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 (CO_2, 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, CO_2 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_mo
del_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_regulariz
ed_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_mo
del.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.pt
h')
```

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_Summ
ary.{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.