Engineering Different ML Models

Ziyuan Li 12211225

This report will introduce different machine learning models and compare their performance on a given dataset. A comprehensive analysis will be conducted to evaluate the impact of different engineering techniques.

November 21, 2024

Contents

1. Introduction

1.1. Background

Over the past 11 weeks, we have learnt about different machine learning models including linear regression, logistic regression, perceptron and MLP. In this report, we will compare the performance of these models on a given dataset (AI4I 2020 Predictive Maintenance Dataset from UCI machine learning library). We will also evaluate the impact of different engineering techniques on the performance of these models.

1.2. Dataset

The dataset consists of 10,000 data points, each described by 14 columns.

- Features

  Each row includes a unique identifier and product ID that classify products into three **quality categories** — Low (50% of products), Medium (30%), and High (20%). Key process parameters such as **air and process temperatures** (normalized to deviations of 2 K and 1 K around 300 K, respectively), **rotational speed, torque, and tool wear** are included. Rotational speed is based on a power of 2860 W, with added noise, while torque values are normally distributed around 40 Nm with no negative values. Tool wear depends on product quality, with H/M/L variants contributing 5, 3, and 2 additional minutes, respectively.

  In practice, **a sum of 5 features** are used in the predictions. These 5 features have mixed data types. For example, the **product ID** is a categorical feature, while the **air temperature** is a continuous numerical feature, and the **rotational speed** is a continuous numerical feature.

- Target

  The dataset includes a binary **"machine failure" label**, which indicates failures resulting from five independent modes: tool wear failure, heat dissipation failure, power failure, overstrain failure, and random failure. Specific failure criteria include conditions like excessive tool wear, insufficient heat dissipation, power deviations, overstrain thresholds varying by product quality, and random failures at a 0.1% likelihood.

  We will **only focus on predicting the machine failure label** instead of classifying the type of failure.

1.3. Methodology

We will first preprocess the dataset by normalizing the features and splitting it into training and testing sets. We will then train the models on the training set and evaluate their performance on the testing set. And then, We will compare the performance of the models using metrics such as accuracy, precision, recall, and F1 score.

We will evaluate the impact of different engineering techniques such as normalization, feature selection, and hyperparameter tuning on the performance of these models.

2. Code Analysis

2.1. Code Structure

```
.
├── client.py
├── data
│   ├── X.csv
│   └── y.csv
├── docs
│   └── report.typ
├── figures
├── models
│   ├── LinearRegression.py
│   ├── LogisticRegression.py
│   ├── MLP.py
│   ├── Perceptron.py
│   └── modules.py
└── utils
    ├── evaluation.py
    ├── feature_engineering.py
    └── preprocessing.py

6 directories, 12 files
```

We organize the project Structure in the manner shown by the tree above.

- `client.py` is the main script that runs the experiments and generates the results.
- `data` directory contains the input data files `X.csv` and `y.csv`.
- `docs` directory contains the report file.
- `figures` directory contains the figures generated during the experiments.
- `models` directory contains the implementation of the machine learning models.
- `utils` directory contains utility functions for preprocessing, feature engineering, and evaluation.

The models are taken from the assignment solutions before and are modified to fit the dataset. The `modules.py` file contains the shared functions used by the models.

2.2. Data preprocessing

The data preprocessing steps are implemented in the `preprocessing.py` file.

It contains the following functions:

- `load_and_clean_data`: The core function of data loading. This function loads the data from the ucimlrepo module, and drops the columns that are not used in the prediction (e.g. product ID, unique identifier, different failure categories). It also converts the categorical columns to one-hot encoding. If given the parameter `normalize=True`, it normalizes the numerical columns using the StandardScaler.
- `convert_to_numpy`: This function converts the pandas DataFrame to numpy arrays.
- `undersample_data`: This function performs undersampling on the dataset to balance the classes. The influence of such an attempt to balance the dataset is evaluated in the report.

- `enhance_data` : This function performs oversampling using the SMOTE technique to balance the classes. This technique is "symmetric" to the undersampling technique, and its impact is also evaluated in the report.
- `undersample_data_numpy` : This function performs undersampling on the dataset represented as NumPy arrays. It is used in the Perceptron model, which requires NumPy arrays as input.
- `map_y` : This function maps the target labels to 1 and 2 to be compatible with the Perceptron model.
- `enhance_undersamp` : wraps up undersampling and oversampling.

2.3. Feature Evaluation

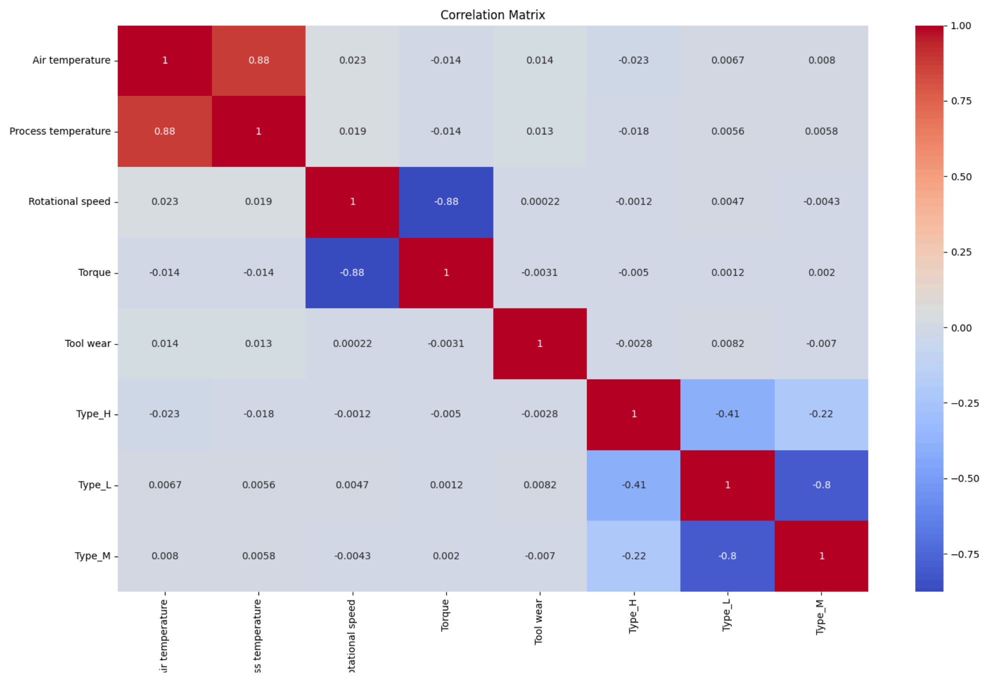
The evaluation of features are implemented in the `feature_engineering.py` file.

It contains the functions below:

- `drop_one_type` : This function drops the `Type_M` column from the dataset.
  - It is designed for the linear regression model. When encoding the categorical feature `Type` , three resulting columns can be linearly dependent. Dropping one of them can prevent multicollinearity.
- `evaluate_correlation` : This function evaluates the correlation between features by calculating a correlation matrix and plotting it.
- `drop_air_temperature` : This function drops the `Air temperature` column from the dataset. As we will see in the `evaluate_correlation` function, this column has a pretty high correlation with the other feature, 'Process temperature'. We will discuss the impact of dropping it later.
- `evaluate_significance` : This function evaluates the significance of the features by calculating the correlation directly between each feature and the target variable. It also plots the correlation values.

- **Evaluation of features**

First we evaluate the correlation between features. The correlation matrix is shown below:



	Air temperature	Process temperature	Rotational speed	Torque	Tool wear	Type_H	Type_L	Type_M
Air temperature	1	0.88	0.023	-0.014	0.014	-0.023	0.0067	0.008
Process temperature	0.88	1	0.019	-0.014	0.013	-0.018	0.0056	0.0058
Rotational speed	0.023	0.019	1	-0.88	0.00022	-0.0012	0.0047	-0.0043
Torque	-0.014	-0.014	-0.88	1	-0.0031	-0.005	0.0012	0.002
Tool wear	0.014	0.013	0.00022	-0.0031	1	-0.0028	0.0082	-0.007
Type_H	-0.023	-0.018	-0.0012	-0.005	-0.0028	1	-0.41	-0.22
Type_L	0.0067	0.0056	0.0047	0.0012	0.0082	-0.41	1	-0.8
Type_M	0.008	0.0058	-0.0043	0.002	-0.007	-0.22	-0.8	1

Figure 1: Correlation Matrix

From the correlation matrix, we can see that the `Air temperature` and `Process temperature` have a high correlation(0.876). This is expected as they are both temperature-related features. Also, `Rotational speed` and `torque` have a strong negative correlation (-0.875), consistent with the physical inverse relationship between speed and torque in mechanical systems. We can try to drop one of the correlated features to reduce multicollinearity.

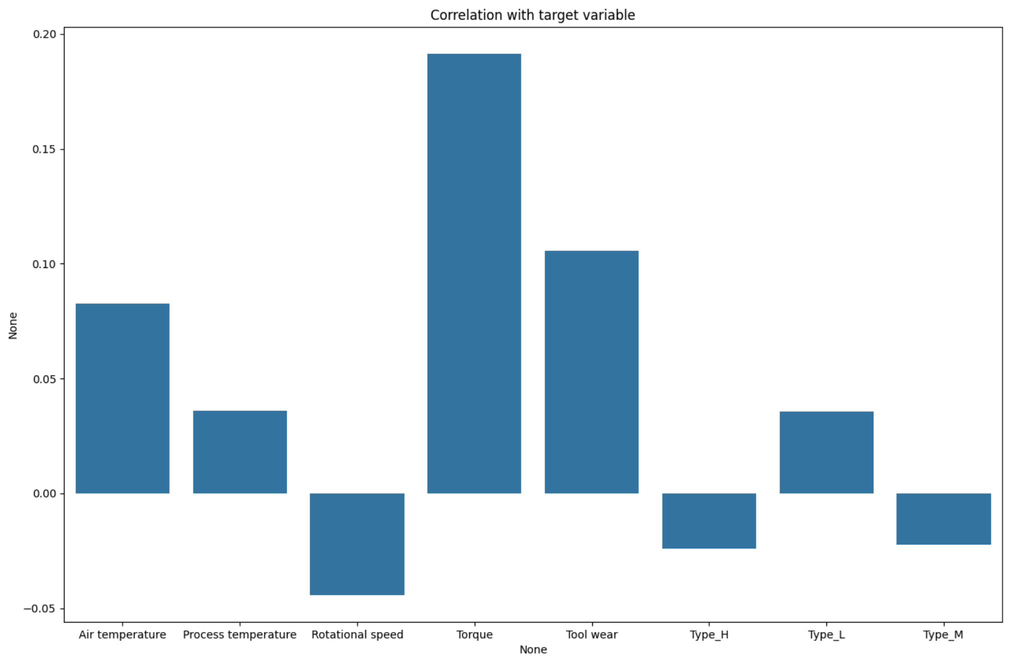


Figure 2: Correlation with Target

```
Air temperature        0.082556
Process temperature    0.035946
Rotational speed      -0.044188
Torque                 0.191321
Tool wear              0.105448
Type_H                -0.023916
Type_L                 0.035643
Type_M                -0.022432
dtype: float64
```

We also do a direct correlation evaluation between the features and the target variable. The results show that `Torque` has the highest correlation with the target variable. However those values are not very high, which means that simply calculating statistic correlation is not enough to determine the target value. This is where the machine learning models come in.

2.4. Model Implementation

The models are implemented in the `models` directory. Each model is implemented in a separate file. They are taken directly from former assignments and modified to fit the dataset. Each model file mainly contains the corresponding class. Because the linear regression model is intrinsically unsuitable to do binary classification, we added a threshold to the prediction to make it compatible with the binary classification task.

3. Experiments and Discussion

3.1. Baseline Performance

Controlling other techniques (normalization, oversampling, etc.), we first evaluate the performance of the models. We try to tune the best hyperparameters for each of the models. And to make the metrics look better and differ from each other, we oversample the training data and undersample the testing data. Training data is normalized.

The results are shown below:

```
Linear Regression Normalized: True
Accuracy: 0.7857142857142857
Recall: 1.0
Precision: 0.7
F1 Score: 0.8235294117647058

Perceptron - Normalized: True
Accuracy: 0.8928571428571429
Recall: 1.0
Precision: 0.8235294117647058
F1 Score: 0.9032258064516129

Logistic Regression
Accuracy: 0.8928571428571429
Recall: 0.9285714285714286
Precision: 0.8666666666666667
F1 Score: 0.896551724137931

Training MBGD: 100%|| 500/500 [00:29<00:00, 16.71it/s]
Accuracy: 0.8928571428571429
Recall: 1.0
Precision: 0.8235294117647058
F1 Score: 0.9032258064516129
```

The results show that the Perceptron model and the MLP performs the best among the models. The linear regression model performs the worst. The logistic regression model has medium performance.

The ranking of the models can be attributed to their ability to handle classification tasks and the nature of the dataset. Perceptron and MLP are inherently designed for classification and can model non-linear decision boundaries, making them more effective for this task, especially after hyperparameter tuning. Logistic Regression, while also designed for classification, is a simpler model and may have limited capacity to capture complex patterns compared to MLP and Perceptron. Linear Regression, on the other hand, is inherently a regression model and struggles with the discrete nature of the target variable, leading to poorer performance.

The reason why the logistic regression have the same scores on four metrics with MLP is that our testing dataset is very small. The test split is 30% of the original dataset, which has 3000 samples and only 14 of them are positive. If we take equal number of positive and negative samples into the testing dataset, there will still be only 28 of them. This small number of samples make there be a great possibility for the models to perform completely equally when evaluated by the metrics.

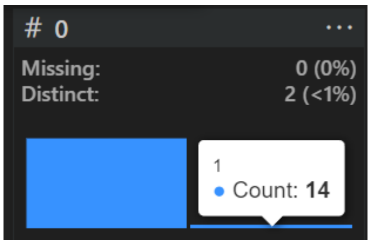


Figure 3: very few positive values in the original test dataset

3.2. Mixed Data types & Normalization

The dataset contains mixed data types, including categorical and numerical features. In order to correctly feed the data in the models, we need to preprocess the data. We will use one-hot encoding for the categorical features and normalization for the numerical features.

In practice, we converted the categorical feature `Type` into three columns `Type_H`, `Type_M`, and `Type_L` using one-hot encoding. We used the pd.get_dummies function to achieve the one-hot encoding, then used the `.astype()` method that comes with the numpy array to convert the numerical values and the one-hot encoded values all into `float64`.

- **Discussion: The use of Normalization**

  We will discuss the impact of normalization on the performance of different models. Here we use a standardized setting where the training data is oversampled and the testing data is undersampled. Other hyperparameters are tuned within normal range where models all converge(at least as their loss graphs show)

  - Linear regression

  Here we use a model `lr = LinearRegression(n_feature=7, lr=1e-6, epochs=500)`

  The Result with normalization is

```
Linear Regression Normalized: True
Accuracy: 0.7857142857142857
Recall: 1.0
Precision: 0.7
F1 Score: 0.8235294117647058
```

  The Result without normalization is

```
Linear Regression
Accuracy: 0.5357142857142857
Recall: 0.2857142857142857
Precision: 0.5714285714285714
F1 Score: 0.380952380952380
```

  For the same hyperparameters, the model performs significantly better with normalization. This is because the features are on different scales, and normalization helps the model converge faster and achieve better performance.

  However when we use a smaller learning rate, the model without normalization can also converge, and the F1 score is also not bad. However, the 2 main drawbacks are:

  1. We cannot use too high learning rate. A rate greater than about 1e-3 could cause overflow.
  2. The final result is not as good as the normalized one when converged.

```
Linear Regression
Accuracy: 0.7142857142857143
Recall: 1.0
Precision: 0.6363636363636364
F1 Score: 0.7777777777777778
```

We have also tried out this for the Perceptron `per = Perceptron(n_feature=8, lr=1e-7, epochs=500)`. This time the percaptron performs better with the same hyperparams.

```
Perceptron - normalized
Accuracy: 0.8214285714285714
Recall: 0.7857142857142857
Precision: 0.8461538461538461
F1 Score: 0.8148148148148148

Perceptron  - Normalized: False
Accuracy: 0.8214285714285714
Recall: 1.0
Precision: 0.7368421052631579
F1 Score: 0.8484848484848484
```

However when we tune the hyperparameters to get the optimal hyperparameters for the normalized case, the performance is significantly better.

```
Perceptron  - Normalized: True
Accuracy: 0.8928571428571429
Recall: 1.0
Precision: 0.8235294117647058
F1 Score: 0.9032258064516129
```

Summary: Normalization is crucial for the linear regression model to perform well. It enables the models to converge better and allows a wider range of hyperparams. Often models can't tolerate too high learning rates if the data is not normalized.

3.3. Data Imbalance

The dataset is imbalanced, with the majority of the data points belonging to the negative class.

All baseline models, ranging from linear regression to MLP, fail to capture the minority class. Even when pushed to extreme hyperparameters, the MLP still fail to capture the minority classes when being fed the dataset with imbalance and testing on the imbalanced dataset.

```
layers = [
Linear(input_size=8, output_size=64),
ReLU(),
Linear(input_size=64, output_size=128),
ReLU(),
Linear(input_size=128, output_size=256),
ReLU(),
Linear(input_size=256, output_size=1),
Sigmoid()
]
```

```
Training MBGD: 100% | 500/500 [01:50<00:00,  4.50it/s]
Accuracy: 0.9953333333333333
Recall: 0.0
```

```
Precision: 0
F1 Score: 0
```

If there are too many negative samples, the model will tend to predict negative for all samples to get a high accuracy (accuracy means correct prediction out of all predictions).

To solve this problem, we can use techniques like oversampling and undersampling. We have implemented these techniques in the `preprocessing.py` file. Oversampling means generating more samples for the minority class, while undersampling means reducing the number of samples for the majority class.

My considerations about whether we should use oversampling or undersampling are as follows:

- **We should not do oversampling with test data.** If the data is imbalanced by the nature, for example machine failures are often rare, then we should not generate those fake machine failures when facing real-life scenarios. Instead, **the taking less samples from the majority class is more reasonable, corresponding to doing a filter to the ordinary cases in real life.**
- For training data, we can try to use both methods, they are both reasonable. **Oversampling should be considered more when we have a small dataset**, because we want to maximize the use of every bit of information. **However, oversampling also introduced more risk of overfitting. Therefore, we should be careful when using it.**

We will use the perceptron on the dataset with oversampling and undersampling to see the difference, for its high accuracy and simplicity to train (when compare with the MLP).

1. Without any oversampling or undersampling

   ```
   Perceptron - Normalized: True
   Accuracy: 0.9913333333333333
   Recall: 0.07142857142857142
   Precision: 0.07142857142857142
   F1 Score: 0.07142857142857142
   ```

   The accuracy is very high but the recall and precision are poor. It is classifying nearly all test data as the negative label, which is useless in practice and won't help us to predict the machine failures.

2. With oversampling in train data, no undersampling in test data

   ```
   Perceptron - Normalized: True
   Accuracy: 0.837
   Recall: 1.0
   Precision: 0.027833001988071572
   F1 Score: 0.05415860735009672
   ```

   The precision is extremely low, indicating that most of the instances predicted as positive by the model are actually false positives. This suggests that the model is overly aggressive in predicting the positive class, likely predicting many negatives as positives.

3. With undersampling in test data, and oversampling in train data

   ```
   Perceptron - Normalized: True
   Accuracy: 0.8928571428571429
   ```

```
Recall: 1.0
Precision: 0.8235294117647058
F1 Score: 0.9032258064516129
```

When the test and training data are both balanced, the model performs the best. The recall and precision are both high, and the F1 score is also high. This can be considered how this model *should* perform.

However this is still less-than legitimate because we have edited the test data.

4. Both undersampling

```
Perceptron - Normalized: True
Accuracy: 0.7857142857142857
Recall: 0.8571428571428571
Precision: 0.75
F1 Score: 0.7999999999999999
```

The overall performance is slightly worse than the balanced case, but still better than the unbalanced case. We might explain this by the fact that we are using less data in training.

5. Adjusting the threshold

```
log = LogisticRegression(n_feature=8, threshold=0.59,learning_rate=1e-3, epochs=500)
```

```
Logistic Regression
Accuracy: 0.98
Recall: 0.07142857142857142
Precision: 0.020833333333333332
F1 Score: 0.03225806451612903
```

We can adjust the threshold of classifying the logit to pull the recall and precision closer. However, the F1 score is still very low. The logistic model will be classifying **wrong** samples as positive.

Summary

The data imbalance is a significant challenge in this dataset. The models tend to predict the majority class, leading to poor performance on the minority class. Techniques like oversampling and undersampling can help balance the dataset and improve the model's performance. However, in our case we cannot achieve high F1 scores unless we also do a filtering to the testing data, which is less than legitimate. Adjusting the threshold can also help balance the recall and precision, but it may not be sufficient to address the data imbalance issue.

3.4. Feature selection

We have evaluated the correlation between features and the target variable. We can also evaluate the importance of features using techniques like feature selection. Now we will try to drop some features and see how it affects the model's performance.

We will use the Perceptron model for this experiment. We will drop the `Air temperature` feature, which has a high correlation with the `Process temperature` feature. We will also try to drop the `Torque` feature.

Here we undersample the test data and oversample the training data. The training data is normalized.

```
per = Perceptron(n_feature=6, lr=1e-4, epochs=500)
```

```
1. No dropping
Perceptron - Normalized: True
Accuracy: 0.75
Recall: 0.6428571428571429
Precision: 0.8181818181818182
F1 Score: 0.7200000000000001

2. Dropping Air temperature
Perceptron - Normalized: True
Accuracy: 0.8928571428571429
Recall: 1.0
Precision: 0.8235294117647058
F1 Score: 0.9032258064516129

3. Dropping both Air temperature and Torque
Perceptron - Normalized: True
Accuracy: 0.8571428571428571
Recall: 0.8571428571428571
Precision: 0.8571428571428571
F1 Score: 0.8571428571428571
```

The results show that dropping the `Air temperature` feature improves the model's performance. This is consistent with the correlation matrix, which shows that `Air temperature` and `Process temperature` have a high correlation. Dropping the `Torque` feature, however, does not improve the model's performance. This suggests that the `Torque` feature is important for the model's predictions.

```
No processing on test sets
Perceptron - Normalized: True
Accuracy: 0.921
Recall: 0.8571428571428571
Precision: 0.048582995951417005
F1 Score: 0.09195402298850576
```

Such an approach can help in resolving the scarce positive label problem. As we can see here the recall rates have gone up after performing the drop_air_temperature and drop_torque method. This is because the model is now more focused on the features that are more important to the target variable.

Actually the overall performance is close to the case where we only do oversampling in training data. This is still somehow meaningful because it decreases the data needed for training.

```
Training MBGD: 100%|| 500/500 [00:36<00:00, 13.74it/s]
Accuracy: 0.9376666666666666
Recall: 0.6428571428571429
Precision: 0.04712041884816754
F1 Score: 0.08780487804878048
```

We also tried using an MLP on dataset with dropped Air temperature and Torque without test set undersampling. The result is not as good as the perceptron model. The reason might be that the MLP is more complex and may need more data to train.

4. Summary

The evaluation of the baseline performance reveals varying capabilities of the models to handle the classification task under controlled settings.

Among the models, the Perceptron and Multi-Layer Perceptron exhibited the best performance, achieving the highest F1 scores and balanced accuracy and recall, reflecting their ability to model non-linear decision boundaries. Logistic regression followed with medium performance, achieving an F1 score of 0.897, attributed to its simplicity and linear decision boundaries. Linear regression, which is inherently a regression model rather than a classifier, performed the worst, highlighting its unsuitability for discrete classification tasks.

We should do conversions of the data types and normalization before feeding the data into the models. Normalization enables the models to converge better and allows a wider range of hyperparameters. Often models can't tolerate too high learning rates if the data is not normalized.

As for feature engineering, here we first evaluated the correlations between features and between features and target, to determine which features we should choose. Dropping some features with high correlation can improve the model's performance or have little performance loss. This can be helpful because it decreases the computation needed.

The data imbalance is a significant challenge in this dataset. The models tend to predict the majority class, leading to poor performance on the minority class. Techniques like oversampling and undersampling can help balance the dataset and improve the model's performance. However, in our case we cannot achieve high F1 scores unless we also do a filtering to the testing data, which is less than legitimate. We also tried adjusting the threshold for the logistic model but it is not sufficient to address the data imbalance issue.

5. Future Work

This dataset about machine failure is not typical classification problem for the sparcity of the positive targets.

There is actually a category of machine learning problem that deals with this type of data, which is called anomaly detection. In anomaly detection, we are not trying to classify the data into different categories, but to detect the anomalies in the data. This is more suitable for the machine failure prediction task, as machine failures are often rare events.

Some key ideas:
- The minority class is treated as anomalous behavior relative to the majority class.
- Train the model only on the majority class to learn its typical characteristics.
- During inference, the minority class is identified as deviations or anomalies from the majority class pattern.

One common algorithm is **autoencoders**. It is an unsupervised learning algorithm that learns a compressed representation of the data and tries to reconstruct the input data from the compressed representation. The reconstruction error, i.e. the difference between the input and the reconstructed data, can be used to detect anomalies. For example, we "reconstruct" the normal data and then compare the reconstruction error of the test data with the normal data. If the error is too large, we can classify it as an anomaly.

However considering that the failure can be directly explained by **five categories,**[1] we can also try **predicting the type of failure directly**, maybe using a decision tree. Or, we can use an **ensemble of models to predict the type of failure**, where each model is trained to predict a specific type of failure, and then summing up the predictions to get the final prediction. Constructing an **ensemble of models** is also another common approach to improve the performance of machine learning models.

[1]

The machine failure consists of five independent failure modes
tool wear failure (TWF): the tool will be replaced of fail at a randomly selected tool wear time between 200 â€" 240 mins (120 times in our dataset). At this point in time, the tool is replaced 69 times, and fails 51 times (randomly assigned).
heat dissipation failure (HDF): heat dissipation causes a process failure, if the difference between air- and process temperature is below 8.6 K and the toolâ€™s rotational speed is below 1380 rpm. This is the case for 115 data points.
power failure (PWF): the product of torque and rotational speed (in rad/s) equals the power required for the process. If this power is below 3500 W or above 9000 W, the process fails, which is the case 95 times in our dataset.
overstrain failure (OSF): if the product of tool wear and torque exceeds 11,000 minNm for the L product variant (12,000 M, 13,000 H), the process fails due to overstrain. This is true for 98 datapoints.
random failures (RNF): each process has a chance of 0,1 % to fail regardless of its process parameters. This is the case for only 5 datapoints, less than could be expected for 10,000 datapoints in our dataset.