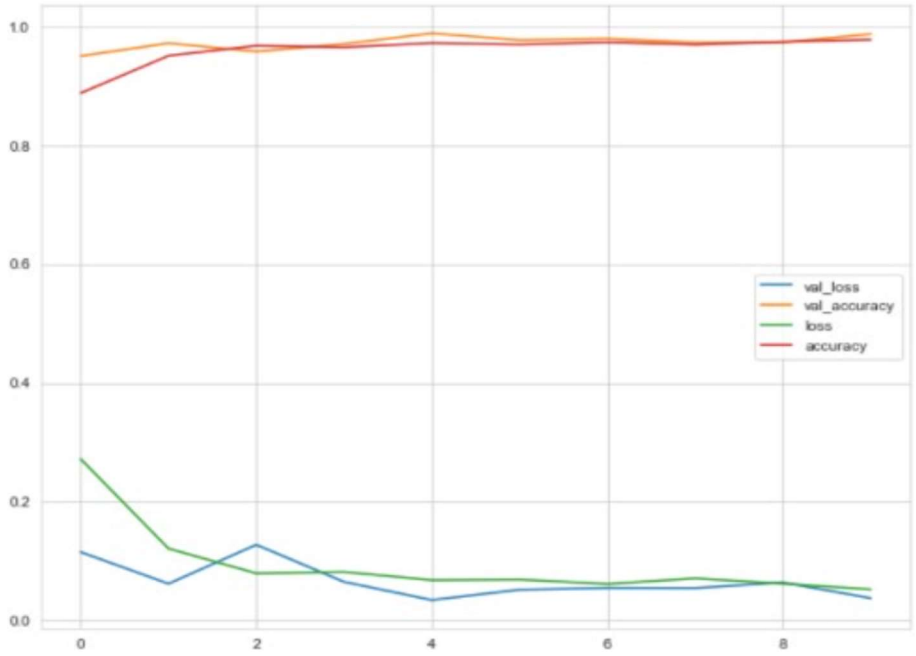
```
# test metrics
scores_test = model.evaluate(seq_array_test_last, label_array_test_last, verbose=2)
print('Accurracy: {}'.format(scores_test[1]))
```

```
Accurracy: 0.9677419066429138
```

Here, the training accuracy was found to be around 98% and testing accuracy was found to be around 97%

```
# compute precision and recall
precision_test = precision_score(y_true_test, y_pred_test)
recall_test = recall_score(y_true_test, y_pred_test)
f1_test = 2 * (precision_test * recall_test) / (precision_test + recall_test)
print( 'Precision: ', precision_test, '\n', 'Recall: ', recall_test,'\n', 'F1-score:', f1_test )
```

```
Precision:  0.9583333333333334
 Recall:  0.92
 F1-score: 0.9387755102040817
```



	Accuracy	Precision	Recall	F1-score
LSTM	0.967742	0.958333	0.92	0.938776
Template Best Model	0.940000	0.952381	0.80	0.869565

Our model performed better than the best template model proposed in the paper.

GRU Model:

```
# build the network
nb_features = seq_array.shape[2]
nb_out = label_array.shape[1]

model = Sequential()

model.add(GRU(
         input_shape=(sequence_length, nb_features),
         units=100,
         return_sequences=True))
model.add(Dropout(0.2))

model.add(GRU(
          units=100,
          return_sequences=True))
# model.add(Dropout(0.2))
model.add(Dropout(0.2))

model.add(GRU(
          units=50,
          return_sequences=True))
model.add(Dropout(0.2))

model.add(GRU(
          units=50,
          return_sequences=False))
model.add(Dropout(0.2))

model.add(Dense(units=nb_out, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())
```

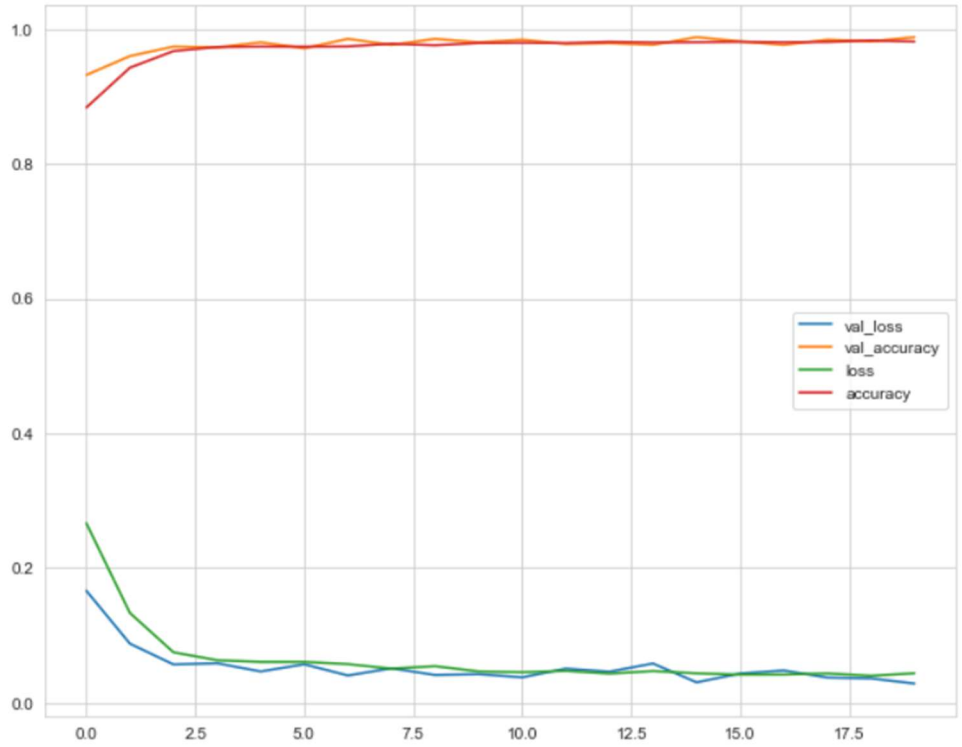
```
# training metrics
scores = model.evaluate(seq_array, label_array, verbose=1, batch_size=200)
print('Accurracy: {}'.format(scores[1]))
```

```
15631/15631 [==============================] - 12s 757us/step
Accurracy: 0.9855415225028992
```

```
# test metrics
scores_test = model.evaluate(seq_array_test_last, label_array_test_last, verbose=2)
print('Accurracy: {}'.format(scores_test[1]))
```

```
Accurracy: 0.9784946441650391
```

Here, our training accuracy is 98.5% and testing accuracy is around 98%. GRU model performed slightly better than LSTM model.



	Accuracy	Precision	Recall	F1-score
GRU	0.978495	0.925926	1.0	0.961538
Template Best Model	0.940000	0.952381	0.8	0.869565

Combining both the Models:

The function we defined here, gives the input to both the models (LSTM and GRU) separately and trains them independent of each other. Once both the models are trained independently, we combine both the features to create a combined model using concatenate function.

```
def best_model():
    input1 = Input(shape=(sequence_length,nb_features),name='input1',dtype='float32')
    input2 = Input(shape=(sequence_length,nb_features),name='input2',dtype='float32')

    lstm1 = LSTM(units=200, return_sequences=True)(input1)
    lstm1 = Dropout(0.2)(lstm1)
    lstm1 = LSTM(units=100, return_sequences=True)(lstm1)
    lstm1 = Dropout(0.2)(lstm1)
    lstm1 = LSTM(units=50, return_sequences=False)(lstm1)
    lstm1 = Dropout(0.2)(lstm1)

    lstm2 = GRU(units=200, return_sequences=True)(input2)
    lstm2 = Dropout(0.2)(lstm2)
    lstm2 = GRU(units=100, return_sequences=True)(lstm2)
    lstm2 = Dropout(0.2)(lstm2)
    lstm2 = GRU(units=50, return_sequences=False)(lstm2)
    lstm2 = Dropout(0.2)(lstm2)

    comb = Concatenate(axis=1)([lstm1,lstm2])
    pred = Dense(units=nb_out, activation='sigmoid', name='output')(comb)
    model = Model(inputs=[input1,input2], outputs=pred)

    return model
```

```
model = best_model()
model.summary()
```

```
# training metrics
scores = model.evaluate([seq_array,seq_array], label_array, verbose=1, batch_size=200)
print('Accurracy: {}'.format(scores[1]))
```

```
15631/15631 [==============================] - 25s 2ms/step
Accurracy: 0.9861813187599182
```

```
# test metrics
scores_test = model.evaluate([seq_array_test_last,seq_array_test_last], label_array_test_last, verbose=2)
print('Accurracy: {}'.format(scores_test[1]))
```

```
Accurracy: 0.9892473220825195
```

	Accuracy	Precision	Recall	F1-score
LSTM-GRU	0.989247	0.961538	1.0	0.980392
Template Best Model	0.940000	0.952381	0.8	0.869565

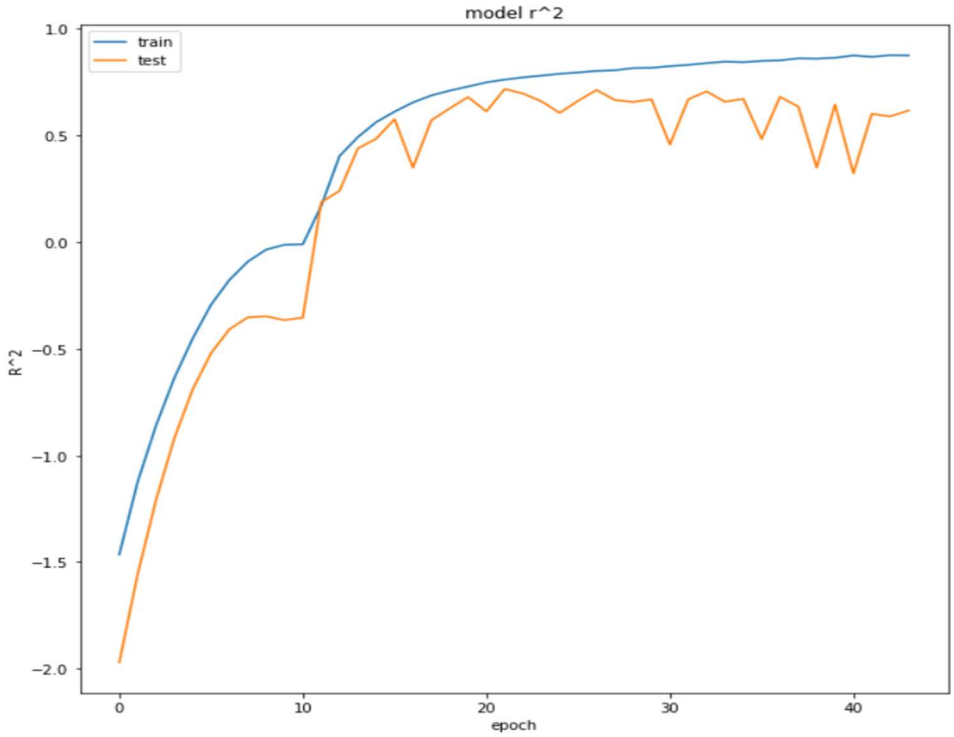
LSTM and GRU models train independently and the custom model combines both the models and uses it to predict the final output. Thus, giving higher accuracy.

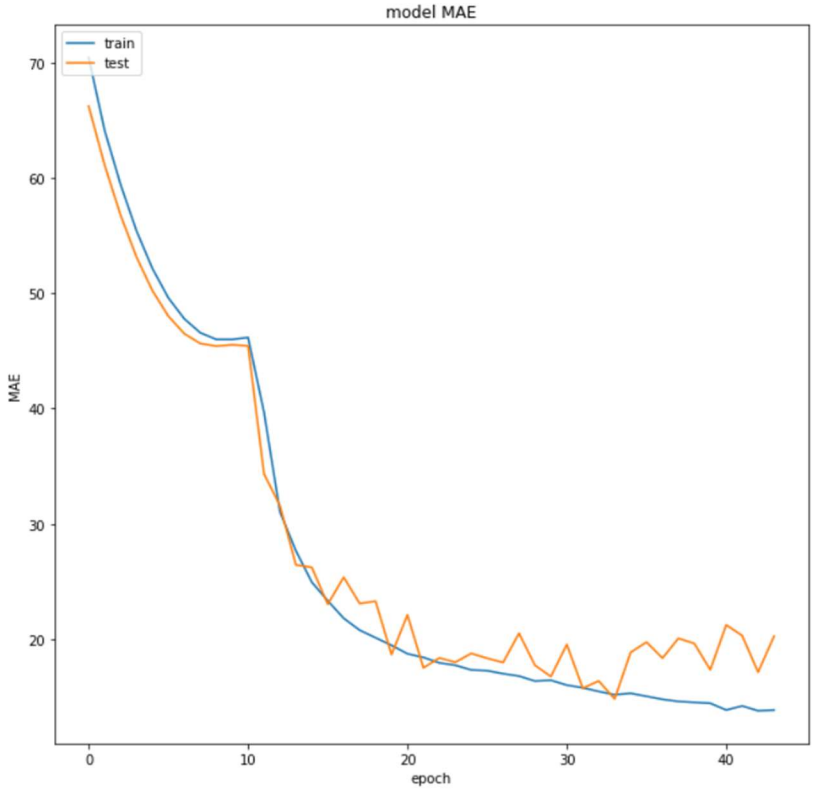
Regression:

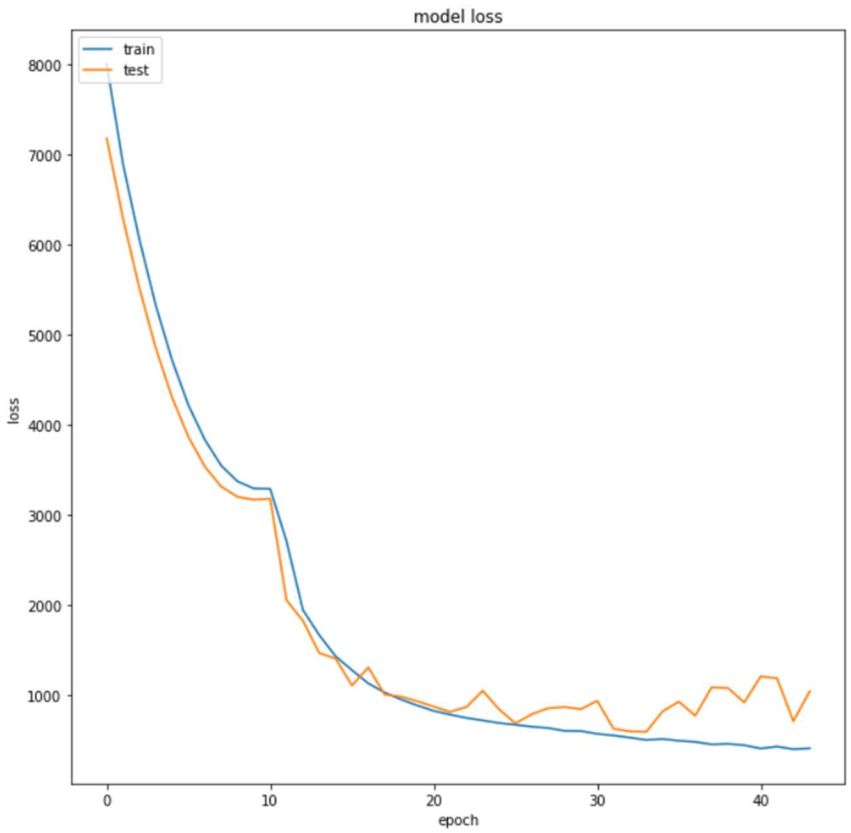
In the regression problem, we aim to estimate the Remaining Useful Life(RUL) of the aircraft engine, which gives insight into how long the engine will run before eventual failure. Since we have made our custom model, which showed very good accuracy for the binary classification problem, we tried the same model for the regression problem.

```
def best_model():
    input1 = Input(shape=(sequence_length,nb_features),name='input1',dtype='float32')
    input2 = Input(shape=(sequence_length,nb_features),name='input2',dtype='float32')
    print(input1.shape)
    print(input2.shape)
    lstm1 = LSTM(units=200, return_sequences=True)(input1)
    lstm1 = Dropout(0.2)(lstm1)
    lstm1 = LSTM(units=100, return_sequences=True)(lstm1)
    lstm1 = Dropout(0.2)(lstm1)
    lstm1 = LSTM(units=50, return_sequences=False)(lstm1)
    lstm1 = Dropout(0.2)(lstm1)

    lstm2 = GRU(units=200, return_sequences=True)(input2)
    lstm2 = Dropout(0.2)(lstm2)
    lstm2 = GRU(units=100, return_sequences=True)(lstm2)
    lstm2 = Dropout(0.2)(lstm2)
    lstm2 = GRU(units=50, return_sequences=False)(lstm2)
    lstm2 = Dropout(0.2)(lstm2)

    comb = Concatenate(axis=1)([lstm1,lstm2])
    pred = Dense(units=nb_out, activation='linear', name='output')(comb)
    model = Model(inputs=[input1,input2], outputs=pred)
    model.compile(loss='mean_squared_error', optimizer='rmsprop',metrics=['mae',r2_keras])

    return model
```

```
def r2_keras(y_true, y_pred):
    """Coefficient of Determination
    """
    SS_res =  K.sum(K.square( y_true - y_pred ))
    SS_tot = K.sum(K.square( y_true - K.mean(y_true) ) )
    return ( 1 - SS_res/(SS_tot + K.epsilon()) )
```


model r^2
train
test
R^2
epoch
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
10
20
30
40


model MAE
train
test
MAE
epoch
70
60
50
40
30
20
0
10
20
30
40




```
# training metrics
scores = model.evaluate([seq_array,seq_array], label_array, verbose=1, batch_size=200)
print('\nMAE: {}'.format(scores[1]))
print('\nR^2: {}'.format(scores[2]))
```

```
15631/15631 [==============================] - 26s 2ms/step

MAE: 14.56912899017334

R^2: 0.8014203906059265
```

```
# test metrics
scores_test = model.evaluate([seq_array_test_last,seq_array_test_last], label_array_test_last, verbose=2)
print('\nMAE: {}'.format(scores_test[1]))
print('\nR^2: {}'.format(scores_test[2]))
```

```
MAE: 13.144071578979492

R^2: 0.760365903377533
```

Here, we got training accuracy as 80% and testing accuracy is around 76%.

CONCLUSION:

RNN is quite useful for predicting time series data, but it does not have long term memory, so looking back into previous data is restricted. This restriction is overcome by using LSTM and GRU. Both LSTM and GRU perform equally well. So, we created a custom model which combines both the models to give higher accuracy. We were able to classify whether the aircraft engine will fail within a certain given window or not with an accuracy of 99% and we were able to predict the Remaining Useful Life (RUL) of an aircraft engine using its sensor data with accuracy of nearly 80%.

FUTURE SCOPE:

Predictive maintenance can be utilized everywhere from small homes to large industries. Predictive maintenance can be employed to monitor heavy machinery in industries such that they can be used to prevent sudden malfunctions. Predictive maintenance can be incorporated with IoT devices as they gather and send live sensor data, which can be used to monitor every electronic device. Adding prognostic capabilities to electronic devices can cut down the cost of maintenance drastically. By adopting a predictive-maintenance (PdM) strategy, you can mine your critical-asset data and identify anomalies or deviations from their standard performance. This can help you avoid unplanned downtime, reduce industrial maintenance overspend, and mitigate safety and environmental risks.

REFERENCES:

1) https://www.kaggle.com/behrad3d/nasa-cmaps - Kaggle
2) https://colah.github.io/posts/2015-08-Understanding-LSTMs/ - LSTM
3) https://builtin.com/data-science/recurrent-neural-networks-and-lstm - RNN and LSTM
4) https://towardsdatascience.com/data-preparation-with-klib-ec4add15303a - Klib
5) https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/#turbofan - Dataset
6) https://gallery.azure.ai/Collection/Predictive-Maintenance-Template-3
7) https://www.kaggle.com/nafisur/dataset-for-predictive-maintenance
8) https://www.youtube.com/watch?v=LfnrRPFhkuY