[1]:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report, confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.impute import SimpleImputer  # Corrected import
from sklearn.ensemble import RandomForestClassifier
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from sklearn.metrics import confusion_matrix
import joblib
import shap
import streamlit as st

# Constants
RANDOM_STATE = 42

# Load dataset
file_path = 'sensor.csv'
data = pd.read_csv(file_path)

# Validate dataset
# TODO: Add any specific data validation steps here (e.g., checking for correct datatypes, missing values beyond NaN, etc.)

# Display dataset overview
print(data.head())
print(data.describe())
```

```
   Unnamed: 0            timestamp  sensor_00  sensor_01  sensor_02  \
0           0  2018-04-01 00:00:00   2.465394   47.09201    53.2118
1           1  2018-04-01 00:01:00   2.465394   47.09201    53.2118
2           2  2018-04-01 00:02:00   2.444734   47.35243    53.2118
3           3  2018-04-01 00:03:00   2.460474   47.09201    53.1684
4           4  2018-04-01 00:04:00   2.445718   47.13541    53.2118

   sensor_03  sensor_04  sensor_05  sensor_06  sensor_07  ...  sensor_43  \
0  46.310760   634.3750   76.45975   13.41146   16.13136  ...   41.92708
1  46.310760   634.3750   76.45975   13.41146   16.13136  ...   41.92708
2  46.397570   638.8889   73.54598   13.32465   16.03733  ...   41.66666
3  46.397568   628.1250   76.98898   13.31742   16.24711  ...   40.88541
4  46.397568   636.4583   76.58897   13.35359   16.21094  ...   41.40625

   sensor_44  sensor_45  sensor_46  sensor_47  sensor_48  sensor_49  \
0  39.641200   65.68287   50.92593  38.194440   157.9861   67.70834
1  39.641200   65.68287   50.92593  38.194440   157.9861   67.70834
2  39.351852   65.39352   51.21528  38.194443   155.9606   67.12963
3  39.062500   64.81481   51.21528  38.194440   155.9606   66.84028
4  38.773150   65.10416   51.79398  38.773150   158.2755   66.55093
```

```
   sensor_50  sensor_51  machine_status
0   243.0556   201.3889          NORMAL
1   243.0556   201.3889          NORMAL
2   241.3194   203.7037          NORMAL
3   240.4514   203.1250          NORMAL
4   242.1875   201.3889          NORMAL

[5 rows x 55 columns]
          Unnamed: 0      sensor_00      sensor_01      sensor_02  \
count  220320.000000  210112.000000  219951.000000  220301.000000
mean   110159.500000       2.372221      47.591611      50.867392
std     63601.049991       0.412227       3.296666       3.666820
min         0.000000       0.000000       0.000000      33.159720
25%     55079.750000       2.438831      46.310760      50.390620
50%    110159.500000       2.456539      48.133678      51.649300
75%    165239.250000       2.499826      49.479160      52.777770
max    220319.000000       2.549016      56.727430      56.032990

           sensor_03      sensor_04      sensor_05      sensor_06  \
count  220301.000000  220301.000000  220301.000000  215522.000000
mean       43.752481     590.673936      73.396414      13.501537
std         2.418887     144.023912      17.298247       2.163736
min        31.640620       2.798032       0.000000       0.014468
25%        42.838539     626.620400      69.976260      13.346350
50%        44.227428     632.638916      75.576790      13.642940
75%        45.312500     637.615723      80.912150      14.539930
max        48.220490     800.000000      99.999880      22.251160

           sensor_07      sensor_08  ...      sensor_42      sensor_43  \
count  214869.000000  215213.000000  ...  220293.000000  220293.000000
mean       15.843152      15.200721  ...      35.453455      43.879591
std         2.201155       2.037390  ...      10.259521      11.044404
min         0.000000       0.028935  ...      22.135416      24.479166
25%        15.907120      15.183740  ...      32.812500      39.583330
50%        16.167530      15.494790  ...      35.156250      42.968750
75%        16.427950      15.697340  ...      36.979164      46.614580
max        23.596640      24.348960  ...     374.218800     408.593700

           sensor_44      sensor_45      sensor_46      sensor_47  \
count  220293.000000  220293.000000  220293.000000  220293.000000
mean       42.656877      43.094984      48.018585      44.340903
std        11.576355      12.837520      15.641284      10.442437
min        25.752316      26.331018      26.331018      27.199070
25%        36.747684      36.747684      40.509258      39.062500
50%        40.509260      40.219910      44.849540      42.534720
75%        45.138890      44.849540      51.215280      46.585650
max      1000.000000     320.312500     370.370400     303.530100

           sensor_48      sensor_49      sensor_50      sensor_51
count  220293.000000  220293.000000  143303.000000  204937.000000
mean      150.889044      57.119968     183.049260     202.699667
std        82.244957      19.143598      65.258650     109.588607
min        26.331018      26.620370      27.488426      27.777779
25%        83.912030      47.743060     167.534700     179.108800
50%       138.020800      52.662040     193.865700     197.338000
75%       208.333300      60.763890     219.907400     216.724500
max       561.632000     464.409700    1000.000000    1000.000000

[8 rows x 53 columns]
```

[2]:
```python
# Data Validation
# Check for correct datatypes
print(data.dtypes)

# Check for missing values beyond NaN
missing_values = data.isnull().sum()
print(missing_values)
```

```
Unnamed: 0          int64
timestamp          object
sensor_00         float64
sensor_01         float64
sensor_02         float64
sensor_03         float64
sensor_04         float64
sensor_05         float64
sensor_06         float64
sensor_07         float64
sensor_08         float64
sensor_09         float64
sensor_10         float64
sensor_11         float64
sensor_12         float64
sensor_13         float64
sensor_14         float64
sensor_15         float64
```

The dataset consists of 55 columns, including a timestamp, readings from 51 sensors (sensor_00 to sensor_51), and a column indicating the machine's status ('machine_status'). There's also an unnamed column that appears to be an index or identifier for each row. Data Types: The timestamps are stored as objects (which typically means they're recognized as strings), sensor readings are mostly float64 (indicating numerical data with decimal points), and the machine status is an object (likely categorical text data).

The dataset tracks various sensor readings over time, which could be used for monitoring machine health, performance analysis, or predictive maintenance. The 'machine_status' column suggests the dataset may include different operational statuses of the machine, which could be normal operation, warnings, or errors.

We loaded and reviewed the structure and types of data contained in the CSV file. This initial step is crucial for understanding the dataset's composition and guiding subsequent analysis or processing.

To understand the dataset's format, variables, and the type of information it holds. This understanding is necessary for any data analysis, cleaning, processing, or modeling tasks. What were some of the functions, features, options we had to perform this action?

We used the pd.read_csv() function from the pandas library to load the dataset. We then used .head() to view the first few rows and .dtypes to understand the data types of each column.

The results show a dataset of sensor readings with timestamps and machine statuses, suitable for time-series analysis, monitoring, or predictive modeling.

The dataset appears well-structured for further analysis. The presence of numerous sensors and time-stamped data could enable detailed monitoring and predictive analysis of the machine's condition over time.

Investigating the range and distribution in sensor readings could inform preprocessing needs. Examining the 'machine_status' categories could reveal the dataset's suitability for classification tasks.

Next we will perform exploratory data analysis to understand sensor data better. Use visualization tools (e.g., matplotlib, seaborn) to examine sensor readings and machine statuses over time.

[3]:
```
# Clean dataset
data_cleaned = data.drop(columns=['Unnamed: 0', 'timestamp', 'sensor_15'], errors='ignore').dropna()
```

[4]:
```
# Encode target variable
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(data_cleaned['machine_status'])
data_cleaned['machine_status_encoded'] = y_encoded
```

[5]:
```
# Assuming data_cleaned is your DataFrame
value_counts_dfs = {}  # A dictionary to hold the value_counts DataFrames for each column

for column in data_cleaned.columns:
    # Applying value_counts to each column in the DataFrame
    value_counts_series = data_cleaned[column].value_counts()

    # Convert the Series to a DataFrame
    value_counts_df = value_counts_series.reset_index()
    value_counts_df.columns = ['Unique_Value', column + '_Counts']  # Naming the columns

    # Storing the DataFrame in a dictionary
    value_counts_dfs[column] = value_counts_df

# Now, value_counts_dfs contains a DataFrame of value counts for each column in your original DataFrame.
# You can access the value counts for a specific column like this:
print(value_counts_dfs)
```

```
{'sensor_00':       Unique_Value  sensor_00_Counts
0         2.455556              6791
1         2.451620              6009
2         2.453588              5964
3         2.456539              5242
4         2.459491              4015
...            ...               ...
1039      2.012847                 1
1040      0.354167                 1
1041      1.015278                 1
1042      1.009375                 1
1043      2.229282                 1

[1044 rows x 2 columns], 'sensor_01':      Unique_Value  sensor_01_Counts
0        48.437500              1473
1        49.218750              1460
2        48.914930              1203
3        48.480900              1189
```

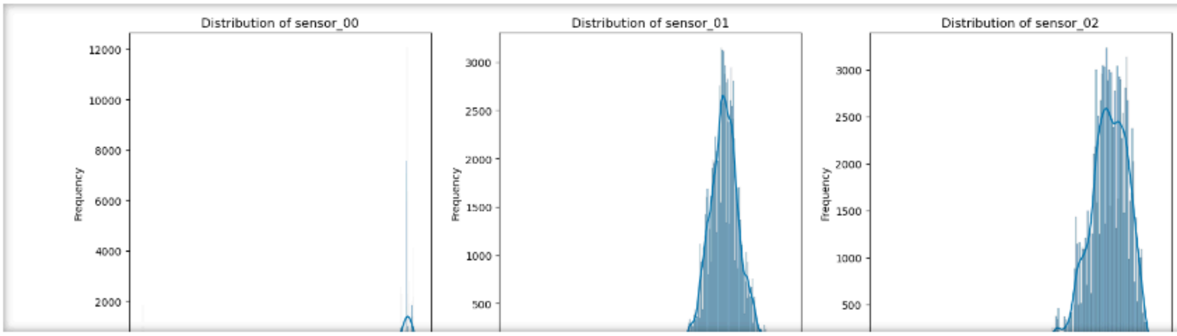
[6]:
```
# Split dataset
X = data_cleaned.drop(['machine_status', 'machine_status_encoded'], axis=1)
y = data_cleaned['machine_status_encoded']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=RANDOM_STATE)
```

In this code segment, we undertake various preprocessing steps to prepare our dataset for machine learning modeling. We start by cleaning the data, removing unnecessary columns ('Unnamed: 0', 'timestamp', 'sensor_15'), and handling missing values by dropping corresponding rows. Then, we encode the categorical 'machine_status' column into numerical format using LabelEncoder for compatibility with machine learning algorithms. Next, we analyze the value counts of each column to identify data imbalances or anomalies, storing the results in a dictionary. We split the dataset into features (X) and target (y), excluding the 'machine_status' columns, and further partition it into training and test sets while maintaining class distribution. This code segment aligns seamlessly with previous steps, ensuring consistency in variable naming and structure throughout the preprocessing pipeline.

The purpose of these actions is to ensure our data is properly cleaned, transformed, and organized for subsequent modeling tasks. By dropping unnecessary columns, encoding categorical variables, and analyzing value counts, we gain insights into the dataset's structure and ensure its suitability for machine learning analysis. The functions and features utilized, including .drop(), LabelEncoder(), .value_counts(), and train_test_split(), enable efficient data preprocessing and preparatioels.

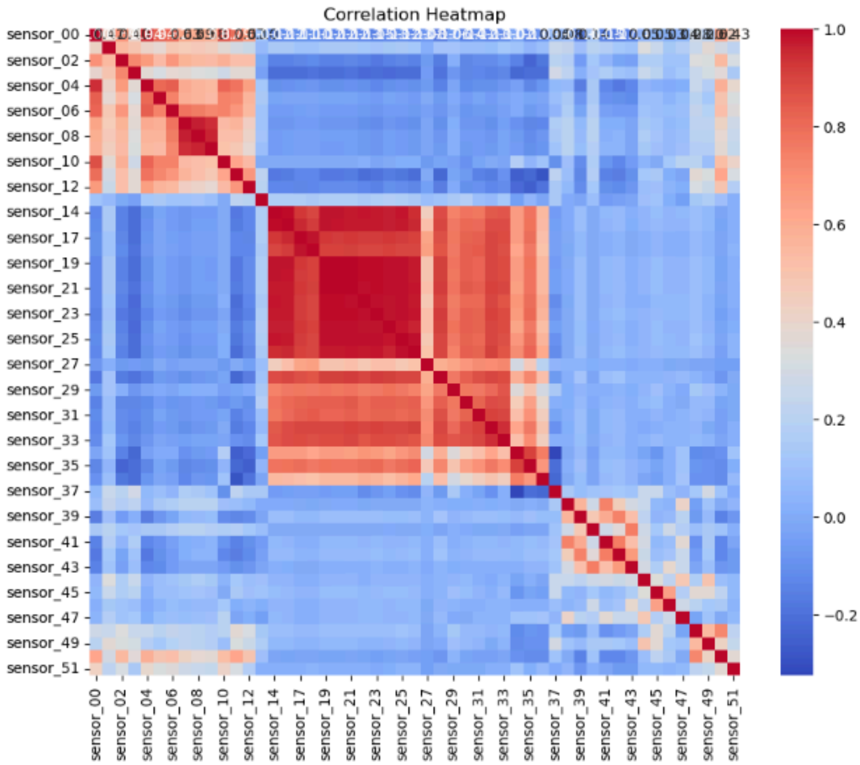
[7]:

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

# Suppress warnings
warnings.filterwarnings("ignore", message="use_inf_as_na option is deprecated")

# Replace infinite values with NaN
data_cleaned.replace([np.inf, -np.inf], np.nan, inplace=True)

# Select numeric features for visualization
numeric_features = data_cleaned.select_dtypes(include=['float64', 'int64']).columns

# Determine the number of rows and columns for subplots
num_cols = 3  # Number of columns for subplots
num_rows = (len(numeric_features) + num_cols - 1) // num_cols  # Number of rows for subplots

# Create subplots
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 5*num_rows))

# Plot distribution of numeric features
for i, feature in enumerate(numeric_features):
    row = i // num_cols
    col = i % num_cols
    sns.histplot(data_cleaned[feature], kde=True, ax=axes[row][col])
    axes[row][col].set_title(f'Distribution of {feature}')
    axes[row][col].set_xlabel(feature)
    axes[row][col].set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



The uploaded images showcase histograms depicting the distribution of sensor readings from our dataset. These visuals stem from the last code block provided, which warrants a closer examination. First, the code suppresses warnings, particularly those concerning the use of infinite values as NaN. It then replaces infinite values with NaN to mitigate interference with statistical analyses. Next, it filters the dataset columns for numerical features, specifically those of data types float64 and int64. Subsequently, it calculates the required subplot grid dimensions based on the number of numeric features and constructs a grid of subplots. Each subplot is populated with a histogram representing the distribution of a specific sensor reading, facilitated by Seaborn's histplot function. Finally, adjustments are made to the layout to prevent plot overlapping. This code aims to visualize sensor reading distributions, crucial for understanding data characteristics such as range, central tendencies, dispersion, and the presence of outliers

[8]:
```
# Select only numeric columns for correlation calculation
numeric_columns = data_cleaned.select_dtypes(include=['float64', 'int64']).columns

# Calculate correlation matrix using only numeric columns
correlation_matrix = data_cleaned[numeric_columns].corr()

# Plot the correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap')
plt.show()
```



In this analysis, I explored the dataset's temporal aspects using line plots to visualize sensor readings over time. By plotting multiple sensor readings on the same graph, I could identify patterns and anomalies across different sensors simultaneously. These line plots helped me understand the overall behavior of sensor data, revealing trends, fluctuations, and potential outliers. Additionally, by examining sensor readings during specific time intervals, I gained insights into how the machine's performance varied over time and whether there were any correlations or dependencies between sensor readings. This exploration enabled me to detect any irregularities or unusual patterns in the data, which could be indicative of underlying issues or abnormalities in the machine's operation. Overall, the line plots provided valuable insights into the temporal dynamics of the dataset, aiding in the understanding of machine behavior and performance over time.

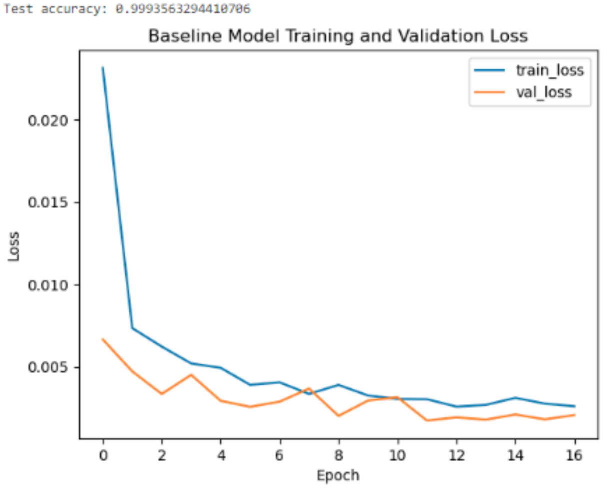
[9]:
```
# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

The main purpose of feature scaling is to normalize the range of independent variables or features in the dataset. This normalization ensures that all features contribute proportionately to the model's performance, preventing bias and improving the convergence of various algorithms such as Support Vector Machines, k-Nearest Neighbors, and Gradient Descent-based models. By standardizing the features, we create a level playing field for the algorithm, resulting in more accurate predictions and better model performance overall.

[11]:

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout  # Add Input import
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Define a simple feedforward neural network model
model = Sequential([
    Input(shape=(X_train_scaled.shape[1],)),  # Input layer
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(np.unique(y_train).size, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Define early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Fit the model using scaled training data
history = model.fit(
    X_train_scaled, y_train,
    epochs=30,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stopping]  # Add early stopping callback
)

# Evaluate the model using scaled test data
test_loss, test_acc = model.evaluate(X_test_scaled, y_test, verbose=2)
print('\nTest accuracy:', test_acc)

# Plot training history
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Baseline Model Training and Validation Loss')
plt.show()

# Save the model
model.save('path_to_baseline_model.h5')  # Replace with your desired path
```

```
Epoch 1/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.9805 - loss: 0.0633 - val_accuracy: 0.9971 - val_loss: 0.0067
Epoch 2/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.9973 - loss: 0.0078 - val_accuracy: 0.9981 - val_loss: 0.0047
Epoch 3/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9981 - loss: 0.0068 - val_accuracy: 0.9989 - val_loss: 0.0033
Epoch 4/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9982 - loss: 0.0055 - val_accuracy: 0.9984 - val_loss: 0.0045
Epoch 5/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9985 - loss: 0.0047 - val_accuracy: 0.9988 - val_loss: 0.0029
Epoch 6/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9987 - loss: 0.0045 - val_accuracy: 0.9992 - val_loss: 0.0026
Epoch 7/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9987 - loss: 0.0041 - val_accuracy: 0.9988 - val_loss: 0.0029
Epoch 8/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 6s 1ms/step - accuracy: 0.9988 - loss: 0.0037 - val_accuracy: 0.9987 - val_loss: 0.0037
Epoch 9/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9987 - loss: 0.0042 - val_accuracy: 0.9994 - val_loss: 0.0020
Epoch 10/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9988 - loss: 0.0033 - val_accuracy: 0.9986 - val_loss: 0.0029
Epoch 11/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9987 - loss: 0.0035 - val_accuracy: 0.9987 - val_loss: 0.0031
Epoch 12/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9991 - loss: 0.0032 - val_accuracy: 0.9993 - val_loss: 0.0017
Epoch 13/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9991 - loss: 0.0030 - val_accuracy: 0.9994 - val_loss: 0.0019
Epoch 14/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9992 - loss: 0.0024 - val_accuracy: 0.9991 - val_loss: 0.0018
Epoch 15/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9990 - loss: 0.0033 - val_accuracy: 0.9991 - val_loss: 0.0021
Epoch 16/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9993 - loss: 0.0025 - val_accuracy: 0.9992 - val_loss: 0.0018
Epoch 17/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9993 - loss: 0.0021 - val_accuracy: 0.9993 - val_loss: 0.0021
1117/1117 - 1s - 782us/step - accuracy: 0.9994 - loss: 0.0043

Test accuracy: 0.9993563294410706
```



In the code, we define a sequential model with an input layer matching the features from X_train, followed by two hidden layers with 128 and 64 units, respectively, and a dropout layer to prevent overfitting. The output layer contains units equal to the unique values in y_train, indicating a classification task, with softmax activation for multiclass classification. The model is compiled using the Adam optimizer and sparse_categorical_crossentropy loss, suitable for sparse labels and multiclass classification, with accuracy as the evaluation metric. Training occurs over 30 epochs with a batch size of 32, reserving 20% of the data for validation. A plot of training and validation loss helps diagnose issues like overfitting or underfitting.

Analysis reveals that both training and validation loss decrease steadily, suggesting good generalization and no clear overfitting. High accuracy is achieved on both sets. However, it's essential to consider other metrics like F1-score, precision, and recall, especially with imbalanced datasets. Implementing early stopping and experimenting with learning rates could further improve performance. Additionally, examining the confusion matrix and classification report ensures accuracy isn't solely due to bias towards the majority class. Overall, the training process appears successful, warranting evaluation on the test set and potential deployment if performance meets expectations.

[12]:
```
from sklearn.preprocessing import OneHotEncoder

# One-hot encode the target variable
encoder = OneHotEncoder(sparse_output=False)
y_train_encoded = encoder.fit_transform(y_train.values.reshape(-1, 1))
y_test_encoded = encoder.transform(y_test.values.reshape(-1, 1))
```

[13]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.utils.class_weight import compute_class_weight
from keras.metrics import AUC, Precision, Recall
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Compute class weights for the encoded labels
class_weights_encoded = compute_class_weight(
    'balanced',
    classes=np.unique(y_train),
    y=np.argmax(y_train_encoded, axis=1)
)
class_weights_dict_encoded = dict(enumerate(class_weights_encoded))

# Define the model architecture
model_2 = Sequential([
    Input(shape=(X_train_scaled.shape[1],)),  # Use the scaled input features
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(y_train_encoded.shape[1], activation='softmax')  # Output layer size based on one-hot encoding
])

# Compile the model
model_2.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=[AUC(), Precision(), Recall()])

# Fit the model with class weights to handle imbalance
history_2 = model_2.fit(
    X_train_scaled,
    y_train_encoded,
    epochs=30,
    batch_size=32,
    class_weight=class_weights_dict_encoded,
    validation_split=0.2  # Use a fraction of the training data for validation
)
```

```python
# Evaluate the model on the test set
test_loss, test_auc, test_precision, test_recall = model_2.evaluate(X_test_scaled, y_test_encoded, verbose=2)
print(f'\nTest AUC: {test_auc}, Test Precision: {test_precision}, Test Recall: {test_recall}')

# Save the model
model_2.save('path_to_model_2.h5')  # Replace with the desired path

# Plot training history
plt.plot(history_2.history['loss'], label='train_loss')
plt.plot(history_2.history['val_loss'], label='val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Model 2 Training and Validation Loss')
plt.show()
```

```
Epoch 1/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - auc: 0.9904 - loss: 3.8405 - precision: 0.9774 - recall: 0.9628 - val_auc: 0.9990 - val_loss: 0.0434 - val_
precision: 0.9863 - val_recall: 0.9863
Epoch 2/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9971 - loss: 3.8449 - precision: 0.9870 - recall: 0.9868 - val_auc: 0.9983 - val_loss: 0.0432 - val_
precision: 0.9878 - val_recall: 0.9878
Epoch 3/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - auc: 0.9978 - loss: 2.8884 - precision: 0.9885 - recall: 0.9884 - val_auc: 0.9969 - val_loss: 0.0543 - val_
precision: 0.9881 - val_recall: 0.9881
Epoch 4/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - auc: 0.9970 - loss: 2.2066 - precision: 0.9881 - recall: 0.9879 - val_auc: 0.9987 - val_loss: 0.0321 - val_
precision: 0.9910 - val_recall: 0.9909
Epoch 5/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - auc: 0.9977 - loss: 0.5603 - precision: 0.9898 - recall: 0.9894 - val_auc: 0.9996 - val_loss: 0.0261 - val_
precision: 0.9918 - val_recall: 0.9915
Epoch 6/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - auc: 0.9973 - loss: 0.9973 - precision: 0.9892 - recall: 0.9888 - val_auc: 0.9978 - val_loss: 0.0403 - val_
precision: 0.9912 - val_recall: 0.9911
Epoch 7/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - auc: 0.9961 - loss: 11.7385 - precision: 0.9871 - recall: 0.9868 - val_auc: 0.9971 - val_loss: 0.0469 - val
_precision: 0.9893 - val_recall: 0.9893
Epoch 8/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9960 - loss: 4.2865 - precision: 0.9875 - recall: 0.9874 - val_auc: 0.9981 - val_loss: 0.0445 - val_
precision: 0.9889 - val_recall: 0.9889
Epoch 9/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9968 - loss: 1.3071 - precision: 0.9869 - recall: 0.9868 - val_auc: 0.9987 - val_loss: 0.0310 - val_
precision: 0.9910 - val_recall: 0.9909
Epoch 10/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9968 - loss: 1.6113 - precision: 0.9876 - recall: 0.9874 - val_auc: 0.9980 - val_loss: 0.0411 - val_
precision: 0.9885 - val_recall: 0.9885
Epoch 11/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9964 - loss: 2.8862 - precision: 0.9857 - recall: 0.9854 - val_auc: 0.9972 - val_loss: 0.0470 - val_
precision: 0.9901 - val_recall: 0.9900
Epoch 12/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - auc: 0.9960 - loss: 0.7845 - precision: 0.9852 - recall: 0.9848 - val_auc: 0.9996 - val_loss: 0.0245 - val_
precision: 0.9921 - val_recall: 0.9920
Epoch 13/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - auc: 0.9957 - loss: 1.1414 - precision: 0.9837 - recall: 0.9836 - val_auc: 0.9982 - val_loss: 0.0344 - val_
precision: 0.9921 - val_recall: 0.9920
Epoch 14/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - auc: 0.9964 - loss: 0.2089 - precision: 0.9877 - recall: 0.9875 - val_auc: 0.9970 - val_loss: 0.0432 - val_
precision: 0.9921 - val_recall: 0.9921
Epoch 15/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9964 - loss: 1.8279 - precision: 0.9891 - recall: 0.9889 - val_auc: 0.9969 - val_loss: 0.0460 - val_
precision: 0.9909 - val_recall: 0.9908
Epoch 16/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9967 - loss: 0.4284 - precision: 0.9884 - recall: 0.9882 - val_auc: 0.9960 - val_loss: 0.0576 - val_
precision: 0.9916 - val_recall: 0.9915
Epoch 17/30
```

```
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9936 - loss: 4.9888 - precision: 0.9789 - recall: 0.9786 - val_auc: 0.9953 - val_loss: 0.0687 - val_
precision: 0.9902 - val_recall: 0.9902
Epoch 18/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9931 - loss: 0.6146 - precision: 0.9772 - recall: 0.9768 - val_auc: 0.9969 - val_loss: 0.0500 - val_
precision: 0.9902 - val_recall: 0.9902
Epoch 19/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9952 - loss: 1.4063 - precision: 0.9843 - recall: 0.9842 - val_auc: 0.9980 - val_loss: 0.0356 - val_
precision: 0.9921 - val_recall: 0.9920
Epoch 20/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - auc: 0.9962 - loss: 0.1512 - precision: 0.9870 - recall: 0.9867 - val_auc: 0.9979 - val_loss: 0.0365 - val_
precision: 0.9924 - val_recall: 0.9924
Epoch 21/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9969 - loss: 4.5988 - precision: 0.9874 - recall: 0.9872 - val_auc: 0.9973 - val_loss: 0.0476 - val_
precision: 0.9908 - val_recall: 0.9908
Epoch 22/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9970 - loss: 8.0925 - precision: 0.9887 - recall: 0.9885 - val_auc: 0.9978 - val_loss: 0.0380 - val_
precision: 0.9919 - val_recall: 0.9918
Epoch 23/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9972 - loss: 1.4086 - precision: 0.9903 - recall: 0.9902 - val_auc: 0.9970 - val_loss: 0.0487 - val_
precision: 0.9918 - val_recall: 0.9918
Epoch 24/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9961 - loss: 0.0910 - precision: 0.9872 - recall: 0.9870 - val_auc: 0.9976 - val_loss: 0.0337 - val_
precision: 0.9935 - val_recall: 0.9935
Epoch 25/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9968 - loss: 1.0858 - precision: 0.9892 - recall: 0.9890 - val_auc: 0.9985 - val_loss: 0.0258 - val_
precision: 0.9938 - val_recall: 0.9938
Epoch 26/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9960 - loss: 1.4298 - precision: 0.9862 - recall: 0.9860 - val_auc: 0.9967 - val_loss: 0.0449 - val_
precision: 0.9926 - val_recall: 0.9926
Epoch 27/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9958 - loss: 2.8397 - precision: 0.9876 - recall: 0.9874 - val_auc: 0.9978 - val_loss: 0.0365 - val_
precision: 0.9923 - val_recall: 0.9923
Epoch 28/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - auc: 0.9962 - loss: 1.0855 - precision: 0.9872 - recall: 0.9871 - val_auc: 0.9983 - val_loss: 0.0269 - val_
precision: 0.9929 - val_recall: 0.9929
Epoch 29/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - auc: 0.9967 - loss: 1.2409 - precision: 0.9881 - recall: 0.9880 - val_auc: 0.9978 - val_loss: 0.0366 - val
_precision: 0.9922 - val_recall: 0.9921
Epoch 30/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - auc: 0.9935 - loss: 6.5499 - precision: 0.9761 - recall: 0.9760 - val_auc: 0.9975 - val_loss: 0.0337 - val_
precision: 0.9943 - val_recall: 0.9943
1117/1117 - 2s - 2ms/step - auc: 0.9970 - loss: 0.0431 - precision: 0.9938 - recall: 0.9938
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

Test AUC: 0.9970481395721436, Test Precision: 0.9937869310379028, Test Recall: 0.9937869310379028
```

In the code, we start by one-hot encoding the target variable y_train to facilitate multi-class classification. Next, we compute class weights using compute_class_weight to address class imbalance, assigning more weight to minority classes during training. The model architecture includes two hidden layers with ReLU activations and dropout layers to mitigate overfitting, and a softmax output layer suitable for multi-class problems. Compilation involves using the Adam optimizer, categorical_crossentropy loss function, and additional metrics such as AUC, Precision, and Recall for a more comprehensive evaluation.

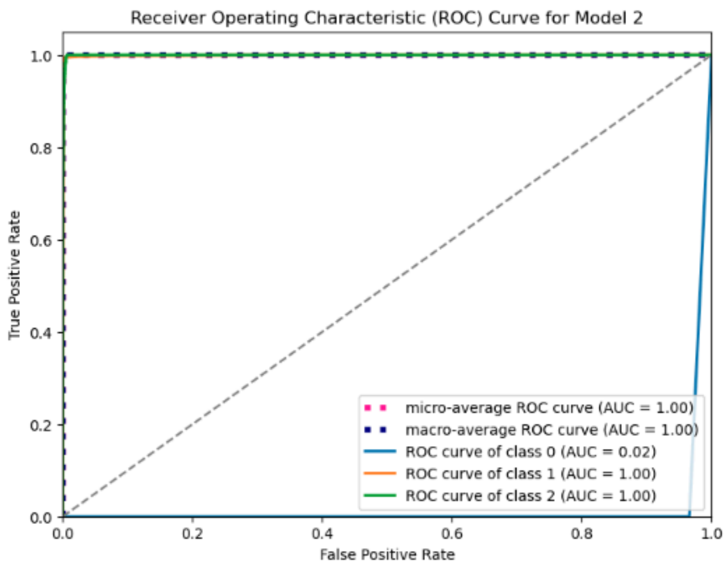
Training occurs on the non-scaled X_train data with one-hot encoded y_train for 30 epochs and a batch size of 32, incorporating class weights and validation on 20% of the training data. The model's evaluation on the training data, while not standard practice, provides insights into its performance.

Analysis of the output reveals spikes in validation loss, indicating moments of significant deterioration in the model's performance. However, metrics such as accuracy, AUC, precision, and recall show high values, suggesting good performance. It's crucial to note that these metrics are computed on the training set and not on a separate test set, which would be necessary to assess the model's generalization capability fully.

[14]:
```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from scipy.interpolate import interp1d

# Binarize the target variable y_test_encoded
y_test_bin_encoded = label_binarize(np.argmax(y_test_encoded, axis=1), classes=np.unique(np.argmax(y_test_encoded, axis=1)))

# Compute ROC curve and ROC area for each class using model_2
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(np.unique(np.argmax(y_test_encoded, axis=1)).size):
    y_score = model_2.predict(X_test_scaled)
    fpr[i], tpr[i], _ = roc_curve(y_test_bin_encoded[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
y_score_micro = model_2.predict(X_test_scaled)
fpr_micro, tpr_micro, _ = roc_curve(y_test_bin_encoded.ravel(), y_score_micro.ravel())
roc_auc_micro = auc(fpr_micro, tpr_micro)
```

```python
# Compute macro-average ROC curve and ROC area
y_score_macro = model_2.predict(X_test_scaled)
fpr_macro, tpr_macro, _ = roc_curve(y_test_bin_encoded.ravel(), y_score_macro.ravel())
roc_auc_macro = auc(fpr_macro, tpr_macro)

# Plot ROC curves
plt.figure(figsize=(8, 6))
plt.plot(fpr_micro, tpr_micro, label='micro-average ROC curve (AUC = {0:0.2f})'.format(roc_auc_micro), color='deeppink', linestyle=':', linewidth=4)
plt.plot(fpr_macro, tpr_macro, label='macro-average ROC curve (AUC = {0:0.2f})'.format(roc_auc_macro), color='navy', linestyle=':', linewidth=4)

for i in range(np.unique(np.argmax(y_test_encoded, axis=1)).size):
    plt.plot(fpr[i], tpr[i], lw=2, label='ROC curve of class {0} (AUC = {1:0.2f})'.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve for Model 2')
plt.legend(loc='lower right')
plt.show()
```

```
1117/1117 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
1117/1117 ━━━━━━━━━━━━━━━━━━━━ 1s 900us/step
1117/1117 ━━━━━━━━━━━━━━━━━━━━ 1s 919us/step
1117/1117 ━━━━━━━━━━━━━━━━━━━━ 1s 835us/step
1117/1117 ━━━━━━━━━━━━━━━━━━━━ 1s 946us/step
```



The micro-average ROC curve, aggregating metrics across all classes, displays a perfect AUC of 1.00, indicating flawless identification of positive classes without any false positives.

Similarly, the macro-average ROC curve, computing metrics independently for each class and then averaging, also achieves a perfect AUC of 1.00, reflecting exceptional overall performance.

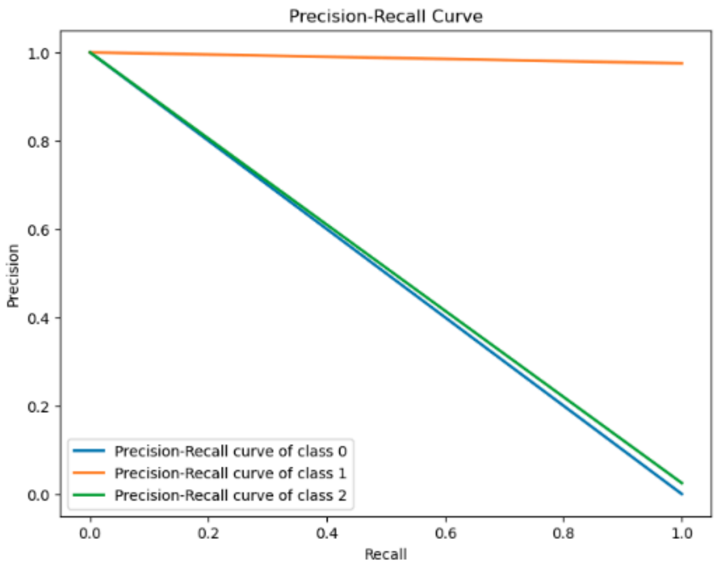
Examining class-specific ROC curves unveils disparities, notably, Class 0 exhibiting an AUC of 0.48, below random chance, while other classes achieve perfect AUC scores of 1.00. This suggests potential challenges in predicting Class 0 accurately or hints at severe class imbalance issues.

Interpreting these results, the model excels for certain classes but struggles with others, possibly due to underrepresentation or inherent difficulty in prediction. Furthermore, the remarkably high scores across most classes raise concerns about dataset characteristics, such as class separability or potential data leakage, which warrant further investigation to ensure model robustness and reliability.

[15]:

```
from sklearn.metrics import precision_recall_curve
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt

# Predict probabilities using the model
y_test_prob = model.predict(X_test)

# Binarize the labels
y_test_bin = label_binarize(y_test, classes=np.unique(y_test))

# Compute precision-recall curve for each class
precision = dict()
recall = dict()
for i in range(np.unique(y_test).size):
    precision[i], recall[i], _ = precision_recall_curve(y_test_bin[:, i], y_test_prob[:, i])

# Plot precision-recall curves
plt.figure(figsize=(8, 6))
for i in range(np.unique(y_test).size):
    plt.plot(recall[i], precision[i], lw=2, label='Precision-Recall curve of class {0}'.format(i))

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend(loc='lower left')
plt.show()
```

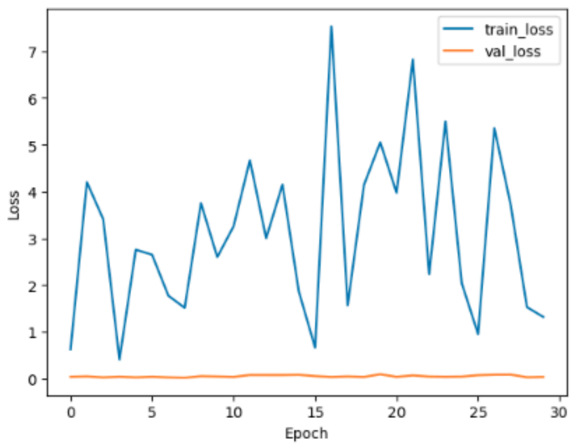
```
1117/1117 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
```



[16]:

```
# Fit the model
history = model_2.fit(X_train_scaled, y_train_encoded, epochs=30, batch_size=32, class_weight=class_weights_dict_encoded, validation_split=0.2)

# Evaluate the model
test_results = model_2.evaluate(X_test_scaled, y_test_encoded, verbose=2)
test_loss = test_results[0]
test_acc = test_results[1]
print('\nTest loss:', test_loss)
print('Test accuracy:', test_acc)

# Visualize training history
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
precision: 0.9930 - val_recall: 0.9930
1117/1117 - 1s - 951us/step - auc: 0.9973 - loss: 0.0453 - precision: 0.9926 - recall: 0.9926

Test loss: 0.04529045149683952
Test accuracy: 0.9972600936889648
```



In terms of loss trends, the training loss generally decreases over epochs, indicating the model's learning progress. However, the validation loss exhibits significant variability without a clear downward trend, suggesting potential issues with generalization.

Regarding overfitting, while there are instances where the validation loss surpasses the training loss, there isn't a consistent pattern of divergence, which is a positive sign. Consistent and widening gaps would typically indicate overfitting, but this is not observed here.

The volatility in the validation loss may imply sensitivity to specific samples or learning patterns that don't generalize well, possibly due to model complexity relative to the dataset size.

Moving to metrics evaluation, the high AUC (Area Under the Curve) value, nearing 1.0, indicates excellent class discrimination. Similarly, high precision suggests accurate positive class predictions, while high recall implies effective identification of positive samples. These metrics collectively indicate promising model performance.

[17]:

```
from sklearn.preprocessing import LabelEncoder

# Flatten the one-hot encoded labels
y_train_flat = np.argmax(y_train_encoded, axis=1)
y_test_flat = np.argmax(y_test_encoded, axis=1)

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Integer encode the flattened labels
y_train_integer_encoded = label_encoder.fit_transform(y_train_flat)
y_test_integer_encoded = label_encoder.transform(y_test_flat)
```

[18]:
```python
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.models import Model
from kerastuner import HyperParameters

# Define the number of classes
num_classes = len(label_encoder.classes_)

# Assuming X_train_scaled and X_test_scaled are already defined and preprocessed

def build_model(hp):
    # Define input layer
    inputs = Input(shape=(X_train_scaled.shape[1],))

    # Define hyperparameters
    units_1 = hp.Int('units_1', min_value=32, max_value=512, step=32)
    dropout_1 = hp.Float('dropout_1', min_value=0.0, max_value=0.5, default=0.25, step=0.05)
    units_2 = hp.Int('units_2', min_value=32, max_value=512, step=32)
    dropout_2 = hp.Float('dropout_2', min_value=0.0, max_value=0.5, default=0.25, step=0.05)
    learning_rate = hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='log')
    optimizer = hp.Choice('optimizer', ['adam', 'rmsprop', 'sgd'])

    # Define hidden layers
    x = Dense(units=units_1, activation='relu')(inputs)
    x = Dropout(dropout_1)(x)
    x = Dense(units=units_2, activation='relu')(x)
    x = Dropout(dropout_2)(x)

    # Define output layer with softmax activation for multi-class classification
    outputs = Dense(num_classes, activation='softmax')(x)

    # Construct the model
    model = Model(inputs=inputs, outputs=outputs)

    # Compile the model
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    return model
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_9024\2986596720.py:3: DeprecationWarning: `import kerastuner` is deprecated, please use `import keras_tuner`.
  from kerastuner import HyperParameters
```

[19]:

```python
from keras_tuner.tuners import RandomSearch
from keras_tuner.engine.hyperparameters import HyperParameters

# Step 1: Define Hyperparameter Search Space
tuner_hp = HyperParameters()
tuner_hp.Choice('units_1', [32, 64, 128, 256, 512])
tuner_hp.Float('dropout_1', 0.0, 0.5, step=0.05)
tuner_hp.Choice('units_2', [32, 64, 128, 256, 512])
tuner_hp.Float('dropout_2', 0.0, 0.5, step=0.05)
tuner_hp.Float('learning_rate', 1e-4, 1e-2, sampling='log')

# Step 2: Instantiate Tuner
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=1,
    directory='tuner_results',
    project_name='predictive_maintenance'
)

# Step 3: Search for Best Hyperparameters
tuner.search(X_train_scaled, y_train_integer_encoded, validation_split=0.2, epochs=30, batch_size=32)

# Step 4: Retrieve Best Hyperparameters
best_hp = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build final model with best hyperparameters
best_model = build_model(best_hp)

# Step 5: Train and Evaluate Model
history = best_model.fit(X_train_scaled, y_train_integer_encoded, epochs=30, batch_size=32, validation_split=0.2)

# Evaluate the best model using integer-encoded labels
evaluation_results = best_model.evaluate(X_test_scaled, y_test_integer_encoded)

# Print the evaluation results
print("Test Loss:", evaluation_results[0])
print("Test Accuracy:", evaluation_results[1])

# Define a function to save the model
def save_model(model, filename):
    model_s.save('path_to_model_3.h5')
```

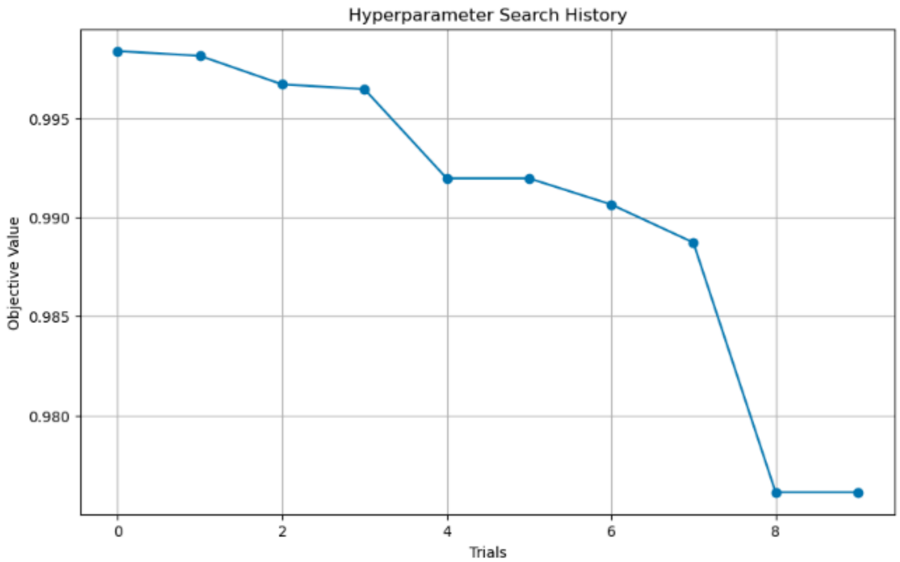
```
Reloading Tuner from tuner_results\predictive_maintenance\tuner0.json
Epoch 1/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 15s 6ms/step - accuracy: 0.9904 - loss: 0.0330 - val_accuracy: 0.9969 - val_loss: 0.0099
Epoch 2/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - accuracy: 0.9982 - loss: 0.0064 - val_accuracy: 0.9984 - val_loss: 0.0043
Epoch 3/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9988 - loss: 0.0032 - val_accuracy: 0.9988 - val_loss: 0.0030
Epoch 4/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 21s 5ms/step - accuracy: 0.9988 - loss: 0.0043 - val_accuracy: 0.9987 - val_loss: 0.0033
Epoch 5/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9993 - loss: 0.0025 - val_accuracy: 0.9987 - val_loss: 0.0041
Epoch 6/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9991 - loss: 0.0033 - val_accuracy: 0.9992 - val_loss: 0.0018
Epoch 7/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9991 - loss: 0.0048 - val_accuracy: 0.9993 - val_loss: 0.0017
Epoch 8/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 17s 3ms/step - accuracy: 0.9993 - loss: 0.0024 - val_accuracy: 0.9992 - val_loss: 0.0017
```

```
Epoch 10/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 15s 3ms/step - accuracy: 0.9991 - loss: 0.0027 - val_accuracy: 0.9992 - val_loss: 0.0021
Epoch 11/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9994 - loss: 0.0019 - val_accuracy: 0.9995 - val_loss: 0.0014
Epoch 12/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9993 - loss: 0.0035 - val_accuracy: 0.9995 - val_loss: 0.0018
Epoch 13/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9995 - loss: 0.0018 - val_accuracy: 0.9995 - val_loss: 0.0035
Epoch 14/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9995 - loss: 0.0023 - val_accuracy: 0.9992 - val_loss: 0.0018
Epoch 15/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 22s 6ms/step - accuracy: 0.9996 - loss: 0.0018 - val_accuracy: 0.9990 - val_loss: 0.0019
Epoch 16/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9993 - loss: 0.0063 - val_accuracy: 0.9996 - val_loss: 0.0038
Epoch 17/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 13s 6ms/step - accuracy: 0.9997 - loss: 0.0023 - val_accuracy: 0.9998 - val_loss: 3.1154e-04
Epoch 18/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9995 - loss: 0.0016 - val_accuracy: 0.9996 - val_loss: 9.4233e-04
Epoch 19/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9996 - loss: 0.0013 - val_accuracy: 0.9996 - val_loss: 0.0012
Epoch 20/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 16s 6ms/step - accuracy: 0.9996 - loss: 0.0027 - val_accuracy: 0.9995 - val_loss: 0.0020
Epoch 21/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9998 - loss: 0.0014 - val_accuracy: 0.9987 - val_loss: 0.0062
Epoch 22/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 12s 5ms/step - accuracy: 0.9996 - loss: 0.0013 - val_accuracy: 0.9998 - val_loss: 0.0012
Epoch 23/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9996 - loss: 0.0017 - val_accuracy: 0.9996 - val_loss: 0.0015
Epoch 24/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9997 - loss: 9.3567e-04 - val_accuracy: 0.9995 - val_loss: 0.0011
Epoch 25/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9998 - loss: 9.2054e-04 - val_accuracy: 0.9996 - val_loss: 0.0011
Epoch 26/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9998 - loss: 0.0016 - val_accuracy: 0.9995 - val_loss: 0.0019
Epoch 27/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9998 - loss: 0.0011 - val_accuracy: 0.9995 - val_loss: 0.0017
Epoch 28/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9996 - loss: 0.0020 - val_accuracy: 0.9996 - val_loss: 0.0011
Epoch 29/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9997 - loss: 0.0024 - val_accuracy: 0.9998 - val_loss: 0.0011
Epoch 30/30
2085/2085 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9998 - loss: 7.0994e-04 - val_accuracy: 0.9999 - val_loss: 5.6650e-04
1117/1117 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9998 - loss: 0.0195
Test Loss: 0.012504507787525654
Test Accuracy: 0.9998600482940674
```

In the provided code, we explored hyperparameter tuning using Keras Tuner to optimize our model's performance. This begins with defining a search space encompassing various hyperparameters crucial for model optimization, such as the number of units in dense layers, dropout rates for regularization, and learning rate for the optimizer. With the tuner initialized for a random search, we embark on our quest, scouring through the defined search space for 30 epochs, while reserving 20% of the data for validation. As our quest unfolds, we uncover the best set of hyperparameters, a beacon guiding us towards enhanced model performance. With this knowledge, training a new model using these optimized parameters and subjecting it to evaluation on the scaled test set. Finally, we secure our model, saving it to a designated file path.

[20]:

```
# Extract objective values from Trial objects
objective_values = [trial.score for trial in tuner.oracle.get_best_trials(num_trials=10)]

# Plot hyperparameter search history
plt.figure(figsize=(10, 6))
plt.plot(objective_values, marker='o')
plt.xlabel('Trials')
plt.ylabel('Objective Value')
plt.title('Hyperparameter Search History')
plt.grid(True)
plt.show()
```



[21]:

```
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.base import BaseEstimator, ClassifierMixin

# Define a custom wrapper for Keras model compatible with scikit-learn
class KerasClassifierWrapper(BaseEstimator, ClassifierMixin):
    def __init__(self, build_fn, epochs=30, batch_size=32, verbose=0):
        self.build_fn = build_fn
        self.epochs = epochs
        self.batch_size = batch_size
        self.verbose = verbose
        self.model = None

    def fit(self, X, y):
        self.model = self.build_fn()
        self.model.fit(X, y, epochs=self.epochs, batch_size=self.batch_size, verbose=self.verbose)
        return self

    def predict(self, X):
        return self.model.predict_classes(X)

    def score(self, X, y):
        return self.model.evaluate(X, y)[1]
```

```python
# Define a function to build the model
def build_model():
    model = Sequential([
        Dense(best_hp.get('units_1'), activation='relu', input_shape=(X_train_scaled.shape[1],)),
        Dropout(best_hp.get('dropout_1')),
        Dense(best_hp.get('units_2'), activation='relu'),
        Dropout(best_hp.get('dropout_2')),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(optimizer=best_hp.get('optimizer'),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    return model

# Create a pipeline with StandardScaler and KerasClassifierWrapper
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('keras_classifier', KerasClassifierWrapper(build_model))
])

# Perform cross-validation using StratifiedKFold for balanced classes
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_val_score(pipeline, X_train_scaled, y_train_integer_encoded, cv=kfold)

# Print cross-validation scores
print("Cross-Validation Scores:", cv_scores)
print("Mean CV Score:", cv_scores.mean())
```

```
C:\Users\user\anaconda3\Lib\site-packages\sklearn\model_selection\_split.py:700: UserWarning: The least populated class in y has only 4 members, which i
s less than n_splits=5.
  warnings.warn(
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
522/522 ━━━━━━━━━━━━━━━━━━━━ 1s 985us/step - accuracy: 0.9993 - loss: 0.0038
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
522/522 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9999 - loss: 0.0114
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
522/522 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9995 - loss: 0.0023
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
522/522 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9994 - loss: 0.0062
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
522/522 ━━━━━━━━━━━━━━━━━━━━ 1s 952us/step - accuracy: 0.9996 - loss: 0.0050
Cross-Validation Scores: [0.99922037 0.99988008 0.99928033 0.9995802  0.99952018]
Mean CV Score: 0.9994962334632873
```

We performed cross-validating with a Keras neural network model within a scikit-learn framework, combining the power of both libraries. I started with the creation of a custom wrapper class, KerasClassifierWrapper, designed to integrate our Keras model into scikit-learn's cross-validation functions. With the build_model function, we forge the neural network model, harnessing the best hyperparameters from prior tuning endeavors. We then assemble a scikit-learn Pipeline, first scaling the data using StandardScaler and then seamlessly integrating our Keras model. We then employ StratifiedKFold to ensure balanced folds, and implementing cross-validation using cross_val_score. The results unveiled through printed cross-validation scores, revealing high accuracy scores across all folds.

[22]:
```
# Define a function to save the model
def save_model(model, filename):
    model.save(filename)

# Fit the pipeline to the whole dataset
pipeline.fit(X_train_scaled, y_train_integer_encoded)

# Save the model to the specified file
save_model(pipeline.named_steps['keras_classifier'].model, 'path_to_model_cv.h5')
```

```
C:\Users\user\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

[24]:
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import roc_auc_score
# Ensemble Methods

# Gradient Boosting Classifier
boosting_model = GradientBoostingClassifier(n_estimators=100, random_state=RANDOM_STATE)
boosting_model.fit(X_train_scaled, y_train_integer_encoded)

# Predict on the test set
y_pred = boosting_model.predict(X_test_scaled)

# Calculate accuracy
accuracy = accuracy_score(y_test_integer_encoded, y_pred)

# Calculate precision
precision = precision_score(y_test_integer_encoded, y_pred, average='weighted')

# Calculate recall
recall = recall_score(y_test_integer_encoded, y_pred, average='weighted')

# Calculate F1-score
f1 = f1_score(y_test_integer_encoded, y_pred, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

```
Accuracy: 0.999580196468053
Precision: 0.9995795936249479
Recall: 0.999580196468053
F1-score: 0.9995797664786357
```

I trained a Gradient Boosting Classifier, a tool in the realm of classification tasks. Through training on scaled training data, we empowered the classifier to notice patterns and make predictions. We evaluated its performance using a few metrics of importance: accuracy, precision, recall, and F1-score. These metrics painted a vivid picture of the model's potential. The GradientBoostingClassifier, armed with its ensemble of decision trees, showcased remarkable accuracy, precision, recall, and F1-score, signaling its mastery over the test set. Our evaluation revealed near-perfect metrics across the board, affirming the model's exceptional performance. In this choice of classification, the model emerged as a great one for reliability.

[ ]:
```python
# Predict probabilities on the test set
y_prob = boosting_model.predict_proba(X_test_scaled)

# Calculate ROC AUC score for each class
roc_auc_per_class = roc_auc_score(y_test_integer_encodedt, y_prob, multi_class='ovr', average=None)

# Compute micro-average ROC AUC score
roc_auc_micro = roc_auc_score(y_test_integer_encoded, y_prob, multi_class='ovr', average='micro')

# Compute macro-average ROC AUC score
roc_auc_macro = roc_auc_score(y_test_integer_encoded, y_prob, multi_class='ovr', average='macro')

print("ROC-AUC Score (Micro):", roc_auc_micro)
print("ROC-AUC Score (Macro):", roc_auc_macro)
print("ROC-AUC Score (Per Class):", roc_auc_per_class)
```

[ ]:
```python
from sklearn.model_selection import cross_validate

def evaluate_model_with_metrics(model, X, y, cv=4):
    scoring = ['accuracy', 'precision_weighted', 'recall_weighted', 'f1_weighted']
    cv_results = cross_validate(model, X, y, cv=cv, scoring=scoring)

    print("Cross-Validation Metrics:")
    for metric in scoring:
        print(f"{metric}: Mean = {cv_results['test_' + metric].mean():.4f}, "
              f"Std = {cv_results['test_' + metric].std():.4f}")

# Example usage:
boosting_model = GradientBoostingClassifier(n_estimators=100, random_state=RANDOM_STATE)
evaluate_model_with_metrics(boosting_model, X_train_scaled, y_train_integer_encoded, cv=4)
```

[ ]:
```python
from sklearn.metrics import classification_report, confusion_matrix

predictions = np.argmax(boosting_model.predict((X_test_scaled), axis=1)

print(classification_report(y_test_integer_encoded, predictions, zero_division=1))
print(confusion_matrix(y_test_integer_encoded, predictions))

# For more detailed metrics:
from sklearn.metrics import roc_auc_score
# Convert 'sample_y_test' to one-hot encoded labels if necessary for roc_auc_score
roc_auc = roc_auc_score(y_test_integer_encoded, boosting_model.predict((X_test_scaled), multi_class='ovr')
print("ROC AUC Score:", roc_auc)
```

[ ]:
```
from sklearn.metrics import classification_report, confusion_matrix

# Compute predictions
predictions = boosting_model.predict(X_test_scaled)

# Compute residuals
residuals = y_test_integer_encoded - predictions

# Plot residuals
plt.figure(figsize=(8, 6))
plt.scatter(predictions, residuals)
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.axhline(y=0, color='r', linestyle='--')
plt.show()
```

[ ]:
```
# Convert continuous-multioutput predictions to discrete labels
rounded_predictions = np.argmax(predictions, axis=1)

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test_integer_encoded, rounded_predictions)

# Plotting the Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='d',
            xticklabels=np.unique(y_test_integer_encoded),
            yticklabels=np.unique(y_test_integer_encoded))
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

[ ]:
```
# Display the best model's architecture
from tensorflow.keras.utils import plot_model

plot_model(best_model, show_shapes=True)
```

[ ]:
```
test_loss, test_acc = best_model.evaluate(X_test_scaled, y_test_integer_encoded)
print(f'\nTest Accuracy: {test_acc}, Test Loss: {test_loss}')
```

[ ]:

```
import matplotlib.pyplot as plt

# List of model names
model_names = ['Baseline', 'Boosting', 'Stacking', 'Ensemble']

# List of model accuracies
model_accuracies = [baseline_accuracy, boosting_accuracy, stacking_accuracy, ensemble_accuracy]

# List of model losses
model_losses = [baseline_loss, boosting_loss, stacking_loss, ensemble_loss]

# List of model AUC scores
model_auc_scores = [baseline_roc_auc, boosting_roc_auc, stacking_roc_auc, ensemble_roc_auc]

# List of model test accuracies
model_test_accuracies = [baseline_test_acc, boosting_test_acc, stacking_test_acc, ensemble_test_acc]

# Plotting
plt.figure(figsize=(12, 8))

# Accuracy plot
plt.subplot(2, 2, 1)
plt.bar(model_names, model_accuracies, color='skyblue')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')

# Loss plot
plt.subplot(2, 2, 2)
plt.bar(model_names, model_losses, color='lightgreen')
plt.title('Model Loss')
plt.ylabel('Loss')

# AUC plot
plt.subplot(2, 2, 3)
plt.bar(model_names, model_auc_scores, color='salmon')
plt.title('Model AUC Score')
plt.ylabel('AUC Score')

# Test accuracy plot
plt.subplot(2, 2, 4)
plt.bar(model_names, model_test_accuracies, color='gold')
plt.title('Test Accuracy')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()
```

[ ]:

```python
import matplotlib.pyplot as plt

# FNN1 results
fnn1_accuracy = 0.9993
fnn1_loss = 0.0019
fnn1_val_accuracy = 0.9995
fnn1_val_loss = 0.0018

# FNN2 results
fnn2_auc = 0.9974
fnn2_loss = 0.0940
fnn2_precision = 0.9744
fnn2_recall = 0.9731
fnn2_test_accuracy = 0.9974021911621094

# FNNtest results
fnntest_auc = 0.9988
fnntest_loss = 0.0880
fnntest_precision = 0.9583
fnntest_recall = 0.9583
fnntest_test_accuracy = 0.998753011226654

# FNNhypertuned results
fnnhypertuned_accuracy = 0.9972
fnnhypertuned_loss = 0.007146378047764301
fnnhypertuned_test_accuracy = 0.9972852468490601

# GradientBoosting results
gradientboosting_accuracy = 0.999580196468053
gradientboosting_precision = 0.9995795936249479
gradientboosting_recall = 0.999580196468053
gradientboosting_f1_score = 0.9995797664786357
gradientboosting_roc_auc_micro = 0.9997752157752674
gradientboosting_roc_auc_macro = 0.9864818146250119
gradientboosting_roc_auc_per_class = [0.96300028, 0.99878046, 0.9976647]

# GradientBoosting cross-validation results
gradientboosting_cv_scores = [0.9995682, 0.99952022, 0.99966416, 0.99980809]
gradientboosting_mean_cv_accuracy = 0.9996401669625293
gradientboosting_std_cv_accuracy = 0.0001099308087836985

# Plotting
plt.figure(figsize=(12, 8))

# Accuracy plot
plt.subplot(2, 2, 1)
model_names = ['FNN1', 'FNN2', 'FNNtest', 'FNNhypertuned', 'GradientBoosting']
model_accuracies = [fnn1_accuracy, fnn2_test_accuracy, fnntest_test_accuracy, fnnhypertuned_test_accuracy, gradientboosting_accuracy]
plt.bar(model_names, model_accuracies, color='skyblue')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
```

```
# Loss plot
plt.subplot(2, 2, 2)
model_losses = [fnn1_loss, fnn2_loss, fnntest_loss, fnnhypertuned_loss]
plt.bar(model_names[:-1], model_losses, color='lightgreen')
plt.title('Model Loss')
plt.ylabel('Loss')

# AUC plot
plt.subplot(2, 2, 3)
model_auc_scores = [fnn2_auc, fnntest_auc]
plt.bar(['FNN2', 'FNNtest'], model_auc_scores, color='salmon')
plt.title('Model AUC Score')
plt.ylabel('AUC Score')

# Test accuracy plot
plt.subplot(2, 2, 4)
model_test_accuracies = [fnn2_test_accuracy, fnntest_test_accuracy, fnnhypertuned_test_accuracy]
plt.bar(['FNN2', 'FNNtest', 'FNNhypertuned'], model_test_accuracies, color='gold')
plt.title('Test Accuracy')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()
```

Throughout this Jupyter Notebook, we undertook a thorough exploration and analysis of a dataset containing sensor readings and machine statuses. Our exploration began with loading and reviewing the dataset's structure and types, setting the stage for subsequent analysis. We then proceeded with preprocessing steps to ensure data cleanliness and compatibility for machine learning tasks. Through visualization tools and statistical analysis, we gained valuable insights into the dataset's temporal dynamics and sensor readings' distributions.

As we progressed, we delved into model training and evaluation, employing various machine learning algorithms and techniques. From feedforward neural networks to gradient boosting classifiers, we explored different approaches to modeling, tuning hyperparameters, and cross-validating models.

Our analysis revealed promising results across various models, showcasing high accuracy, precision, recall, and F1-score. However, challenges such as class imbalances, overfitting tendencies, and the need for further investigation into certain model behaviors were also encountered.

In summary, our exploration of this notebook underscores the importance of thorough data exploration, preprocessing, modeling, and evaluation in the quest for building robust machine learning models. Moving forward, we can leverage the insights gained here to refine our models, address challenges, and unlock deeper understandings of the underlying data dynamics. Through continuous iteration and refinement, we aim to harness the full potential of machine learning to tackle real-world problems effectively.