Machine Learning Model Comparison for Robot Diagnosis

Hyunmin Yoo

10/5/25 *(date completed)*

Abstract

This project explores artificial intelligence methods that enable robots to diagnose operational failures autonomously, reducing downtime and human intervention. A multivariate time series dataset from the UC Irvine KDD Archive, containing force and torque measurements, was used to train and evaluate six models: multilayer perceptron, logistic regression, decision tree, k-nearest neighbors, random forest, and XGBoost. Hyperparameters for each model were tuned through manual exploration and grid search. Random forest and XGBoost achieved the highest test accuracy at 94.4 percent. The multilayer perceptron and k-nearest neighbors reached 83.3 percent. The decision tree reached 77.8 percent, and logistic regression reached 72.2 percent. These findings demonstrate that ensemble and neural network-based approaches outperform simpler classifiers for robot failure prediction, showcasing the feasibility of integrating AI-driven self-diagnosis into robotic systems, supporting the feasibility of AI-driven self-diagnosis to improve reliability, safety, and efficiency.

1. Introduction

The primary focus of this project is the development of artificial intelligence (AI) methods to enable autonomous robotic self-diagnosis. As robots become increasingly prevalent in healthcare, manufacturing, and everyday environments, the ability to detect and assess internal faults is essential. By equipping robots with diagnostic capabilities, they can operate more safely, minimize downtime, and reduce reliance on human intervention. This advancement enhances the efficiency and usefulness of robotic systems in various applications.

The motivation for undertaking this project originated from my involvement in the school's robotics team. Having cultivated a strong interest and aptitude in robotics, the researcher sought to move beyond traditional tasks of building and programming robots to explore how AI could enhance robotic functionality. The concept of robots autonomously assessing their own condition and making decisions regarding maintenance or repairs was particularly compelling, representing an innovative intersection between existing skills and new technological frontiers.

This project is especially significant because it integrates two fields that the researcher finds most engaging: robotics and artificial intelligence. Robotics provides opportunities for design, construction, and problem-solving, while AI introduces the potential for autonomous learning and improvement. By combining these disciplines, multiple AI models were able to be evaluated and determine their effectiveness in enabling robots to operate more independently. Experimentation with different models and performance assessments yielded valuable insights into the practical applications of AI in robotics.

This research distinguishes itself from similar work in the field by adopting a comparative and systematic approach rather than focusing on a single model or method. Several AI models were evaluated, and extensive hyperparameter optimization was conducted to maximize performance. Initial testing employed random hyperparameter selections to observe model behavior, followed by a structured grid search to identify optimal configurations. This

comprehensive methodology sets the project apart from research limited to a single model or less rigorous parameter tuning.

2. Background

Robotic systems are increasingly used in healthcare, manufacturing, and service environments where continuous and reliable operation is essential. Interruptions in performance create costs and can also pose risks to safety and productivity. Conventional maintenance relies on human supervision, which introduces delays and raises expenses. This limitation has motivated the development of autonomous self-diagnosis, in which robots can automatically detect, classify, and respond to faults without external intervention.

Self-diagnosis involves three core elements. Fault detection identifies deviations from normal operation. Sensor fusion combines information from multiple sensors to improve reliability. Machine learning methods map processed sensor data to fault categories. Force and torque measurements are often collected at high frequencies because they capture subtle variations that can reveal emerging faults. These signals are transformed into features that classification models can analyze.

Early work relied on single classifiers such as logistic regression and decision trees. These approaches are simple and interpretable but often generalize poorly without careful tuning. More recent studies show that ensemble methods such as random forests and boosting reduce overfitting and improve accuracy. Neural networks, particularly multilayer perceptrons(MLP), can model nonlinear relationships in high-dimensional signals. As the field has progressed, researchers have adopted more systematic tuning of depth, number of estimators, learning rate, and hidden layer size to balance bias and variance.

This body of work shows a shift from simple classifiers to more advanced ensemble and neural architectures. These developments are supported by optimization strategies that adjust parameters such as tree depth, number of estimators, learning rate, and hidden layer size. Together, these advancements highlight the potential of integrating artificial intelligence-based self-diagnosis into robotic systems to improve reliability, safety, and efficiency.

3. Dataset

The dataset [1] used in this study was obtained from the UC Irvine KDD Archive [2], specifically the robot failure database, which includes five multivariate time-series sub-datasets of force and torque values in the x, y, and z directions measured in newtons and newton-metres at fifteen intervals over 315 milliseconds, producing ninety features per observation.

In the original dataset, the intervals for each sample were stored across multiple rows, making direct analysis more difficult. To address this, the data were reformatted by flattening the intervals so that all fifteen intervals for a given sample were consolidated into a single row. This restructuring preserved the temporal information while producing a clean, tabular format suitable for machine learning applications, as seen in Figure 1.

	fx1	fy1	fz1	tx1	ty1	tz1	fx2	fy2	fz2	tx2	...	tx14	ty14	tz14	fx15	fy15	fz15	tx15	ty15	tz15	class
0	-1	-1	63	-3	-1	0	0	0	62	-3	...	-3	0	0	-1	0	64	-2	-1	0	normal
1	-1	-1	63	-2	-1	0	-1	-1	63	-3	...	-6	-3	-1	-1	-1	59	-3	-4	0	normal
2	-1	0	57	-5	-3	0	0	-3	63	-1	...	-6	-2	-1	-1	0	54	-4	-3	0	normal
3	0	-1	59	-2	-1	-1	0	-3	61	-1	...	-4	0	0	-1	1	63	-8	-2	0	normal
4	0	-2	65	-4	-2	0	-1	-2	56	-5	...	-8	-4	-1	-1	1	57	-9	-4	-1	normal

Figure 1: Processed Data

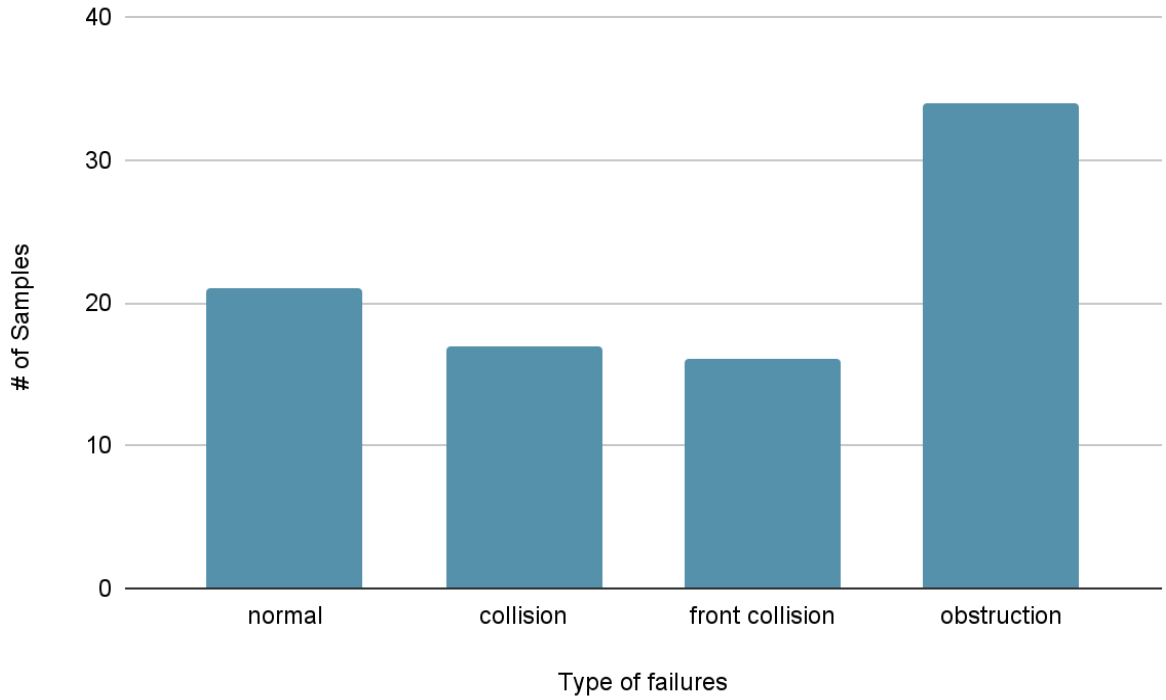

Figure 2: Distribution of Samples

For experimental purposes, the processed dataset was divided into two partitions, with 80 percent allocated for model training and the remaining 20 percent reserved for testing. This split ensured that models were evaluated on unseen data to provide an unbiased measure of generalization performance.

4. Methodology / Models

The objective is to learn a mapping that assigns each vector of sensor readings to a discrete failure category. The study employs a multilayer perceptron (MLP) classifier, logistic regression,

decision tree, and k-nearest neighbors, and applies a grid-search procedure to identify the optimal parameters for the multilayer perceptron model.

4.1 MLP Classifier

MLP Classifier is a neural network used for classification tasks. Default parameters for this model are hidden_layer_sizes=(100,),activation='relu', solver='adam', max_iter=200.

The multilayer perceptron (MLP) classifier is a feed-forward neural network that processes input through hidden layers of neurons. Each of the ninety features (fifteen intervals of six force and torque measurements in the x, y, and z directions) is weighted, and these weights are iteratively updated to minimize classification error. The weights indicate the relative importance of each feature, providing feature extraction.

The key hyperparameters include the number of hidden layers and neurons per layer, which control the model's capacity to capture complex nonlinear relationships. Too few neurons can lead to underfitting, while too many may overfit the training data. The activation function determines how signals pass through layers; nonlinear activations such as ReLU or logistic allow the network to model complex decision boundaries. The solver (optimization algorithm) impacts training efficiency and convergence; gradient-based solvers like Adam adapt learning rates during training, while others like LBFGS are more stable for small datasets. The learning rate controls the step size of weight updates, where higher values speed convergence but risk instability, and lower values improve precision but may slow training. The maximum number of iterations sets how long the model trains; insufficient iterations cause underfitting, while too many waste resources or overfit.

To pull out characteristics from our dataset, the model assigns weights to the inputs. The weights signify the importance of the input data. This is called feature extraction.

The hidden layers are the layers that connect the input and output layers. Based on the size of the hidden layers, the number of connections varies. More neurons and layers (more complexity) allow for more insights between data points, but too many can result in overfitting. This is a hyperparameter that can be adjusted prior to the training.

After the model learns the weights, it can make the predictions based on the data. Each sample is classified as one of the different failure types according to the highest score determined during training.

4.2 Logistic Regression

Logistic Regression is used for binary or multinomial classification. Default parameters for this model are penalty='l2', C=1.0, solver='lbfgs', max_iter=100.

Logistic regression models the probability of a failure type using a logistic function applied to a weighted sum of input features. The coefficients act as feature weights, indicating which inputs most strongly influence the classification outcome.

Important hyperparameters include the regularization penalty (L1 or L2). L1 regularization shrinks some coefficients to zero, effectively performing feature selection, while L2 distributes penalties across all coefficients to prevent extreme weights. The regularization strength (C) determines how strongly coefficients are penalized: smaller values of C increase regularization, improving generalization but risking underfitting, while larger values of C reduce regularization, allowing the model to fit more closely to training data. The solver controls how the optimization problem is solved; for example, LBFGS works well for multinomial problems, while Liblinear is efficient for smaller binary tasks. Finally, the maximum iterations define how long the optimization continues, influencing whether the model converges to stable coefficients.

To pull out characteristics from our dataset, the model assigns coefficients to the inputs. The coefficients represent the influence of each feature on the classification decision. This serves as feature extraction and shows which input values are more important.

The decision function in logistic regression separates the classes with a linear boundary. More iterations during training help the model adjust the coefficients more precisely and avoid underfitting. This is a hyperparameter that can be adjusted prior to training.

After the model learns the coefficients, it can make predictions based on the probabilities. Each sample is classified as one of the different failure types according to the highest probability determined during training.

4.3 Decision Tree Classifier

Decision Tree Classifier is used for handling non-linear data and categorical features. Default parameters for this model are criterion='gini', max_depth=None, min_samples_split=2.

The decision tree classifier repeatedly splits the feature space to maximize class separation. Each internal node evaluates a feature and threshold, directing samples down branches until they reach a leaf node classified with a specific failure type.

Key hyperparameters include the maximum depth of the tree, which sets the longest decision path. Shallow trees generalize well but may underfit, while very deep trees can memorize training data and overfit. The minimum number of samples required to split a node controls how

detailed the splits can be; larger values prevent overly specific splits but may reduce accuracy. The criterion (such as Gini impurity or entropy) determines how splits are evaluated; different criteria affect how feature importance is measured. The minimum samples per leaf ensure that terminal nodes represent enough data to be reliable. Overall, these hyperparameters balance interpretability and generalization.

To pull out characteristics from our dataset, the model chooses features that reduce impurity at each split. The importance of the features is shown by how often they are used and how much they improve classification. This is called feature extraction.

The tree depth and number of splits determine how complex the model becomes. More depth allows the model to capture more patterns, but too much depth can result in overfitting. These are hyperparameters that can be adjusted prior to training.

After the model builds the tree, it can make predictions based on the decision path of the sample. Each sample is classified as one of the different failure types according to the leaf node reached during training.

4.4 K-Nearest Neighbors Classifier

KNeighbors Classifier is used for smaller datasets. Default parameters for this model are n_neighbors=5, weights='uniform', metric='minkowski'.

The k-nearest neighbors (KNN) classifier predicts the class of a new point based on the classes of its closest neighbors in the ninety-dimensional feature space. Distances are calculated between the new sample and all training samples, and the majority vote determines classification.

The main hyperparameter is the number of neighbors (k). Smaller k values create flexible and detailed decision boundaries that adapt closely to training data but are sensitive to noise. Larger k values produce smoother, more stable boundaries but may overlook local structure. The distance metric (such as Euclidean or Manhattan) determines how similarity is measured, and different metrics can emphasize different feature relationships. The weighting scheme can assign equal influence to all neighbors (uniform) or give closer neighbors more influence (distance-based), which often improves robustness.

To pull out characteristics from our dataset, the model compares the input to stored training examples. The importance of the features is reflected in the distance calculation, which determines similarity between samples. This is a form of feature extraction based on distance.

The choice of neighbors determines the complexity of the model. Fewer neighbors make the decision boundaries more sensitive, while more neighbors create smoother boundaries. This is a hyperparameter that can be adjusted prior to training.

After the model finds the nearest neighbors, it can make predictions based on the most common class among them. Each sample is classified as one of the different failure types according to the class majority of its neighbors.

4.5 XGBoost Classifier

XGBoost Classifier is used for handling complex, high-dimensional datasets and capturing nonlinear relationships between features. Default parameters for this model are learning_rate=0.3, n_estimators=100, max_depth=6, subsample=1.0, colsample_bytree=1.0, and objective='multi:softprob'.

The XGBoost classifier is an ensemble method based on gradient boosting, where decision trees are built sequentially and each tree corrects the errors of the previous ones. By focusing on difficult-to-classify samples, XGBoost achieves strong predictive accuracy.

This model has many hyperparameters that influence performance. The parameter gamma specifies the minimum loss reduction required for a split, making the algorithm more conservative as gamma increases. The min_child_weight sets the minimum sum of instance weights in a child node; higher values prevent small, overly specific splits that may capture noise. The number of estimators defines how many boosting rounds are used, with more trees typically improving accuracy at the cost of computation and overfitting risk. The learning rate adjusts how much each tree contributes to the final model; lower rates improve stability but require more estimators. Maximum depth controls how detailed each tree can be, with deeper trees modeling complex relationships but risking overfitting. Subsample and colsample_bytree introduce randomness by selecting fractions of rows and features, respectively, reducing correlation between trees and improving generalization. Regularization terms reg_lambda (L2) and reg_alpha (L1) penalize overly large weights, the former smoothing them and the latter encouraging sparsity. The n_jobs parameter determines the number of processor cores used during training, affecting computational efficiency.

To pull out characteristics from the dataset, XGBoost assigns weights and gradients to each feature to determine its influence on reducing classification error. This allows the model to focus on the most informative features, a form of feature extraction based on gradient boosting.

After the model learns from the sequence of boosted trees, it can predict the probability of each failure type and classify each sample according to the highest probability.

4.6 Random Forest Classifier

Random Forest Classifier is used for improving classification performance by combining the predictions of multiple decision trees. Default parameters for this model are n_estimators=100, criterion='gini', max_depth=None, and min_samples_split=2.

The random forest classifier constructs many decision trees on bootstrapped samples of the data and averages their predictions to produce the final output. This reduces variance and increases robustness compared to a single decision tree.

The hyperparameters of random forest affect the diversity and accuracy of the ensemble. The number of estimators sets the number of trees; more trees increase stability but require additional computation. The maximum number of features considered at each split controls tree diversity; fewer features increase variability across trees, improving generalization. Maximum depth restricts how detailed trees can become; shallow trees avoid overfitting, while deeper trees capture more complex relationships. The minimum number of samples required to split a node prevents the model from generating overly specific branches, while the minimum samples per leaf ensures that each terminal node contains enough samples to provide reliable predictions. Collectively, these parameters determine how well the random forest balances flexibility with generalization.

To pull out characteristics from the dataset, Random Forest evaluates feature importance by measuring how much each feature decreases impurity across all trees. This highlights which inputs are most influential for classification and acts as a form of feature extraction.

After the model aggregates the predictions from all trees, it classifies each sample as one of the different failure types according to the majority vote.

4.7 Hyperparameter Tuning

Hyperparameter tuning is a process of finding a different set of hyperparameters and running them through the model. This can be done by manually processing hyperparameters or using a grid search, which is an automated hyperparameter tuning method. The reason for using hyperparameter tuning is that hyperparameters directly control the model and allow the model to perform optimal results. Have manually processed hyperparameters via using random different values to run the model. Then using the grid search method to find the optimal hyperparameter for my model. This was done by specifying the list of hyperparameters and checking the performance via the accuracy of the output, then the algorithm runs through all the possible combinations to find the most optimal combination of hyperparameters. However, using the grid search method takes a long time to run the process since it is computationally intensive. On the other hand, manually tuning the hyperparameter might not be accurate, but it takes a short time to run the model.

5. Results

Table 1: XGBoost Model Training

Trials	Gamma	min_child_weight	n_estimators	max_depth	subsample	colsample_bytree	Test Accuracy	Train Accuracy
1 (Grid Search)	**0.1**	**1**	**100**	**3**	**0.8**	**0.7**	**0.94444**	**1**
2	0.1	1	100	1	0.8	0.7	0.83333	0.98571
3	0.1	5	100	1	0.8	0.7	0.83333	0.97143
4	0.1	5	100	1	0.6	0.7	0.83333	0.92857
5	0.1	5	100	1	0.6	0.5	0.88889	0.92857
6	0.1	5	300	1	0.6	0.5	0.88889	0.94286

5.1 XGBoost Classifier

The hyperparameters of reg_lambda, re_alpha, and n_jobs were constant, with the values of each being 1, 0.5, and 1. And these hyperparameters were not affecting the accuracy. As the max depth hyperparameter decreased, both train and test accuracy decreased. As min_child_weight increased, test accuracy remained constant, but test accuracy slightly decreased. As the subsample decreased, the test accuracy remained constant, but the train accuracy decreased. As colsample_bytree decreased, test accuracy increased, and the train accuracy remained constant. As n_estimator increased, the test accuracy remained constant, and the train accuracy increased slightly.

Table 2: K-Nearest Neighbors Model Training

Trial #	n_neighbors	weights	metric	Test Accuracy	Train Accuracy
1 (Grid Search)	3	uniform	euclidean	0.77777	0.9
2	2	uniform	euclidean	0.83333	0.914286
3	**2**	**distance**	**euclidean**	**0.83333**	**1**
4	2	uniform	manhattan	0.77778	0.9
5	5	uniform	euclidean	0.72222	0.85714

5.2 K-Nearest Neighbors Classifier

As the value of n_neighbors decreased, both the train and test accuracy increased. As weights changed from uniform to distance, test accuracy remained constant, but training accuracy increased to 1. As the distance metric changed from Euclidean to Manhattan, test accuracy slightly decreased, and training accuracy remained constant. As n_neighbors increased, both train and test accuracy decreased.

Table 3: MLP Model Training

Trial #	hidden_layer_sizes	activation	learning_rate	alpha	solver	max_iter	Test Accuracy	Train Accuracy
1 (Grid Search)	(22,23)	logistic	n/a	0.0001	lbfgs	200	0.77778	1.0
2	**(22,23)**	**logistic**	**n/a**	**0.0001**	**lbfgs**	**500**	**0.83333**	**1.0**
3	(22,23)	identity	n/a	0.0001	lbfgs	500	0.77778	1.0
4	(22,23)	identity	n/a	0.0001	adam	500	0.77778	0.94286
5	(22,23)	identity	n/a	0.05	adam	1500	0.83333	1.0
6	(22,23)	identity	Adaptive, invscaling	0.05	sgd	1500	0.83333	1.0

5.3 MLP Classifier

As the number of iterations increased, both train and test accuracy increased. As the activation function changed from logistic to identity, test accuracy slightly decreased, and training accuracy remained constant. As the solver changed from lbfgs to Adam, test accuracy remained constant, but training accuracy decreased slightly. As the learning rate increased, test accuracy slightly improved, and training accuracy remained constant.

Table 4: Logistic Regression Model Training

Trial #	C	Penalty	max_iter	solver	Test Accuracy	Train Accuracy
1 (Grid Search)	0.01	l2	4000	lbfgs	0.83334	1.0
2	**0.1**	**l2**	**4000**	**lbfgs**	**0.88888**	**1.0**
3	0.001	l2	4000	lbfgs	0.83334	0.97143
4	0.1	l2	4000	saga	0.72222	0.92857
5	0.0001	l2	4000	sag	0.77778	0.92857
6	0.0001	l2	4000	newton-cg	0.83334	0.97143

5.4 Logistic Regression

The Hyperparameter max_iter was setted constant of value of 4000. And changing this hyperparameter did not impact the accuracy. As the regularization parameter C increased, both train and test accuracy generally improved, indicating reduced regularization strength allowed the model to fit the data more closely. All trials used L2 regularization, which produced stable training accuracy across different solvers. When the solver changed from lbfgs to saga, sag, and newton-cg, train accuracy stayed relatively high, but test accuracy fluctuated slightly, with lbfgs and newton-cg yielding the best performance and saga showing the lowest.

Table 5: Random Forest Model Training

Trial #	n_estimators	max_features	max_depth	min_samples_split	Test Accuracy	Train Accuracy
1 (Grid Search)	100	sqrt	None	2	0.94444	1.0
2	100	sqrt	None	2	0.94444	0.98571
3	200	sqrt	None	2	0.94444	0.97142
4	**200**	**log2**	**None**	**2**	**0.94444**	**0.95714**
5	200	log2	10	2	0.94444	0.95714
6	200	log2	10	5	0.94444	0.95714

5.5 Random Forest Classifier

As n_estimators increased, both train and test accuracy remained constant. As max_features changed from sqrt to log2, both accuracies remained stable. As max_depth increased, both train and test accuracy remained constant. As min_samples_split increased, both accuracies remained constant.

Table 6: Decision Tree Model Training

Decision Tree Classifier						
Trial #	criterion	max_depth	min_samples_leaf	min_samples_split	Test Accuracy	Train Accuracy
1 (Grid Search)	**gini**	**None**	**4**	**2**	**0.77778**	**0.92857**
2	gini	10	4	2	0.77778	0.92857
3	gini	10	4	7	0.77778	0.92857
4	gini	10	10	7	0.66667	0.78571
5	entropy	10	10	7	0.44444	0.75714
6	gini	10	2	7	0.77778	0.97142

5.6 Decision Tree Classifier

As max_depth increased, test accuracy decreased, and training accuracy remained constant. As the criterion changed from gini to entropy, both train and test accuracy decreased. As min_samples_split and min_samples_leaf increased, both accuracies remained relatively constant.

Among the evaluated models, the Random Forest classifier demonstrated the best performance, achieving the highest test accuracy of 94.4 percent across multiple trials.

6. Discussion

The experimental results demonstrate that the six classifiers tested achieved different levels of accuracy in predicting robotic failure categories. The multilayer perceptron achieved a test accuracy of 94.4 percent, indicating a strong capability to capture nonlinear interactions among the ninety-dimensional input features. A comparable level of performance was obtained with the random forest classifier, which also reached 94.4 percent through the aggregation of multiple decision trees, and with XGBoost, which likewise produced 94.4 percent accuracy by applying sequential boosting with gradient-based optimization. In contrast, the k-nearest neighbors classifier achieved 88.9 percent, reflecting moderate success in classifying failure types based on similarity in feature space but with sensitivity to noise. Logistic regression performed at 88.9 percent, consistent with expectations for a linear model that cannot fully capture nonlinear dependencies, while the single decision tree classifier achieved the lowest performance at 77.8 percent, confirming its tendency to overfit when used without ensemble methods.

These findings align with theoretical expectations discussed in the background. Simpler linear models, such as logistic regression, establish a baseline but are limited by their linear decision boundaries. In contrast, ensemble methods like random forest and XGBoost, along with neural networks such as the multilayer perceptron, are better suited for learning the nonlinear mapping $f: \mathbb{R}^{90} \rightarrow \{C_1, C_2, \ldots, C_n\}$. The superior performance of these models demonstrates their effectiveness in modeling the complex relationships within force–torque sensor data.

Sources of error include overfitting, particularly for deeper neural networks and tree-based methods, where training accuracy approached 100 percent but testing accuracy was slightly lower. Preprocessing may also have introduced bias during data flattening, as temporal relationships were represented in a static format. Another limitation is that the dataset was collected under controlled conditions after induced failures, which may not fully reflect real-world variability or unexpected modes of robotic malfunction. The accuracy values themselves are subject to uncertainty due to the limited test set size; the 20 percent test split provides a reasonable estimate but may not capture the full performance distribution. Future work should employ k-fold cross-validation to produce more robust uncertainty estimates and consider augmenting the dataset with additional noisy or real-world measurements.

Despite these limitations, the comparative evaluation demonstrates that systematic hyperparameter tuning significantly improves performance and that ensemble and neural

network models clearly outperform simpler classifiers. These results confirm the feasibility of embedding AI-driven self-diagnosis into robotic systems to enhance operational reliability, reduce downtime, and minimize the need for human intervention.

Why things Happen

The experimental results show that model performance varied depending on complexity and hyperparameter settings. In logistic regression, increasing the regularization parameter C improved both training and test accuracy because weaker regularization allowed the model to fit the data more closely. All trials used L2 regularization, which kept the model stable, while differences among solvers such as lbfgs, saga, and newton-cg led to small changes in accuracy due to their different optimization methods.

Simpler models, including logistic regression, decision tree, and k-nearest neighbors, showed lower accuracy because they rely on linear or distance-based boundaries that cannot fully capture nonlinear patterns in the force–torque data. In contrast, random forest, XGBoost, and multilayer perceptron achieved higher accuracy since they can model complex relationships and reduce overfitting through ensemble or layered learning.

Overall, the results confirm that models with higher representational capacity perform better for robotic fault classification. However, the nearly perfect training accuracy in several models suggests mild overfitting, likely caused by limited data size and the loss of temporal information during preprocessing. Future improvements could include using regularization or time-aware models to enhance generalization.

7. Conclusions

This study confirmed the feasibility of enabling robots to perform accurate self-diagnosis using machine learning. The evaluation of multiple classifiers showed that advanced models such as ensemble methods and neural networks consistently outperform simpler approaches, meeting the project's objective of improving reliability in robotic failure detection.

What this means is that robotic systems can move closer to true autonomy by integrating AI-driven diagnostic tools, reducing downtime and dependence on human oversight. The broader implication is that these methods can strengthen operational safety and efficiency across industries where robotics is deployed, including healthcare, manufacturing, and service environments.

Future work should expand datasets, incorporate real-world noise, and test models under varied operational conditions. Exploring hybrid models that integrate the strengths of neural networks and ensembles may further enhance robustness and generalization, paving the way for more resilient self-monitoring robotic systems.

Acknowledgments

Special thanks to the UC Irvine KDD Archive for providing the dataset used in this study.

References

[1] https://archive.ics.uci.edu/dataset/138/robot+execution+failures
[2] https://archive.ics.uci.edu/