AI4I Binary Classification Prediction

Author: Ying Yiwen Number: 12210159

Abstract

This report applies machine learning to the AI4I 2020 Predictive Maintenance Dataset for predicting machine failures. Models including linear regression, perceptrons, logistic regression, and multi-layer perceptrons were evaluated. The MLP outperformed others with an F1-score of 0.841, effectively handling complex patterns and minimizing false negatives. These results emphasize the potential of machine learning in improving industrial predictive maintenance.

Contents

1 Introduction

1.1 Dataset

Predictive maintenance is a critical aspect of modern industrial operations, enabling proactive measures to reduce downtime, optimize maintenance schedules, and prevent costly machine failures. This report explores the application of supervised machine learning models on the AI4I 2020 Predictive Maintenance Dataset, a comprehensive dataset containing 10,000 data points with 14 features. Each data point describes a specific machine state and includes a binary target variable, machine failure, which indicates whether a failure occurred.

The dataset provides rich information about machine performance, incorporating various operational parameters such as **air temperature, process temperature, rotational speed, torque, and tool wear**. Additionally, it simulates five distinct failure modes:

- Tool Wear Failure (**TWF**): Triggered by tool wear exceeding specific thresholds.
- Heat Dissipation Failure (**HDF**): Occurs when the temperature difference between air and process is too small and rotational speed drops below a critical level.
- Power Failure (**PWF**): Happens when power requirements deviate significantly from acceptable ranges.
- Overstrain Failure (**OSF**): Caused by excessive wear and torque exceeding thresholds specific to product quality variants.
- Random Failures (**RNF**): Randomized rare events independent of machine parameters.

If any of these failure modes are true, the machine failure label is set to 1, creating a multi-faceted prediction problem where the specific failure mode is unknown to the learning model.

1.2 Object

The primary goal of this project is to develop and evaluate machine learning models capable of predicting machine failures based on the dataset's features. By doing so, the project simulates a real-world predictive maintenance task, allowing for a deep exploration of the following key aspects:

(i) **Data Preprocessing**: Cleaning and preparing the dataset for analysis, addressing potential challenges like feature scaling and noise.

(ii) **Feature Engineering**: Identifying critical features related to machine failure and analyzing their influence on prediction accuracy.

(iii) **Model Implementation**: Applying diverse machine learning models, including logistic regression, perceptrons, and multilayer perceptrons (MLP), to understand their effectiveness in capturing failure patterns.

(iv) **Model Evaluation and Comparison**: Utilizing metrics like accuracy, precision, recall, F1-score, and ROC-AUC to assess model performance and their suitability for predictive maintenance scenarios.

(v) **Hyperparameter Optimization**: Refining models for enhanced performance through systematic tuning of hyperparameters.

1.3 Significance

In industrial settings, unexpected machine failures can lead to significant costs, including production downtime, damaged equipment, and increased maintenance expenses. Predictive maintenance addresses this challenge by forecasting potential failures before they occur, enabling proactive interventions and resource optimization. However, accurately predicting failures in complex systems is not trivial, especially when multiple failure modes, dynamic operating conditions, and interdependent parameters are involved.

This scenario exemplifies the need for predictive approaches, as traditional methods relying on fixed thresholds or manual inspections are often insufficient for capturing subtle and nonlinear patterns leading to failures. Machine learning offers a robust alternative by leveraging data-driven insights to uncover complex relationships among operational parameters, enabling more precise and timely predictions.

The **AI4I 2020 Predictive Maintenance Dataset** serves as an ideal example of this challenge, encompassing:

- Multiple failure modes: Five independent failure mechanisms, each driven by different parameter thresholds or random occurrences, make the problem highly complex and multifaceted.
- Nonlinear dependencies: Interactions between features like torque, rotational speed, and temperatures demand models capable of capturing intricate patterns beyond simple rules.
- Class imbalance and rare events: Scenarios like random failures and specific failure thresholds necessitate models that handle skewed distributions and detect rare occurrences.

In such cases, machine learning becomes a necessity, providing the capability to:

(i) Adapt to variability: Learn from operational data across different machines, environments, and conditions, offering flexibility beyond rule-based systems.

(ii) Enhance efficiency: Automate failure detection, reducing reliance on manual efforts and improving response times for maintenance actions.

By applying machine learning techniques to this dataset, this project demonstrates the potential of predictive algorithms in addressing real-world industrial challenges. It highlights the value of data-driven models in reducing costs, improving reliability, and driving smarter decision-making in predictive maintenance applications.

2 Principle

2.1 Data Preprocessing

Data preprocessing is a critical step in ensuring the dataset is clean, balanced, and suitable for training machine learning models. This section outlines the techniques used to prepare the dataset for analysis.

Algorithm 1 Dataset Loading and Preprocessing

```
Input: file_path: Path to the dataset file.
Output: Processed training and testing datasets: $X_{train}$, $Y_{train}$, $X_{test}$, $Y_{test}$.
 1: Step 1: load dataset and remove rows with missing values.
 2: Step 2: Drop unnecessary columns: UDI, Product ID, TWF, HDF, PWF, OSF, RNF.
 3: Step 3: Replace ‘Type‘ column values: L -> 11, M -> 12, H -> 13.
 4: Step 4: Add new features:
 5:     Air temperature [K] * Process temperature [K]
 6:     Rotational speed [rpm] * Torque [Nm]
 7:     Torque [Nm] * Tool wear [min] * Type
 8: Step 5: Mean Normalize.
 9: Step 6: Split dataset into features and labels
10: Step 7: Split dataset into training and testing sets with a 7:3 ratio.
11: Step 8: Balance the dataset
12:     ADASYN: Oversample the minority class.
13:     Cluster Undersampling: Reduce the majority class to a defined ratio.
14: return $X_{train}$, $Y_{train}$, $X_{test}$, $Y_{test}$
```

2.1.1 Split and Shuffle

To evaluate model performance reliably, the dataset is split into a training set and a test set. Shuffling ensures that the split is randomized, avoiding biases due to the order of data points. Typically, the dataset is divided into 70% for training and 30% for testing. The rationale for shuffling and splitting is to simulate unseen data conditions during testing.

```
features = data.iloc[:, :-1].values
labels = data.iloc[:, -1].values
```

```
indices = np.random.permutation(len(labels))
features = features[indices], labels = labels[indices]
```

```
train_size = int(train_ratio * len(labels))
X_train = features[:train_size], Y_train = labels[:train_size]
X_test = features[train_size:], Y_test = labels[train_size:]
```

2.1.2 Upsampling

Due to the imbalanced nature of the dataset, where machine failures are relatively rare compared to non-failure cases, upsampling is used to replicate minority class samples. This technique ensures that the learning algorithm is not biased toward the majority class and can effectively identify failures.

$$\text{Difficulty of Categorization } r_i = \frac{\text{Number of Majority near the Minority } x_i}{k} \tag{1}$$

$$\text{Amount of Generation } g_i = round(\frac{r_i}{\sum_{j=1}^{N_minority} r_j} \cdot \beta \cdot (N_{majority} - N_{minority})) \tag{2}$$

$$\text{New Sample } x_{synthetic} = x_i + \lambda \cdot (x_{neighbour} - x_i), \quad \lambda \sim U(0,1) \tag{3}$$

Algorithm 2 ADASYN Algorithm

```
Input: Dataset $(X, y)$, minority class label, $k$ neighbors, balance ratio $\beta$
Output: Resampled dataset $(X_{\text{resampled}}, y_{\text{resampled}})$
1: Split $X$ into $X_{\text{minority}}$ and $X_{\text{majority}}$
2: Compute number of samples to generate: $G = \beta \times (\#\text{majority} - \#\text{minority})$
3: Calculate difficulty for each minority sample using $k$-nearest neighbors
4: Normalize difficulty to get sampling weights
5: for each minority sample $x_i$ do
6:     Generate $g_i$ synthetic samples:
7:     Randomly select a neighbor and create new samples along the line
8: end for
9: Append synthetic samples to the original dataset
10: return $(X_{\text{resampled}}, y_{\text{resampled}})$
```

2.1.3 Downsampling

Alternatively, downsampling reduces the number of majority class samples to balance the dataset. This approach is useful when computational resources are limited or when over-representing the minority class might introduce redundancy. However, it comes at the cost of losing some information from the majority class.

$$\text{Class Center } \mu_k = \frac{1}{|C_k|} \sum_{x \in C_k} x, \quad k = 1, 2, \ldots, n_{\text{majority\_target}} \tag{4}$$

$$\text{Representative Sample } x_{\text{representative}} = \arg\min_{x \in C_k} \|x - \mu k\|^2 \quad (5)$$

Algorithm 3 Cluster-Based Undersampling Algorithm

```
Input: Dataset $(X, y)$, undersampling ratio ratio
Output: Resampled dataset $(X_{\text{resampled}}, y_{\text{resampled}})$
1: Identify majority and minority classes based on $y$
2: Split $X$ into $X_{\text{majority}}$ and $X_{\text{minority}}$
3: Compute target majority class size: $n_{\text{majority\_target}} = \frac{n_{\text{minority}}}{(1-\text{ratio})} - n_{\text{minority}}$
4: Apply $K$-Means clustering on $X_{\text{majority}}$ with $n_{\text{majority\_target}}$ clusters
5: Select one representative sample (nearest to cluster center) from each cluster
6: Combine $X_{\text{minority}}$ with the representative samples
7: return $(X_{\text{resampled}}, y_{\text{resampled}})$
```

2.2 Characteristic Engineering

2.2.1 Trend of Each Feature

I organized the data so that the first 9661 samples had a label of 0 and the last 339 samples had a label of 1. Drawing out the table I found that it was hard to separate the labels with individual features, and that it might be necessary to do interactive features or other non-linear features.

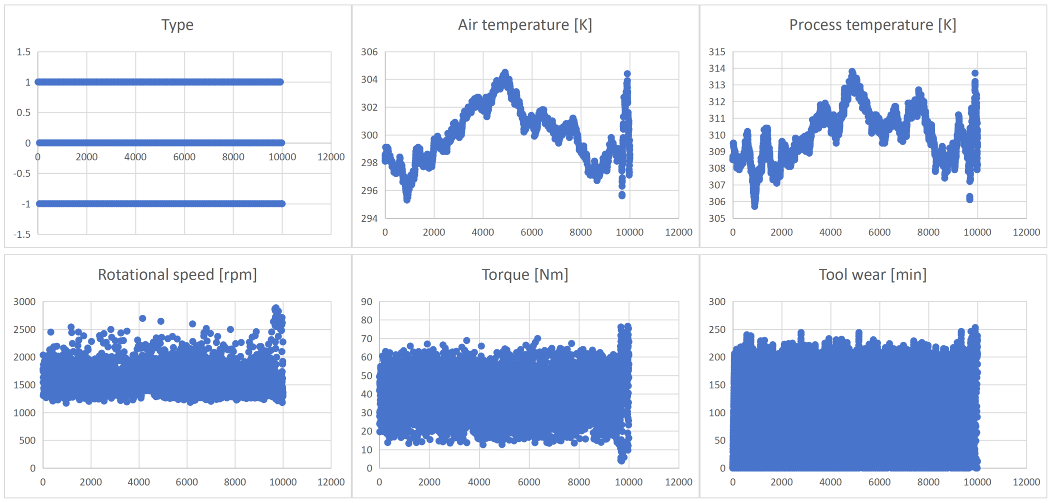


Fig. 1: Trend of Each Feature

2.2.2 Interactive Feature

According to the information of the dataset:

> 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.

The five failure modes in the dataset are determined by different variables and their complex interactions, such as tool wear time, temperature difference, rotational speed, power range, and product type. In addition, the triggering conditions for some modes are nonlinear or dependent on combinations between features (e.g., the product of power and torque). To accurately capture these complex relationships and improve the predictive performance of the model, feature engineering can help us extract, transform, and construct more representative information that allows the model to learn the key features of the failure mode more effectively.

```
data_cleaned['Type'] = data_cleaned['Type'].replace({'L': 11, 'M': 12, 'H': 13}).
    infer_objects(copy=False).astype('float64')
data_cleaned['AirTemp_ProcessTemp'] = data_cleaned['Air temperature [K]'] -
    data_cleaned['Process temperature [K]']
data_cleaned['RotSpeed_Torque'] = data_cleaned['Rotational speed [rpm]'] *
    data_cleaned['Torque [Nm]']
data_cleaned['Torque_ToolWear_TypeL'] = data_cleaned['Torque [Nm]'] * data_cleaned
    ['Tool wear [min]'] * data_cleaned['Type']
```

To be able to get a reasonable product when the third new variable is added, I assign a value to the Type variable based on the coefficients in the dataset description, and the Type variable is normalized when it is used on its own, so my assignment doesn't affect its use on its own.

2.3 Model

This section describes the machine learning models used in this study. Four different models were implemented and evaluated: Linear Regression, Perceptron, Logistic Regression, and Multi-Layer Perceptron (MLP). Below, we provide detailed explanations of each model, their mathematical formulations, and their role in the experiments.

Model	Output Format	Loss Function	Features
Linear Regression	Numerical	Mean Squared Error (MSE)	Simple and interpretable
Perceptron	Class (+1/-1)	No explicit loss, update rule	Requires linearly separable data
Logistic Regression	Probability values	Cross-Entropy Loss	Outputs probabilities, extendable to multi-class
Multilayer Perceptron	Numerical or class probabilities	Typically Cross-Entropy Loss or MSE	Handles non-linearity, suitable for complex patterns

Table 1: Comparison of Linear Regression, Perceptron, Logistic Regression, and Multi-layer Perceptron

2.3.1 Linear Regression

Linear Regression is one of the simplest and most widely used supervised learning algorithms. It assumes a linear relationship between the input features X and the target variable y. The mathematical representation of the model is:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon \tag{6}$$

where β_0 is the intercept, $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients of the model, and ϵ is the error term.

To estimate the coefficients, the model minimizes the Mean Squared Error (MSE):

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{7}$$

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and N is the number of samples.

In this study, Linear Regression served as a baseline model for predicting continuous variables. It allowed us to assess the relationship between features and the target variable while providing a point of comparison for more complex models.

2.3.2 Perceptron

The Perceptron is an early neural network model designed for binary classification tasks. It works by finding a linear decision boundary to separate two classes. The decision function is defined as:

$$y = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \tag{8}$$

where $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the input vector, and b is the bias term.

The Perceptron algorithm iteratively updates the weights based on misclassified samples. The update rule is:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y_i - \hat{y}_i)\mathbf{x}_i \tag{9}$$

where η is the learning rate, y_i is the true label, and $\hat{y}_i$ is the predicted label.

While the Perceptron is only effective for linearly separable data, it provides valuable insights into the basics of neural network learning. In this study, it was employed to classify binary datasets and to demonstrate its limitations on more complex, non-linearly separable data.

2.3.3 Logistic Regression

Logistic Regression is a popular classification algorithm that predicts probabilities rather than direct class labels. It extends linear regression by applying a sigmoid function to the output, mapping values to the range [0, 1]:

$$P(y = 1|X) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}} \tag{10}$$

where $\mathbf{w}$ and b are the model parameters.

The goal of Logistic Regression is to maximize the likelihood of the observed data, or equivalently, to minimize the negative log-likelihood loss:

$$L(\mathbf{w}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{11}$$

where $\hat{y}_i$ is the predicted probability for sample i.

Logistic Regression can be extended to multiclass problems using techniques such as the one-vs-all approach or the softmax function. In this study, Logistic Regression was applied to both binary and multiclass classification tasks. Its interpretability and simplicity made it an essential benchmark for evaluating more complex models.

2.3.4 Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP) is a type of feedforward neural network that consists of an input layer, one or more hidden layers, and an output layer. Each layer performs a linear transformation followed by a nonlinear activation function. The computation in a single layer is given by:

$$h^{(l)} = \sigma(W^{(l)} h^{(l-1)} + b^{(l)}) \tag{12}$$

where $h^{(l)}$ is the output of layer l, $W^{(l)}$ and $b^{(l)}$ are the weights and biases, and σ is the activation function (e.g., ReLU, Sigmoid).

The MLP learns by minimizing a loss function, such as cross-entropy for classification, through backpropagation and gradient descent. The model's capacity to capture complex, non-linear relationships makes it suitable for a wide range of tasks.

In this study, the MLP was used for both regression and classification tasks. By tuning hyperparameters such as the number of layers, neurons per layer, learning rate, and regularization strength, we optimized its performance on high-dimensional datasets. The flexibility of MLP allowed it to outperform simpler models on non-linearly separable data, demonstrating its utility for more challenging problems.

2.4 Model Optimization

After the basic implementation of the model is complete, for problems as complex as this task, we still need many optimization adjustments to the model:

2.4.1 Activation Function

For nonlinear problem, we need activation function. There are kinds of activation functions, and we try them in our test to find the best one in our task.

Sigmoid Function

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{13}$$

- Output range: $(0, 1)$
- Used primarily in binary classification problems.
- Smooth and differentiable, but suffers from the vanishing gradient problem for large input values.
- The output can be interpreted as a probability, making it useful for probability-based decision-making.

Tanh Function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{14}$$

- Output range: $(-1, 1)$
- Often used in hidden layers of neural networks.
- Zero-centered, which can help speed up convergence during training.
- Still suffers from the vanishing gradient problem for very large or small input values.

ReLU Function

$$\text{ReLU}(x) = \max(0, x) \tag{15}$$

- Output range: $[0, +\infty)$
- Very simple and computationally efficient.
- Helps to avoid the vanishing gradient problem.
- The output is non-negative, and negative inputs are squashed to zero, which may lead to ”dead neurons” in some cases.

Leaky ReLU Function

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{16}$$

where α is a small constant (e.g., 0.01).

- Output range: $(-\infty, +\infty)$
- Addresses the ”dead neuron” problem by allowing a small, non-zero gradient for negative input values.
- Simpler to compute than some other activations, like sigmoid or tanh.
- Helps models learn faster and may prevent neurons from dying during training.

Softmax Function

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{17}$$

- Output range: $(0, 1)$ with all outputs summing to 1.
- Commonly used in the output layer of a classification model to represent class probabilities.
- Useful for multi-class classification problems.
- Transforms raw scores into probabilities that can be interpreted as class membership probabilities.

GELU (Gaussian Error Linear Unit) Function

$$\text{GELU}(x) = x \cdot \Phi(x) \tag{18}$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution.

- Output range: $(-\infty, +\infty)$
- Smooth activation function that combines elements of ReLU and a probabilistic approach.
- It performs better than ReLU in some cases, especially in deep networks such as transformers.
- The approximation of the cumulative distribution allows for a smooth, non-linear behavior.

Softplus Function

$$\text{Softplus}(x) = \log(1 + e^x) \tag{19}$$

- Output range: $(0, +\infty)$
- A smooth approximation of the ReLU function.
- Helps avoid the issue of dead neurons found in ReLU.
- Computationally more expensive than ReLU, but provides a continuous transition from negative to positive values.

2.4.2 Loss Function

The loss function is used to measure the gap between the model's predictions and the true value and is a central tool for optimizing the model. By minimizing the loss function, the model can progressively learn more accurate predictions, which improves performance and generalization, and is a key element in the training process.

Mean Squared Error Loss

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{20}$$

- Measures the average squared difference between actual (y_i) and predicted $(\hat{y}_i)$ values.
- Output is always non-negative, and smaller values indicate better predictions.
- Commonly used for regression tasks.
- Sensitive to outliers due to the squared term, which gives larger penalties for larger errors.

$$\text{Weighted MSE} = \frac{1}{n}\sum_{i=1}^{n} w_i(y_i - \hat{y}_i)^2 \tag{21}$$

For unbalanced dataset, it would be better if we have weighted version. It's useful when certain samples are more important than others. It allows for prioritization of key data points during model training.

Cross-Entropy Loss

$$\text{Binary Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \tag{22}$$

$$\text{Categorical Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{i,k}\log(\hat{y}_{i,k}) \tag{23}$$

- Measures the dissimilarity between the predicted probability distribution and the true distribution.
- Particularly effective for classification tasks.
- Encourages predicted probabilities to be close to the true class probabilities.
- Sensitive to imbalanced datasets, which may require modifications like weighting.

$$\text{Weighted Binary Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n} w_i\left[y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \tag{24}$$

$$\text{Weighted Categorical Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} w_k y_{i,k}\log(\hat{y}_{i,k}) \tag{25}$$

Also, weighted version may works well. It's useful for addressing class imbalance by giving higher weights to under-represented classes. It helps models focus more on important classes or samples.

2.4.3 Dynamic Hyperparameter

Learning Rate
The learning rate is an important hyperparameter that controls the pace of updating the model parameters, and it determines the step size of each gradient descent. Too large a learning rate may cause the optimization process to diverge, while too small a learning rate may lead to slow convergence or fall into a local optimum. Therefore, choosing an appropriate learning rate is crucial for model training.
However, a single learning rate may not be able to balance the efficiency and stability during the training process, so the method of dynamically adjusting the learning rate is introduced. By using a larger learning rate in the early stage of training, convergence can be accelerated and the optimal region can be approached quickly, while gradually reducing the learning rate in the later stage can avoid parameter oscillations or skipping the optimal solution, and thus achieve finer optimization. Strategies that dynamically adjust the learning rate, such as learning rate decay, adaptive optimization methods, or performance-based tuning, can improve training efficiency, stability, and the final performance of the model.

```
self.lr = 0.9999 * self.lr
```

Momentum
Momentum tuning is a technique commonly used in optimization algorithms to accelerate convergence and improve the stability of the optimization process. The core idea is to introduce a "momentum" variable, which is adjusted at each parameter update by combining the current gradient and historical gradient information.
Specifically, momentum accumulates the previous gradient information so that the parameter update not only depends on the current gradient, but also is influenced by the direction of previous updates. In this way, momentum adjustment can effectively accelerate convergence in the direction of smooth gradient changes and reduce gradient oscillations, especially in loss functions with long "canyon" shapes. A commonly used momentum coefficient (e.g., 0.9) controls the weight of the historical gradient on the update, and its successful application has widely improved the training efficiency and effectiveness of deep learning models.

$$v_t = \gamma v_{t-1} + \eta\nabla J(\theta_t) \tag{26}$$

$$\theta_{t+1} = \theta_t - v_t \tag{27}$$

where v_t is current momemtum, γ is momemtum coefficient, η is learning rate, $\nabla J(\theta_t)$ is current gradient.

```
self.velocity = self.momentum * self.velocity + self.lr * grad
self.W -= self.velocity
```

Adam
Adam (Adaptive Moment Estimation) is a widely used optimization algorithm that combines the advantages of momentum and adaptive learning rates to enhance efficiency and stability during optimization. Adam computes the first-order moment (mean) and the second-order moment (variance) of gradients to adaptively adjust the learning rate for each parameter. This allows Adam to handle sparse gradients and non-stationary objectives effectively.

$$\text{Gradient } g_t = \nabla J(\theta_t) \tag{28}$$

$$\text{First Order Momemtum } m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{29}$$

$$\text{Second Order Momemtum } v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{30}$$

$$\text{Bias-Corrected Values } \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{31}$$

$$\text{Update Method: } \theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{32}$$

- g_t is the gradient of the loss function $J(\theta)$ at time step t.
- m_t and v_t are the first-order and second-order moment estimates, respectively.
- β_1 and β_2 are the exponential decay rates for the moment estimates, typically set to 0.9 and 0.999.
- η is the learning rate.
- ϵ is a small constant (e.g., 10^{-8}) to avoid division by zero.

By dynamically adjusting the learning rate for each parameter based on its gradient statistics, Adam ensures stable and efficient optimization, making it one of the most popular optimizers in deep learning.

Algorithm 4 Adam Optimization Algorithm

```
Input: Initial parameters $\theta_0$, learning rate $\eta$, exponential decay rates $\beta_1, \beta_2 \in [0,1)$, small constant $\epsilon$ (e.g., $10^{-8}$),
    number of iterations $T$
Output: Optimized parameters $\theta_T$
 1: Initialize: $m_0 \leftarrow 0$ (initialize first moment vector), $v_0 \leftarrow 0$ (initialize second moment vector), $t \leftarrow 0$ (initialize
    time step)
 2: for $t = 1$ to $T$ do
 3:     Compute gradient
 4:     Update biased first moment estimate
 5:     Update biased second moment estimate
 6:     Correct bias in first moment estimate
 7:     Correct bias in second moment estimate
 8:     Update parameters
 9: end for
10: return Optimized parameters $\theta_T$
```

Dropout
Dropout is a regularization technique used to prevent overfitting in deep neural networks. It works by randomly ”dropping out” (i.e., setting to zero) a subset of neurons during training with a probability $1-p$, where p is the probability of retaining a neuron. This forces the network to learn more robust and distributed feature representations, as it cannot rely on specific neurons. During testing, Dropout is not applied; instead, the outputs of neurons are scaled by p to ensure consistency between training and inference. Mathematically, during training, the output of a neuron is modified as:

$$\tilde{z} = r \cdot z, \quad r \sim \text{Bernoulli}(p), \tag{33}$$

where z is the original output, r is a random variable sampled from a Bernoulli distribution, and p is the retention probability. During testing, the output is scaled as:

$$z_{\text{test}} = p \cdot z. \tag{34}$$

This technique improves generalization by reducing co-adaptation of neurons and effectively creates an ensemble of sub-networks during training, enhancing the model's robustness and performance.

```python
def dropout(self, x):
  if self.training:
    mask = np.random.rand(*x.shape) > self.dropout_rate
    return x * mask / (1 - self.dropout_rate)
  return x
```

2.5 Evaluation

After training, it's necessary to evaluate our model. We can compare the prediction and true label, score them or visualize them.

2.5.1 Score Evaluation

The confusion matrix gives a good indication of the performance of the binary classification problem and consists of:

- True Positive (TP): the number of positive classes that the model correctly predicts as positive.
- False Positive (FP): the number of negative classes that the model incorrectly predicts as positive.
- True Negative (TN): the number of negative classes that the model correctly predicts as negative.
- False Negative (FN): the number of negative classes that the model incorrectly predicts as negative.

To evaluate on the confusion matrix, we can see those scores:

- The total proportion of true prediction:

$$\textbf{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

- For all predicted true, how many samples are really true:

$$\textbf{Precision} = \frac{TP}{TP + FP}$$

- For all exact true samples, how many samples are predicted to be found out true:

$$\textbf{Recall} = \frac{TP}{TP + FN}$$

- The harmonic mean of precision and recall:

$$\textbf{F1-Score} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

In this task, we are making predictions about whether a machine has malfunctioned and needs to be repaired, such that it is particularly important to avoid the situation where a machine needs to be repaired but is not detected, i.e., the value of false negatives should be low and the recall should be high. For the rest of the equivalents, one should also focus on the F1 scores to get the most comprehensive estimate of the model effect.

2.5.2 Visualization

We can also choose some features (eg. 3 dimensions) to see the feature distribution, and observe TP/TN/FP/FN by marking.
As what we found before, we have add three cross feature, so we can take a look of them.

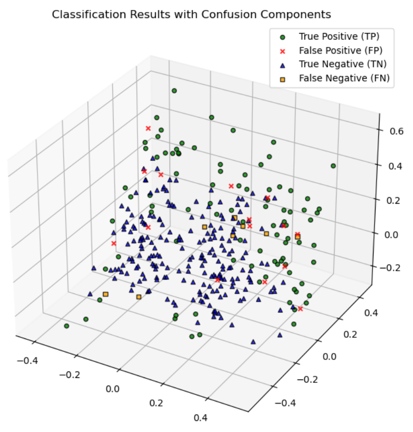


Fig. 2: Visualization Example

3 Experiment Result

We apply all the stuff above, choose different parameters to find the most suitable one, then comes out with the Trained Model.

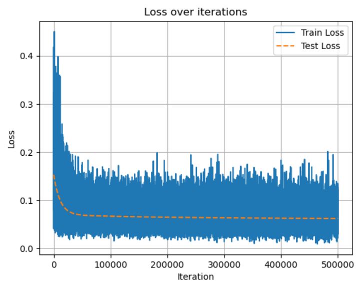
For all model, we have those results:

- Loss Trend During Training
- Visualization of My Model
- Using sklearn, Model Performance
- Confusion Matrix
- Total Execution Time(Maybe not accurate as I didn't close the text and figure output)

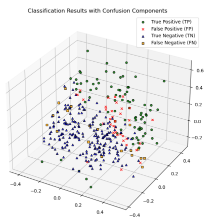
3.1 Linear Regression

linear_model = LinearRegression(n_iter=50000, lr=8e-4, batch_size=64)
TP: 83 TN: 206 FP: 36 FN: 21
Accuracy: 0.8352601156069365 Precision: 0.6974789915966386 Recall: 0.7980769230769231 F1 Score: 0.7443946188340808
Total time taken: 38.09750461578369 seconds

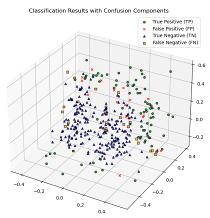
linear_sklearn = LinearRegression()
TP: 78 TN: 217 FP: 20 FN: 24
Accuracy: 0.8702064896755162 Precision: 0.7959183673469388 Recall: 0.7647058823529411 F1 Score: 0.7799999999999999
Total time taken: 5.636926174163818 seconds



(a) Loss Trend



(b) My Model Performance



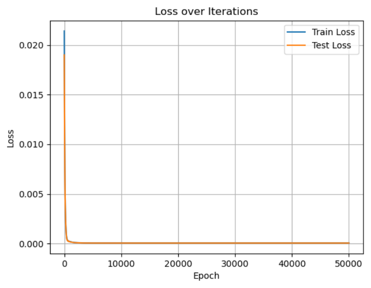
(c) scikit-learn Model Performance

Fig. 3: Linear Regression

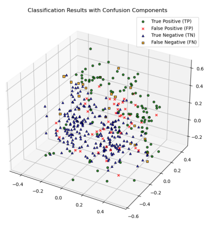
3.2 Perceptron

perceptron_model = Perceptron(n_iter=50000, lr=2e-3, batch_size=64)
TP: 102 TN: 242 FP: 44 FN: 21
Accuracy: 0.8410757946210269 Precision: 0.6986301369863014 Recall: 0.8292682926829268 F1 Score: 0.758364312267658
Total time taken: 19.484343767166138 seconds

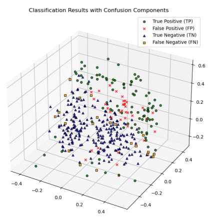
perceptron_sklearn = Perceptron(max_iter=100000, random_state=42)
TP: 94 TN: 232 FP: 36 FN: 21
Accuracy: 0.8511749347258486 Precision: 0.7230769230769231 Recall: 0.8173913043478261 F1 Score: 0.7673469387755102
Total time taken: 2.7166144847869873 seconds



(a) Loss Trend



(b) My Model Performance



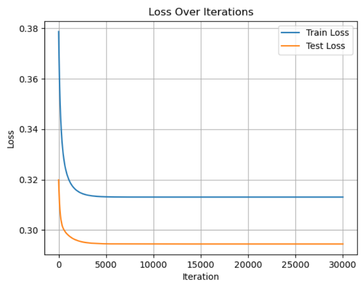
(c) scikit-learn Model Performance

Fig. 4: Perceptron

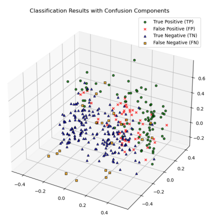
3.3 Logisitic Regression

logistic_model = LogisticRegression(n_iter=30000, lr=3e-3, batch_size=64)
TP: 85 TN: 195 FP: 52 FN: 21
Accuracy: 0.7932011331444759 Precision: 0.6204379562043796 Recall: 0.8018867924528302 F1 Score: 0.6995884773662552
Total time taken: 19.374540090560913 seconds

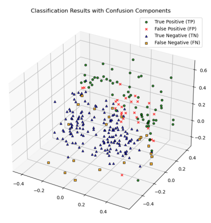
logistic_sklearn = LogisticRegression(max_iter=100000, random_state=42)
TP: 61 TN: 165 FP: 26 FN: 21
Accuracy: 0.8278388278388278 Precision: 0.7011494252873564 Recall: 0.7439024390243902 F1 Score: 0.7218934911242605
Total time taken: 2.442545175552368 seconds



(a) Loss Trend



(b) My Model Performance



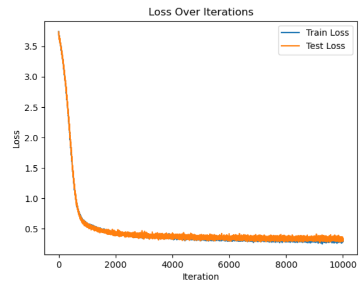
(c) scikit-learn Model Performance

Fig. 5: Logisitic Regression

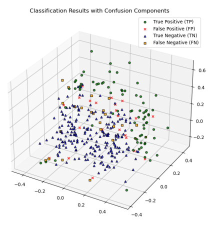
3.4 Multi-layer Perceptron

mlp_model = MultiLayerPerceptron(layer_sizes=[X_train.shape[1],47,101,32], n_iter=10000, lr=1e-5, batch_size=32)
TP: 87 TN: 213 FP: 20 FN: 13
Accuracy: 0.9009009009009009 Precision: 0.8130841121495327 Recall: 0.87 F1 Score: 0.8405797101449274
Total time taken: 421.797244310379 seconds

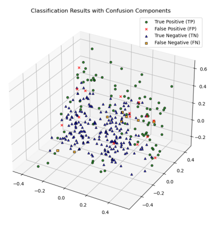
mlp_sklearn = MLPClassifier(hidden_layer_sizes=(12,47,101,11), activation='relu', solver='adam', max_iter=10000, random_state=42)
TP: 90 TN: 213 FP: 15 FN: 8
Accuracy: 0.9294478527607362 Precision: 0.8571428571428571 Recall: 0.9183673469387755 F1 Score: 0.8866995073891625
Total time taken: 8.705824136734009 seconds



(a) Loss Trend



(b) My Model Performance



(c) scikit-learn Model Performance

Fig. 6: Multi-layer Perceptron

4 Analyzation

4.1 Performance

Model	TP	TN	FP	FN	Accuracy	Precision	Recall	F1-Score
Linear Regression	83	206	36	21	0.8353	0.6977	0.7980	0.7443
Perceptron	102	242	44	21	0.8411	0.6986	0.8292	0.7584
Logistic Regression	85	195	52	21	0.7932	0.6204	0.8019	0.6996
Multi-layer Perceptron	87	213	20	13	0.9009	0.8131	0.8700	0.8405

Table 2: Performance of Linear Regression, Perceptron, Logistic Regression, and Multi-layer Perceptron

Linear Regression Linear regression, while performing moderately well in terms of accuracy (83.53%) and F1 score (0.7443), had a low precision (0.6977), suggesting that it is prone to a high number of false positives (FP = 36). In addition, its number of false negatives (FN = 21) was not satisfactory enough to effectively avoid underreporting. More importantly, linear regression is a linear model with limited modeling ability for nonlinear problems, which makes it difficult to capture complex input feature patterns and thus may lead to limited overall performance.

Perceptron Perceptron's accuracy (84.11%) is slightly higher than that of linear regression, but its F1 score (0.7584) and precision (0.6986) are still lower, especially in terms of precision, which is similar to that of linear regression, suggesting that it is still more prominent in false-positive problems (FP = 44). Although the perceptron introduces a simple threshold determination mechanism, it is still essentially a linear model with limited ability to handle non-linear problems. At the same time, the perceptual machine is prone to unstable performance in datasets with more complex boundaries than more complex models.

Logistic Regression The overall performance of logistic regression is inferior to the other models, with the lowest accuracy (79.32%), F1 score (0.6996), and precision (0.6204). Logistic regression had more false positives (FP = 52). Although logistic regression is able to handle some of the nonlinear relationships through feature transformations, it is limited in its ability to model nonlinear patterns as deeply as more complex models such as neural networks. In addition, logistic regression has operational efficiency advantages when data volumes are small, but may be difficult to scale in big data scenarios.

Multi-layer Perceptron Multi-layer Perceptron performs best in terms of recall (Recall, 0.8700), F1 score (0.8405), and precision (0.8131), and has the lowest number of false negatives (FN = 13), which makes it particularly suitable for our scenario. However, its running time and resource consumption are much higher than other models, especially in the training phase, where MLP needs to adjust numerous parameters (e.g., learning rate, number of neurons in the hidden layer) and the process of hyper-parameter tuning is more complicated. In addition, the excellent performance of MLP relies on a sufficient amount of data, which may not be robust enough in small data scenarios or noisy data, and there may be a risk of overfitting.

The limitations of linear regression and perceptual machines on nonlinear problems make them difficult to cope with the complexity requirements of this problem; logistic regression is able to partially model nonlinear relationships, but its performance metrics are overall poor. Multilayer Perceptron is the most suitable model for the problem at hand, especially in scenarios where false negatives need to be minimized, but at the cost of more computational resources and time. Taken together, the choice of the MLP, combined with appropriate hyperparameter tuning, is the best trade-off between performance and computational cost in the problem at hand.

4.2 Advanced Try

According to the article, Matzka, S. (2020). Explainable Artificial Intelligence for Predictive Maintenance Applications. 2020 Third International Conference on Artificial Intelligence for Industries (AI4I), 69-74.

> In this paper, two methods to provide an explanation for the classification result of a complex ensemble classifier are evaluated on a synthetic predictive maintenance dataset. Both methods have inherent strengths and weaknesses, but provide an overall benefit for the user without high additional costs.
>
> The explanations provided by the decision trees tends to be of a higher quality, but in a considerable number of cases do not provide any explanation. On the other side, normalized feature deviations provide explanations of a consistent, yet slightly lower, explanatory quality.

The AI4I dataset is made to test decision tree. So we can try the performance on it.

decision_tree_model = DecisionTreeClassifier(criterion='entropy', max_depth=None, random_state=42)
My simple decision tree test: TP: 77 TN: 2643 FP: 271 FN: 9
Bagged decision tree test according to the article: TP: 294 TN: 9540 FP: 121 FN: 45
We have similar result(just different amount of test), and it really works well here.

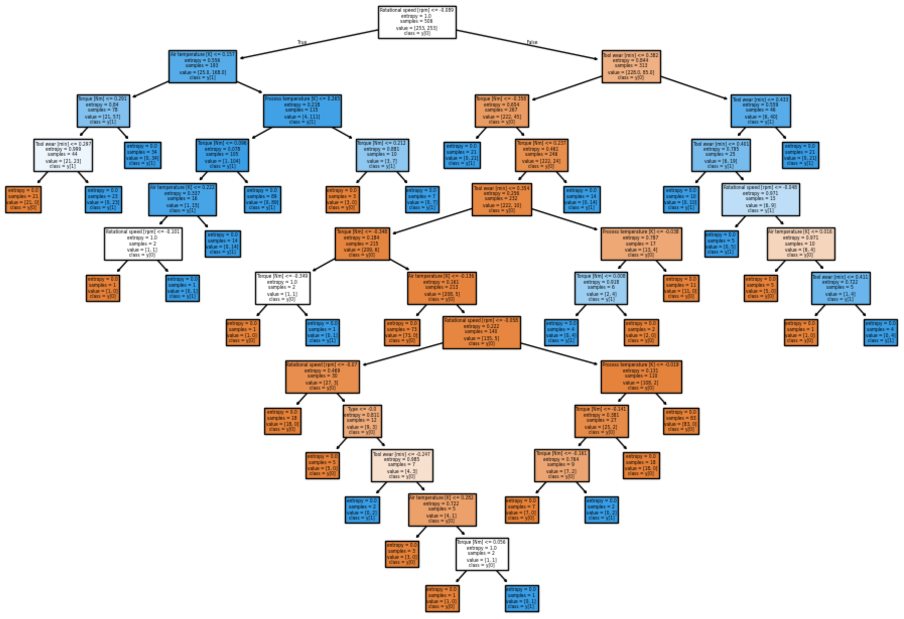


Fig. 7: Decision Tree

5 Conclusion

In this project, various machine learning models were employed to predict machine failures using the AI4I 2020 dataset, showcasing the application of advanced algorithms in predictive maintenance. Key findings reveal that linear regression and perceptrons, while simple and interpretable, struggled to capture the nonlinear relationships in the data. Logistic regression demonstrated some ability to address these complexities but ultimately fell short in terms of precision and recall.

The multi-layer perceptron (MLP) stood out as the most effective model, achieving superior metrics such as an F1-score of 0.841 and the lowest number of false negatives. This makes MLP particularly well-suited for scenarios where undetected failures carry significant risks. However, these advantages come with increased computational costs and the need for careful hyperparameter tuning to prevent overfitting and optimize performance.

The study also highlights the importance of data preprocessing and feature engineering, such as handling class imbalances and generating interactive features, which significantly impacted model accuracy. The findings demonstrate that while complex models like MLPs are powerful, their success depends on thoughtful preparation of the data and appropriate optimization strategies.

Overall, this work underscores the transformative role of machine learning in predictive maintenance, providing industries with tools to reduce downtime, optimize resources, and enhance operational reliability. Future work could explore integrating explainable AI methods to improve transparency and trust in these predictive models.