APS Failure Detection – End-to-End ML Pipeline

Muhammad Muqsit Islam
Instructor: Dr. Lukács Gergely István

Report presented for the course of
Data Mining and Machine Learning

Pázmány Péter
Katolikus Egyetem

Semester: 2025-2026

Department of Information Technology and Bionics

Pazmany Peter Catholic University

Budapest, Práter u. 50/A, 1083

Contents

1 Introduction

The goal of this project is to build a predictive model for detecting failures in the Air Pressure System (APS) of Scania trucks. The APS generates pressurised air for safety-critical components such as the braking system and gear changes. In the dataset used in this assignment, the positive class corresponds to failures of a specific APS component, while the negative class corresponds to trucks where the APS is not at fault (failures stem from other components). The task is to decide, based on anonymised operational sensor data, whether the APS should be repaired.

From a practical perspective, the two types of prediction errors have very different consequences. A false positive means that the APS is checked and possibly repaired even though it is not faulty, leading mainly to unnecessary maintenance costs and downtime. A false negative is more critical: a faulty APS is not detected and remains in use, which may result in safety issues and more severe failures later on. Because of this asymmetry, and because the dataset is strongly imbalanced (far more negative than positive examples), accuracy alone is not an appropriate metric. The associated Kaggle competition therefore uses

balanced accuracy as the main evaluation measure, giving equal weight to the performance on the minority and majority classes.

At the end of the day, you'd much rather annoy a mechanic with a false alarm than let a truck with faulty brakes onto the highway. The cost of a false positive is just money, but the cost of a false negative is way too high to ignore, so optimizing for recall is the only strategy that actually makes sense here.

In this project I follow a structured workflow: (i) exploratory data analysis to understand the main characteristics of the APS dataset (missing values, class imbalance, outliers), (ii) data preparation including feature filtering based on missingness, imputation and scaling, (iii) training and tuning several classification algorithms within sklearn Pipelines to avoid data leakage, and (iv) model selection and threshold optimisation with respect to balanced accuracy. The final model is then used to generate predictions for the hidden test set and submitted to the Kaggle competition. Throughout the work, reproducibility is ensured by fixing random seeds and by keeping all preprocessing and learning steps in a single, well-defined pipeline.

2 Exploratory Data Analysis

Before training any models, I carried out a short exploratory analysis to understand the main characteristics of the APS dataset and to identify potential issues such as severe class imbalance, missing values and outliers.

2.1 Target distribution and class imbalance

The first step was to inspect the distribution of the target labels in the training data. The labels are given as "neg" (no APS failure) and "pos" (APS component failure). After encoding them as 0 and 1, I computed the class counts and relative frequencies.

As expected from the problem description, the data is strongly imbalanced, with the negative class dominating the dataset and the positive class representing only a small fraction of all examples. This imbalance motivated two later design choices:

- explicitly incorporating class-imbalance handling in the models
- using balanced accuracy as the main model selection metric instead of plain accuracy

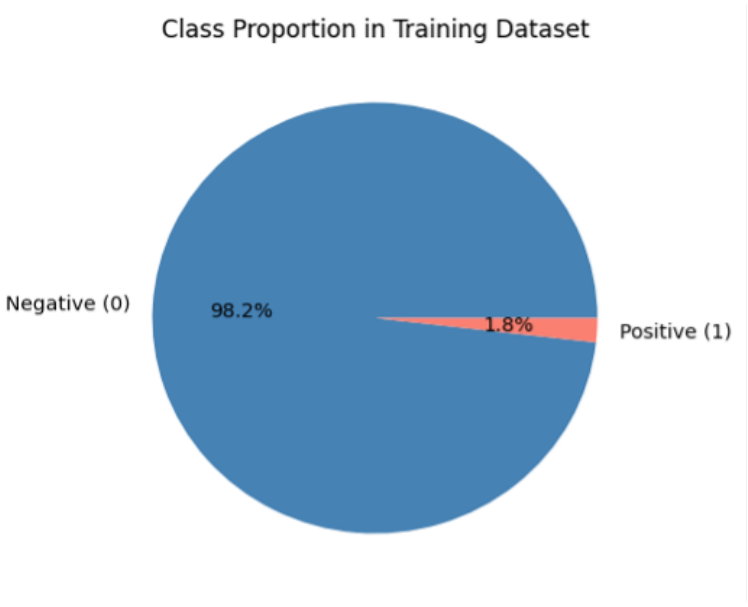

Figure 1: Class imbalance by percentage

2.2 Feature overview and missing values

The dataset contains 171 anonymised numeric attributes. Since many of them have missing entries, I first analysed the overall pattern of missingness before deciding which features to keep.

To understand how aggressive the feature dropping should be: I computed, for a range of thresholds, how many columns would be removed if I dropped all features with a missing-value ratio above that threshold. For example, at a threshold of 0.1 (10 percent missing values) already 28 features would be discarded, while at 0.5 only 8 features are removed and 162 remain. Increasing the threshold beyond 0.5 has almost no effect: between 0.5 and 0.7 only one additional feature is dropped, and at 0.9 no feature would be removed at all. This suggested that only a small subset of attributes are truly "very sparse".

Threshold	N Dropped	N Remaining
0.1	28	142
0.2	24	146
0.3	10	160
0.4	9	161
0.5	8	162
0.6	8	162
0.7	7	163
0.8	2	168
0.9	0	170

Table 1: Number of features dropped and remaining at different missing value thresholds.

To visualise this trade-off, Figure 2 shows the same numbers as a line plot: the x-axis is the drop threshold (from 0.1 to 0.9), and the y-axis is the number of features that would be removed. The curve decreases steeply between 0.1 and 0.3, and then flattens out. This indicates that a very strict threshold (e.g. 0.1) would aggressively shrink the feature space, while thresholds around 0.5 already capture most of the very sparse variables without discarding too many columns.

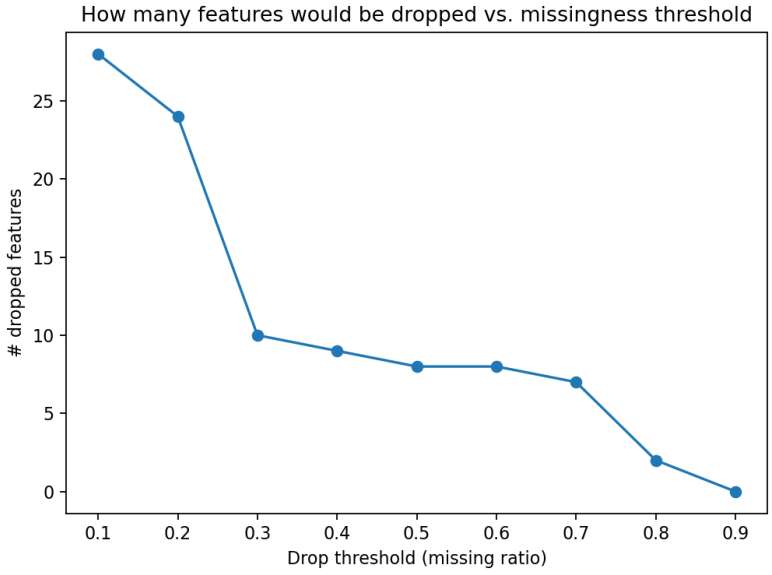

Figure 2: dropped features vs threshold

While Table 1 and Figure 2 look at aggregate counts, Figure 3 shows the distribution of missing-value ratios across all features. Most attributes have very low missingness (a large bar near 0 on the x-axis), with

only a few features having missing ratios above 0.2. This confirms that the dataset is not uniformly noisy; instead, a small group of features is problematic, and the majority are reasonably complete.

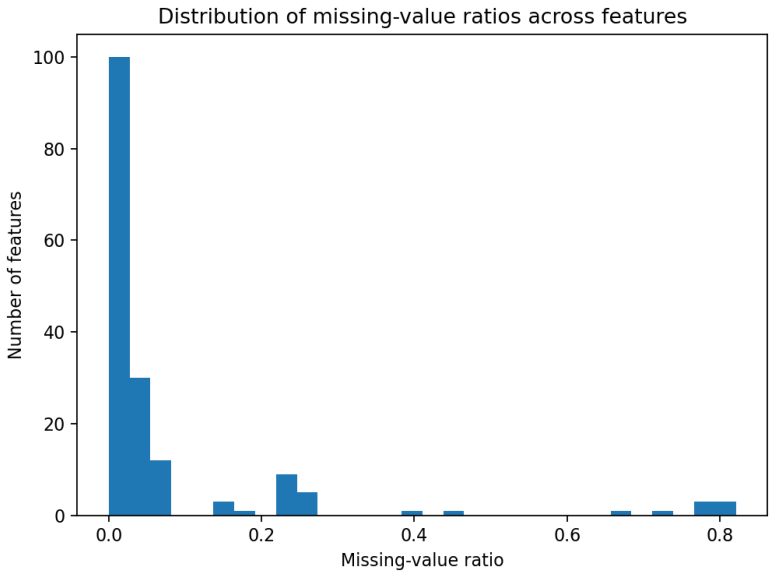

Figure 3: Distribution of MV ratio

To inspect these problematic attributes more closely, Figure 5 plots the top 25 features ordered by missing-value ratio. Here we can clearly see that a handful of variables have missingness above 0.75, followed by a second group with ratios between roughly 0.25 and 0.45, and then a long tail around 0.2–0.25. Because the feature names are anonymised, I cannot interpret these variables semantically, but from a purely statistical perspective they contribute very little usable information.

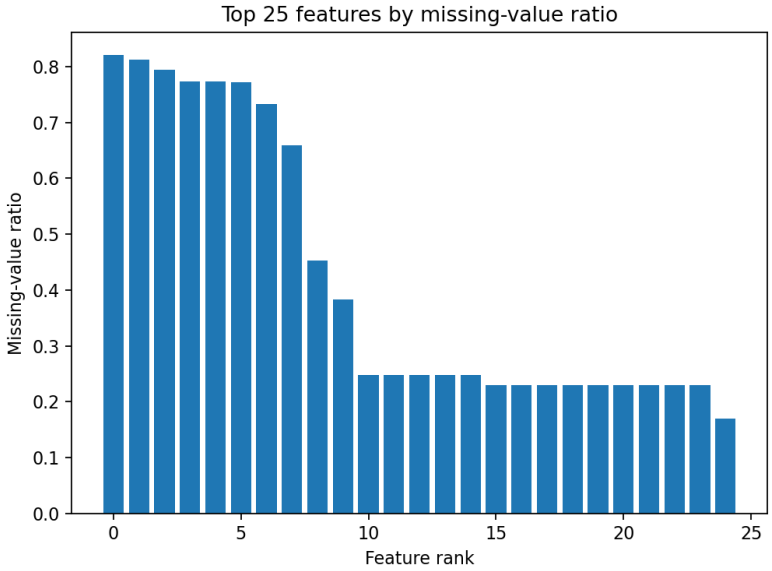

Figure 4: Top 25 MV ratio

Based on this analysis, I chose a drop threshold of 0.5 for the DropHighMissingColumns transformer. With this setting, the worst 8 features (those with more than 50 percent missing values) are removed in each

training fold, while 162 attributes remain. This provides a mild dimensionality reduction and gets rid of the clearly unusable variables, but keeps almost all potentially informative features for the subsequent modelling steps.

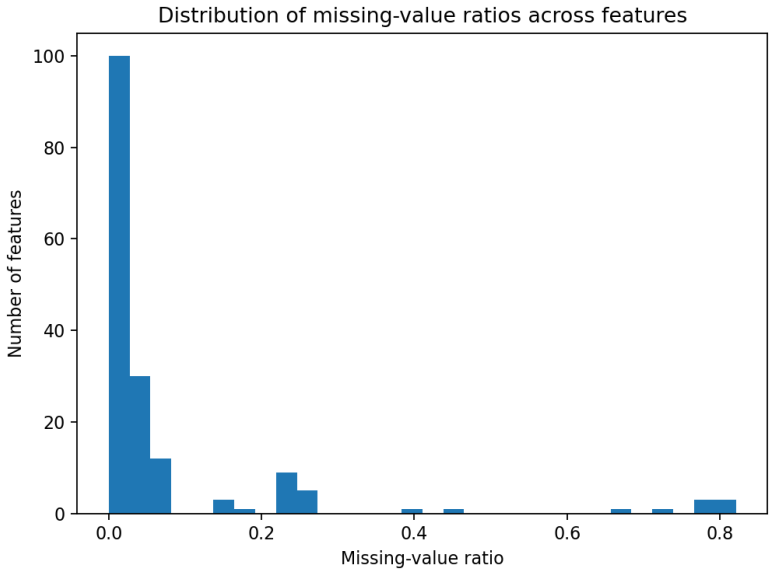

Figure 5: Top 25 MV ratio

3 Data Preparation

I applied a small but consistent set of preprocessing steps that are implemented as sklearn transformers and combined into Pipelines. Keeping all transformations inside the pipeline was important to avoid data leakage during cross-validation and to make the whole workflow reproducible.

3.1 Removing highly sparse features

The EDA showed that only a small subset of attributes suffers from extremely high missingness (Section 2.2). To remove these clearly unusable variables in a controlled way, I implemented a custom transformer DropHighMissingColumns.

For each training split, this transformer:

- computes the missing-value ratio for every feature
- identifies the columns whose ratio exceeds a predefined threshold
- stores the list of columns to be dropped

During the subsequent transform step, only those columns are removed from both the training and validation data. Because the transformer is placed inside the sklearn Pipeline, it is refitted separately on each training fold, so the decision "which columns are too sparse" is always based only on the current training data and never on the corresponding validation fold.

Guided by the analysis in 1, I set the drop threshold to 0.5 such that features with more than 50 percent missing values are discarded. This removes 8 of the 170+ attributes and keeps 162 features for modelling, which turned out to be a good compromise between robustness and retaining potentially informative variables.

3.2 Imputation and scaling

After dropping high-missing features, the remaining attributes still contain some missing values. For all models I applied median imputation using SimpleImputer(strategy="median"). The median is less sensitive to the heavy-tailed distributions and outliers observed in many variables, and in preliminary tests it performed at least as well as mean imputation.

Scaling is handled model-specifically:

- For Logistic Regression, which is sensitive to the relative scale of the inputs, I added a StandardScaler step after imputation. This centres each feature to zero mean and unit variance and stabilises the optimisation
- For tree-based models (Random Forest and XGBoost), no explicit scaling is applied. These algorithms split on feature thresholds and are largely invariant to monotonic transformations of individual features, so standardisation is not necessary

All of these operations (dropping, imputation, optional scaling) are defined inside the corresponding sklearn Pipelines, ensuring that the exact same transformations are applied to the training folds, validation folds and, finally, to the Kaggle test set.

3.3 Label encoding

The competition provides the target variable as strings "neg" (no APS failure) and "pos" (APS failure). For convenience, I converted these labels to integers using a small helper function:

- "neg" $\rightarrow 0$
- "pos" $\rightarrow 1$

If any unexpected label values were present, the function would raise an error. This simple check helped catch potential inconsistencies early. All subsequent models are trained on this numeric target representation.

Overall, the data preparation is kept deliberately minimal: instead of heavy feature engineering, the focus is on a clean and reproducible pipeline that handles missing values, scales features where necessary and avoids any kind of leakage between training and validation data.

4 Algorithms

- **Logistic Regression** Used as a simple, interpretable baseline. With proper scaling and class weights it often performs surprisingly well on high-dimensional tabular data and helps to benchmark more complex models.
- **Random Forest** Non-linear tree ensemble that can capture interactions between features, is robust to outliers and different scales, and works well "out of the box" on tabular data.
- **XGBoost** Gradient-boosted decision trees with explicit support for class imbalance. Selected as a stronger, more flexible model that can exploit subtle non-linear patterns in the APS sensor data.

5 Hyperparameter Tuning

For all models I used the same tuning strategy:

- Stratified 5-fold cross-validation, with shuffle=True and a fixed random seed to make the results reproducible.

- Tuning via GridSearchCV on top of the sklearn Pipelines, so that every combination of hyperparameters is evaluated with the full preprocessing (dropping sparse features, imputation, scaling where needed).
- The selection metric is balanced accuracy, matching the competition's evaluation measure and compensating for the strong class imbalance.
- All CV results (mean and standard deviation of the score, plus the parameters) are stored in a single experiments.csv file, which I later use to compare models.

Below are the most important hyperparameters I tuned for each algorithm:

6 Hyperparameter Search Grids

6.1 Logistic Regression

- C (inverse regularization strength): values from a small grid around 1.0 to control model complexity.
- `penalty` (`"l1"` vs `"l2"`): to compare sparse vs dense solutions under the same preprocessing.

6.2 Random Forest

- `n_estimators`: number of trees in the forest (smaller vs larger ensembles).
- `max_depth`: maximum depth of each tree, controlling overfitting vs flexibility.
- `min_samples_leaf`: minimum number of samples per leaf, used as an additional regularization knob.
- `max_features`: fraction or type of features considered at each split (e.g. `"sqrt"` vs fixed fractions).

6.3 XGBoost

- `n_estimators`: number of boosting rounds.
- `max_depth`: depth of individual trees (shallow vs slightly deeper trees).
- `learning_rate`: step size of each boosting update, trading off speed vs stability.
- `subsample`: row subsampling ratio per tree.
- `colsample_bytree`: column subsampling ratio per tree.

7 Combined Approaches

In all experiments I used sklearn `Pipeline`s that bundle preprocessing, the classifier and hyperparameter tuning into a single, reproducible workflow. This ensures that feature dropping, imputation and scaling are refit *only* on the training folds during cross-validation and then applied consistently to the validation folds and the final test set.

Table 2 summarises the concrete combinations of preprocessing, algorithms and tuning that I actually used.

Pipeline	Preprocessing	Cost Function	Tuning setup
Logistic Regression	• Drop features (>50% missing) • Median imputation • Standardisation	`class_weight=balanced`	• 5-fold stratified CV • Grid search C, penalty • Select by mean balanced accuracy
Random Forest	• Drop features (>50% missing) • Median imputation • No scaling	`class_weight=balanced`	• 5-fold stratified CV • Grid search *n_est*, depth • Select by mean balanced accuracy
XGBoost	• Drop features (>50% missing) • Median imputation • No scaling	`scale_pos_weight` = N_{neg}/N_{pos}	• 5-fold stratified CV • Grid search *n_est*, depth • Select by mean balanced accuracy

Table 2: Overview of the combined preprocessing + model + tuning setups.

8 Evaluation

8.1 Evaluation protocol

All models were evaluated using *stratified 5-fold cross-validation* on the training data. Folds were shuffled with a fixed random seed (`random_state = 42`) to ensure reproducibility and to maintain the original class proportions in each fold.

The entire preprocessing chain

$$\text{drop high-missing features} \rightarrow \text{median imputation} \rightarrow \text{(optional) scaling} \rightarrow \text{classifier}$$

was wrapped inside sklearn `Pipeline`s and passed directly to `GridSearchCV`. This has two important consequences:

- For each fold, all preprocessing steps are *fitted only on the training split* and then applied to the validation split.
- Feature dropping, imputing and scaling are never computed on the full dataset at once, which prevents data leakage from the validation folds into the training process.

All cross-validation results (one row per parameter setting and model) were stored in a single `experiments.csv` file and later used for model comparison and analysis.

8.2 Metrics

Because the dataset is strongly imbalanced and the competition itself uses it, the primary metric for all experiments was **balanced accuracy**. Balanced accuracy gives equal weight to the performance on the positive and negative class and is defined as the average of sensitivity and specificity.

`GridSearchCV` was configured with a scorer based on balanced accuracy, and

- model selection *within* each algorithm (e.g. best C and penalty for Logistic Regression), and
- comparison *between* algorithms (Logistic Regression vs Random Forest vs XGBoost)

were both based on the *mean cross-validated balanced accuracy*. For the best model, I additionally inspected class-wise performance (true/false positives and negatives) on the validation folds to confirm that improvements in balanced accuracy corresponded to better detection of the minority (failure) class.

8.3 Results overview

The `experiments.csv` file provides a leaderboard of all evaluated configurations. Table 8.3 summarises the best configuration for each of the three models.

Model	Mean CV bal. acc.	Std. CV	Mean train bal. acc.	Std. train
Logistic Regression	0.9451	0.0074	0.9556	0.0008
Random Forest	0.9442	0.0083	0.9852	0.0007
XGBoost	0.9608	0.0085	0.9876	0.0008

Table 3: Best cross-validated configuration per model (balanced accuracy).

Compared to Logistic Regression and Random Forest, XGBoost achieves the highest validation performance while keeping the gap between training and validation balanced accuracy relatively small (approximately 0.9876 vs. 0.9608). The Random Forest shows a noticeably larger gap between train and validation scores (0.9852 vs. 0.9442), which suggests stronger overfitting despite similar overall validation performance to Logistic Regression. The XGBoost model therefore offers the best trade-off between performance and overfitting among the tested algorithms.

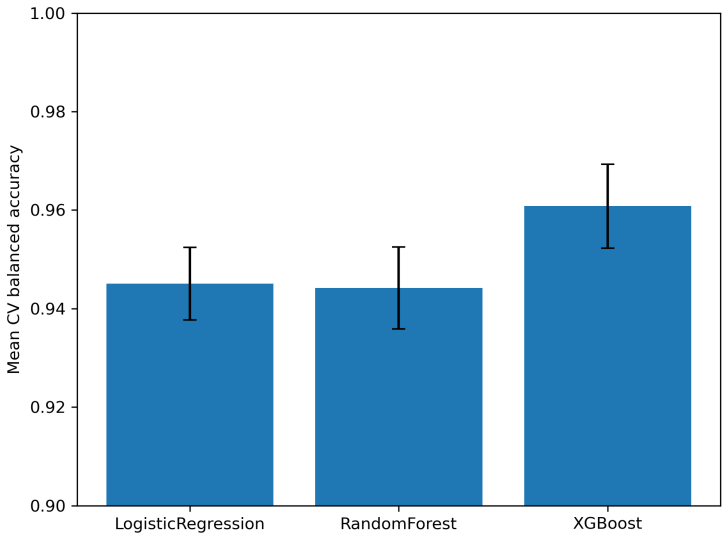

Figure 6: Best cross-validated configuration per model. Bars show the mean balanced accuracy over 5 stratified folds; error bars indicate the standard deviation.

To better understand the XGBoost hyperparameter search, Figure 7 visualises how the balanced accuracy varies with the number of boosting rounds (`n_estimators`) for different tree depths. The selected configuration (`n_estimators` = 300, `max_depth` = 3) lies in a region where performance is consistently high and further increasing the number of trees does not yield systematic improvements.

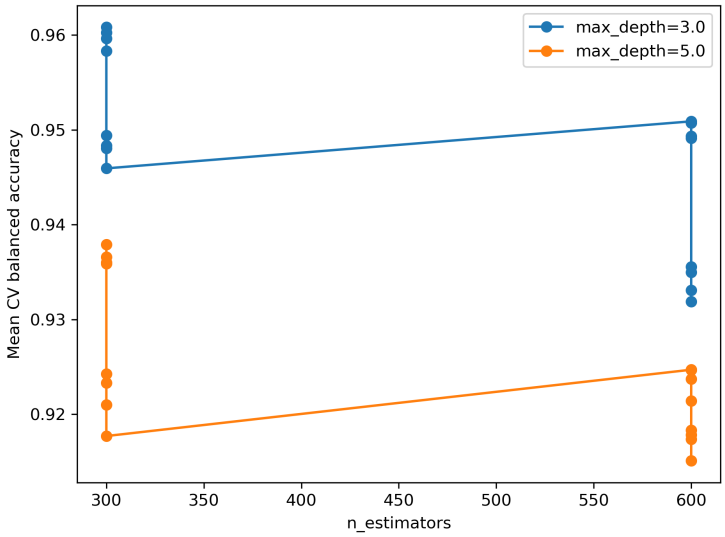

Figure 7: Effect of `n_estimators` and `max_depth` on XGBoost performance. Each line corresponds to a different tree depth; points show individual hyperparameter configurations from the grid search.

Figure 8 compares the mean training and validation balanced accuracy for all evaluated configurations. Random Forest models tend to have a larger gap between train and validation scores, which is consistent with stronger overfitting. In contrast, the best Logistic Regression and XGBoost setups stay closer to the diagonal, indicating a more favourable trade-off between fitting the training data and generalising to unseen examples.

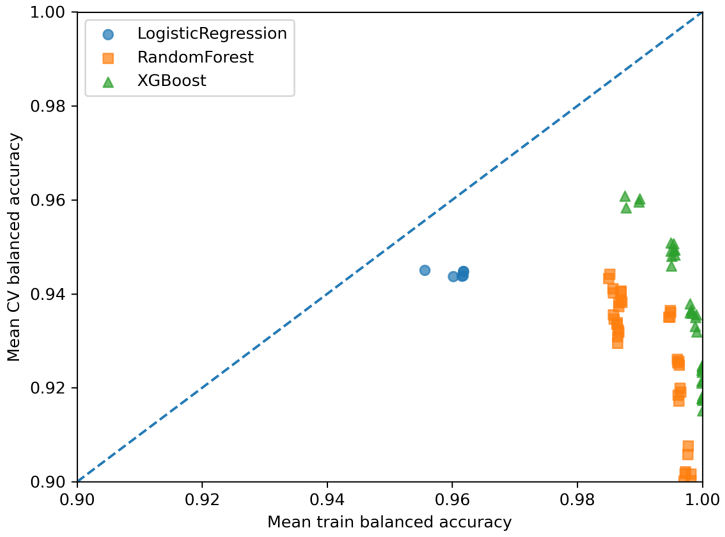

Figure 8: Train vs cross-validated balanced accuracy for all evaluated configurations.

8.4 Threshold Tuning

After selecting this best estimator from `GridSearchCV`, I performed *threshold tuning* on its cross-validated predictions using the same 5-fold splits. Instead of always predicting the positive class when the estimated probability is ≥ 0.5, I evaluated a grid of candidate thresholds and computed the corresponding balanced accuracy for each. The results were saved in `threshold_tuning.csv`. The final decision threshold was chosen as the one that maximised the mean balanced accuracy across folds and was then fixed for generating predictions on the Kaggle test set.

The final XGBoost model, with this tuned threshold, was trained on the *full* training set and used to produce the submission file for Kaggle. The public leaderboard score served as an external sanity check and was satisfactory for me being an improvement over my baseline submission and recent efforts without running grid search, but all model and threshold choices were based solely on cross-validation results.

The curve in Figure 9 shows that the default threshold of 0.5 is not optimal; a slightly shifted threshold leads to a small but consistent improvement in balanced accuracy. The threshold corresponding to the maximum of this curve is used in the final model when generating predictions for the Kaggle test set.

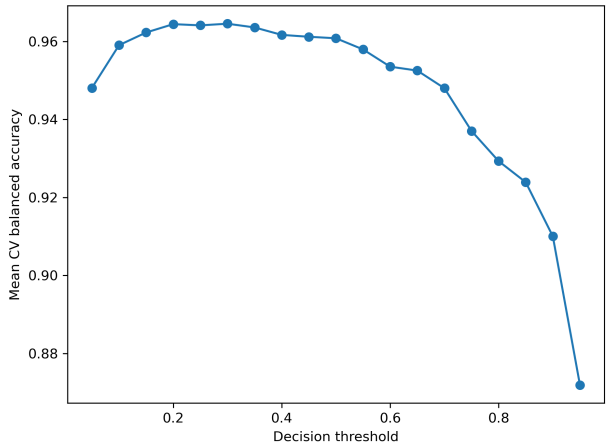

Figure 9: Mean cross-validated balanced accuracy as a function of the decision threshold

8.5 Avoiding overfitting and data leakage

Several design choices were made specifically to avoid common evaluation pitfalls:

- **No test-set peeking**: the Kaggle test labels are unknown and were never used during model development. All decisions (model, hyperparameters, threshold) are based solely on cross-validation on the training data.
- **Pipelines for preprocessing**: dropping high-missing features, imputing and scaling are part of the sklearn `Pipeline` and therefore refitted in each CV training fold. This prevents subtle leakage where statistics (e.g. medians, missingness ratios) are accidentally computed on the full dataset.
- **Stratified folds**: stratification preserves the class imbalance structure in each fold, making the validation performance a more realistic estimate of how the model will behave on unseen data.
- **Moderate hyperparameter grids**: the grids for Logistic Regression, Random Forest and XGBoost were chosen to explore meaningful variations in model capacity and regularisation, but not so large that overfitting to cross-validation noise becomes likely.
- **Fixed random seeds**: all random components (CV splits, tree construction, boosting) use fixed seeds, which makes results reproducible and helps detect inconsistencies during experimentation.

9 Experiences and Best Results

9.1 What worked well

Overall, the combination of a simple but robust preprocessing pipeline and a small set of well-chosen models worked very well for this task. Removing only the most sparse features (more than 50% missing values) and using median imputation turned out to be sufficient; more aggressive feature removal or heavier feature engineering did not seem necessary.

Among the algorithms, XGBoost clearly provided the best results (Section 8). The use of `scale_pos_weight` to reflect the strong class imbalance, together with relatively shallow trees ($max\_depth = 3$) and a moderate number of estimators ($n\_estimators = 300$), produced a model that generalised well in cross-validation. The grid search confirmed that increasing tree depth or the number of trees beyond this point did not systematically improve balanced accuracy (Figure 7).

Threshold tuning on top of the best XGBoost configuration also proved useful. Instead of relying on the default probability threshold of 0.5, I selected the threshold that maximised balanced accuracy on the cross-validation folds (Figure 9). Although the improvement in the metric is relatively small, it is consistent and comes at essentially no extra cost once the model is trained.

9.2 What did not help much

From a practical point of view, the full grid search was also a bit painful. Running the XGBoost and Logistic Regression grids end-to-end on my M1 MacBook Pro took almost a full day, and some parameter combinations simply refused to converge within a reasonable time.

In particular, I originally included `C = 10.0` in the Logistic Regression grid, but removed it after seeing repeated convergence warnings and stuck optimisation. This is one of the reasons why I kept the final grids relatively small instead of trying dozens of additional hyperparameter values.

The experiments showed that Random Forests, while competitive in terms of raw validation performance, tended to overfit more strongly than the other models. As illustrated in Figure 8, many Random Forest configurations achieved very high training balanced accuracy but a noticeably lower value on the validation folds. Even with hyperparameters that controlled tree depth and minimum samples per leaf, the generalisation gap remained larger than for Logistic Regression and XGBoost.

Logistic Regression performed surprisingly well given its simplicity, with a mean cross-validated balanced accuracy only slightly below that of the Random Forest. However, further tuning beyond the tested grid of C values and penalties did not lead to substantial gains, suggesting that the main performance limit is the model's linear decision boundary rather than the regularisation strength.

I also considered more aggressive preprocessing options (e.g. dropping features at lower missingness thresholds or applying transformations to reduce skewness), but preliminary tests indicated no clear improvement in cross-validated performance. Given the assignment's emphasis on reproducibility and avoiding unnecessary complexity, I decided to keep the preprocessing pipeline as simple as possible.

9.3 Best final result

The best overall model is the XGBoost pipeline with the hyperparameters

$$\texttt{n\_estimators} = 300, \quad \texttt{max\_depth} = 3, \quad \texttt{lr} = 0.05, \quad \texttt{subsample} = 1.0, \quad \texttt{colsample\_bytree} = 0.8,$$

combined with the CV-optimised decision threshold described above. This configuration achieved a mean cross-validated balanced accuracy of approximately **0.9608**, which is clearly higher than the best Logistic Regression and Random Forest setups (Table 8.3).

This final model was retrained on the full training set and used to generate predictions for the hidden test set in the Kaggle competition.

The public leaderboard score was consistent with the cross-validation estimate in fact actually improving upon the estimate by almost 1.7% ergo not revealing any obvious overfitting or implementation issues.

As a reference point, my very first submission used a self-written Neural Network implementation in NumPy, without any feature dropping or grid search. Even this simple baseline already achieved a public leaderboard balanced accuracy of roughly 0.95, which initially made me unsure whether it was even worth trying to improve further.

However, In class, we were studying ensembling techniques and professor Lukács asked the class "In which scenarios do white box models work better than Neural Networks?" and a student confidently and quickly answered "they tend to fare better with structured tabular data" and that clicked with me, I decided it was still useful to systematically explore a few classical models (sklearn Logistic Regression, Random Forest and XGBoost) with proper preprocessing and cross-validation, rather than stopping at the first decent result since model complexity should also be tried to minimize if the results remain consistently good.

10 Conclusion

The goal of this project was to build a reliable classifier for APS failures in Scania trucks using the anonymised operational dataset provided in the course competition. The main challenges were the strong class imbalance, the presence of missing values and outliers, and the lack of semantic information about individual features.

In practice I ended up using a fairly simple pipeline: drop the worst features, impute, maybe scale, then try a few models on top. Exploratory data analysis revealed that only a small subset of attributes suffered from extremely high missingness, while most features were reasonably complete. This motivated a preprocessing pipeline that removes only the most sparse features (more than 50% missing values), applies median imputation to the remaining attributes and, where appropriate, standardises the features.

On top of this pipeline I trained and tuned three different models: Logistic Regression, Random Forest and XGBoost. All experiments were carried out using stratified 5-fold cross-validation with balanced accuracy as the primary metric, and all preprocessing steps were included inside sklearn `Pipeline`s to avoid data leakage. The cross-validation results showed that a relatively shallow XGBoost model with class-imbalance handling clearly outperformed the simpler baseline models while still maintaining a reasonable gap between training and validation performance. A small additional gain was achieved by tuning the decision threshold on the validation folds instead of using the default value of 0.5.

The final XGBoost pipeline, combined with the CV-optimised threshold, achieved a mean cross-validated balanced accuracy of approximately 0.96 and produced a Kaggle leaderboard score that was consistent with this estimate.

For me the two main takeaways were: (1) handling class imbalance and picking the right metric is more important than fancy feature engineering here and (2) something I have always found to be the backbone in machine learning tasks in my past experiences: putting everything in a single sklearn pipeline makes your life much easier.

Attribution

Unless otherwise noted, all images in this document were generated through `matplotlib` code written by the author. Their use is purely for educational and non-commercial purposes.

Generative AI has been used for summarising information and improving the readability. All insights, experiences and opinions stated belong to and are written by the author.