Predictive Maintenance of Turbofan Engines

Building a Remaining Useful Life Prediction System Based on the NASA FD001 Dataset

Tibor Nagy

Contents

1. Introduction **2**

2. Exploratory Data Analysis **2**

2.1 Experimental Scenario 2

2.2 Dataset Dimensions 3

2.3 Preview of the Dataset 3

2.4 Distribution of Time Cycles 3

3. Preprocessing **4**

3.1 Adding a RUL column 4

3.2 Sensor Reading Curves 4

3.2 Removing Features 5

3.3 Decomposing the sensor reading curves 5

3.4 Performance Evaluation 6

4. Models **7**

4.1 Kaplan-Meier Model 7

4.2 Baseline Linear Regression Model 8

4.3 kNN model 10

4.4 randomForest model 11

4.5 SVR model 12

5. Conclusion **13**

1. Introduction

In this paper I will try to build a prediction system which predicts the Remaining Useful Life (RUL) for the turbofan engines included in the NASA's FD001 dataset. This is one of the four datasets NASA released in a data challenge competition. It includes Run-to-Failure simulated data from turbo fan jet engines. It consists of one operational condition (sea level) and one failure mode (HPC Degradation). The goal is to predict the RUL of each engine in the test subset, based on 3 operational settings and the readings of 21 sensors.
Why am I interested in this dataset after more than a decade it was released? I stumbled upon this when I was searching for a dataset for my capstone project in a data science training course. I chose an easier dataset then, but I realized that analyzing this can be good introduction to the predictive maintenance of any machines. The dataset does not include the name of the sensors and operational settings, thus we cannot use any domain knowledge. Our results are based on applying the correct techniques. We can easily use these techniques for the predictive maintenance of other machines. I did not find any R based comprehensive analysis of this dataset on the net to learn from. I am still new in data science, so I decided to make my own analysis to learn, and to gain some experience in applying some of the most popular and computationally light methods.

2. Exploratory Data Analysis

At first, we need to be familiarized with the 3 subsets of the FD001 dataset (*train_FD001, test_FD001, RUL_FD001*) we are working with.

2.1 Experimental Scenario

The datasets consist of multiple multivariate time series. Each dataset is further divided into training and test subsets. Each time series is from a different engine – i.e., the data can be considered to be from a fleet of engines of the same type. Each engine starts with different degrees of initial wear and manufacturing variation which is unknown to the user. This wear and variation is considered normal, i.e., it is not considered a fault condition. There are three operational settings that have a substantial effect on engine performance. These settings are also included in the data. The data is contaminated with sensor noise.
The engine is operating normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure. In the test set, the time series ends some time prior to system failure. The objective is to predict the number of remaining operational cycles before failure in the test set, i.e., the number of operational cycles after the last cycle that the engine will continue to operate. Also provided a vector of true Remaining Useful Life (RUL) values for the test data. Each row is a snapshot of data taken during a single operational cycle, each column is a different variable. The columns correspond to:

1, unit number
2, time, in cycles
3, operational setting 1
4, operational setting 2
5, operational setting 3
6, sensor measurement 1
7, sensor measurement 2
...
26, sensor measurement 21

2.2 Dataset Dimensions

I will use the *train_FD001* as the training set and save the *test_FD001* and *RUL_FD001* for evaluating the overall accuracy of the final algorithm. Both the train and test sets consist of 100 trajectories.

Table 1: Dataset Dimensions

Dataset	No. of Rows	No. of Columns
train_FD001	20631	26
test_FD001	13096	26

2.3 Preview of the Dataset

Table 2: Preview of the dataset

unit_no	time_cycles	op_1	op_2	op_3	s_01	s_02	s_03	s_04	s_05	s_06
1	1	-0.0007	-4e-04	100	518.67	641.82	1589.70	1400.60	14.62	21.61
1	2	0.0019	-3e-04	100	518.67	642.15	1591.82	1403.14	14.62	21.61
1	3	-0.0043	3e-04	100	518.67	642.35	1587.99	1404.20	14.62	21.61
1	4	0.0007	0e+00	100	518.67	642.35	1582.79	1401.87	14.62	21.61
1	5	-0.0019	-2e-04	100	518.67	642.37	1582.85	1406.22	14.62	21.61
1	6	-0.0043	-1e-04	100	518.67	642.10	1584.47	1398.37	14.62	21.61

Table 3: Preview of the dataset

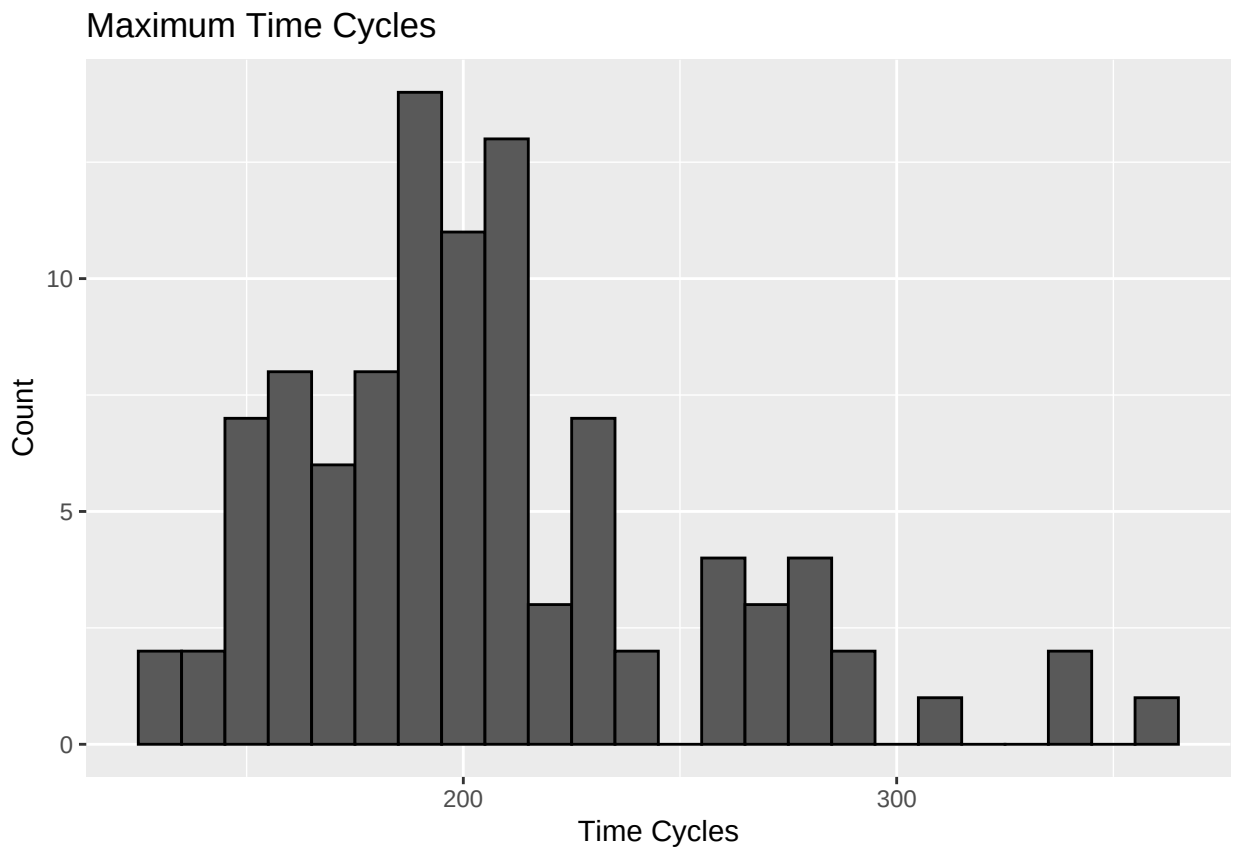
s_07	s_08	s_09	s_10	s_11	s_12	s_13	s_14	s_15	s_16	s_17	s_18
554.36	2388.06	9046.19	1.3	47.47	521.66	2388.02	8138.62	8.4195	0.03	392	2388
553.75	2388.04	9044.07	1.3	47.49	522.28	2388.07	8131.49	8.4318	0.03	392	2388
554.26	2388.08	9052.94	1.3	47.27	522.42	2388.03	8133.23	8.4178	0.03	390	2388
554.45	2388.11	9049.48	1.3	47.13	522.86	2388.08	8133.83	8.3682	0.03	392	2388
554.00	2388.06	9055.15	1.3	47.28	522.19	2388.04	8133.80	8.4294	0.03	393	2388
554.67	2388.02	9049.68	1.3	47.16	521.68	2388.03	8132.85	8.4108	0.03	391	2388

Table 4: Preview of the dataset

s_19	s_20	s_21
100	39.06	23.4190
100	39.00	23.4236
100	38.95	23.3442
100	38.88	23.3739
100	38.90	23.4044
100	38.98	23.3669

2.4 Distribution of Time Cycles

In the following plot we can see the distribution of the maximum operational *time_cycles* of the engines.



This is clearly not a normal distribution. The maximum of the curve is around 200 cycles. Most engines fail around the maximum, but some of them functioning over 300 cycles.

3. Preprocessing

In machine learning, we must examine the predictors before running the machine algorithms. It is often needed to transform the predictors for some reason. We also remove predictors that are clearly not useful.

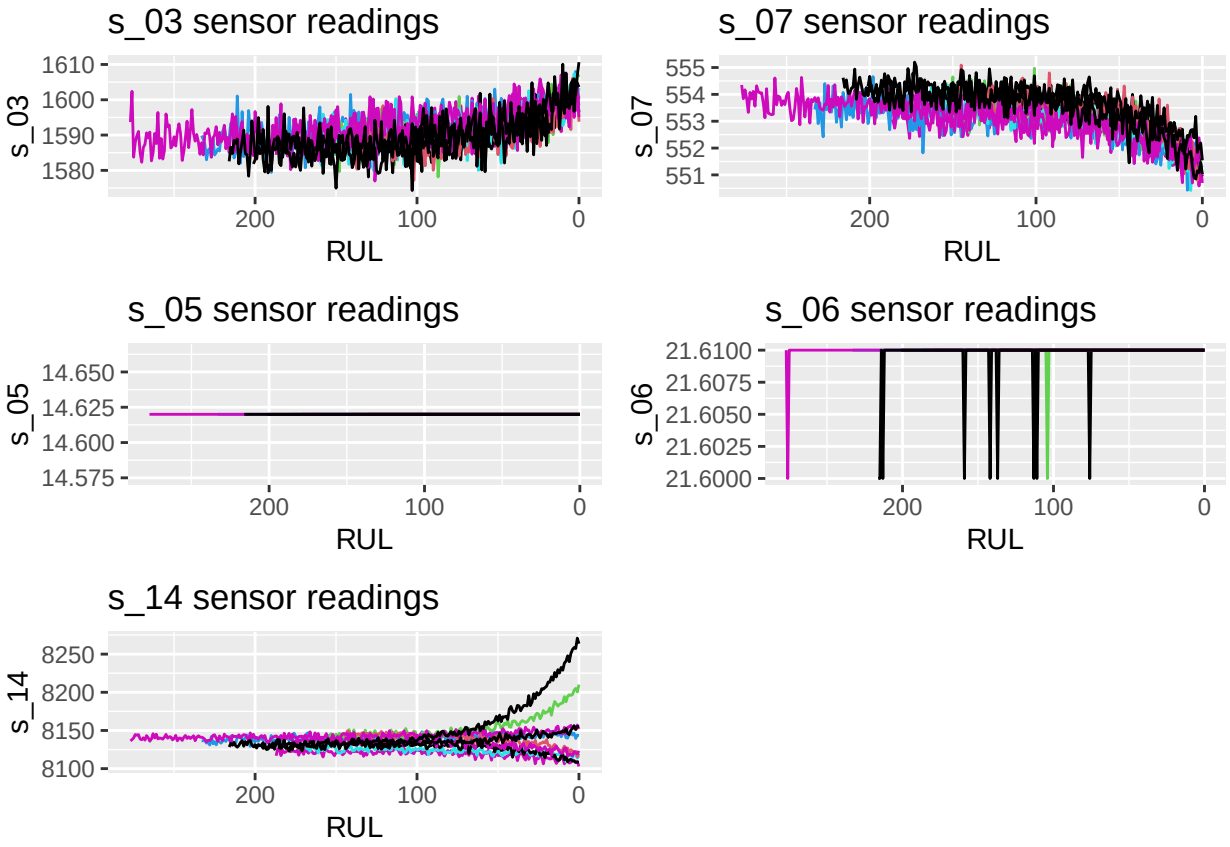
3.1 Adding a RUL column

For the predictions we need the RUL feature, but unfortunately the train set does not contain it. I computed it with the following equation, after grouping the data by *unit_no*.

RUL = max(time_cycle) - time_cycle

3.2 Sensor Reading Curves

Due to the large number of the sensors and engines, I will not print the sensor readings for each sensor and engine. Instead, I randomly selected 10 engines. I noticed the sensors can be divided into 5 groups. A sample plot for each group is shown below.



- Group 1: Sensors 2, 3, 4, 8, 11, 13, 15 and 17 show an inclining trend.
- Group 2: Sensors 7, 12, 20 and 21 show a declining trend.
- Group 3: Sensors 1, 5, 10, 16, 18 and 19 show a constant trend.
- Group 4: Sensor 6 is unique. It shows a constant trend with some downward spikes.
- Group 5: Sensor 9 has a similar pattern as sensor 14, they show a different trend for each engine.

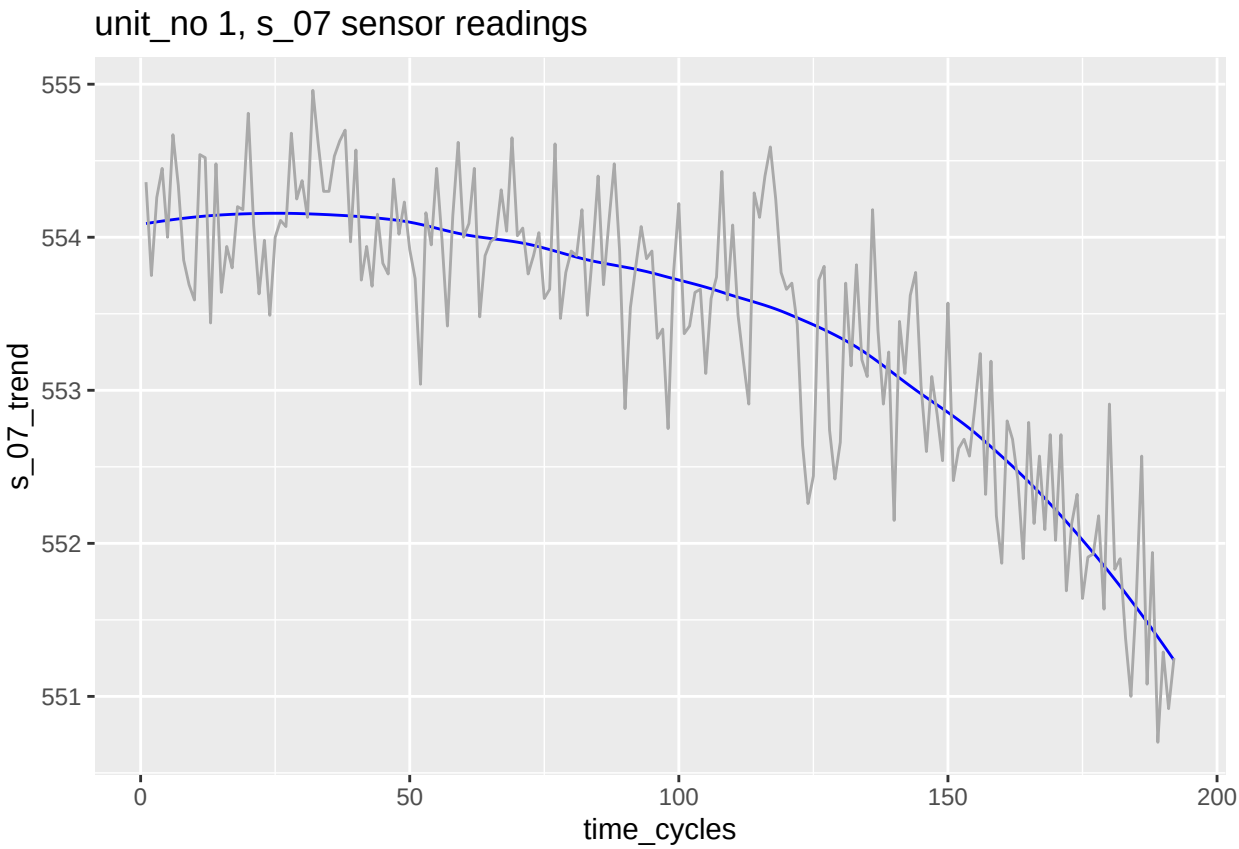
3.2 Removing Features

Features that have no clear relation to the RUL do not contain useful information, so they can be removed. These are the readings of sensors 1, 5, 6, 9, 10, 14, 16, 18 and 19. I also dropped the operational settings, because I do not need them in the further analysis.

3.3 Decomposing the sensor reading curves

We can see from the sensor reading plots, that the curves of the readings are quite wiggly. We can improve our model's performance if we smooth the curves, remove the noise and keep the smoothed trends only. To estimate the trends I used the *loess()* function.

We can see an example of the results on the following chart.



3.4 Performance Evaluation

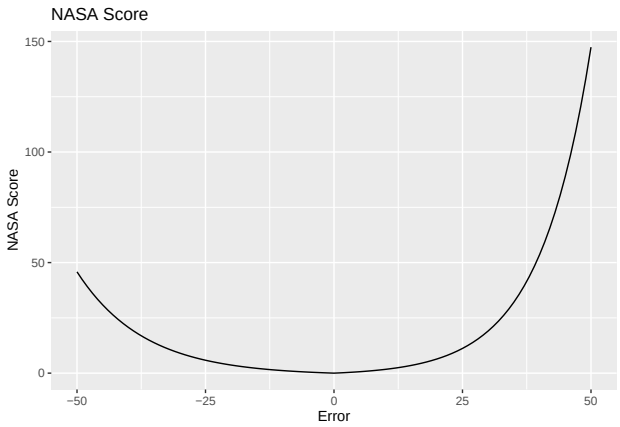
NASA did not use RMSE for performance evaluation, they used the following custom algorithm that penalize late predictions more heavily than early predictions instead.

$$s = \sum_{i=1}^{n} e^{\frac{-d}{10}} - 1 \ for \ d < 0$$

$$s = \sum_{i=1}^{n} e^{\frac{d}{13}} - 1 \ for \ d \geq 0$$

where s is the computed score, n is the number of UUTs,

$$d = \hat{t}_{RUL} \breve{} t_{RUL} \ (EstimatedRUL \breve{} TrueRUL).$$



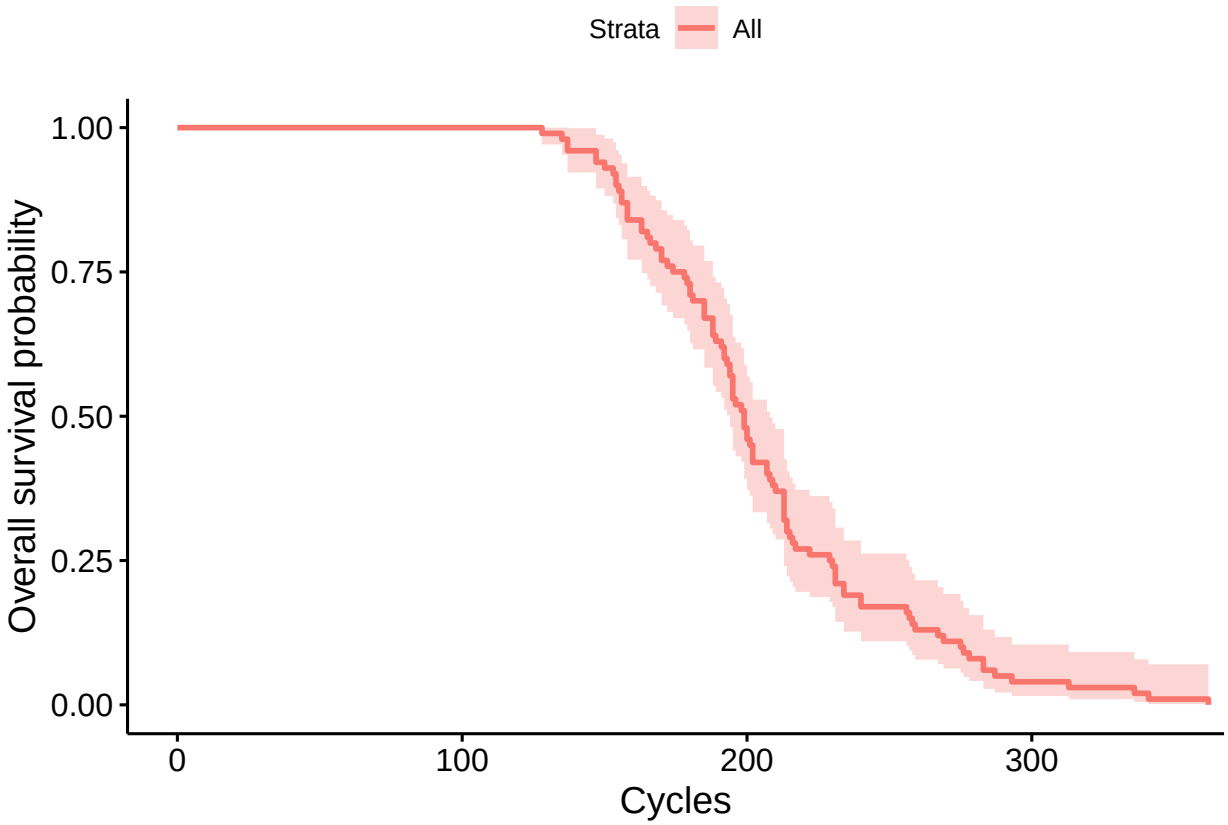
This makes the model building a bit more challenging, because in default the *train()* function optimizes for RMSE. Fortunately, we can pass our custom optimality criterion (*NASA_Score*) to the *train()* function by using the *summaryFunction* argument in the *trainControl()* function.

```
NASA_Score <- function(data, lev = NULL, model = NULL){
  d <- data[, "pred"]- data[, "obs"]
  out <- ifelse(d >= 0, exp(d/10)-1, exp(-d/13)-1)
  out_sum <- sum(out)
  names(out_sum) <- "NASA_Score"
  return(out_sum)}
```

4. Models

4.1 Kaplan-Meier Model

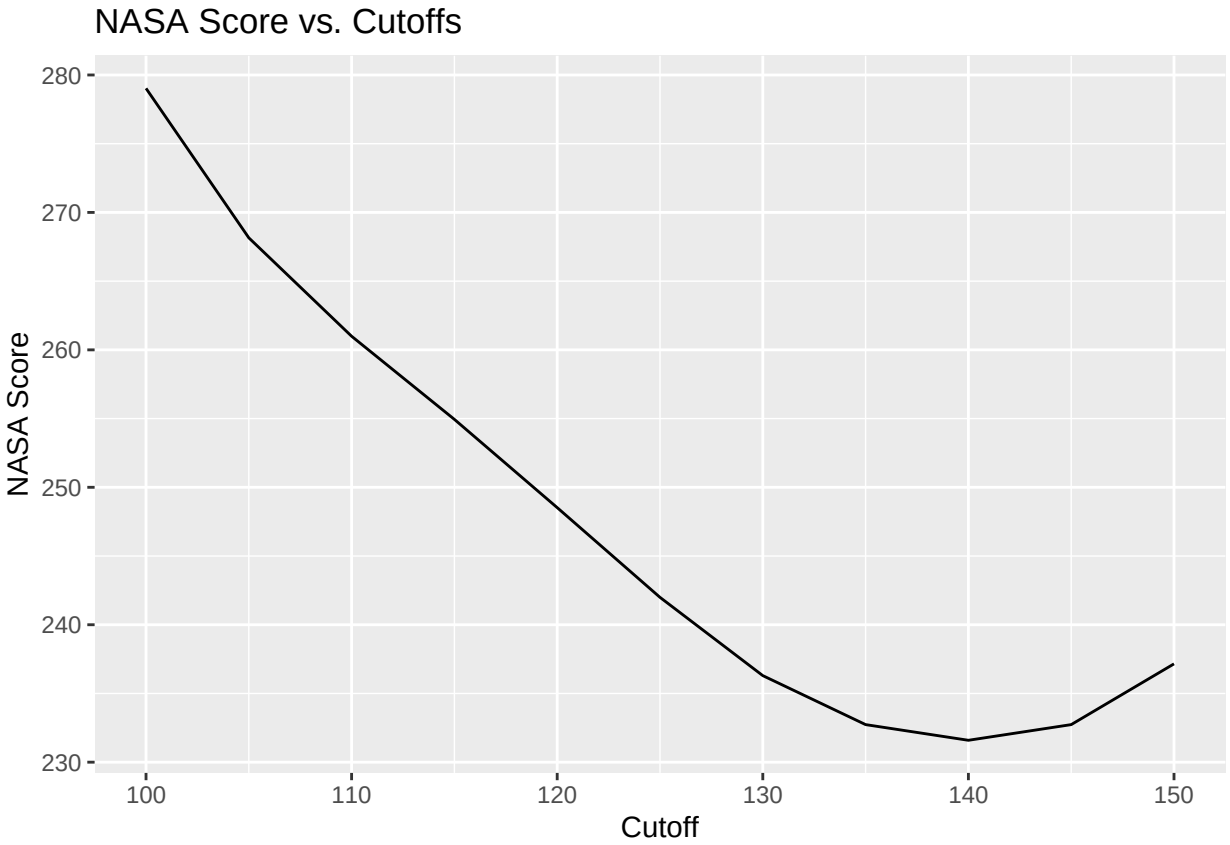
The Kaplan-Meier is a very popular and widely used method in survival analysis. It shows us how does a survival function that describes engine survival over time looks like. It requires the serial time of the engine (time_cycles) and the engine status (broken down or functioning) at the end of the serial time. It only gives the probability that an engine will survive past a particular time. We cannot extrapolate the results beyond that time. It means we cannot make predictions for the engines in the test set, we can apply this method to the train set only.
Our train set does not contain the engine status, so I added a new *status* column. The status is *0* if the engine is functioning, *1* if it is broken down. For the Kaplan-Meier survival curve we need the rows where the status is *1*. From the curve above we can see that the survival probability below 125 cycles is 100%.



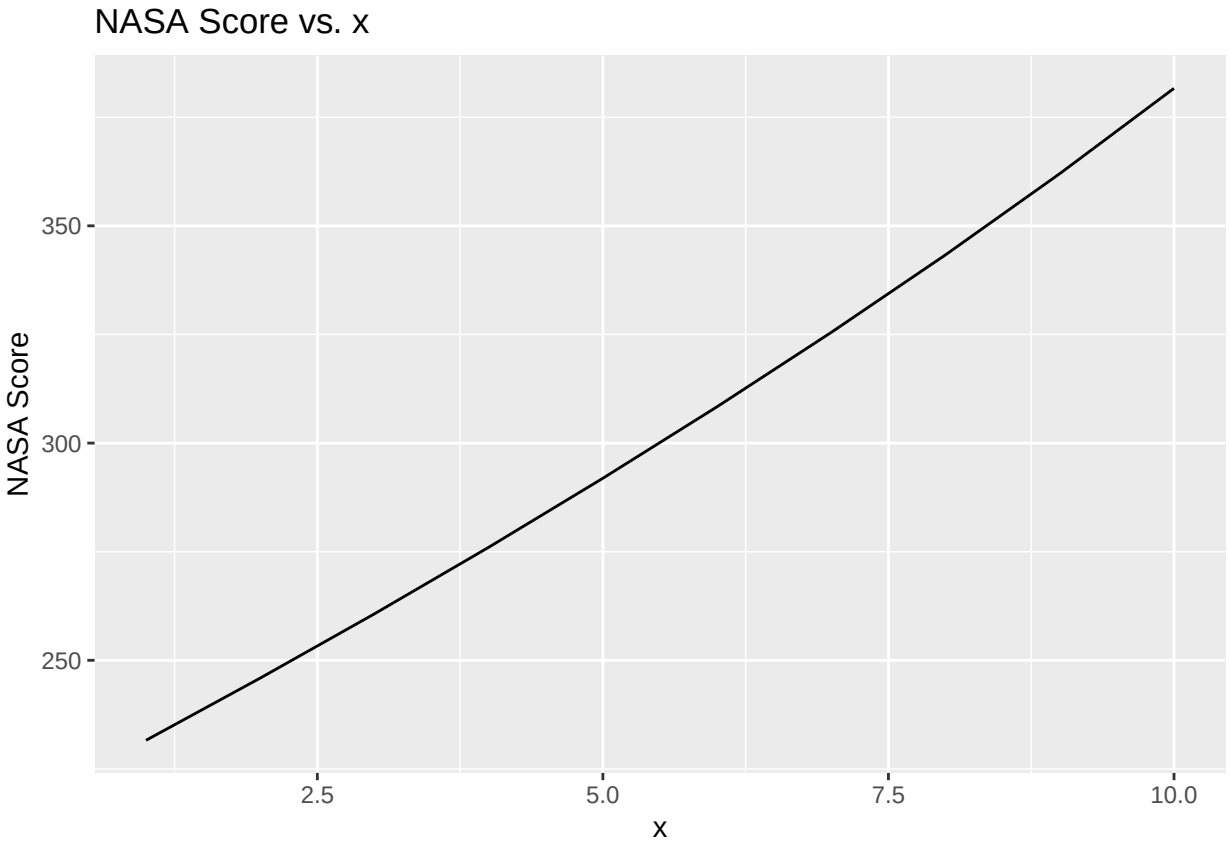
4.2 Baseline Linear Regression Model

From the sensor reading plots we can see that the trends of the remaining sensors below some hundred cycles are almost horizontal. This means we cannot give accurate predictions about the RUL using these readings, we can remove them from the dataset. I will try some cutoffs to see which cutoff yields the best improvement on our model. To avoid overtraining I divided the *train_FD001* set to a train set and a test set temporarily that contains the 80% and the 20% of the units respectively. I removed the first 1/3 of data for each unit from the test set, and selected one random row for each unit to make the predictions and choose the best cutoff.

Then I used the *test_FD001* set for validation. The results are shown on the following graph. The best cutoff is at 140 time cycles.



We could predict the RUL based on only the last observations of each unit, but in this case we drop the great majority of information from the test set, so probably there are other ways to achieve more accuracy. We could make predictions on the last x cycles and average them. What value of x would give the best accuracy? To find out I divided the *train_FD001* set to a train set and a test set temporarily again. For each unit in the test set, I selected a random row, and I kept the last x rows that precede the random rows only. I tried x*'s* from a range from 1 to 10.



As we can see from the plot above, the relationship between the *NASA_Score* and the number of rows we use for prediction is almost linear. We obtain the best accuracy if we predict from the last row only. This is because the higher the RUL the more inaccurate the prediction gets. If we increase the number of rows we involve more and more inaccurate predictions into our model.

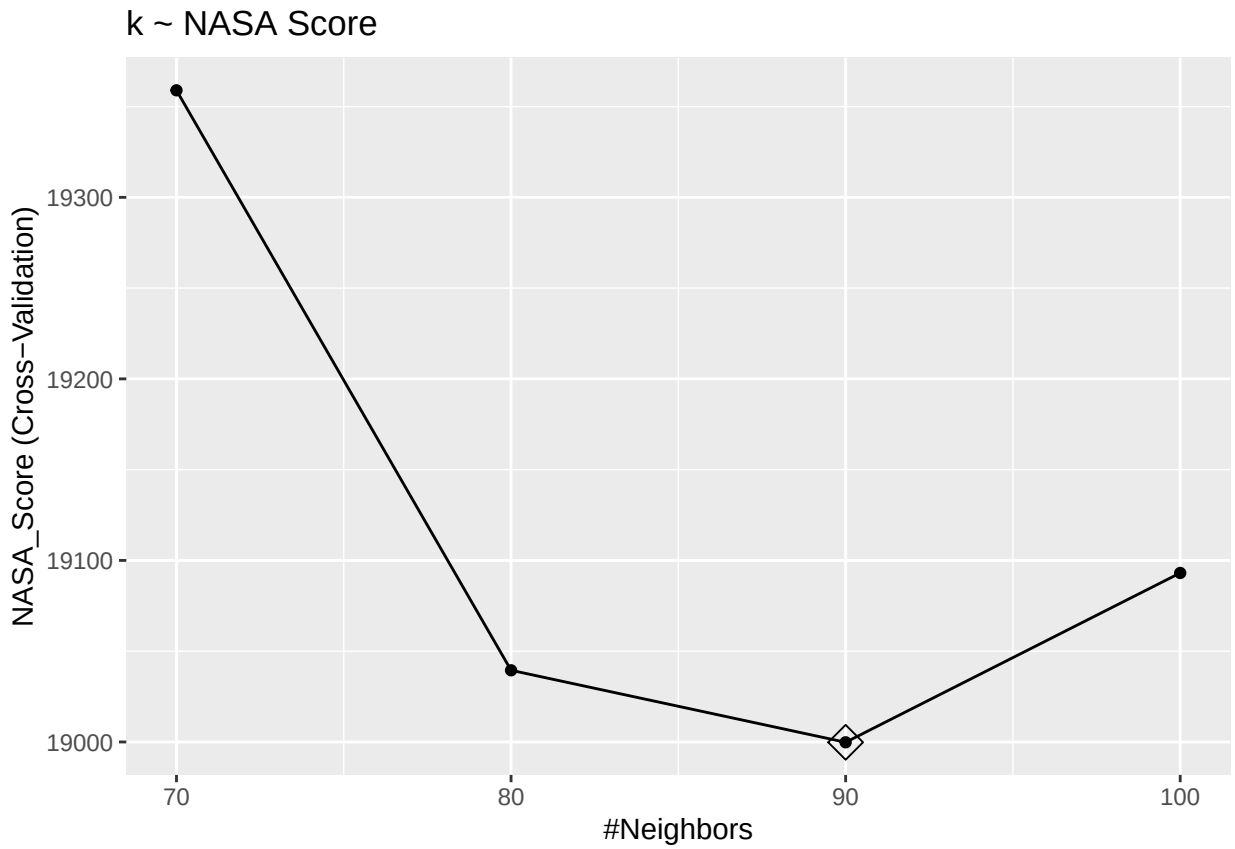
So for the final *lm* model I used the entire *train_FD001* dataset with cutoff of 140, and I made the predictions based on the last row of each unit.

Table 5: Performance Metrics

Method	Score	RMSE	Error
lm	2476	25.7058	[-63,53]

4.3 kNN model

K-nearest neighbors (kNN) is a pattern recognition algorithm that uses training datasets to find the k closest relatives in future examples. When kNN is used in classification, we calculate to place data within the category of its nearest neighbor. I used the *train* function from the *caret* package to train the algorithm. To increase accuracy, the *train* function performs 5-fold cross validation. For the kNN algorithm the tuning parameter is *k*. The default of the *train* function is to try k= 5, 7, 9. To find the best accuracy I tried it on a much different scale: k= 70, 80, 90, 100.



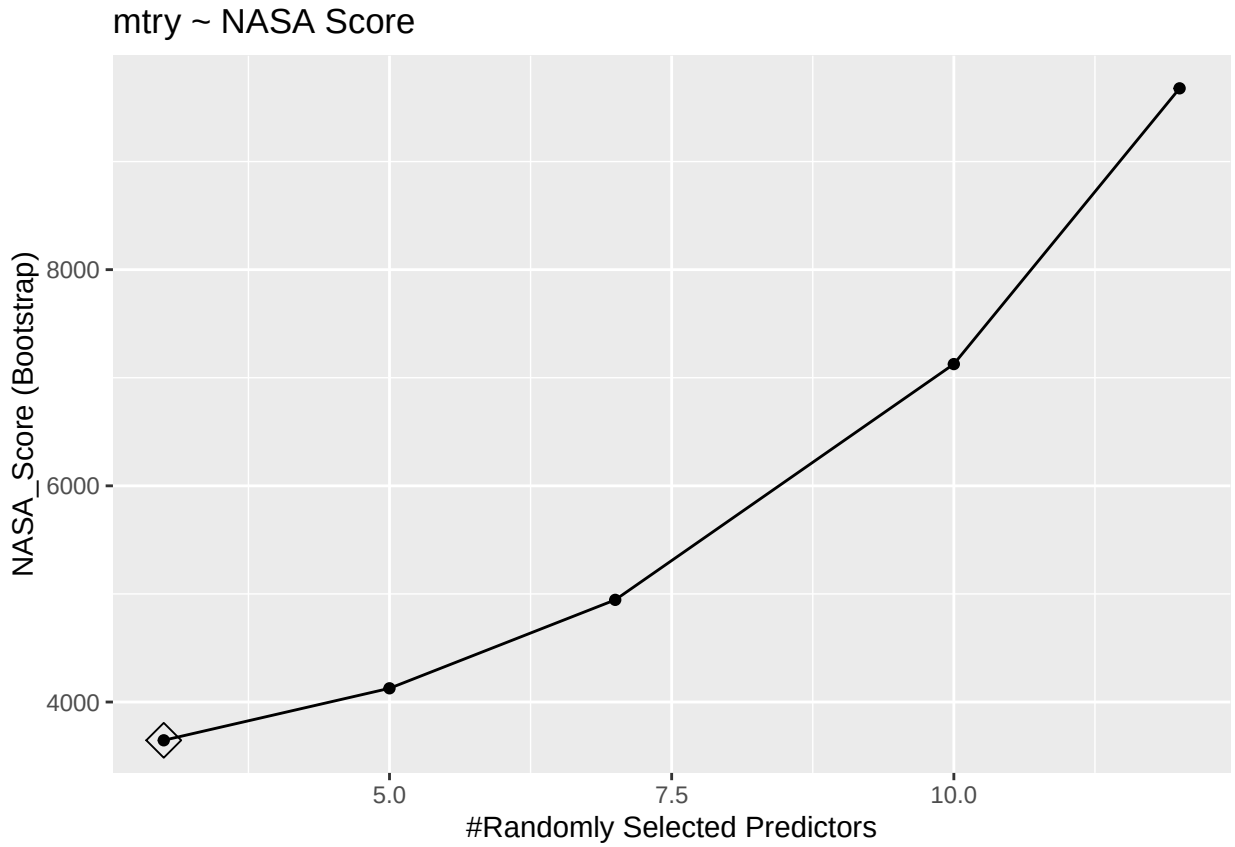
We can see that the k value that gives the best accuracy is 90. It is much larger than the default values of the *train* function. The NASA Score on the test set is 1129, which is less than the half of the *lm* method's score. We can see ~10% improvement in the RMSE too.

Table 6: Performance Metrics

Method	Score	RMSE	Error
lm	2476	25.7058	[-63,53]
kNN	1129	23.0176	[-51,63]

4.4 randomForest model

The random forest algorithm is an expansion of decision tree. A decision tree is a learning algorithm that is perfect for classification problems, as it is able to order classes on a precise level. It works like a flow chart, separating data points into two similar categories at a time from the "tree trunk" to "branches", to "leaves", where the categories become more finitely similar. This creates categories within categories, allowing for organic classification with limited human supervision.
The randomForest improves prediction performance of the decision tree by averaging multiple decision trees (a forest of trees constructed with randomness).
To increase accuracy, I tried the tuning parameter *mtry* values 3, 5, 7, 10, 12.



As we can see in the following plot the optimal *mtry* is 3. By further reducing the *mtry* value we could end up with overfitting.
The result on the test set:

Table 7: Performance Metrics

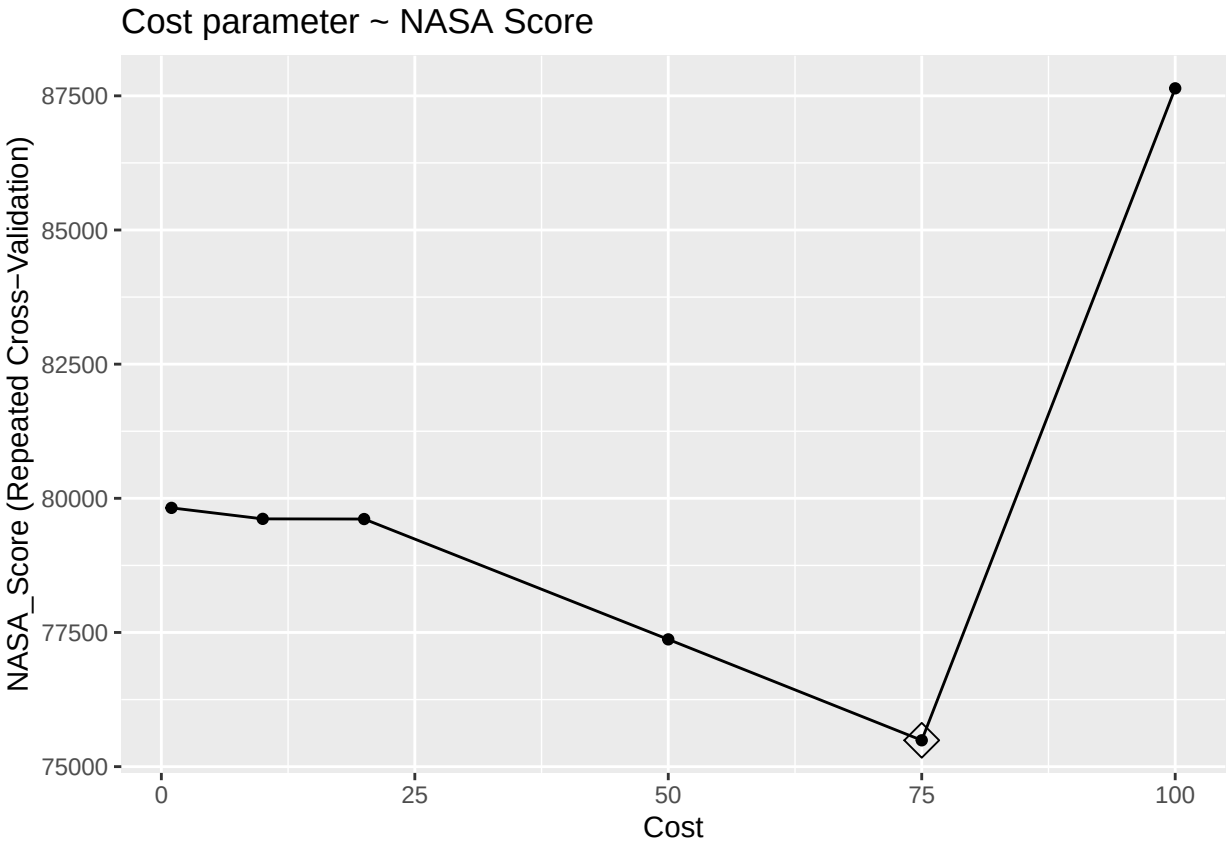
Method	Score	RMSE	Error
lm	2476	25.7058	[-63,53]
kNN	1129	23.0176	[-51,63]
RF	1304	25.1841	[-43,73]

The Score is about the half of the linear model's, however the RMSE is just slightly better.

4.5 SVR model

The Support Vector Regression is an very popular, computationally light method. We can use it by calling the SVM (Support Vector Machine) function. It can be used for both regression and classification, because the function automatically detects if the data is categorical or continuous, and uses the applicable method to compute the results. We can choose from numerous kernels, which gives flexibility to the method. In this project I used the linear kernel, and performed 5-fold cross validation to increase accuracy. Using the linear kernel we can tune the method by trying the *Cost parameter* on a scale. I tried the values of 1, 10, 20, 50, 75 and 100. I normalized the variables to make their scale comparable by setting the *preProcess* option to *center* and *scale*. This is a very useful feature in terms of a distance-based model.

The plot below shows us the best *Cost parameter* for this method is 75.



This is a simple method, there is no surprise in it's performance. In terms of scores it performs much better than the baseline linear method, on the same level as the two remaining models.

Table 8: Performance Metrics

Method	Score	RMSE	Error
lm	2476	25.7058	[-63,53]
kNN	1129	23.0176	[-51,63]
RF	1304	25.1841	[-43,73]
SVR	1245	25.8639	[-43,62]

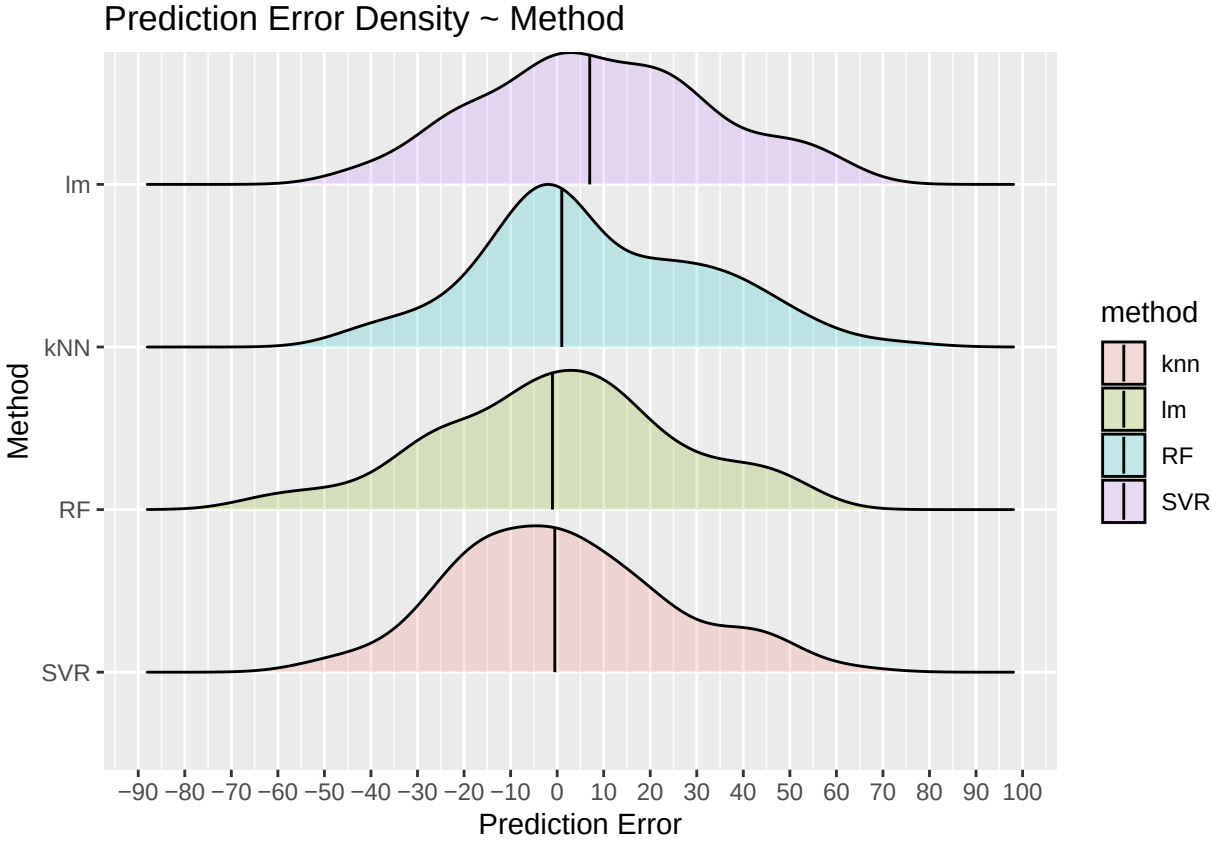
5. Conclusion

In this paper I analyzed the NASA's FD001 dataset, and built some prediction models which predicts the Remaining Useful Life (RUL) for the turbofan engines included in. I started with the Kaplan-Meier method, which actually cannot make predictions for the RUL, but it can improve our understanding about the data. Then, I built 4 prediction models. I chose some of the most popular and computationally light models, the *lm*, *kNN*, *randomForest* and the *SVR*. I used the scoring system provided by NASA as the optimality criterion to train and evaluate the models. As a result, the simplest model, the linear regression (*lm*) turned out to be the most inaccurate. The other three models' performance were pretty much the same, but the *kNN* performed a bit better than the others. Only the *randomForest*'s performance was a bit disappointing because this is the computationally heaviest model, and it couldn't outperform the other 3 models. It provided only an average score.

The performance summary table and the density plot of the prediction errors by method are shown below. The medians are also marked on the density plots.

Table 9: Performance Metrics

Method	Score	RMSE	Error
lm	2476	25.7058	[-63,53]
kNN	1129	23.0176	[-51,63]
RF	1304	25.1841	[-43,73]
SVR	1245	25.8639	[-43,62]



If we would like to obtain more accuracy, we must use more complex models like neural networks, hybrid or boosted models. But with the complexity of the model, the hardware requirements get higher, and we can quickly end up with a model that cannot be trained on an average laptop.