Maintenance cost reduction through predictive techniques

Contents

Maintenance cost reduction through predictive techniques

Introduction

Predictive maintenance, as opposed to reactive or preventive maintenance, can be useful in minimizing device downtime and might as well maximize its lifetime. Because maintenance is performed only when needed, it also reduces labor and material cost. It can be useful for fleet and inventory management and operational planning. This is analogous to preventive medicine as opposed to precision medicine.

This document is organized as follows:

1. Answers to the questions in the case study prompt.
2. The solution process documentation.
3. The solution deployment.

Answers to the case study prompts

a. Are there things that didn't work that need to be addressed?
   Answer: No.

b. What would need to be done next to bring this to production? What are the possible dependencies?
   Answer:

   Data

   - Testing should be done on real data before productionizing the model to avoid surprise problems in production, that could occur due to possible differences in real data from the training data.
   - If the model will receive telemetry time-series data in production, then it will need to be trained on time-series data.
   - Is the model going to do *Batch or Real-time* Prediction? This will define concerns about data storage/location/privacy and ethics.
   - The way the real data will be retrieved in production should be known and handled accordingly.
   - The size of the production data should be defined.
   - The whole process of data loading and processing should be automated to avoid errors in production.

Logging
Should this application be pushed to production, a logging system is critical, essentially one that tells us the requests that were made, the data made with them, the status of the response and more information. This can help easily identify any issues or bugs in the future.

Deployment

Another route that could be followed would be to use docker-compose, this is a more organized way to structure the dockerfile, and does not require following some of the steps taken in this demo. In this case we would create a yml configuration file, which would be used to run the docker file as well as other functionalities.

c. Why did you make the design choices you made?
Answer:

1. In feature selection I dropped metric7 because it is redundant and does not give any added information to our predictive model. I made this decision based on the features' frequency distribution and density curves as well as the feature correlation, both plotted in **figures 3 and 4.**

2. For balancing the data, I chose to do the oversampling using the SMOTE* method rather than doing random oversampling of the failure class, or using a weighted Random Forest classifier. I made this decision based on the resulting model performance in each case which was evaluated using the Area Under the receiver operating characteristic Curve (AUC) (**fig. 7**).

3. Optimum number of samples to oversample (optimum # of datapoints) from the failure class in order to balance the dataset. I made this decision based on the best resulting model performance (AUC) at each possible sample size (**fig.6**).

4. Model Selection, I chose Random Forest classifier because it is a high accuracy supervised learning algorithm that is well-suited to this type of classification problems, and compared to another 6 algorithms, it consistently outperformed all of them. This is demonstrated in **figure 7** below. Although XGBoost's performance was close to that of Random Forest, yet Random Forest's performed better most of the times, and had the added benefit of an interpretable model.

5. Building an ensemble of Random Forests; I made this decision in order to remedy any chance of the model learning the failure class and overfitting due to over-sampling the failure class. The decision was backed by the improved AUC of the ensemble over a single Random Forest.

6. Ensemble Validation using *RepeatedStratifiedKFold (k=10, repeats=10)* , which is a robust cross validation method in which the data is repeatedly randomly stratified into 10

partitions, each time holding out one partition for validation of the model (i.e. 9 for training and 1 for validation). This is repeated 10 times for a total of 100 validations. I made this decision in order to rule out any doubt of model overfitting (**fig**. **8**).

7. Model deployment; I packaged, dockerized the solution and deployed it to AWS.

   The reasons I have set up and deployed the model this way are:
   - Contained environment.
   - Easy to modify and debug, localized.
   - All requirements and dependencies are on the server.
   - No data privacy concerns mentioned in this case, hence data is also handled on the cloud.
   - Accessibility from any device.
   - Processing power, scalability and fault tolerance of the cloud.
   - Computationally intensive tasks done on the server.
   - The setup is also completely scalable and robust to any changes in data size or structure. This is due to the modularity of the solution and the pipeline that loads the input data, and trains and saves models custom made for that data. This means, if the number of data features are changed, or if the type or number of classes are changed, or if the data gets more or less balanced, the tool will adapt and will always build the optimal solution.

d. What business outcomes can you see springing from your solution?

   The solution can be useful in any predictive maintenance context, whether maintenance of a fleet of devices in general, medical devices in Telemedicine, human health and precision medicine, etc.

   In all cases it helps to reduce maintenance cost by avoiding unneeded maintenance, reduce cost of device downtime, which could be fatal in cases of telemedicine and precision medicine, increase device lifetime and supports better inventory management and operational planning.

   The modularity and efficiency of the solution design, the solution delivery as a package that includes all possible dependencies, the fault tolerance and scalability of the cloud (i.e. AWS) makes the solution capable of handling data and businesses of different sizes and nature.

e. What data limitations did you encounter and how might those be reduced in the future?

   1. The data was highly imbalanced, with very few samples from the failure class as opposed to the non-failure class. The model performance suffered from this problem. If more data can be added from the failure class in the future, that will help developing a better model.

2. The data imbalance if not handled carefully, could result in misleading model evaluation and the wrong choice of ML algorithm.
3. No variability of the failure class in the training data. This would not match real data. When training data is very different from the actual real data, the model performance will be degraded in production.
4. If historical data and device degradation data is captured, many functionalities could be added that would resemble a real life scenario, like estimating the time before failure, and doing failure root cause analysis.
5. Information about any data privacy concerns is missing; which will warrant on-premise data handling. I assumed no data privacy concerns in this case study.
6. Whether the model will deal with telemetry time-series data in production.
7. How large is real data.
8. How will the model retrieve data for prediction in production.

Solution Process

The sequence of steps that I have taken in designing, developing and deploying the predictive maintenance model is delineated in the Process Flow diagram (**fig. 1**).

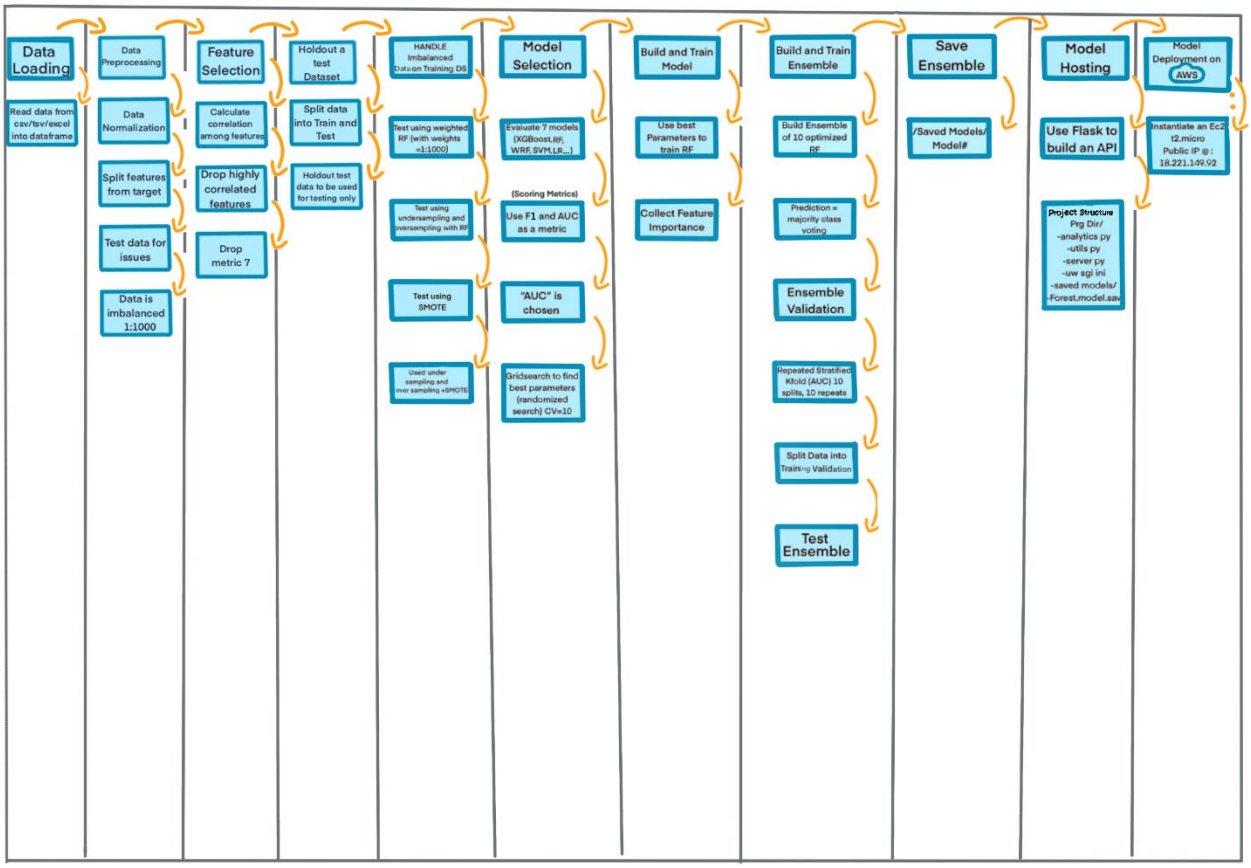

Figure 1. Process Flow. The sequence of steps taken in designing, developing and deploying the predictive maintenance model.

1. Data Loading

Data is read from the input file and the code checks if the input is in csv/tsv/excel format. The descriptive statistics of the data is shown in **fig. 2** below.

# Classes	# Features	# Samples	# Samples in class '0'	# Samples in class '1'
2	9	124,494	124,388	106

Figure 2. Descriptive Statistics of the predictive maintenance dataset.

2. Data Preprocessing

Before building the model for prediction, the data needs to be cleaned and prepared. The following are my first eye observations on the data:

- The input data file looked clean; no missing values to be imputed and no type mismatches.
- The input data had few features (9), which could be good or bad. It is good in the sense that we will not have to struggle with the curse of dimensionality, but could be bad if the features do not have enough variability to train a good model.
- The data is severely imbalanced between the two classes (~1:1000 failure vs. non-failure)

So the first thing done was to ***normalize*** the data. This means to scale it down between 0 and 1, for this step I use *Sklearn MinMaxScaler*. Then separate all features (metrics) from the target (failure).

Second, I ***visualize the variability*** in the data to see if there are any collinearity or redundancy issues. For this I plotted the distribution plots for each of the features within the two classes. The histogram and density curves for each of the metrics within the Failure and Non-Failure classes are shown in **fig.3**. I also calculated and displayed the correlations among the features shown in **fig.4**.

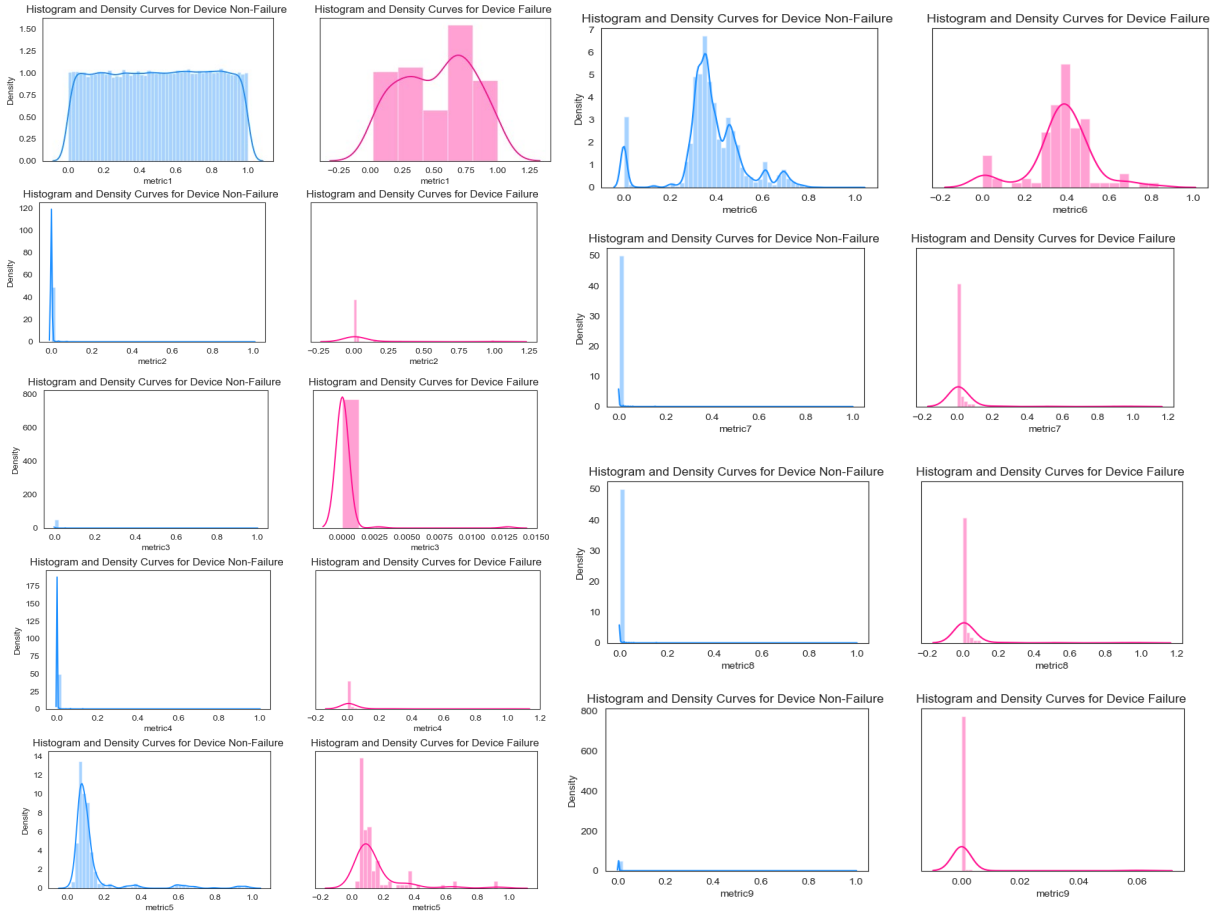

Figure 3. Features' Distribution. The histogram and density curves for each of the metrics within the Non-Failure (blue) and Failure (pink) classes, show the variability of the data of each metric across the two classes. Also shows that metric 7 and metric 8 are identical.

Fig.3, shows some variability between the features within the two classes and shows that metric7 and metric8 have the same exact distributions. This is further confirmed in **fig.4**, which shows that metric7 and metric8 have perfect correlation, meaning they provide essentially the same information when it comes to prediction, therefore I removed metric 7 from the feature set. The plot also shows that metric3 and metric9 have 0.5 correlation. The remaining features are all uncorrelated with each other, which renders them, all, possibly useful for prediction.

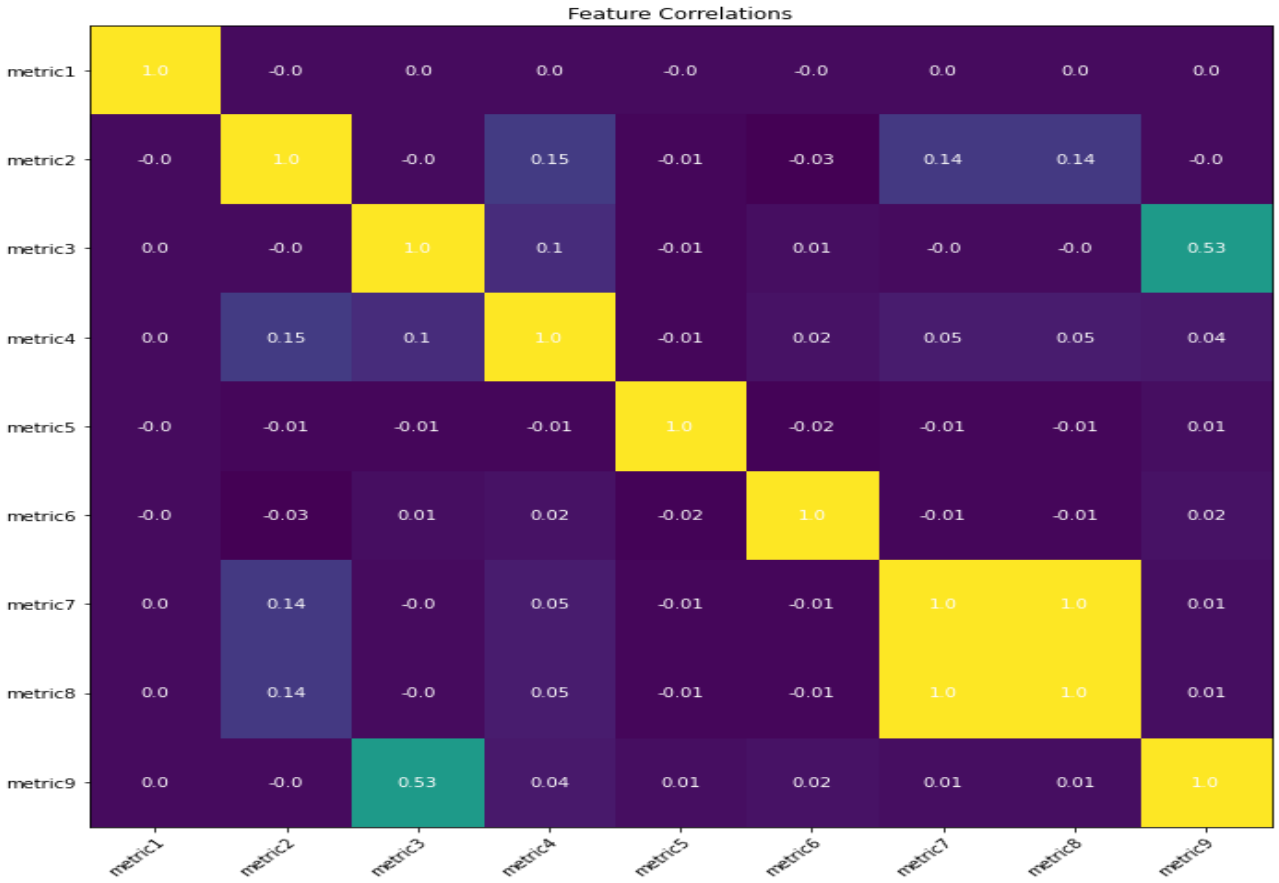

Figure 4. Feature Correlation. The heatmap shows the correlation between the 9 features, with the yellow color representing highest correlation and purple represents no correlation. The heatmap shows that metric7 and metric8 are redundant.

To investigate whether there are different failure subtypes within the failure class, I performed a Primary Component Analysis (***PCA***) of the failure class data to achieve two primary data components. I then clustered the data of the two components using *k-means* clustering algorithm. I first evaluated different cluster sizes using the elbow method, looking at both inertia and silhouette scores. Based on this I plotted the results with 2 and 4 clusters. The plots in **fig. 5** show most of the data points are located within the same cluster, the other clusters are rather small in size, which might mean they are just outliers. I believe there is not enough variation in the failure class data, or the data is too small to really discern clear clusters. From this I conclude that the data currently does not seem to support that there is more than one failure type, however, more data might give an answer to such question. This could be useful if we will need to do root cause analysis and identify the type of failure in the future.

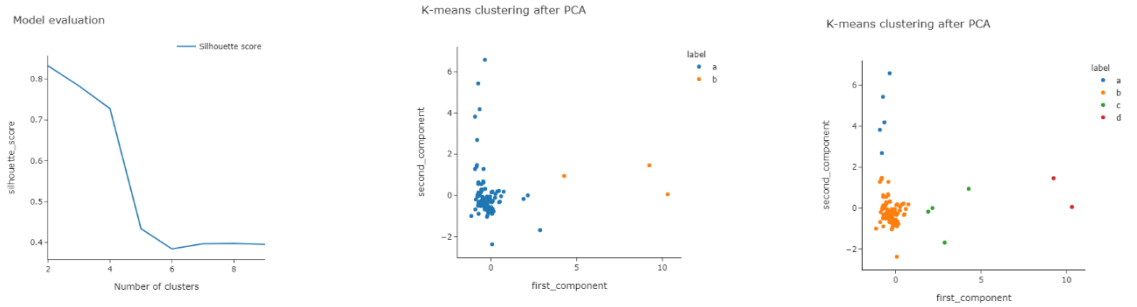

Figure 5. Failure Subtypes. The figure shows the clustering of the failure class to 2 (middle) and 4(right) clusters and the clustering algorithm evaluation; the silhouette score (left).

Next I split the data using *sklearn's train_test_split* which provides a *holdout test dataset* and a *training dataset* for both targets and features. ***From this point further until the testing of the model, I will be only working on the training dataset.***

3. Dealing with imbalanced data

Like mentioned earlier in this document, the dataset is severely imbalanced data (1:1000) between the two classes; with the minority class being the positive or failure class.

<u>To handle this problem, I tried several strategies:</u>

1. Using a weighted Random Forest Classifier with weights of 1:1000 for the Non-Failure:Failure classes.
2. Random oversampling of the Failure class and under-sampling of the Non-Failure class to balance the data. This is done by randomly duplicating failure data points and removing non-failure data points.
3. Oversampling the Failure class using the SMOTE method and randomly under-sampling the Non-Failure class.

To do #2 and #3, I first figured out the optimum number of samples (data points) from each class (**fig. 6**). After the dataset becomes balanced, I move on to selecting which model to work with.

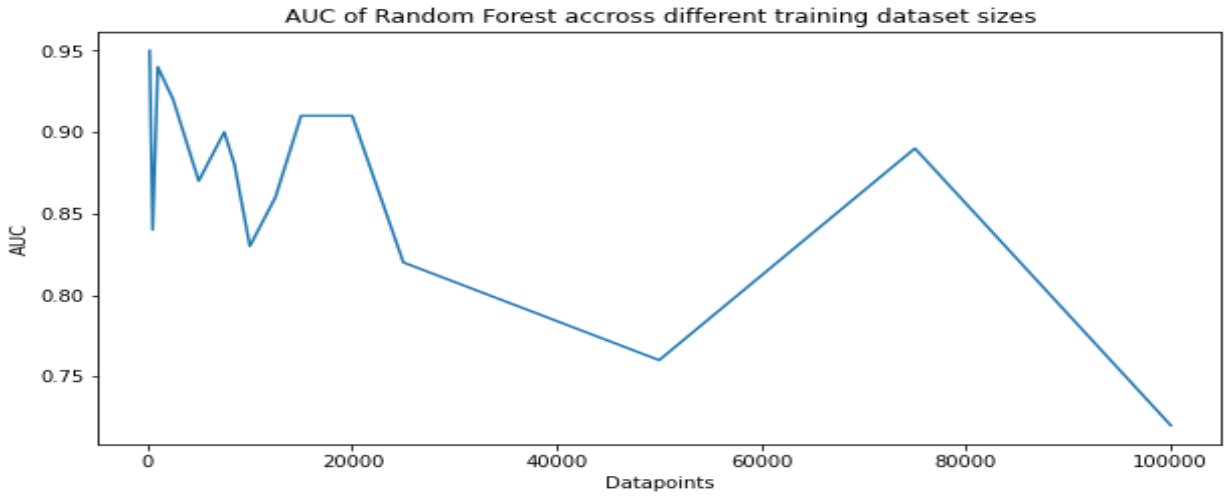

Figure 6. Optimum number of samples. The Random Forest Algorithm is run using different sizes of data (for Oversampling/Under-sampling) to figure the sample size that yields optimum model performance represented by the optimum ROC AUC.

4. Model Selection

I considered the following models for the classification task:

1. Logistic Regression
2. Random Forest
3. Decision Tree
4. XGBoost
5. Weighted Decision Tree
6. Weighted Random Forest
7. Linear SVM

In order to score the models, I look at different metrics in order to compare their performance:

Accuracy
This metric tells us how many of the correct classes our model predicts. The problem with this metric is that it is not representative of how good our model is doing in this situation, due to the imbalanced nature. If our model only predicts 0's, it will arrive at a very high accuracy, which would be misleading since our majority class is 0's. In this case, we look towards relevance, a metric which precision and recall will show us.

Precision
Precision is a metric with a formula of True Positive / (True Positive + False Positive). This is a score of when our model does predict a class if positive, how accurate that prediction is. Meaning if we have numerous positive predictions, but a low precision score, it is predicting many negatives as positives, whereas if the precision is high and number of positive predictions is low, it accurately predicts what labels are positive.

Recall

Recall is a metric with a formula of True Positive / (True Positive + False Negative). This means that a model with a high recall value is very accurate when it predicts a positive (failure), and does not often incorrectly predict a positive to be a negative.

F1-Score which is a harmonic balance between precision and recall.

Area Under Curve

AUC is a metric that combines both precision and recall.

Since in this business case, there is a need to minimize both false positives and false negatives, I compared using F1-Score and AUC score, and I decided to go with AUC as the metric for model evaluation. **Figure 7** shows a comparison of the performance of the seven models using ROC AUC as the models evaluation metric.

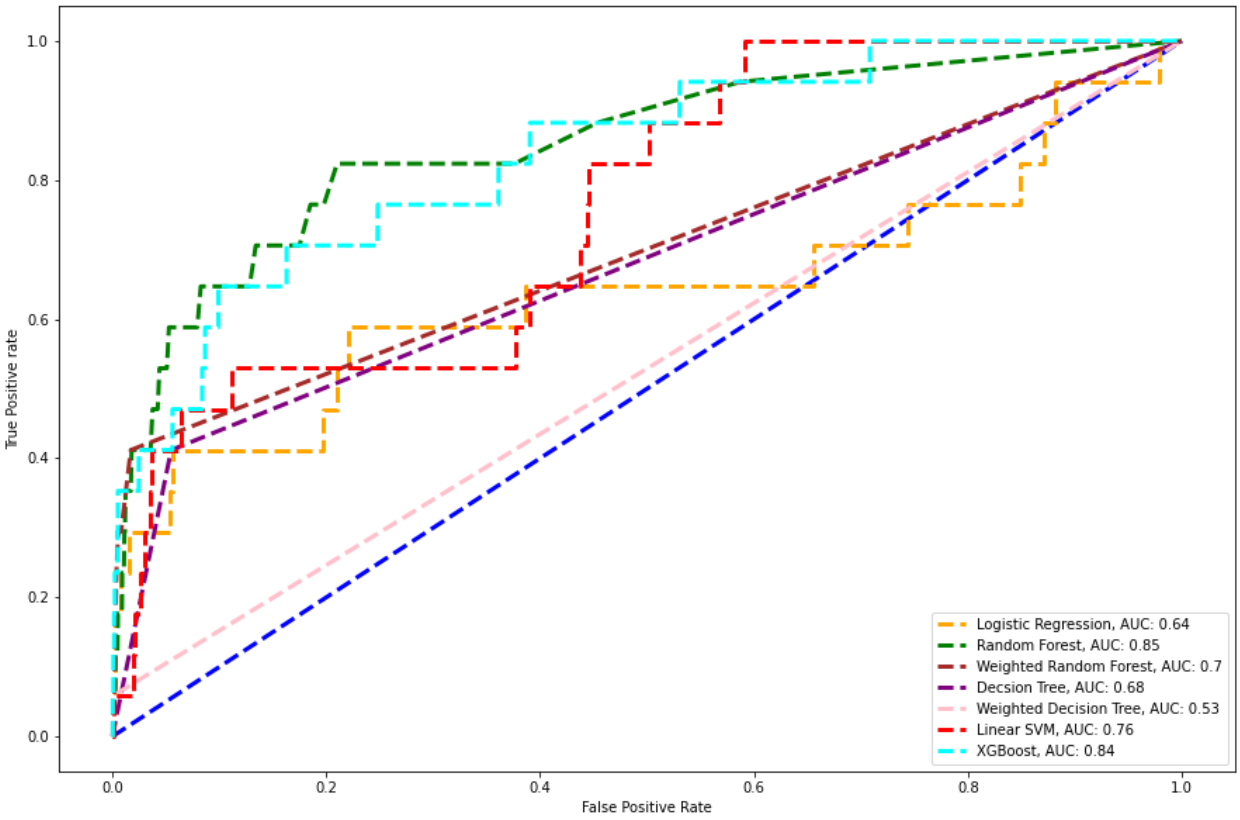

Figure 7. Model Comparison. The ROC curves of the 7 tested ML models comparing their performance in terms of AUC. Random Forest scores highest with AUC = 0.85.

From **fig. 7** we can conclude that Random Forest and XGBoost perform very close, however Random Forest is slightly better. Therefore, I chose the latter as my model for prediction. An added benefit of using Random Forest is model ***interpretability*** i.e. we get a better understanding of the reasoning for the prediction decision, by knowing the features used in making the prediction and their relative importance (**fig. 8**).

5. Model Tuning

Grid Search to find best hyperparameters
After selecting Random Forest, I use Randomized Search Cross Validation in order to tune the model's hyperparameters below:

bootstrap
max_depth
max_features
min_samples_leaf
min_samples_split
n_estimators

The scoring for this search is set to the AUC.

6. Building an Ensemble

The next step to follow is to build an ensemble of the Random Forest model, this is due to the under-sampling and over-sampling that I have done before in order to balance the data. In order for the models to use more non failure data, and to avoid overfitting with failure data via oversampling, I keep the same amount of data points but build an ensemble of 10 random forests, each with the optimal parameters found before.

Ensemble Metrics
Once an ensemble of forests is built, I use a hard-voting system to determine the class. This means that I run the same prediction across all of the models, and take the majority vote as to what the predicted class is.

7. Ensemble Validation

For validation, I use RepeatedStratifiedKFold validation. This validation process repeatedly randomly splits the original training data into training dataset and validation dataset. In this case, I did 10 data splits and 10 repeats, for a total of 100 cross validations, using AUC as the scoring metric. This makes the validation of the ensemble robust, to rule out any chance of overfitting. **Fig. 8** shows the resulting ROC curves with mean AUC = 0.896 , over 100 iterations.

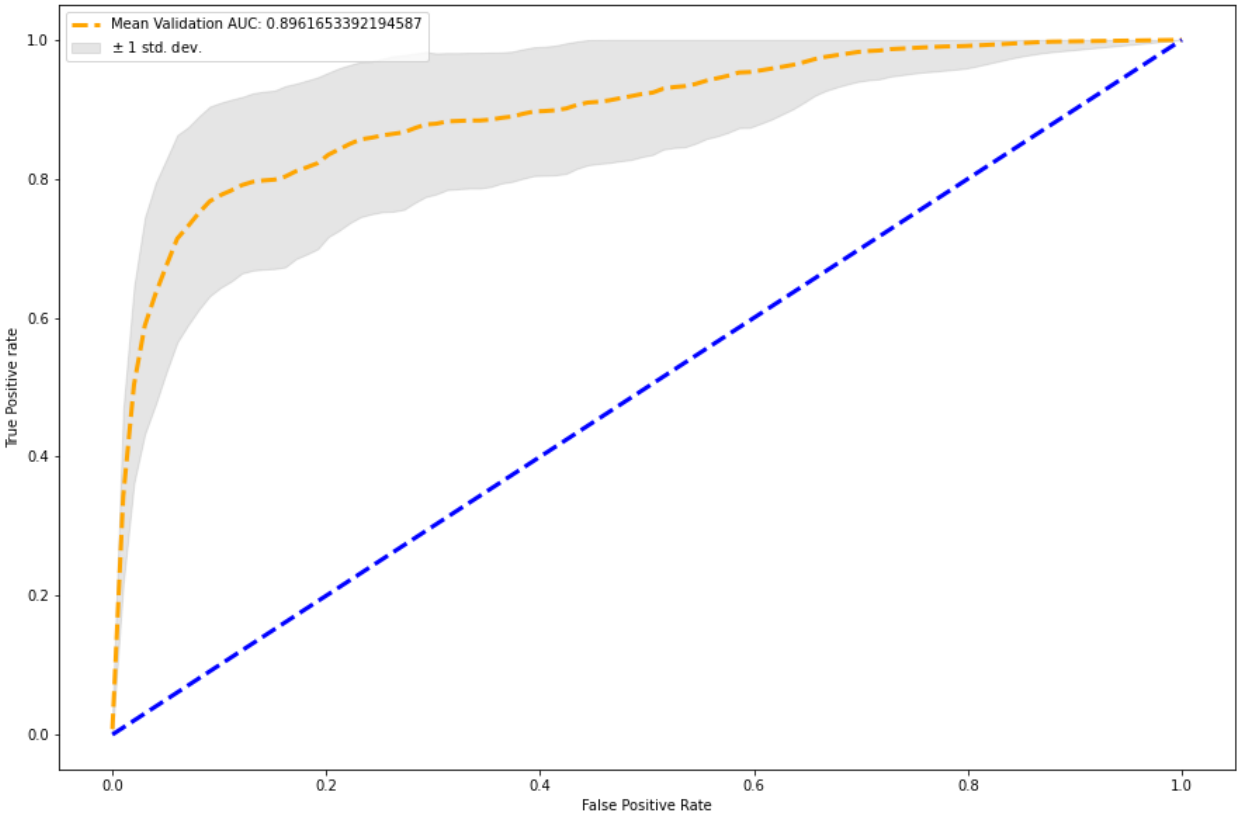

Figure 8. Ensemble Validation. The ROC curves of the ensemble validation with mean AUC = 0.896 over 100 cross validations.

8. Ensemble Test

The performance of the ensemble classifier is tested on the original test dataset. The comparison of the model performance on the validation dataset and the test dataset is shown in **fig. 9**.

9. Model Saving

The trained models are saved under /Saved Models

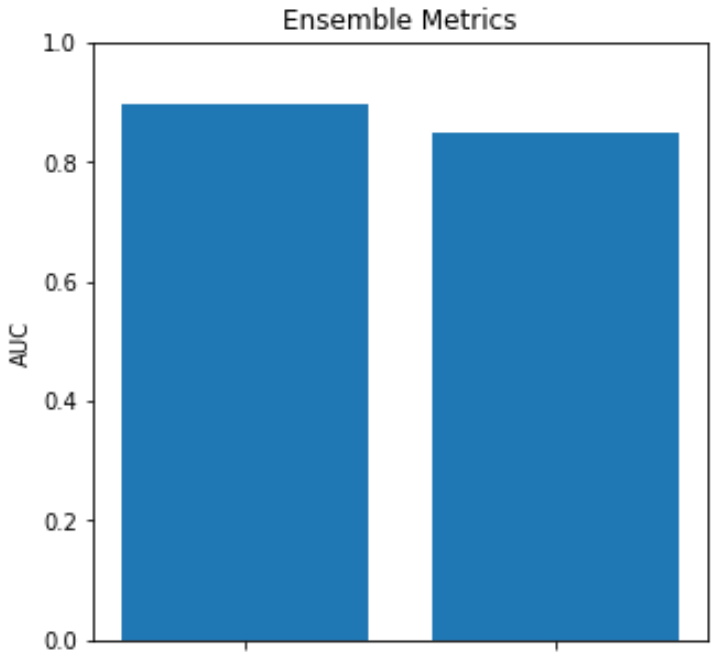

Figure 9. Validation vs. Test. A comparison of model performance on validation dataset and Test dataset. Validation AUC = 0.896 while Test AUC is slightly less.

Model Deployment

Model Hosting

For hosting our model, and building an API through it, I use the popular hosting library flask.

I set up the service in **server.py**, and follow conventional flask documentation outlined in:

https://flask.palletsprojects.com/en/1.1.x/quickstart/

The project structure looks like this:

-> project dir/
    ->analytics.py
    ->utils.py
    ->server.py
    ->uwsgi.ini
    -> Saved Models/
        -> forest_model_1.sav
        -> forest_model_2.sav
        -> forest_model_3.sav
        -> forest_model_4.sav
        -> forest_model_5.sav

-> forest_model_6.sav
-> forest_model_7.sav
-> forest_model_8.sav
-> forest_model_9.sav
-> forest_model_10.sav

utils.py

utils.py holds functionality that is not analysis or server related, therefore it has miscellaneous data/model loading functions:

load_ensemble takes the ensemble saved files as input and loads the models, storing them into a list.

save_model saves a single model.

load_data loads the excel/csv/tsv file and returns a pandas dataframe.

normalize_data applies a min max scaler to each of the feature columns in the provided dataframe.

preproccess_data applies all data balancing techniques described before.

analytics.py

analytics.py holds all analysis related functionality, this encompasses model predicting, data exploration and basic analysis.

ensemble_predict takes in the stored models in the ensemble and makes a prediction.

train_model builds one random forest with the provided features and target datapoints.

train_ensemble is the root function that builds and trains all models in the ensemble, it calls the following functions.

```
load_data()
preprocess_data()
train_model()
save_model()
```

server.py

Imports and application startup.

The server uses the *load_ensemble* function from **utils** to load in the saved models.

Then I set up the routing: I link it to the base url. I expect the requests to be *post* requests rather than *get* requests. In order to collect the data that is sent to the api, ***flask*** request instance is used and the

data is collected from it. The Analysis document includes an example of a request and the corresponding output provided from the server.

In order to call the service, I use the python requests library, and embed our features in a dictionary, then ***jsonify*** that dictionary for the request to understand it.

To ensure the server can run at any time, I create a ***wsgi*** configuration. This is done in a file called **uswgi.ini**. this routes the application configuration in the correct way. The following is the configuration that is set up:

```
#In uwsgi.ini file
[uwsgi]
Module = main
Callable = app
Master = true
```

Docker Building

For best practice, the service is then dockerized, this means that the tool is put into a package that contains all project files and dependencies so it can be easily run anywhere. It builds the project with all dependencies and requirements, making it run the same on any device or system. It builds this tool in a virtual environment.

In order to do this, two files are created, first is ***requirements.txt***. This tells the environment exactly what library dependencies are being used:

```
numpy == 1.19.5
flask == 1.1.2
joblib == 1.0.0
scikit-learn == 0.24.1
pandas == 1.2.0
imbalanced-learn == 0.8.0
in requirements.txt
```

The second is the ***Dockerfile***, this outlines the steps docker should follow when building the project as follows:

```
FROM python:3.8
```
lets docker know which version of python it should install in this environment.

```
COPY ./requirements.txt .
```
creates a copy of our requirements.txt in the docker environment, allowing it to read the file.

```
RUN pip install --upgrade pip
RUN pip install -r requirements.txt
```
These two lines install pip (Pip Installs Packages) and all the requirements listed in requirements.txt in the virtual environment.

```
EXPOSE 80
```

COPY . .
CMP python3 server.py
Finally, we copy all files found in the directory to be able read from them, and run the command line command "python3 server.py" which will host our tool.

In order to run the docker container, I first build it then run it.

To build it, we follow the following command format:

docker build -t [project name] [directory]

which when translated becomes:

docker build -t app

Then to run the app, run the following command:

docker run -p 5000:5000 app (expose app to 5000, the flask run port)

Server Connection
The server is set up on a AWS; EC2 t2.micro instance. The public ip address for the instance is 18.221.149.92. So in order to connect to it I use the same request setup used for local.

Code used to call api
response = requests.post("http://18.221.149.92:5000/", data=json.dumps(sample_data))
response
json.loads(response.content)