Impact of Contextual and Weather Data on Occupancy Prediction Using Deep Learning and Explainable AI

Md Istiak ahammed

**1. Project Overview**

**Objective**

The project investigates the influence of contextual and weather features on indoor occupancy prediction using deep learning models. By incorporating multimodal data streams (indoor sensor data, contextual features, and weather attributes), we aim to optimize energy-efficient HVAC operations in smart buildings.

**Key Contributions**

1. Developed and analyzed three datasets:
   * **D1**: Indoor sensor data (CO2, temperature, humidity, light).
   * **D2**: Indoor + Contextual data (hour\_of\_day, is\_weekend).
   * **D3**: Indoor + Contextual + Weather data (wind\_speed, humidity, cloud\_cover, etc.).
2. Evaluated five deep learning models:
   * LSTM, CNN-LSTM, CNN-BiLSTM, CNN-BiLSTM-Attention, and Transformer.
3. Used SHapley Additive exPlanations (SHAP) analysis to assess feature impact.
4. Performance evaluated using accuracy, precision, recall, and F1-score.

**2. Datasets**

**2.1 Dataset Descriptions**

Three distinct datasets were created to evaluate the impact of integrating environmental, contextual, and weather features on indoor occupancy prediction:

**D1: Indoor Environmental Dataset**Includes indoor environmental sensor data such as temperature, humidity, CO₂ concentration, illuminance, and humidity ratio, collected from an office environment in Mons, Belgium (February 2–18, 2015).

**D2: Indoor Environmental and Contextual Dataset**Combines D1 with contextual features such as the hour of the day and the day of the week to account for periodic human activity patterns.

**D3: Indoor Environmental, Contextual, and Weather Dataset**

Extends D2 by integrating weather data (e.g., outdoor temperature, wind speed, humidity, cloud cover) obtained from the Visual Crossing Weather Data platform for the same time period and location.

**3. Methodology**

**3.1 Models**

1. **LSTM**: Captures temporal dependencies.
2. **CNN-LSTM**: Combines convolutional feature extraction with sequential learning.
3. **CNN-BiLSTM**: Enhances temporal learning with bidirectional LSTMs.
4. **CNN-BiLSTM-Attention**: Adds attention mechanisms to prioritize key features.
5. **Transformer**: Employs self-attention for efficient modeling of long-range dependencies.

**3.2 Code for Models**

**LSTM Model**

|  |
| --- |
| **Step 1: Import necessary libraries**  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.preprocessing import MinMaxScaler  from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  import torch  import torch.nn as nn  import torch.optim as optim  from torch.utils.data import DataLoader, TensorDataset  **Step 2: Load dataset and Preprocessing (only D3 dataset)**  data = pd.read\_csv('../data/final\_indoor\_weather\_context.csv', parse\_dates=['date'], index\_col='date')  scaler = MinMaxScaler()  data\_scaled = scaler.fit\_transform(data.drop(columns=['occupancy']))  *# Preparing data for LSTM*  sequence\_length = 60  def create\_dataset(data, target, sequence\_length):      X, y = [], []      for i in range(len(data) - sequence\_length):          X.append(data[i:i + sequence\_length])          y.append(target[i + sequence\_length])      return np.array(X), np.array(y)  X, y = create\_dataset(data\_scaled, data['occupancy'].values, sequence\_length)  *# Converting to PyTorch tensors*  X = torch.tensor(X, dtype=torch.float32)  y = torch.tensor(y, dtype=torch.float32)  *# Splitting into training, validation, and testing*  train\_size = int(0.7 \* len(X))  val\_size = int(0.15 \* len(X))  test\_size = len(X) - train\_size - val\_size  X\_train, X\_val, X\_test = X[:train\_size], X[train\_size:train\_size + val\_size], X[train\_size + val\_size:]  y\_train, y\_val, y\_test = y[:train\_size], y[train\_size:train\_size + val\_size], y[train\_size + val\_size:]  *# Creating DataLoader*  batch\_size = 32  train\_dataset = TensorDataset(X\_train, y\_train)  val\_dataset = TensorDataset(X\_val, y\_val)  test\_dataset = TensorDataset(X\_test, y\_test)  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)  test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)  **Step 3: Model Development and Initialization**  *# Defining the LSTM model*  class LSTMModel(nn.Module):      def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):          super(LSTMModel, self).\_\_init\_\_()          self.lstm = nn.LSTM(input\_size, hidden\_size, batch\_first=True)          self.dropout = nn.Dropout(0.3)  # Increased dropout rate to reduce overfitting          self.fc = nn.Linear(hidden\_size, output\_size)          self.sigmoid = nn.Sigmoid()      def forward(self, x):          \_, (hn, \_) = self.lstm(x)          x = self.dropout(hn[-1])          x = self.fc(x)          x = self.sigmoid(x)          return x  *# Model parameters*  input\_size = X\_train.shape[2]  hidden\_size = 30  # Reduced hidden size to decrease model complexity  output\_size = 1  *# Model, loss, optimizer*  model = LSTMModel(input\_size, hidden\_size, output\_size)  criterion = nn.BCELoss()  optimizer = optim.Adam(model.parameters(), lr=0.001)  **Step 4: Model Training**  # Training the model with early stopping  epochs = 50  train\_losses = []  val\_losses = []  best\_val\_loss = float('inf')  patience = 5  # Number of epochs to wait before early stopping  trigger\_times = 0  for epoch in range(epochs):      model.train()      running\_loss = 0.0      for X\_batch, y\_batch in train\_loader:          optimizer.zero\_grad()          outputs = model(X\_batch)          loss = criterion(outputs.squeeze(), y\_batch)          loss.backward()          optimizer.step()          running\_loss += loss.item() \* X\_batch.size(0)      epoch\_loss = running\_loss / len(train\_loader.dataset)      train\_losses.append(epoch\_loss)      print(f'Epoch {epoch+1}/{epochs}, Loss: {epoch\_loss:.4f}')  *# Validation*      model.eval()      val\_loss = 0.0      with torch.no\_grad():          for X\_batch, y\_batch in val\_loader:              outputs = model(X\_batch)              loss = criterion(outputs.squeeze(), y\_batch)              val\_loss += loss.item() \* X\_batch.size(0)      val\_loss /= len(val\_loader.dataset)      val\_losses.append(val\_loss)      print(f'Validation Loss: {val\_loss:.4f}')  *# Early Stopping Logic*      if val\_loss < best\_val\_loss:          best\_val\_loss = val\_loss          trigger\_times = 0          # Save the best model          torch.save(model.state\_dict(), 'Saved\_models\_indoor\_context\_weather/indoor\_weather\_context\_lstm\_best\_model\_shap.pth')      else:          trigger\_times += 1          if trigger\_times >= patience:              print("Early stopping triggered")              break  **Step 5: Evaluation**  *# Evaluating the model*  model.eval()  y\_pred = []  y\_true = []  with torch.no\_grad():      for X\_batch, y\_batch in test\_loader:          outputs = model(X\_batch)          y\_pred.extend((outputs.squeeze() > 0.5).int().tolist())          y\_true.extend(y\_batch.int().tolist())  *# Accuracy and classification report*  y\_pred = np.array(y\_pred)  y\_true = np.array(y\_true)  accuracy = accuracy\_score(y\_true, y\_pred)  conf\_matrix = confusion\_matrix(y\_true, y\_pred)  class\_report = classification\_report(y\_true, y\_pred)  print(f"Accuracy: {accuracy:.2f}")  print("Confusion Matrix:")  print(conf\_matrix)  print("Classification Report:")  print(class\_report)  *# Plotting training and validation loss over epochs*  plt.figure(figsize=(10, 6))  plt.plot(range(1, len(train\_losses) + 1), train\_losses, label='Training Loss')  plt.plot(range(1, len(val\_losses) + 1), val\_losses, label='Validation Loss')  plt.xlabel('Epochs')  plt.ylabel('Loss')  plt.title('Training and Validation Loss Over Epochs')  plt.legend()  plt.show()  *# Plotting confusion matrix using seaborn*  plt.figure(figsize=(8, 6))  sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)  plt.xlabel('Predicted Label')  plt.ylabel('True Label')  plt.title('Confusion Matrix')  plt.show()  *# Saving the model*  torch.save(model.state\_dict(), 'Saved\_models\_indoor\_context\_weather/indoor\_weather\_context\_lstm\_regularized\_model.pth') |

**CNN-LSTM Model**

|  |
| --- |
| **Step 1: Import necessary libraries**  *# Import necessary libraries*  import torch  import torch.nn as nn  import torch.optim as optim  from torch.utils.data import DataLoader, Dataset  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import MinMaxScaler  **Step 2: Load dataset and Preprocessing (only D3 dataset)**  *# Load dataset*  data = pd.read\_csv('../data/final\_indoor\_weather\_context.csv', parse\_dates=['date'], index\_col='date')  *# Data preparation*  scaler = MinMaxScaler()  data\_scaled = scaler.fit\_transform(data.drop(columns=['occupancy']))  y = data['occupancy'].values  *# Reshape data for LSTM (samples, timesteps, features)*  X = []  for i in range(len(data\_scaled) - 24):  # Using a window size of 24 timesteps      X.append(data\_scaled[i:i+24])  X = np.array(X)  y = y[24:]  # Shift labels to match the shape of X  *# Splitting the data into train, validation, and test sets*  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp, y\_temp, test\_size=0.5, random\_state=42)  *# Defining a custom dataset class*  class TimeSeriesDataset(Dataset):      def \_\_init\_\_(self, X, y):          self.X = X          self.y = y      def \_\_len\_\_(self):          return len(self.X)      def \_\_getitem\_\_(self, idx):          return torch.tensor(self.X[idx], dtype=torch.float32), torch.tensor(self.y[idx], dtype=torch.float32)  **Step 3: Model Development and Initialization**  *# CNN-LSTM Model Definition*  class CNNLSTM(nn.Module):      def \_\_init\_\_(self, input\_dim, cnn\_channels, lstm\_hidden\_units, output\_dim):          super(CNNLSTM, self).\_\_init\_\_()          self.cnn = nn.Sequential(              nn.Conv1d(in\_channels=input\_dim, out\_channels=cnn\_channels, kernel\_size=3, stride=1, padding=1),              nn.ReLU(),              nn.MaxPool1d(kernel\_size=2, stride=2)          )          self.lstm = nn.LSTM(input\_size=cnn\_channels, hidden\_size=lstm\_hidden\_units, batch\_first=True)          self.dropout = nn.Dropout(0.3)          self.fc = nn.Linear(lstm\_hidden\_units, output\_dim)          self.sigmoid = nn.Sigmoid()      def forward(self, x):  *# Input shape: (batch\_size, seq\_len, input\_dim)*          x = x.permute(0, 2, 1)  # Change shape to (batch\_size, input\_dim, seq\_len) for CNN          x = self.cnn(x)  # Apply CNN          x = x.permute(0, 2, 1)  # Change shape back to (batch\_size, seq\_len, cnn\_channels) for LSTM          lstm\_out, \_ = self.lstm(x)  # Apply LSTM          lstm\_out = self.dropout(lstm\_out[:, -1, :])  # Take the output of the last time step and apply dropout          out = self.fc(lstm\_out)  # Fully connected layer          return self.sigmoid(out)  *# Hyperparameters*  input\_dim = 17  # Number of features (e.g., temperature, humidity, etc.)  cnn\_channels = 16  lstm\_hidden\_units = 32  output\_dim = 1  learning\_rate = 0.001  batch\_size = 64  epochs = 50  *# Create dataset and data loaders*  train\_dataset = TimeSeriesDataset(X\_train, y\_train)  val\_dataset = TimeSeriesDataset(X\_val, y\_val)  test\_dataset = TimeSeriesDataset(X\_test, y\_test)  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)  test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)  *# Initialize model, loss function, and optimizer*  model = CNNLSTM(input\_dim, cnn\_channels, lstm\_hidden\_units, output\_dim)  criterion = nn.BCELoss()  optimizer = optim.Adam(model.parameters(), lr=learning\_rate)  **Step 4: Model Training**  *# Training loop*  def train\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs):      model.train()      train\_losses = []      val\_losses = []      early\_stopping\_patience = 5      best\_val\_loss = float('inf')      patience\_counter = 0      for epoch in range(epochs):          running\_loss = 0.0          for X\_batch, y\_batch in train\_loader:              optimizer.zero\_grad()              outputs = model(X\_batch).squeeze()              loss = criterion(outputs, y\_batch)              loss.backward()              optimizer.step()              running\_loss += loss.item() \* X\_batch.size(0)          epoch\_loss = running\_loss / len(train\_loader.dataset)          train\_losses.append(epoch\_loss)  *# Validation phase*          model.eval()          val\_running\_loss = 0.0          with torch.no\_grad():              for X\_val\_batch, y\_val\_batch in val\_loader:                  val\_outputs = model(X\_val\_batch).squeeze()                  val\_loss = criterion(val\_outputs, y\_val\_batch)                  val\_running\_loss += val\_loss.item() \* X\_val\_batch.size(0)          val\_loss\_epoch = val\_running\_loss / len(val\_loader.dataset)          val\_losses.append(val\_loss\_epoch)          print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch\_loss:.4f}, Validation Loss: {val\_loss\_epoch:.4f}")  *# Early stopping*          if val\_loss\_epoch < best\_val\_loss:              best\_val\_loss = val\_loss\_epoch              patience\_counter = 0          else:              patience\_counter += 1          if patience\_counter >= early\_stopping\_patience:              print("Early stopping triggered")              break          model.train()      return train\_losses, val\_losses  train\_losses, val\_losses = train\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs)  *# Saving the model*  torch.save(model.state\_dict(), 'Saved\_models\_indoor\_context\_weather/cnn\_lstm\_indoor\_weather\_context\_model.pth')  **Step 5: Evaluation**  *# Testing the model*  def evaluate\_model(model, test\_loader):      model.eval()      y\_true = []      y\_pred = []      with torch.no\_grad():          for X\_batch, y\_batch in test\_loader:              outputs = model(X\_batch).squeeze()              predicted = (outputs >= 0.5).float()              y\_true.extend(y\_batch.int().numpy())              y\_pred.extend(predicted.int().numpy())      accuracy = accuracy\_score(y\_true, y\_pred)      conf\_matrix = confusion\_matrix(y\_true, y\_pred)      class\_report = classification\_report(y\_true, y\_pred)      return accuracy, conf\_matrix, class\_report  accuracy, conf\_matrix, class\_report = evaluate\_model(model, test\_loader)  print(f"Accuracy: {accuracy:.2f}")  print(f"Confusion Matrix:\n{conf\_matrix}")  print(f"Classification Report:\n{class\_report}")  *# Plotting confusion matrix*  import seaborn as sns  sns.heatmap(conf\_matrix, annot=True, cmap='Blues', fmt='g')  plt.xlabel('Predicted Label')  plt.ylabel('True Label')  plt.title('Confusion Matrix')  plt.show() |

**CNN-BiLSTM Model**

|  |
| --- |
| **Step 1: Import necessary libraries**  *# Import necessary libraries*  import torch  import torch.nn as nn  import torch.optim as optim  from torch.utils.data import DataLoader, Dataset  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import MinMaxScaler  import seaborn as sns  **Step 2: Load dataset and Preprocessing (only D3 dataset)**  *# Load dataset*  data = pd.read\_csv('../data/final\_indoor\_weather\_context.csv', parse\_dates=['date'], index\_col='date')  # Data preparation  scaler = MinMaxScaler()  data\_scaled = scaler.fit\_transform(data.drop(columns=['occupancy']))  y = data['occupancy'].values.astype(int)  *# Reshape data for model (samples, timesteps, features)*  input\_dim = data\_scaled.shape[1]  # Number of features in the dataset  X = []  for i in range(len(data\_scaled) - 24):  # Using a window size of 24 timesteps      X.append(data\_scaled[i:i+24])  X = np.array(X)  y = y[24:]  # Shift labels to match the shape of X  *# Splitting the data into train, validation, and test sets*  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp, y\_temp, test\_size=0.5, random\_state=42)  *# Defining a custom dataset class*  class TimeSeriesDataset(Dataset):      def \_\_init\_\_(self, X, y):          self.X = X          self.y = y      def \_\_len\_\_(self):          return len(self.X)      def \_\_getitem\_\_(self, idx):          return torch.tensor(self.X[idx], dtype=torch.float32), torch.tensor(self.y[idx], dtype=torch.float32)  **Step 3: Model Development and Initialization**  *# CNN-BiLSTM Model Definition*  class CNNBiLSTM(nn.Module):      def \_\_init\_\_(self, input\_dim, cnn\_channels, lstm\_hidden\_units, output\_dim):          super(CNNBiLSTM, self).\_\_init\_\_()          self.cnn = nn.Sequential(              nn.Conv1d(in\_channels=input\_dim, out\_channels=cnn\_channels, kernel\_size=3, stride=1, padding=1),              nn.ReLU(),              nn.MaxPool1d(kernel\_size=2, stride=2)          )          self.bilstm = nn.LSTM(input\_size=cnn\_channels, hidden\_size=lstm\_hidden\_units, num\_layers=1, batch\_first=True, bidirectional=True)          self.dropout = nn.Dropout(0.3)          self.fc = nn.Linear(lstm\_hidden\_units \* 2, output\_dim)          self.sigmoid = nn.Sigmoid()      def forward(self, x):  *# Input shape: (batch\_size, seq\_len, input\_dim)*          x = x.permute(0, 2, 1)  # Change shape to (batch\_size, input\_dim, seq\_len) for CNN          x = self.cnn(x)  # Apply CNN          x = x.permute(0, 2, 1)  # Change shape back to (batch\_size, seq\_len, cnn\_channels) for LSTM          lstm\_out, \_ = self.bilstm(x)  # Apply BiLSTM          context\_vector = torch.sum(lstm\_out, dim=1)  # Summing over the sequence dimension          context\_vector = self.dropout(context\_vector)  # Apply dropout          out = self.fc(context\_vector)  # Fully connected layer          return self.sigmoid(out)  *# Hyperparameters*  input\_dim = 17  # Number of features (e.g., temperature, humidity, etc.)  cnn\_channels = 16  lstm\_hidden\_units = 32  output\_dim = 1  learning\_rate = 0.001  batch\_size = 64  epochs = 50  *# Create dataset and data loaders*  train\_dataset = TimeSeriesDataset(X\_train, y\_train)  val\_dataset = TimeSeriesDataset(X\_val, y\_val)  test\_dataset = TimeSeriesDataset(X\_test, y\_test)  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)  test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)  *# Initialize model, loss function, and optimizer*  model = CNNBiLSTM(input\_dim, cnn\_channels, lstm\_hidden\_units, output\_dim)  criterion = nn.BCELoss()  optimizer = optim.Adam(model.parameters(), lr=learning\_rate)  **Step 4: Model Training**  *# Training loop*  def train\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs):      model.train()      train\_losses = []      val\_losses = []      early\_stopping\_patience = 5      best\_val\_loss = float('inf')      patience\_counter = 0      for epoch in range(epochs):          running\_loss = 0.0          for X\_batch, y\_batch in train\_loader:              optimizer.zero\_grad()              outputs = model(X\_batch).squeeze()              loss = criterion(outputs, y\_batch)              loss.backward()              optimizer.step()              running\_loss += loss.item() \* X\_batch.size(0)          epoch\_loss = running\_loss / len(train\_loader.dataset)          train\_losses.append(epoch\_loss)  *# Validation phase*          model.eval()          val\_running\_loss = 0.0          with torch.no\_grad():              for X\_val\_batch, y\_val\_batch in val\_loader:                  val\_outputs = model(X\_val\_batch).squeeze()                  val\_loss = criterion(val\_outputs, y\_val\_batch)                  val\_running\_loss += val\_loss.item() \* X\_val\_batch.size(0)          val\_loss\_epoch = val\_running\_loss / len(val\_loader.dataset)          val\_losses.append(val\_loss\_epoch)          print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch\_loss:.4f}, Validation Loss: {val\_loss\_epoch:.4f}")  *# Early stopping*          if val\_loss\_epoch < best\_val\_loss:              best\_val\_loss = val\_loss\_epoch              patience\_counter = 0          else:              patience\_counter += 1          if patience\_counter >= early\_stopping\_patience:              print("Early stopping triggered")              break          model.train()      return train\_losses, val\_losses  train\_losses, val\_losses = train\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs)  *# Saving the model*  torch.save(model.state\_dict(), 'Saved\_models/cnn\_bilstm\_indoor\_context\_weather.pth')  **Step 5: Evaluation**  *# Testing the model*  def evaluate\_model(model, test\_loader):      model.eval()      y\_true = []      y\_pred = []      with torch.no\_grad():          for X\_batch, y\_batch in test\_loader:              outputs = model(X\_batch).squeeze()              predicted = (outputs >= 0.5).float()              y\_true.extend(y\_batch.int().numpy())              y\_pred.extend(predicted.int().numpy())      accuracy = accuracy\_score(y\_true, y\_pred)      conf\_matrix = confusion\_matrix(y\_true, y\_pred)      class\_report = classification\_report(y\_true, y\_pred)      return accuracy, conf\_matrix, class\_report  accuracy, conf\_matrix, class\_report = evaluate\_model(model, test\_loader)  print(f"Accuracy: {accuracy:.2f}")  print(f"Confusion Matrix:\n{conf\_matrix}")  print(f"Classification Report:\n{class\_report}")  *# Plotting confusion matrix*  sns.heatmap(conf\_matrix, annot=True, cmap='Blues', fmt='g')  plt.xlabel('Predicted Label')  plt.ylabel('True Label')  plt.title('Confusion Matrix')  plt.show() |

**CNN-BiLSTM-Attention**

|  |
| --- |
| **Step 1: Import necessary libraries**  *# Necessary Libraries*  import torch  import torch.nn as nn  import torch.optim as optim  from torch.utils.data import DataLoader, Dataset  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import MinMaxScaler  import seaborn as sns  **Step 2: Load dataset and Preprocessing (only D3 dataset)**  *# Load dataset*  data = pd.read\_csv('../data/final\_indoor\_weather\_context.csv', parse\_dates=['date'], index\_col='date')  *# Data preparation*  scaler = MinMaxScaler()  data\_scaled = scaler.fit\_transform(data.drop(columns=['occupancy']))  y = data['occupancy'].values.astype(int)  *# Reshape data for model (samples, timesteps, features)*  input\_dim = data\_scaled.shape[1]  # Number of features in the dataset  X = []  for i in range(len(data\_scaled) - 24):  # Using a window size of 24 timesteps      X.append(data\_scaled[i:i+24])  X = np.array(X)  y = y[24:]  # Shift labels to match the shape of X  *# Splitting the data into train, validation, and test sets*  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp, y\_temp, test\_size=0.5, random\_state=42)  *# Defining a custom dataset class*  class TimeSeriesDataset(Dataset):      def \_\_init\_\_(self, X, y):          self.X = X          self.y = y      def \_\_len\_\_(self):          return len(self.X)      def \_\_getitem\_\_(self, idx):          return torch.tensor(self.X[idx], dtype=torch.float32), torch.tensor(self.y[idx], dtype=torch.float32)  **Step 3: Model Development and Initialization**  *# CNN-BiLSTM-Attention Model Definition*  class CNNBiLSTMAttention(nn.Module):      def \_\_init\_\_(self, input\_dim, cnn\_channels, lstm\_hidden\_units, output\_dim):          super(CNNBiLSTMAttention, self).\_\_init\_\_()          self.cnn = nn.Sequential(              nn.Conv1d(in\_channels=input\_dim, out\_channels=cnn\_channels, kernel\_size=3, stride=1, padding=1),              nn.ReLU(),              nn.MaxPool1d(kernel\_size=2, stride=2)          )          self.bilstm = nn.LSTM(input\_size=cnn\_channels, hidden\_size=lstm\_hidden\_units, num\_layers=1, batch\_first=True, bidirectional=True)          self.attention = nn.Linear(lstm\_hidden\_units \* 2, 1)  # Attention layer          self.dropout = nn.Dropout(0.3)          self.fc = nn.Linear(lstm\_hidden\_units \* 2, output\_dim)          self.sigmoid = nn.Sigmoid()      def forward(self, x):  *# Input shape: (batch\_size, seq\_len, input\_dim)*          x = x.permute(0, 2, 1)  # Change shape to (batch\_size, input\_dim, seq\_len) for CNN          x = self.cnn(x)  # Apply CNN          x = x.permute(0, 2, 1)  # Change shape back to (batch\_size, seq\_len, cnn\_channels) for LSTM          lstm\_out, \_ = self.bilstm(x)  # Apply BiLSTM          # Attention mechanism          attention\_weights = torch.softmax(self.attention(lstm\_out), dim=1)          context\_vector = torch.sum(attention\_weights \* lstm\_out, dim=1)          context\_vector = self.dropout(context\_vector)  # Apply dropout          out = self.fc(context\_vector)  # Fully connected layer          return self.sigmoid(out)  *# Hyperparameters*  input\_dim = 17  # Number of features (e.g., temperature, humidity, etc.)  cnn\_channels = 16  lstm\_hidden\_units = 32  output\_dim = 1  learning\_rate = 0.001  batch\_size = 64  epochs = 50  *# Create dataset and data loaders*  train\_dataset = TimeSeriesDataset(X\_train, y\_train)  val\_dataset = TimeSeriesDataset(X\_val, y\_val)  test\_dataset = TimeSeriesDataset(X\_test, y\_test)  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)  test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)  *# Initialize model, loss function, and optimizer*  model = CNNBiLSTMAttention(input\_dim, cnn\_channels, lstm\_hidden\_units, output\_dim)  criterion = nn.BCELoss()  optimizer = optim.Adam(model.parameters(), lr=learning\_rate)  **Step 4: Model Training**  *# Training loop*  def train\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs):      model.train()      train\_losses = []      val\_losses = []      early\_stopping\_patience = 5      best\_val\_loss = float('inf')      patience\_counter = 0      for epoch in range(epochs):          running\_loss = 0.0          for X\_batch, y\_batch in train\_loader:              optimizer.zero\_grad()              outputs = model(X\_batch).squeeze()              loss = criterion(outputs, y\_batch)              loss.backward()              optimizer.step()              running\_loss += loss.item() \* X\_batch.size(0)          epoch\_loss = running\_loss / len(train\_loader.dataset)          train\_losses.append(epoch\_loss)  *# Validation phase*          model.eval()          val\_running\_loss = 0.0          with torch.no\_grad():              for X\_val\_batch, y\_val\_batch in val\_loader:                  val\_outputs = model(X\_val\_batch).squeeze()                  val\_loss = criterion(val\_outputs, y\_val\_batch)                  val\_running\_loss += val\_loss.item() \* X\_val\_batch.size(0)          val\_loss\_epoch = val\_running\_loss / len(val\_loader.dataset)          val\_losses.append(val\_loss\_epoch)          print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch\_loss:.4f}, Validation Loss: {val\_loss\_epoch:.4f}")  *# Early stopping*          if val\_loss\_epoch < best\_val\_loss:              best\_val\_loss = val\_loss\_epoch              patience\_counter = 0          else:              patience\_counter += 1          if patience\_counter >= early\_stopping\_patience:              print("Early stopping triggered")              break          model.train()      return train\_losses, val\_losses  train\_losses, val\_losses = train\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs)  *# Saving the model*  torch.save(model.state\_dict(), 'Saved\_models/cnn\_bilstm\_attention\_indoor\_weather\_context.pth')  **Step 5: Evaluation**  *# Testing the model*  def evaluate\_model(model, test\_loader):      model.eval()      y\_true = []      y\_pred = []      with torch.no\_grad():          for X\_batch, y\_batch in test\_loader:              outputs = model(X\_batch).squeeze()              predicted = (outputs >= 0.5).float()              y\_true.extend(y\_batch.int().numpy())              y\_pred.extend(predicted.int().numpy())      accuracy = accuracy\_score(y\_true, y\_pred)      conf\_matrix = confusion\_matrix(y\_true, y\_pred)      class\_report = classification\_report(y\_true, y\_pred)      return accuracy, conf\_matrix, class\_report  accuracy, conf\_matrix, class\_report = evaluate\_model(model, test\_loader)  print(f"Accuracy: {accuracy:.2f}")  print(f"Confusion Matrix:\n{conf\_matrix}")  print(f"Classification Report:\n{class\_report}")  *# Plotting confusion matrix*  sns.heatmap(conf\_matrix, annot=True, cmap='Blues', fmt='g')  plt.xlabel('Predicted Label')  plt.ylabel('True Label')  plt.title('Confusion Matrix')  plt.show() |

**Transformer**

|  |
| --- |
| **Step 1: Import necessary libraries**  *# Import necessary libraries*  import numpy as np  import pandas as pd  import torch  import torch.nn as nn  from torch.utils.data import DataLoader, Dataset  import torch.optim as optim  from sklearn.preprocessing import MinMaxScaler  from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  from sklearn.model\_selection import train\_test\_split  import matplotlib.pyplot as plt  import seaborn as sns  **Step 2: Load dataset and Preprocessing (only D3 dataset)**  *# Load dataset*  data = pd.read\_csv('../data/final\_indoor\_weather\_context.csv', parse\_dates=['date'], index\_col='date')  *# Separate features and target*  X = data.drop(columns=['occupancy']).values  # Replace 'occupancy' with the target column if needed  y = data['occupancy'].values  *# Scale features*  scaler = MinMaxScaler()  X\_scaled = scaler.fit\_transform(X)  *# Sequence preparation function*  def create\_sequences(data, target, sequence\_length):      """      Creates sequences of data and their corresponding target labels.      Parameters:      - data (np.array): Scaled feature data      - target (np.array): Target labels      - sequence\_length (int): Number of timesteps in each sequence      Returns:      - X (np.array): Array of sequences      - y (np.array): Array of corresponding labels      """      X, y = [], []      for i in range(len(data) - sequence\_length):          X.append(data[i:i + sequence\_length])          y.append(target[i + sequence\_length])      return np.array(X), np.array(y)  *# Define sequence length*  sequence\_length = 60  # Similar to the previous models  *# Create sequences*  X\_sequences, y\_sequences = create\_sequences(X\_scaled, y, sequence\_length)  *# Split dataset into training, validation, and test sets*  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X\_sequences, y\_sequences, test\_size=0.3, random\_state=42)  X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp, y\_temp, test\_size=0.5, random\_state=42)  *# Define a custom PyTorch dataset*  class TimeSeriesDataset(Dataset):      """      Custom PyTorch dataset for time-series data.      Parameters:      - X (np.array): Input feature sequences      - y (np.array): Target labels      """      def \_\_init\_\_(self, X, y):          self.X = torch.tensor(X, dtype=torch.float32)          self.y = torch.tensor(y, dtype=torch.float32)      def \_\_len\_\_(self):          return len(self.X)      def \_\_getitem\_\_(self, idx):          return self.X[idx], self.y[idx]    *# Create datasets*  train\_dataset = TimeSeriesDataset(X\_train, y\_train)  val\_dataset = TimeSeriesDataset(X\_val, y\_val)  test\_dataset = TimeSeriesDataset(X\_test, y\_test)  *# Define data loaders*  batch\_size = 64  # Adjustable based on memory  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)  test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)  *# Sanity check: Print dataset sizes*  print(f"Training samples: {len(train\_dataset)}")  print(f"Validation samples: {len(val\_dataset)}")  print(f"Test samples: {len(test\_dataset)}")  **Step 3: Model Development and Initialization**  class TransformerModel(nn.Module):      def \_\_init\_\_(self, input\_dim, num\_heads, num\_encoder\_layers, hidden\_dim, dropout=0.1):          super(TransformerModel, self).\_\_init\_\_()  *# Embedding layer to project input features*          self.embedding = nn.Linear(input\_dim, hidden\_dim)    *# Transformer Encoder*          encoder\_layer = nn.TransformerEncoderLayer(              d\_model=hidden\_dim,              nhead=num\_heads,              dim\_feedforward=hidden\_dim \* 4,              dropout=dropout,              batch\_first=True  # Ensure (batch\_size, seq\_len, features) format          )          self.transformer\_encoder = nn.TransformerEncoder(              encoder\_layer,              num\_layers=num\_encoder\_layers          )    *# Fully connected layer for binary output*          self.fc = nn.Linear(hidden\_dim, 1)    *# Activation function for binary classification*          self.sigmoid = nn.Sigmoid()      def forward(self, x):  *# Project input features to hidden dimension*          x = self.embedding(x)  # Shape: (batch\_size, seq\_len, hidden\_dim)    *# Transformer encoder*          transformer\_output = self.transformer\_encoder(x)  # Shape: (batch\_size, seq\_len, hidden\_dim)    *# Pooling: Aggregate over sequence dimension (e.g., mean pooling)*          pooled\_output = torch.mean(transformer\_output, dim=1)  # Shape: (batch\_size, hidden\_dim)    *# Fully connected layer*          output = self.fc(pooled\_output)  # Shape: (batch\_size, 1)          return self.sigmoid(output)  # Apply sigmoid for binary classification  *# Initialize the model with 8 transformer encoder layers*  input\_dim = X\_train.shape[2]  # Number of features in the dataset  model = TransformerModel(      input\_dim=input\_dim,      num\_heads=4,  # Number of attention heads      num\_encoder\_layers=4,  # Increased from 2 to 8 layers      hidden\_dim=128,  # Hidden dimension size      dropout=0.18350578676977397  # Dropout rate  )  print(model)  criterion = nn.BCELoss()  # Binary Cross-Entropy Loss for binary classification  optimizer = torch.optim.Adam(model.parameters(), lr= 0.00037228561102902407)  epochs = 50  **Step 4: Model Training**  *# Training loop with early stopping (Fixed)*  def train\_transformer\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs, early\_stopping\_patience=5):      train\_losses = []      val\_losses = []      best\_val\_loss = float('inf')      patience\_counter = 0      for epoch in range(epochs):  *# Training phase*          model.train()          running\_train\_loss = 0.0          for X\_batch, y\_batch in train\_loader:              optimizer.zero\_grad()    *# Ensure X\_batch has shape (batch\_size, seq\_len, feature\_dim)*  *# No need to unsqueeze here*              outputs = model(X\_batch).squeeze()  # Shape: (batch\_size,)              y\_batch = y\_batch.float().view\_as(outputs)  # Ensure target matches output shape    *# Compute loss*              loss = criterion(outputs, y\_batch)              loss.backward()              optimizer.step()                running\_train\_loss += loss.item() \* X\_batch.size(0)    *# Average training loss*          epoch\_train\_loss = running\_train\_loss / len(train\_loader.dataset)          train\_losses.append(epoch\_train\_loss)  *# Validation phase*          model.eval()          running\_val\_loss = 0.0          with torch.no\_grad():              for X\_val\_batch, y\_val\_batch in val\_loader:  *# No need to unsqueeze here*                  val\_outputs = model(X\_val\_batch).squeeze()  # Shape: (batch\_size,)                  y\_val\_batch = y\_val\_batch.float().view\_as(val\_outputs)  # Ensure target matches output shape    *# Compute loss*                  val\_loss = criterion(val\_outputs, y\_val\_batch)                  running\_val\_loss += val\_loss.item() \* X\_val\_batch.size(0)    *# Average validation loss*          epoch\_val\_loss = running\_val\_loss / len(val\_loader.dataset)          val\_losses.append(epoch\_val\_loss)  *# Print losses*          print(f"Epoch {epoch+1}/{epochs}, Train Loss: {epoch\_train\_loss:.4f}, Validation Loss: {epoch\_val\_loss:.4f}")  *# Early stopping mechanism*          if epoch\_val\_loss < best\_val\_loss:              best\_val\_loss = epoch\_val\_loss              patience\_counter = 0              best\_model\_state = model.state\_dict()  # Save the best model state          else:              patience\_counter += 1          if patience\_counter >= early\_stopping\_patience:              print("Early stopping triggered. Restoring best model state.")              model.load\_state\_dict(best\_model\_state)  # Restore the best model state              break      return train\_losses, val\_losses  *# Train the Transformer-based model*  train\_losses, val\_losses = train\_transformer\_model(model, train\_loader, val\_loader, criterion, optimizer, epochs)  *# Saving the model*  torch.save(model.state\_dict(), 'Saved\_models/transformer\_indoor\_context\_weather\_SL\_60\_BS\_64\_HD\_128.pth')  **Step 5: Evaluation**  *# Testing loop*  def evaluate\_transformer\_model(model, test\_loader):      model.eval()      y\_true = []      y\_pred = []      with torch.no\_grad():          for X\_batch, y\_batch in test\_loader:              # Forward pass              outputs = model(X\_batch).squeeze()  # Shape: (batch\_size,)              predictions = (outputs >= 0.5).float()  # Threshold for binary classification    *# Collect true and predicted labels*              y\_true.extend(y\_batch.numpy())              y\_pred.extend(predictions.numpy())  *# Calculate metrics*      accuracy = accuracy\_score(y\_true, y\_pred)      conf\_matrix = confusion\_matrix(y\_true, y\_pred)      class\_report = classification\_report(y\_true, y\_pred, zero\_division=1)      print(f"Accuracy: {accuracy:.2f}")      print(f"Confusion Matrix:\n{conf\_matrix}")      print(f"Classification Report:\n{class\_report}")      return accuracy, conf\_matrix, class\_report  *# Evaluate the Transformer-based model*  accuracy, conf\_matrix, class\_report = evaluate\_transformer\_model(model, test\_loader)  sns.heatmap(conf\_matrix, annot=True, cmap='Blues', fmt='g')  plt.xlabel('Predicted Label')  plt.ylabel('True Label')  plt.title('Confusion Matrix')  plt.show() |

**4. Evaluation**

**4.1 Metrics**

* **Accuracy**: (TP + TN) / (TP + TN + FP + FN)
* **Precision**: TP / (TP + FP)
* **Recall**: TP / (TP + FN)
* **F1-Score**: 2 \* (Precision \* Recall) / (Precision + Recall)

**5. SHAP Analysis**

**5.1 Summary plot**

**Explainable AI (SHAP Analysis)**

|  |
| --- |
| import shap  import numpy as np  import torch  *# Define feature names for D3 dataset*  feature\_names = [      'indoor\_temp', 'indoor\_humidity', 'light', 'co2', 'indoor\_humidity\_ratio',      'outdoor\_temp', 'outdoor\_humidity', 'dew\_point', 'wind\_speed', 'wind\_direction',      'wind\_gust', 'sea\_level\_pressure', 'cloud\_cover', 'visibility', 'hour\_of\_day',      'day\_of\_week', 'is\_weekend'  ]  *# Define a prediction function compatible with SHAP*  def predict\_fn(input\_data):      """      Prediction function for SHAP analysis. Reshapes the 2D flattened input data back to 3D.      """  *# Reshape the input data back to (samples, timesteps, features)*      timesteps = 60  # Match the sequence length used in training      features = input\_data.shape[1] // timesteps      input\_tensor = torch.tensor(input\_data.reshape(-1, timesteps, features), dtype=torch.float32)        with torch.no\_grad():          predictions = model(input\_tensor).squeeze().numpy()      return predictions.flatten()  # Return a 1D array of predictions  *# Flatten the test dataset for SHAP*  X\_test\_flat = X\_test.numpy().reshape(X\_test.shape[0], -1)  # Flatten the test set to 2D  X\_background\_2d = X\_test\_flat[:50]  # Use the first 50 samples as background  X\_test\_subset\_2d = X\_test\_flat[:50]  # Use the first 20 samples for SHAP analysis  *# Initialize SHAP KernelExplainer*  explainer = shap.KernelExplainer(predict\_fn, X\_background\_2d)  *# Compute SHAP values*  print("Computing SHAP values, this might take a while...")  shap\_values = explainer.shap\_values(X\_test\_subset\_2d)  *# Aggregate SHAP values across timesteps for each feature*  timesteps = 60  # Match the sequence length  num\_features = len(feature\_names)  *# Reshape SHAP values to (num\_samples, timesteps, num\_features)*  shap\_values\_reshaped = np.array(shap\_values).reshape(      len(X\_test\_subset\_2d), timesteps, num\_features  )  *# Aggregate SHAP values by summing across timesteps*  shap\_values\_aggregated = shap\_values\_reshaped.sum(axis=1)  **Generate the SHAP value Table**  import pandas as pd  import numpy as np  *# Assuming `shap\_values\_aggregated` and `feature\_names` are available*  mean\_shap\_values = np.mean(np.abs(shap\_values\_aggregated), axis=0)  *# Create a DataFrame*  shap\_table = pd.DataFrame({      'Feature Name': feature\_names,      'Mean SHAP Value': mean\_shap\_values  })  *# Add rank column based on mean SHAP value*  shap\_table['Rank'] = shap\_table['Mean SHAP Value'].rank(ascending=False).astype(int)  *# Sort the table by rank*  shap\_table = shap\_table.sort\_values(by='Rank')  *# Display the table*  print(shap\_table)  **SHAP Summary Plot for Contextual Features**  import shap  import matplotlib.pyplot as plt  *# Define feature groups*  contextual\_features = ['hour\_of\_day', 'day\_of\_week', 'is\_weekend']  weather\_features = ['wind\_speed', 'cloud\_cover', 'outdoor\_temp', 'outdoor\_humidity', 'dew\_point']  *# Identify feature indices*  contextual\_indices = [feature\_names.index(f) for f in contextual\_features]  weather\_indices = [feature\_names.index(f) for f in weather\_features]  *# Set custom font properties for Matplotlib*  plt.rcParams['font.family'] = 'Arial'  plt.rcParams['font.size'] = 10  # Default font size for other text elements  *# Create a SHAP summary plot for contextual features*  *# plt.figure(figsize=(3.5, 3))  # Set the figure size*  shap.summary\_plot(      shap\_values\_aggregated[:, contextual\_indices],      X\_test\_subset\_2d[:, contextual\_indices],      feature\_names=[feature\_names[i] for i in contextual\_indices],      show=False  # Prevent automatic display  )  *# Resize the current figure*  plt.gcf().set\_size\_inches(3.5, 3)  # Set width and height in inches  *# Adjust x-axis label font size*  ax = plt.gca()  # Get the current Axes object  ax.set\_xlabel("SHAP value", fontsize=10)  # Set x-axis label and font size  *# Adjust ticks font size*  plt.title("(a)", fontsize=12, fontname="Arial")  plt.xticks(fontsize=9)  plt.yticks(fontsize=9)  plt.tight\_layout()  # Ensure proper layout  *# Save the figure in multiple formats*  formats = ['jpg', 'png', 'svg', 'tiff']  for fmt in formats:      plt.savefig(f"SHAP\_SUMMARY/LSTM\_Contextual\_Features\_SHAP\_Summary.{fmt}", dpi=300, format=fmt)  plt.show()  **SHAP Summary Plot for Contextual Features**  *# Set custom font properties for Matplotlib*  plt.rcParams['font.family'] = 'Arial'  plt.rcParams['font.size'] = 10  # Default font size for other text elements  *# Create the SHAP summary plot*  shap.summary\_plot(      shap\_values\_aggregated[:, weather\_indices],      X\_test\_subset\_2d[:, weather\_indices],      feature\_names=[feature\_names[i] for i in weather\_indices],      show=False  # Prevent the plot from displaying immediately  )  *# Resize the current figure*  plt.gcf().set\_size\_inches(3.5, 3)  # Set width and height in inches  *# Adjust x-axis label font size*  ax = plt.gca()  # Get the current Axes object  ax.set\_xlabel("SHAP value", fontsize=8)  # Set x-axis label and font size  *# Adjust ticks font size*  plt.title("(a)", fontsize=12, fontname="Arial")  plt.xticks(fontsize=9)  plt.yticks(fontsize=9)  plt.tight\_layout()  # Ensure proper layout  *# Save the figure in multiple formats*  formats = ['jpg', 'png', 'svg', 'tiff']  for fmt in formats:      plt.savefig(f"SHAP\_SUMMARY/LSTM\_Weather\_Features\_SHAP\_Summary.{fmt}", dpi=300, format=fmt)  plt.show() |

**6. Conclusion**

* Contextual and weather data significantly enhance occupancy detection.
* The Transformer model achieved the highest performance across all metrics.
* SHAP analysis validated the critical role of contextual and weather features.