Code Documentation

---

1. Libraries Used

1.1 Data Manipulation

- `pandas`:
    - `pd.read_csv()`: Reads the dataset from a CSV file.
    - `pd.Series()`: Handles class distributions and counts.

1.2 Data Preprocessing

- `sklearn.preprocessing`:
    - `MinMaxScaler`: Scales feature values to a specific range (default `[0, 1]`).
    - `LabelEncoder`: Converts categorical labels into integers.

1.3 Machine Learning and Metrics

- `imblearn`:
    - `SMOTE`: Handles imbalanced datasets by oversampling the minority class.
    - `RandomUnderSampler`: An optional method to under-sample the majority class.

1.4 Neural Networks

- `tensorflow.keras`:
    - Layers: `Dense`, `Dropout`, `Input`, `LSTM`.
    - Utilities: `to_categorical` (for one-hot encoding).
    - Optimizers: Adam optimizer.
    - Initializers: `GlorotUniform`, `HeNormal`, etc.

1.5 Visualization

- `matplotlib.pyplot`: For plotting accuracy, loss curves, and SHAP visualizations.

- `seaborn`: For heatmap visualizations.

1.6 Model Explanations

- `shap`: Explains feature importance using SHAP values.

---

2. Data Loading and Inspection

```
dataset = pd.read_csv('/kaggle/input/combined-network-faults-data/combined_network_faults_data.csv')
print(dataset.head())
print(dataset.isnull().sum())
```

- **Purpose**:
  - Loads the dataset.
  - Displays the first few rows.
  - Checks for missing values in each column.

---

3. Handling Missing Data

```
dataset = dataset.dropna()
print(dataset.isnull().sum())
```

- **Purpose**: Removes rows containing missing values and rechecks for nulls.

---

4. Checking Class Imbalance

```
print(dataset['0'].value_counts())
```

- **Purpose**: Displays the distribution of classes in the target column (`'0'`).

---

5. Handling Class Imbalance Using SMOTE

```
features = dataset.iloc[:, :-1]  # All columns except the las
t (target).
target = dataset['0']           # Target column.

smote = SMOTE(sampling_strategy='auto')
X_resampled, y_resampled = smote.fit_resample(features, targe
t)

print(pd.Series(y_resampled).value_counts())
```

- **Purpose**:
  - Uses Synthetic Minority Oversampling Technique (SMOTE) to balance class distribution.
  - **Parameters**:
    - `sampling_strategy`: Specifies oversampling ratio or strategy.

---

6. Feature Scaling

```
scaler = MinMaxScaler()
scaled_features = scaler.fit_transform(X_resampled)
```

- **Purpose**: Scales features to a range of `[0, 1]`.

---

7. Label Encoding and One-Hot Encoding

```
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(y_resampled)
one_hot_labels = to_categorical(encoded_labels, num_classes=
5)
```

- **Purpose**:
  - Encodes target labels into integers.

- Converts integer labels into one-hot encoded vectors.

---

8. Reshaping Data for LSTM

```
X_resampled_reshaped = scaled_features.reshape((scaled_featur
es.shape[0], 1, scaled_features.shape[1]))
```

- **Purpose**: Reshapes the input data to fit the LSTM input format (`samples, time_steps, features`).

---

9. Splitting Data

```
X_train, X_test, y_train, y_test = train_test_split(X_resampl
ed_reshaped, one_hot_labels, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_
train, test_size=0.2, random_state=42)
```

- **Purpose**:
    - Splits the data into training (64%), validation (16%), and testing (20%) subsets.

---

10. Model Building and Training

```
model = Sequential([
    LSTM(128, activation='relu', input_shape=(X_train.shape
[1], X_train.shape[2])),
    Dropout(0.4),
    Dense(64, activation='relu'),
    Dense(5, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentrop
y', metrics=['accuracy'])
```

```
history = model.fit(X_train, y_train, validation_data=(X_val,
y_val), epochs=10, batch_size=32)
```

- **Purpose**:
  - Builds an LSTM model.
  - **Layers**:
    - `LSTM`: Learns temporal dependencies.
    - `Dropout`: Reduces overfitting by dropping nodes.
    - `Dense`: Fully connected layers for classification.
  - **Parameters**:
    - `epochs`: Number of iterations over the entire dataset.
    - `batch_size`: Number of samples per gradient update.

---

11. Hyperparameter Tuning

```
param_grid = {'lstm_units': [64, 128], 'dropout_rate': [0.3,
0.4], 'learning_rate': [0.001, 0.01]}
```

- **Purpose**:
  - Tests different combinations of LSTM units, dropout rates, and learning rates to find the optimal configuration.

---

12. Visualizing Performance

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation A
ccuracy')
```

- **Purpose**: Plots accuracy and loss curves for each training configuration.

---

13. Feature Importance (SHAP)

```
explainer = shap.KernelExplainer(predict_fn, X_val_flat[:100])
shap_values = explainer.shap_values(X_val_flat[:10])
shap.summary_plot(shap_values, X_val_flat[:10], feature_names=feature_names)
```

- **Purpose**: Explains feature contributions using SHAP values.

---

14. Weight Initialization Techniques

```
initializers = {
    'glorot_uniform': GlorotUniform(),
    'he_normal': HeNormal(),
    'orthogonal': Orthogonal(),
    'random_normal': RandomNormal(mean=0.0, stddev=0.05),
    'lecun_uniform': LecunUniform()
}
```

- **Purpose**: Tests different weight initializations for better model convergence.

Explanation of Weight Initialization Techniques

Weight initialization is crucial to deep learning, as it determines how well a model converges during training. These techniques set the initial values of a model's weights before training begins, ensuring gradients flow properly through the network and preventing issues like vanishing or exploding gradients.

Here's a detailed explanation of each initializer:

---

1. `glorot_uniform` (Xavier Initialization)

- **Definition**: The weights are initialized from a uniform distribution with bounds where:

$$[-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}],$$

- ninn_{\text{in}}: Number of input units.
- noutn_{\text{out}}: Number of output units.

- **Purpose**: Balances the variance of weights across layers, helping the gradient propagate effectively during backpropagation.
- **Use Case**: Suitable for activations like `tanh` or `sigmoid`.
- **Advantages**: Works well for shallow and deep networks, reducing the risk of vanishing/exploding gradients.

---

2. `he_normal` (He Initialization)

- **Definition**: The weights are initialized from a normal distribution with a mean of 0 and a standard deviation of 2nin\sqrt{\frac{2}{n_{\text{in}}}}, where ninn_{\text{in}} is the number of input units.
- **Purpose**: Addresses the limitations of Xavier initialization, especially for rectified linear unit (ReLU) activations, by considering their asymmetric behavior (outputs non-zero only for positive inputs).
- **Use Case**: Ideal for layers with ReLU or its variants (e.g., LeakyReLU).
- **Advantages**: Ensures that the variance of weights remains appropriate for ReLU activations.

---

3. `orthogonal`

- **Definition**: The weights are initialized to form an orthogonal matrix, where the dot product of any two rows/columns is zero.
- **Purpose**: Maintains the independence of weights, ensuring that information flows through the network without interference.
- **Use Case**: Commonly used for RNNs, LSTMs, and GRUs, where preserving orthogonality is important to avoid exploding or vanishing gradients over long sequences.
- **Advantages**: Helps stabilize gradients in recurrent architectures.

---

4. `random_normal`

- **Definition**: The weights are initialized from a normal distribution with a specified mean (default 0.0) and standard deviation (default 0.05).
- **Purpose**: Provides control over the randomness of weight initialization.
- **Use Case**: Useful for custom experiments where specific ranges or distributions are required.
- **Advantages**:
  - Flexibility in defining the mean and variance.
  - Simpler but less sophisticated compared to Xavier or He initialization.

---

5. `lecun_uniform`

- **Definition**: The weights are initialized from a uniform distribution with bounds [−3nin,3nin][- \sqrt{\frac{3}{n_{\text{in}}}}, \sqrt{\frac{3}{n_{\text{in}}}}], where ninn_{\text{in}} is the number of input units.
- **Purpose**: Specifically designed for use with `selu` (Scaled Exponential Linear Unit) activations.
- **Use Case**: Used in networks where self-normalizing properties are desired, particularly for deep architectures.
- **Advantages**: Works well with `selu` activations to achieve self-normalizing behavior in deep networks.

Summary Table of Initializers

Initializer	Distribution	Formula/Properties	Best For
`glorot_uniform`	Uniform	$[-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}]$	`tanh`, `sigmoid` activations
`he_normal`	Normal	$\text{Mean} = 0, \text{StdDev} = \sqrt{\frac{2}{n_{\text{in}}}}$	`ReLU` and variants
`orthogonal`	Matrix	Orthogonal weight matrix	RNNs, LSTMs, GRUs
`random_normal`	Normal	Mean and StdDev are configurable	Custom experiments
`lecun_uniform`	Uniform	$[-\sqrt{\frac{3}{n_{\text{in}}}}, \sqrt{\frac{3}{n_{\text{in}}}}]$	`selu` activations

---

Choosing the right initializer depends on the network architecture and activation functions used. For most general cases:

- Use `he_normal` with ReLU-like activations.
- Use `glorot_uniform` with sigmoid/tanh activations.
- Use `orthogonal` for recurrent networks.
- Experiment with `random_normal` and `lecun_uniform` in specific scenarios.

---

15. Prediction on Unseen Data

```
unseen_data_scaled = scaler.transform(unseen_data)
unseen_data_reshaped = unseen_data_scaled.reshape((unseen_data_scaled.shape[0], 1, unseen_data_scaled.shape[1]))
prediction = model.predict(unseen_data_reshaped)
predicted_class = np.argmax(prediction)
```

- **Purpose**: Predicts the class of new input data.

---

16. Key Parameters

Data Preprocessing

Parameter	Description
`sampling_strategy`	Ratio for SMOTE oversampling.

Model Training

Parameter	Description
`lstm_units`	Number of units in the LSTM layer.
`dropout_rate`	Fraction of units to drop in dropout layer.
`learning_rate`	Optimizer step size.
`epochs`	Number of complete passes over the dataset.
`batch_size`	Number of samples per gradient update.
