Statistical Learning for Predictive Maintenance

Sam Cartwright, Student 461871

School of Aerospace, Transport and Manufacturing
16th of November 2024 – Word Count: 2998 (Not including Introduction)

Information categorisation:
Open

1 Introduction and Background

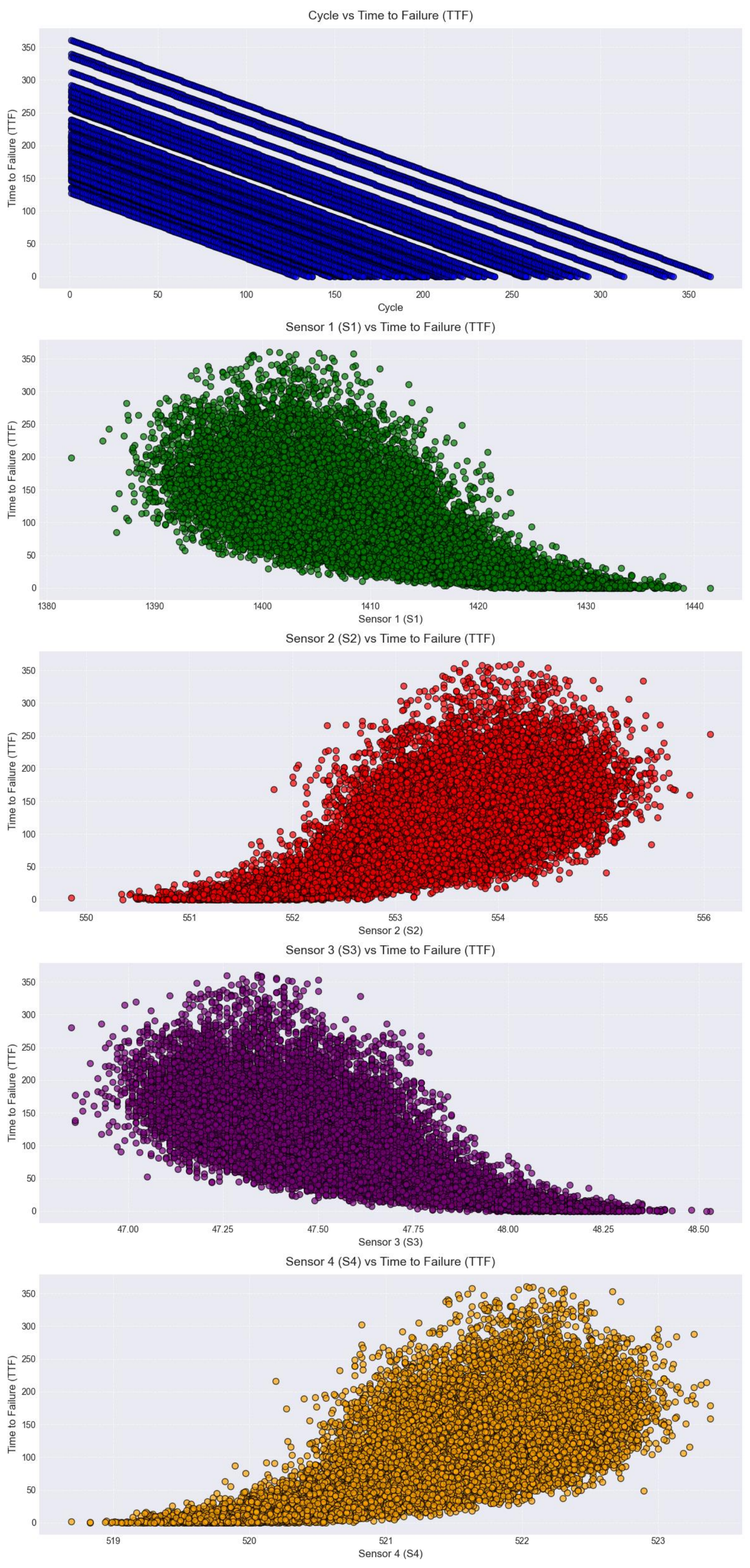
Predictive maintenance refers to a maintenance strategy that uses Data Analytics, Machine Learning, and other predictive techniques to forecast when equipment or machinery is likely to fail, allowing maintenance to be performed before a breakdown occurs [1]. It is a method that emerged mainly due to “the shortcomings of traditional preventative maintenance” [2]; this could include inflexibility, long downtimes or poor allocation of resources, all of which drive costs up. Ultimately then, it is the case that the application of statistical learning in this area has a clear focus of saving money by streamlining the maintenance procedure.

This short report focuses on the implementation of Statistical Learning Methods for predictive maintenance, specifically in the Aerospace industry, and should serve as an introduction to the opportunities that predictive maintenance offers in the field. The work is split into two main sections, highlighting some major use cases for Aircraft manufacturers. The first of which is a Time-To-Failure (TTF) prediction for an engine using a regression model, and the second is to use a binary classification model to predict whether an engine can be considered faulty within a specified cycle. Each section includes discussion on the selection of model and the results with consideration of the wider context. The data to complete the work consists of two csv files. One is a training file that contains measurements of 4 simulated sensors for 100 aircraft engines running up to failure, with the other being a test file, with measurements from the 4 sensors at randomly selected cycles of engine operation. Both files also contain labels for classification, set to 1 when TTF is <30 cycles, as well as labels for each specific engine. The data has been extracted from a larger dataset generated by Microsoft, that was used for a project within the Springboard DS Career Track Bootcamp.

The implementation of the models in this report is completed using Python 3 code, within a Jupyter Notebook. Some code is discussed within the main body of the report where necessary, with the full code used for each model being available in the appendix.

1.1 Data Analysis

Before approaching the statistical learning problems, analysing the data to gain a greater understanding of the distributions provided by each sensor is important, aiding in the selection of method and to help read into results. Firstly, each variable is plotted against TT.. For ease, this is completed using the matplotlib library in python.



Graphs 1-5 Scatter graphs plotting each sensor against TTF, and cycles against TTF

As expected, due to the TTF measurement essentially being ‘Remaining Cycles’, there is a perfect inverse proportionality between cycles and TTF. It could be interesting to note that the majority of the engines fail after around 150-250 cycles, and only four engines reach over 300 cycles. These could be considered outliers and therefore could be omitted, or make some attempt to reduce their weightings when applied to the models used, especially for regression. However, it would be important for a model predicting TTF for an engine in a real-life application to be able to generalise even in the case of outliers, therefore they are left in for this simple case. A larger dataset could be used in practice to offset this.

The four sensors themselves display very similar characteristics. Firstly, sensors 1 and 3 display a distribution that approaches higher readings when TTF is lowered, and sensors 2 and 4 show the opposite. This is helpful for a potential classification model as the greater difference between the sensors when TTF is lower is very identifiable. When TTF is higher, sensor measurements seem to be much more distributed with a greater variance. However, when TTF reaches below 50, each of the sensors begins to converge much more in their readings. This likely means that a TTF prediction when low will be much more accurate than when TTF is high, an important observation to be addressed later. Finally, each of the sensors provide readings in a small range - for example sensor 4, between around 523-519 – and at very differing values. This is important for the selection of the model, with normalisation of the data also being brought into question.

2 Regression Model for TTF Prediction

This section of the work covers the selection, implementation and results of a regression model in order to predict TTF for an engine based upon available variables. The aim is to provide the most accurate predictions for TTF, with the selected model being used to discover underlying patterns in the data that may be too difficult to detect manually.

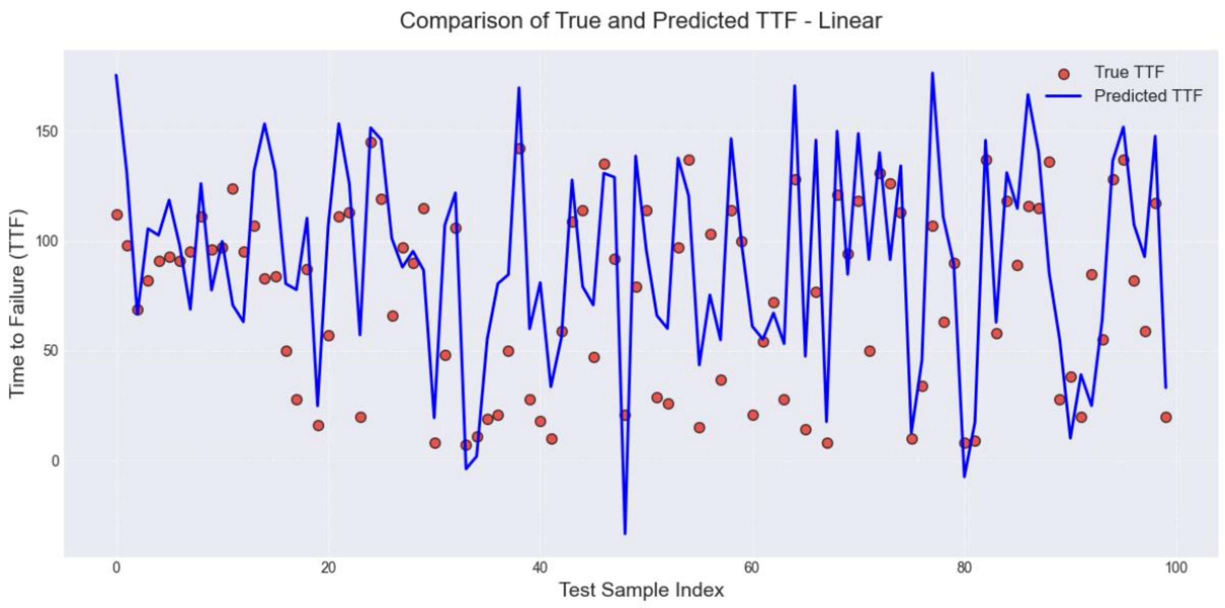
2.1 Model Selection and Implementation

With the knowledge of the data distribution, evaluation of the best models to use is made easier. We can see some linearity from the sensor readings, but large portions of the data for each case display high levels of variance away from what would be easily regressed with a simple line. A deep learning model could be used here to potentially predict with greater accuracy, but with a large decrease in efficiency for relatively small changes in accuracy vs simpler models for a dataset such as used in this report, this is generally avoided in cases of regression. If the data provided was much more complex without clear patterns, the use of deep learning would become more viable. With this in mind, two simple models are chosen initially to determine the best route. One is to use a Simple Linear Regression model, and the other a Random Forest Regressor. Both could have their advantages in this case, with a linear regression model being extremely efficient to train, and easy interpretability, compared to the Random Forest which takes longer to train as well as needing hyperparameter tuning to ensure best possible results.

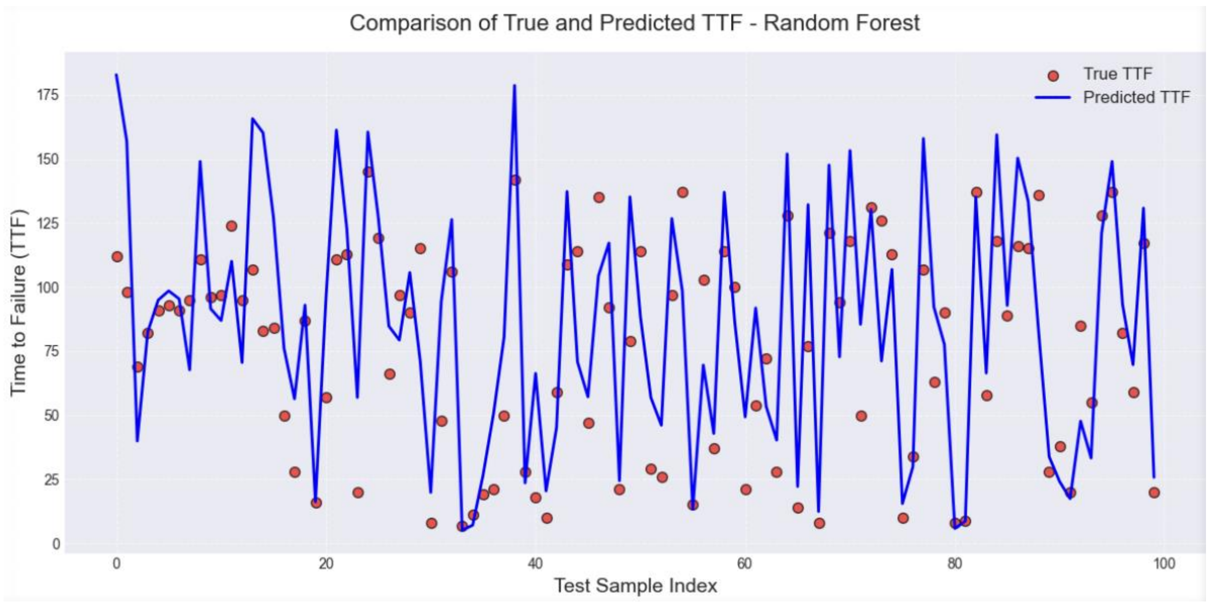
The metrics used for model evaluation in regression are based significantly around evaluating the predictive performance of the model [3], and so error metrics are used to provide the best insight. Three are chosen in this case to provide a rounded view of the results. Mean absolute error, or MAE measures the average difference between predicted values and the ground truth, and is something we want to minimise. Root mean squared error, or RMSE - which is simply the square root of the MSE, to help understand the result clearer in the context of the problem – places some more emphasis on the effect of outliers and can be used to evaluate how the model can deal with data points that are unexpected. Like MAE, it is to be minimised. Finally, R^2 is used which “indicates the proportion of the variance in the dependent variable that is predictable from the independent variables" [3]. It is between 0 and 1 (unless the prediction is worse than simply using a mean), and a higher R^2 indicates the model explains more variance in the data.

For the setup of the data in the notebook before model implementation, both files 'train_selected.csv' and 'test_selected.csv' are imported and stored using pandas. The true TTF values for the test data are also imported and stored in an array. The data is normalised using a Min-Max Scaler (Appendix A). This technique is preferred to other scaling options in this case due to the nature of the data. Using a scaler that simply divides each value by the largest reading for each sensor would result in bunched data as the values have a very small range. The Min-Max Scaler avoids this issue by normalising the values across 0-1, keeping the relative distances between each point. It is also worth noting that another benefit of the Random Forest Regressor can often be used without the requirement of normalising data. Even so, it is used here to ensure best possible accuracy, and because the dataset is not particularly large.

Following this, each model is implemented simply, to determine the performance before choosing the better model to fine tune (Appendix B). It is found that the Random Forest Regressor had a lower RMSE of 28.6 compared to 32.4 for the linear model. This goes to show that even without hyperparameter tuning the random forest model is more accurate, suggesting the data contains more non-linear patterns more easily defined by the random forest. On top of this, the random forest model boasts a significant difference in R^2 value of 0.53 vs the linear model's 0.39. This value shows that the random forest model captures more variance from the data than the linear model.



Information categorisation:
Open



Graphs 6-7 True TTF vs Prediction for each Model

The two graphs are also produced to display the accuracy of each model in predicting the TTF. This helps to visualise the difference between the two models, as the random forest can be seen to reach points that the linear model failed to, showing its greater understanding of the data.

From this analysis, it is decided that the Random Forest Model is superior, and some parameter tuning is completed to try to minimise RMSE before results are discussed. A grid search is completed (Appendix C) for all the major parameters, with the 'Best' model being used in the final analysis. A grid search can be computationally expensive due to iterations increasing exponentially with each parameter. In many cases a random search may be preferred, however due to the relatively small dataset a grid search is completed within reasonable time.

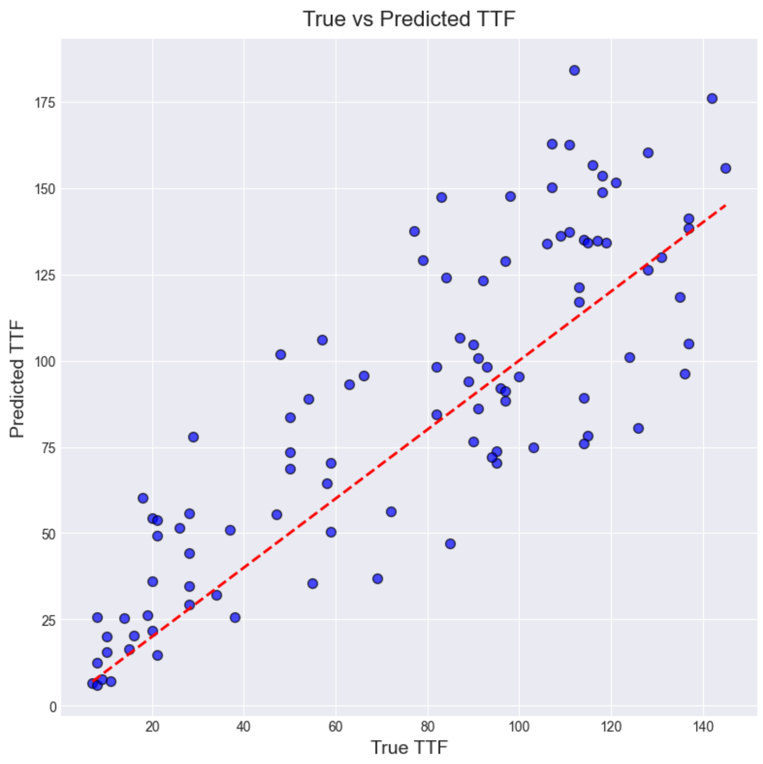
2.2 Analysis and Discussion of Model Results

The grid search allows for identification of the parameters that give the most successful results, which were outputted and used for analysis. The parameters are as follows; n_estimators=500, max_features='sqrt', max_depth=7, min_samples_leaf=5, min_samples_split=2, random_state=42.

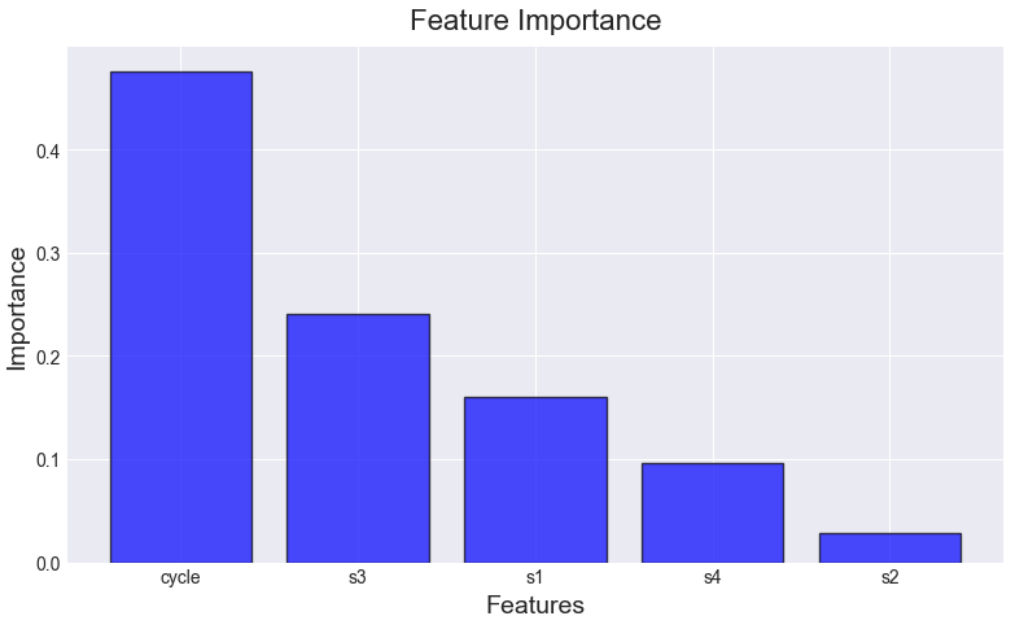
The number of estimators refers to the total number of trees to be constructed that will come to a prediction for each point, and the random state set to 42 ensures that results can be repeatable by assigning a specific sequence to the randomness exhibited by the random forest regressor. The other parameters are all features of each tree to be constructed.

The final values for the three performance metrics after fine tuning the random forest regressor are 27.7, 22.1, and 0.56 for RMSE, MAE and R^2 respectively. Upon first glance at these results, this is only a relatively small improvement from the initial model. The difference between RMSE and MAE suggests the model has some struggle predicting outliers, and the MAE of 22.1 itself is quite poor. In the context of the problem, this means the average prediction for TTF is over 22 cycles away from the correct value. In the aircraft industry, safety is paramount, especially regarding engine failure, and this average suggests that the use of predictive maintenance in this fashion may not be viable. However, there are some other points of interest within the results outside these metrics.

Firstly, a plot of the predicted TTF vs True TTF, as seen below in Graph 8, seems to suggest that when True TTF is closer to zero, predicted TTF is more accurate. This is promising as in practical use, predictions when there are fewer cycles left would be much more important than a prediction with many cycles left. The graph also shows that the model is more likely to overpredict TTF than underpredict. This may be influenced by the small number of outliers of engines mentioned in the earlier data analysis, leading the model to believe that a high TTF is more common than the true case.



Graph 8 A plot showing deviation of Predicted vs True TTF



Graph 9 Feature Importance Plot

From the feature importance graph it is clear to see that cycles are by far the most important variable for the regressor, with almost double the importance of the highest ranked sensor. This is likely due to the linearity of the ‘cycles’ variable, and it’s very direct and predictable relationship with TTF. This makes it easier for the model to weight cycle number heavier when making predictions. Another important observation is the extremely low importance placed on sensor 2. At that level of influence, it would suggest that removing sensor 2 may be a feasible solution, which would likely not have a large effect on predictions, as well as save money for the manufacturer when gathering data.

Focus on the sensors also bring into question the assumption that every sensor is accurate, providing no incorrect readings. An analysis on, this especially from engine to engine could provide valuable in predicting TTF, as the removal of faulty readings, or noise, would lead to an improvement in accuracy of the predictor. Another assumption made that could be impactful is that all engines are considered equal or of the same type, which may not be correct.

Finally, a consideration of how to improve the results is important. One suggestion would be to produce extra variables from the available data, such as rolling features of the sensors. This would involve construction of variables that produced values based on the past data for each sensor, for example up to 5 readings backwards, which would update when new data is obtained. Due to the nature of the data used in this report for testing, this was not something that could be included, but it is reasonable to suggest that in real time practice a model would be able to use past data points to predict TTF at a given point. This would likely allow greater accuracy in predictions as the model would be able to pick up on trending patterns in sensor readings from point to point. Another potential improvement would be to normalise the TTF data in training using mean and standard deviation, removing values outside 3 standard deviations, to remove outliers negatively influencing the model. On top of this, it would be expected that a larger database of values for training would allow for greater accuracy, as the influence of outliers could be reduced, and perhaps more patterns could be uncovered by the model.

3 Binary Classification Model

This section of the work covers the selection, implementation and discussion of a binary classification system, using the same dataset as the regression model, to be able to predict whether an engine is within the last 30 cycles before failure. A successful model would produce an output of 1 when an engine is within its final 30 cycles, and 0 otherwise.

3.1 Model Selection and Implementation

Considering the understanding from the previous part – the data responded better to a non-linear model for prediction – it is decided that the classification model should follow this. The models considered to implement were a Random Forest Classifier and a Gradient Boosting model, both of which are relatively sophisticated models than can deal with non-linearities well, and do not require huge computation, which are factors in why they would be suitable for the dataset. Both use ensembles of decision trees, but differ in the process used to train and tune them – see section 8.2 in [4]. The deciding factor for which model to implement was the consideration of noise/outliers, which from the regression discussion likely had some effect on the final accuracy. According to [5], although a gradient boosting method may be able to reach some more accurate results, they can be prone to noise, and training to reduce this effect can be difficult. Therefore, a Random Forest Classifier is chosen.

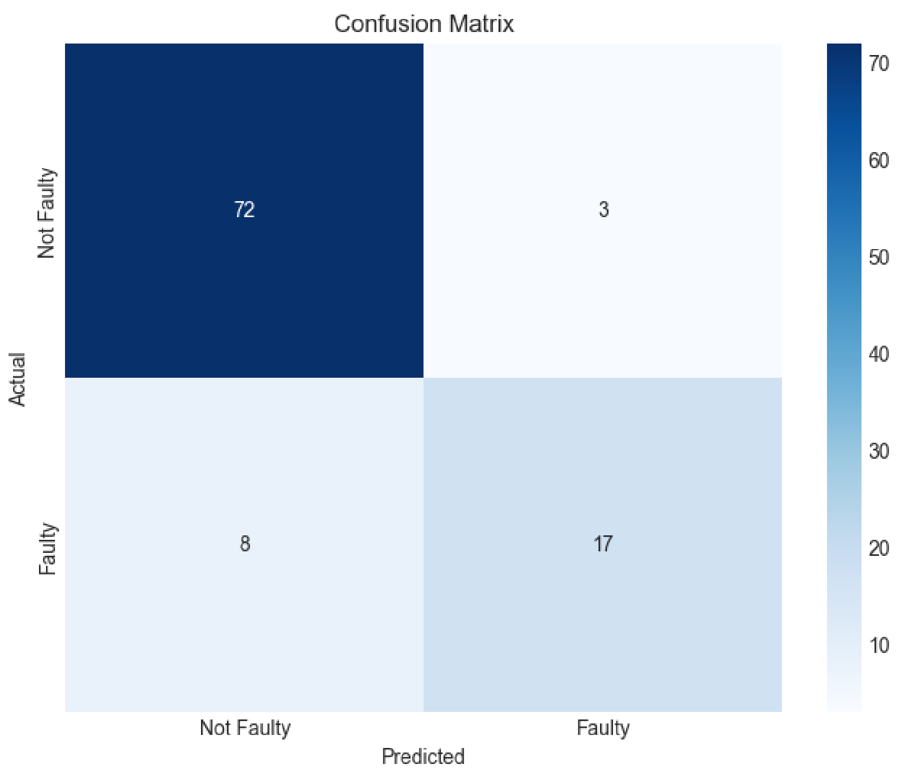
In terms of evaluation metrics, classification is different to regression, as the output is a discrete variable – in the binary case, 1 or 0 - compared to a continuous output like the TTF prediction. In binary classification, the use of Accuracy, Precision, Recall, and F1 score are best placed to provide input towards evaluating the results, and are easily interpreted. Accuracy refers to the rate at which the model produces the correct result, whereas precision and recall explain how accurate the model is at predicting a positive in relation to the total number of positives predicted, and the true number of positive respectively. The F1 Score provides a more balanced score accounting for both precision and recall.

The first steps in beginning implementation of the model are to import the data again, and normalise it in the same way as used previously. A simple line of code is used to return a set of classification labels for the test data, which returned 1 for every value of TTF <=30. This is followed by importing and running the Random Forest Classifier from Scikit-Learn. Finally, a grid search for the best parameters is completed, with scoring based upon optimising Recall. This was chosen as the target parameter due to the context of the problem, which is discussed further in the results (Appendix D).

3.2 Analysis and Discussion of Model Results

Following identification of the best parameters for the Random Forest Classifier - 'class_weight': None, 'max_depth': 7, 'min_samples_leaf': 7, 'min_samples_split': 2, 'n_estimators': 100 – and keeping the random state at 42, it was found that the best possible values for the scoring metrics were as follows: Accuracy = 0.89, Precision = 0.85, Recall = 0.68, F1 Score = 0.76. Before discussing these results, it is important to note that the iterated parameters provided very little improvement from simply limiting the number of trees in the classifier, which suggests that the model is being heavily influenced by some part of the data that doesn't see any alteration from small changes to parameters. Also, there were many sets of parameters that reached the same scores as those listed. To address this, perhaps a larger test set could be used.

The Accuracy value of 0.89 suggests the model is classifying relatively well, as around 9/10 predictions would be correct. This could be higher and at industry level would likely not be acceptable, but looking at the precision and recall is important to see where the inaccuracies lie. Precision of 0.85 means a relatively low rate of false positives, which in context, under the assumption that a positive result would lead to an engine maintenance check, would likely outperform usual time-based maintenance procedures in terms of costs, but could also be improved to reduce the need for maintenance checks that are unnecessary. The most significant metric however, recall, is very low, which is concerning. For better visualisation, a confusion matrix is produced.



Graph 10 Classifier confusion matrix

The confusion matrix shows the number of times the predictor is correct and incorrect and the distribution of these. From the test data, it is shown that 89/100 predictions are correct, 3 are false positives, and 8 are false negatives. In the aircraft industry, safety means safety of customers lives, and so is the highest importance. This brings attention to the bottom left of the confusion matrix, which relates to the recall value of 0.68. This means there are 8 occasions in the test data where the model does not believe an engine is in its last 30 cycles, when in reality it is. This would be a rate of false negatives that is highly unacceptable if a model like this were to be used in practice. One way to improve this might mean that the value of 30 cycles for the predictor be increased to a higher value, for example 40 or 50, which would likely improve the recall and therefore increase safety, but at some increased level of cost, with more overall positive values being delivered.

Similar to the regressor, implementation of time-series data in the test dataset could also be a viable option for real life application, which could allow the use of LSTM models, such as the one used in [6] in a very similar classification of engine failure problem, albeit with a larger dataset. Here, a recall of over 0.9 is achieved, which is much more promising. This method would be much more computationally expensive, but the payoff for a large manufacturer would also be much greater.

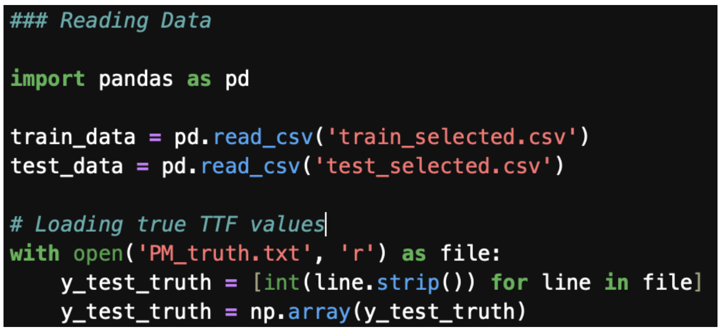
Conclusion

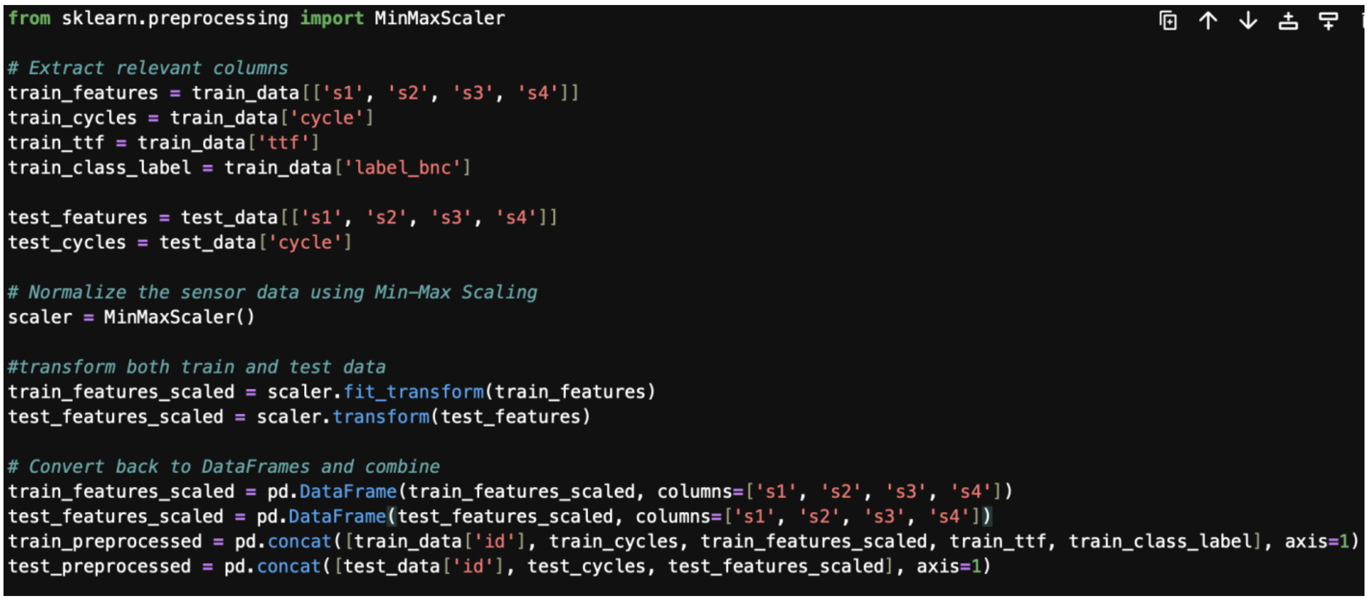
The work successfully provides insight into some simple statistical learning models for the contextual problem, and provides some promising results and discussion. The end use of predictive maintenance in the aircraft industry is to reduce costs, but the overriding focus on safety is also present, and so for a more useful implementation of models such as those discussed, it would likely be recommended to look into deep learning, as well as some other ways to manipulate the provided data to increase performance, such as noise reduction or outlier identification.

References

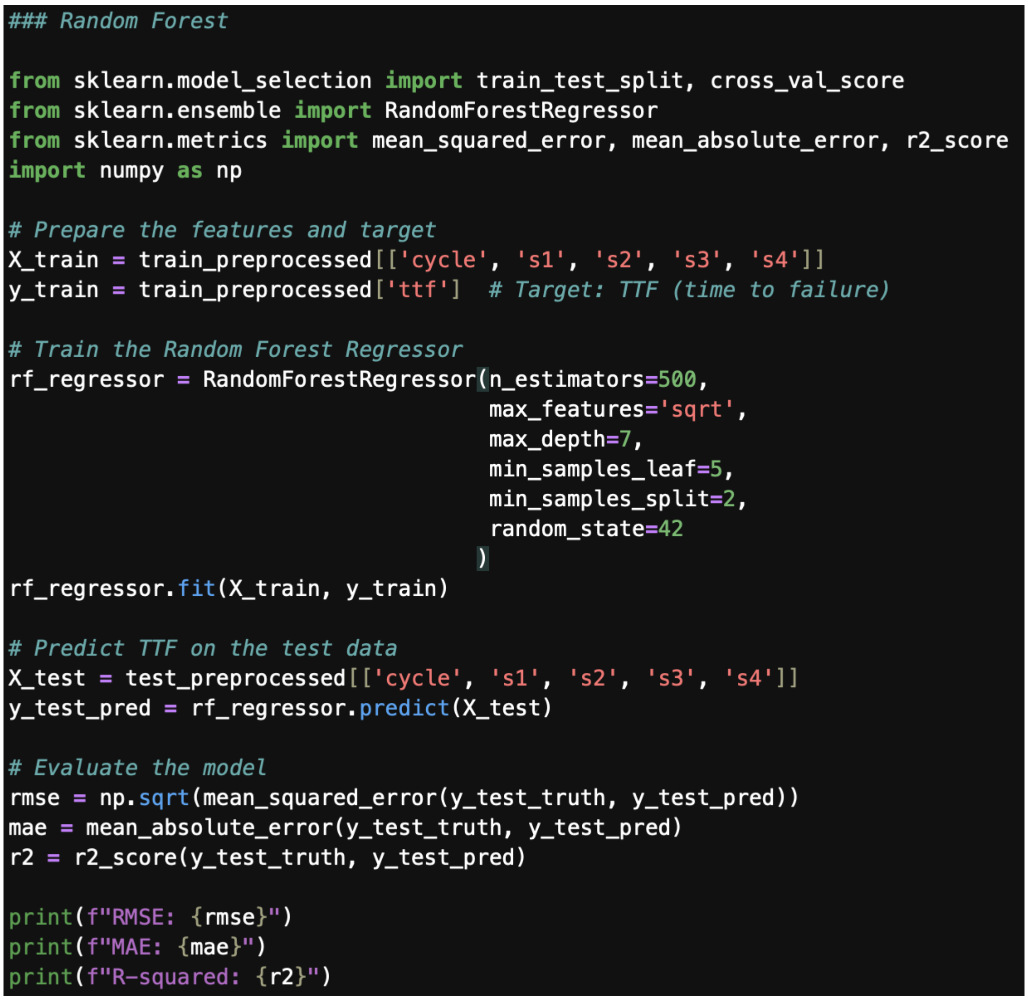
[1] Leleko S, Chupryna R. Predictive Maintenance with Machine Learning: A Complete Guide [Internet]. 2024 [cited 2024 Nov 20]. Available from: https://spd.tech/machine-learning/predictive-maintenance/
[2] Kulik J. How to Implement Predictive Maintenance Using Machine Learning [Internet]. 2024 [cited 2024 Nov 20]. Available from: https://neurosys.com/blog/predictive-maintenance-using-machine-learning#article-1
[3] Tor MA. Evaluation Metrics for Regression Problems [Internet]. 2024 [cited 2024 Nov 20]. Available from: https://medium.com/@mehmetalitor/evaulation-metrics-for-regression-problems-e07d082d8fb1
[4] James G, Witten D, Hastie T, Tibshirani R. An Introduction to Statistical Learning [Internet]. New York, NY: Springer US; 2021 [cited 2024 Nov 20]. Available from: https://doi.org/10.1007/978-1-0716-1418-1
[5] Simic MM. Gradient Boosting Trees vs. Random Forests: A tutorial [Internet]. 2024 [cited 2024 Nov 20]. Available from: https://www.baeldung.com/cs/gradient-boosting-trees-vs-random-forests
[6] Makone A. Aircraft Engine Failure Prediction: A Classification Case Study [Internet]. 2020 [cited 2024 Nov 20]. Available from: https://ashutoshmakone.medium.com/aircraft-engine-failure-prediction-a-classification-case-study-bb2a5f8ff733

Appendix A

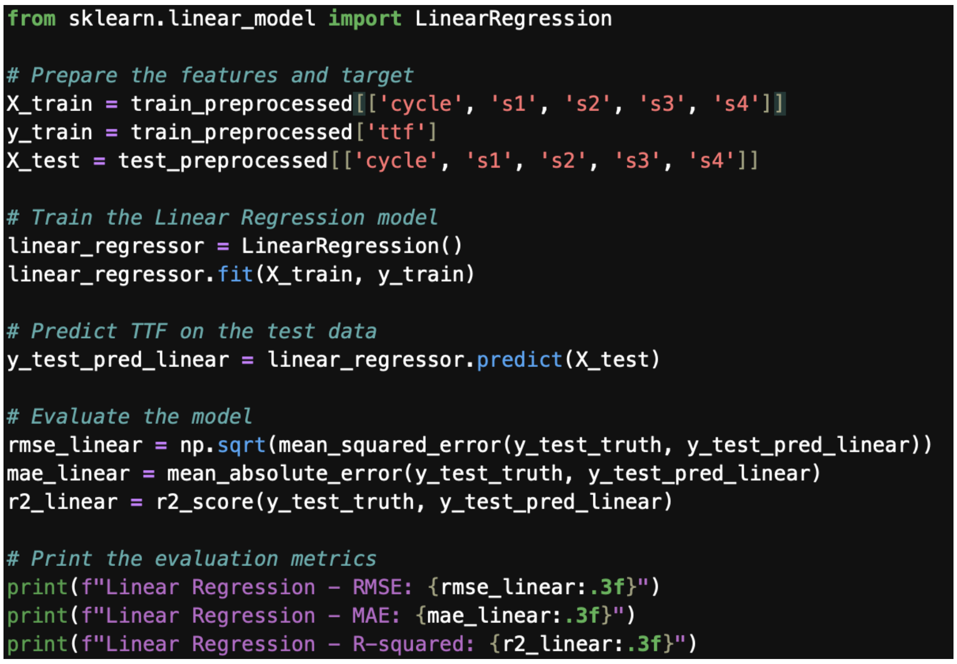

```python
### Reading Data

import pandas as pd

train_data = pd.read_csv('train_selected.csv')
test_data = pd.read_csv('test_selected.csv')

# Loading true TTF values
with open('PM_truth.txt', 'r') as file:
    y_test_truth = [int(line.strip()) for line in file]
    y_test_truth = np.array(y_test_truth)
```


```python
from sklearn.preprocessing import MinMaxScaler

# Extract relevant columns
train_features = train_data[['s1', 's2', 's3', 's4']]
train_cycles = train_data['cycle']
train_ttf = train_data['ttf']
train_class_label = train_data['label_bnc']

test_features = test_data[['s1', 's2', 's3', 's4']]
test_cycles = test_data['cycle']

# Normalize the sensor data using Min-Max Scaling
scaler = MinMaxScaler()

#transform both train and test data
train_features_scaled = scaler.fit_transform(train_features)
test_features_scaled = scaler.transform(test_features)

# Convert back to DataFrames and combine
train_features_scaled = pd.DataFrame(train_features_scaled, columns=['s1', 's2', 's3', 's4'])
test_features_scaled = pd.DataFrame(test_features_scaled, columns=['s1', 's2', 's3', 's4'])
train_preprocessed = pd.concat([train_data['id'], train_cycles, train_features_scaled, train_ttf, train_class_label], axis=1)
test_preprocessed = pd.concat([test_data['id'], test_cycles, test_features_scaled], axis=1)
```

Appendix B

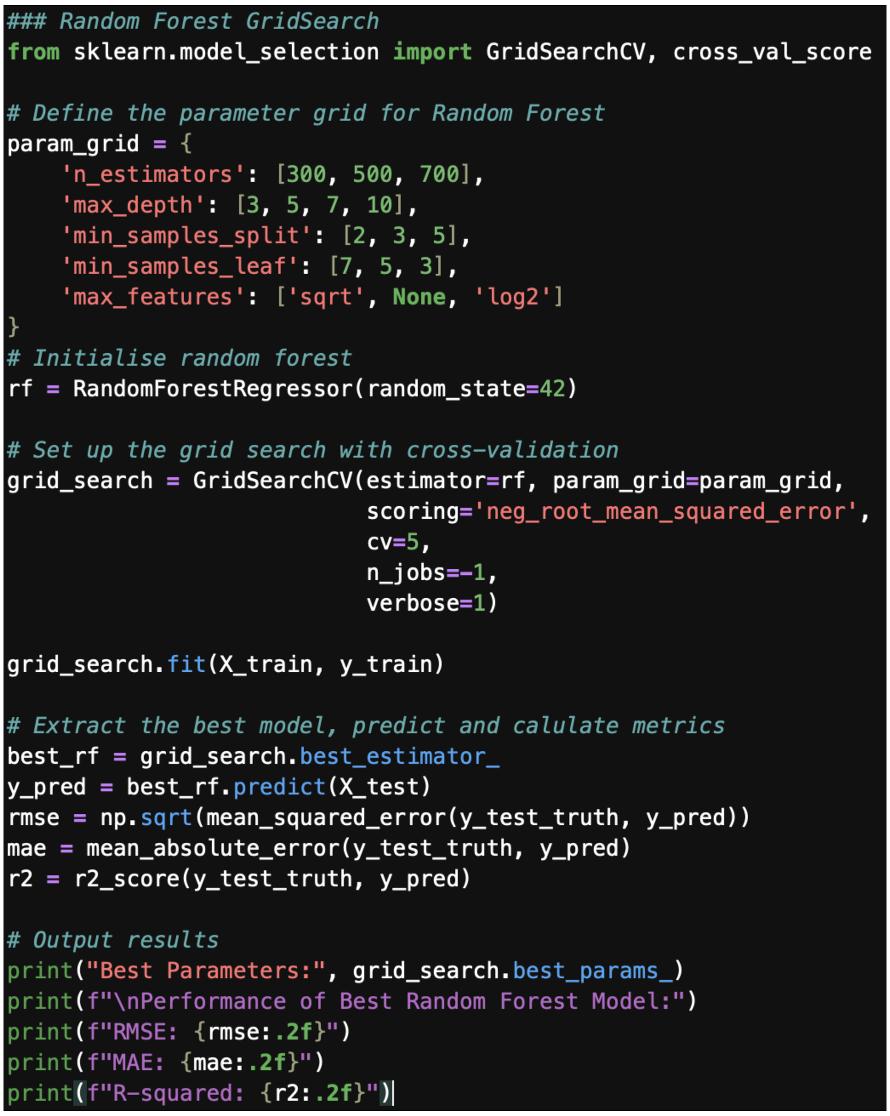


```python
### Random Forest

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Prepare the features and target
X_train = train_preprocessed[['cycle', 's1', 's2', 's3', 's4']]
y_train = train_preprocessed['ttf']  # Target: TTF (time to failure)

# Train the Random Forest Regressor
rf_regressor = RandomForestRegressor(n_estimators=500,
                                     max_features='sqrt',
                                     max_depth=7,
                                     min_samples_leaf=5,
                                     min_samples_split=2,
                                     random_state=42
                                    )
rf_regressor.fit(X_train, y_train)

# Predict TTF on the test data
X_test = test_preprocessed[['cycle', 's1', 's2', 's3', 's4']]
y_test_pred = rf_regressor.predict(X_test)

# Evaluate the model
rmse = np.sqrt(mean_squared_error(y_test_truth, y_test_pred))
mae = mean_absolute_error(y_test_truth, y_test_pred)
r2 = r2_score(y_test_truth, y_test_pred)

print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
print(f"R-squared: {r2}")
```



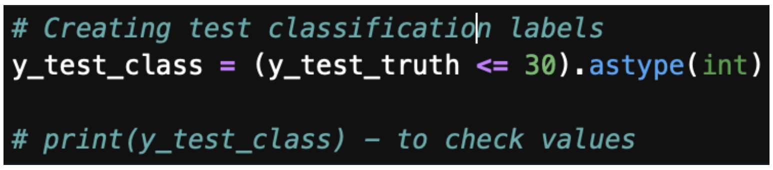
```python
from sklearn.linear_model import LinearRegression

# Prepare the features and target
X_train = train_preprocessed[['cycle', 's1', 's2', 's3', 's4']]
y_train = train_preprocessed['ttf']
X_test = test_preprocessed[['cycle', 's1', 's2', 's3', 's4']]

# Train the Linear Regression model
linear_regressor = LinearRegression()
linear_regressor.fit(X_train, y_train)

# Predict TTF on the test data
y_test_pred_linear = linear_regressor.predict(X_test)

# Evaluate the model
rmse_linear = np.sqrt(mean_squared_error(y_test_truth, y_test_pred_linear))
mae_linear = mean_absolute_error(y_test_truth, y_test_pred_linear)
r2_linear = r2_score(y_test_truth, y_test_pred_linear)

# Print the evaluation metrics
print(f"Linear Regression - RMSE: {rmse_linear:.3f}")
print(f"Linear Regression - MAE: {mae_linear:.3f}")
print(f"Linear Regression - R-squared: {r2_linear:.3f}")
```

Appendix C

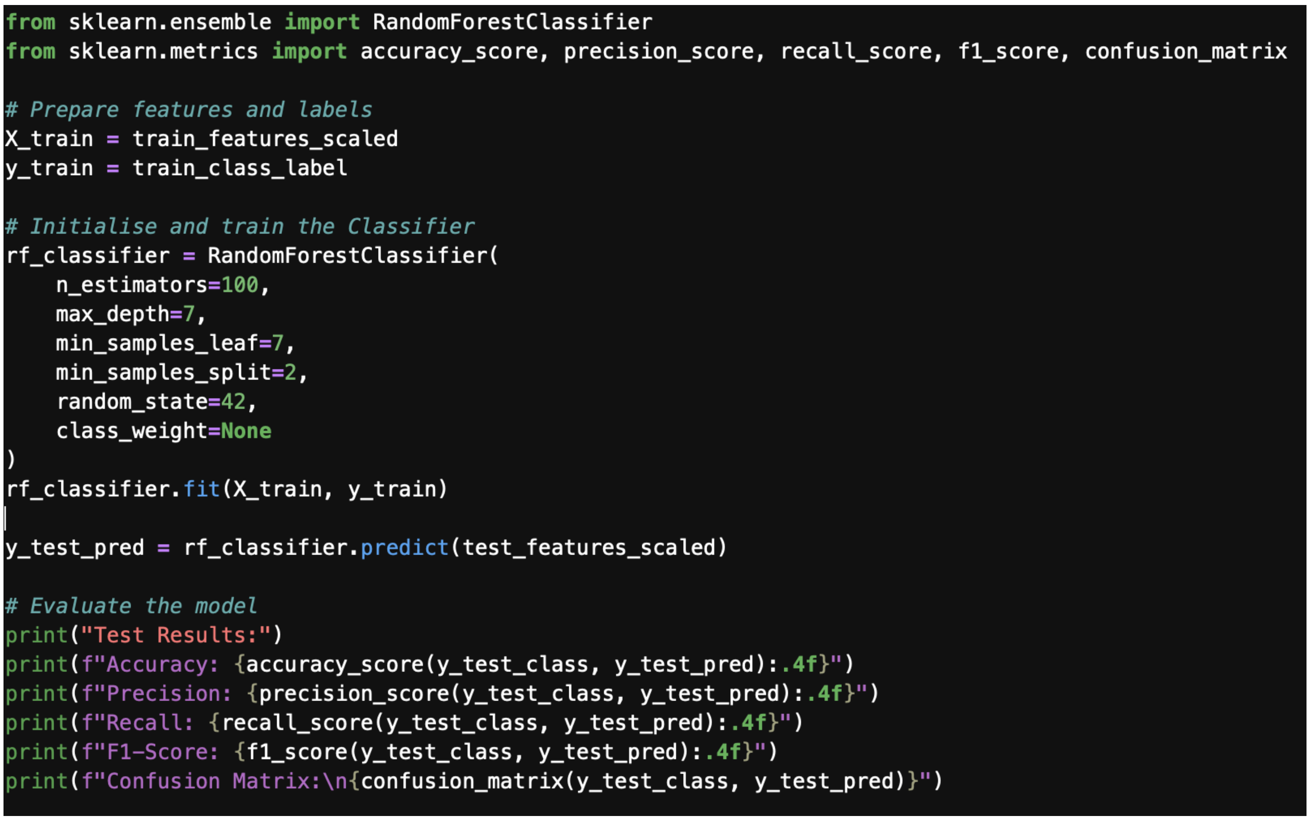


```python
### Random Forest GridSearch
from sklearn.model_selection import GridSearchCV, cross_val_score

# Define the parameter grid for Random Forest
param_grid = {
    'n_estimators': [300, 500, 700],
    'max_depth': [3, 5, 7, 10],
    'min_samples_split': [2, 3, 5],
    'min_samples_leaf': [7, 5, 3],
    'max_features': ['sqrt', None, 'log2']
}
# Initialise random forest
rf = RandomForestRegressor(random_state=42)

# Set up the grid search with cross-validation
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           scoring='neg_root_mean_squared_error',
                           cv=5,
                           n_jobs=-1,
                           verbose=1)

grid_search.fit(X_train, y_train)

# Extract the best model, predict and calulate metrics
best_rf = grid_search.best_estimator_
y_pred = best_rf.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test_truth, y_pred))
mae = mean_absolute_error(y_test_truth, y_pred)
r2 = r2_score(y_test_truth, y_pred)

# Output results
print("Best Parameters:", grid_search.best_params_)
print(f"\nPerformance of Best Random Forest Model:")
print(f"RMSE: {rmse:.2f}")
print(f"MAE: {mae:.2f}")
print(f"R-squared: {r2:.2f}")
```

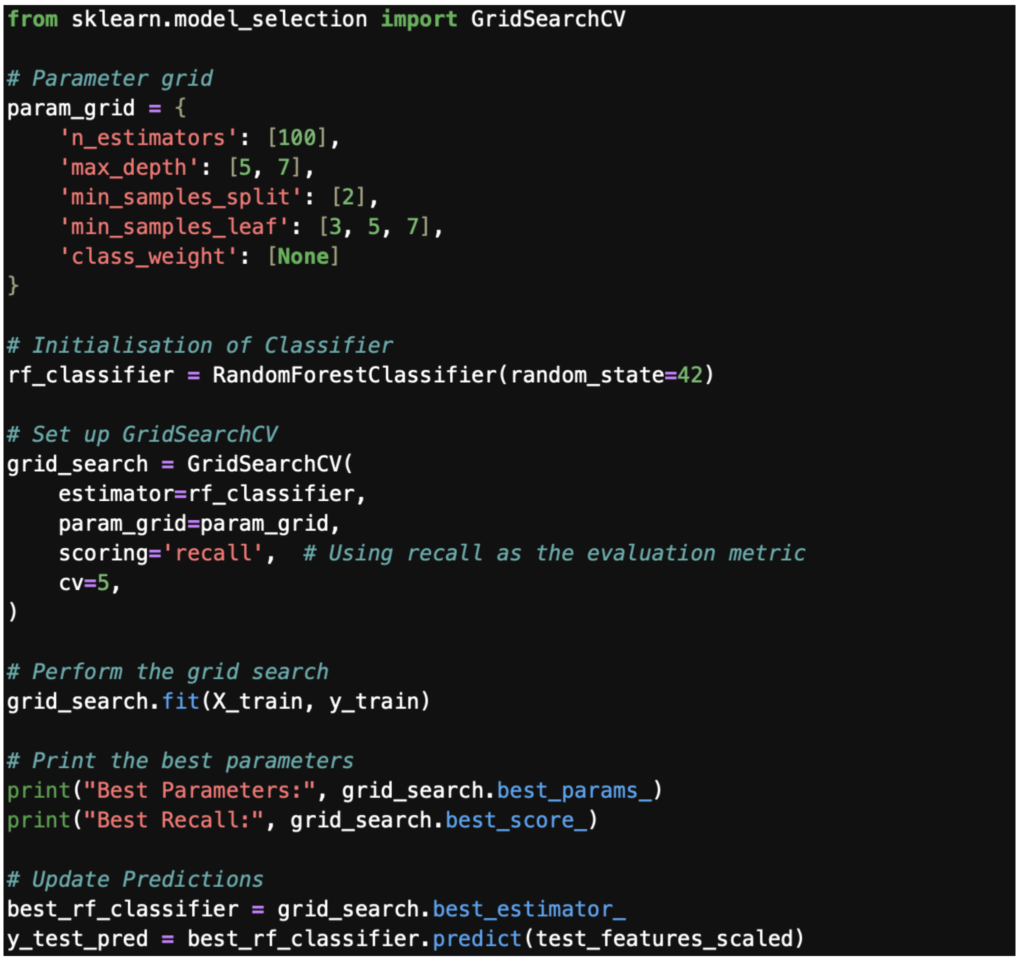
Appendix D



```python
# Creating test classification labels
y_test_class = (y_test_truth <= 30).astype(int)

# print(y_test_class) - to check values
```



```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

# Prepare features and labels
X_train = train_features_scaled
y_train = train_class_label

# Initialise and train the Classifier
rf_classifier = RandomForestClassifier(
    n_estimators=100,
    max_depth=7,
    min_samples_leaf=7,
    min_samples_split=2,
    random_state=42,
    class_weight=None
)
rf_classifier.fit(X_train, y_train)

y_test_pred = rf_classifier.predict(test_features_scaled)

# Evaluate the model
print("Test Results:")
print(f"Accuracy: {accuracy_score(y_test_class, y_test_pred):.4f}")
print(f"Precision: {precision_score(y_test_class, y_test_pred):.4f}")
print(f"Recall: {recall_score(y_test_class, y_test_pred):.4f}")
print(f"F1-Score: {f1_score(y_test_class, y_test_pred):.4f}")
print(f"Confusion Matrix:\n{confusion_matrix(y_test_class, y_test_pred)}")
```



```python
from sklearn.model_selection import GridSearchCV

# Parameter grid
param_grid = {
    'n_estimators': [100],
    'max_depth': [5, 7],
    'min_samples_split': [2],
    'min_samples_leaf': [3, 5, 7],
    'class_weight': [None]
}

# Initialisation of Classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Set up GridSearchCV
grid_search = GridSearchCV(
    estimator=rf_classifier,
    param_grid=param_grid,
    scoring='recall',  # Using recall as the evaluation metric
    cv=5,
)

# Perform the grid search
grid_search.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters:", grid_search.best_params_)
print("Best Recall:", grid_search.best_score_)

# Update Predictions
best_rf_classifier = grid_search.best_estimator_
y_test_pred = best_rf_classifier.predict(test_features_scaled)
```