Project Documentation

```
import pandas as pd
import warnings
import matplotlib.pyplot as plt
warnings.filterwarnings("ignore")
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
from sklearn.metrics import accuracy_score
import numpy as np
```

Modules Used

- `pandas`: Used to handle dataframes for data manipulation.
- `warnings`: Helps in handling warnings and ignoring them for cleaner outputs.
- `matplotlib`: Used for plotting and visualizing data.
- `warnings.filterwarnings("ignore")`: Ignores warnings to avoid cluttering the output during execution.
- `train_test_split`: Part of `sklearn`, it splits the dataset into training and testing sets.
- `Sequential`: Creates a linear stack of layers for building a neural network.
- `LSTM`: Long Short-Term Memory layer used for sequence prediction tasks.

- `Dense`: Fully connected layer used at the end of the model for classification/regression tasks.
- `Dropout`: Regularization technique to prevent overfitting.
- `Adam`: An adaptive learning rate optimizer.
- `l2`: L2 regularization term to penalize large weights and avoid overfitting.
- `StandardScaler`: Scales the features of the dataset by removing the mean and scaling to unit variance.
- `accuracy_score`: Measures the accuracy of the model's predictions.

```
Q1 = df2.quantile(0.25)
Q3 = df2.quantile(0.75)
IQR = Q3 - Q1
outliers = ((df2 < (Q1 - 1.5 * IQR)) | (df2 > (Q3 + 1.5 * IQR))
outliers.sum()
```

- `Q1 = df2.quantile(0.25)`: Calculates the first quartile (Q1), which is the 25th percentile of the data.
- `Q3 = df2.quantile(0.75)`: Calculates the third quartile (Q3), which is the 75th percentile of the data.
- `IQR = Q3 - Q1`: Computes the Interquartile Range (IQR) as the difference between Q3 and Q1.
- `outliers = ((df2 < (Q1 - 1.5 * IQR)) | (df2 > (Q3 + 1.5 * IQR)))`: Creates a boolean DataFrame to identify outliers by checking if any values in `df2` are below `(Q1 - 1.5 * IQR)` or above `(Q3 + 1.5 * IQR)`.
- `outliers.sum()`: Sums up the boolean values to count the number of outliers detected.

```
rows_with_outliers = outliers.any(axis=1)
df3 = df2[~rows_with_outliers]
```

- `rows_with_outliers = outliers.any(axis=1)`: Determines which rows in `df2` contain any outlier by checking along the columns.
- `df3 = df2[~rows_with_outliers]`: Filters the original DataFrame `df2` to exclude rows with outliers. This creates a new DataFrame `df3` that only contains data without outliers.

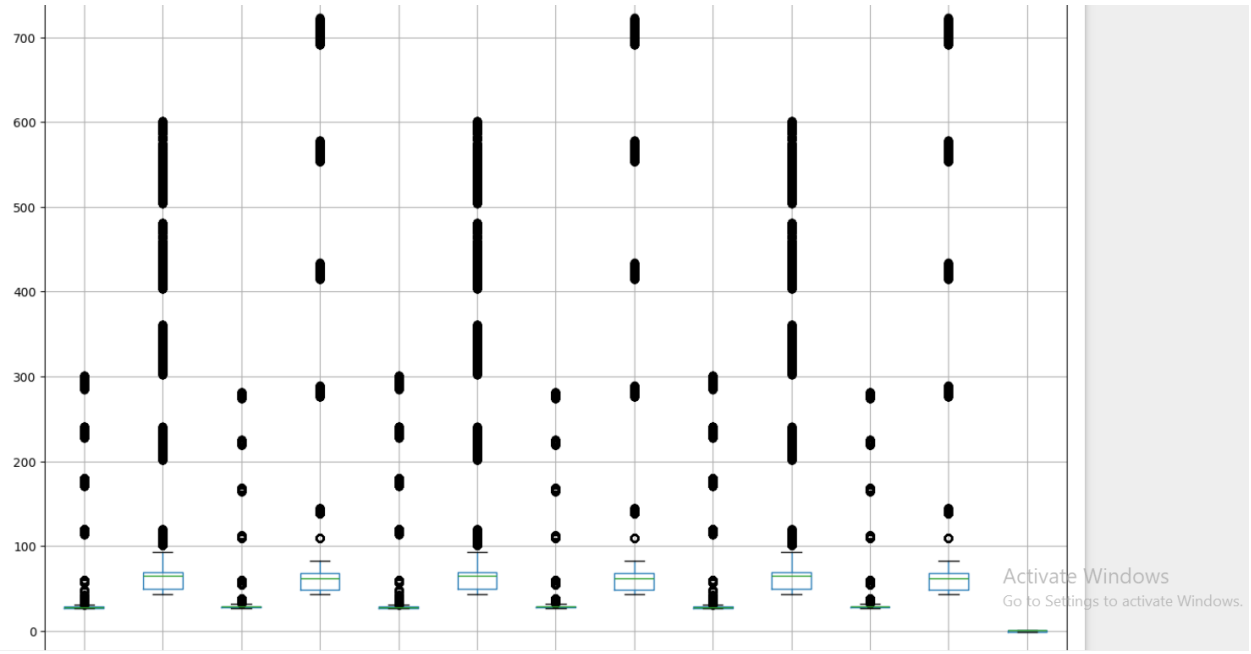
Key Parameters

- `0.25` **and** `0.75`: Percentile values used to calculate Q1 and Q3.
- `1.5`: The constant multiplier used in the IQR method to define outliers (`1.5 * IQR`).

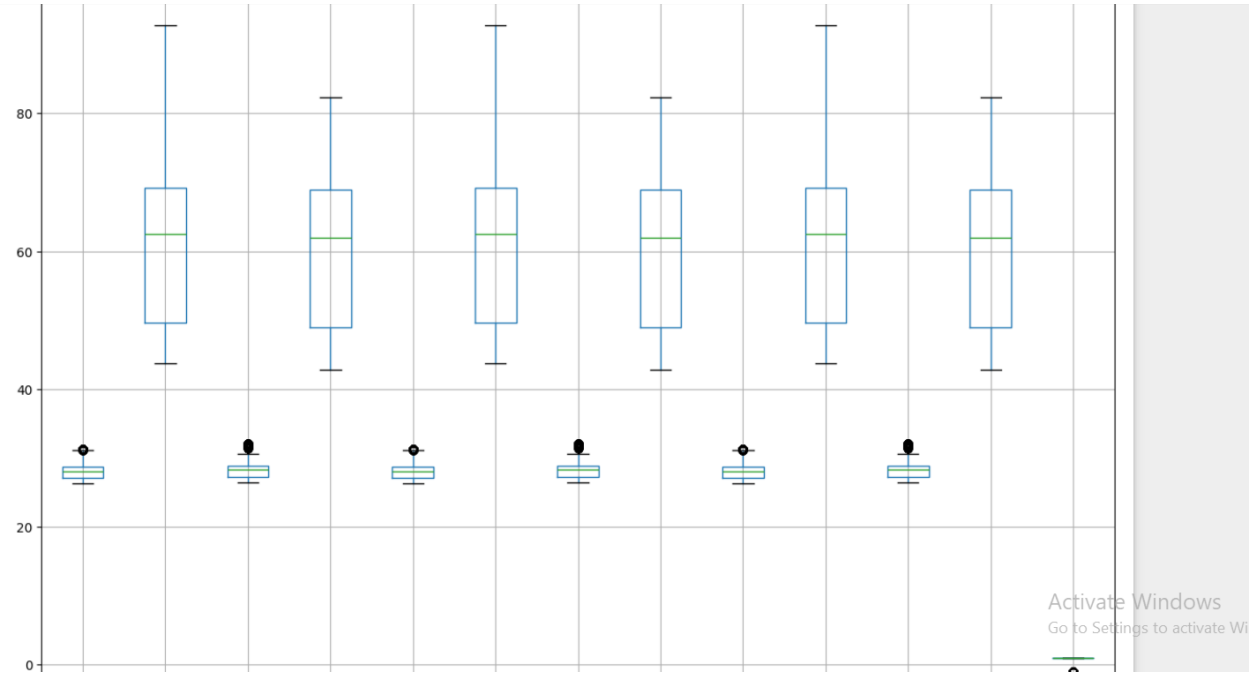
Visual and Statistical Checks

- **Boxplot**: Provides a visual way to detect outliers by showing the spread of data, median, and outliers as points beyond the whiskers.
- **IQR Method**: A statistical method to detect outliers by calculating the range where most data should fall, and identifying data outside this range as outliers.

BoxPlot Before Removing the Outliers



BoxPlot AfterRemoving the Outliers



`df3['Label'] = df3['Label'].map({1: 0, -1: 1})` is used to invert the labels in the DataFrame `df3`.

- **Mapping Values in a DataFrame Column**:
  - The `map()` function is used to transform values in a pandas Series (in this case, the column `Label` of `df3`).
- **Mapping Logic**:
  - `{1: 0, -1: 1}`: This dictionary specifies how to map the original values in the `Label` column.
    - `1` is mapped to `0`: This changes all instances where `Label` was `1` to `0`.
    - `1` is mapped to `1`: This changes all instances where `Label` was `1` to `1`.

```
x = df3.drop(columns=['Label'])
y = df3['Label']
```

- `x = df3.drop(columns=['Label'])`:
  - This line is used to separate the features (`X` values) from the target variable (`Label`).
  - `df3.drop(columns=['Label'])` creates a new DataFrame `x` that contains all the columns of `df3` except for the `Label` column. These columns are the features used to train the model.
  - `x` will be a DataFrame where each row represents a sample (or observation) and each column represents a feature (independent variable) used for prediction.
- `y = df3['Label']`:
  - This line extracts the target variable (label) column from `df3`.
  - `df3['Label']` is a Series representing the labels (`0` or `1` in this case), which is what we want to predict.
  - `y` will be a pandas Series containing the target variable values for each row in `df3`.

```
X_train, X_test, y_train, y_test = train_test_split(x, y, test_s
```

`train_test_split()`

- This function from `sklearn.model_selection` is used to split the dataset into training and testing sets.
- `x`: The features DataFrame.
- `y`: The target variable Series.
- `test_size=0.2`: Specifies that 20% of the data should be used for testing and 80% for training. This is a common ratio, but you can adjust it based on your dataset size and the problem requirements.
- `random_state=42`: Sets a seed for the random number generator to ensure reproducibility of the data split. This makes sure that each time you run the code with the same seed, the split will be the same.

```
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

class_weights = compute_class_weight('balanced', classes=np.unic

class_weights_dict = dict(enumerate(class_weights))

print(class_weights_dict)
```

Explanation:

- `compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)`:
  - This function calculates the weights for each class to handle class imbalance.
  - `classes=np.unique(y_train)`: The unique classes from the target variable `y_train`. It computes weights based on the frequency of each class in `y_train`.

- `'balanced'` specifies that the weights should be calculated inversely proportional to the class frequencies, so the class with fewer instances will get a higher weight to compensate for its under-representation.
- `y_train`: The training labels.

- `class_weights_dict = dict(enumerate(class_weights))`:
  - Converts the computed class weights from a numpy array into a dictionary where the keys are class labels (indices) and values are the corresponding weights.
  - `enumerate(class_weights)`: This function pairs each weight with its class index to create a dictionary.
  - `print(class_weights_dict)`: Prints the class weights dictionary, which can be useful for verification or logging.

```
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Explanation:

- `StandardScaler()`:
  - Initializes the `StandardScaler`, which will be used to normalize the features.
- `fit_transform(X_train)`:
  - Fits the scaler on the `X_train` data and then transforms it.
  - The `fit_transform` method calculates the mean and standard deviation of each feature in `X_train` and scales each feature by subtracting the mean and dividing by the standard deviation.
  - The result is a scaled version of `X_train` where all features have a mean of 0 and a standard deviation of 1.
- `transform(X_test)`:

- Uses the parameters (mean and standard deviation) calculated from `X_train` to transform the `X_test` data.
- It scales `X_test` using the same transformation learned from `X_train`, ensuring that the test data is on the same scale as the training data.

```
model = Sequential([
    Dense(128, activation='relu',input_shape=(X_trained_scaled.s
    Dropout(0.3),
    Dense(64,activation="relu"),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')

])

model.compile(optimizer='adam',loss='binary_crossentropy',metric

history = model.fit(X_trained_scaled,y_train,epochs=3,class_weig
model.compile(optimizer='adam', loss='binary_crossentropy', metr
history = model.fit(X_trained_scaled, y_train, epochs=3, class_w
```

- `Dense(128, activation='relu', input_shape=(X_trained_scaled.shape[1],))`:
  - This is the first hidden layer with 128 neurons (units) and ReLU activation function.
  - `input_shape=(X_trained_scaled.shape[1],)`: Specifies the number of input features, which corresponds to the number of features in `X_train_scaled`.
  - The ReLU (Rectified Linear Unit) activation function is used here because it's widely used in hidden layers due to its ability to introduce non-linearity and allow the model to learn complex patterns.
  - `Dropout(0.3)`: Regularization technique to prevent overfitting. It randomly sets 30% of the input units to 0 at each update during training.
- `Dense(64, activation='relu')`:

    - The second hidden layer with 64 neurons and ReLU activation.
    - Another dropout layer with `Dropout(0.3)` to randomly drop out 30% of the units during training.
- `Dense(32, activation='relu')`:
    - The third hidden layer with 32 neurons and ReLU activation.
- `Dense(1, activation='sigmoid')`:
    - The output layer with a single neuron for binary classification.
    - The sigmoid activation function is used here because it outputs a value between 0 and 1, which is suitable for binary classification tasks.
- `optimizer='adam'`:
    - Adam is used as the optimizer, which combines the advantages of stochastic gradient descent with adaptive learning rates for each parameter. It adjusts the learning rate dynamically during training.
- `loss='binary_crossentropy'`:
    - This is the loss function used to measure the difference between the predicted output and the actual labels. Binary cross-entropy (log loss) is suitable for binary classification tasks because it penalizes incorrect predictions based on the probability of those predictions.
- `metrics=['accuracy']`:
    - This lists the evaluation metrics to be used. In this case, we're using accuracy to monitor how well the model predicts the labels. It gives the fraction of correct predictions out of the total predictions.

`model.fit()` :

- This function trains the model on the provided data.
- `X_trained_scaled`: The features after scaling using `StandardScaler`.
- `y_train`: The target labels for the training data.
- `epochs=3`: The number of iterations (epochs) to pass through the entire dataset. 3 epochs is typically a starting point; you might adjust it based on convergence and performance.

- `class_weight=class_weights_dict`: Specifies the class weights for training. This parameter helps balance the importance of different classes during training, addressing class imbalance by giving more weight to the minority class.

```
loss, acc = model.evaluate(X_test_scaled, y_test)
X_train_rnn = X_trained_scaled.reshape(X_trained_scaled.shape[0]
X_test_rnn = X_test_scaled.reshape(X_test_scaled.shape[0], 3, 4)
```

Explanation:

- `model.evaluate(X_test_scaled, y_test)`:
  - This function evaluates the model on the test dataset.
  - `X_test_scaled`: The scaled features of the test set after applying the same `StandardScaler` transformation as for the training set. This ensures that the test data is on the same scale as the training data.
  - `y_test`: The true labels for the test set.
  - The function returns two values:
    - `loss`: The value of the loss function for the test data. This can be used to assess how well the model performs on unseen data.
    - `acc`: The accuracy of the model on the test set, which is the proportion of correctly predicted labels.
- `X_trained_scaled.reshape(X_trained_scaled.shape[0], 3, 4)`:
  - This reshapes the training data from a flat feature set into a format suitable for RNNs:
    - `X_trained_scaled.shape[0]`: The number of samples (rows) in the dataset.
    - `3`: Number of time steps. This is an arbitrary choice for this example; the actual number should match your problem's requirements.
    - `4`: Number of features. This corresponds to the feature count from the `X_trained_scaled` DataFrame.

- The reshaped `X_train_rnn` now has the shape `(samples, time steps, features)`, which matches the expected input format for RNNs where each sample corresponds to a sequence of observations.

- `X_test_scaled.reshape(X_test_scaled.shape[0], 3, 4)`:
  - This reshapes the test data using the same logic as for the training data. The number of samples remains unchanged, but the reshaping will provide sequences of observations for the test data, allowing it to be fed into the RNN model.

```
model = Sequential([
    LSTM(128, activation='tanh', input_shape=(3, 4), return_sequ
    Dropout(0.3),
    LSTM(64, activation='tanh'),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

Explanation:

- `LSTM(128, activation='tanh', input_shape=(3, 4), return_sequences=True)`:
  - This layer defines an LSTM with 128 units, `tanh` activation function, and expects an input shape of `(3, 4)`.
  - `return_sequences=True`: This parameter is used to return the full sequence of output at each time step, which is useful when you want the model to learn from the sequence of outputs.

- `Dropout(0.3)`:
  - Regularization to prevent overfitting by randomly dropping 30% of the LSTM units during training.

- `LSTM(64, activation='tanh')`:
  - This is the second LSTM layer with 64 units and `tanh` activation.

- `Dropout(0.3)`:

    - Another dropout layer with 30% dropout to prevent overfitting.
- `Dense(32, activation='relu')`:
    - A fully connected layer with 32 neurons and ReLU activation.
- `Dense(1, activation='sigmoid')`:
    - The output layer with a single neuron and a sigmoid activation function to predict binary labels (0 or 1).

```
model.compile(optimizer='adam', loss='binary_crossentropy', metr
```

- `optimizer='adam'`:
    - The Adam optimizer is used for training the model. It adjusts the learning rate dynamically based on the training process.
- `loss='binary_crossentropy'`:
    - This is the loss function used to calculate the difference between predicted and actual labels. Binary cross-entropy is appropriate for binary classification tasks.
- `metrics=['accuracy']`:
    - Evaluation metric to monitor the model's performance during training. Accuracy is calculated as the proportion of correct predictions.

```
history = model.fit(X_train_rnn, y_train, epochs=3, class_weight
```

`model.fit(X_train_rnn, y_train, epochs=3, class_weight=class_weights_dict)`

- This function trains the model with the following parameters:
- `X_train_rnn`: The reshaped training data suitable for RNN (LSTM) input.
- `y_train`: The target labels for the training data.
- `epochs=3`: Number of epochs to train the model. This value can be adjusted based on the model's convergence and performance.
- `class_weight=class_weights_dict`: Specifies the class weights to handle class imbalance. This helps to balance the importance of each class during training.

```
loss, acc = model.evaluate(X_test_rnn, y_test)
```

`model.evaluate(X_test_rnn, y_test)`

- This evaluates the model on the test dataset.
- `X_test_rnn`: The reshaped test data for RNN input.
- `y_test`: The true labels for the test data.
- The function returns the loss and accuracy of the model on the test data, providing an assessment of how well the model performs on unseen data.

```
input_data = np.array([
    [27.49, 67.81, 55.2, 67.78, 27.49, 67.81, 55.14, 67.87, 27.4
])
input_data_scaled = scaler.transform(input_data)
input_data2 = input_data_scaled.reshape(1, 3, 4)
predicted_output = model.predict(input_data2)
predicted_class = (predicted_output > 0.5).astype(int)
prediction_label = "Normal" if predicted_class == 0 else "Faulty
```

Explanation:

- `input_data`: This is a new data point that needs to be classified using the trained model. It's reshaped into the required input format for the LSTM model.
- The data point provided here consists of 12 features, which should correspond to the feature count expected by the LSTM model after scaling and reshaping.
- `scaler.transform(input_data)`: Scales the new data point using the same scaling process applied to the training data, ensuring that the new data point is on the same scale as the training data.
- `input_data2 = input_data_scaled.reshape(1, 3, 4)`: Reshapes the new data point to match the expected input shape for the LSTM model. Here, 1 represents the batch size, 3 represents the number of time steps, and 4 represents the number of features.

- `model.predict(input_data2)`: Predicts the output for the new data point. This returns a probability value between 0 and 1, representing the likelihood of the sample belonging to the positive class.
- `predicted_class`: Converts the probability output into a binary label (0 or 1) using a threshold of 0.5. If the probability is greater than 0.5, it's classified as 1; otherwise, it's classified as 0.
- `prediction_label`: Maps the predicted binary label to a human-readable label: "Normal" for class 0 and "Faulty" for class 1.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam

def build_and_train_model(units_1, units_2, dropout_1, dropout_2
    model = Sequential([
        LSTM(units_1, activation='tanh', input_shape=(3, 4), ret
        Dropout(dropout_1),
        LSTM(units_2, activation='tanh'),
        Dropout(dropout_2),
        Dense(32, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=Adam(learning_rate=learning_rate), l

    model.fit(X_train_rnn, y_train, validation_split=0.2, epochs

    _, test_accuracy = model.evaluate(X_test_rnn, y_test, verbos

    return model, test_accuracy

combinations = [
    {'units_1': 128, 'units_2': 64, 'dropout_1': 0.2, 'dropout_2
    {'units_1': 128, 'units_2': 64, 'dropout_1': 0.3, 'dropout_2
    {'units_1': 64, 'units_2': 32, 'dropout_1': 0.2, 'dropout_2
]
```

```
best_accuracy = 0
best_model = None

for params in combinations:
    print(f"Testing combination: {params}")
    model, test_accuracy = build_and_train_model(**params)
    print(f"Test Accuracy: {test_accuracy}")

    if test_accuracy > best_accuracy:
        best_accuracy = test_accuracy
        best_model = model

print(f"Best Test Accuracy: {best_accuracy}")
```

Function `build_and_train_model(units_1, units_2, dropout_1, dropout_2, learning_rate)`

This function creates and trains an LSTM model based on specified hyperparameters. The key steps involved are:

1. **Model Definition**:
    - A `Sequential` model is initialized.
    - The first LSTM layer (`LSTM(units_1, activation='tanh', input_shape=(3, 4), return_sequences=True)`) accepts input data with shape `(3, 4)` which corresponds to (samples, time steps, features). It returns sequences (`return_sequences=True`), which is useful for multi-layer LSTM models.
    - `Dropout(dropout_1)`: Introduces dropout regularization to prevent overfitting.
    - The second LSTM layer (`LSTM(units_2, activation='tanh')`) further processes the data.
    - `Dropout(dropout_2)`: Another dropout layer with a different dropout rate.
    - A fully connected layer (`Dense(32, activation='relu')`) adds non-linearity to the network.

- The output layer (`Dense(1, activation='sigmoid')`) predicts the binary class (0 or 1) using a sigmoid activation function.

2. **Model Compilation**:
   - The model is compiled using the Adam optimizer with a learning rate specified by `learning_rate`.
   - The loss function is set to `binary_crossentropy` since this is a binary classification problem.
   - The evaluation metric is `accuracy` to measure the performance.
3. **Model Training**:
   - The `model.fit()` method is used to train the model with the following parameters:
     - `X_train_rnn`: The reshaped training data formatted for LSTM input.
     - `y_train`: The true labels for the training data.
     - `validation_split=0.2`: Splits 20% of the training data for validation to monitor model performance during training.
     - `epochs=3`: Specifies the number of epochs for which the model should be trained.
     - `verbose=0`: Suppresses verbose output, providing a cleaner interface without training logs.
4. **Model Evaluation**:
   - The model's performance is evaluated using the `model.evaluate()` method on the test dataset (`X_test_rnn` and `y_test`).
   - The test accuracy is stored and returned by the function.
5. **Return Values**:
   - The trained model object and the test accuracy score are returned.

```
from sklearn.metrics import classification_report, confusion_mat

y_pred = (model.predict(X_test_rnn) > 0.5).astype("int32")
```

```
print(classification_report(y_test,y_pred,target_names=["Normal"

print(confusion_matrix(y_test,y_pred))
```

- **Prediction on Test Data**:
  - `y_pred = (model.predict(X_test_rnn) > 0.5).astype("int32")`:
    - This line generates predictions using the trained LSTM model on the test set (`X_test_rnn`).
    - The prediction values are passed through a sigmoid function, which produces probabilities.
    - The prediction is then binarized by checking if the probability is greater than 0.5 (thresholding).
    - `astype("int32")` converts the binary predictions to integer type for further evaluation.
- **Classification Report**:
  - `print(classification_report(y_test, y_pred, target_names=["Normal","Faulty"]))`:
    - The `classification_report` function from `sklearn` is used to evaluate the model performance.
    - `y_test` represents the true labels from the test dataset.
    - `y_pred` are the predicted labels.
    - `target_names=["Normal","Faulty"]` provides the labels for the classes, allowing the report to clearly indicate performance metrics for both classes.
    - The report includes precision, recall, f1-score, and support for each class, which helps in understanding the performance for individual classes.
- **Confusion Matrix**:
- `print(confusion_matrix(y_test, y_pred))`:

- The `confusion_matrix` function computes the confusion matrix from the true and predicted labels.
- It provides a summary of the correct and incorrect predictions:
  - **True Positives (TP)**: Correctly predicted Faulty cases.
  - **True Negatives (TN)**: Correctly predicted Normal cases.
  - **False Positives (FP)**: Incorrectly predicted Faulty cases as Normal.
  - **False Negatives (FN)**: Incorrectly predicted Normal cases as Faulty.

```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

class_names = ["Normal","Faulty"]

plt.figure(figsize=(6,4))
sns.heatmap(cm, annot=True, fmt="d",cmap="Blues",xticklabels=cla
plt.title("Confustion Matrix")
plt.ylabel("Actual Class")
plt.xlabel("Predicted Class")
plt.show()
```

Heatmap of the Confusion Matrix:

- `cm = confusion_matrix(y_test, y_pred)`: Computes the confusion matrix.
- `class_names = ["Normal","Faulty"]`: The labels used to define the classes.
- `plt.figure(figsize=(6,4))`: Sets the size of the plot.
- `sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names, yticklabels=class_names)`:
  - This line generates a heatmap to visualize the confusion matrix.
  - `annot=True` adds annotations with the count in each cell.

    - `fmt="d"` ensures integer formatting for the annotations.
    - `cmap="Blues"` sets the color map for the heatmap.
    - `xticklabels` and `yticklabels` set the labels for the X-axis (Predicted) and Y-axis (Actual) respectively.
- `plt.title("Confusion Matrix")`: Adds a title to the plot.
- `plt.ylabel("Actual Class")` and `plt.xlabel("Predicted Class")`: Labels the axes for clarity.
- `plt.show()`: Displays the heatmap.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense,Dropout
from tensorflow.keras.initializers import HeNormal, GlorotUnifor

model = Sequential([
    LSTM(128,activation='tanh',kernel_initializer=GlorotUniform
    Dropout(0.3),
    LSTM(64, activation='tanh',kernel_initializer=GlorotUniform
    Dropout(0.3),
    Dense(32,activation='relu',kernel_initializer=HeNormal()),
    Dense(1,activation='sigmoid',kernel_initializer=GlorotUnifor

])

model.compile(optimizer='adam',loss='binary_crossentropy',metric
history = model.fit(X_train_rnn,y_train, epochs=3, class_weight=
```

- **Model Architecture**:
    - **Sequential()**: Initializes a sequential model where layers are stacked one after another.
    - **LSTM Layer**:
        - `LSTM(128, activation='tanh', kernel_initializer=GlorotUniform(), input_shape=(3,4), return_sequences=True)`:

- **128** units: The number of units in the LSTM layer, determining the complexity and depth of the learned features.
- **activation='tanh'**: The activation function used in LSTM units, which helps in learning the input-to-state mapping.
- **kernel_initializer=GlorotUniform()**: The initializer for the weights of the kernel. The Glorot Uniform initializer ensures that the weights are initialized uniformly which prevents the model from getting stuck in local minima during training.
- **input_shape=(3,4)**: The shape of the input data (number of time steps, number of features).
- **return_sequences=True**: The LSTM layer returns a sequence (a sequence of outputs for each input at each time step).

- **Dropout Layer**:
  - `Dropout(0.3)`: Introduces a dropout of 30%, which helps in regularizing the model and preventing overfitting by randomly dropping 30% of the input units during each training iteration.
- **Second LSTM Layer**:
  - `LSTM(64, activation='tanh', kernel_initializer=GlorotUniform())`:
    - **64** units: The number of units in the LSTM layer.
    - **activation='tanh'**: The activation function used in LSTM units.
    - **kernel_initializer=GlorotUniform()**: The weights of the kernel are initialized using the Glorot Uniform initializer to maintain uniform distribution.
- **Dropout Layer**:
  - `Dropout(0.3)`: Another dropout layer with 30% dropout, which is applied after the second LSTM layer.
- **Dense Layer**:
  - `Dense(32, activation='relu', kernel_initializer=HeNormal())`:
    - **32 units**: The number of neurons in the dense layer.

- **activation='relu'**: The activation function used to introduce non-linearity into the model.
- **kernel_initializer=HeNormal()**: The initializer for the weights of the kernel. He Normal initializer is used here because it is well-suited for ReLU activations and helps in faster convergence.

- **Output Dense Layer**:
  - `Dense(1, activation='sigmoid', kernel_initializer=GlorotUniform())`:
    - **1 unit**: The output unit for binary classification.
    - **activation='sigmoid'**: The activation function used for binary classification tasks to constrain the output between 0 and 1.
    - **kernel_initializer=GlorotUniform()**: The weights of the kernel are initialized uniformly using the Glorot Uniform initializer.

- **Model Compilation**:
  - `model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])`:
    - **optimizer='adam'**: The Adam optimizer is used for minimizing the binary cross-entropy loss.
    - **loss='binary_crossentropy'**: The binary cross-entropy loss function is used to evaluate the model's performance in terms of binary classification.
    - **metrics=['accuracy']**: Accuracy is used as a metric to monitor how well the model performs in classification.
- **Model Training**:
  - `history = model.fit(X_train_rnn, y_train, epochs=3, class_weight=class_weights_dict)`:
    - **X_train_rnn**: Training data reshaped for the LSTM model (`samples, time steps, features`).
    - **y_train**: The corresponding labels for the training data.
    - **epochs=3**: The model is trained for 3 epochs.

        - **class_weight=class_weights_dict**: Applies class weights to handle class imbalance during training.

- **Model Evaluation**:
    - `loss, acc = model.evaluate(X_test_rnn, y_test)`:
        - **X_test_rnn**: Test data reshaped for the LSTM model.
        - **y_test**: The corresponding labels for the test data.
        - `loss`: The loss value computed on the test set.
        - `acc`: The accuracy of the model on the test set.

Explanation:

- **Initializer Choices**:
    - `GlorotUniform()`: Suitable for weight initialization in the first LSTM and output layer, as it helps in maintaining weight variance across layers.
    - `HeNormal()`: Best suited for the dense layers with ReLU activation functions, as it takes the square root of 2 times the variance to prevent vanishing gradient problems.
- **Dropout**:
    - Applied after each LSTM layer with a dropout rate of 0.3 to improve generalization and reduce overfitting by randomly setting 30% of the weights to zero.
- **Optimizer**:
    - `Adam` optimizer, a popular choice in deep learning, combines the advantages of both Adagrad and RMSprop optimizers and is known for its efficiency and convergence speed.
- **Loss Function**:
    - `binary_crossentropy` is used for binary classification problems, indicating the likelihood that the predictions differ from the actual outputs.
- **Metrics**:

- `accuracy` is used to monitor the model's performance during training. It is a common metric for classification tasks.

Initializer	Description	Use Cases	Impact on Training
GlorotUniform()	Initializes weights uniformly within the range	General use for deep networks, often recommended for hidden layers with non-sigmoid activations like ReLU.	Helps in faster convergence and prevents vanishing/exploding gradients.
GlorotNormal()	Initializes weights normally with mean 0 and standard deviation	When a normal distribution is preferred over uniform. Also suitable for hidden layers with ReLU activations.	Prevents gradients from vanishing or exploding in deeper networks.
HeNormal()	Initializes weights normally with mean 0 and standard deviation	For networks using ReLU or its variants (e.g., LeakyReLU).	Accelerates training and improves model convergence.
HeUniform()	Initializes weights uniformly within the range	Similar to HeNormal(), but can be used when a uniform distribution is desired.	Works well with ReLU activation functions, accelerates convergence.
Zeros()	Initializes weights to zero.	Used for bias weights or when bias initialization is preferred for specific tasks.	Can lead to slower convergence and poor model performance due to zero gradients.

Ones()	Initializes weights to one.	Very specific use cases where uniform output is desired, e.g., bias layers.	Leads to zero gradient issues during training, resulting in poor learning.
RandomNormal(mean, stddev)	Initializes weights normally with a given mean and standard deviation.	Use when initial values need a specific distribution, not just 0 or 1.	Allows better control over initial weights' distribution and properties.
RandomUniform(minval, maxval)	Initializes weights uniformly within a specific range.	When there is a need for weights to be in a specific interval.	Helps maintain uniform distribution of weights, useful in specific activation functions or data properties.

Equation:

GlorotUniform():

$$x = \sqrt{\frac{6}{inputs + outputs}}$$

GlorotNormal():

$$\sqrt{2/(input + output)}.$$

HeNormal():

$$\sqrt{2/input}.$$

```
def compute_permutation_importance(model,X, y, metric=accuracy_s
    baseline_accuracy = metric(y,(model.predict(X) > 0.5).astype
    feature_importance = []

    for i in range(X.shape[1]):
        for j in range(X.shape[2]):
            X_permuted = X.copy()
            np.random.shuffle(X_permuted[:, i, j])

            accuracy = metric(y,(model.predict(X_permuted) > 0.5
            importance = baseline_accuracy - accuracy
            feature_importance.append(importance)

    return np.array(feature_importance)

importance = compute_permutation_importance(model,X_test_rnn,y_t
```

Function Breakdown:

1. **Inputs**:
    - `model`: The trained TensorFlow/Keras model.
    - `X`: The input data used for prediction and feature importance calculation.
    - `y`: The actual labels corresponding to the input data.
    - `metric`: A function (defaulting to `accuracy_score`) used to evaluate model performance.
2. **Outputs**:
    - `np.array(importance)`: An array of feature importances, where each value represents the impact of a feature on the model's performance when permuted.
3. **Steps**:
    - **Baseline Accuracy**: Calculate the initial accuracy of the model on the unaltered test set using the provided metric (defaulting to `accuracy_score`).

- **Feature Importance Calculation**:
  - Iterate over each feature of the input data (`X.shape[1]` features in `X`):
    - For each feature at a specific timestep (`X.shape[2]` timesteps):
      - Create a copy of `X`.
      - Shuffle (permute) the feature values.
      - Pass the permuted input through the model to get predictions.
      - Calculate the accuracy of the permuted predictions using the provided metric.
      - Compute the importance of the feature by comparing the baseline accuracy with the accuracy of the permuted predictions (`importance = baseline_accuracy - permuted_accuracy`).
  - Store the computed importance for each feature in the `feature_importance` list.

4. **Return**:
   - Convert the `feature_importance` list to a NumPy array and return it.

```
for i, layer in enumerate(model.layers):
    print(f"Layer {i} - {layer.name}")

    if len(layer.get_weights()) > 0:
        weights = layer.get_weights()
        if "lstm" in layer.name:
            kernel, recurrent_kernel, biases = weights
            print(f"Kernel shape (input-to-hidden):{kernel.shape
            print(f"Recurrent kernel shape (hidden-to-hidden):{r
            print(f"Biases shape: {biases.shape}")

        else:
            weights, biases = weights
            print(f"Weights shape:{weights.shape}")
            print(f"Biases shape: {biases.shape}")
```

```
    else:
        print("This layer has no trainable weights.")
```

Explanation:

1. **Loop Through Layers**:
   - `for i, layer in enumerate(model.layers)`: This loop iterates over the layers of the model using `enumerate` to get both the index (`i`) and the layer object (`layer`).
2. **Print Layer Name**:
   - `print(f"Layer {i} - {layer.name}")`: Print the index and the name of the current layer.
3. **Check if Layer has Weights**:
   - `if len(layer.get_weights()) > 0`: Check if the layer has any weights (i.e., if it is a trainable layer).
4. **Different Handling for Layers**:
   - **For `LSTM` layers**:
     - `if "lstm" in layer.name`: If the layer is an LSTM, it retrieves and prints:
       - `weights = layer.get_weights()`: Get the weights of the LSTM layer.
       - `kernel, recurrent_kernel, biases = weights`: Unpack the weights into the kernel (input-to-hidden connections), recurrent kernel (hidden-to-hidden connections), and biases.
       - `print(f"Kernel shape (input-to-hidden):{kernel.shape}")`, `print(f"Recurrent kernel shape (hidden-to-hidden):{recurrent_kernel.shape}")`, `print(f"Biases shape: {biases.shape}")`: Print the shapes of these weights.
   - **For other layers**:
     - `else`: If the layer is not LSTM:
       - `weights, biases = layer.get_weights()`: Get the weights and biases.
       - `print(f"Weights shape:{weights.shape}")`: Print the shape of the weights.

- `print(f"Biases shape: {biases.shape}")`: Print the shape of the biases.

5. **No Trainable Weights**:
   - `else`: If the layer does not have any weights (e.g., a `Dropout` layer), print "This layer has no trainable weights."

```
layer_name = "dense"

for layer in model.layers:
    if layer.name.startswith(layer_name):
        weights, biases = layer.get_weights()

        plt.figure(figsize=(10,6))
        plt.title(f"Weights of Layer: {layer.name}")
        plt.imshow(weights, aspect='auto',cmap="viridis")
        plt.colorbar()
        plt.xlabel("Neurons")
        plt.ylabel("Input Features")
        plt.show()


        plt.figure(figsize=(10,2))
        plt.title(f"Biases of Layer:{layer.name}")
        plt.bar(np.arange(len(biases)),biases)
        plt.xlabel("Neurons")
        plt.ylabel("Bias Value")
        plt.show()
```

Explanation:

1. **Specify the Layer Name**:
   - `layer_name = "dense"`: This specifies the name of the layer for which you want to visualize weights and biases.
2. **Loop through the model layers**:
   - `for layer in model.layers`: Iterate over each layer in the model.

3. **Check if the layer name starts with `layer_name`**:
    - `if layer.name.startswith(layer_name)`: Check if the current layer's name matches the specified `layer_name`.
4. **Get Weights and Biases**:
    - `weights, biases = layer.get_weights()`: Retrieve the weights and biases of the current layer.
5. **Visualize Weights**:
    - **Weights Plot**:
        - `plt.figure(figsize=(10,6))`: Create a new figure with specified dimensions.
        - `plt.title(f"Weights of Layer: {layer.name}")`: Set the title for the plot.
        - `plt.imshow(weights, aspect='auto', cmap="viridis")`: Display the weights as an image with a colormap.
        - `plt.colorbar()`: Add a color bar to the plot to show the range of weights.
        - `plt.xlabel("Neurons")`, `plt.ylabel("Input Features")`: Set labels for the x and y axes.
        - `plt.show()`: Display the plot.
6. **Visualize Biases**:
    - **Biases Plot**:
        - `plt.figure(figsize=(10,2))`: Create a new figure with specified dimensions.
        - `plt.title(f"Biases of Layer:{layer.name}")`: Set the title for the plot.
        - `plt.bar(np.arange(len(biases)), biases)`: Display the biases as a bar plot.
        - `plt.xlabel("Neurons")`, `plt.ylabel("Bias Value")`: Set labels for the x and y axes.
        - `plt.show()`: Display the plot.

```
lstm_layer = model.layers[0]
kernel, recurrent_kernel, biases = lstm_layer.get_weights()

print("Weights shapes:")
print(f"Kernel (input-to-hidden): {kernel.shape}")
print(f"Recurrent kernel (hidden-to-hidden): {recurrent_kernel.s
print(f"Biases: {biases.shape}")

plt.figure(figsize=(10, 6))
plt.title("LSTM Kernel Weights (Input-to-Hidden)")
plt.imshow(kernel, aspect='auto', cmap='viridis')
plt.colorbar()
plt.xlabel("LSTM Units")
plt.ylabel("Input Features")
plt.show()

plt.figure(figsize=(10, 6))
plt.title("LSTM Recurrent Kernel Weights (Hidden-to-Hidden)")
plt.imshow(recurrent_kernel, aspect='auto', cmap='viridis')
plt.colorbar()
plt.xlabel("LSTM Units")
plt.ylabel("LSTM Units")
plt.show()

# Visualize the biases
plt.figure(figsize=(10, 2))
plt.title("LSTM Biases")
plt.bar(np.arange(len(biases)), biases)
plt.xlabel("Gates (Input, Forget, Output, Candidate)")
plt.ylabel("Bias Value")
plt.show()
```

- `lstm_layer` is assigned as the first layer of your model (`model.layers[0]`).

- `get_weights()` retrieves the weights for the `kernel` (input-to-hidden), `recurrent_kernel` (hidden-to-hidden), and `biases` from the LSTM layer.
- Outputs the shapes of `kernel`, `recurrent_kernel`, and `biases`
- Creates a plot with a heatmap to visualize the `kernel` weights, representing the connections from input features to the LSTM units.
- The `aspect='auto'` ensures the correct aspect ratio for visualization.
- `plt.colorbar()` adds a color bar to interpret the intensity of weights.
- `plt.xlabel` and `plt.ylabel` label the axes for clarity.
- Similar to the kernel weights, but now visualizing the recurrent connections from LSTM unit to LSTM unit.
- The matrix is square because recurrent connections are between the same units.
- Creates a bar plot to visualize the biases for the LSTM layer.
- Biases are typically used for gates (input, forget, output, and candidate).
- `np.arange(len(biases))` creates an index for each bias gate.
- `plt.xlabel` and `plt.ylabel` provide clear labels for the axes.