Predicting Permanent Magnet Synchronous Motors Temperature using Machine Learning and Deep Learning Models

COMP-5421 – Deep Learning

Group Project

Meher Hassan : 1155645

Chandreen Ravihari : 1158931

Naheem Olaniyan : 1146472

Jahin Ahmed : 1165142

Member Contribution

Criteria	Chandreen Ravihari	Meher Hassan	Naheem Olaniyan	Jahin Ahmed
#of lines of code written	125	120	95	80
# of paragraphs written in the report	12	8	10	8
Avg. hours spent per week	5hrs	5hrs	4hrs	4hrs
% of overall contribution	30%	25%	25%	20%

Outline

- Introduction
- Literature Review
- Methodology
  - Data Preprocessing
  - Machine Learning Models
  - Deep Learning Models
- Results
- Discussion and Conclusion
- Acknowledgement

Introduction

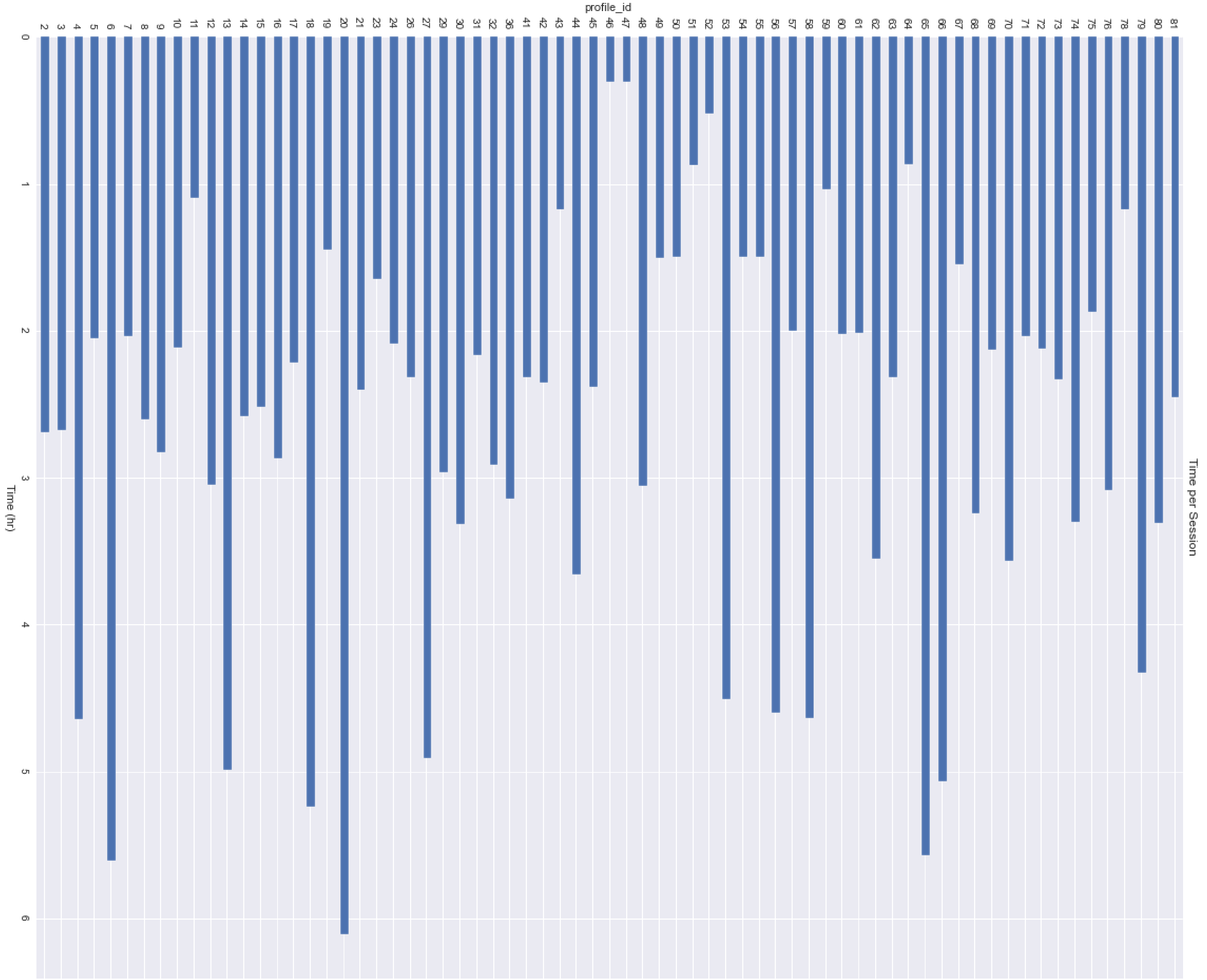
- Permanent Magnet Synchronous Motors(PMSMs) are a very popular choice in electric vehicles.
- PMSMs are Sensitive to high temperatures.
- Traditional thermal monitoring is costly and technically infeasible.
- Why Data driven temperature prediction is better?
    - Efficient
    - Cost effective
    - No expert knowledge required
- This study focus on predicting PMSMs motor temperature using machine learning and deep learning techniques

Literature Review

- Common models developed
  - deep neural networks [1]–[6] (Long-Short-Term-Memory(LSTM), CNN, Residual Neural Network (ResNet)
  - Machine learning approaches, such as KNN [1], [6], [7], support vector machine (SVM) [1], [8], Random Forests [1], [7]–[9], ordinary least squares[1].
- Many studies have predicted four components as the outputs [2]–[4], [6], [10].

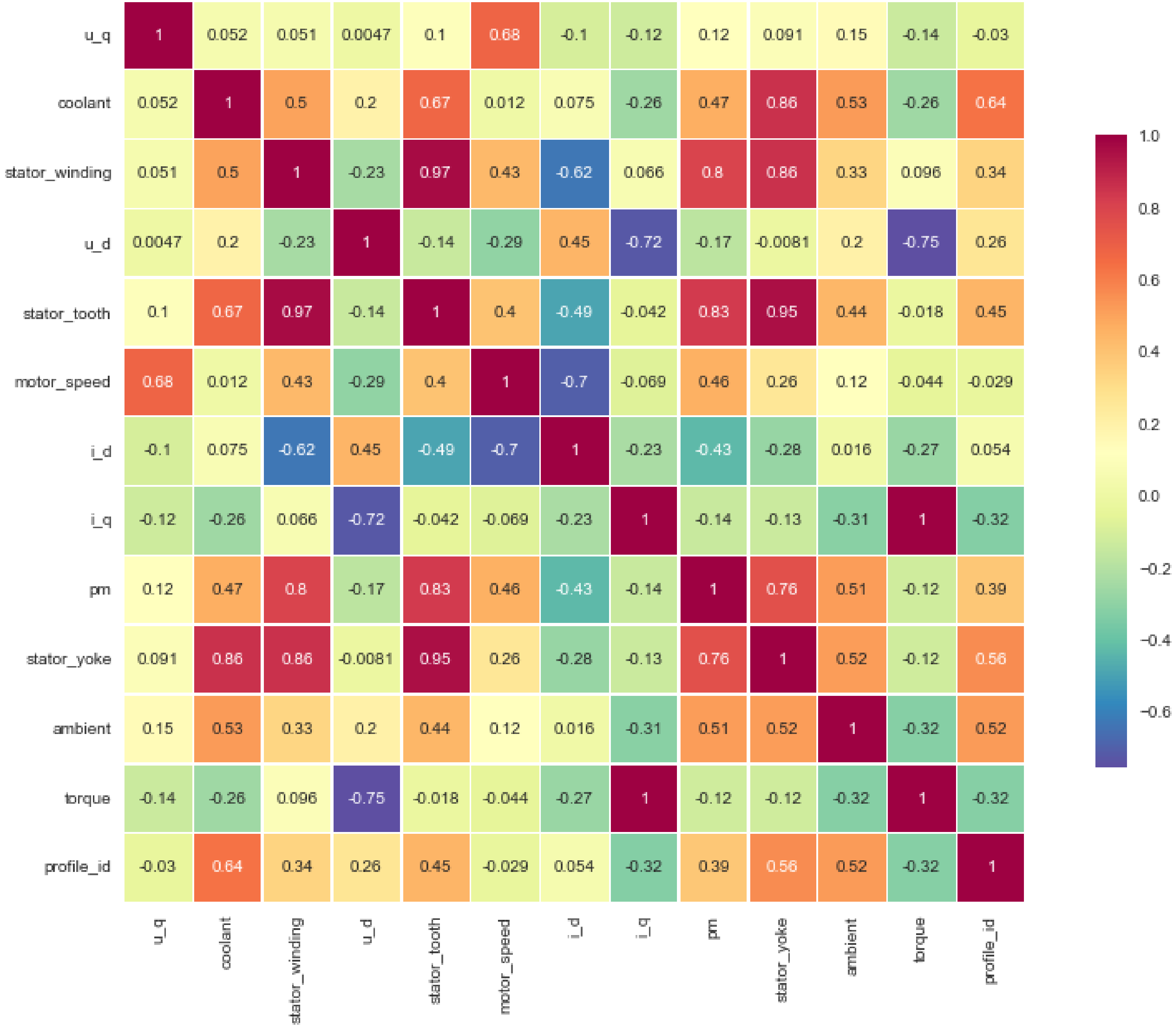
Methodology: Exploratory Data Analysis (EDA)

- Fig1 shows the total duration of all the sessions in hours.
- Most distributed around <3 hours.



Methodology: Exploratory Data Analysis (EDA)

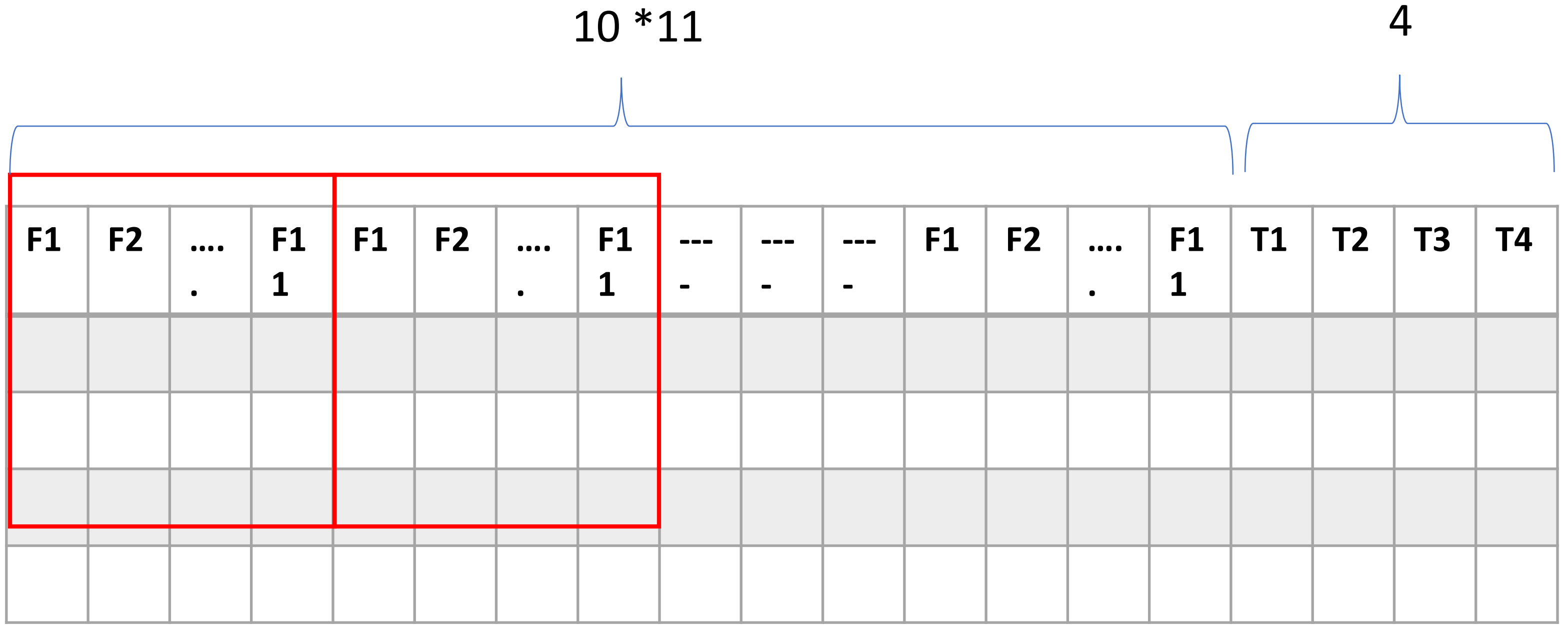
- Fig2 shows the correlation between each feature.
- torque and i_d have less correlation with targets
- Target attributes have higher inter-correlation



	u_q	coolant	stator_winding	u_d	stator_tooth	motor_speed	i_d	i_q	pm	stator_yoke	ambient	torque	profile_id
u_q	1	0.052	0.051	0.0047	0.1	0.68	-0.1	-0.12	0.12	0.091	0.15	-0.14	-0.03
coolant	0.052	1	0.5	0.2	0.67	0.012	0.075	-0.26	0.47	0.86	0.53	-0.26	0.64
stator_winding	0.051	0.5	1	-0.23	0.97	0.43	-0.62	0.066	0.8	0.86	0.33	0.096	0.34
u_d	0.0047	0.2	-0.23	1	-0.14	-0.29	0.45	-0.72	-0.17	-0.0081	0.2	-0.75	0.26
stator_tooth	0.1	0.67	0.97	-0.14	1	0.4	-0.49	-0.042	0.83	0.95	0.44	-0.018	0.45
motor_speed	0.68	0.012	0.43	-0.29	0.4	1	-0.7	-0.069	0.46	0.26	0.12	-0.044	-0.029
i_d	-0.1	0.075	-0.62	0.45	-0.49	-0.7	1	-0.23	-0.43	-0.28	0.016	-0.27	0.054
i_q	-0.12	-0.26	0.066	-0.72	-0.042	-0.069	-0.23	1	-0.14	-0.13	-0.31	1	-0.32
pm	0.12	0.47	0.8	-0.17	0.83	0.46	-0.43	-0.14	1	0.76	0.51	-0.12	0.39
stator_yoke	0.091	0.86	0.86	-0.0081	0.95	0.26	-0.28	-0.13	0.76	1	0.52	-0.12	0.56
ambient	0.15	0.53	0.33	0.2	0.44	0.12	0.016	-0.31	0.51	0.52	1	-0.32	0.52
torque	-0.14	-0.26	0.096	-0.75	-0.018	-0.044	-0.27	1	-0.12	-0.12	-0.32	1	-0.32
profile_id	-0.03	0.64	0.34	0.26	0.45	-0.029	0.054	-0.32	0.39	0.56	0.52	-0.32	1

Methodology : Data Preprocessing and Transformation

- This is a multi-target time-series regression problem
- **Input format**
  - # of input features (F): 11
  - # of target features (T): 04
  - # of past records: 10
  - # of future records: 1
  - Total records:
    - Training: 983106
    - Testing: 15291, 40084
- **Preprocessing functions:**
  - Standard Scaling
  - Removed features



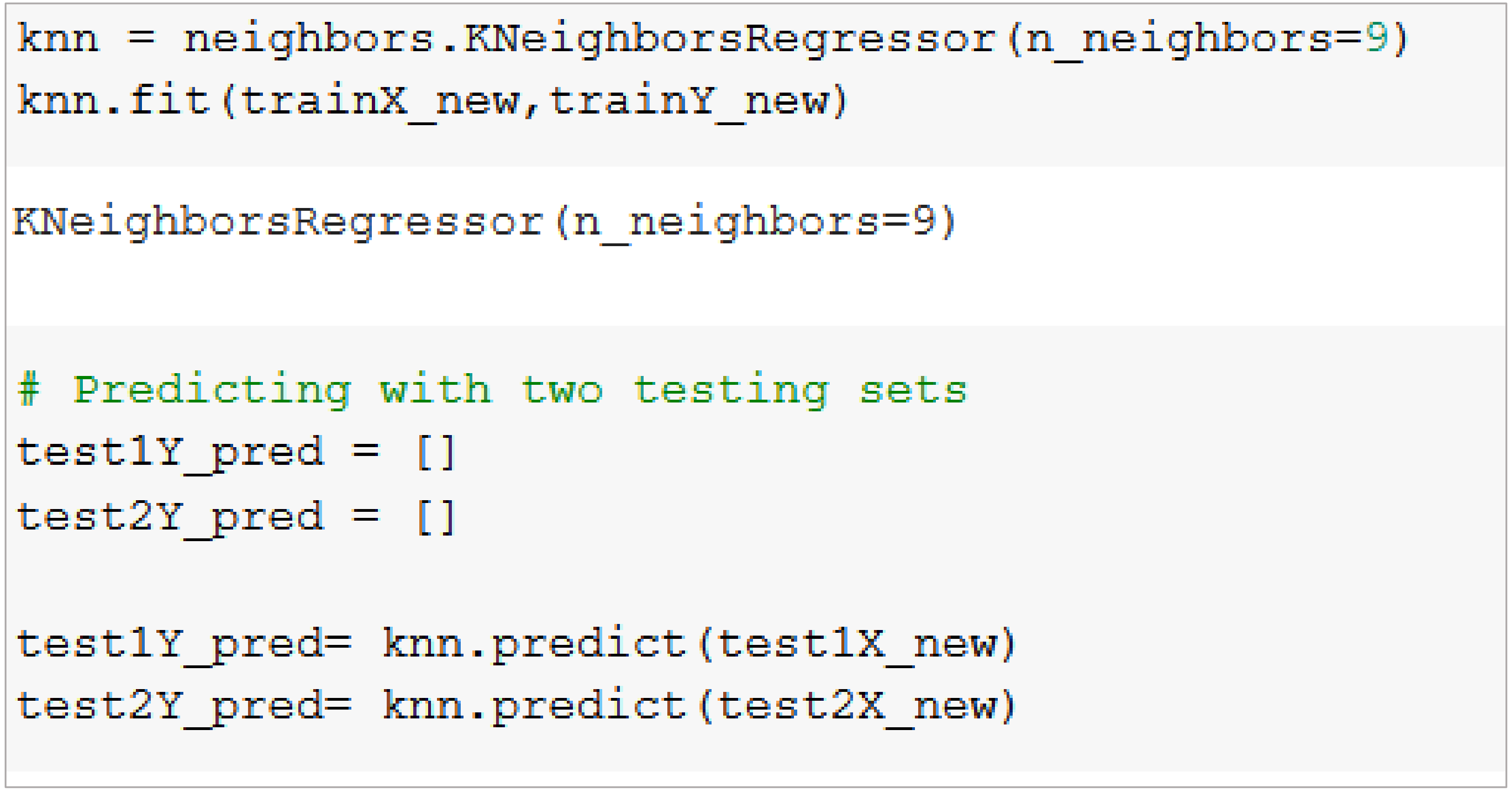
F1	F2	.....	F11	F1	F2	.....	F11	----	----	----	F1	F2	.....	F11	T1	T2	T3	T4

Methodology : Model Development

- We have developed 6 models to perform the timeseries regression forecasting
  - K-Nearest Neighbor
  - Linear Regression
  - Random Forest
  - Decision Tree
  - Long-Short Term Memory model (LSTM)
  - Convolutional Neural Network (CNN)

Methodology : **K-Nearest Neighbor (KNN)**

- Fig3 shows the code snippet for KNN model



```
knn = neighbors.KNeighborsRegressor(n_neighbors=9)
knn.fit(trainX_new,trainY_new)
```

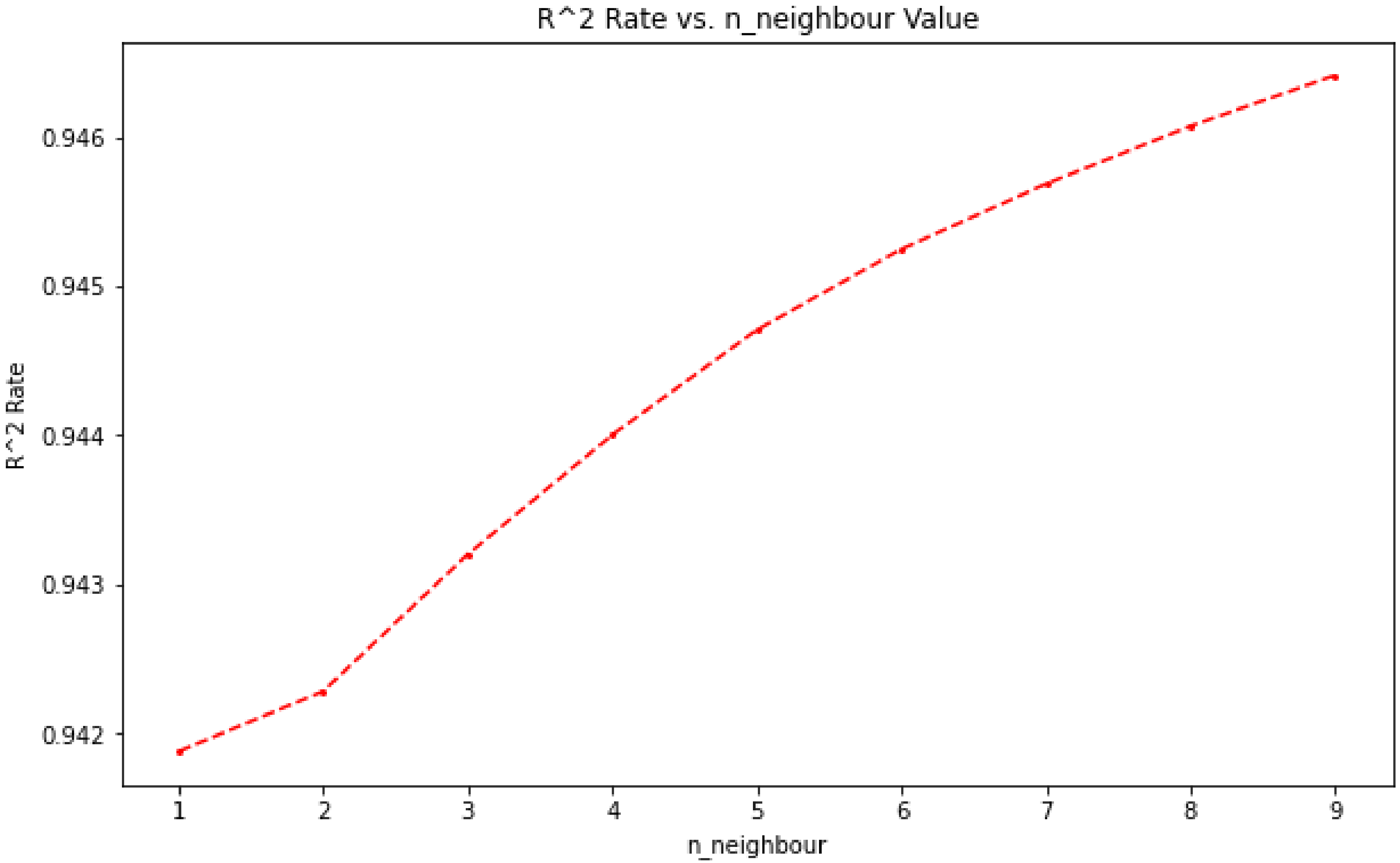
```
KNeighborsRegressor(n_neighbors=9)
```

```
# Predicting with two testing sets
test1Y_pred = []
test2Y_pred = []

test1Y_pred= knn.predict(test1X_new)
test2Y_pred= knn.predict(test2X_new)
```

Methodology : **K-Nearest Neighbor (KNN)**

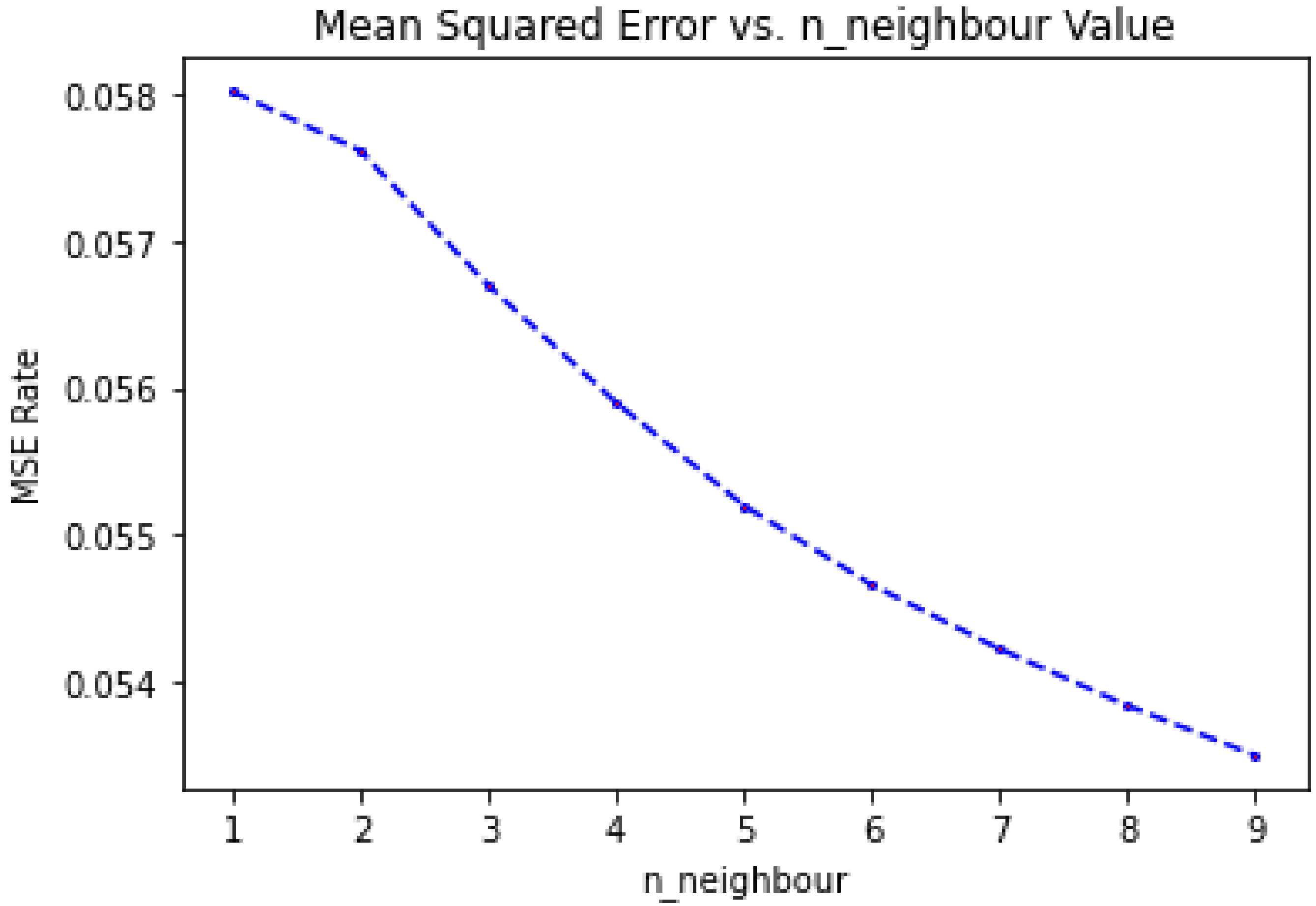
- Hyper-parameters used:
  - n_neighbour = 9

- Fig4 shows the R2 value for different number of neighbour values



Methodology : **K-Nearest Neighbor (KNN)**

- Fig5 shows the MSE value for different number of neighbour values



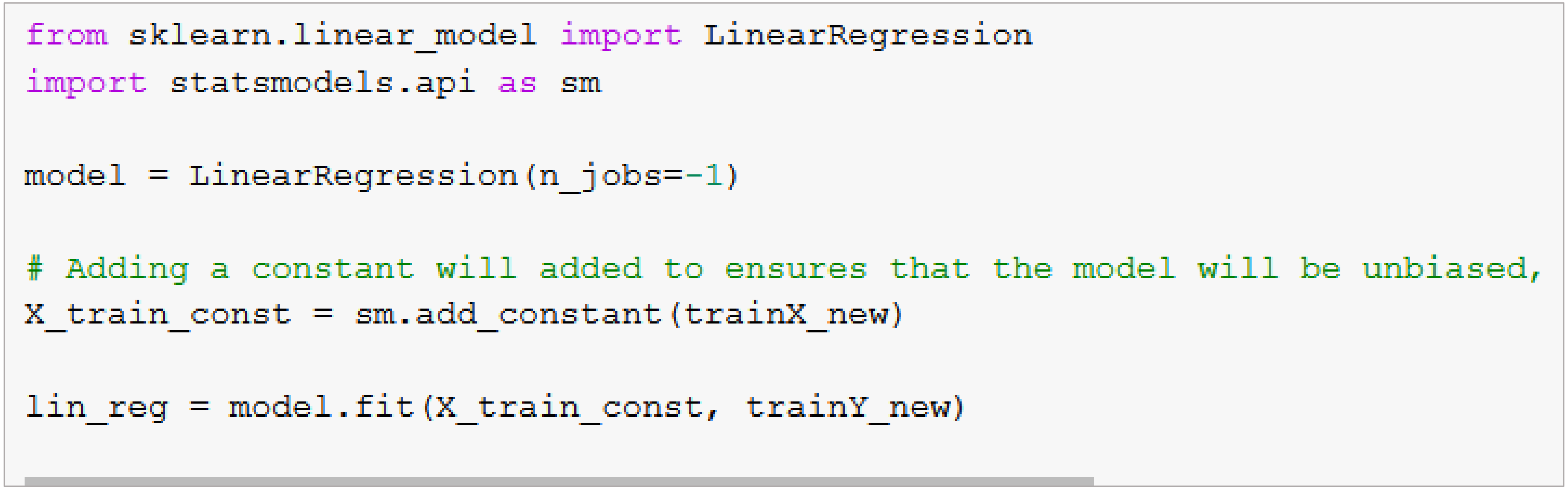
Results : **K-Nearest Neighbor (KNN)**

- Table 1: Results of KNN

Measurement	Stator_tooth		Stator_yoke		Stator _winding		Pm		Average	
	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72
R2	0.9785	0.9783	0.9727	0.9752	0.9548	0.9568	0.8794	0.9126	0.9464	0.9557
MSE	0.0214	0.0216	0.0272	0.0247	0.0450	0.0431	0.1202	0.0871	0.0534	0.0441
MAE	0.1079	0.1068	0.1254	0.1210	0.1547	0.1516	0.2847	0.2405	0.1682	0.1550

Methodology : **Linear Regression**

- Fig6 shows the code snippet for Linear Regression model



```
from sklearn.linear_model import LinearRegression
import statsmodels.api as sm

model = LinearRegression(n_jobs=-1)

# Adding a constant will added to ensures that the model will be unbiased,
X_train_const = sm.add_constant(trainX_new)

lin_reg = model.fit(X_train_const, trainY_new)
```

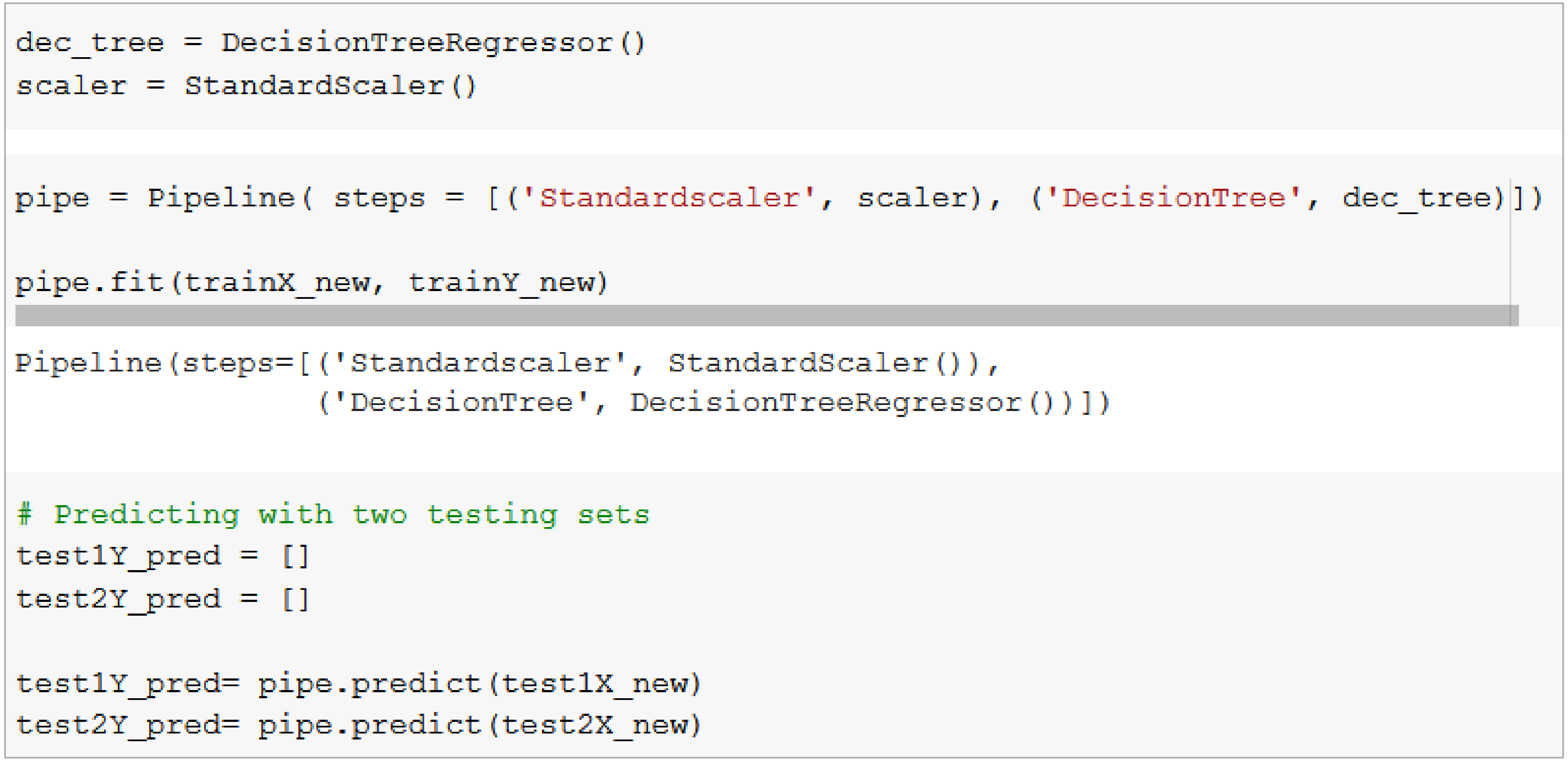
Results : **Linear Regression**

- Table 2: Results of Linear Regression

Measurement	Stator_tooth		Stator_Yoke		Stator _Winding		Pm		Average	
	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72
R2	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9998	0.9999	0.9999	0.9999
MSE	2.761e-06	2.298e-06	4.031e-06	3.226e-06	5.810e-06	4.414e-06	1.027e-04	5.416e-05	2.883e-05	1.602e-05
MAE	0.0011	0.0009	0.0010	0.0011	0.0014	0.00121	0.00411	0.0033	0.00194	0.00166

Methodology : **Decision Tree**

- Fig7 shows the code snippet for Decision Tree model



```
dec_tree = DecisionTreeRegressor()
scaler = StandardScaler()
```

```
pipe = Pipeline( steps = [('Standardscaler', scaler), ('DecisionTree', dec_tree)])

pipe.fit(trainX_new, trainY_new)
```

```
Pipeline(steps=[('Standardscaler', StandardScaler()),
                ('DecisionTree', DecisionTreeRegressor())])
```

```
# Predicting with two testing sets
test1Y_pred = []
test2Y_pred = []

test1Y_pred= pipe.predict(test1X_new)
test2Y_pred= pipe.predict(test2X_new)
```

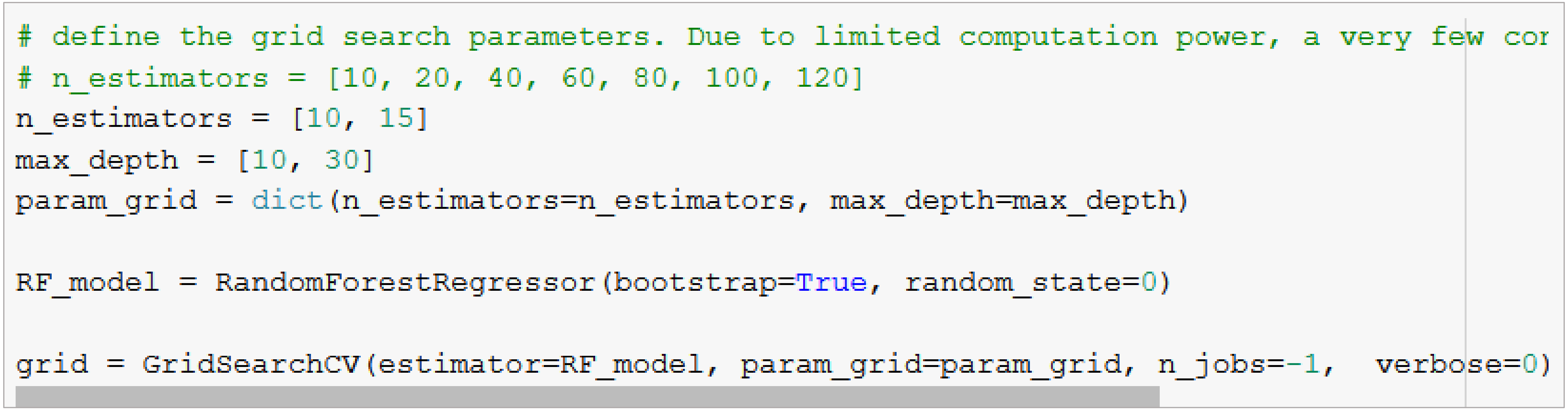
Results : **Decision Tree**

- Table 3: Results of Decision Tree

Measurement	Stator_tooth		Stator_Yoke		Stator _Winding		Pm		Average	
	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72
R2	0.9942	0.9848	0.9934	0.9849	0.9888	0.9796	0.9890	0.9652	0.9914	0.9786
MSE	0.0057	0.0151	0.0065	0.0150	0.0111	0.0203	0.0109	0.0347	0.0085	0.0213
MAE	0.0512	0.0465	0.0581	0.0780	0.0721	0.0627	0.0721	0.0760	0.0634	0.0658

Methodology : **Random Forest**

- Fig8 shows the code snippet for Random Forest model



```
# define the grid search parameters. Due to limited computation power, a very few cor
# n_estimators = [10, 20, 40, 60, 80, 100, 120]
n_estimators = [10, 15]
max_depth = [10, 30]
param_grid = dict(n_estimators=n_estimators, max_depth=max_depth)

RF_model = RandomForestRegressor(bootstrap=True, random_state=0)

grid = GridSearchCV(estimator=RF_model, param_grid=param_grid, n_jobs=-1,  verbose=0)
```

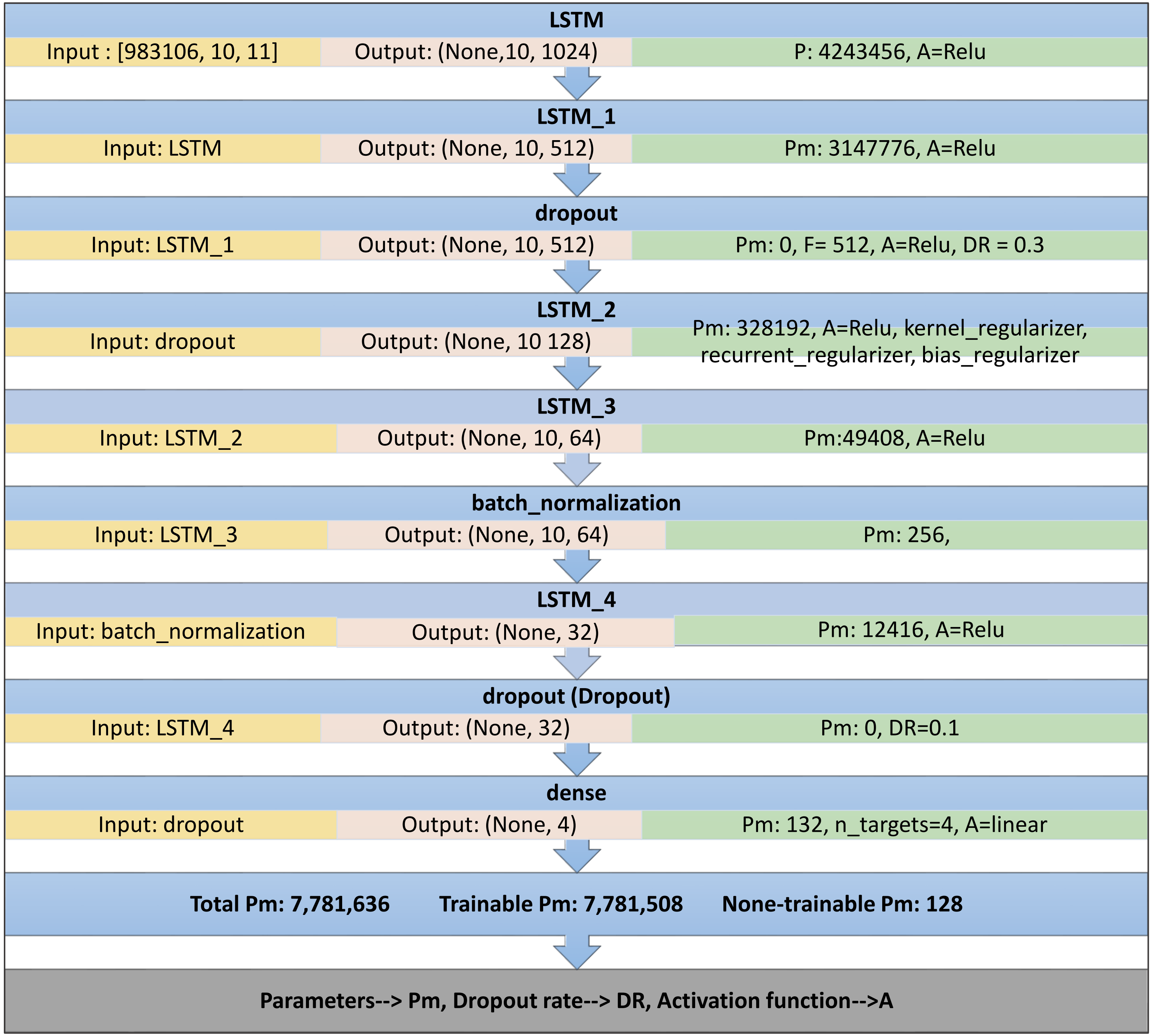
Results : **Random Forest**

- Table 4: Results of Random Forest

Measurement	Stator_tooth		Stator_Yoke		Stator _Winding		Pm		Average	
	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72
R2	0.9992	0.9898	0.9987	0.9909	0.9868	0.9807	0.991	0.9862	0.9932	0.9886
MSE	0.0048	0.0091	0.0095	0.0150	0.0191	0.0199	0.0110	0.0237	0.0075	0.0193
MAE	0.0413	0.0362	0.0583	0.0680	0.0604	0.0597	0.0529	0.0630	0.0594	0.0602

Methodology : **Long Short - Term Memory Model (LSTM)**

- Fig9 shows the architecture of LSTM model



LSTM		
Input : [983106, 10, 11]	Output: (None,10, 1024)	P: 4243456, A=Relu
LSTM_1		
Input: LSTM	Output: (None, 10, 512)	Pm: 3147776, A=Relu
dropout		
Input: LSTM_1	Output: (None, 10, 512)	Pm: 0, F= 512, A=Relu, DR = 0.3
LSTM_2		
Input: dropout	Output: (None, 10 128)	Pm: 328192, A=Relu, kernel_regularizer, recurrent_regularizer, bias_regularizer
LSTM_3		
Input: LSTM_2	Output: (None, 10, 64)	Pm:49408, A=Relu
batch_normalization		
Input: LSTM_3	Output: (None, 10, 64)	Pm: 256,
LSTM_4		
Input: batch_normalization	Output: (None, 32)	Pm: 12416, A=Relu
dropout (Dropout)		
Input: LSTM_4	Output: (None, 32)	Pm: 0, DR=0.1
dense		
Input: dropout	Output: (None, 4)	Pm: 132, n_targets=4, A=linear
Total Pm: 7,781,636	**Trainable Pm: 7,781,508**	**None-trainable Pm: 128**

Parameters--> Pm, Dropout rate--> DR, Activation function-->A

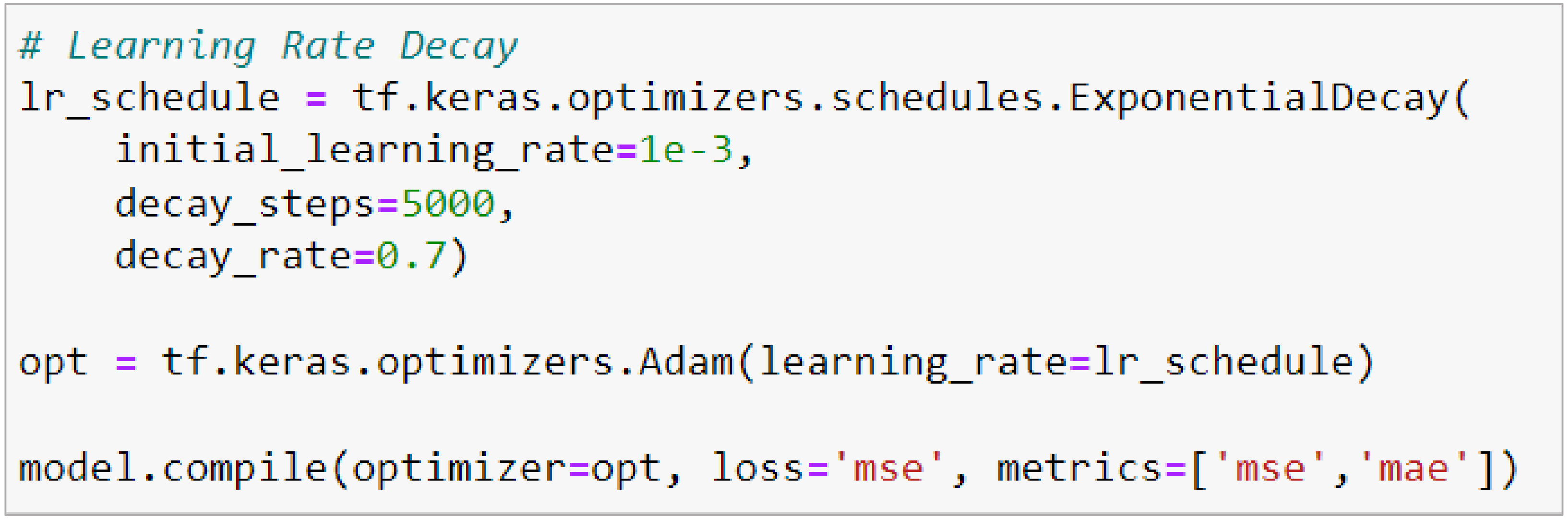
Methodology : Long Short -Term Memory Model (LSTM)

- Table shows the hyper-parameter list for the LSTM model

Hyperparameter/parameter	**Configuration**
Number of LSTM layers	5
Last layer activation	linear
Total number of parameters	7,781, 636
Initial learning rate	0.001
epochs	20
Batch size	1000
optimizer	Adam
Loss function	MSE

Methodology : **Long Short -Term Memory Model (LSTM)**

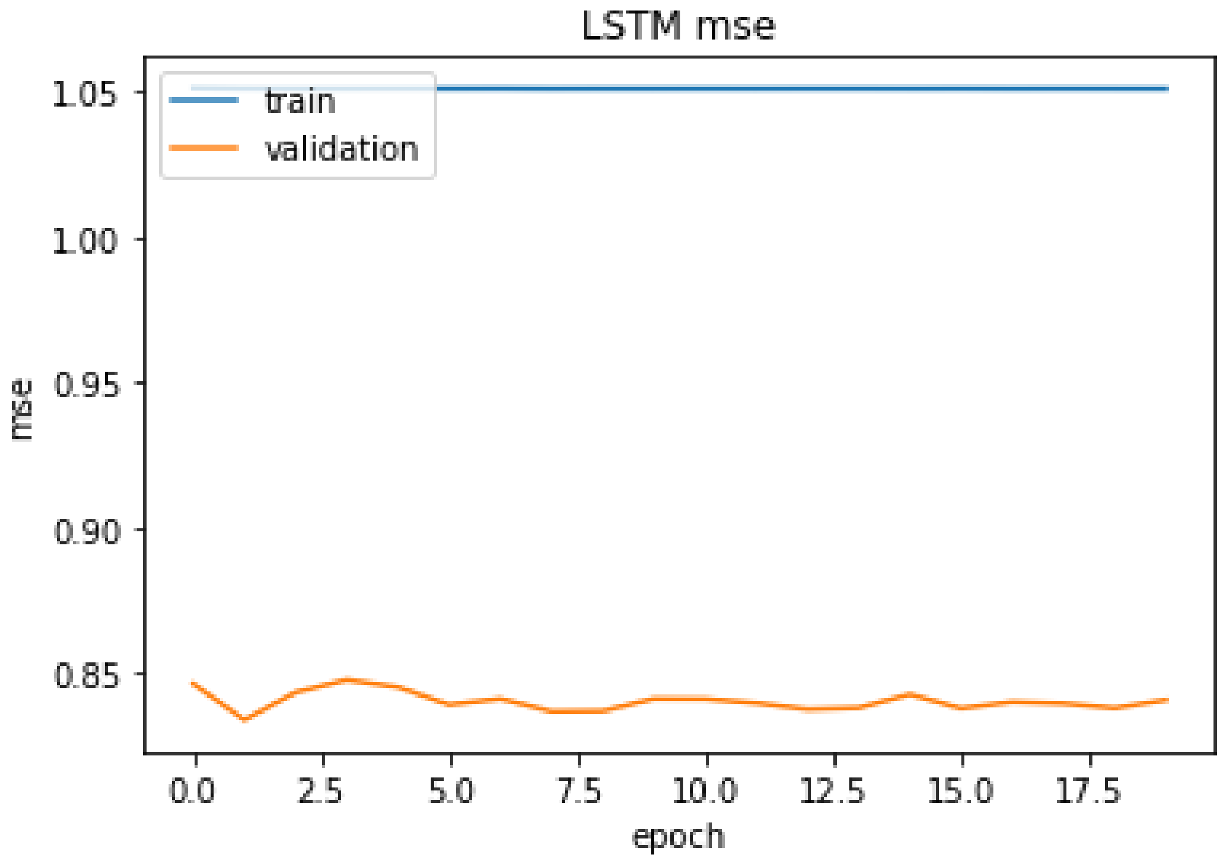
- Techniques taken to avoid overfitting and improve performance
  - kernel_regularizer=l2(0.01)
  - recurrent_regularizer=l2(0.01)
  - bias_regularizer=l2(0.01)
  - Dropout(0.3) and Dropout(0.1)
  - Batch normalization
  - Early-stopping
  - Learning rate decay

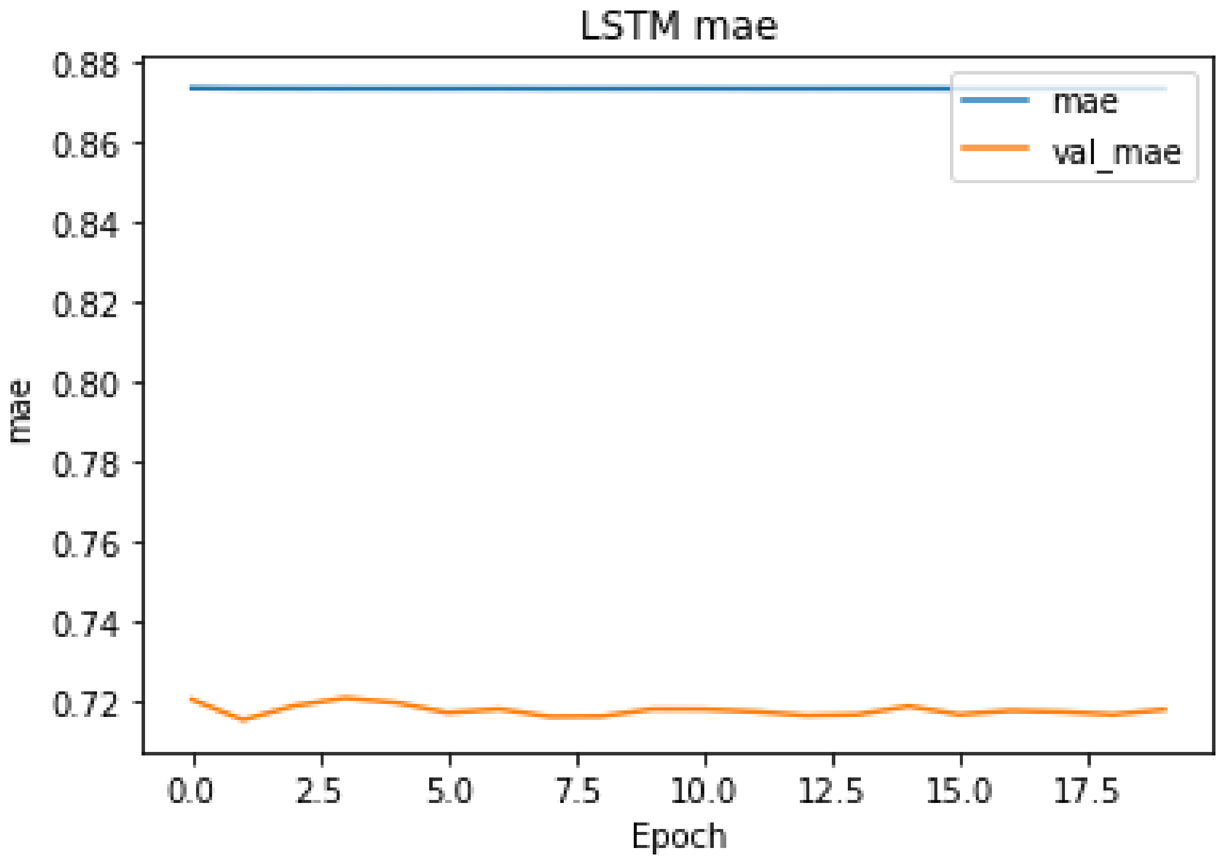


```
# Learning Rate Decay
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-3,
    decay_steps=5000,
    decay_rate=0.7)

opt = tf.keras.optimizers.Adam(learning_rate=lr_schedule)

model.compile(optimizer=opt, loss='mse', metrics=['mse','mae'])
```

- Fig10 shows the code snippet of learning rate decay schedular

Results : **Long Short- Term Memory Model (LSTM)**





- Fig11 shows the MSE Vs. Epoch plot for LSTM training and validation

- Fig12 shows the MAE Vs. Epoch plot for LSTM training and validation

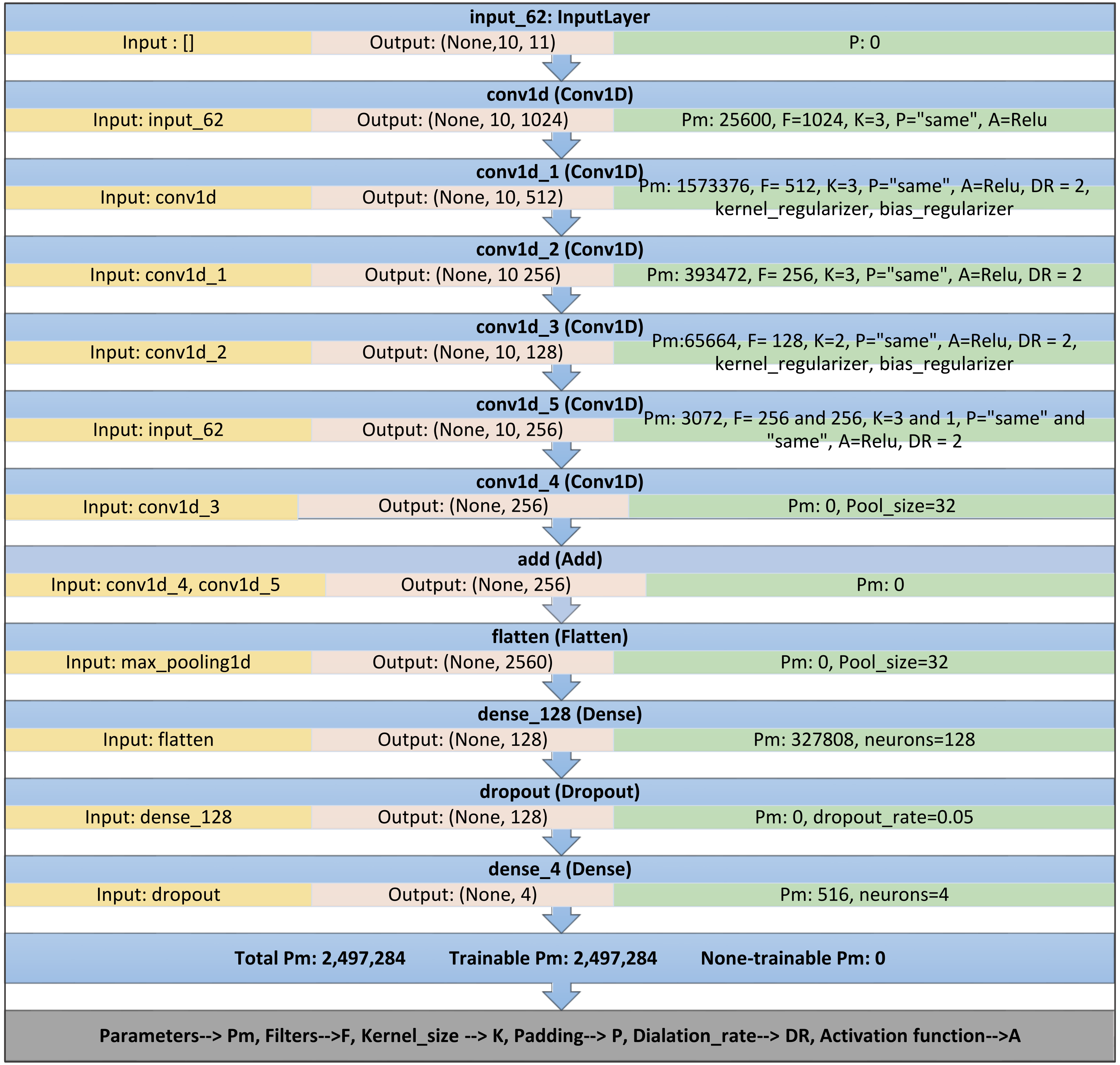
Results : **Long Short -Term Memory Model (LSTM)**

- Table 5: Results of LSTM

Measurement	Stator_tooth		Stator_Yoke		Stator _Winding		Pm		Average	
	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72
R2	-0.0095	-0.0094	-0.0170	-0.0170	-0.0041	-0.0094	-0.0057	-0.0055	-0.0091	-0.0090
MSE	1.0089	1.0088	1.0168	1.0164	1.0033	1.0035	1.0030	1.0042	1.0080	1.0082
MAE	0.8763	0.8712	0.8507	0.8584	0.8835	0.8811	0.8507	0.8442	0.8717	0.8637

Methodology : **Convolutional Neural Network Model (CNN)**

- Fig13 shows the architecture of the CNN model



input_62: InputLayer

Input : [] | Output: (None,10, 11) | P: 0

conv1d (Conv1D)

Input: input_62 | Output: (None, 10, 1024) | Pm: 25600, F=1024, K=3, P="same", A=Relu

conv1d_1 (Conv1D)

Input: conv1d | Output: (None, 10, 512) | Pm: 1573376, F= 512, K=3, P="same", A=Relu, DR = 2, kernel_regularizer, bias_regularizer

conv1d_2 (Conv1D)

Input: conv1d_1 | Output: (None, 10 256) | Pm: 393472, F= 256, K=3, P="same", A=Relu, DR = 2

conv1d_3 (Conv1D)

Input: conv1d_2 | Output: (None, 10, 128) | Pm:65664, F= 128, K=2, P="same", A=Relu, DR = 2, kernel_regularizer, bias_regularizer

conv1d_5 (Conv1D)

Input: input_62 | Output: (None, 10, 256) | Pm: 3072, F= 256 and 256, K=3 and 1, P="same" and "same", A=Relu, DR = 2

conv1d_4 (Conv1D)

Input: conv1d_3 | Output: (None, 256) | Pm: 0, Pool_size=32

add (Add)

Input: conv1d_4, conv1d_5 | Output: (None, 256) | Pm: 0

flatten (Flatten)

Input: max_pooling1d | Output: (None, 2560) | Pm: 0, Pool_size=32

dense_128 (Dense)

Input: flatten | Output: (None, 128) | Pm: 327808, neurons=128

dropout (Dropout)

Input: dense_128 | Output: (None, 128) | Pm: 0, dropout_rate=0.05

dense_4 (Dense)

Input: dropout | Output: (None, 4) | Pm: 516, neurons=4

Total Pm: 2,497,284   Trainable Pm: 2,497,284   None-trainable Pm: 0

Parameters--> Pm, Filters-->F, Kernel_size --> K, Padding--> P, Dialation_rate--> DR, Activation function-->A

Methodology : **CNN**

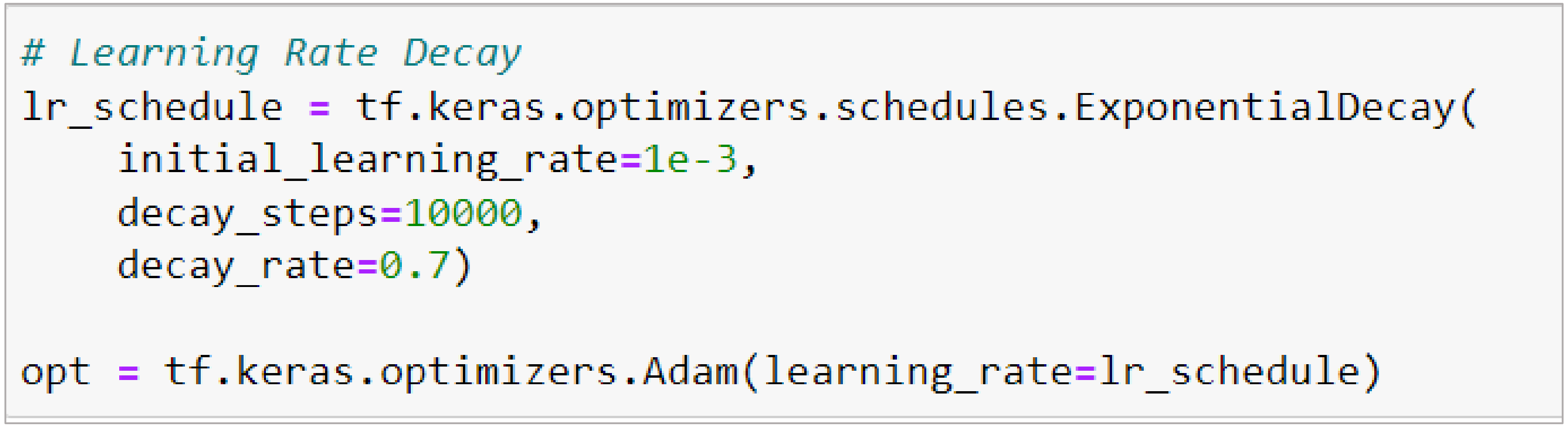
- Table 6 shows the hyper-parameter/parameter list for the CNN model

Hyperparameter/parameter	**Configuration**
Number of Conv layers	6
Last layer activation	linear
Total number of parameters	2,497,284
Initial learning rate	0.001
epochs	50
Batch size	1000
optimizer	Adam
Loss function	MSE

Methodology : **CNN**

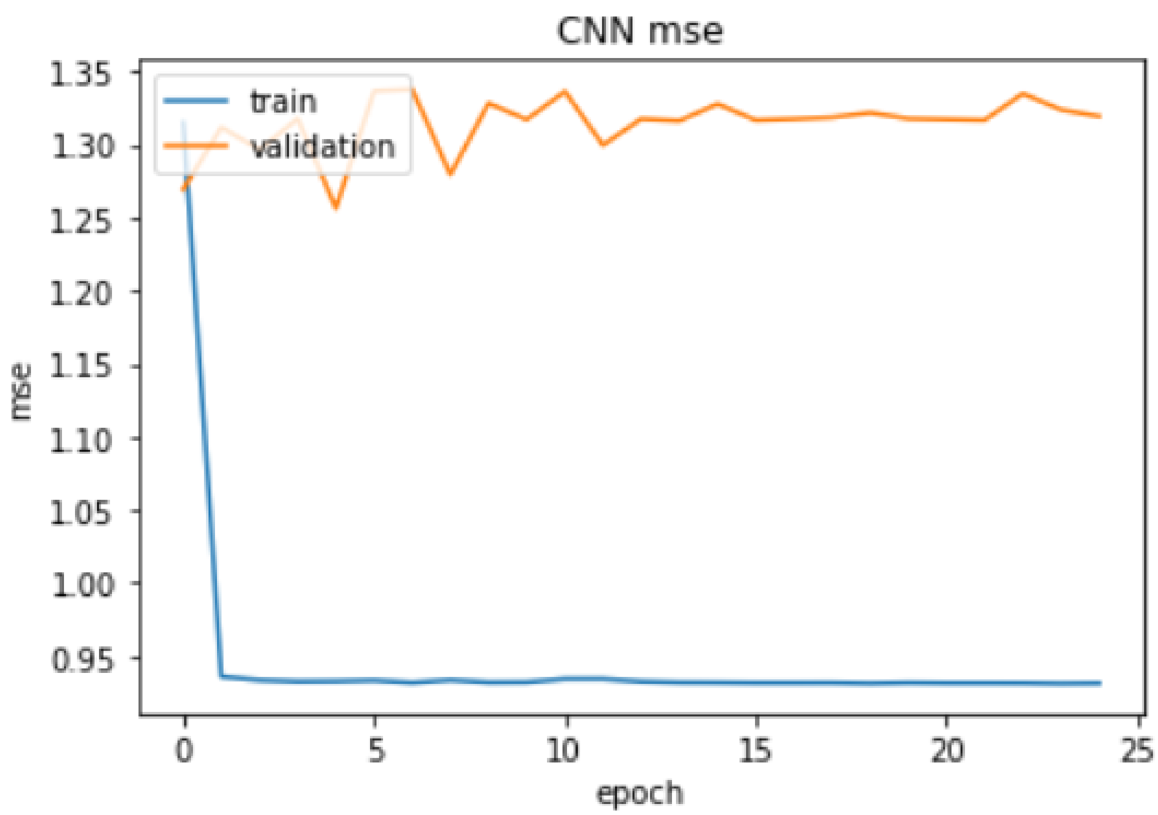
- Techniques taken to avoid overfitting and improve performance
  - kernel_regularizer=l2(0.01)
  - recurrent_regularizer=l2(0.01)
  - bias_regularizer=l2(0.01)
  - Dropout(rate=0.05)
  - Early-stopping
  - Learning rate decay

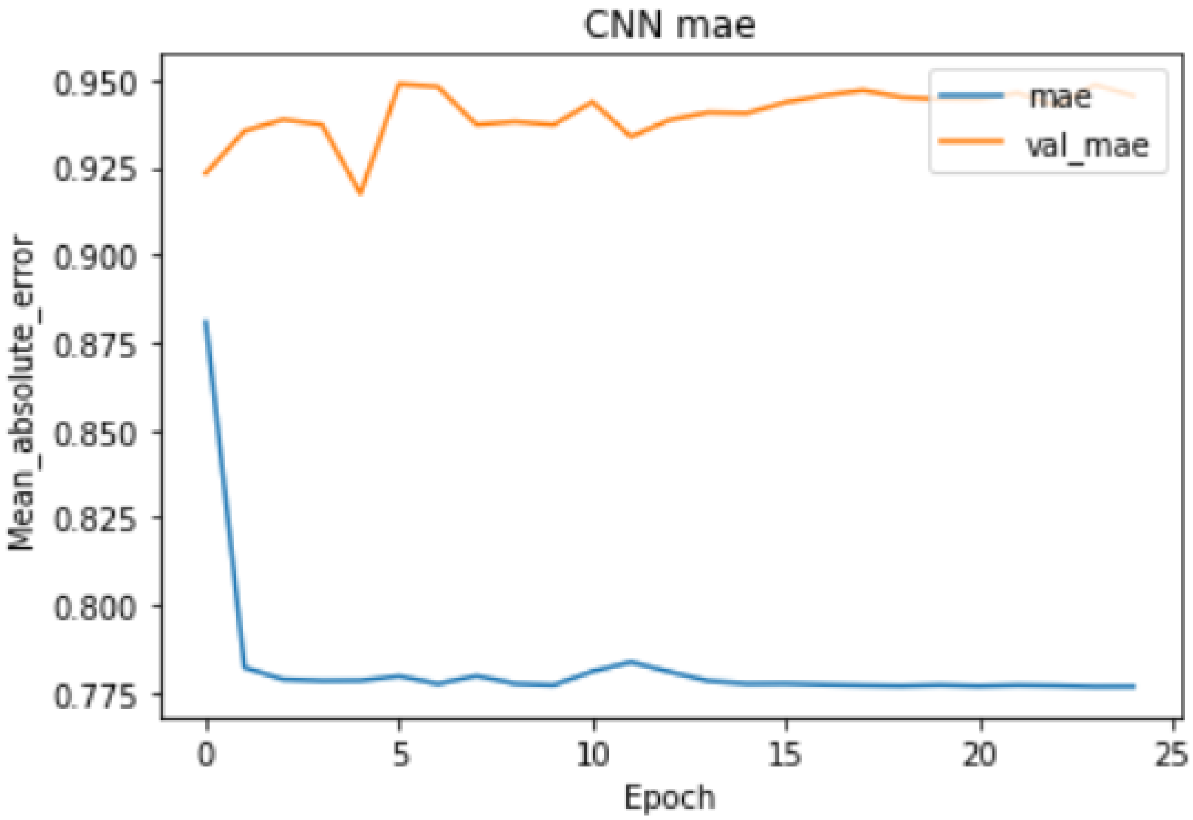
- Fig14 shows the code snippet of learning rate decay schedular



```
# Learning Rate Decay
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-3,
    decay_steps=10000,
    decay_rate=0.7)

opt = tf.keras.optimizers.Adam(learning_rate=lr_schedule)
```

Results : **CNN**





- Fig11 shows the MSE Vs. Epoch plot for CNN training and validation

- Fig12 shows the MAE Vs. Epoch plot for CNN training and validation

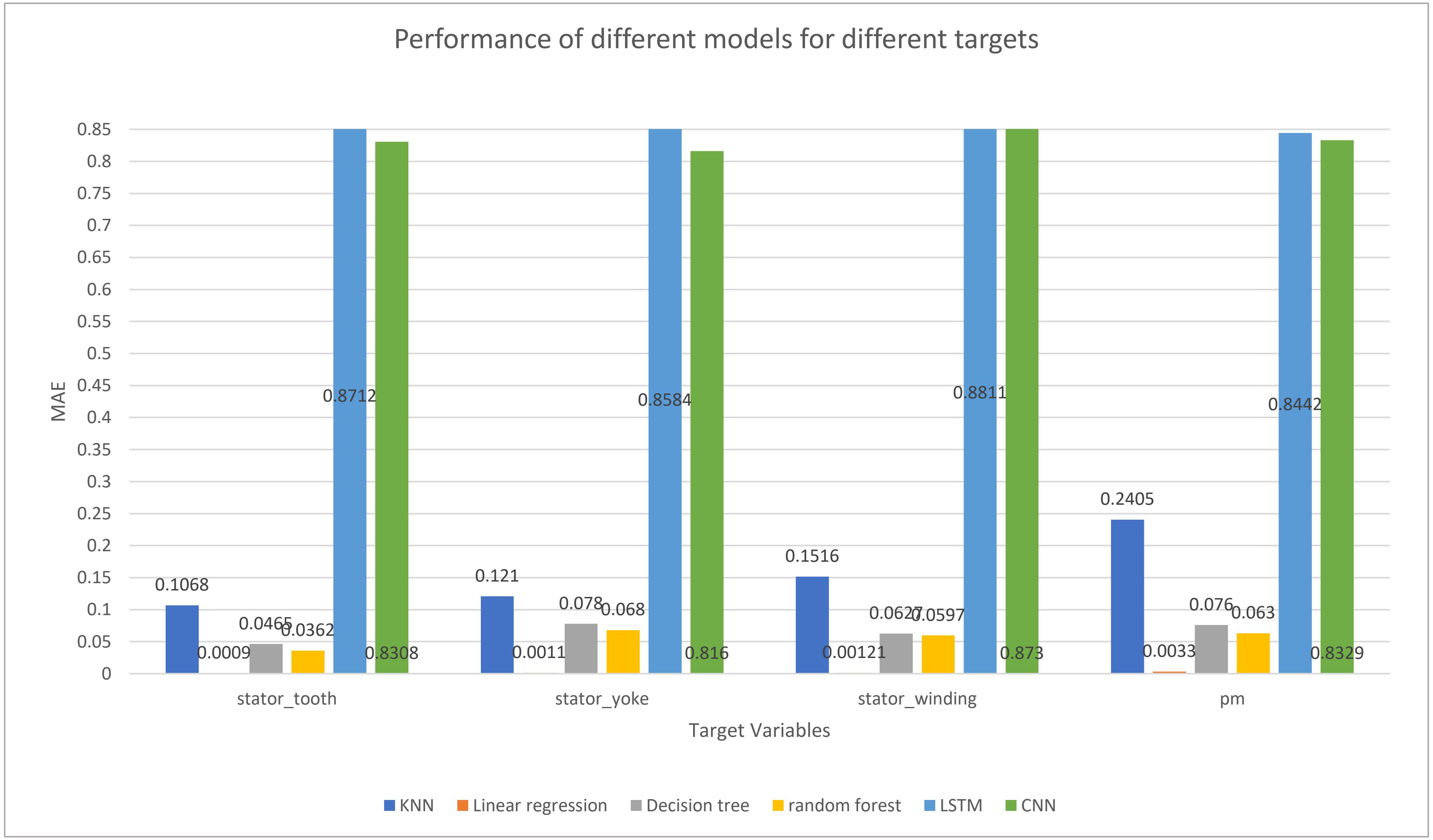
Results : **CNN**

- Table 7: Results of CNN

Measurement	Stator_tooth 3		Stator_Yoke 2		Stator4 _Winding		Pm 1		Average	
	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72	PID 65	PID 72
R2	0.01924	0.0880	0.08141	0.0227	0.02134	0.06461	-0.0011	0.01226		
MSE	0.9796	0.9343	0.9181	0.9108	0.9996	0.9996	0.9736	0.9745	0.9677	0.9516
MAE	0.8667	0.8308	0.8398	0.8160	0.8839	0.8730	0.8423	0.8329	0.8582	0.8382

Discussion and Conclusion

- Fig15 shows a bar plot of performance of each model for predicting each target variable.



Discussion and Conclusion

- EDA helped to familiarize with the dataset and to identify the correlation between different attributes.
- Machine learning models performed better than Deep Learning models
    - Higher goodness of fit value
    - Lower MSE and MAE
    - Less computational requirements
    - Faster
- The best Machine Learning model is : Linear Regression
- Performance of PID65 > PID72
- Limitations: Higher computational power required for deep models and GridSerachCV function

Future Work

- Perform on **different datasets**
  - different motors of the same manufacturer or even among different manufacturers.
- Improve the performance of deep learning models.

Acknowledgement

- We would like to thank professor Thangarajah Akilan for initiating this group project to enhance our theoretical, technical and group working skills.

- We thank Paderborn University and Dr. Ing Joachim Bocker for the electric temperature dataset.

References

- [1] W. Kirchg¨assner, O. Wallscheid, and J. B¨ocker, “Data-Driven Permanent Magnet Temperature Estimation in Synchronous Motors with Supervised Machine Learning: A Benchmark,” IEEE Trans. Energy Convers., vol. 36, no. 3, pp. 2059–2067, 2021.
- [2] W. Kirchgassner, O. Wallscheid, and J. Bocker, “Estimating Electric Motor Temperatures with Deep Residual Machine Learning,” IEEE Trans. Power Electron., vol. 36, no. 7, pp. 7480–7488, 2021.
- [3] O. Wallscheid, W. Kirchgassner, and J. Bocker, “Investigation of long short-term memory networks to temperature prediction for permanent magnet synchronous motors,” Proc. Int. Jt. Conf. Neural Networks, vol. 2017-May, pp. 1940–1947, 2017.
- [4] W. Kirchgassner, O. Wallscheid, and J. Bocker, “Deep residual convolutional and recurrent neural networks for temperature estimation in permanent magnet synchronous motors,” 2019 IEEE Int. Electr. Mach. Drives Conf. IEMDC 2019, pp. 1439–1446, 2019.
- [5] H. Guo, Q. Ding, Y. Song, H. Tang, L. Wang, and J. Zhao, “Predicting temperature of permanent magnet synchronous motor based on deep neural network,” Energies, vol. 13, no. 18, 2020.
- [6] R. Le, K. He, and A. Hu, “Motor Temperature Prediction with K-Nearest Neighbors and Convolutional Neural Network,” pp. 3–9, 2019.
- [7] K. Anuforo and V. Milosavljevic, “Temperature Estimation in Permanent Magnet Synchronous Motor (PMSM) Components using Machine Learning,” 2020.

References

- [8] W. Kirchg¨assner, O. Wallscheid, and J. B¨ocker, “Data-Driven Permanent Magnet Temperature Estimation in Synchronous Motors with Supervised Machine Learning: A Benchmark,” IEEE Trans. Energy Convers., vol. 36, no. 3, pp. 2059–2067, 2021.
- [9] W. Kirchgassner, O. Wallscheid, and J. Bocker, “Estimating Electric Motor Temperatures with Deep Residual Machine Learning,” IEEE Trans. Power Electron., vol. 36, no. 7, pp. 7480–7488, 2021.
- [10] O. Wallscheid, W. Kirchgassner, and J. Bocker, “Investigation of long short-term memory networks to temperature prediction for permanent magnet synchronous motors,” Proc. Int. Jt. Conf. Neural Networks, vol. 2017-May, pp. 1940–1947, 2017.

Thank you