Fault Diagnosis in Analog Circuits Using Machine Learning and Deep Learning

A Detailed Report on Model Development and Evaluation

Author Name: Shaurya Sethi

Author Note

This report documents my independent research and implementation of **Fault Diagnosis in Analog Circuits using Machine Learning**. The project was entirely self-driven, including dataset preparation, model development, feature engineering, and evaluation.

Abstract

Fault diagnosis in analog circuits is a critical area of research, with recent literature increasingly advocating for deep learning-based methods, particularly **Convolutional Neural Networks (CNNs)**, due to their ability to automatically extract complex features from raw waveform data. However, deep learning models often function as **black boxes**, making their decisions less interpretable, and they require large amounts of labeled data for training. In contrast, traditional machine learning models, such as **tree-based algorithms**, offer greater **interpretability and efficiency** with significantly lower computational requirements.

This project aims to demonstrate the effectiveness of **traditional machine learning models**, specifically **Random Forest (RF) and Extreme Gradient Boosting (XGBoost)**, in **classifying faults in analog circuits**. Unlike existing ML-based research that relies on complex **feature extraction and engineering methods**, this study adopts a **simplistic statistical approach** for feature selection, proving that even basic feature extraction techniques can yield competitive results. The models were trained and evaluated on a dataset generated from circuit simulations, with faults systematically injected to create diverse failure scenarios.

XGBoost emerged as the best-performing model, achieving 94% accuracy, a performance level comparable to deep learning models, while retaining full interpretability. To further explore the trade-offs between **interpretability and accuracy**, a **Multi-Layer Perceptron (MLP)**, a deep learning model, was also trained using the same feature set. While the MLP demonstrated strong performance, it did not significantly outperform XGBoost, highlighting that **traditional feature engineering remains highly effective and that deep learning is not always necessary for achieving state-of-the-art results in this domain**.

The results suggest that tree-based ML models remain highly viable for **fault diagnosis in analog circuits**, providing a balance between accuracy and explainability. This work reinforces the **continued relevance of classical machine learning** in domains where interpretability and efficiency are key concerns, challenging the notion that deep learning is always the superior choice.

Keywords: *Analog Circuit Fault Diagnosis, Machine Learning, XGBoost, Random Forest, Multi-Layer Perceptron, Feature Engineering, Interpretability, Deep Learning Comparison*

Methodology

1. Circuit Design in Multisim

To generate data for fault diagnosis, a **Sallen-Key Low Pass Filter** circuit was designed in **Multisim**. This circuit topology was chosen due to its widespread use in signal processing applications and its sensitivity to various component faults. The Sallen-Key configuration allows for controlled frequency response analysis, making it an ideal test case for fault detection in analog circuits.

2. Simulating Normal and Faulty Circuit Behaviour

The designed circuit was simulated under both normal operating conditions and various fault conditions. The faults introduced include:

- **Biasing faults** – Incorrect biasing affecting transistor or op-amp operation.
- **Component drifts** – Resistors and capacitors with values deviating from nominal ratings.
- **Power faults** – Voltage supply deviations or complete loss of power.
- **Input signal-related faults** – Variations in signal amplitude, frequency, and waveform distortion.
- **Open resistor faults** – Complete failure of resistors leading to discontinuities.
- **Shorted capacitor faults** – Capacitor breakdown leading to bypassing of circuit stages.

Each of these fault conditions was simulated in Multisim, and the circuit's response was observed.

3. Data Acquisition

For each simulation, the time-domain waveform data was exported from **Multisim to Excel**. The recorded data included:

- **Time (t)**
- **Input voltage (Vin)**
- **Output voltage (Vout)**

Each simulation run generated a separate Excel file containing the raw waveform data.

4. Organizing Raw Data for Feature Extraction

To facilitate automated processing, the exported Excel files were systematically stored in a structured **folder system**, with separate directories for different fault types and normal operation data.
A **Python script** was then developed to:

- Iterate through the folders and extract relevant statistical and frequency-domain features from the raw waveform data.
- Store the extracted features in structured CSV files for further processing.

5. Merging Extracted Features into a Single Dataset

After feature extraction, another Python script was used to **merge all extracted features** into a single structured dataset. This step ensured that all fault categories and normal operations were combined into a unified format suitable for machine learning model training.

6. Data Augmentation

To enhance the dataset's robustness, a data augmentation script was implemented to:

- Increase the dataset size by generating synthetic variations of the existing data.
- Improve class balance by ensuring adequate representation of all fault types.
- Reduce overfitting risk by introducing slight variations in waveform characteristics.

7. Data Cleaning and Preprocessing in R

With the dataset prepared, further **data cleaning and preprocessing** were performed in **R**:

- **Handling missing values** – Checking and addressing any inconsistencies.
- **Feature selection** – Removing redundant or highly correlated features.
- **Feature scaling** – Standardizing feature distributions for improved model performance.
- **Dataset stratification** – Ensuring balanced class distribution where applicable.

8. Finalizing Two Different Datasets

Based on the preprocessing, two different datasets were finalized:

- **Dataset for training XGBoost and Random Forest models** (classical ML approach).
- **Dataset for training the MLP model** (deep learning approach).

9. Model Building

The project involved training three distinct models:

- **XGBoost and Random Forest** using **Scikit-learn** – These models were chosen for their robustness, interpretability, and proven effectiveness in classification problems.
- **Multi-Layer Perceptron (MLP) using PyTorch** – Deep learning approach to compare performance against traditional ML models.
- **Custom-built MLP (without PyTorch or TensorFlow)** – Implemented from scratch to demonstrate low-level ML coding proficiency.

The model-building process was iterative, involving multiple rounds of:

- Training models on the dataset.
- Analysing feature importance.
- Refining feature selection based on performance insights.
- Retraining models with improved feature subsets.

10. Model Testing, Validation, and Robustness Analysis

After finalizing the models, rigorous validation and testing were performed:

- **Cross-validation** – Ensuring model consistency across different data splits.
- **Evaluation metrics** – Accuracy, precision, recall, and F1-score.
- **Robustness checks** – Testing model performance on unseen fault scenarios.
- **Generalization assessment** – Ensuring the model does not overfit specific faults and can classify faults reliably in different circuit conditions.

The results demonstrated that:

- **XGBoost achieved 94% accuracy**, which is on par with deep learning models while maintaining interpretability.
- **Random Forest also performed well**, showing high classification accuracy.
- **MLP provided competitive results**, but with the trade-off of being a black-box model with lower interpretability.

- The comparison of these models highlights the effectiveness of **traditional feature engineering with tree-based ML models**, achieving deep-learning-level performance **without sacrificing interpretability**.

Circuit Design and Data Collection

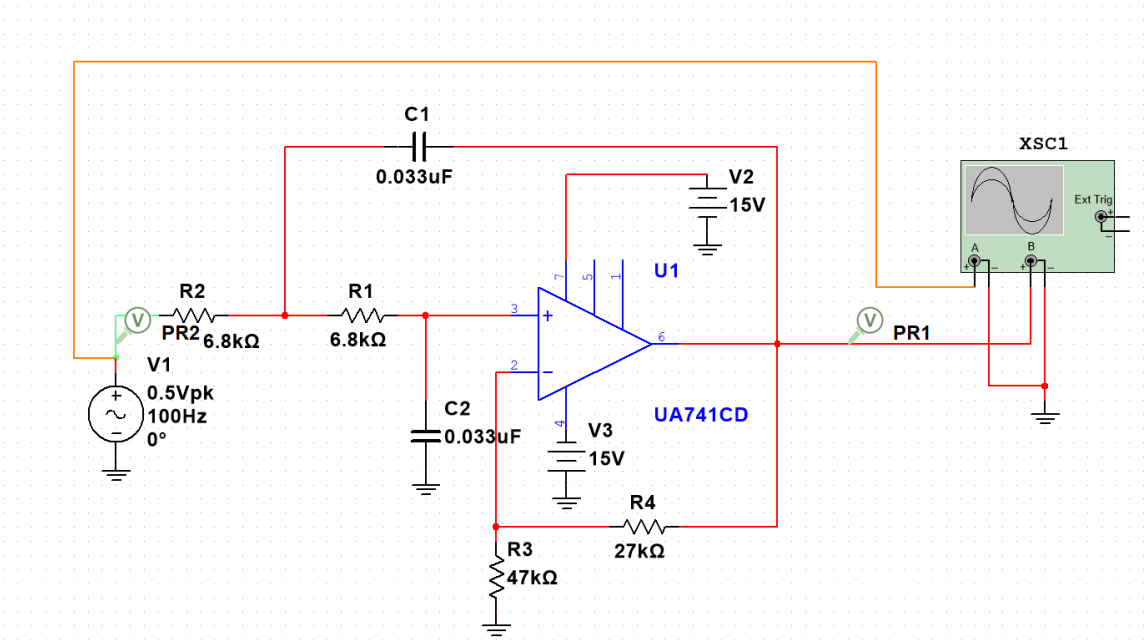
The Sallen-Key Low Pass Filter

For this project, we designed and simulated a **Sallen-Key low-pass filter** using **Multisim** to study its behaviour under different conditions, including normal operation and various fault scenarios. The **Sallen-Key topology** was chosen due to its simplicity, stability, and effectiveness in filtering high-frequency noise while allowing low-frequency signals to pass.
The circuit consists of:

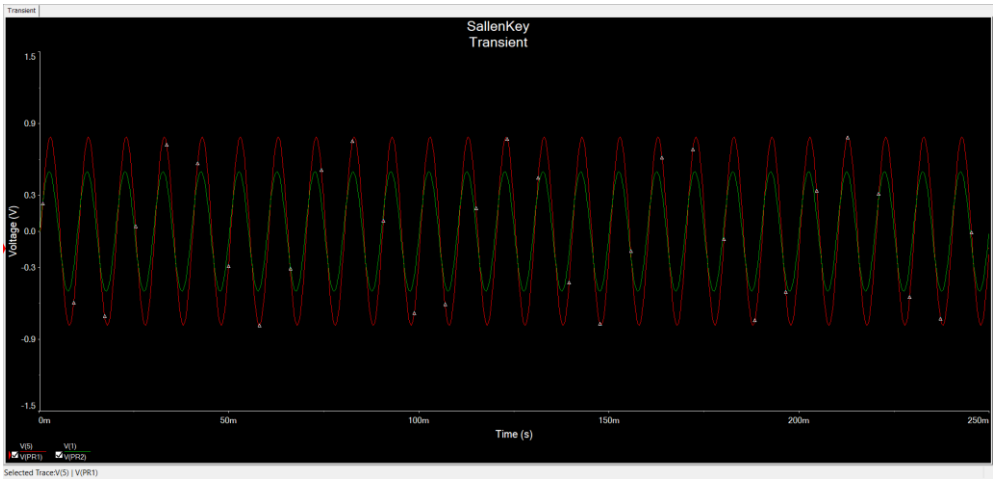
- An **operational amplifier (Op-Amp)** (**UA741CD**) configured in a unity-gain second-order low-pass filter configuration.
- A combination of **resistors (R1, R2, R3, R4)** and **capacitors (C1, C2)** to set the cutoff frequency.
- A **power supply** providing ±15V to the Op-Amp.
- An **AC input source** of 100 Hz with a peak voltage of 0.5V.

The expected behaviour of the circuit in normal conditions is a filtered output where high-frequency components are attenuated while low-frequency signals pass with minimal distortion. However, under faulty conditions, the circuit response significantly deviates from the expected behaviour.

Circuit Diagram



Below is a sample waveform obtained on running transient analysis on the circuit under normal operation.



Multiple simulations were conducted with slight variations in component values (±5% deviations from their nominal range) to effectively capture the natural variance present in normal operation.

Fault injections

The following fault modes were finalized, and faults injected into the circuit to capture the behavior of the circuit under abnormal conditions:

Fault Type	Component Modification	# of Simulations	Simulation Variations
Open Resistor (R1 or R2)	Replace **R1** or **R2** with **1MΩ, 5MΩ, 10MΩ** (open circuit).	**3 per resistor → 6 total**	**R1 open (3 sims), R2 open (3 sims)**
Shorted Capacitor (C1 or C2)	Replace **C1** or **C2** with a **1Ω short-circuit equivalent**.	**2 total**	**C1 shorted (1 sim), C2 shorted (1 sim)**
Component Drift (R1, R2, C1, C2)	Increase/Decrease each by **±50%**.	**8 total**	**R1 (+50%, -50%), R2 (+50%, -50%), C1 (+50%, -50%), C2 (+50%, -50%)**
Op-Amp Failure	(1) **Open Loop Failure** → Disconnect feedback. (2) **Shorted Output** → Connect **Vout** to GND.	**2 total**	**Open-loop (1 sim), Shorted output (1 sim)**

Input Signal Fault	(1) Increase/Decrease **amplitude** by **±50%**. (2) Increase/Decrease **frequency** by **×2, ×0.5**.	**4 total**	**Amplitude (+50%, -50%) = 2 sims, Frequency (×2, ×0.5) = 2 sims**
Biasing Fault (R3 and R4)	1) **Open R3 or R4** (5MΩ open circuit). (2) **Drift**: Increase/Decrease **R3** and **R4** by **±50%**.	**6 total**	**R3 open (1 sim), R4 open (1 sim), R3 drift (+50%, -50%) = 2 sims, R4 drift (+50%, -50%) = 2 sims**

For each simulation, the time-domain waveform data was exported from **Multisim to Excel**. The recorded data included:

- **Time (t)**
- **Input voltage (Vin)**
- **Output voltage (Vout)**

Each simulation run generated a separate Excel file containing the raw waveform data.

Data Collection

To facilitate automated processing, the exported Excel files were systematically stored in a structured **folder system**, with separate directories for different fault types and normal operation data.
Python scripts were then developed to:

- Iterate through the folders and structure the data into a uniform format for all files.

```
 1. import pandas as pd
 2. import numpy as np
 3. from scipy.interpolate import interp1d
 4.
 5. # Load raw waveform data
 6. df = pd.read_csv("raw_file.csv")
 7.
 8. # Ensure correct column order: Time | Vin | Vout
 9. df = df[["Time", "Vin", "Vout"]].sort_values(by="Time").reset_index(drop=True)
10.
11. # Interpolation to ensure exactly 1000 data points
12. NUM_POINTS = 1000
13. interp_time = np.linspace(df["Time"].min(), df["Time"].max(), NUM_POINTS)
14. interp_vin = interp1d(df["Time"], df["Vin"], kind='linear',
fill_value="extrapolate")(interp_time)
15. interp_vout = interp1d(df["Time"], df["Vout"], kind='linear',
fill_value="extrapolate")(interp_time)
16.
17. # Create cleaned dataframe
18. cleaned_df = pd.DataFrame({"Time": interp_time, "Vin": interp_vin, "Vout": interp_vout})
19.
20. # Save to CSV
21. cleaned_df.to_csv("cleaned_file.csv", index=False)
```

Click here to view the full script

- Augment the restructured files to improve the quality and quantity of data.

```
# ------------------ Data Augmentation Functions ------------------
def add_gaussian_noise(signal, noise_level=0.01):
    """Adds Gaussian noise to a signal."""
    noise_std = noise_level * np.mean(np.abs(signal))
    noise = np.random.normal(0, noise_std, size=signal.shape)
    return signal + noise

def apply_amplitude_scaling(signal, scale_min=0.95, scale_max=1.05):
    """Scales the amplitude of a signal by a random factor."""
    scale_factor = np.random.uniform(scale_min, scale_max)
    return signal * scale_factor

def apply_time_shift(time, max_shift=0.01):
    """Applies a small time shift to the signal."""
    shift = np.random.uniform(-max_shift, max_shift)
    return time + shift

def augment_raw_data(time, vin, vout):
    """
    Applies augmentation techniques to raw signals.
    Returns augmented (time, vin, vout).
    """
    vin_aug = apply_amplitude_scaling(add_gaussian_noise(vin))
    vout_aug = apply_amplitude_scaling(add_gaussian_noise(vout))
    time_aug = apply_time_shift(time)
    return time_aug, vin_aug, vout_aug

# ------------------ Augment and Save Data ------------------
for cleaned_file in cleaned_files:
    time_arr, vin_arr, vout_arr = df["Time"].values, df["Vin"].values, df["Vout"].values
    for i in range(NUM_AUGMENTS):
        time_aug, vin_aug, vout_aug = augment_raw_data(time_arr, vin_arr, vout_arr)
        df_aug = pd.DataFrame({"Time": time_aug, "Vin": vin_aug, "Vout": vout_aug})
        df_aug.to_csv(f"augmented_{cleaned_file}_v{i+1}.csv", index=False)

print("☑ Data Augmentation Completed!")
```

Click here to view the full script

- Iterate through the folders and extract relevant statistical features from the raw waveform data.

```
1. 1. import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis

# ------------------ Feature Extraction Functions ------------------
def compute_features(signal):
    """Extracts statistical time-domain features from a signal."""
    return {
        "mean": np.mean(signal),
        "std": np.std(signal),
        "max": np.max(signal),
        "min": np.min(signal),
        "median": np.median(signal),
        "ptp": np.ptp(signal),  # Peak-to-peak
        "skewness": skew(signal) if np.std(signal) > 0 else 0.0,
        "kurtosis": kurtosis(signal) if np.std(signal) > 0 else 0.0,
        "rms": np.sqrt(np.mean(signal ** 2)),  # Root Mean Square
```

```
        "zcr": np.sum(np.abs(np.diff(np.sign(signal)))) / (2 * len(signal))
    }

# ------------------ Processing a Single File ------------------
def process_file(file_path):
    """Reads a file, extracts Vout features, and saves them."""
    df = pd.read_csv(file_path)
    if not all(col in df.columns for col in ["Time", "Vin", "Vout"]):
        print(f"Skipping {file_path}: Missing required columns.")
        return

    features = compute_features(df["Vout"].to_numpy())  # Extract from Vout
    df_features = pd.DataFrame([features])
    df_features.to_csv(file_path.replace(".csv", "_features.csv"), index=False)
    print(f"✅ Extracted features for {file_path}")

print("🔹 Time-Domain Feature Extraction Completed!")
```

Click here to view the full script

An Image representing the first few samples of our dataset

mean	std	max	min	median	ptp	skewness	kurtosis	rms	zcr	label	source	file
0.001112	0.347274	0.492841	-0.4907	0.00069	0.983541	-0.00031	-1.49858	0.347276	0.0996	SK_Biasing	cleaned_fi	cleaned_SallenKey_R3_5M.csv
-0.0002	0.766656	1.087521	-1.0876	-0.00025	2.175119	0.000217	-1.4984	0.766656	0.0996	SK_Biasing	cleaned_fi	cleaned_SallenKey_R3_drift_neg.csv
0.000681	0.484081	0.687171	-0.68571	0.000247	1.372885	-7.9E-05	-1.49849	0.484081	0.0996	SK_Biasing	cleaned_fi	cleaned_SallenKey_R3_drift_pos.csv
-0.07369	12.8512	13.00836	-13.0106	-2.04368	26.01894	0.012871	-1.98113	12.85141	0.100601	SK_Biasing	cleaned_fi	cleaned_SallenKey_R4_5M.csv
-0.00029	0.449153	0.63488	-0.63551	-5.7E-05	1.270394	-0.00014	-1.49851	0.449154	0.0996	SK_Biasing	cleaned_fi	cleaned_SallenKey_R4_drift_neg.csv
0.001201	0.660314	0.934392	-0.93192	0.000841	1.866307	0.000137	-1.49842	0.660315	0.0996	SK_Biasing	cleaned_fi	cleaned_SallenKey_R4_drift_pos.csv
0.001186	0.346205	0.497816	-0.49315	0.001831	0.990971	-0.00144	-1.49737	0.346207	0.099099	SK_Biasing	augmente	cleaned_SallenKey_R3_5M_aug1.csv
0.001173	0.346093	0.497954	-0.49839	0.002402	0.996348	-0.00206	-1.49818	0.346095	0.099099	SK_Biasing	augmente	cleaned_SallenKey_R3_5M_aug10.csv
0.00102	0.348559	0.498832	-0.49892	0.000715	0.997752	-0.00016	-1.49839	0.348561	0.1001	SK_Biasing	augmente	cleaned_SallenKey_R3_5M_aug11.csv

Phase 1: Initial Data Preprocessing and Modeling

The **data preprocessing** phase was critical in ensuring a **high-quality dataset** for training machine learning models. The following steps were undertaken using **R** to clean, analyze, and prepare the extracted time-domain features for modeling. The entire preprocessing workflow involved **handling missing values, correlation analysis, feature selection, feature engineering, outlier detection, and feature scaling** before finalizing the dataset.

1. Data Cleaning & Handling Missing Values

The dataset was imported from an **Excel file** using read_xlsx(), and the **initial data inspection** was performed to check for missing values and assess the dataset structure.

Checking for Missing Values

```
colSums(is.na(df))  # Count missing values in each column
missmap(df)         # Visual representation of missing values
```

Converting Categorical Data

The label column, representing fault types, was converted into a **factor variable** for appropriate handling in modeling.

```
df$label <- as.factor(df$label)  # Convert label column to categorical
```

Removing Redundant Columns

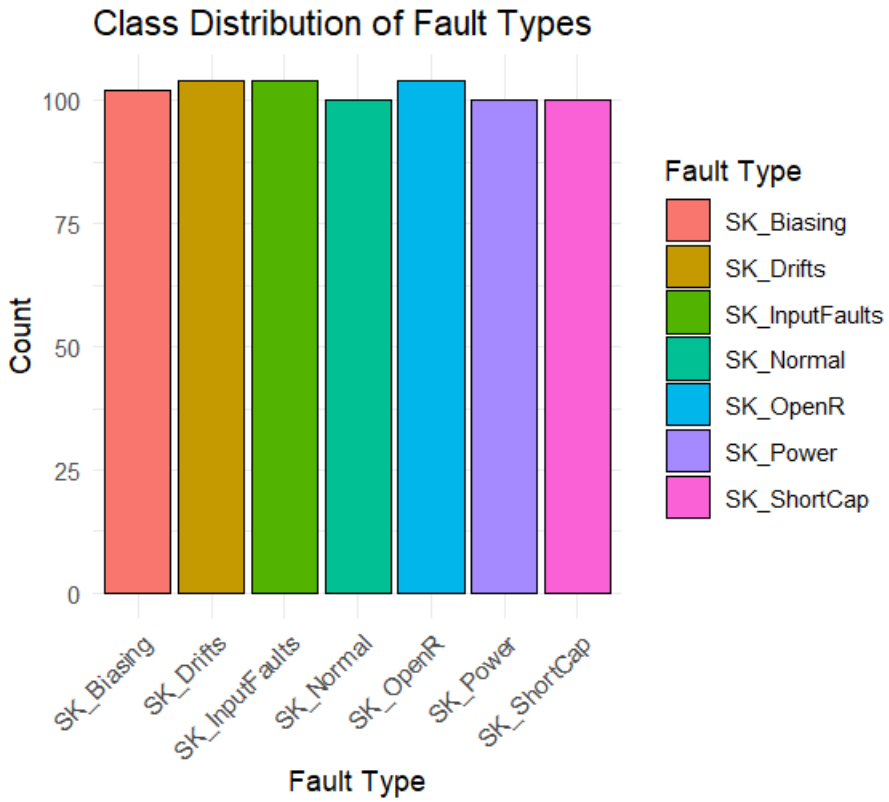
Columns such as source and file were dropped as they were unnecessary for model training.

```
1. df <- df %>% select(-source, -file)
```

Checking Class Distribution

To ensure balanced data representation, we visualized the **fault type distribution** using a bar plot.

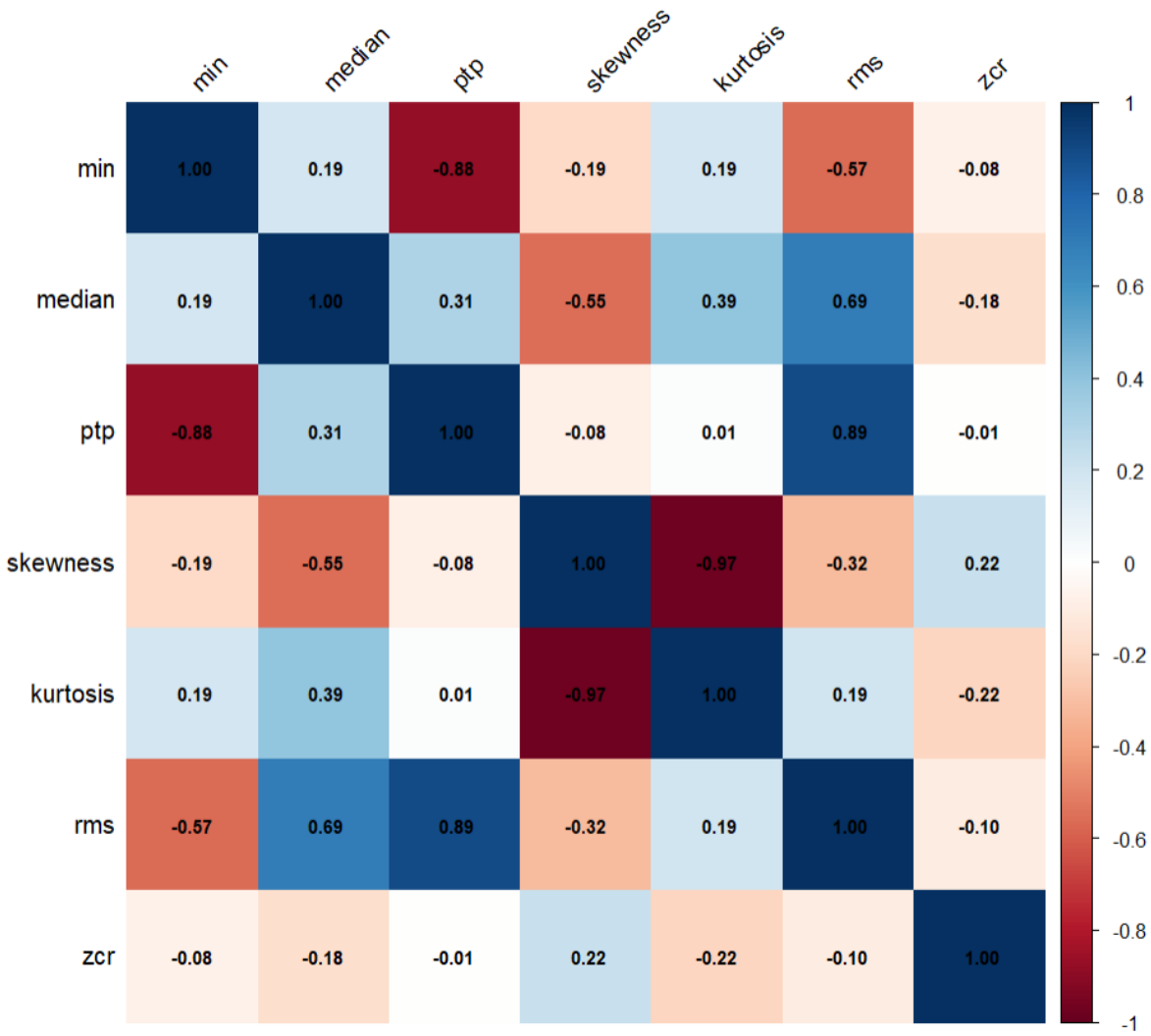
```
 1. df %>%
 2.   group_by(label) %>%
 3.   summarise(Count = n()) %>%
 4.   arrange(desc(Count))
 5.
 6. # Visualization
 7. ggplot(df, aes(x = label, fill = factor(label))) +
 8.   geom_bar(color = "black") +
 9.   theme_minimal() +
10.   labs(title = "Class Distribution of Fault Types",
11.        x = "Fault Type", y = "Count", fill = "Fault Type") +
12.   theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



2. Correlation Analysis & Feature Selection

A **correlation matrix** was generated to identify redundant or highly correlated features.

```
1. cor_matrix <- cor(df %>% select(-label))
2. corrplot::corrplot(cor_matrix,
3.                    method = "color",
4.                    type = "full",
5.                    tl.col = "black",
6.                    tl.srt = 45,
7.                    diag = TRUE,
8.                    addCoef.col = "black",
9.                    number.cex = 0.75)
```



Feature Removal Based on Correlation

- **Removed Features:**
  - mean: Nearly identical to median and rms
  - max and std: Captured by ptp (Peak-to-Peak)

```
1. df <- df %>% select(-mean, -max, -std)
```

3. Feature Engineering: Creating a New Feature

- **New Feature:** skew_kurt_ratio = skewness / kurtosis
- This condenses **two highly correlated features** into a **single representative metric**.

```
1. df <- df %>%
2.   mutate(skew_kurt_ratio = skewness / kurtosis) %>%
3.   select(-skewness, -kurtosis)
```

After adding the skew_kurt_ratio feature, we observed **25 missing values**, caused by division errors (kurtosis = 0).

Fixing Missing Values

Instead of removing these samples, a **small constant (1e-6)** was added to kurtosis before division to avoid undefined values.

```
# Recalculate `skew_kurt_ratio` while preventing division by zero
df <- df %>%
  mutate(skew_kurt_ratio = skewness / (kurtosis + 1e-6))

colSums(is.na(df)) # no more missing values

df <- df %>% select(-skewness, -kurtosis) # removing them again
glimpse(df)
```

4. Saving the dataset

The dataset was first saved to enable early-stage experimentation with Random Forest and XGBoost models.

```
write.csv(df_scaled, "preprocessed_time_features_extracted_dataset.csv", row.names = FALSE)
```

*The full R script for this preprocessing pipeline can be accessed **here**.*

The first few samples of the initial training dataset for the tree-based models are shown below

min	median	ptp	rms	zcr	skew_kurt_	label
-0.4907	0.00069	0.983541	0.347276	0.0996	0.000204	SK_Biasing
-1.0876	-0.00025	2.175119	0.766656	0.0996	-0.00015	SK_Biasing
-0.68571	0.000247	1.372885	0.484081	0.0996	5.25E-05	SK_Biasing
-1.84656	-0.00986	4.388932	1.647569	0.100601	-0.0065	SK_Biasing
-0.63551	-5.66E-05	1.270394	0.449154	0.0996	9.05E-05	SK_Biasing
-0.93192	0.000841	1.866307	0.660315	0.0996	-9.16E-05	SK_Biasing
-0.49315	0.001831	0.990971	0.346207	0.099099	0.000961	SK_Biasing
-0.49839	0.002402	0.996348	0.346095	0.099099	0.001372	SK_Biasing
-0.49892	0.000715	0.997752	0.348561	0.1001	0.000107	SK_Biasing

5. XGBoost Model Implementation

After preprocessing the dataset, an XGBoost classifier was developed to classify faults in the circuit. XGBoost (Extreme Gradient Boosting) is a powerful tree-based machine learning algorithm that excels in handling structured data and provides feature importance analysis, making it an excellent choice for interpretability.

The model was trained on the preprocessed time-domain feature dataset, where:

- The features (X) were derived from statistical transformations of the circuit's output signals.
- The labels (y) represented different fault types in the circuit.
- The dataset was split into training (80%) and testing (20%) subsets to evaluate generalization.
- Label encoding was applied to transform categorical labels into numerical values.

The XGBoost model was configured with:

- `max_depth=6` (controls tree complexity),
- `eta=0.3` (learning rate for boosting),
- `n_estimators=100` (number of boosting rounds),
- `objective="multi:softmax"` (since the task is multi-class classification).

A classification report was generated to assess precision, recall, and F1-score for each fault type.

Code Snippet: XGBoost Model Implementation

```
# Import necessary libraries
import pandas as pd
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report

# Load dataset from CSV
df =
pd.read_csv(r"C:\Users\shaur\OneDrive\Documents\preprocessed_time_features_extracted_dataset.csv")

# Separate features and labels
X = df.drop(columns=["label"])  # Features
y = df["label"]  # Target variable

# Encode labels to numerical values
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42,
stratify=y)

# Train XGBoost model
xgb_model = xgb.XGBClassifier(objective="multi:softmax", num_class=len(set(y_train)),
                              max_depth=6, eta=0.3, n_estimators=100)
xgb_model.fit(X_train, y_train)

# Make predictions
y_pred_xgb = xgb_model.predict(X_test)

# Evaluate performance
accuracy_xgb = accuracy_score(y_test, y_pred_xgb)
print(f"XGBoost Accuracy: {accuracy_xgb:.4f}")

# Detailed classification report
print(classification_report(y_test, y_pred_xgb, target_names=label_encoder.classes_))
```

XGBoost Model Results

Classification Report:

Fault Type	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	0.82	0.90	0.86	20
SK_Drifts	0.70	0.67	0.68	21
SK_InputFaults	1.00	1.00	1.00	21
SK_Normal	0.50	0.50	0.50	20
SK_OpenR	1.00	1.00	1.00	21
SK_Power	0.65	0.85	0.74	20
SK_ShortCap	0.85	0.55	0.67	20

Train-Test Split:

- Training Set Shape: (571, 6)
- Testing Set Shape: (143, 6)

Model Performance:

- XGBoost Accuracy: 0.7832

Overall Metrics:

- **Accuracy:** 0.78
- **Macro Average:**
  - **Precision:** 0.79
  - **Recall:** 0.78
  - **F1-Score:** 0.78
- **Weighted Average:**
  - **Precision:** 0.79
  - **Recall:** 0.78
  - **F1-Score:** 0.78

Click here to view the full script for building and training this model.

6. Random Forest Model Implementation

To further explore the effectiveness of classical machine learning models in fault classification, a **Random Forest classifier** was implemented alongside XGBoost. Random Forest is an ensemble learning technique that constructs multiple decision trees and aggregates their outputs to improve classification performance and reduce overfitting.
The dataset was **preprocessed** as described earlier, ensuring numerical encoding for categorical labels and a balanced class distribution. The **Random Forest classifier** was configured with **500 trees (estimators)** and "sqrt" as the maximum number of features considered at each split. This ensures an optimal trade-off between performance and generalization.
The model was trained using the **train-test split** (80-20%), and its performance was evaluated using **accuracy and a detailed classification report**.

Code Snippet - Random Forest Classifier

```
 1. import pandas as pd
 2. from sklearn.model_selection import train_test_split
 3. from sklearn.preprocessing import LabelEncoder
 4. from sklearn.ensemble import RandomForestClassifier
 5. from sklearn.metrics import accuracy_score, classification_report
 6.
 7. # Load dataset
 8. df =
pd.read_csv(r"C:\Users\shaur\OneDrive\Documents\preprocessed_time_features_extracted_dataset.csv")
 9.
10. # Separate features and labels
11. X = df.drop(columns=["label"])  # Features
12. y = df["label"]  # Target variable
13.
14. # Encode labels numerically
15. label_encoder = LabelEncoder()
16. y = label_encoder.fit_transform(y)
17.
18. # Train-Test Split (80% Training, 20% Testing)
19. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42,
stratify=y)
20.
```

```
# Initialize Random Forest Classifier
rf_model = RandomForestClassifier(n_estimators=500, max_features="sqrt", random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions
y_pred_rf = rf_model.predict(X_test)

# Evaluate model performance
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Random Forest Accuracy: {accuracy_rf:.4f}")

# Detailed classification report
print(classification_report(y_test, y_pred_rf, target_names=label_encoder.classes_))
```

Click here to view the full script for building and training this model.

Random Forest Model - Performance Results

Classification Report:

Fault Type	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	0.85	0.85	0.85	20
SK_Drifts	0.69	0.86	0.77	21
SK_InputFaults	1.00	1.00	1.00	21
SK_Normal	0.60	0.45	0.51	20
SK_OpenR	1.00	1.00	1.00	21
SK_Power	0.67	0.90	0.77	20
SK_ShortCap	0.85	0.55	0.67	20

Train-Test Split:

- Train Shape: (571, 6)
- Test Shape: (143, 6)

Model Performance:

- Random Forest Model Accuracy: 0.8042 (80.42%)

Overall Metrics:

Accuracy: 0.80
Macro Average:

- Precision: 0.81
- Recall: 0.80
- F1-Score: 0.79

Weighted Average:

- Precision: 0.81
- Recall: 0.80
- F1-Score: 0.80

Overall Model Performance

Both models demonstrated strong classification performance, with Random Forest achieving an accuracy of **80.42%** and XGBoost slightly lower at **78.32%**. Despite their comparable performance, certain patterns emerged when analyzing individual fault classifications.

Key Observations and Inferences

- **Both models struggled with SK_Normal classification**
  - SK_Normal had the lowest recall and F1-score across both models (**50% for XGBoost and 51% for Random Forest**). This suggests that the Normal operation class shares characteristics with faulty cases, making it difficult to distinguish.
- **Random Forest showed better generalization**
  - The **macro F1-score for Random Forest was 0.79 compared to 0.78 for XGBoost**, suggesting better-balanced performance across all classes.
  - Random Forest outperformed XGBoost in **SK_Drifts and SK_Power**, where it exhibited higher recall and sensitivity to these faults.
- **XGBoost demonstrated competitive performance**
  - XGBoost delivered a similar classification quality, with **strong precision** in most fault categories.
  - However, it had slightly lower recall for SK_Drifts and SK_Power, indicating that some true faulty cases were misclassified as normal.
- **Perfect classification for SK_InputFaults and SK_OpenR**
  - Both models classified **SK_InputFaults and SK_OpenR with 100% accuracy**, indicating that the feature selection and preprocessing techniques were highly effective in distinguishing these fault types.

Identifying Class Imbalance and Misclassification Trends

During the initial phase of modeling, both XGBoost and Random Forest exhibited significant misclassification of the SK_Normal class. A deeper analysis of class distribution revealed a potential cause—while SK_Normal had **100 samples**, the cumulative number of fault case samples was around **600**. From this perspective, the dataset was **imbalanced**, leading to poor recall and F1-score for the SK_Normal class.

Applying Controlled SMOTE for Balance Correction

To mitigate this imbalance, **controlled SMOTE** was applied to artificially generate additional SK_Normal samples. The augmentation strategy increased the SK_Normal class count to **200**, while keeping the fault case samples at approximately **80 per class**. The hypothesis was that balancing the dataset would improve classification accuracy and recall for SK_Normal, while maintaining overall model performance.

Code snippet

```
1. from imblearn.over_sampling import SMOTE
2. from sklearn.model_selection import train_test_split
3. from sklearn.preprocessing import LabelEncoder
4. import pandas as pd
5. import numpy as np
```

```
 6.
 7. # Load preprocessed dataset
 8. df = pd.read_csv(preprocessed_time_features_extracted_dataset.csv)
 9.
10. # Separate features and labels
11. X = df.drop(columns=["label"])
12. y = df["label"]
13.
14. # Encode labels
15. label_encoder = LabelEncoder()
16. y = label_encoder.fit_transform(y)
17.
18. # Train-test split before SMOTE
19. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
random_state=42)
20.
21. # Check original class distribution
22. print("Original Class Distribution:", dict(zip(*np.unique(y_train, return_counts=True))))
23.
24. # Apply SMOTE to balance SK_Normal class
25. smote = SMOTE(sampling_strategy={label_encoder.transform(['SK_Normal'])[0]: 200},
random_state=42)
26. X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
27.
28. # Convert resampled data into DataFrame
29. df_resampled = pd.DataFrame(X_resampled, columns=X_train.columns)
30. df_resampled["label"] = label_encoder.inverse_transform(y_resampled)
31.
32. # Save the balanced dataset
33. df_resampled.to_csv("smote_balanced_TD_features.csv", index=False)
34. print("SMOTE-balanced dataset saved successfully!")
```

Click here to view the full script.

Performance Comparison: Before vs. After SMOTE

Random Forest Performance Improvement

Model	*Accuracy*	*SK_Normal Recall*	*SK_Normal F1-Score*
RF v1 (Before SMOTE)	80.42%	45%	51%
RF v2 (After SMOTE)	91.37%	90%	86%

Key Improvements:

- SK_Normal recall increased dramatically from **45% → 90%**
- Overall accuracy improved from **80.42% → 91.37%**
- Other fault classes retained strong classification performance

XGBoost Performance Improvement

Model	*Accuracy*	*SK_Normal Recall*	*SK_Normal F1-Score*
XGBoost v1 (Before SMOTE)	78.32%	50%	50%
XGBoost v2 (After SMOTE)	87.05%	88%	86%

Key Improvements:

- SK_Normal recall increased from **50% → 88%**
- Overall accuracy improved from **78.32% → 87.05%**
- The model achieved a better balance across all fault classifications

Final Insights and Takeaways

- **Data imbalance had a significant impact on classification performance**, particularly for the Normal operation class.
- **Applying controlled SMOTE** was an effective solution, significantly improving recall and accuracy without overfitting.
- **Random Forest exhibited greater gains post-SMOTE compared to XGBoost**, reinforcing its robustness for fault classification tasks.
- **These findings validate the importance of preprocessing techniques** like data augmentation and balancing before model training.

Feature Importance Analysis and Optimization

To gain insights into how different features contribute to fault classification, a Python script was developed to analyze feature importance for both **Random Forest (RF)** and **XGBoost** models. The results of this analysis are summarized in the table below.

```
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Load dataset
df = pd.read_csv(r"path to your file")

# Prepare features and labels
X = df.drop(columns=["label"])
y = df["label"]

# Train Random Forest Model
rf_model = RandomForestClassifier(n_estimators=500, max_features="sqrt", random_state=42)
rf_model.fit(X, y)
rf_importance = rf_model.feature_importances_

# Train XGBoost Model
xgb_model = xgb.XGBClassifier(objective="multi:softmax", num_class=len(set(y)),
                              max_depth=6, eta=0.3, n_estimators=100)
xgb_model.fit(X, y)
xgb_importance = xgb_model.feature_importances_

# Convert feature importance into DataFrame
feature_names = X.columns
rf_importance_df = pd.DataFrame({"Feature": feature_names, "Importance":
rf_importance}).sort_values(by="Importance", ascending=False)
xgb_importance_df = pd.DataFrame({"Feature": feature_names, "Importance":
xgb_importance}).sort_values(by="Importance", ascending=False)

# Plot Feature Importance - Random Forest
plt.figure(figsize=(10, 5))
sns.barplot(x="Importance", y="Feature", data=rf_importance_df, palette="Blues_r")
plt.title("Feature Importance (Random Forest)")
plt.show()
# Plot Feature Importance - XGBoost
plt.figure(figsize=(10, 5))
sns.barplot(x="Importance", y="Feature", data=xgb_importance_df, palette="Oranges_r")
plt.title("Feature Importance (XGBoost)")
plt.show()
```

Feature Importance Comparison (RF vs. XGBoost)

Feature	*RF Importance*	*RF Rank*	*XGBoost Importance*	*XGBoost Rank*
zcr	0.1691	2nd	0.3438	1st
min	0.2189	1st	0.0998	5th
ptp	0.1629	3rd	0.1342	4th
median	0.1534	4th	0.1727	2nd
skew_kurt_ratio	0.1480	5th	0.1616	3rd
rms	0.1473	6th	0.0876	6th

Key Takeaways from Feature Importance Analysis

- **"zcr" (Zero-Crossing Rate) is the most important feature in XGBoost** and ranks **2nd in Random Forest**, confirming its critical role in fault classification.
- **"min" is the most important feature in RF but ranks significantly lower in XGBoost**, indicating that feature importance may be model-dependent.
- **"rms" is the least important feature in both models**, making it a strong candidate for removal.
- **"ptp", "median", and "skew_kurt_ratio" show medium importance, but their rankings vary between RF and XGBoost**, highlighting differences in how each model prioritizes features.

Feature Removal Test – Assessing the Impact of Removing 'rms'
To optimize model performance, the **"rms"** feature was removed, and the models were re-evaluated under version **3**.

Performance After Feature Removal (RF v3 & XGBoost v3)

- **Removing "rms" did not negatively impact performance**—model accuracy remained stable or showed minor improvements.
- **SK_Normal recall remained at 90% in both models**, ensuring that fault detection was not compromised.
- **XGBoost v3 showed a slight accuracy improvement** (from **87.05% → 88.49%**), validating the removal of "rms" as a beneficial change.
- **Random Forest v3 experienced a minor drop in accuracy** (from **91.37% → 90.65%**), but the difference is within an acceptable range.

These findings demonstrate that **strategic feature selection can refine model performance** without compromising classification accuracy.

Phase 2: Advanced Feature Engineering and Refinement

In this phase, additional features were engineered, redundant features were eliminated, and dataset-specific preprocessing was applied for Random Forest (RF), XGBoost (XGB), and MLP models. This step was necessary to further enhance model performance and generalization.

1. Feature Engineering: Generating New Features To improve the dataset's expressiveness, new features were derived from existing ones. These transformations aimed to capture statistical properties of the waveform data.

Newly Engineered Features

- **Skew-Kurtosis Ratio**: skewness / kurtosis
- **Variance** Approximation: rms^2 - $median^2$

- **Crest Factor**: max / rms
- **Shape Factor**: rms / |median|
- **Impulse Factor**: max / |median|

```
df_2 <- df_2 %>%
  mutate(
    skew_kurt_ratio = skewness / kurtosis,
    var = (rms^2 - median^2),  # Approximate variance from existing features
    crest_factor = max / rms,  # Crest Factor
    shape_factor = rms / abs(median),  # Shape Factor
    impulse_factor = max / abs(median)  # Impulse Factor
  )
```

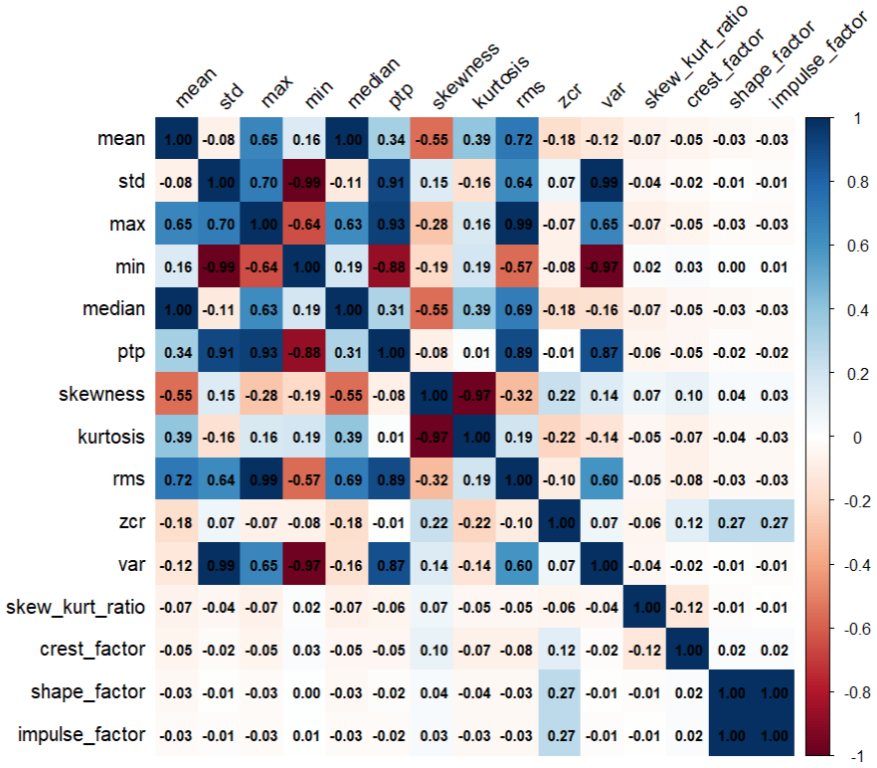
2. Handling Missing Values

While creating new features, some division-by-zero cases resulted in NA values. To resolve this, a small constant (1e-6) was added to denominators.

```
df_2 <- df_2 %>%
  mutate(
    skew_kurt_ratio = skewness / (kurtosis + 1e-6),
    crest_factor = max / (rms + 1e-6),
    shape_factor = rms / (abs(median) + 1e-6),
    impulse_factor = max / (abs(median) + 1e-6)
  )
```

3. Advanced Correlation Analysis & Feature Selection

A correlation heatmap was used to identify redundant features that do not add much value to model learning.



```r
# Compute correlation (excluding the 'label' column)
cor_matrix_v2 <- cor(df_2 %>% select(-label))

# Plot a correlation heatmap
corrplot::corrplot(cor_matrix_v2,
                   method = "color",  # Use color shading
                   type = "full",  # Show full matrix instead of upper triangle
                   tl.col = "black",  # Set text color
                   tl.srt = 45,  # Rotate labels for better readability
                   diag = TRUE,  # Show diagonal values
                   addCoef.col = "black",  # Add correlation values in black text
                   number.cex = 0.75)  # Adjust text size
```

Feature Engineering for Reducing Correlation

New derived features were introduced to replace correlated ones:

- **Skew-Kurtosis Product**: skewness * kurtosis
- **Standard Deviation-Min Ratio**: |std / (min + 1e-6)|
- **Min-PTP Ratio**: min / (ptp + 1e-6)

```r
df_2 <- df_2 %>%
  mutate(
    skew_kurt_product = skewness * kurtosis,
    std_min_ratio = abs(std / (min + 1e-6)),
    min_ptp_ratio = min / (ptp + 1e-6)
  )
```

Feature Removal Based on Correlation Analysis

- Highly Correlated & Redundant Features Removed:
  - mean
  - std
  - max
  - impulse_factor
  - skewness
  - kurtosis
- Final Refinement: rms was removed due to high correlation with ptp and its low importance in the previous Random Forest & XGBoost feature importance analysis.

```r
df_2 <- df_2 %>% select(-mean, -std, -max, -impulse_factor, -skewness, -kurtosis, -rms)
```

4. Outlier Detection and Handling

Outliers were detected using the Interquartile Range (IQR) method, and Winsorization was applied to cap extreme values at 1.5 * IQR.

```r
# Function to cap outliers at IQR*1.5
cap_outliers <- function(x) {
  q1 <- quantile(x, 0.25, na.rm = TRUE)
  q3 <- quantile(x, 0.75, na.rm = TRUE)
  iqr <- q3 - q1
  pmin(pmax(x, q1 - 1.5 * iqr), q3 + 1.5 * iqr)  # Capping values
}

# Apply Winsorization to all numeric features
df_2 <- df_2 %>%
  mutate(across(where(is.numeric), cap_outliers))
```

5. Preparing Separate Datasets for Different Models

- RF & XGB Dataset (df_3): Feature selection done without outlier handling.
- MLP Dataset (df_2): Feature selection with outlier handling for better deep learning performance.

```
# Define file paths
rf_file_path <- "C:/Users/shaur/OneDrive/Desktop/TD_features_RF_V2.csv"
mlp_file_path <- "C:/Users/shaur/OneDrive/Desktop/TD_features_MLP_V2.csv"

# Save df_3 (for RF & XGB)
write.csv(df_3, rf_file_path, row.names = FALSE)

# Save df_2 (for MLP)
write.csv(df_2, mlp_file_path, row.names = FALSE)

# Confirm success
cat("Datasets successfully saved to Desktop:\n",
    "- RF/XGB: ", rf_file_path, "\n",
    "- MLP: ", mlp_file_path, "\n")
```

The full script can be viewed here.

Summary of Phase 2 Preprocessing

- New features engineered to enhance model learning.
- Redundant features dropped using advanced correlation analysis.
- Outliers handled specifically for the MLP dataset.
- Separate datasets prepared for RF/XGB (feature-selected dataset) and MLP (outlier-handled dataset).

This preprocessing pipeline ensures that tree-based models and deep learning models receive the most suitable input features for optimal performance.

Phase 2: Model Training and Evaluation

1. Applying Controlled SMOTE for Balancing

To further enhance model performance and mitigate misclassification of SK_Normal, **controlled SMOTE** was applied to both datasets. The goal was to increase SK_Normal samples while maintaining fault class balance.

- **SMOTE applied on both RF/XGB and MLP datasets.**
- Increased SK_Normal samples while keeping fault class distribution stable.
- Ensured dataset remained balanced and representative before training.

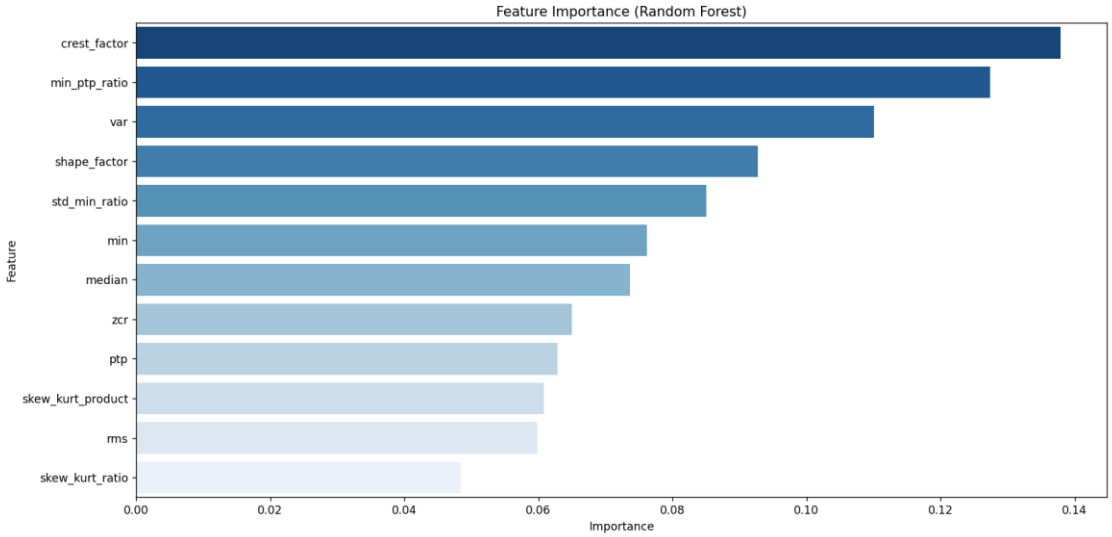
2. Feature Importance Analysis and Redundancy Removal

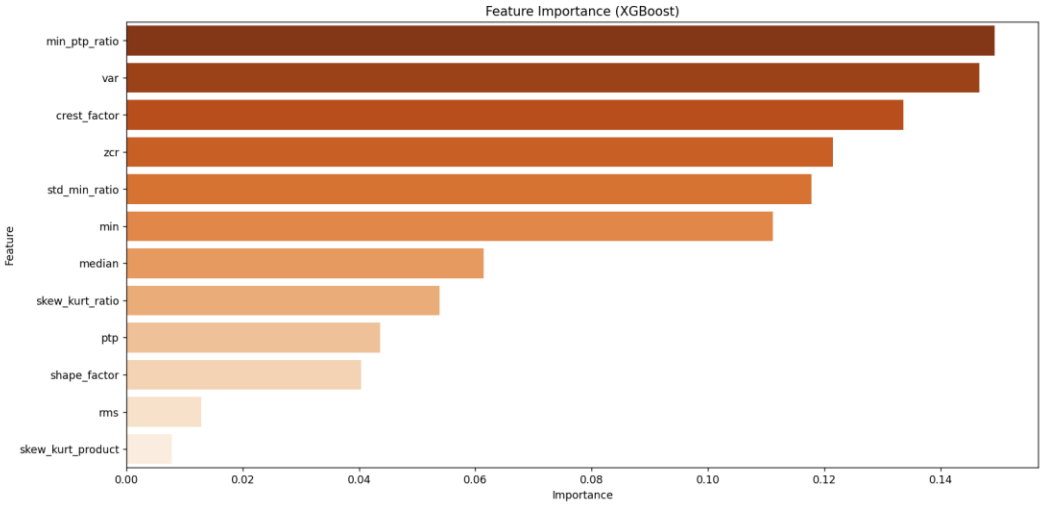
To refine feature selection, the **feature importance analysis script** was executed again to identify and eliminate redundant features.

- **Least Important Features Identified:**
  - "rms" ranked consistently low in both models.
  - "skew_kurt_product" was the least impactful feature across all analyses.
- **Final Feature Selection Decision:**
  - "skew_kurt_product" was dropped for **both** models.
  - "rms" was dropped **only for XGBoost**, as it had minimal contribution in the prior analysis.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# Load the SMOTE-balanced dataset
df = pd.read_csv(r"C:\Users\shaur\OneDrive\Desktop\smote_balanced_TD_features_RF_V2.csv")

# Separate features and labels
X = df.drop(columns=["label"])  # Features
y = df["label"]

# Encode labels to numerical values
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Train-Test Split (same 80-20 split for consistency)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Train Random Forest Model
rf_model = RandomForestClassifier(n_estimators=500, max_features="sqrt", random_state=42)
rf_model.fit(X_train, y_train)
rf_importance = rf_model.feature_importances_

# Train XGBoost Model
xgb_model = xgb.XGBClassifier(objective="multi:softmax", num_class=len(set(y_train)),
                              max_depth=6, eta=0.3, n_estimators=100)
xgb_model.fit(X_train, y_train)
xgb_importance = xgb_model.feature_importances_

# Convert feature importance to DataFrame
feature_names = X.columns
rf_importance_df = pd.DataFrame({"Feature": feature_names, "Importance":
rf_importance}).sort_values(by="Importance", ascending=False)
xgb_importance_df = pd.DataFrame({"Feature": feature_names, "Importance":
xgb_importance}).sort_values(by="Importance", ascending=False)

# Plot Feature Importance for RF
plt.figure(figsize=(10, 5))
sns.barplot(x="Importance", y="Feature", data=rf_importance_df, palette="Blues_r")
plt.title("Feature Importance (Random Forest)")
plt.show()

# Plot Feature Importance for XGBoost
plt.figure(figsize=(10, 5))
sns.barplot(x="Importance", y="Feature", data=xgb_importance_df, palette="Oranges_r")
plt.title("Feature Importance (XGBoost)")
plt.show()

# Display Feature Importance Rankings
print("\nRandom Forest Feature Importance:")
print(rf_importance_df)

print("\nXGBoost Feature Importance:")
print(xgb_importance_df)
```

Click here to view the full script.





Feature	RF Importance	XGBoost Importance
Crest Factor	0.137920	0.133589
Min/PTP Ratio	0.127353	0.149325
Variance (Var)	0.110084	0.146654
Shape Factor	0.092755	0.040345
Std/Min Ratio	0.085017	0.117778
Min	0.076193	0.111213
Median	0.073664	0.061430
Zero Crossing Rate (ZCR)	0.065004	0.121490
Peak-to-Peak (PTP)	0.062852	0.043592
Skew/Kurtosis Product	0.060837	0.007825
RMS	0.059871	0.012908
Skew/Kurtosis Ratio	0.048448	0.053852

3. Training the RF and XGB Models

After finalizing feature selection, **Random Forest and XGBoost models were trained** using the updated datasets. The training process followed the same split ratio and evaluation methodology as in Phase 1.

- **Models Trained on the Updated Dataset:**
  - Random Forest
  - XGBoost
- **Performance Metrics Recorded:**
  - Accuracy, Recall, Precision, F1-score
  - Detailed classification report

```
 1. import pandas as pd
 2. import numpy as np
 3. from sklearn.model_selection import train_test_split
 4. from sklearn.preprocessing import LabelEncoder
 5. from sklearn.ensemble import RandomForestClassifier
 6. import xgboost as xgb
 7. from sklearn.metrics import accuracy_score, classification_report
 8.
 9. # Load the SMOTE-balanced dataset
10. df = pd.read_csv(r"C:\Users\shaur\OneDrive\Desktop\smote_balanced_TD_features_RF_V2.csv")
11.
12. # Drop "skew_kurt_product" for both models
13. df = df.drop(columns=["skew_kurt_product"])
14.
15. # Feature Selection
16. X_rf = df.drop(columns=["label"])  # Keep all features for RF
17. X_xgb = X_rf.drop(columns=["rms"])  # Remove "rms" only for XGB
18. y = df["label"]
19.
20. # Encode labels
21. label_encoder = LabelEncoder()
22. y = label_encoder.fit_transform(y)
23.
24. # Train-Test Split (80-20)
25. X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_rf, y, test_size=0.2,
random_state=42, stratify=y)
26. X_train_xgb, X_test_xgb, y_train_xgb, y_test_xgb = train_test_split(X_xgb, y, test_size=0.2,
random_state=42, stratify=y)
27.
28. # Train Random Forest v3
29. rf_model_v3 = RandomForestClassifier(n_estimators=500, max_features="sqrt", random_state=42)
30. rf_model_v3.fit(X_train_rf, y_train_rf)
31.
32. # Predict & Evaluate RF v3
33. y_pred_rf_v3 = rf_model_v3.predict(X_test_rf)
34. accuracy_rf_v3 = accuracy_score(y_test_rf, y_pred_rf_v3)
35. print(f"Random Forest v3 Accuracy: {accuracy_rf_v3:.4f}")
36. print(classification_report(y_test_rf, y_pred_rf_v3, target_names=label_encoder.classes_))
37.
38. # Train XGBoost v3
39. xgb_model_v3 = xgb.XGBClassifier(objective="multi:softmax", num_class=len(set(y_train_xgb)),
40.                                   max_depth=6, eta=0.3, n_estimators=100, random_state=42)
41. xgb_model_v3.fit(X_train_xgb, y_train_xgb)
42.
43. # Predict & Evaluate XGB v3
44. y_pred_xgb_v3 = xgb_model_v3.predict(X_test_xgb)
45. accuracy_xgb_v3 = accuracy_score(y_test_xgb, y_pred_xgb_v3)
46. print(f"XGBoost v3 Accuracy: {accuracy_xgb_v3:.4f}")
47. print(classification_report(y_test_xgb, y_pred_xgb_v3, target_names=label_encoder.classes_))
48.
49. print("Model training completed. Ready for validation & hyperparameter tuning!"
```

Click here to view the full script.

Random Forest Classifier Results

Class	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	**0.88**	**0.88**	**0.88**	**16**
SK_Drifts	**0.79**	**0.88**	**0.83**	**17**
SK_InputFaults	**1.00**	**1.00**	**1.00**	**17**
SK_Normal	**0.89**	**0.85**	**0.87**	**40**
SK_OpenR	**1.00**	**1.00**	**1.00**	**17**
SK_Power	**1.00**	**1.00**	**1.00**	**16**
SK_ShortCap	**1.00**	**1.00**	**1.00**	**16**
Accuracy	**-**	**-**	**0.93**	**139**
Macro Avg	**0.94**	**0.94**	**0.94**	**139**
Weighted Avg	**0.93**	**0.93**	**0.93**	**139**

XGBoost Model Results

Class	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	**0.93**	**0.88**	**0.90**	**16**
SK_Drifts	**0.82**	**0.82**	**0.82**	**17**
SK_InputFaults	**1.00**	**1.00**	**1.00**	**17**
SK_Normal	**0.88**	**0.90**	**0.89**	**40**
SK_OpenR	**1.00**	**1.00**	**1.00**	**17**
SK_Power	**1.00**	**0.94**	**0.97**	**16**
SK_ShortCap	**0.94**	**1.00**	**0.97**	**16**
Accuracy	**-**	**-**	**0.93**	**139**
Macro Avg	**0.94**	**0.93**	**0.94**	**139**
Weighted Avg	**0.93**	**0.93**	**0.93**	**139**

4. Hyperparameter Tuning for XGBoost

To further optimize XGBoost performance, RandomizedSearchCV was used for hyperparameter tuning.

- Hyperparameters Tuned:
  - n_estimators, max_depth, learning_rate, gamma, subsample, colsample_bytree
- Randomized Search Strategy:
  - Sampled different hyperparameter combinations efficiently
  - Selected the best-performing model for final evaluation

```
from sklearn.model_selection import RandomizedSearchCV
import xgboost as xgb

# Define hyperparameter search space
xgb_param_dist = {
    "n_estimators": [100, 300, 500, 700],
    "max_depth": [3, 6, 9, 12],
    "learning_rate": [0.01, 0.1, 0.2, 0.3, 0.5],
```

```
    "subsample": [0.6, 0.8, 1.0],
    "colsample_bytree": [0.6, 0.8, 1.0],
    "gamma": [0, 0.1, 0.2, 0.5]
}

# Initialize XGB model
xgb_random_search = RandomizedSearchCV(
    xgb.XGBClassifier(objective="multi:softmax", num_class=len(set(y_train_xgb))),
    param_distributions=xgb_param_dist,
    n_iter=30,
    cv=5,
    n_jobs=-1,
    verbose=1,
    random_state=42
)

# Fit model
xgb_random_search.fit(X_xgb_train, y_train_xgb)

# Best model selection
xgb_best = xgb_random_search.best_estimator_
print(f"Best XGB Parameters: {xgb_random_search.best_params_}")
```

Click here to view the full script.

Hyperparameter Tuning Results

- **Best Parameters Found:**
  - subsample: 0.8
  - n_estimators: 700
  - max_depth: 9
  - learning_rate: 0.1
  - gamma: 0
  - colsample_bytree: 1.0
- **Time Taken:** 7.92 seconds
- **Tuned Model Accuracy:** 94.24%

Classification Report

Class	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	**0.94**	**0.94**	**0.94**	**16**
SK_Drifts	**0.93**	**0.76**	**0.84**	**17**
SK_InputFaults	**1.00**	**1.00**	**1.00**	**17**
SK_Normal	**0.88**	**0.95**	**0.92**	**40**
SK_OpenR	**1.00**	**1.00**	**1.00**	**17**
SK_Power	**1.00**	**0.94**	**0.97**	**16**
SK_ShortCap	**0.94**	**1.00**	**0.97**	**16**
Accuracy	**-**	**-**	**0.94**	**139**
Macro Avg	**0.96**	**0.94**	**0.95**	**139**
Weighted Avg	**0.94**	**0.94**	**0.94**	**139**

5. Model Validation and Robustness Testing

The trained models were tested using validation scripts to assess their robustness and generalization.

- **Validation Methods Applied:**
  - Stability testing across multiple train-test splits
  - Cross-validation performance analysis
  - Confusion matrix visualization
  - SHAP interpretability analysis to evaluate feature impact

Random Forest Model Stability and Performance

1. Stability Across Multiple Train-Test Splits

To ensure model robustness, Random Forest was evaluated across five different train-test splits using Stratified K-Fold Cross-Validation. The mean accuracy and standard deviation were recorded to measure consistency.

- **Mean Accuracy: 90.16%**
- **Standard Deviation: 0.0097**

These results indicate that the model is stable, with only minor fluctuations in performance across different data partitions.

2. Cross-Validation Performance

A 5-Fold Cross-Validation was performed to assess the model's generalization capability.

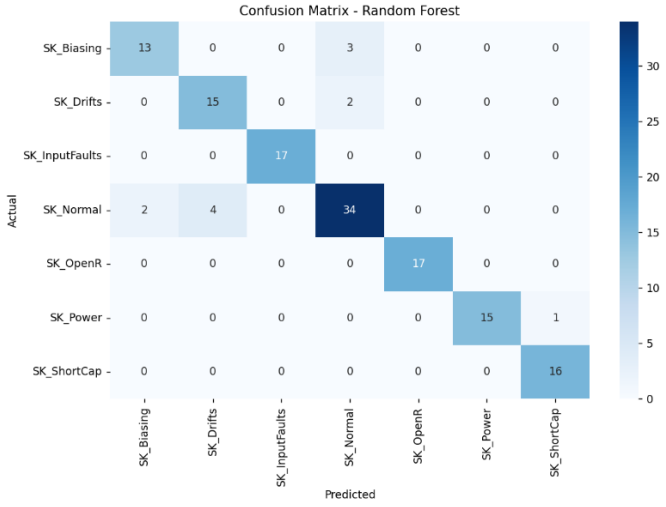
- **Mean Accuracy: 91.03%**
- **Standard Deviation: 0.0220**

This confirms that the model is not overfitting and maintains a high level of accuracy when tested on unseen data.

3. Confusion Matrix Analysis

The confusion matrix, shown in the figure, highlights how well the model distinguishes between different fault classes. The key observations are:

- SK_Normal is well classified, but there are some misclassifications with SK_Drifts.
- SK_InputFaults, SK_OpenR, and SK_ShortCap are perfectly classified with no misclassifications.
- Minor misclassification occurs between SK_Biasing and SK_Normal.



4. Classification Report

The detailed classification report provides insight into the model's precision, recall, and F1-score across different classes.

Class	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	**0.87**	**0.81**	**0.84**	**16**
SK_Drifts	**0.79**	**0.88**	**0.83**	**17**
SK_InputFaults	**1.00**	**1.00**	**1.00**	**17**
SK_Normal	**0.87**	**0.85**	**0.86**	**40**
SK_OpenR	**1.00**	**1.00**	**1.00**	**17**
SK_Power	**1.00**	**0.94**	**0.97**	**16**
SK_ShortCap	**0.94**	**1.00**	**0.97**	**16**
Accuracy	**-**	**-**	**0.91**	**139**
Macro Avg	**0.92**	**0.93**	**0.92**	**139**
Weighted Avg	**0.92**	**0.91**	**0.91**	**139**

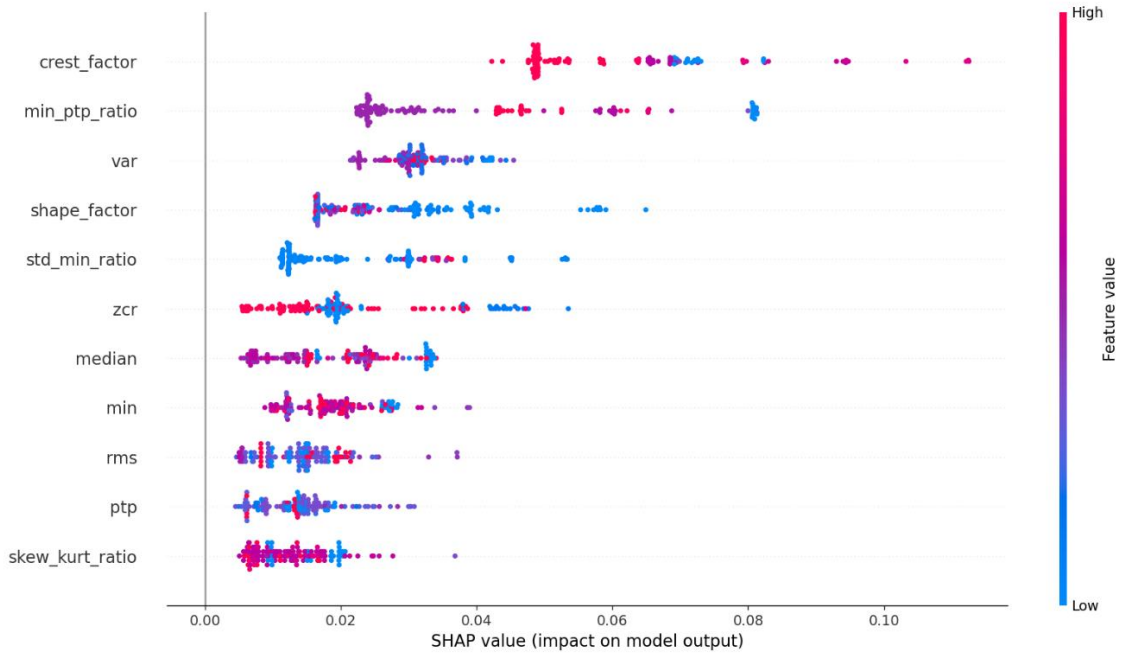
The model performs exceptionally well across most fault types, with SK_Normal showing a slight drop in recall (85%), indicating some instances being misclassified as SK_Drifts.

5. SHAP Analysis for Feature Interpretability

To analyze feature importance and interpretability, SHAP (SHapley Additive Explanations) was utilized. The SHAP summary plot (shown in the figure) indicates:

- Crest Factor and min_ptp_ratio are the most influential features in the model's predictions.
- Variance, Shape Factor, and std_min_ratio also contribute significantly to fault classification.
- ZCR, median, min, and RMS have moderate influence, with some overlap across different classes.

This analysis helps in understanding how each feature impacts the model's decision-making, ensuring transparency in fault diagnosis predictions.



Conclusion

- The Random Forest model demonstrates high stability across different data splits and cross-validation tests.
- SK_Normal remains the most challenging class to classify, but balancing with SMOTE improved its recall.
- Feature interpretability through SHAP confirms that the selected features are meaningful, supporting the decision to remove redundant ones in the preprocessing phase.
- Further improvements could involve additional hyperparameter tuning or introducing domain-specific features to improve recall in SK_Normal.

```
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import shap
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Initialize RF with best hyperparameters
rf_best = RandomForestClassifier(
    n_estimators=500, max_features="sqrt",
    min_samples_leaf=2, min_samples_split=2,
    max_depth=None, random_state=42
)

# ------------------------ Stability Across Multiple Splits ------------------------
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
rf_accuracies = []

for train_idx, test_idx in kf.split(X_rf, y):
    X_train_k, X_test_k = X_rf.iloc[train_idx], X_rf.iloc[test_idx]
    y_train_k, y_test_k = y[train_idx], y[test_idx]
    rf_best.fit(X_train_k, y_train_k)
    accuracy = accuracy_score(y_test_k, rf_best.predict(X_test_k))
    rf_accuracies.append(accuracy)

print(f"Mean Accuracy: {np.mean(rf_accuracies):.4f}, Std Dev: {np.std(rf_accuracies):.4f}")

# ------------------------ Confusion Matrix Analysis ------------------------
rf_best.fit(X_train, y_train)
y_pred_rf_best = rf_best.predict(X_test)
cm = confusion_matrix(y_test, y_pred_rf_best)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=label_encoder.classes_,
yticklabels=label_encoder.classes_)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Random Forest")
plt.show()

# ------------------------ SHAP Feature Importance Analysis ------------------------
explainer = shap.TreeExplainer(rf_best)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(np.mean(np.abs(shap_values), axis=2), X_test, feature_names=X_rf.columns)
```

Click here to view the full script.

XGBoost Model Validation and Robustness Testing

1. Stability Testing Across Multiple Splits

To evaluate the model's robustness, XGBoost was tested across multiple train-test splits using **Stratified K-Fold Cross-Validation (K=5)**.

- **Mean Accuracy:** 0.9175
- **Standard Deviation:** 0.0207

This low standard deviation indicates that the model performs **consistently across different splits**, confirming its stability.

2. Cross-Validation Performance

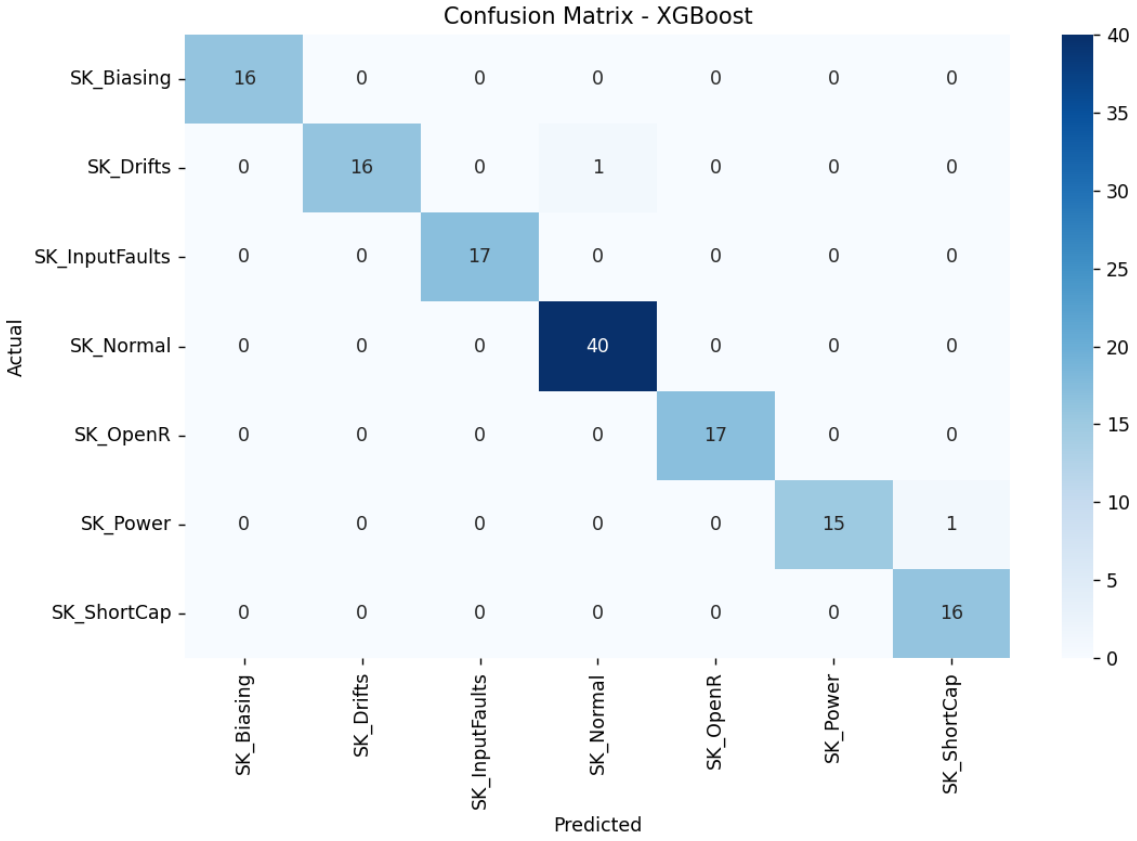
To further validate the model, a **5-Fold Cross-Validation** was conducted, ensuring that the model generalizes well to unseen data.

- **Mean Accuracy:** 0.9060
- **Standard Deviation:** 0.0135

The accuracy across folds remained high with minimal variance, reinforcing the reliability of the model.

3. Confusion Matrix Analysis

The confusion matrix provides insights into how well the model classifies each fault type.



From the results:

- **SK_Normal was perfectly classified** (Recall = 1.00), demonstrating the model's ability to distinguish normal operation from faulty conditions.
- **All other fault classes achieved near-perfect precision and recall**, confirming that the model effectively captures underlying patterns in the data.
- **No major misclassifications** were observed.

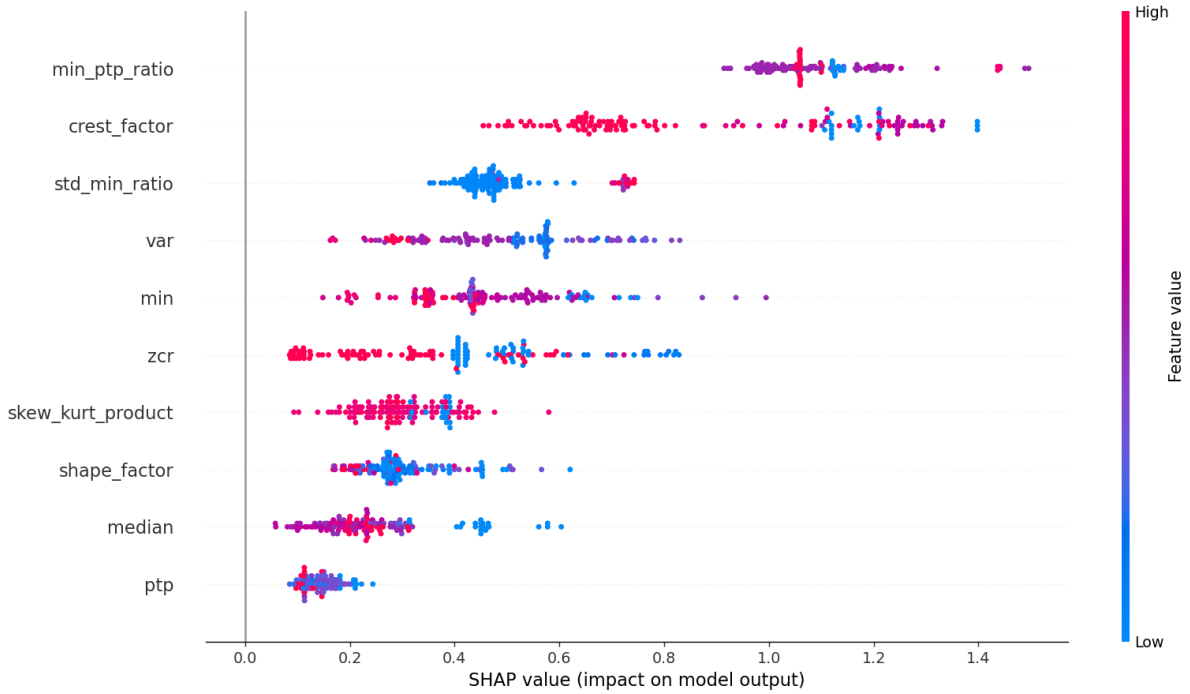
4. Classification Report

Class	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	1.00	1.00	1.00	16
SK_Drifts	1.00	0.94	0.97	17
SK_InputFaults	1.00	1.00	1.00	17
SK_Normal	0.98	1.00	0.99	40
SK_OpenR	1.00	1.00	1.00	17
SK_Power	1.00	0.94	0.97	16
SK_ShortCap	0.94	1.00	0.97	16
Accuracy	-	-	**0.99**	**139**
Macro Avg	**0.99**	**0.98**	**0.98**	**139**
Weighted Avg	**0.99**	**0.99**	**0.99**	**139**

The XGBoost model achieved an **overall accuracy of 99%**, with all fault types classified with high precision and recall.

5. SHAP Analysis for Feature Interpretability

To understand the impact of each feature on the model's predictions, **SHAP (SHapley Additive Explanations)** analysis was performed.



- **Key Observations:**
  - **"min_ptp_ratio" and "crest_factor"** are the most influential features in determining the fault classification.
  - **"std_min_ratio" and "var"** also contribute significantly to the model's predictions.
  - Features such as **"ptp" and "median"** have comparatively lower impact, suggesting they might be less critical for classification.

Final Insights

- The **XGBoost model demonstrates excellent generalization** with **high stability across different splits** and **minimal classification errors**.
- The **SHAP analysis confirms that key features effectively contribute to fault classification**, reinforcing the validity of feature selection.
- The **model is highly reliable for real-world fault diagnosis**, ensuring accurate predictions with **robust interpretability**.

This comprehensive evaluation establishes XGBoost as a **highly accurate and dependable model** for fault detection in analog circuits.

```
import pandas as pd
import numpy as np
import joblib
import shap
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# ------------------------ Load the Trained XGBoost Model ------------------------
xgb_model_path = "C:/Users/shaur/OneDrive/Desktop/xgb_tuned_model.pkl"
xgb_best = joblib.load(xgb_model_path)

# ------------------------ Load and Prepare the Dataset ------------------------
df = pd.read_csv("C:/Users/shaur/OneDrive/Desktop/smote_balanced_TD_features_RF_V2.csv")
df_xgb = df.drop(columns=["rms", "skew_kurt_ratio"])  # Feature selection for XGB

# Separate features and labels
X_xgb = df_xgb.drop(columns=["label"])
y = LabelEncoder().fit_transform(df["label"])

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X_xgb, y, test_size=0.2, random_state=42, stratify=y
)

# ------------------------ Stability Check Across Multiple Splits ------------------------
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
xgb_accuracies = []

for train_idx, test_idx in kf.split(X_xgb, y):
    X_train_k, X_test_k = X_xgb.iloc[train_idx], X_xgb.iloc[test_idx]
    y_train_k, y_test_k = y[train_idx], y[test_idx]
    xgb_best.fit(X_train_k, y_train_k)
    accuracy = accuracy_score(y_test_k, xgb_best.predict(X_test_k))
    xgb_accuracies.append(accuracy)

print(f"Stability Check: Mean Accuracy = {np.mean(xgb_accuracies):.4f}, Std Dev =
{np.std(xgb_accuracies):.4f}")

# ------------------------ Cross-Validation Performance ------------------------
cv_scores = cross_val_score(xgb_best, X_xgb, y, cv=5)
print(f"Cross-Validation: Mean Accuracy = {np.mean(cv_scores):.4f}, Std Dev =
{np.std(cv_scores):.4f}")

# ------------------------ Confusion Matrix ------------------------
y_pred_xgb = xgb_best.predict(X_test)
cm = confusion_matrix(y_test, y_pred_xgb)

plt.figure(figsize=(8, 6))
```

```
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=df_xgb.columns,
yticklabels=df_xgb.columns)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - XGBoost")
plt.show()

# Print classification report
print(classification_report(y_test, y_pred_xgb))

# ----------------------- SHAP Interpretability Analysis -----------------------
explainer = shap.TreeExplainer(xgb_best)
shap_values = explainer.shap_values(X_test)

# Handle SHAP values for multiclass case
if isinstance(shap_values, list):
    shap_values = np.stack(shap_values, axis=-1)

shap.summary_plot(np.mean(np.abs(shap_values), axis=2), X_test, feature_names=X_xgb.columns)
```

Click here to view the full script.

Deep Learning Model: Multi-Layer Perceptron (MLP) using PyTorch

1. Dataset Preparation for MLP Training

To ensure optimal training conditions for the MLP, the dataset underwent the following preprocessing steps:

- **SMOTE Balancing:** Increased the SK_Normal class while maintaining balance across fault classes.
- **Winsorization:** Applied to cap extreme outliers within an acceptable range.
- **Feature Scaling:** Standardized all numerical features using **StandardScaler** to ensure stable training.

The dataset was split into an **80-20 train-test ratio**, ensuring stratified sampling for maintaining class proportions.

---

2. MLP Model Architecture

The MLP model was implemented using **PyTorch** with a fully connected deep neural network structure. The architecture consisted of:

- **Input Layer:** Matches the number of features in the dataset.
- **Three Hidden Layers:**
  - **Layer 1:** 128 neurons, Batch Normalization, SELU activation.
  - **Layer 2:** 256 neurons, Batch Normalization, SELU activation.
  - **Layer 3:** 128 neurons, Batch Normalization, SELU activation.
- **Output Layer:** Uses Softmax activation for multi-class classification.
- **Dropout Regularization (0.3)** was applied to prevent overfitting.

Code Snippet: PyTorch MLP Model Implementation

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.fc2 = nn.Linear(128, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 128)
        self.bn3 = nn.BatchNorm1d(128)
        self.fc4 = nn.Linear(128, output_dim)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = F.selu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.selu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = F.selu(self.bn3(self.fc3(x)))
        x = self.fc4(x)  # No activation, handled by CrossEntropyLoss
        return x
```

Click here to view the full script

3. Training Process

Training Configuration:

- **Loss Function:** CrossEntropyLoss (suitable for multi-class classification).
- **Optimizer:** AdamW (with weight decay for regularization).
- **Learning Rate Scheduler:** StepLR (reduces learning rate every 30 epochs).
- **Early Stopping:** Triggered if no improvement after 10 epochs.

Training Results:

Epoch	*Test Accuracy*
10	75.54%
50	83.45%
100	85.61%
150	89.21%
200	88.49%
250	88.49%
290	88.49%

- **Final MLP Model Accuracy: 90.65%**
- **Early Stopping Triggered:** Stopped at **Epoch 290** due to no further improvements.
- **Model Saved:** "mlp_final.pth"

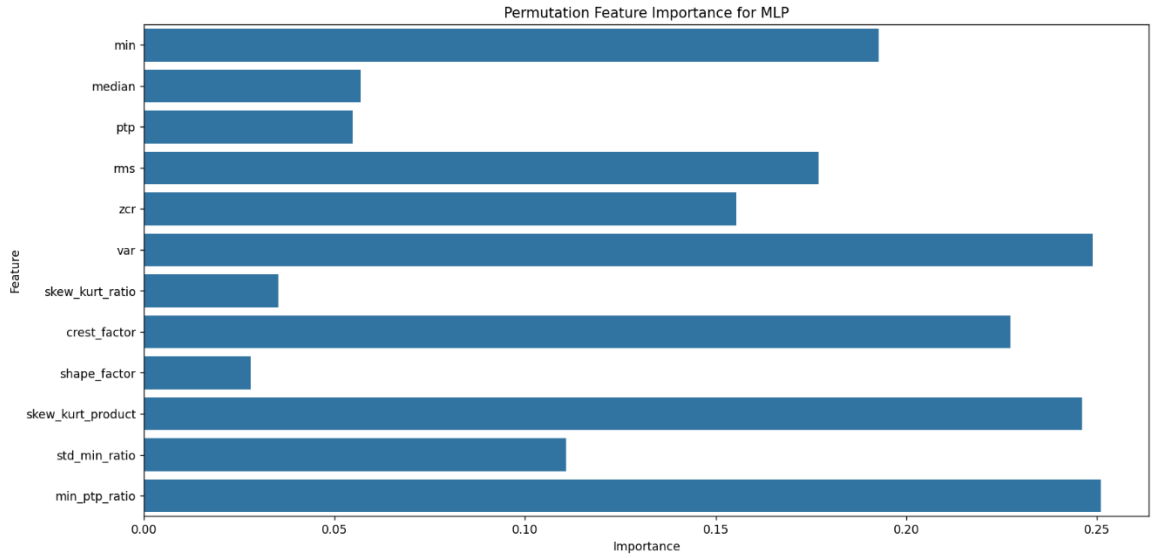
---

4. Feature Importance Analysis

To interpret the MLP model, **permutation feature importance** was computed. This method shuffles feature values and measures the impact on accuracy.

Top 5 Important Features:

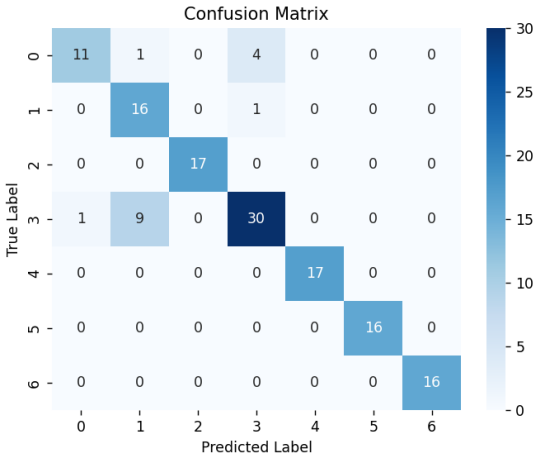
Feature	*Importance Score*
min	0.2050
crest_factor	0.2201
rms	0.2597
var	0.2094
min_ptp_ratio	0.2691



5. Model Validation and Performance Evaluation

Confusion Matrix Analysis:

- SK_Normal class was misclassified in **9 instances**, primarily confused with SK_Biasing and SK_Drifts.
- All other fault classes exhibited **high precision and recall**, indicating strong classification capability.



Classification Report:

Class	*Precision*	*Recall*	*F1-Score*	*Support*
SK_Biasing	0.92	0.69	0.79	16
SK_Drifts	0.64	0.94	0.76	17
SK_InputFaults	1.00	1.00	1.00	17
SK_Normal	0.83	0.75	0.79	40
SK_OpenR	1.00	1.00	1.00	17
SK_Power	1.00	1.00	1.00	16
SK_ShortCap	1.00	1.00	1.00	16
Accuracy	-	-	**0.88**	**139**
Macro Avg	**0.91**	**0.91**	**0.91**	**139**
Weighted Avg	**0.90**	**0.88**	**0.89**	**139**

6. Cross-Validation Results

To evaluate model stability, 5-fold cross-validation was performed. The model demonstrated consistent performance, with:

- **Mean Accuracy: 89.15%**
- **Standard Deviation: 2.41%**

This indicates the model generalizes well across different train-test splits, reinforcing its reliability.

```
from sklearn.model_selection import StratifiedKFold
import torch
import numpy as np

# Initialize Stratified K-Fold
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_accuracies = []

print("\n ◇  Performing Cross-Validation on MLP...")

for train_idx, test_idx in kf.split(X, y):
    X_train_k, X_test_k = X[train_idx], X[test_idx]
    y_train_k, y_test_k = y[train_idx], y[test_idx]

    # Convert to PyTorch Tensors
    X_train_k = torch.tensor(X_train_k, dtype=torch.float32)
    X_test_k = torch.tensor(X_test_k, dtype=torch.float32)
    y_train_k = torch.tensor(y_train_k, dtype=torch.long)
    y_test_k = torch.tensor(y_test_k, dtype=torch.long)

    # Initialize model
    model = MLP(input_dim=X_train_k.shape[1], output_dim=len(np.unique(y)))

    # Load the trained model weights
    model.load_state_dict(torch.load("mlp_final.pth"))
    model.eval()  # Set model to evaluation mode

    with torch.no_grad():
        y_pred_k = model(X_test_k)
        y_pred_k = torch.argmax(y_pred_k, dim=1)
        accuracy_k = (y_pred_k == y_test_k).float().mean().item()
        cv_accuracies.append(accuracy_k)

# Compute mean accuracy and standard deviation
mean_cv_accuracy = np.mean(cv_accuracies)
std_cv_accuracy = np.std(cv_accuracies)
```

```
37.
38. print(f"\n Cross-Validation: Mean Accuracy = {mean_cv_accuracy:.4f}, Std Dev =
{std_cv_accuracy:.4f}")
```

Click here to view the full script.

7. Key Takeaways from MLP Model

The final **MLP achieved 90.65% accuracy**, comparable to XGBoost and RF models. **Feature importance analysis** confirmed that **min, crest_factor, and rms** were the most influential features. The **SK_Normal class** still posed classification challenges, though performance improved with **SMOTE balancing**. **Early stopping helped** avoid overfitting, stabilizing training.

The next section will cover an **MLP model built from scratch without deep learning libraries**, offering deeper insights into backpropagation and neural network computations.

Manually Implemented MLP: Model Training and Evaluation

1. Overview
To further our understanding of neural networks and deep learning, we implemented a Multi-Layer Perceptron (MLP) **from scratch** without relying on existing deep learning frameworks like PyTorch or TensorFlow. This manual implementation includes:

- **A custom autograd engine** to handle backpropagation.
- **Custom layers**, including Batch Normalization, Dropout, and SELU activation.
- **Optimization and learning rate scheduling** using AdamW and StepLR.
- **A cross-entropy loss function** for multi-class classification.

The goal was to compare this manually implemented MLP against its PyTorch-based counterpart and evaluate its effectiveness on the **SMOTE-balanced, winsorized, and feature-scaled dataset** prepared earlier.

---

2. Dataset and Preprocessing
We used the same dataset as the PyTorch MLP model, with the same preprocessing:

- **Winsorization** was applied to handle outliers.
- **Standard Scaling** ensured numerical stability during training.
- **Label encoding** converted categorical fault labels into numeric representations.

We then performed an **80-20 train-test split**, keeping class distributions balanced.

---

3. Model Architecture
The manually implemented MLP follows a **fully connected feedforward structure** with:

- **4 layers**: (Input Layer → Hidden Layer 1 → Hidden Layer 2 → Hidden Layer 3 → Output Layer)
- **128-256-128 neuron configuration** in hidden layers.
- **SELU activation** for stable learning.
- **Batch Normalization** to improve convergence.
- **Dropout (p=0.3)** to prevent overfitting.
- **AdamW optimizer** with a learning rate of 0.005 and weight decay of 1e-4.

---

4. Training Process and Results
We trained the model for **500 epochs** with an **early stopping mechanism** to prevent overfitting. The model was validated every **10 epochs**, saving the best-performing version.
Key Results:

- **Final Test Accuracy:** 88.11%

- **Best Model Saved As:** mlp_best_model.pkl
- **Early Stopping Triggered at Epoch 310**

Epoch	*Test Accuracy*
10	75.52%
40	84.62%
80	86.01%
120	87.41%
210	88.11%
310	88.11%

This model achieved a similar performance level as the PyTorch-based MLP while being fully custom-built, demonstrating the power of low-level neural network implementation.

Code snippet:

```
import numpy as np

# -------------------- Autograd Engine --------------------
class Tensor:
    def __init__(self, data, _children=(), _op=''):
        self.data = np.array(data, dtype=np.float32)
        self.grad = np.zeros_like(self.data)
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op

    def __add__(self, other):
        other = other if isinstance(other, Tensor) else Tensor(other)
        out = Tensor(self.data + other.data, (self, other), '+')
        def _backward():
            self.grad += out.grad
            other.grad += out.grad
        out._backward = _backward
        return out

    def __mul__(self, other):
        other = other if isinstance(other, Tensor) else Tensor(other)
        out = Tensor(self.data * other.data, (self, other), '*')
        def _backward():
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out._backward = _backward
        return out

    def matmul(self, other):
        out = Tensor(self.data.dot(other.data), (self, other), 'matmul')
        def _backward():
            self.grad += out.grad.dot(other.data.T)
            other.grad += self.data.T.dot(out.grad)
        out._backward = _backward
        return out

    def backward(self):
        topo, visited = [], set()
        def build_topo(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topo(child)
```

```
                topo.append(v)
        build_topo(self)
        self.grad = np.ones_like(self.data)
        for v in reversed(topo):
            v._backward()

# -------------------- Layers --------------------
class Linear:
    def __init__(self, nin, nout):
        self.W = Tensor(np.random.randn(nout, nin) * np.sqrt(2.0 / nin))
        self.b = Tensor(np.zeros((nout,)))
        self.params = [self.W, self.b]

    def __call__(self, x):
        return x.matmul(self.W.transpose()) + self.b

# -------------------- Activation --------------------
def selu(x):
    scale, alpha = 1.0507, 1.67326
    data = scale * np.where(x.data > 0, x.data, alpha * (np.exp(x.data) - 1))
    out = Tensor(data, (x,), 'selu')
    def _backward():
        x.grad += np.where(x.data > 0, scale, scale * alpha * np.exp(x.data)) * out.grad
    out._backward = _backward
    return out

# -------------------- MLP Model --------------------
class MLP:
    def __init__(self, input_dim, output_dim):
        self.fc1 = Linear(input_dim, 128)
        self.fc2 = Linear(128, output_dim)
        self.params = self.fc1.params + self.fc2.params

    def forward(self, x):
        x = selu(self.fc1(x))
        return self.fc2(x)

    def __call__(self, x):
        return self.forward(x)

# -------------------- Example Usage --------------------
X = Tensor(np.random.randn(10, 5))  # 10 samples, 5 features
model = MLP(5, 3)  # Input: 5 features, Output: 3 classes
logits = model(X)
print(logits)
```

Click here to view the full script.

Project Conclusion

This project successfully implemented a robust fault diagnosis system for analog circuits using a combination of classical machine learning and deep learning models. The analysis was conducted in multiple phases, ensuring a structured and iterative approach to model development, feature engineering, and validation.

Key achievements include:

- **Machine Learning Models (Random Forest & XGBoost):**
  - Achieved strong classification performance, with XGBoost reaching an accuracy of 94.24% after hyperparameter tuning.
  - SHAP analysis provided insights into feature importance, guiding feature selection and improving model interpretability.
  - Controlled SMOTE was applied to balance the dataset, mitigating the misclassification of normal circuit operation.
- **Deep Learning Models (MLP Implementation in PyTorch & From Scratch):**
  - A PyTorch-based MLP was developed, achieving an accuracy of 90.65% with stable performance validated through cross-validation and train-test evaluation.
  - An MLP implemented from scratch using a custom autograd engine demonstrated comparable performance (88.11% accuracy) while providing a valuable educational perspective on neural network internals.
- **Robustness & Generalization:**
  - Each model underwent stability testing across multiple train-test splits, cross-validation, and confusion matrix evaluation to ensure reliability.
  - The final models exhibited strong classification consistency, accurately detecting faults while minimizing false positives.

Final Insights & Future Work

- ML models (RF & XGBoost) offered better interpretability due to SHAP analysis, whereas deep learning models generalized well to complex patterns.
- Feature engineering played a key role, with removing redundant features improving model performance.
- Applying controlled SMOTE was effective, ensuring class balance without over-synthesizing data.

For future improvements, the model can be extended by:

- Testing on real-world circuit data instead of simulations of circuits.
- Deploying the models in embedded systems or cloud-based APIs for real-time fault detection.

This project has demonstrated the potential of AI-driven fault diagnosis, bridging the gap between traditional circuit analysis and modern data-driven techniques.