Binary classification of machine failures

Project work in Machine Learning & Data Mining

Andrea Terenziani
andrea.terenziani@studio.unibo.it

University of Bologna

Contents

INTRODUCTION

The project

This project is based on the Kaggle challenge of the same name [4].
The data consisted in a set of various industrial devices, each described through attributes like torque or operating temperature, that did or did not suffer some kind of failure.
The task was therefore a binary classification between failed (class 1) and not-failed (class 0), which was done through several different models, from decision trees to a support vector machine.
All estimators (except for the first, as will be shown) were first run using default parameters, to achieve a baseline performance, then tuned with cross validation. All this was done using the `scikit-learn` python package [3], with the cross validation being performed using the `GridSearchCV` and `StratifiedKFold` classes.

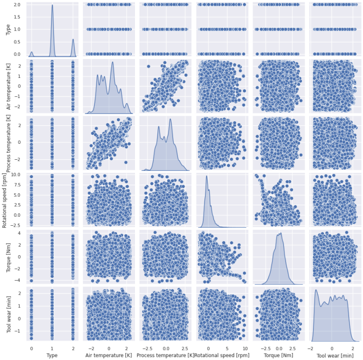
Overview of the data

The attributes of each device are the following:

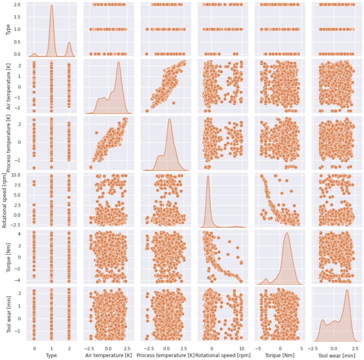
Attribute	Datatype	Description
id	`Int`	Device identifier
Product Id	`String`	Unique Id, combination of the Type attribute and a number identifier
Type	`String`	Type of product/device (possible values: "L","M","H")
Air Temperature	`Float`	Air temperature (Kelvin)
Process Temperature	`Float`	Production process temperature (Kelvin)
Rotational Speed	`Int`	Speed in RPM
Torque	`Float`	Torque in Nm (Newton Meter)
Tool Wear	`Int`	Time unit needed to wear down the product/tool
TWF	`Int`	Tool Wear Failure (binary)
HDF	`Int`	Heat Dissipation Failure (binary)
PWF	`Int`	Power Failure (binary)
OSF	`Int`	Overstrain Failure (binary)
RNF	`Int`	Random Failure (binary)
Machine Failure	`Int`	Failure binary feature (*class attribute*)

The fraction of devices belonging to each class is 98.43% for class 0 and 1.57% for class 1, a ratio of around 62 to 1.

Overview of the data



(a) Class 0 pairplot



(b) Class 1 pairplot

Figure 1: Feature distributions per class (after preprocessing)

Challenges

The main issue with the data is its extreme *imbalance* between majority and minority class (see slide 5). This is addressed by:

1. oversampling the minority class (since undersampling the majority, in this case, would mean removing a large number of rows)
2. choosing an appropriate metric for tuning

Secondly, the dataset was actually generated from another one, leading to a few rows (822 out of around 132k) being clearly invalid and getting removed (their data couldn't be considered "reliable"). Specifically, these had a class inconsistent with their binary failure point (e.g., "Machine Failure" set to 0 but "HDF" set to 1).

Preprocessing and Resampling

The preprocessing simply comprised:

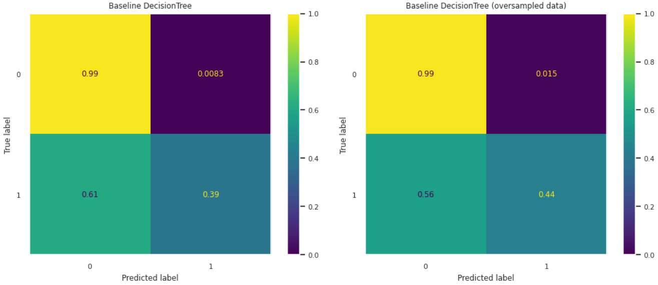
- Removing the aforementioned incoherent rows, together with the TWF, HDF, PWF, OSF, RNF attributes (which would just be a copy of the class attribute)
- Removing the Product ID attribute, not relevant to the classification
- Normalizing all remaining features (except for Type, which is categorical)

The resampling specifically consisted in *oversampling* the minority class to reach a minority-majority ratio of 1 to 10. This was done using the `imbalanced-learn` Python package [2], which offers an implementation of the SMOTE oversampling method [1].

MODELLING

Decision Tree : Baselines(s)

The first model tested was a `DecisionTreeClassifier`, which was also used to fine-tune the method used for subsequent estimators. From figure 2b, we see that using the oversampled data already produced a slight improvement.

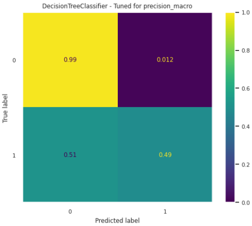


(a) Using the original dataset **(b)** Using oversampled dataset

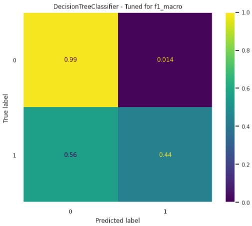
Figure 2: `DecisionTreeClassifier` baselines

Decision Tree Grid search

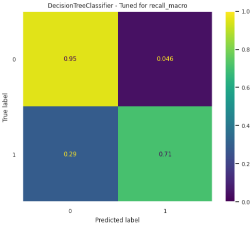
After confirming the choice of using the oversampled data, a grid search for each of the four main scoring metrics shows that the one we should focus on is `recall_macro`, see figure 3b, which will be used for all subsequent tunings.
This is because it "requires" a model that correctly classifies as many classes as possible, giving more importance to the minority.



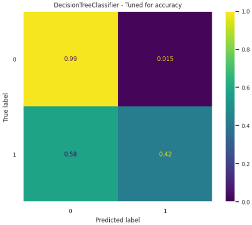
(a) `precision_macro`



(c) `f1_macro`



(b) `recall_macro`

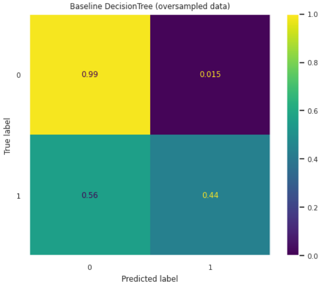


(d) `accuracy`

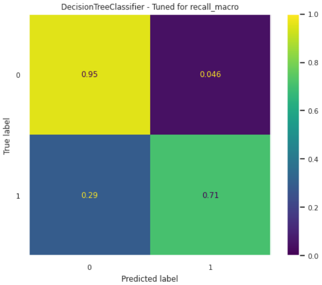
Decision Tree: Final results

parameter	tuned value
max_depth	14
criterion	"gini"
class_weight	"balanced"

metric	result
recall (class 1)	71%
recall_macro	83%



(a) Baseline



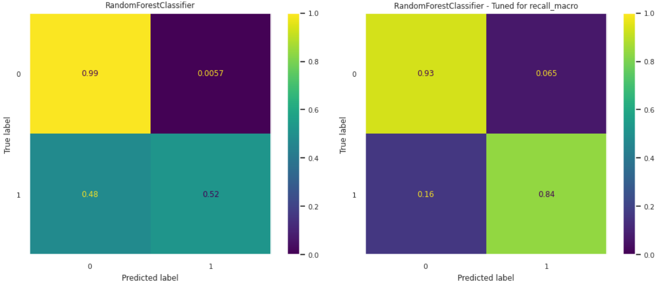
(b) Tuned

Figure 4: Decision Tree results

Random Forest

parameter	tuned value
max_depth	9
criterion	"gini"
class_weight	"balanced"
n_estimators	100

metric	result
recall (class 1)	84%
recall_macro	89%



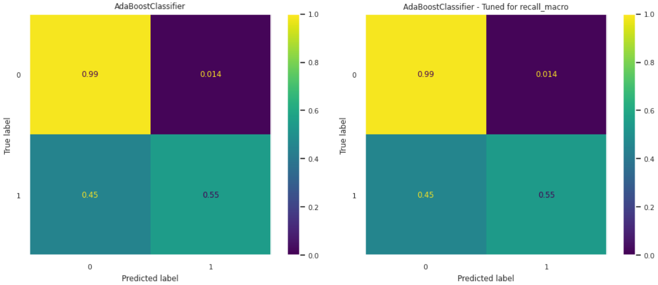
(a) Baseline **(b)** Tuned

Figure 5: Random Forest results

AdaBoost

parameter	tuned value
learning_rate	1.4
n_estimators	100

metric	result
recall (class 1)	55%
recall_macro	77%



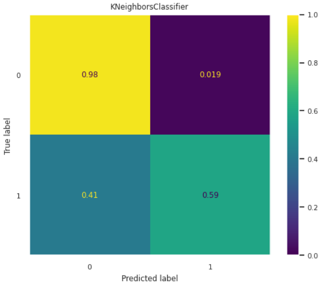
(a) Baseline **(b)** Tuned

Figure 6: AdaBoost results

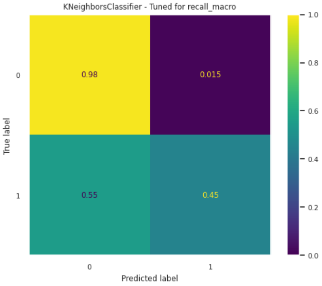
K-Nearest Neighbor

parameter	tuned value
algorithm	"kd_tree"
metric	"manhattan"
leaf_size	40
n_neighbors	1

metric	result
recall (class 1)	45%
recall_macro	72%



(a) Baseline



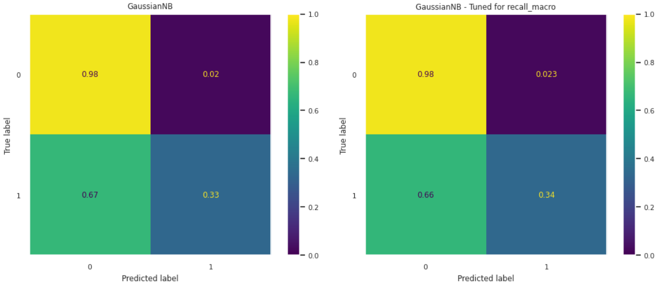
(b) Tuned

Figure 7: K-Nearest Neighbor results

Gaussian Naive Bayes

parameter	tuned value
`var_smoothing`	10^{-8}

metric	result
`recall (class 1)`	34%
`recall_macro`	66%



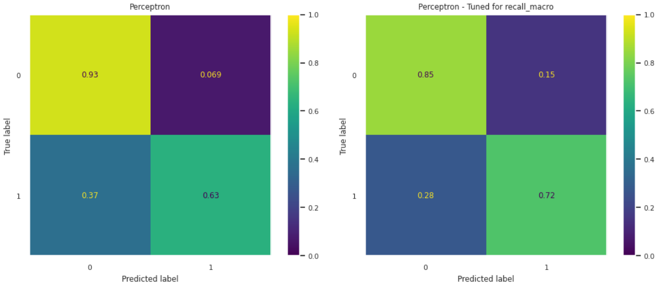
(a) Baseline **(b)** Tuned

Figure 8: Gaussian Naive Bayes results

Perceptron

parameter	tuned value
`early_stopping`	`True`
`penalty`	`None`
`class_weight`	`"balanced"`
`eta0`	`1.0`

metric	result
`recall` (class 1)	72%
`recall_macro`	79%



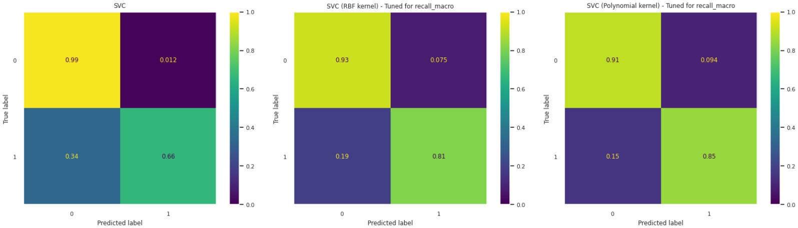
(a) Baseline **(b)** Tuned

Figure 9: Perceptron results

Support Vector Machine

parameter	tuned value
kernel	"poly"
class_weight	"balanced"
shrinking	False
gamma	"auto"
C	100

metric	result
recall (class 1)	85%
recall_macro	88%



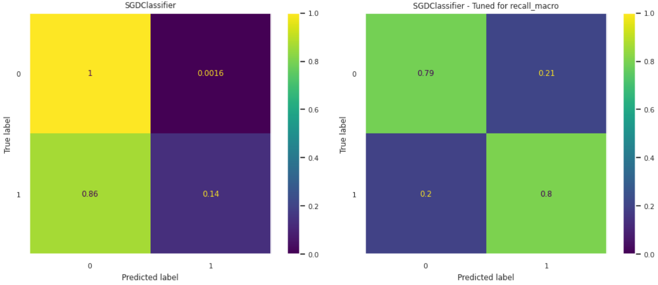
(a) Baseline **(b)** Tuned (RBF) **(c)** Tuned (Polynomial)

Figure 10: Support Vector Machine results

Gradient Descent

parameter	tuned value
class_weight	"balanced"
penalty	"elasticnet"
learning_rate	"invscaling"
loss	"hinge"
alpha	1.0
eta0	0.1

metric	result
recall (class 1)	80%
recall_macro	80%



(a) Baseline **(b)** Tuned

Figure 11: Gradient Descent results

RESULTS

Model performances

After tuning, the models can be ranked as follows:

	Class 1 recall	Recall macro average
Random Forest	84%	**89%**
SVM	**85%**	88%
SGD	80%	80%
Perceptron	72%	79%
Decision Tree	71%	83%
AdaBoost	55%	77%
KNN	45%	72%
Gaussian NB	34%	66%

We therefore see that the Random Forest and SVM models can both be considered the most performant. However, being a few orders of magnitude faster to train, the former may be considered more optimal.

Model performances

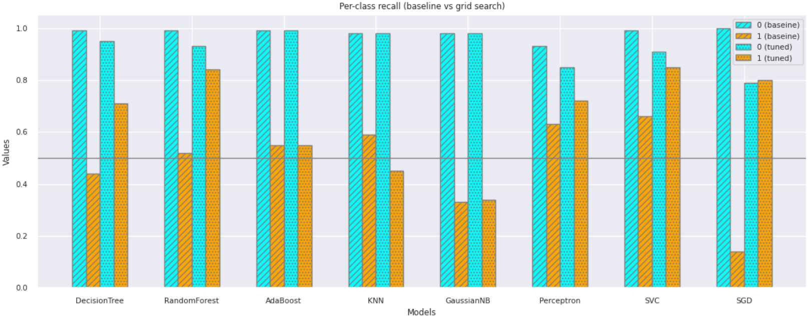


Figure 12: Class recall before and after tuning

Model performances

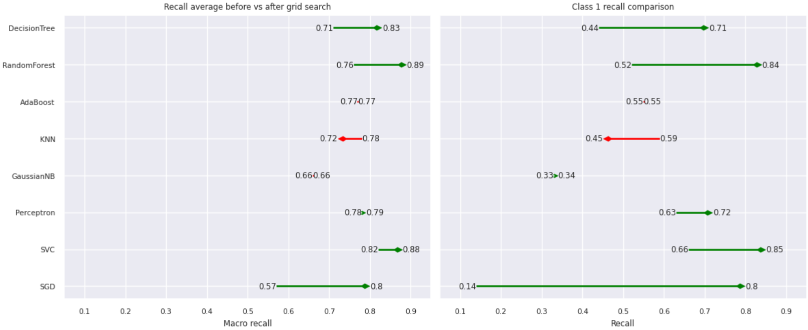


Figure 13: Class 1 recall and `recall_macro` change with tuning

Conclusions

The particularly bad performance of the KNN and Gaussian Naive Bayes classifiers was, to be fair, expected, though for different reasons:

- For the former, the plots in figure 1 show that the two classes are mostly overlapping and to not form well distinct clusters
- For the latter, its basic assumption of conditional independence between the attributes is clearly faulty in this context, since (for example) the rotational speed of a machine obviously influences its operating temperature

Furthermore, figure 13 shows that these two models, and AdaBoost, either didn't improve or became worse. This may have two reasons:

- none have a "class weight" parameter, and thus couldn't be tuned to "pay more attention" to one class than the other
- cross-validation is performed on a *slice* of the training set (unlike the default model, which used all of it). This is especially damning for the KNN classifier because it doesn't really perform any analysis of the training data, instead simply using it "as-is".

Conclusions

The main focus of this project ended up being *how to deal with severely imbalanced data*, more than a classification task itself. This, at least for the given dataset, was achieved in three main ways:

1. using a per-class weighing of the datapoints, when supported
2. performing dataset resampling during preprocessing to make the minority class easier to learn
3. targeting the `recall_macro` scoring metric when tuning instead of accuracy or precision
    - this metric is an *unweighted* average of the recalls for each class, making thus sure that the result doesn't de-facto only account for the majority datapoints
    - the majority class being so dominant, it is acceptable to be slightly less accurate in classifying it if it means being way more sensitive to the minority

References I

[1] Kevin W. Bowyer, Nitesh V. Chawla, Lawrence O. Hall, and W. Philip Kegelmeyer.
SMOTE: synthetic minority over-sampling technique.
CoRR, abs/1106.1813, 2011.
URL: http://arxiv.org/abs/1106.1813, arXiv:1106.1813.

[2] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas.
Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning.
Journal of Machine Learning Research, 18(17):1–5, 2017.
URL: http://jmlr.org/papers/v18/16-365.html.

References II

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay.
Scikit-learn: Machine learning in Python.
Journal of Machine Learning Research, 12:2825–2830, 2011.
URL: https://scikit-learn.org/stable/.

[4] Ashley Chow Walter Reade.
Binary classification of machine failures, 2023.
URL: https://kaggle.com/competitions/playground-series-s3e17.