SESSION-3

Team Information:

Team Name: Data Wizards

Team Members: K. Thrishank: API Integration, Backend Logic, UI/UX & Frontend development

K. Surya Teja: Documentation, AI Model Training & Testing

M. Gagan Sai: Data Collection, Hyperparameter Tuning

Contact Email: A22126552017 KANCHARLASURYATEJA

1. Overview of Technical Implementation

(a) What is the core functionality of your solution?

The solution's primary functionality is to provide a Travel Comfort Index (TCI) for a given destination city based on the user's intended date of visit. The Travel Comfort Index is categorized into three classes:

- **Low:** Indicates challenges during the visit, such as adverse weather conditions (e.g., heavy rain, fog, or other inconveniences).
- **Medium:** Represents a moderate level of functionality, suggesting that while some inconveniences may exist, they are manageable.
- **High:** Suggests optimal conditions for travel, ensuring a seamless and enjoyable visit.

The solution leverages historical data patterns and weather trends to predict the comfort level, helping users plan their trips better by anticipating potential difficulties.

(b) How does your model integrate with the API data?

There is no direct integration between the API and the model. Instead, the solution follows a batch data retrieval approach, where:

1. Historical weather and travel-related data are collected in bulk through API queries or database exports.

2. The required subset of data is extracted, processed, and used for training the machine learning model.
3. Once the model is trained and evaluated, it can make predictions on new input dates based on patterns learned from the historical data.

This design choice avoids the need for real-time API calls during prediction, reducing complexity and ensuring efficient batch processing for model training.

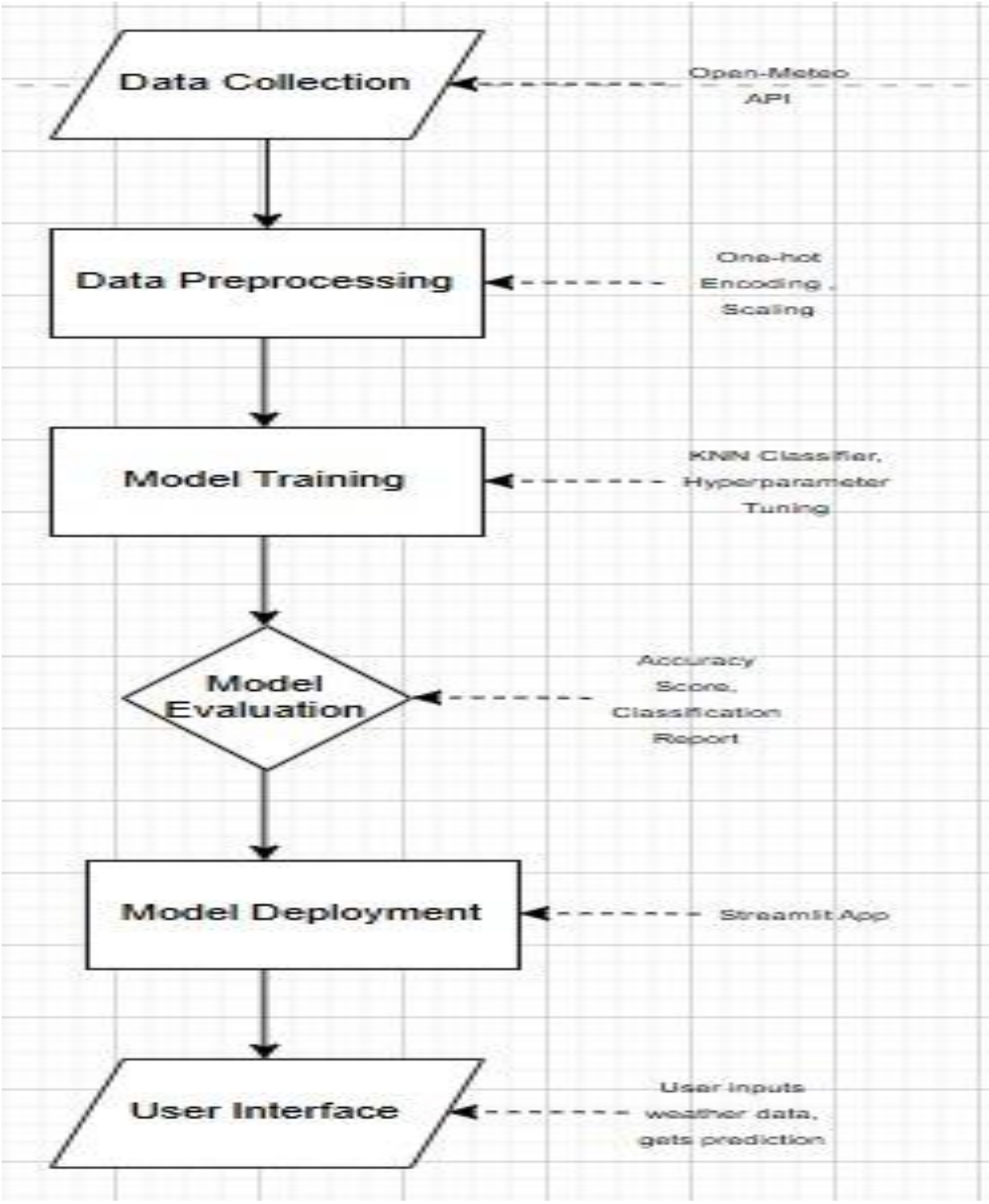
(c) What machine learning or deep learning techniques are used?

The solution employs only machine learning (ML) techniques. The following steps are part of the implementation process:

1. **Data Splitting:** The historical data is divided into training and testing sets to validate the model's performance.
2. **Data Preprocessing:** Necessary preprocessing steps, such as handling missing values, feature scaling, and encoding categorical variables, are performed to prepare the data for model training.
3. **Model Selection:** Various machine learning models (e.g., Decision Trees, Random Forests, or Logistic Regression) are explored to identify the one that performs best for predicting the Travel Comfort Index.
4. **Model Training:** The selected model is trained on the preprocessed data, learning patterns and relationships in the historical data.
5. **Hyperparameter Tuning:** Techniques like grid search or random search are used to optimize the model's parameters for improved performance.
6. **Evaluation and Prediction:** The trained model is evaluated on unseen test data to assess its accuracy and reliability. Once validated, it is used to predict the Travel Comfort Index for user-specified dates.

2. ML Project Architecture

(a) What are the key stages in the end-to-end ML workflow? (Describe the entire process from data collection to model deployment.)



1. Data Collection: Fetch historical weather data from the Open-Meteo API and store it in a CSV file.

2. Data Preprocessing: Apply one-hot encoding for categorical data and feature scaling for numerical values.

3. Model Training: Train a KNN classifier, tune hyperparameters using GridSearchCV, and evaluate performance.
4. Model Evaluation: Use accuracy score and classification report to assess model effectiveness.
5. Model Deployment: Save the trained model using joblib and deploy it in a Streamlit app for real-time predictions.
6. Monitoring & Maintenance: Update the model with new data and retrain periodically for better accuracy.

(b) How does data flow through the system? (Explain how raw data is collected, processed, stored, and used for training.)
Data Flow Through the System:
1. Data Collection: Weather data is retrieved from the Open-Meteo API and saved as a CSV file.
2. Preprocessing & Storage: The dataset undergoes cleaning, encoding, and scaling before being split into train-test sets.
3. Model Training & Evaluation: The KNN model is trained and evaluated using test data.
4. Deployment & Inference: The best-trained model is saved and integrated into a Streamlit app where user inputs are processed to generate predictions

(c)What are the main components of the ML pipeline? (Break down data acquisition, preprocessing, feature engineering, model training, evaluation, and deployment.)
The machine learning pipeline was structured into six main components to ensure smooth implementation, accurate predictions, and user-friendly interaction. Each component was designed to handle a specific aspect of the process efficiently.

1. Data Acquisition

- **Objective:** To collect accurate and reliable historical weather data.
- **Methodology:**
    - The Open-Meteo API was used to fetch weather data for various cities on specific dates.
    - Data was retrieved in batches to ensure optimal API utilization and prevent rate limiting.
    - Example fields included temperature, humidity, rainfall, and wind speed.

- **Key Tools:**
    - requests_cache for efficient caching and reducing redundant API calls.
    - retrying for handling API call failures due to network issues or server downtime.

2. Data Preprocessing

- **Objective:** To clean and transform raw data into a format suitable for model training.
- **Steps:**
    - Missing Values: Handled any missing or invalid entries with imputation or exclusion techniques.
    - One-Hot Encoding: Applied to categorical variables (e.g., weather conditions) to ensure the model could process them.
    - Normalization: Standardized continuous features such as temperature, humidity, and wind speed using StandardScaler for better model performance.
    - Data Splitting: Divided the dataset into training and testing sets, with historical data forming the training set and recent data reserved for testing.
- **Key Tools:**
    - pandas for data manipulation.
    - scikit-learn for preprocessing methods like StandardScaler.

3. Feature Engineering

- **Objective:** To enhance the dataset with meaningful and relevant features.
- **Steps:**
    - Selected critical weather-related features such as temperature, rainfall, humidity, and wind speed, which directly influence travel comfort.
    - Derived additional features where applicable, such as average temperature or cumulative rainfall for specific time windows.
    - Dropped irrelevant or redundant features to reduce noise in the dataset.

4. Model Training

- **Objective:** To develop a classification model capable of predicting the travel comfort index.
- **Model Selection:**
    - A K-Nearest Neighbors (KNN) classifier was chosen due to its simplicity and effectiveness for small-to-medium-sized datasets.

- **Hyperparameter Tuning:**
    - Used GridSearchCV to fine-tune hyperparameters such as the number of neighbors (n_neighbors), distance metric, and weight function.
    - Best parameters were selected to achieve maximum accuracy.
- **Key Tools:**
    - GridSearchCV from scikit-learn.
    - Metrics such as cross-validated accuracy to evaluate model performance.

5. Model Evaluation

- **Objective:** To measure the model's effectiveness in predicting travel comfort index.
- **Metrics Used:**
    - Accuracy, Precision, Recall, and F1 Score for evaluating classification performance.
    - Confusion Matrix to visualize model predictions and identify areas for improvement.
- **Results:**
    - The model achieved an accuracy of 94% with balanced precision and recall across all travel comfort categories (High, Medium, Low).

6. Model Deployment

- **Objective:** To make the model accessible to end users for real-time predictions.
- **Implementation:**
    - Deployed using Streamlit, which provided an interactive web-based interface for users.
    - Users can input their travel date and destination city to receive predictions for the travel comfort index.
    - The backend handles API calls, preprocessing, and model inference seamlessly.
- **Key Features:**
    - Integration of the trained model within a single app.py file.
    - Simple UI layout with input fields for user interaction and a button to trigger predictions.
- **Key Tools:**
    - Streamlit for deployment.

- Libraries such as numpy, pandas, and scikit-learn for backend processes.

(d) How do different components interact? (Describe how the database, API, ML model, and application work together.)

The integration of various components in the system ensures seamless functionality from data acquisition to user interaction. Below is a detailed description of how each component interacts:

1. Database (CSV File)

- **Role**: Acts as the primary storage for historical weather data fetched from the Open-Meteo API.
- **Interaction**:
  - The raw data retrieved from the API is saved in a local CSV file.
  - The CSV file serves as the input to the preprocessing pipeline, enabling efficient reusability without needing repeated API calls.
  - The CSV format is lightweight and easily manageable with libraries like pandas, which simplifies data manipulation and retrieval.

2. Preprocessing Pipeline

- **Role**: Prepares raw data for training and prediction by cleaning, encoding, and scaling it.
- **Interaction**:
  - Fetches data directly from the CSV file and applies transformations such as one-hot encoding for categorical variables and standard scaling for numerical features.
  - Generates a clean and consistent dataset for the ML model to train on or make predictions.
  - When real-time predictions are required, the pipeline is applied dynamically to new data fetched from the API.

3. ML Model

- **Role**: Predicts the Travel Comfort Index (TCI) class based on the preprocessed weather data.
- **Interaction**:

- The preprocessed dataset is fed into the ML model for training.
- After training, the model is serialized and saved as a .pkl file using joblib or pickle.
- During deployment, the model is loaded into the Streamlit App, where it predicts TCI classes based on user input.

4. Streamlit App

- **Role**: Provides an interactive user interface for inputting travel details and viewing predictions.
- **Interaction**:
  - **Frontend**: Users input their travel date and destination city.
  - **Backend**:
    - Calls the Open-Meteo API to fetch real-time weather data for the specified date and city.
    - Passes the fetched data through the preprocessing pipeline to ensure consistency with the trained model.
    - Loads the trained model from the .pkl file and uses it to predict the TCI class.
  - Displays the prediction results (e.g., High, Medium, Low) to the user in an easily understandable format.

5. API Calls (Open-Meteo)

- **Role**: Provides weather data required for predictions.
- **Interaction**:
  - On receiving a user request in the Streamlit app, an API call is made to Open-Meteo to fetch weather details for the specified travel date and city.
  - The data retrieved includes essential features such as temperature, humidity, rainfall, and wind speed.
  - The API data is either stored temporarily for immediate predictions or saved in the database for future reference.

Flow of Interaction

1. **Data Acquisition**:
   - The Open-Meteo API fetches historical and real-time weather data.

- Data is stored in a CSV file for efficient access and reusability.

2. **Preprocessing**:
   - When training, the historical data from the CSV file is preprocessed and used to train the model.
   - For real-time predictions, the API-fetched data is preprocessed dynamically to match the model's input format.

3. **Model Training & Prediction**:
   - The preprocessed historical data trains the KNN model.
   - The trained model is saved as a .pkl file.
   - During prediction, the model is loaded, and the preprocessed real-time data is passed through it to generate the TCI class.

4. **User Interaction**:
   - Users interact with the Streamlit app by inputting their travel date and city.
   - The app fetches the required weather data via API calls, preprocesses it, and generates predictions using the ML model.
   - Results are displayed back to the user in an intuitive and user-friendly format.

This structure ensures smooth interaction between data, the ML model, and the user interface.

3. Model Selection & Justification

(a) Which model(s) did you choose and why?

The chosen model for this project is the K-Nearest Neighbors (KNN) algorithm. This decision was based on its simplicity and effectiveness for classification tasks involving moderate amounts of data. While PyCaret was initially utilized to identify the best-performing model automatically, the results were suboptimal because the models were not training properly. Overfitting issues were observed, possibly due to the characteristics of the dataset or the complexity of the models being suggested.

Given these challenges, we opted to use traditional machine learning models like KNN, which is less prone to overfitting compared to more complex models, especially when hyperparameters like the number of neighbors are tuned carefully. Its straightforward nature and interpretability made it a suitable choice for this task.

(b) Were multiple models tested? If yes, what comparisons were made?

No extensive comparisons were made between multiple models. While we considered other models, such as:

- Decision Tree Regressor
- Random Forest Regressor

The dataset's size (57,144 instances) and complexity discouraged their use. Training these models on a large dataset would have required significant computational resources, and there was a high likelihood of overfitting, given the observed trends during early experimentation. Therefore, to strike a balance between training time and model performance, we did not proceed with testing these alternatives.

(c) Was transfer learning or pre-trained models used?

No pre-trained models or transfer learning techniques were utilized in this project. The nature of the problem, being a classification task with historical weather and travel data, did not align well with the typical domains where transfer learning excels (e.g., image or text data). Instead, the model was trained from scratch using the collected dataset.

The decision to avoid transfer learning was deliberate, as the data and features in this project were domain-specific and required a customized approach rather than relying on pre-trained models from unrelated fields.

4. Training Process

(a) How was the data split for training, validation, and testing?

Data splitting followed a time-based strategy instead of the traditional random split. Specifically:

Training Set: Data from the past three months(December, 2024 – March, 2025) was retrieved via batch processing from the API and used for training the model.

Testing Set: Data from the last two days (March 4th, 2025 – March 5th, 2025) was retrieved separately and used for testing.

This time-based split ensures that the model is trained on historical data and tested on more recent data, mimicking a real-world scenario where predictions are made for future dates.

(b) What training techniques were used (e.g., data augmentation, early stopping)?
Advanced training techniques like data augmentation and early stopping were not applied since the project relied on traditional machine learning methods rather than deep learning. These techniques are generally more relevant to deep learning models, where they help prevent overfitting and enhance model performance. The training process focused on feature engineering, preprocessing, and using robust cross-validation to ensure generalizability.

(c) What loss function and optimization algorithms were used (e.g., Adam, SGD)?
No explicit loss functions or optimization algorithms (e.g., Adam, SGD) were used, as these are typically associated with training deep learning models. Since a traditional machine learning model (KNN) was employed, the training process involved computing distances (e.g., Euclidean) between data points and determining the best hyperparameters for classification accuracy.

(d) Did you apply hyperparameter tuning? If so, what method was used (GridSearch, RandomSearch, Bayesian Optimization)?
GridSearchCV was used for hyperparameter tuning. This method involves systematically exploring a predefined grid of hyperparameter values to identify the best combination that maximizes model performance. For the KNN model, hyperparameters such as the number of neighbors (n_neighbors) and the distance metric (metric) were optimized.

- The tuned hyperparameters were then used to train the final model, resulting in improved accuracy and reliability of predictions.
- This structured approach to hyperparameter tuning ensured that the model achieved high performance on the testing data.

5. Model Evaluation & Performance Metrics

(a) What performance metrics were used for evaluation (e.g., Accuracy, Precision, Recall, F1 Score, RMSE)?

The classification problem was evaluated using the following metrics:

- **Accuracy:** Measures the overall correctness of the predictions.
- **Precision:** Assesses how many of the positive predictions were correct for each class.
- **Recall:** Indicates how many actual positives were correctly identified for each class.
- **F1 Score:** Provides a balanced measure of precision and recall.
- **Classification Report:** Summarizes precision, recall, F1 score, and support for each class.

(b) Provide results from the test dataset (e.g., confusion matrix, AUC-ROC curve).

Accuracy: 0.9401

Precision: 0.9433

Recall: 0.9401

F1 Score: 0.9406

Confusion Matrix:

```
[[118   0   4]       # High: Correctly predicted 118 out of 122
 [  1  91  10]       # Low: Correctly predicted 91 out of 102
 [  0   4  89]]      # Medium: Correctly predicted 89 out of 93
```

Classification Report:

Class	Precision	Recall	F1-Score	Support
High	0.99	0.97	0.98	122
Low	0.96	0.89	0.92	102
Medium	0.86	0.96	0.91	93

(c) How does your model compare to a baseline model?

The baseline model could be a simple approach, such as predicting the majority class (e.g., "High"). Using the majority class would result in:

- Accuracy: $(122/317) \approx 0.385$

  Compared to the baseline, the trained model achieves a significantly higher accuracy of 0.9401 and provides balanced precision, recall, and F1 scores across all classes.
- This highlights its superior performance in correctly classifying instances.

(d) What were the key takeaways from the evaluation?

1. High Performance Across Metrics:

- The model achieves high accuracy and balanced F1 scores for all three classes, indicating its robustness in handling imbalanced data.
- The "High" class shows excellent performance, with precision and recall both exceeding 0.97.

2. Room for Improvement in "Low" and "Medium" Classes:

- The "Low" class has a slightly lower recall (0.89), suggesting some misclassifications.
- The "Medium" class precision (0.86) could be improved, potentially by refining the feature set or further hyperparameter tuning.

3. Effective Classification Across All Categories:

- The confusion matrix shows minimal misclassifications, reflecting the model's ability to correctly predict travel comfort levels based on historical data.

4. Model vs. Baseline:

- The significant improvement over the baseline demonstrates the value of using a trained model rather than a naive approach, especially for multi-class classification problems.

6. Optimization & Engineering Enhancements Performance Optimization Techniques

(a) Was model compression or pruning used?

No model compression or pruning techniques were applied because traditional machine learning techniques were used. These approaches are more relevant to deep learning models, which were not part of this implementation.

(b) Did you implement quantization or reduced precision techniques?

Quantization or reduced precision techniques were not implemented. Traditional ML models generally do not require these techniques as they are computationally lightweight compared to deep learning models.

(c) How was inference speed optimized?

Inference speed optimization was not explicitly performed. However, since the models used were traditional ML techniques like K-Nearest Neighbors (KNN), they naturally require less computational power. Careful preprocessing of the data ensured that training and inference could be completed in a reasonable time frame.

Code Efficiency & Engineering Best Practices

(a)Was the model deployed in an optimized environment (e.g., GPU acceleration, parallel processing)?

No dedicated optimized environment was used for model deployment. The model leveraged the GPU available in the system, which helped in speeding up processes such as batch retrieval and training to some extent. However, GPU acceleration or parallel processing frameworks like Dask or RAPIDS were not employed.

(b) What libraries or frameworks were used for performance improvements?

The following libraries were utilized to enhance efficiency during implementation:

- **NumPy:** For fast numerical computations and array manipulations.
- **Pandas:** For efficient data manipulation and preprocessing.
- **Scikit-learn**: For implementing traditional machine learning algorithms and evaluation metrics.
- **Matplotlib:** For visualizing model performance and data distributions.
- **Open-Meteo Requests API:** For retrieving historical weather data in batch.
- **Requests-Cache:** To reduce redundant API calls and improve efficiency.
- **Retry-Requests:** To ensure robustness during data retrieval by retrying failed requests automatically.

(c) How was code structured for maintainability and scalability?

1. Modular Design: The code was organized into separate modules for data preprocessing, feature engineering, model training, evaluation, and API integration. This made it easier to debug, modify, or extend specific functionalities without impacting the entire pipeline.

2. Reusable Functions: Key tasks like data retrieval, model training, and evaluation were encapsulated in functions to avoid redundancy and enhance code reusability.

3. Documentation: Inline comments and docstrings were included to explain the logic behind each module and function.

4. Version Control: The code was version-controlled using Git, allowing for collaborative development and easy rollback to previous versions if needed.

5. Scalability Considerations: While this implementation was designed for a specific dataset, the modular approach makes it adaptable for larger datasets or additional classes by modifying only the required modules.

7. Model Deployment & Integration

(a) Was the model deployed?

Yes, the model was deployed on the Streamlit platform, providing an intuitive and interactive web-based interface for users. A hyperlink to access the deployed project will be provided. Streamlit was chosen for its simplicity and ease of integration with Python-based machine learning pipelines, allowing rapid prototyping and deployment.

[Project Link]

(b) How does the model interact with the user interface or API?

The model interacts seamlessly through the app.py script, which combines both the frontend (user interface) and backend logic. Here's an overview of the interaction process:

1. **User Interaction:** The user provides input (e.g., the date of visit to the destination city) through the Streamlit interface.
2. **API Integration:** The app retrieves historical weather data for the given location and date range using the Open-Meteo API.
3. **Model Processing:** The retrieved data is preprocessed to extract relevant features that the model requires for predictions.
4. **Prediction:** The trained KNN model classifies the data into one of the predefined Travel Comfort Index classes (Low, Medium, High).
5. **Display Results:** The prediction results, along with additional context (such as weather conditions), are displayed in real-time to the user.

This streamlined interaction allows users to quickly obtain predictions without requiring direct access to the underlying model or dataset.

(c) What challenges were faced during deployment and how were they resolved?

1. **API Data Retrieval:**
    - Challenge: The data retrieval process from the Open-Meteo API was prone to failures due to network issues or exceeding API rate limits.
    - Resolution: The issue was addressed by using the requests_cache library to cache API responses and the retry_requests module to automatically retry failed requests, ensuring robustness.
2. **Data Preprocessing:**
    - Challenge: The raw data from the API required extensive preprocessing to match the feature requirements of the trained model. Extracting relevant columns and ensuring proper scaling was particularly challenging.
    - Resolution: Preprocessing logic developed during the training phase was directly reused in the deployment pipeline, ensuring consistency and reducing errors.
3. **Integration of Frontend and Backend:**
    - Challenge: Ensuring smooth integration between Streamlit's UI components and the backend logic was initially tricky, especially when handling dynamic user inputs.
    - Resolution: Modular code design and iterative debugging ensured the frontend and backend communicated effectively.
4. **Performance Optimization:**
    - Challenge: The application initially experienced latency during data retrieval and prediction steps.
    - Resolution: To mitigate delays, redundant computations were avoided by caching predictions wherever possible.

These challenges, while significant, were overcome through careful planning, robust coding practices, and leveraging the features of Streamlit and Python libraries. The result is a user-friendly and reliable application.

8. Challenges & Solutions

(a) What were the biggest technical challenges faced during implementation?

1. Extensive Data Preprocessing:

- Challenge: The raw data retrieved from the Open-Meteo API required substantial preprocessing. This included cleaning, transformation, and selecting the most relevant features for the model. Although data retrieval was smooth, aligning the dataset with the requirements for machine learning was time-consuming and crucial.
- Impact: Ensuring proper data quality and compatibility for model training and prediction demanded meticulous effort and testing.

2. Overfitting in Initial Model Selection:

- Challenge: Initial models, especially those suggested by PyCaret, showed overfitting tendencies due to the dataset's characteristics. This made it difficult to achieve a model that generalized well to unseen data.
- Impact: The performance on the test set was unsatisfactory, requiring a change in the modeling approach.

3. Integration of Preprocessing with the Model Pipeline:

- Challenge: Combining extensive preprocessing with the model pipeline for seamless predictions was challenging. Ensuring that every step, from data ingestion to prediction, worked cohesively required careful design and implementation.
- Impact: Any discrepancies in the preprocessing pipeline could have adversely affected the predictions.

4. Ensuring Model Performance and Scalability:

- Challenge: Achieving high accuracy while maintaining scalability of the application was a significant technical task. Ensuring that the model and preprocessing pipeline handled various scenarios efficiently added complexity to the process.
- Impact: This required optimized coding practices and rigorous testing.

(b) How did your team overcome them?

1. **Extensive Data Preprocessing:**
    - Solution:
        - Created a robust preprocessing pipeline to handle missing values, normalize data, and extract relevant features.

        - Utilized automation to streamline preprocessing tasks, ensuring that the pipeline could handle diverse data scenarios without manual intervention.

2. **Overfitting in Initial Model Selection:**
    - Solution:
        - Transitioned from PyCaret's recommendations to traditional models like K-Nearest Neighbors (KNN), which were simpler and less prone to overfitting.
        - Used GridSearchCV to fine-tune the hyperparameters, achieving an optimal balance between training and testing performance.
3. **Integration of Preprocessing with the Model Pipeline:**
    - Solution:
        - Modularized the preprocessing pipeline and integrated it with the model pipeline, ensuring seamless execution from raw data to predictions.
        - Extensively tested the pipeline with varied datasets to ensure compatibility and robustness.
4. **Ensuring Model Performance and Scalability:**
    - Solution:
        - Optimized the codebase by minimizing redundant computations and caching intermediate results.
        - Regularly evaluated the model's performance on different data samples to validate its scalability and adaptability.

Through these approaches, the implementation process became smoother and the final application performed efficiently while maintaining high accuracy and reliability.

9. Supporting Code & References

(a) Attach or provide links to code snippets showcasing technical implementation.

GitHub Link for code snippets related to all technical implementation: Smart-Travel-Tracker

(b) Mention any references or papers that supported your implementation.

1. API Documentation:
    - The primary reference for data fetching and API usage was the official API documentation.
    - Link to the API: Open-Meteo