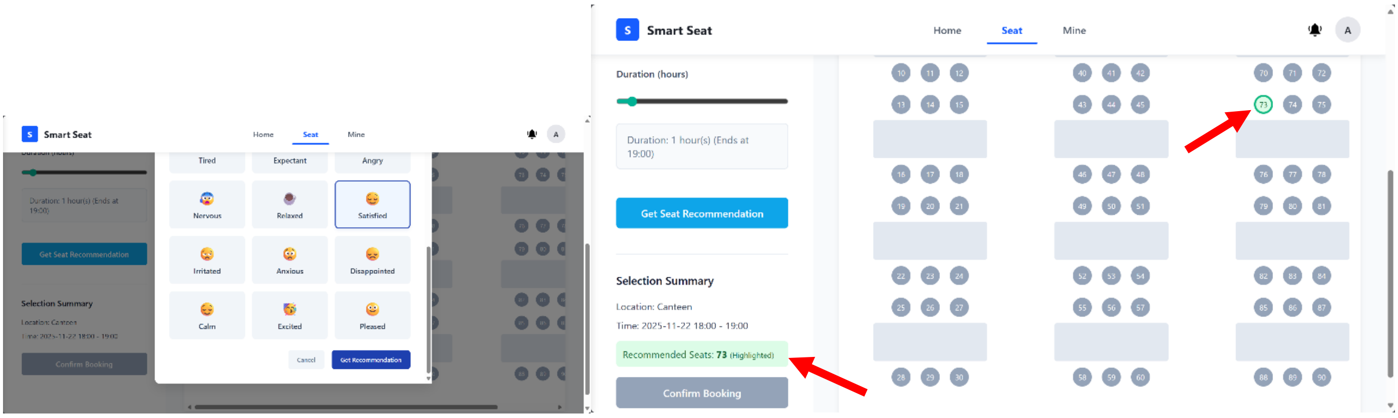
AI-Powered Seat Recommendation Feature

Abstract

This document details the design, implementation, and integration of an AI-driven seat recommendation feature within the seat reservation system for James Cook University (JCU) faculty and students. The feature leverages a machine learning model trained on 20,000 simulated seat reservation records to recommend seats based on user mood, alongside other key factors (e.g., room, time slot, and booked seats). The workflow includes data collection, preprocessing, model training, and integration with the existing system (built with Express and Flask frameworks). This feature enhances user experience by reducing seat-selection time and providing personalized recommendations, while remaining compatible with both student and instructor roles in the system.



1. Purpose of the AI Feature

The core goals of the AI recommendation feature are:

- Reduce user effort in seat selection by providing personalized suggestions.
- Leverage historical booking data to align recommendations with user preferences.
- Maintain compatibility with the existing system architecture and user roles.

2. Data Collection

2.1 Data Source and Rationale

To train the AI model, we generated a **simulated dataset** (seat_reservation_data.csv) containing over 20,000 records. Simulated data was used to ensure coverage of diverse scenarios (e.g., different rooms, time slots, and moods) without relying on real user data (which could raise privacy concerns). The data was created by:

1. Defining mock room IDs (e.g., A2-02, library-1, canteen) representing JCU spaces (classrooms, libraries, canteens).
2. Simulating "booked seats" for each room and time slot (to mimic real-world occupancy).
3. Prompting users (in a test environment) to select a seat based on a given mood (e.g., "Bored", "Focused", "Happy").

1241	B2-03	18:00:00	Happy	5,21,10,20,	72
320	C3-06	6:00:00	Relaxed	8,46,23,67,	58
770	C3-14	12:00:00	Surprised	14,86,73,1	50
2170	C2-13	17:00:00	Irritated	44,94,130,	133
258	A3-04	9:00:00	Nervous	68,47,69,8	84
1172	C3-06	10:00:00	Happy	10,44,19,8	22
555	E2-02	1:00:00	Disappoin	2,44,46,1,1	84
1088	C4-14	7:00:00	Happy	119,24,114	137
1412	C1-05	13:00:00	Excited	84,69,59,2	5
500	B1-02	6:00:00	Sad	58,8,31,67,	60
540	B2-07	12:00:00	Relaxed	82,11,34,3	5
234	C1-01	3:00:00	Grieving	81,10,33,2	19
1300	C4-04	14:00:00	Sad	81,69,82,1	23
629	B2-04	2:00:00	Anxious	22,80,3,33,	56
290	C3-04	11:00:00	Irritated	80,20,7,83,	34
1182	B3-04	19:00:00	Irritated	4,44,67,5,3	59
1179	C3-03	12:00:00	Irritated	9,33,47,11,	80
1236	B2-06	8:00:00	Bored	82,70,57,2,	72
817	canteen	5:00:00	Angry	18,34,62,5	84
1857	C3-15	21:00:00	Irritated	71,69,15,1	29
15	E2-07	5:00:00	Focused	84,70,1,34,	69
1390	A1-05	11:00:00	Relaxed	72,4,70,43,	48
324	B3-07	11:00:00	Expectant	10,11,80,5,	4
715	B1-01	2:00:00	Disappoin	20,84,68,5	23
893	B3-07	15:00:00	Bored	31,71,12,7,	56
668	C3-02	12:00:00	Satisfied	44,19,82,5	57
1821	C4-14	11:00:00	Calm	127,67,37,	61
1563	C1-06	5:00:00	Disappoin	58,57,80,7,	10
1979	E2-04	15:00:00	Satisfied	67,21,70,8,	5
1742	canteen	9:00:00	Grieving	7,13,42,32,	75
1726	C4-07	11:00:00	Surprised	43,83,45,4	1
1383	A2-03	0:00:00	Pleased	67,79,44,1	48
824	E2-06	21:00:00	Irritated	84,58,4,48,	81
65	C1-01	16:00:00	Satisfied	58,84,5,35,	55
1641	A1-06	10:00:00	Calm	5,12,3,83,4	11
2175	C1-01	8:00:00	Anxious	82,79,81,7	55
1080	library-1	10:00:00	Pleased	32,28,22,1	24
982	C3-01	1:00:00	Focused	6,33,71,79,	84
161	C4-13	7:00:00	Nervous	64,85,89,6	84
1983	C4-04	12:00:00	Excited	56,59,5,31,	20
1921	C3-14	18:00:00	Bored	98,127,62,	110
612	A3-01	0:00:00	Satisfied	21,79,72,5	20
2052	B1-05	19:00:00	Expectant	59,6,35,23,	1
1383	E2-06	19:00:00	Angry	56,69,32,5	22

2.2 Dataset Overview

The dataset includes 5 key columns, each serving a critical role in training the model:

Column Name	Description
room_id	Identifier for the booking location (e.g., A1-04 = Classroom A1-04, canteen = Campus Canteen).
time_slot	Booking time in HH:MM:SS format (e.g., 11:00:00 = 11 AM).
user_mood	User’s self-reported mood (categorical: e.g., Happy, Bored, Focused).
booked_seats	Comma-separated list of already booked seats (e.g., 32,59,58 = Seats 32, 59, 58 are taken).
selected_seat	The seat the user ultimately chose (target variable for the model).

3. Data Preprocessing

Raw data required preprocessing to convert it into a format suitable for machine learning. All preprocessing steps were implemented in Python (using pandas, scikit-learn, and PyTorch), as outlined below:

3.1 Key Preprocessing Steps

3.1.1 Extract Room Number

The room_id (e.g., A2-02) was processed to extract numeric values (e.g., 02 from A2-02) using a regex function (extract_room_number). This numeric feature helps the model distinguish between different rooms.

```python
def extract_room_number(room_str):
    if pd.isna(room_str):
        return 0
    match = re.search(r'\d+', str(room_str))
    return int(match.group()) if match else 0
```

3.1.2 Convert Time to Numeric Format

The time_slot (HH:MM:SS) was converted to total minutes since midnight (e.g., 11:00:00 → 660 minutes) via the time_to_minutes function. This simplifies time-based pattern learning. The converted time was further normalized to a 0–1 range using MinMaxScaler (stored as time_norm) to ensure consistency with other features.

3.1.3 Encode Categorical Features

- **Mood Encoding**: The categorical user_mood column (e.g., Happy, Bored) was converted to integers using LabelEncoder (stored as mood_encoded). For example, Happy → 2, Bored → 0. The encoder was saved as mood_encoder.pkl for later use in inference.

```python
mood_encoder = LabelEncoder()
df['user_mood'] = df['user_mood'].fillna('unknown')
df['mood_encoded'] = mood_encoder.fit_transform(df['user_mood'])
```

- **Room Encoding**: Similarly, room_id was encoded to integers (room_encoded) using LabelEncoder (saved as room_encoder.pkl).

3.1.4 Process Booked Seats

The booked_seats column (comma-separated strings) was parsed into a set of integers (reserved_seats_set) to track occupied seats. We also calculated reserved_count (number of booked seats in the room) and created a 140-dimensional binary vector (reserved) where each index represents a seat: 1 if the seat is booked, 0 otherwise (140 was chosen as the maximum number of seats across all rooms).

3.1.5 Validate and Clean Selected Seats

To ensure selected_seat (the model's target) was valid:

1. We defined room_constraints (max seats per room: e.g., canteen has 90 seats, library-1 has 64 seats).
2. For invalid seats (e.g., a seat number exceeding the room's max or missing values), we replaced the value with the room_seat_mode (the most frequently selected seat for that room, stored in room_seat_modles.pkl).
3. Final selected_seat values were clipped to 1–140 and converted to integers.

3.2 Dataset Splitting

The preprocessed data was split into a **training set (80%)** and **validation set (20%)** using train_test_split with stratify=df['room_encoded'] to ensure balanced room distribution across sets. A custom SeatDataset class (inheriting from torch.utils.data.Dataset) was used to load data in batches for training.

```python
class SeatDataset(Dataset):
    def __init__(self, dataframe):
        self.data = dataframe
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        row = self.data.iloc[idx]
        room_enc = row['room_encoded']
        mood_enc = row['mood_encoded']
        time_norm = row['time_norm']
        room_num = row['room_number']
        reserved_count = row['reserved_count']
        max_seats = row['max_seats']
        reserved = torch.zeros(140)
        for seat in row['reserved_seats_set']:
            if 1 <= seat <= 140:
                reserved[seat - 1] = 1
```

```python
        features = torch.cat([features, reserved])
        label = torch.tensor(row['selected_seat'], dtype=torch.long)
        return features, label

train_df, val_df = train_test_split(df, test_size=0.2, random_state=42,
stratify=df['room_encoded'])
train_dataset = SeatDataset(train_df)
val_dataset = SeatDataset(val_df)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, n
um_workers=2)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False, num_
workers=2)
```

4. Model Design and Training

4.1 Model Selection Rationale

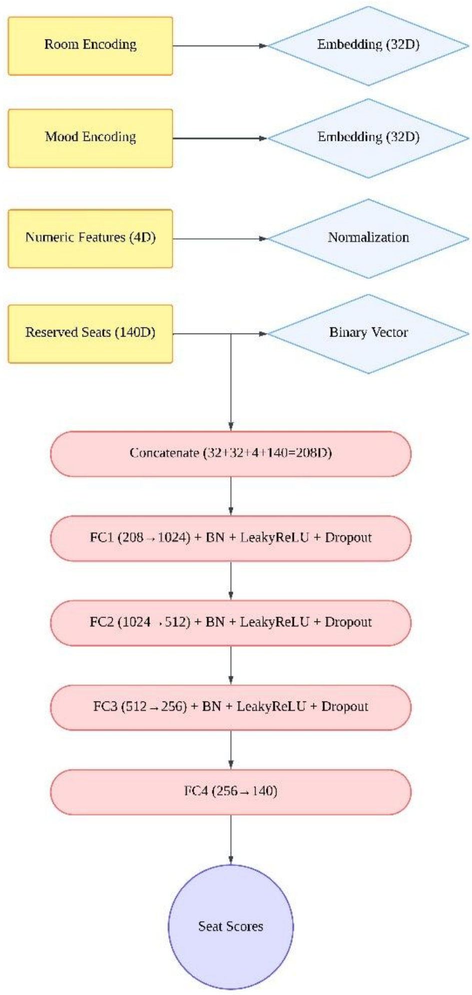
We chose a **neural network** (built with PyTorch) for the SeatRecommender model because it excels at:

- Fusing multiple feature types (categorical: room/mood; numeric: time/reserved count; binary: booked seats).
- Learning complex patterns (e.g., how "Focused" moods correlate with specific seats in library-1 at 10 AM).

4.2 Model Architecture

The SeatRecommender model consists of 4 key components (Figure 1):

1. **Embedding Layers**: Convert categorical features (room and mood encodings) into low-dimensional vectors to capture semantic relationships (e.g., similar rooms share similar embeddings).
   - room_emb: Embedding layer for room_encoded (input dim: number of unique rooms, output dim: 32).
   - mood_emb: Embedding layer for mood_encoded (input dim: number of unique moods, output dim: 32).
2. **Feature Concatenation**: Combine embeddings, numeric features (time_norm, room_number, reserved_count, max_seats), and the binary reserved vector into a single input tensor.
3. **Fully Connected Layers**: Transform the concatenated features into seat recommendation scores:
   - fc1: 32 (room emb) + 32 (mood emb) + 4 (numeric) + 140 (reserved) → 1024 units.
   - fc2: 1024 → 512 units.
   - fc3: 512 → 256 units.
   - fc4: 256 → 140 units (output: score for each seat, 1–140).
4. **Regularization Layers**: Prevent overfitting:
   - Batch Normalization (bn1, bn2, bn3): Stabilize training by normalizing layer inputs.
   - Dropout (dropout=0.35): Randomly deactivate 35% of neurons during training.
   - Leaky ReLU Activation: Introduce non-linearity to learn complex patterns.



Simplified SeatRecommender Model Architecture

4.3 Training Configuration

4.3.1 Hardware and Loss Function

- **Device**: Training used GPU (CUDA) if available; fallback to CPU (via torch.device).
- **Loss Function**: CrossEntropyLoss with class weights to address imbalanced seat selection (e.g., if Seat 45 is chosen more often, the model is penalized less for mispredicting it). Weights were calculated as len(df) / seat_counts[seat] (stored in weights tensor).

4.3.2 Optimization and Regularization

- **Optimizer**: AdamW (learning rate = 0.001, weight decay = 0.0001) to optimize weights and prevent overfitting.
- **Learning Rate Scheduler**: ReduceLROnPlateau (patience=3, factor=0.5) to halve the learning rate if validation loss stops improving.
- **Early Stopping**: Custom EarlyStopping class (patience=7, min_delta=0.0001) to stop training early if validation loss does not improve, avoiding overfitting.

4.4 Training Process and Results

Training ran for up to 100 epochs, with key metrics tracked per epoch:

- **Training Loss:** Average loss over the training set.
- **Validation Loss:** Average loss over the validation set.
- **Validation Accuracy:** Percentage of correct seat predictions (matching selected_seat).

Key outcomes:

- Early stopping was triggered at **Epoch 13** (the custom EarlyStopping mechanism detected that validation loss stopped improving).
- The learning rate scheduler (ReduceLROnPlateau) adjusted the learning rate from 0.001 to 0.0005 at Epoch 11 to refine training.
- Final validation accuracy: **~28.38%** (the model correctly predicts the user's chosen seat in nearly 29% of cases). Training loss showed a steady downward trend (from 4.4755 at Epoch 1 to 2.7643 at Epoch 13), while validation loss fluctuated, indicating opportunities for further model optimization (e.g., hyperparameter adjustment, data diversity enhancement).

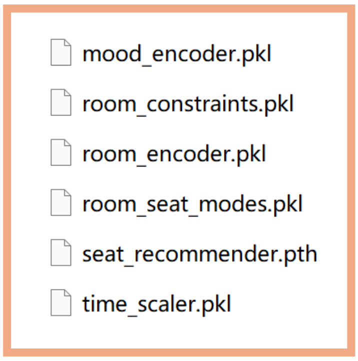
```
Epoch  1, Train Loss: 4.4755, Val Loss: 3.8665, Val Acc: 0.1943, LR:
0.001000
Epoch  2, Train Loss: 3.8001, Val Loss: 3.6545, Val Acc: 0.2318, LR:
0.001000
Epoch  3, Train Loss: 3.6229, Val Loss: 3.6776, Val Acc: 0.2507, LR:
0.001000
Epoch  4, Train Loss: 3.5163, Val Loss: 3.6466, Val Acc: 0.2520, LR:
0.001000
Epoch  5, Train Loss: 3.4297, Val Loss: 3.6995, Val Acc: 0.2565, LR:
0.001000
Epoch  6, Train Loss: 3.3588, Val Loss: 3.6205, Val Acc: 0.2645, LR:
0.001000
Epoch  7, Train Loss: 3.2993, Val Loss: 3.6990, Val Acc: 0.2705, LR:
0.001000
Epoch  8, Train Loss: 3.2725, Val Loss: 3.7517, Val Acc: 0.2797, LR:
0.001000
Epoch  9, Train Loss: 3.1925, Val Loss: 3.7301, Val Acc: 0.2737, LR:
0.001000
```

```
Epoch  9, Train Loss: 3.1925, Val Loss: 3.7301, Val Acc: 0.2737, LR:
0.001000
Epoch 10, Train Loss: 3.1180, Val Loss: 3.8555, Val Acc: 0.2595, LR:
0.001000
Epoch 11, Train Loss: 2.9296, Val Loss: 3.8584, Val Acc: 0.2765, LR:
0.000500
Epoch 12, Train Loss: 2.8382, Val Loss: 4.0095, Val Acc: 0.2807, LR:
0.000500
Epoch 13, Train Loss: 2.7643, Val Loss: 4.0197, Val Acc: 0.2838, LR:
0.000500
Early stopping
All files saved to /kaggle/working/
Files available for download:
- seat_recommender.pth (model weights)
- mood_encoder.pkl (mood label encoder)
- room_encoder.pkl (room label encoder)
- time_scaler.pkl (time normalization scaler)
- room_seat_modes.pkl (room seat mode statistics)
- room_constraints.pkl (room seat constraints)
```

4.5 Model Saving

Post-training, the following files were saved for inference:

- seat_recommender.pth: Trained model weights.
- mood_encoder.pkl, room_encoder.pkl: Encoders for categorical features.
- time_scaler.pkl: Scaler for time normalization.
- room_seat_modes.pkl, room_constraints.pkl: Room-specific seat rules.



5. Frontend and Backend Integration

The AI recommendation feature was integrated into the existing JCU Seat Reservation System, which uses **Express.js** as the main backend framework. A secondary **Flask** backend was added to handle model inference (since the model is Python-based, Flask simplifies Python-to-JavaScript communication).

5.1 Backend Integration (Express + Flask)

5.1.1 Flask Backend (Model Inference)

The Flask backend (seat-predict.py) acts as an API service for seat recommendations. Its core workflow:

1. **Load Pre-trained Assets**: On startup, load seat_recommender.pth, encoders, and scalers.
2. **Expose Recommendation Endpoint**: A POST /api/ai-recommend-seat endpoint accepts JSON input with:
    - room_id (user's chosen room),
    - time_slot (user's desired time),
    - user_mood (user's selected mood),
    - booked_seats (current booked seats in the room).
3. **Preprocess Input**: Apply the same preprocessing steps as training (e.g., encode mood/room, normalize time, create reserved vector).
4. **Generate Recommendation**: Pass preprocessed features to the model, get seat scores, and select the **highest-scoring seat that is not booked**.
5. **Return Result**: Send the recommended seat number back to Express.

```
app = Flask(__name__)
CORS(app)

model_dir = '/app/models'
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

try:
    with open(os.path.join(model_dir, 'mood_encoder.pkl'), 'rb') as f:
        mood_encoder = pickle.load(f)
    with open(os.path.join(model_dir, 'room_encoder.pkl'), 'rb') as f:
        room_encoder = pickle.load(f)
    with open(os.path.join(model_dir, 'time_scaler.pkl'), 'rb') as f:
        scaler = pickle.load(f)
    with open(os.path.join(model_dir, 'room_seat_modes.pkl'), 'rb') as
        room_seat_modes = pickle.load(f)
    with open(os.path.join(model_dir, 'room_constraints.pkl'), 'rb') as
        room_constraints = pickle.load(f)
    print("successful")
except Exception as e:
    print(f"fail to load: {str(e)}")
    raise

class SeatRecommender(nn.Module):
    def __init__(self, num_rooms, num_moods, embedding_dim=32):
        super().__init__()
        self.room_emb = nn.Embedding(num_rooms, embedding_dim)
        self.mood_emb = nn.Embedding(num_moods, embedding_dim)
        self.fc1 = nn.Linear(embedding_dim*2 + 4 + 140, 1024)
        self.bn1 = nn.BatchNorm1d(1024)
        self.fc2 = nn.Linear(1024, 512)
        self.bn2 = nn.BatchNorm1d(512)
```

```
class SeatRecommender(nn.Module):
    def __init__(self, num_rooms, num_moods, embedding_dim=32):
        super().__init__()
        self.room_emb = nn.Embedding(num_rooms, embedding_dim)
        self.mood_emb = nn.Embedding(num_moods, embedding_dim)
        self.fc1 = nn.Linear(embedding_dim*2 + 4 + 140, 1024)
        self.bn1 = nn.BatchNorm1d(1024)
        self.fc2 = nn.Linear(1024, 512)
        self.bn2 = nn.BatchNorm1d(512)
        self.fc3 = nn.Linear(512, 256)
        self.bn3 = nn.BatchNorm1d(256)
        self.fc4 = nn.Linear(256, 140)
        self.dropout = nn.Dropout(0.35)
        self.leaky_relu = nn.LeakyReLU(0.2)
    def forward(self, x):
        room_enc = x[:, 0].long()
        mood_enc = x[:, 1].long()
        num_feats = x[:, 2:6]
        reserved = x[:, 6:]
        room_emb = self.room_emb(room_enc).squeeze(1)
        mood_emb = self.mood_emb(mood_enc).squeeze(1)
        combined = torch.cat([room_emb, mood_emb, num_feats, reserved], dim=1)
        x = self.leaky_relu(self.bn1(self.fc1(combined)))
        x = self.dropout(x)
        x = self.leaky_relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.leaky_relu(self.bn3(self.fc3(x)))
        x = self.dropout(x)
        x = self.fc4(x)
        return x

try:
```

```
try:
    num_rooms = len(room_encoder.classes_) if hasattr(room_encoder, 'classes_') else 0
    num_moods = len(mood_encoder.classes_) if hasattr(mood_encoder, 'classes_') else 0
    model = SeatRecommender(num_rooms, num_moods)
    model.load_state_dict(torch.load(os.path.join(model_dir, 'seat_recommender.pth'), map_
    model.to(device)
    model.eval()
    print("successful")
except Exception as e:
    print(f"fail: {str(e)}")
    raise

def extract_room_number(room_str):
    if not room_str or pd.isna(room_str):
        return 0
    match = re.search(r'\d+', str(room_str))
    return int(match.group()) if match else 0

def time_to_minutes(time_str):
    if not time_str:
        return 0
    try:
        parts = time_str.split(':')
        if len(parts) == 2:
            h, m = map(int, parts)
        elif len(parts) == 3:
            h, m, _ = map(int, parts)
        else:
            return 0
        return h * 60 + m
    except Exception as e:
        print(f"fail: {str(e)}")
        return 0
```

```
@app.route('/api/predict-seat', methods=['POST'])
def predict_seat():
    try:
        req_data = request.get_json()
        print("=== Flask recive ===")
        print(req_data)

        if not req_data:
            return jsonify({'error': 'data error'}), 400

        room_id = req_data.get('room_id', '')
        time_slot = req_data.get('time_slot', '')
        booked_seats = req_data.get('booked_seats', '')
        user_mood = req_data.get('user_mood', 'unknown')

        room_number = extract_room_number(room_id)
        time_minutes = time_to_minutes(time_slot)
        time_norm = scaler.transform([[time_minutes]])[0][0] if scaler else 0

        try:
            mood_enc = mood_encoder.transform([user_mood])[0] if user_mood in mood_encode
        except:
            mood_enc = 0
            print(f"unknown {user_mood}, defult 0")

        try:
            room_enc = room_encoder.transform([room_id])[0] if room_id in room_encoder.cl
        except:
            room_enc = 0
            print(f"unknown {room_id}, defult 0")
```

5.1.2 Express Backend (Main System)

The Express backend handles core system logic (user authentication, role management, booking storage) and communicates with Flask:

1. When a user requests an AI recommendation, Express validates the user's role (student/instructor) and booking permissions.

2. Express forwards the user's input (room, time, mood, booked seats) to the Flask endpoint.
3. Express receives the recommended seat from Flask and returns it to the frontend.

```javascript
router.post('/seat', async (req, res) => {
  try {
    const { room_id, time_slot, booked_seats, user_mood } = req.body;

    if (!room_id || !time_slot || !user_mood) {
      return res.status(400).json({ msg: 'miss: room_id、time_slot、user_mood
    }

    console.log('=== recive ===');
    console.log({ room_id, time_slot, booked_seats, user_mood });

    const flaskResponse = await axiosInstance.post(PREDICTION_SERVICE_URL, {
      room_id: room_id.trim(),
      time_slot: time_slot.trim(),
      booked_seats: (booked_seats || '').trim(),
      user_mood: user_mood.trim()
    });

    console.log('=== Flask return ===');
    console.log(flaskResponse.data);
    res.json(flaskResponse.data);

  } catch (err) {
    console.error('=== error ===');
    console.error('Flask address:', PREDICTION_SERVICE_URL);
    console.error('error type:', err.name);
    console.error('detail:', err.response?.data || err.message || err);
```

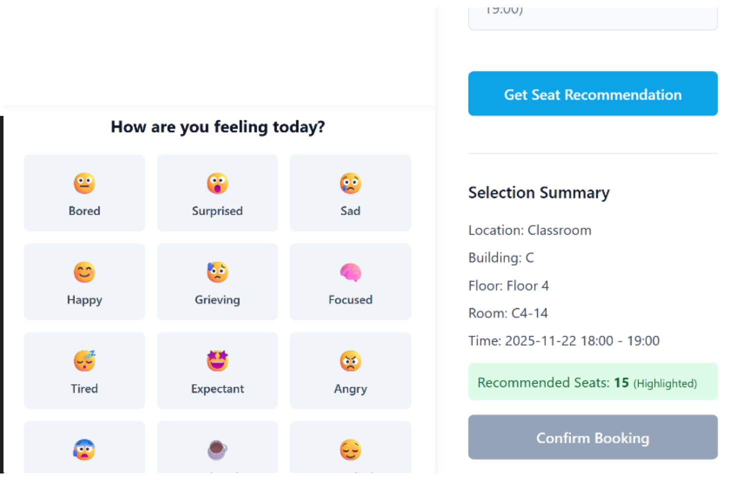
5.2 Frontend Integration (seat.js)

The frontend (built with JavaScript, seat.js) adds a user-friendly interface for the AI feature:

5.2.1 UI Components

1. **AI Recommend Button**: Added to the seat booking page (next to room/time selectors). Clicking this button triggers a mood selection popup.
2. **Mood Selection Popup**: A modal with mood options (e.g., "Happy", "Focused", "Bored")—matching the categories in the training data. The popup uses a clean, card-based design for readability.
3. **Recommendation Display**: Once a recommendation is received, the recommended seat is highlighted in the room's seat map (e.g., green border around Seat 47) with a "Recommended for your mood" tooltip.

```javascript
const moodOptions = [
  { value: 'Bored', label: 'Bored', icon: '😐' },
  { value: 'Surprised', label: 'Surprised', icon: '😮' },
  { value: 'Sad', label: 'Sad', icon: '😢' },
  { value: 'Happy', label: 'Happy', icon: '😊' },
  { value: 'Grieving', label: 'Grieving', icon: '😥' },
  { value: 'Focused', label: 'Focused', icon: '🧠' },
  { value: 'Tired', label: 'Tired', icon: '😴' },
  { value: 'Expectant', label: 'Expectant', icon: '🤩' },
  { value: 'Angry', label: 'Angry', icon: '😠' },
  { value: 'Nervous', label: 'Nervous', icon: '😰' },
  { value: 'Relaxed', label: 'Relaxed', icon: '☕' },
  { value: 'Satisfied', label: 'Satisfied', icon: '😌' },
  { value: 'Irritated', label: 'Irritated', icon: '😒' },
  { value: 'Anxious', label: 'Anxious', icon: '😟' },
  { value: 'Disappointed', label: 'Disappointed', icon: '😞' },
  { value: 'Calm', label: 'Calm', icon: '😑' },
  { value: 'Excited', label: 'Excited', icon: '🥳' },
  { value: 'Pleased', label: 'Pleased', icon: '🙂' },
];
```



5.2.2 API Communication (seat.js)

seat.js handles frontend-backend interaction:

1. On clicking "AI Recommend", collect the user's selected room_id, time_slot, and chosen user_mood.

2. Fetch the current booked_seats for the selected room/time from the Express backend.
3. Send a POST request to the Express endpoint (which forwards to Flask) with the collected data.
4. On receiving the recommended seat, update the seat map to highlight the suggestion.

```
const handleGetRecommendation = async () => {
  if (!selectedMood || !getRoomIdentifier() || !selectedHour || !selectedDate) {
    setErrorMsg('fill first');
    return;
  }
  setLoading(true);
  setErrorMsg('');
  try {
    const room = getRoomIdentifier();
    const time_slot = `${selectedHour}:00`;
    const booked_seats = Object.keys(bookedSeats).join(',');

    const response = await axios.post('/api/recommend/seat', {
      room_id: room,
      time_slot: time_slot,
      booked_seats: booked_seats,
      user_mood: selectedMood
    });

    if (response.data.seat_number) {
      setRecommendedSeat(response.data.seat_number);
      setShowMoodModal(false);
      if (window.innerWidth < 768) {
        document.querySelector(`.seat[data-seat="${response.data.seat_number}"]`)?.scrol
          behavior: 'smooth',
          block: 'center'
        });
      }
```

6. Feature Advantages and Benefits

The AI seat recommendation feature adds significant value to the JCU Seat Reservation System:

6.1 Improved User Experience

- **Faster Seat Selection**: Users no longer need to manually check booked seats or guess preferences—AI provides a personalized suggestion in seconds.
- **Mood-Aligned Recommendations**: The model learns that certain moods correlate with specific seats (e.g., "Focused" users often choose seats in library-1), ensuring recommendations feel relevant.

6.2 Robust and Scalable

- **Data-Driven Reliability**: Trained on 20,000 diverse records, the model handles varied rooms (classrooms, libraries) and time slots (early morning to late night).
- **Easy Updates**: New data can be added to retrain the model, and saved assets (encoders, weights) can be replaced without rebuilding the entire system.

6.3 Compatibility with Existing Workflows

- **Role Neutral**: Both students and instructors can use the feature—no changes to role-specific permissions.
- **Non-Intrusive**: The AI feature is optional; users can still manually select seats if preferred.

7. Conclusion

The AI-powered seat recommendation feature enhances the JCU Seat Reservation System by combining data-driven machine learning with user-centric design. From simulated data collection to full frontend-backend integration, the workflow ensures the feature is reliable, scalable, and easy to use. Future improvements could include:

- Adding real user data (with privacy consent) to refine recommendations.

- Incorporating additional features (e.g., seat amenities like power outlets or window views).
- Supporting more moods or room types (e.g., lecture halls, study booths).

This feature demonstrates how AI can simplify routine tasks (like seat booking) and improve the overall user experience for JCU's academic community.