```python
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model, Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split

# Define the path to the dataset. You can change this to your local
file path if needed.
path =
'http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv'

# Read the ECG dataset into a Pandas DataFrame
data = pd.read_csv(path, header=None)

data.head()
```

{"type":"dataframe","variable_name":"data"}

```python
# Get information about the dataset, such as column data types and
non-null counts
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4998 entries, 0 to 4997
Columns: 141 entries, 0 to 140
dtypes: float64(141)
memory usage: 5.4 MB
```

```python
# Splitting the dataset into features and target
features = data.drop(140, axis=1)  # Features are all columns except
the last (column 140)
target = data[140]  # Target is the last column (column 140)

# Split the data into training and testing sets (80% training, 20%
testing)
x_train, x_test, y_train, y_test = train_test_split(
    features, target, test_size=0.2
)

# Get the indices of the training data points labeled as "1"
(anomalies)
train_index = y_train[y_train == 1].index

# Select the training data points that are anomalies
train_data = x_train.loc[train_index]
```

```python
# Initialize the Min-Max Scaler to scale the data between 0 and 1
min_max_scaler = MinMaxScaler(feature_range=(0, 1))

# Scale the training data
x_train_scaled = min_max_scaler.fit_transform(train_data.copy())

# Scale the testing data using the same scaler
x_test_scaled = min_max_scaler.transform(x_test.copy())

# Creating an Autoencoder model by extending the Model class from
Keras
class AutoEncoder(Model):
    def __init__(self, output_units, ldim=8):
        super().__init__()
        # Define the encoder part of the Autoencoder
        self.encoder = Sequential([
            Dense(64, activation='relu'),
            Dropout(0.1),
            Dense(32, activation='relu'),
            Dropout(0.1),
            Dense(16, activation='relu'),
            Dropout(0.1),
            Dense(ldim, activation='relu')
        ])
        # Define the decoder part of the Autoencoder
        self.decoder = Sequential([
            Dense(16, activation='relu'),
            Dropout(0.1),
            Dense(32, activation='relu'),
            Dropout(0.1),
            Dense(64, activation='relu'),
            Dropout(0.1),
            Dense(output_units, activation='sigmoid')
        ])

    def call(self, inputs):
        # Forward pass through the Autoencoder
        encoded = self.encoder(inputs)
        decoded = self.decoder(encoded)
        return decoded

# Create an instance of the AutoEncoder model with the appropriate
output units
model = AutoEncoder(output_units=x_train_scaled.shape[1])

# Compile the model with Mean Squared Logarithmic Error (MSLE) loss
and Mean Squared Error (MSE) metric
model.compile(loss='msle', metrics=['mse'], optimizer='adam')

# Train the model using the scaled training data
```

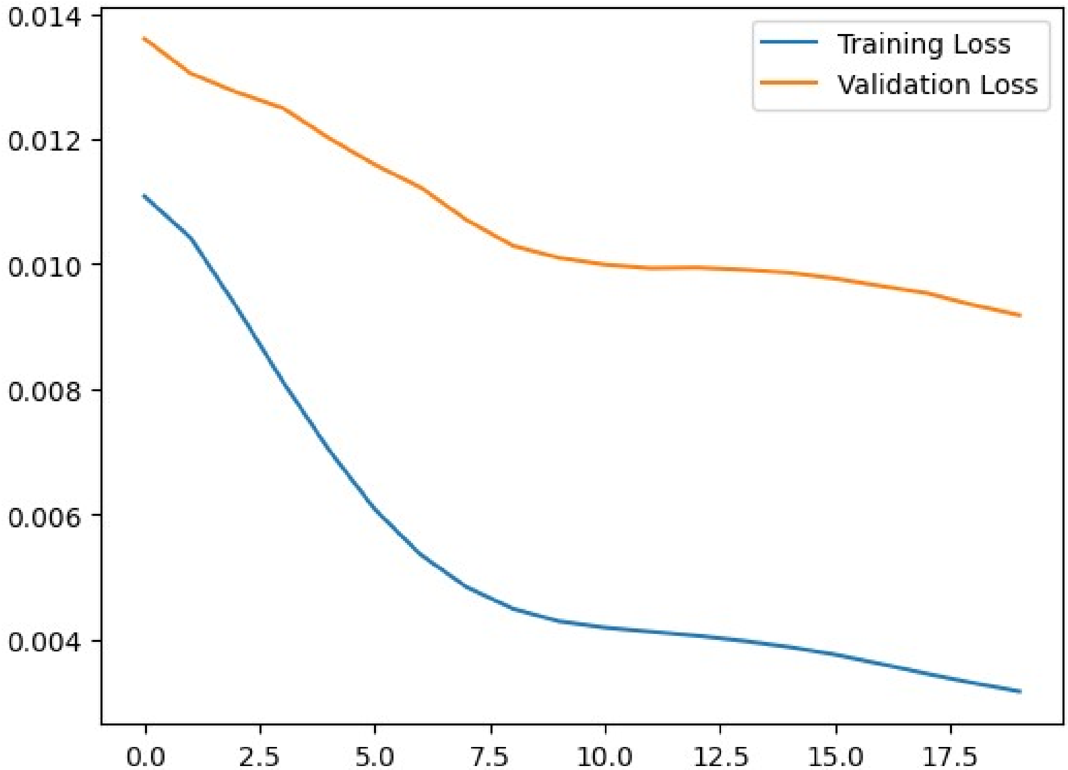
```
history = model.fit(
    x_train_scaled,  # Input data for training
    x_train_scaled,  # Target data for training (autoencoder
reconstructs the input)
    epochs=20,        # Number of training epochs
    batch_size=512,   # Batch size
    validation_data=(x_test_scaled, x_test_scaled),  # Validation data
    shuffle=True     # Shuffle the data during training
)
```

```
Epoch 1/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 3s 71ms/step - loss: 0.0111 - mse: 0.0251 -
val_loss: 0.0136 - val_mse: 0.0315
Epoch 2/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - loss: 0.0105 - mse: 0.0236 -
val_loss: 0.0130 - val_mse: 0.0302
Epoch 3/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - loss: 0.0094 - mse: 0.0211 -
val_loss: 0.0127 - val_mse: 0.0294
Epoch 4/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - loss: 0.0083 - mse: 0.0187 -
val_loss: 0.0125 - val_mse: 0.0288
Epoch 5/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - loss: 0.0071 - mse: 0.0160 -
val_loss: 0.0120 - val_mse: 0.0278
Epoch 6/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0062 - mse: 0.0140 -
val_loss: 0.0116 - val_mse: 0.0268
Epoch 7/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - loss: 0.0055 - mse: 0.0123 -
val_loss: 0.0112 - val_mse: 0.0260
Epoch 8/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0050 - mse: 0.0113 -
val_loss: 0.0107 - val_mse: 0.0248
Epoch 9/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0045 - mse: 0.0102 -
val_loss: 0.0103 - val_mse: 0.0239
Epoch 10/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 0.0043 - mse: 0.0097 -
val_loss: 0.0101 - val_mse: 0.0235
Epoch 11/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - loss: 0.0042 - mse: 0.0095 -
val_loss: 0.0100 - val_mse: 0.0233
Epoch 12/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 0.0041 - mse: 0.0092 -
val_loss: 0.0099 - val_mse: 0.0232
Epoch 13/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - loss: 0.0041 - mse: 0.0092 -
val_loss: 0.0099 - val_mse: 0.0232
Epoch 14/20
```

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - loss: 0.0039 - mse: 0.0089 -
val_loss: 0.0099 - val_mse: 0.0231
Epoch 15/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - loss: 0.0039 - mse: 0.0087 -
val_loss: 0.0099 - val_mse: 0.0230
Epoch 16/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.0039 - mse: 0.0087 -
val_loss: 0.0098 - val_mse: 0.0227
Epoch 17/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 0.0036 - mse: 0.0081 -
val_loss: 0.0097 - val_mse: 0.0224
Epoch 18/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 0.0035 - mse: 0.0080 -
val_loss: 0.0095 - val_mse: 0.0221
Epoch 19/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - loss: 0.0033 - mse: 0.0074 -
val_loss: 0.0093 - val_mse: 0.0216
Epoch 20/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 0.0032 - mse: 0.0073 -
val_loss: 0.0092 - val_mse: 0.0212
```

```python
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fcffc6c78b0>
```




```
# Function to find the threshold for anomalies based on the training
data
def find_threshold(model, x_train_scaled):
    # Reconstruct the data using the model
    recons = model.predict(x_train_scaled)

    # Calculate the mean squared log error between reconstructed data
and the original data
    recons_error = tf.keras.metrics.msle(recons, x_train_scaled)

    # Set the threshold as the mean error plus one standard deviation
    threshold = np.mean(recons_error.numpy()) +
np.std(recons_error.numpy())

    return threshold

# Function to make predictions for anomalies based on the threshold
def get_predictions(model, x_test_scaled, threshold):
    # Reconstruct the data using the model
    predictions = model.predict(x_test_scaled)

    # Calculate the mean squared log error between reconstructed data
and the original data
    errors = tf.keras.losses.msle(predictions, x_test_scaled)

    # Create a mask for anomalies based on the threshold
```

```
    anomaly_mask = pd.Series(errors) > threshold

    # Map True (anomalies) to 0 and False (normal data) to 1
    preds = anomaly_mask.map(lambda x: 0.0 if x == True else 1.0)

    return preds

# Find the threshold for anomalies
threshold = find_threshold(model, x_train_scaled)
print(f"Threshold: {threshold}")
```

```
74/74 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Threshold: 0.007391941009531054
```

```
# Get predictions for anomalies based on the model and threshold
predictions = get_predictions(model, x_test_scaled, threshold)

# Calculate the accuracy score by comparing the predicted anomalies to
the true labels
accuracy = accuracy_score(predictions, y_test)

# Print the accuracy score
print(f"Accuracy Score: {accuracy}")
```

```
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Accuracy Score: 0.958
```