Smart Predictive Maintenance for Industrial Equipment : Final Report

Stella HU - DIA2 A5 ALT ESILV - Applied Machine Learning I

The aim of this project is to do predictive maintenance using ensemble learning, imbalanced data handling, reinforcement learning, and AutoML. The goal is to predict machine failures before they occur, optimizing maintenance schedules and reducing downtime. This project will integrate key machine learning techniques covered in the course, such as ensemble learning, handling imbalanced data, reinforcement learning, and automated machine learning (AutoML), to deliver an effective and scalable solution.

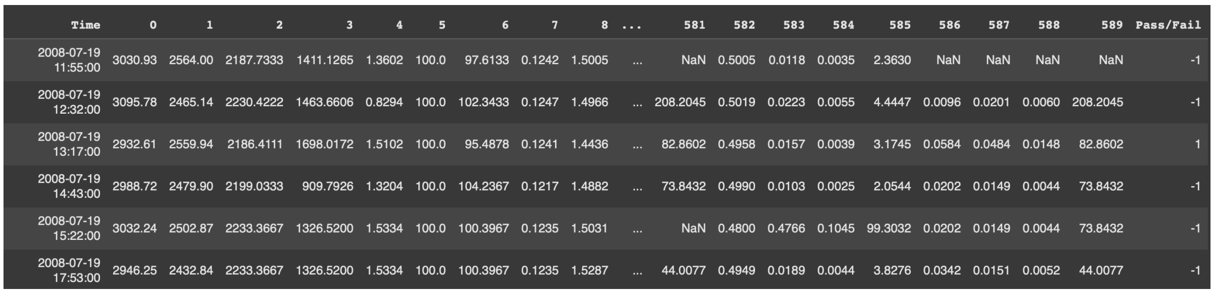
The dataset used is : **SECOM - Semiconductor manufacturing process data (2008)**

FAB data collected from 590 sensors in the semiconductor manufacturing process. The data consists of two files, `'secom.data'` and `'secom_labels.data'`. The `'secom.data'` file consists of 1567 * 590 matrices with 590 features and 1567 examples, as each of the 590 features corresponds to a sensor's collected data. `'secom_labels.data'` file contains classifications value and data time stamp value for each 1567 data. In `'secom_labels.data'` file, **-1 indicates pass (normal) and 1 indicates fail (abnormal)**. Here, the dataset used is a single CSV file version to facilitate data loading,that can be found on Kaggle (https://www.kaggle.com/datasets/paresh2047/uci-semcom/data)

Data preprocessing

1. EDA

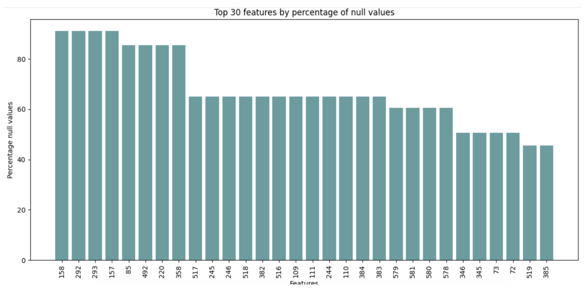
Exploring the dataset and understanding its constitution is the first step to acknowledge the stakes and determine data preprocessing steps. Firstly, it was important to know how did the initial dataset looked like in a dataframe :

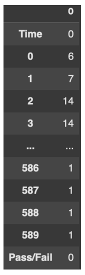


Time	0	1	2	3	4	5	6	7	8	...	581	582	583	584	585	586	587	588	589	Pass/Fail
2008-07-19 11:55:00	3030.93	2564.00	2187.7333	1411.1265	1.3602	100.0	97.6133	0.1242	1.5005	...	NaN	0.5005	0.0118	0.0035	2.3630	NaN	NaN	NaN	NaN	-1
2008-07-19 12:32:00	3095.78	2465.14	2230.4222	1463.6606	0.8294	100.0	102.3433	0.1247	1.4966	...	208.2045	0.5019	0.0223	0.0055	4.4447	0.0096	0.0201	0.0060	208.2045	-1
2008-07-19 13:17:00	2932.61	2559.94	2186.4111	1698.0172	1.5102	100.0	95.4878	0.1241	1.4436	...	82.8602	0.4958	0.0157	0.0039	3.1745	0.0584	0.0484	0.0148	82.8602	1
2008-07-19 14:43:00	2988.72	2479.90	2199.0333	909.7926	1.3204	100.0	104.2367	0.1217	1.4882	...	73.8432	0.4990	0.0103	0.0025	2.0544	0.0202	0.0149	0.0044	73.8432	-1
2008-07-19 15:22:00	3032.24	2502.87	2233.3667	1326.5200	1.5334	100.0	100.3967	0.1235	1.5031	...	NaN	0.4800	0.4766	0.1045	99.3032	0.0202	0.0149	0.0044	73.8432	-1
2008-07-19 17:53:00	2946.25	2432.84	2233.3667	1326.5200	1.5334	100.0	100.3967	0.1235	1.5287	...	44.0077	0.4949	0.0189	0.0044	3.8276	0.0342	0.0151	0.0052	44.0077	-1

This way, it was possible to confirm the existence of the 590 sensors' data, the target column 'Pass/Fail' and a 'Time' column.

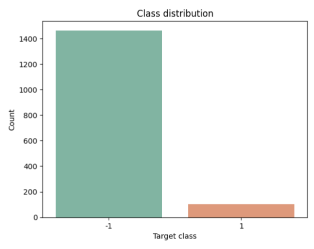
Visualizing the number of null values and the class distribution of the targets was also necessary. With a dataset with so many features, it is important to filter out the features that can behave as noise or be unnecessary for training.





	0
Time	0
0	6
1	7
2	14
3	14
...	...
586	1
587	1
588	1
589	1
Pass/Fail	0

Almost 30 columns have more than 50% of null values, and many other columns have at least one null value. When searching more information about the dataset, it was noticeable that only one column is not numerical, which is the ‘Time’ column. Searching for the types is important if the null values need to be filled.



The target class distribution shows that the dataset is highly unbalanced. The fail (1) class is the minority class. Not balancing the dataset can lead the model to be unable to predict failures.

2. Feature selection

As there are too many features for the future models to work efficiently, is was important to select the right features. Firstly, it seemed evident to remove the columns with more that 50% of missing values. There was 28 features that had more than 50% percent of missing values, leaving 562 features at this point. The ‘Time’ column was also removed to only let numerical columns in the dataset. Finally, low variance features, therefore features that only have one unique value were also dropped out of the dataset because they don’t bring any relevance, representing 116 columns. In the end, the shape of the dataframe became (1567, 447), meaning that 446 features remained, instead of 590 (without taking in account the target column).

3. Missing values

The fact that all features were numerical makes the imputation of missing values easier. Indeed, the missing values were all filled with the mean of each column they belong using SimpleImputer `SimpleImputer(strategy='mean')`.

4. Dimensionality reduction and scaling

Even though almost 150 columns were dropped, there was still too many dimensions to ensure efficiency for model training. The strategy was therefore to apply **Principal Component Analysis** (PCA) to retain around 95% of the explained variance, reducing the number of features to a more manageable level.

Before applying PCA, it was important to define the features and target. The target was obviously 'Pass/Fail' column (y) and the others columns were the features (X). After that, the last step before PCA was to put our features on the same scale.

As our dataset has a lot of features on very different scales, many of them having a gaussian distribution, and their unities are mostly unknown, it was preferable to apply standardization. The role of standardization is to put the mean to 0 and the standard deviation to 1. `StandardScaler()` was therefore used on the features to standardize them, while the target classes were put to 0 (pass) and 1 (fail) for better understanding.

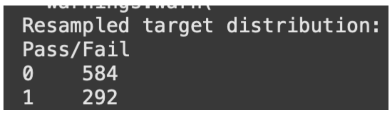
Applying PCA with `PCA(n_components=0.95)` enabled to reduce the number of dimensions of the dataset, reducing the number of features to 162.

5. Class balancing

The last issue to resolve before training models was the imbalance between pass and fail classes. Indeed, as seen before, the fail (1) class represented less that 10% on the original dataset. It was therefore relevant to combine SMOTE and undersampling to achieve a 2:1 ratio between the majority (pass) and minority (fail) classes.

A fully balanced 1:1 ratio can over-represent the minority class, especially in a real-world industrial equipment situation where the majority class dominates. A 2:1 ratio helps ensure that the majority class still contributes significantly to model training while reducing the imbalance enough to improve model performance on the minority class.

SMOTE generates synthetic examples for the minority class to increase it to 20% of majority class while the majority class is also undersampled to 50% of minority. combining SMOTE and undersampling helps avoid overfitting caused by oversampling alone and reduces the size of the majority class without throwing away too much information.

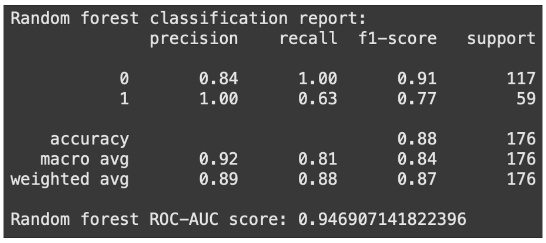


```
Resampled target distribution:
Pass/Fail
0    584
1    292
```

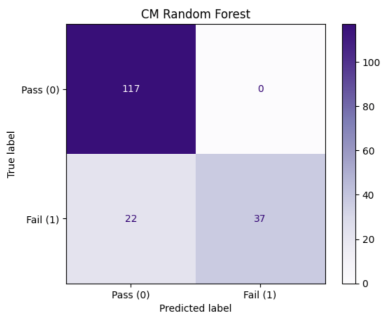
Bagging/Boosting models

Three models were evaluated: Random forest, Adaboost, and XGboost. Performance metrics included accuracy, precision, recall, F1 score, and ROC-AUC. The dataset used was the one previously preprocessed.

1. Random forest

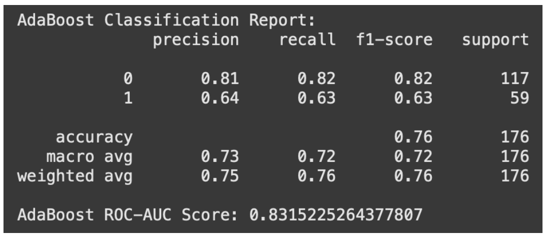


```
Random forest classification report:
              precision    recall  f1-score   support

           0       0.84      1.00      0.91       117
           1       1.00      0.63      0.77        59

    accuracy                           0.88       176
   macro avg       0.92      0.81      0.84       176
weighted avg       0.89      0.88      0.87       176

Random forest ROC-AUC score: 0.946907141822396
```

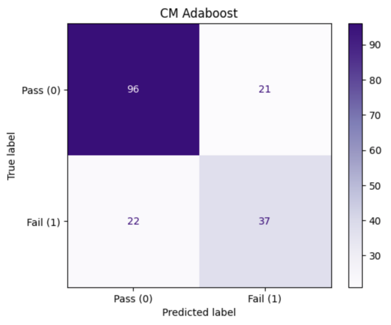


The random forest model shows a pretty good performance with an accuracy of 0.88. With its precision of 1.00 for (1) class, it perfectly predicts the Pass class when it's actually Pass. However, it failed at detecting some failures when they were actually failures, meaning that it is sensitive to false negatives. The ROC-AUC score measures how well the model can rank predictions. With a score of 0.95, the random forest model has a 95% probability of correctly ranking a randomly chosen positive instance higher than a randomly chosen negative instance.

2. Adaboost

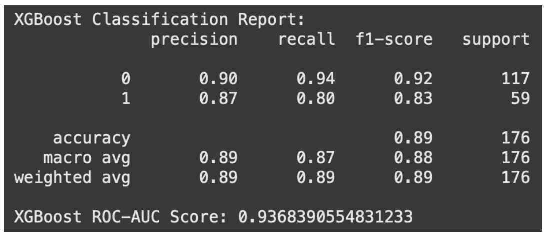


```
AdaBoost Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.82      0.82       117
           1       0.64      0.63      0.63        59

    accuracy                           0.76       176
   macro avg       0.73      0.72      0.72       176
weighted avg       0.75      0.76      0.76       176

AdaBoost ROC-AUC Score: 0.8315225264377807
```

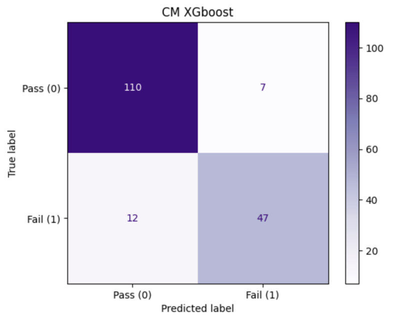


Adaboost also shows good performance even though the metrics show that it doesn't perform as good as Random forest. The confusion matrix indicates that Adaboost also has the same problem as Random forest for failing at detecting actual failures, but also predicts false positives, which means that few failures were detected when there were actually not. The ROC-AUC score of 0.88 confirms the statement.

3. XGBoost



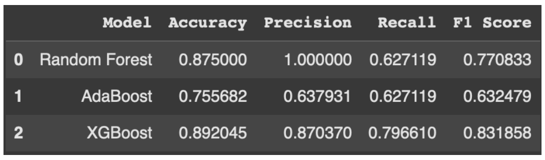
```
XGBoost Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.94      0.92       117
           1       0.87      0.80      0.83        59

    accuracy                           0.89       176
   macro avg       0.89      0.87      0.88       176
weighted avg       0.89      0.89      0.89       176

XGBoost ROC-AUC Score: 0.9368390554831233
```



XGboost shows the best accuracy so far with an accuracy of 0.89. Even though its precision scores are not as high as random forest's ones, it shows higher F1 scores, with means that the model is highly effective at predicting the positive class. There is a good balance between precision and recall, showing that the model does not generate too many false positives while identifying most true positives.

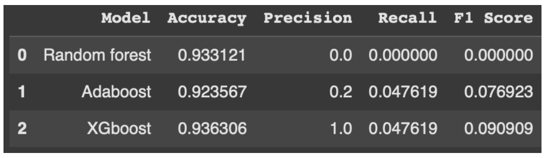
XGBoost emerges as the best-performing model. Indeed, applied to real-world situations, it is better to predict the less false negative possible. In case of industrial machine failure, a machine predicting inexistent failures isn't as dramatic as not predicting a failure when there is one. With 12 failures predicted as Pass (22 for the other models), and only 7 false positives, XGboost stands up as the best model for real-life predictive maintenance for industrial equipment situation.

4. Comparison on unbalanced data



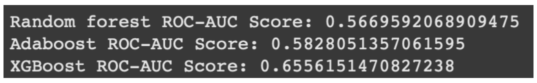
	Model	Accuracy	Precision	Recall	F1 Score
0	Random Forest	0.875000	1.000000	0.627119	0.770833
1	AdaBoost	0.755682	0.637931	0.627119	0.632479
2	XGBoost	0.892045	0.870370	0.796610	0.831858

Models' metrics on balanced dataset



	Model	Accuracy	Precision	Recall	F1 Score
0	Random forest	0.933121	0.0	0.000000	0.000000
1	Adaboost	0.923567	0.2	0.047619	0.076923
2	XGboost	0.936306	1.0	0.047619	0.090909

Models' metrics on unbalanced dataset



```
Random forest ROC-AUC Score: 0.5669592068909475
Adaboost ROC-AUC Score: 0.5828051357061595
XGBoost ROC-AUC Score: 0.6556151470827238
```

On the unbalanced dataset (without SMOTE and undersampling), the accuracies are better for each model but the metrics show poor scores for precision, recall and F1 scores, as well as ROC-AUC scores. These metrics indicates a domination of a class, here 0 (Pass), introducing biaises in the models' trainings and in the accuracies. The models in fact predicts the Pass instances but as there are the majority, they increase the accuracy while strongly failing at detecting failures (1).

These insights highlight the importance of balancing the dataset, and more generally preprocessing, before model training.

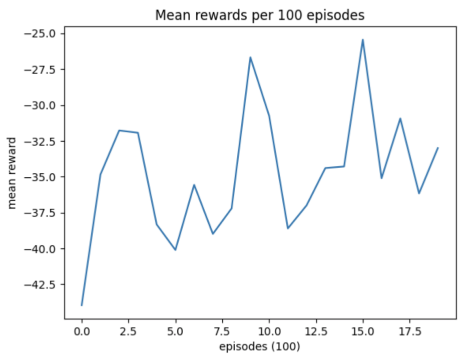
Reinforcement learning

Reinforcement learning applied to the current situation aims to modelize and optimize maintenance decisions in a simulated environment. The approach used is Q-learning algorithm, that uses the learning of Q-values (action-state) to determine the best actions.

The first step was to initialize a Q table to stock the values of actions-states pairs, with all the values initialized to 0. A customized reward function is then defined to assign rewards based on the current state and the chosen action, which is performing maintenance or taking no action. Rewards are made to reflect real-world situations, with high rewards for performing maintenance during a failure and penalties for unnecessary maintenance or failure without intervention.

The Q-learning training process continues over multiple episodes, while key parameters include the number of episodes set to 2000, the maximum steps per episode set to 100, a learning rate of 0.8 and a number of states of 10 representing different levels of machine health. During each step of an episode, the agent selects an action based on the -greedy policy, observes the next state and the received reward, and updates the Q-value.

To evaluate the agent's performance, the average rewards obtained over per 100 episodes are calculated. However, the results show that the rewards remain largely negative and little improvement.



Despite the training, the negative rewards suggest that the learned policy might not effectively handle failures. This implementation demonstrates how Q-learning can be applied to determine maintenance policy, but the results reveal the difficulties in achieving good outcomes. The negative rewards highlighst dysfunctions in the reward function or algorithm configuration. Future work should focus on improving the reward structure, adjusting hyperparameters, or experimenting with other reinforcement learning algorithms to overpassed the faced issues.

AutoML

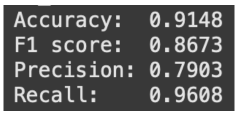
AutoML is a great way to find the est model by automating model selection and hyperparameter tuning. Here, **TPOT** which is an AutoML tool made to automatically design and optimize a machine learning pipeline is used on the previously preprocessed SECOM data. TPOT uses genetic programming to explore various model combinations, hyperparameters, and preprocessing steps to identify the best performing configuration for detecting fail or pass classes.

To build the pipeline, the resampled data is reused for training and testing, ensuring that class imbalance has been fixed. The data is then split into training and testing sets using an 80/20 ratio with `random_state=42` for reproducibility, which are the same as the previously manually tested models.

`TPOTClassifier(generations=5, population_size=20, random_state=42)` is then used to configure TPOT with 5 evolutionary iterations, a population size set to 20, defining the number of pipelines in each generation and a random state of 42 to make sure that the results stay consistent across runs. The TPOT classifier is trained to the resampled preprocessed data and iteratively evaluates various machine learning pipelines.

The final pipeline is used to make predictions on the test set and the accuracy of the selected pipeline is calculated using the accuracy score, as the best model has the best accuracy score (default). Finally, the best pipeline discovered by TPOT is exported to the python file `tpot_best_model_pipeline.py` ,which is ideal for future use.

The best accuracy found is 0.91, a very good accuracy achieved thanks to the automation of model selection. In the actual application, the accuracy doesn't seem to be the most ideal score : previous comparison models trained with balanced / unbalanced data scores showed that the good accuracy scores achieved by the models trained by unbalanced data, sometimes even better than the ones trained with balanced data, didn't implicate the good prediction of failures (with recalls, precisions and f1 scores near 0 and very average ROC-AUC scores). This is why the other metrics are displayed for the model with the best accuracy, to check that despite its very good accuracy, it is able to predict the less false negative possible (predicting the less 'Pass' (0) class when it is a failure (1).



```
Accuracy:  0.9148
F1 score:  0.8673
Precision: 0.7903
Recall:    0.9608
```

The limitations of TPOT include computationally intensiveness : the search for the best model is very time costly, around 17 minutes were needed with a CPU. AutoMM therefore reduces significantly the human time needed for parameters tuning and model search but at the expense of computational time.

Recommendations for future works

For future works, other models can be explored, additional ensemble models or even deep learning models based on neural networks. To balance the data, it is also possible to experiment with other balancing techniques, such as ADASYN or cost-sensitive learning. Saving trained models can also avoid losing best performances. Indeed, while training models, it was sometimes possible to obtain much better scores than the ones presented in this report.

The automation of model selection can also be improved by learning more about the necessary computer resources, and by testing other techniques such as H2O or MLflow. The metric to select the best model can also be changed to F1 score for example, or another metric depending on which metric would be the more important in the situation.

The reinforcement learning part can highly be improved by simulating more adapted environments and testing other parameters. Other advanced algorithms like Deep Q-Networks can also be explored and the time dimension can also be incorporated for dynamic policy updates.

In the end, testing on the SECOM dataset is a nice introduction to predictive maintenance for industrial equipment, and testing the system on larger datasets from different industries would be an ideal way to improve the training of effective models made for predictive maintenance tasks.