Ceylon-Travelgine\ Tourist Demand Forecasting Engine

1. Import Libraries and Data Loading

This section imports all the necessary Python libraries for data manipulation, visualization, and machine learning, and then loads the main dataset into a pandas DataFrame.

```
# Core libraries for data manipulation
import pandas as pd
import numpy as np

# Libraries for data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Machine learning libraries from scikit-learn
from sklearn.model_selection import train_test_split, GridSearchCV, TimeSeriesSplit
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Advanced modeling library
import xgboost as xgb

# Library for forecasting total arrivals
from prophet import Prophet

# Library for saving the model
import joblib
import json

# Load your master CSV file into a pandas DataFrame
df = pd.read_csv('/content/drive/MyDrive/data backup/tourguide meta data/tourist_demand_forcasting_master.csv')

print("Dataset loaded successfully. Shape:", df.shape)
display(df.head())
```

```
Dataset loaded successfully. Shape: (4800, 19)
```

	Year	Month	District	Tourist_Nights	Tripadvisor_Reviews	GoogleTrends_Index	Avg_Temperature_C	Rainfall_mm	Sunshine_Hours	S
0	2010.0	january	Colombo	150878	754	45	27.0	41.0	271.0	Nor Mo
1	2010.0	january	Gampaha	30176	151	35	27.0	58.0	265.0	Nor Mo
2	2010.0	january	Kalutara	42246	211	38	27.0	60.0	260.0	Nor Mo
3	2010.0	january	Kandy	72421	362	42	23.0	88.0	192.0	Nor Mo
4	2010.0	january	Matale	48281	241	37	24.0	140.0	185.0	Nor Mo

2. Data Cleaning & Initial Checks

This section performs initial data cleaning steps, including handling missing values, converting data types for arrival columns that were loaded as objects due to commas, and dropping rows with missing essential information.

```
print("--- Initial Data Info ---")
df.info()

# The arrival columns are loaded as 'object' type because of commas.
# We need to convert them to numeric types.
cols_to_clean = ['Total_Arrivals', 'India', 'UK', 'China', 'Germany', 'Russia', 'Other']
```

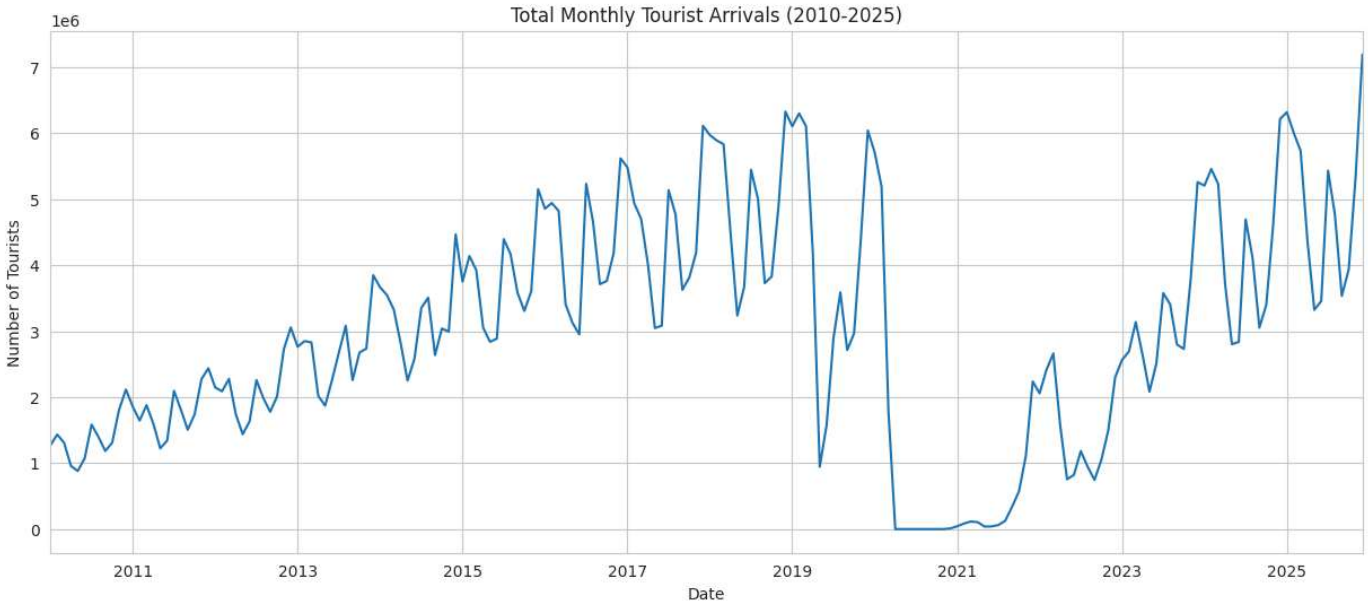
```
for col in cols_to_clean:
    if df[col].dtype == 'object':
        # Handle missing values in the column before attempting to convert to integer
        df[col] = df[col].str.replace(',', '')
        df[col] = pd.to_numeric(df[col], errors='coerce')

# Check for any missing values (NaNs)
print("\n--- Missing Value Check ---")
print(df.isnull().sum())

# Address remaining missing values (if any) - e.g., drop rows with NaNs in key columns
# Based on previous checks, 'Year', 'Avg_Temperature_C', etc. had a few NaNs.
# Dropping rows with missing values in 'Year' or 'Month' before creating the Date column is crucial.
df = df.dropna(subset=['Year', 'Month']).copy() # Ensure Year and Month are not NaN for Date creation

print("\nMissing value check and handling complete.")
print("Updated Dataset Shape:", df.shape)
```

```
--- Initial Data Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4800 entries, 0 to 4799
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Year                 4798 non-null   float64
 1   Month                4800 non-null   object
 2   District             4800 non-null   object
 3   Tourist_Nights       4800 non-null   int64
 4   Tripadvisor_Reviews  4800 non-null   int64
 5   GoogleTrends_Index   4800 non-null   int64
 6   Avg_Temperature_C    4798 non-null   float64
 7   Rainfall_mm          4798 non-null   float64
 8   Sunshine_Hours       4798 non-null   float64
 9   Season               4798 non-null   object
 10  Event_Impact         4800 non-null   object
 11  Event_Count          4800 non-null   int64
 12  Total_Arrivals       4798 non-null   object
 13  India                4798 non-null   object
 14  UK                   4798 non-null   object
 15  China                4798 non-null   object
 16  Germany              4798 non-null   object
 17  Russia               4798 non-null   object
 18  Other                4798 non-null   object
dtypes: float64(4), int64(4), object(11)
memory usage: 712.6+ KB

--- Missing Value Check ---
Year                   2
Month                  0
District               0
Tourist_Nights         0
Tripadvisor_Reviews    0
GoogleTrends_Index     0
Avg_Temperature_C      2
Rainfall_mm            2
Sunshine_Hours         2
Season                 2
Event_Impact           0
Event_Count            0
Total_Arrivals         2
India                  2
UK                     2
China                  2
Germany                2
Russia                 2
Other                  2
dtype: int64

Missing value check and handling complete.
Updated Dataset Shape: (4798, 19)
```

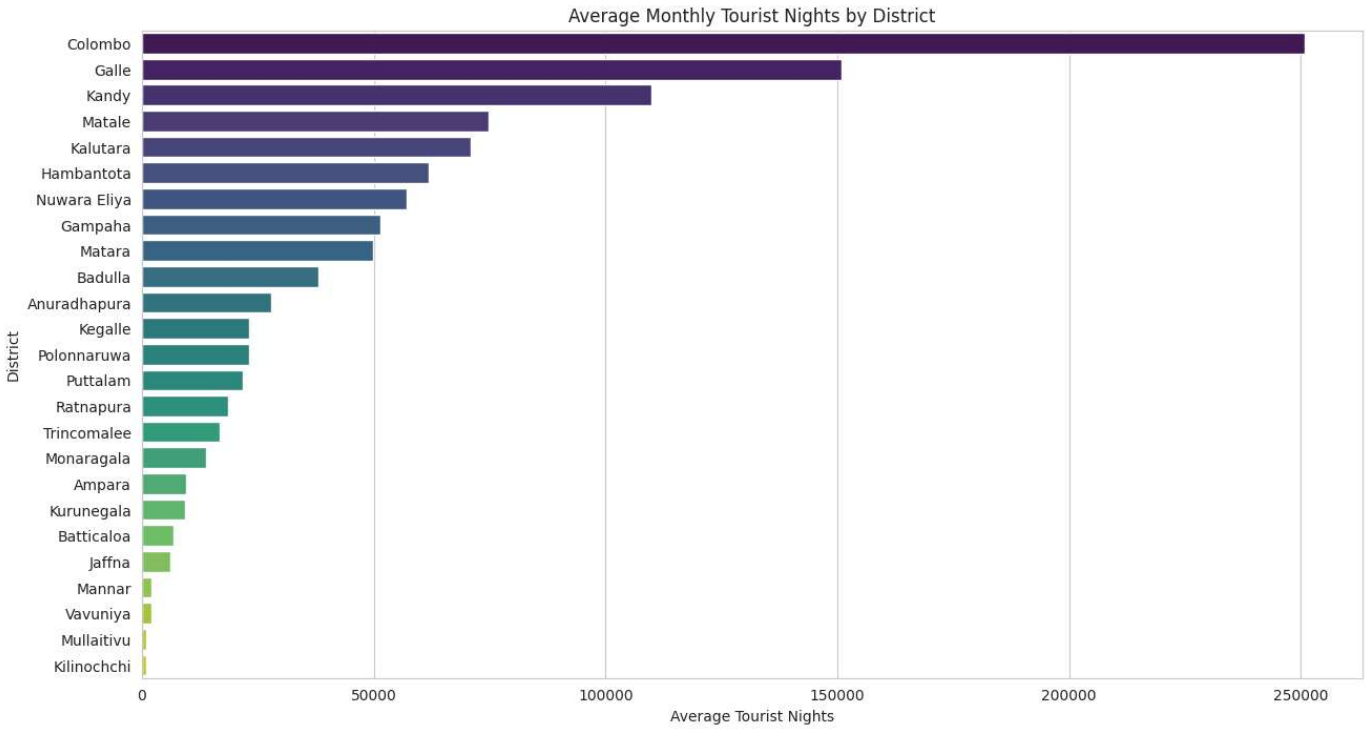
3. Exploratory Data Analysis (EDA)

This section performs exploratory data analysis to understand the data patterns. It includes creating a proper date column and generating various visualizations such as total arrivals over time, average tourist nights by district, and distributions by month and season, as well as a correlation matrix.

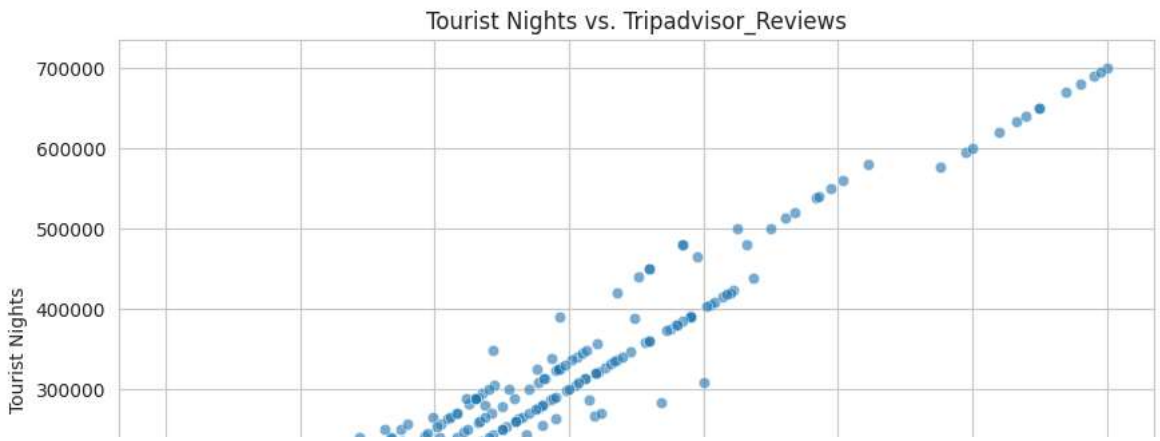
```
# Set plot style
sns.set_style("whitegrid")

# Create a proper date column for plotting and time-based analysis
df['Date'] = pd.to_datetime(df['Year'].astype(int).astype(str) + '-' + df['Month'], format='%Y-%B')

# 1. Plot Total Tourist Arrivals Over Time
plt.figure(figsize=(15, 6))
monthly_arrivals = df.groupby('Date')['Total_Arrivals'].sum() # Sum Total_Arrivals for each month
monthly_arrivals.plot(title='Total Monthly Tourist Arrivals (2010-2025)')
plt.ylabel('Number of Tourists')
plt.show()

# 2. Plot Average Tourist Nights by District
plt.figure(figsize=(15, 8))
avg_nights_by_district = df.groupby('District')['Tourist_Nights'].mean().sort_values(ascending=False)
sns.barplot(x=avg_nights_by_district.values, y=avg_nights_by_district.index, palette='viridis')
plt.title('Average Monthly Tourist Nights by District')
plt.xlabel('Average Tourist Nights')
plt.show()

# --- Visualization: Feature Relationships (Numerical) ---

print("\n--- Visualizing Relationships: Numerical Features vs. Tourist Nights ---")

# Select some key numerical features to visualize
numerical_features_eda = ['Tripadvisor_Reviews', 'GoogleTrends_Index', 'Avg_Temperature_C', 'Rainfall_mm', 'Sunshine_Hours']

# Create scatter plots
for feature in numerical_features_eda:
    plt.figure(figsize=(10, 6))
    sns.scatterplot(data=df, x=feature, y='Tourist_Nights', alpha=0.6)
    plt.title(f'Tourist Nights vs. {feature}')
    plt.xlabel(feature)
    plt.ylabel('Tourist Nights')
    plt.show()


# --- Visualization: Tourist Nights by Categorical Features ---

print("\n--- Visualizing Tourist Nights by Categorical Features ---")

# Box plot for Tourist Nights by Month
plt.figure(figsize=(14, 7))
sns.boxplot(data=df, x='Month', y='Tourist_Nights', palette='viridis')
plt.title('Tourist Nights Distribution by Month')
plt.xlabel('Month')
plt.ylabel('Tourist Nights')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# Box plot for Tourist Nights by Season
plt.figure(figsize=(10, 6))
sns.boxplot(data=df, x='Season', y='Tourist_Nights', palette='viridis')
plt.title('Tourist Nights Distribution by Season')
plt.xlabel('Season')
plt.ylabel('Tourist Nights')
plt.tight_layout()
plt.show()


# 5. Visualize Correlation Matrix (Numerical Features)
print("\n--- Visualizing Correlation Matrix ---")
numerical_cols = df.select_dtypes(include=np.number).columns.tolist()
# Remove Year and Month_Num as they will be recreated in feature engineering
numerical_cols = [col for col in numerical_cols if col not in ['Year', 'Month_Num']]

correlation_matrix = df[numerical_cols].corr()

plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5)
plt.title('Correlation Matrix of Initial Numerical Features')
plt.tight_layout()
plt.show()
```

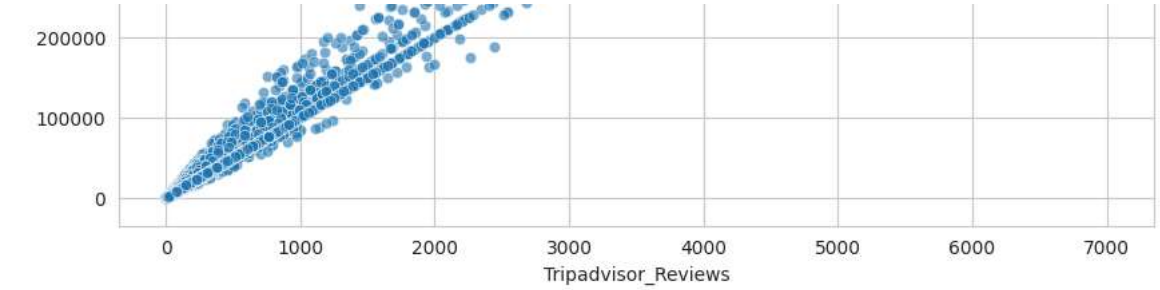


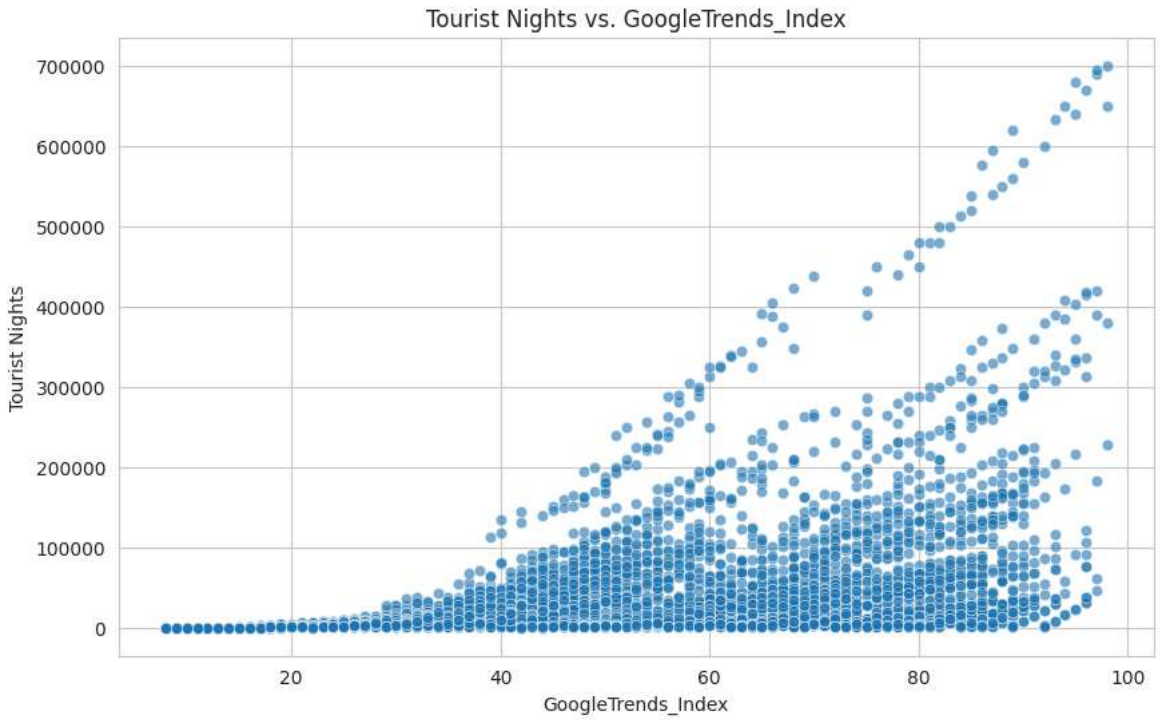

```
/tmp/ipython-input-2710496831.py:17: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `lege

  sns.barplot(x=avg_nights_by_district.values, y=avg_nights_by_district.index, palette='viridis')
```

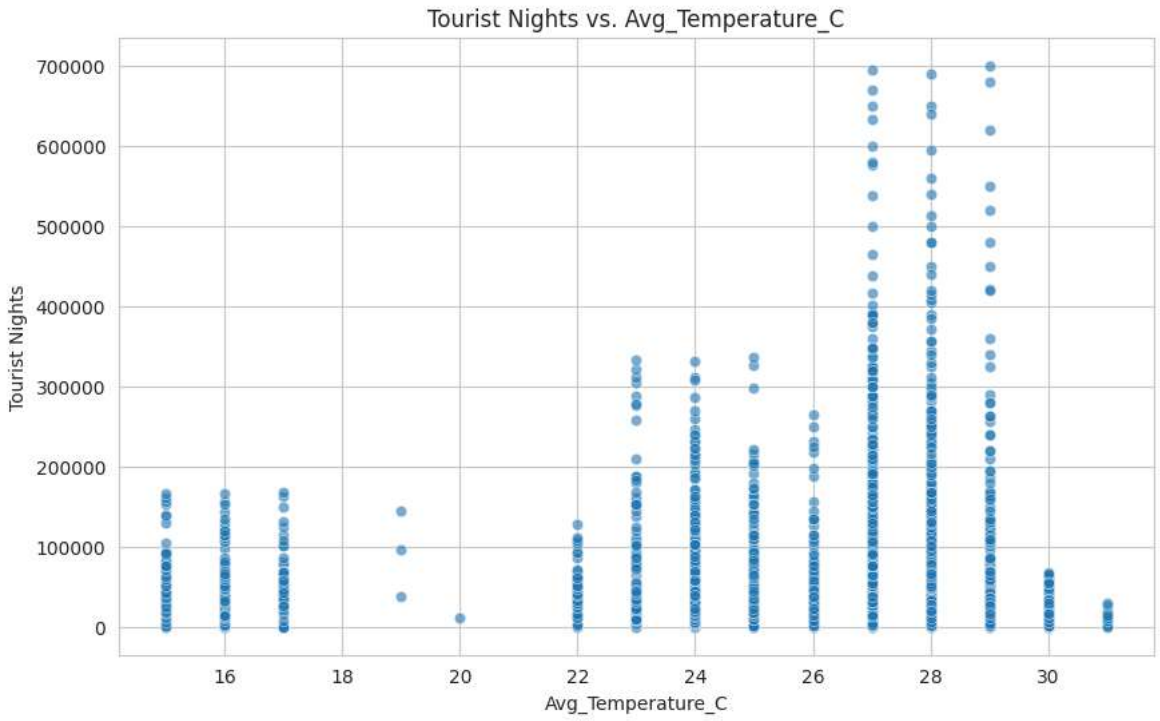



```
--- Visualizing Relationships: Numerical Features vs. Tourist Nights ---
```

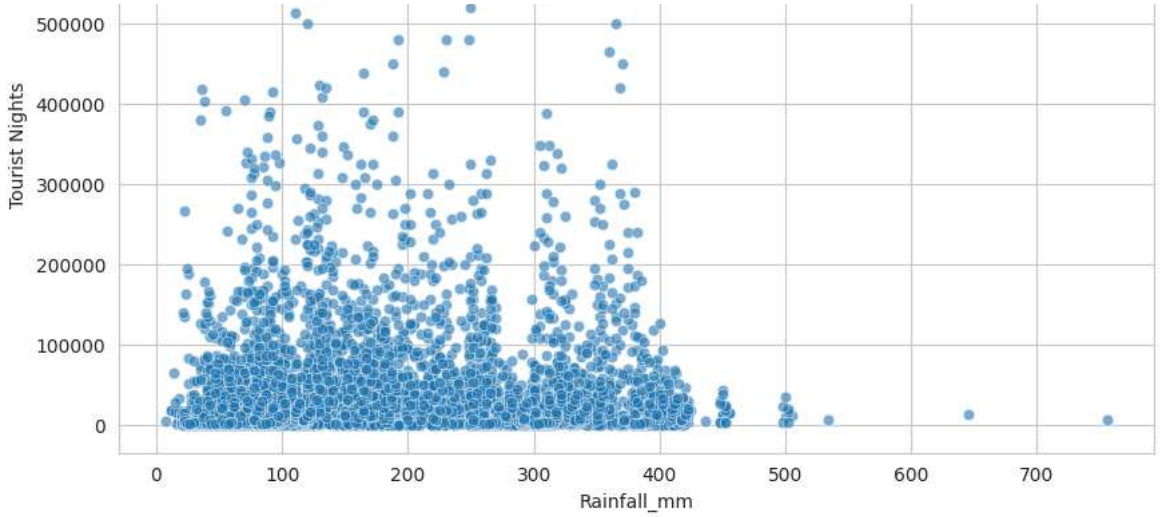


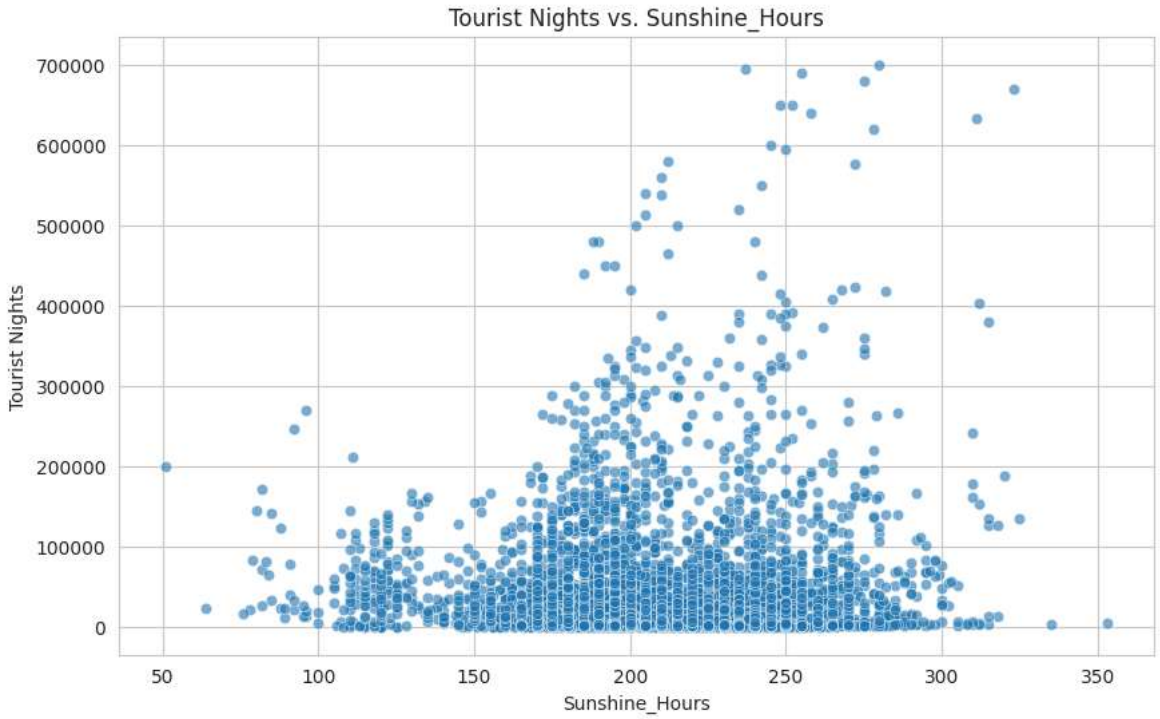

200000
100000
0
0
1000
2000
3000
4000
5000
6000
7000
Tripadvisor_Reviews


Tourist Nights vs. GoogleTrends_Index
Tourist Nights
700000
600000
500000
400000
300000
200000
100000
0
20
40
60
80
100
GoogleTrends_Index

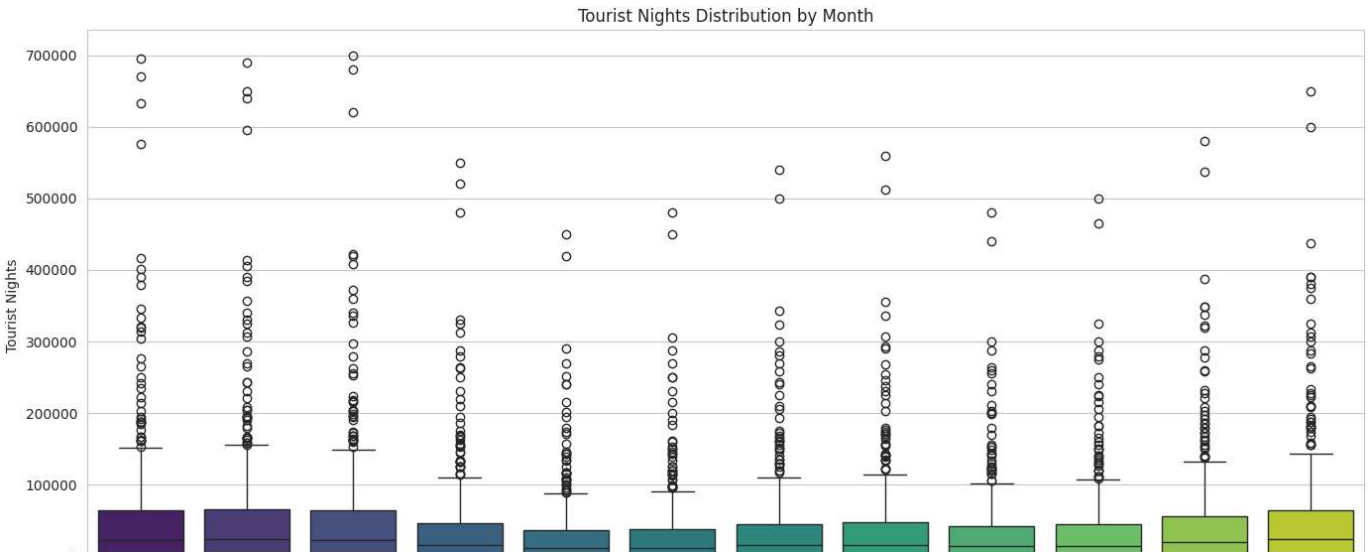

Tourist Nights vs. Avg_Temperature_C
Tourist Nights
700000
600000
500000
400000
300000
200000
100000
0
16
18
20
22
24
26
28
30
Avg_Temperature_C


Tourist Nights vs. Rainfall_mm
700000
600000

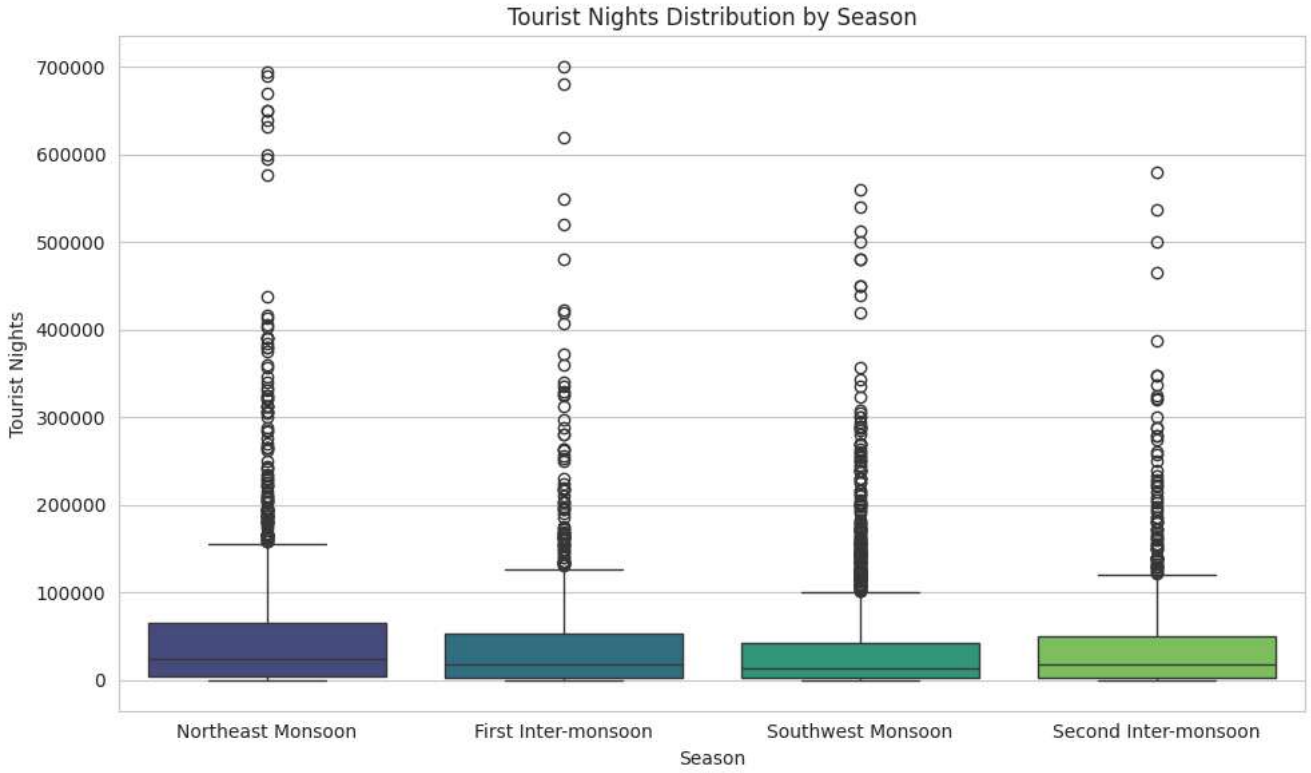




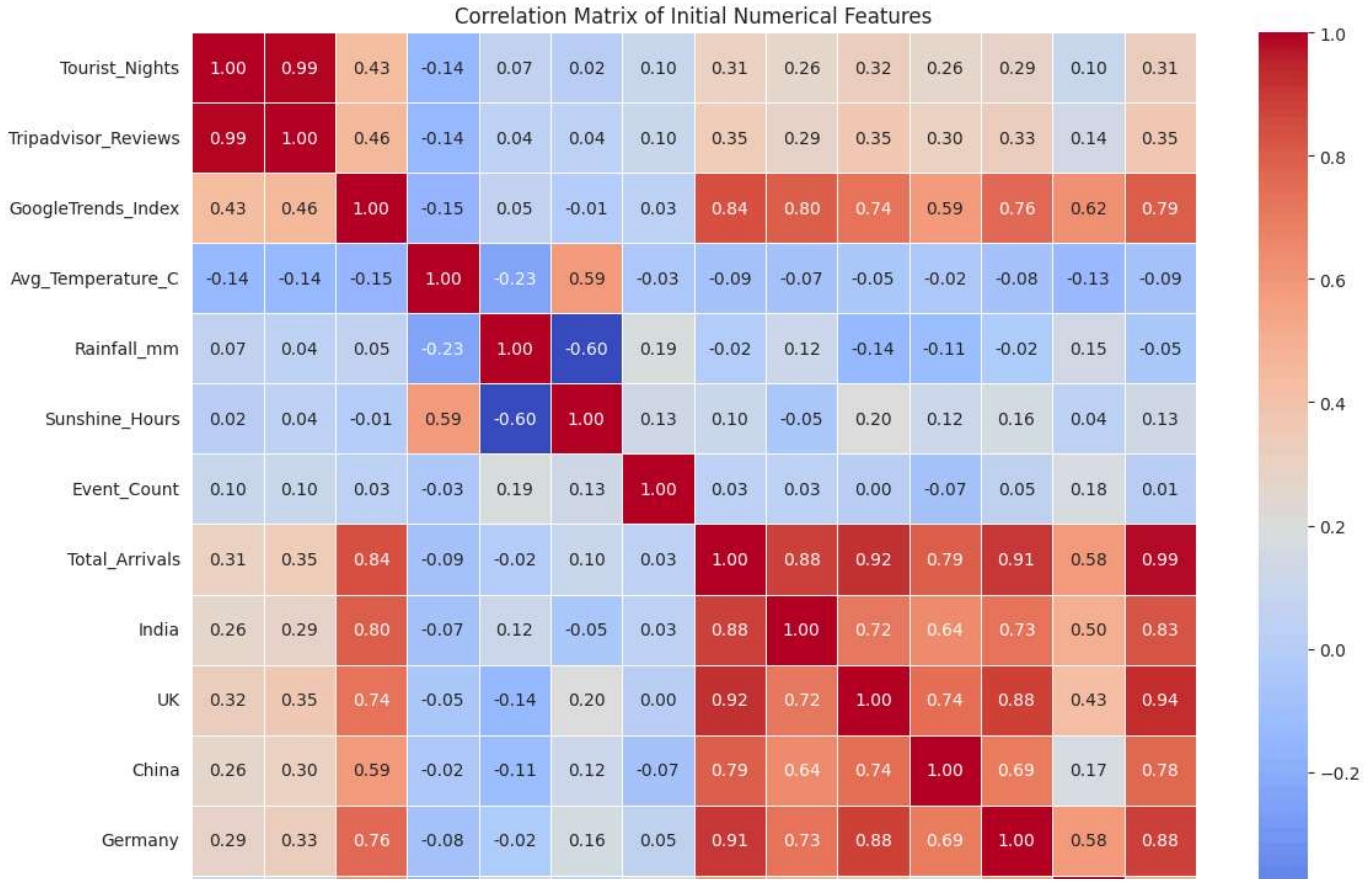

```
--- Visualizing Tourist Nights by Categorical Features ---
/tmp/ipython-input-2710496831.py:45: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `lege

  sns.boxplot(data=df, x='Month', y='Tourist_Nights', palette='viridis')
```

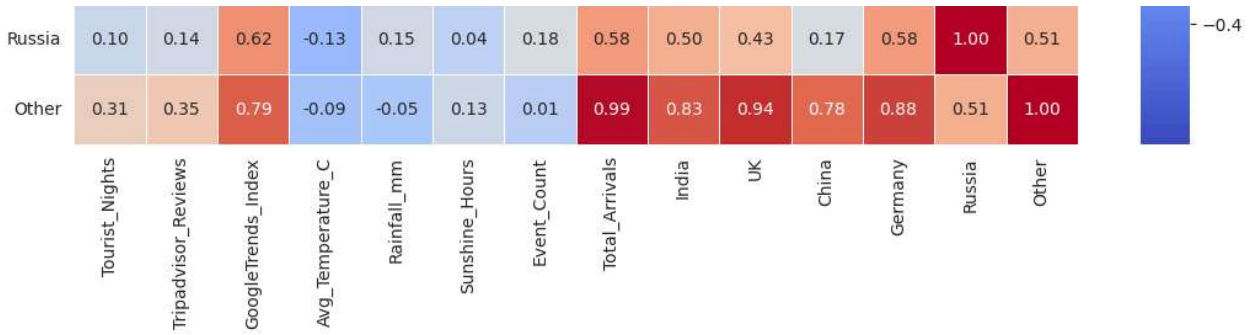



```
/tmp/ipython-input-2710496831.py:55: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `lege

  sns.boxplot(data=df, x='Season', y='Tourist_Nights', palette='viridis')
```




```
--- Visualizing Correlation Matrix ---
```




Russia
Other
0.10 0.14 0.62 -0.13 0.15 0.04 0.18 0.58 0.50 0.43 0.17 0.58 1.00 0.51
0.31 0.35 0.79 -0.09 -0.05 0.13 0.01 0.99 0.83 0.94 0.78 0.88 0.51 1.00
Tourist_Nights
Tripadvisor_Reviews
GoogleTrends_Index
Avg_Temperature_C
Rainfall_mm
Sunshine_Hours
Event_Count
Total_Arrivals
India
UK
China
Germany
Russia
Other
−0.4

4. Feature Engineering

This section focuses on creating new features from the existing data, including time-based features, one-hot encoding for categorical variables, generating lag features (e.g., tourist nights from the previous year), and integrating a Prophet forecast for total arrivals as a potential leading indicator.

```
# Create time-based features
df['Month_Num'] = df['Date'].dt.month
df['Year'] = df['Date'].dt.year

# Encode Categorical Features
df_encoded = pd.get_dummies(df, columns=['Season', 'Event_Impact'], drop_first=True)

# Create Lag Features
df_encoded = df_encoded.sort_values(by=['District', 'Date'])
df_encoded['Tourist_Nights_Lag_12'] = df_encoded.groupby('District')['Tourist_Nights'].shift(12)

# Add Prophet Forecast for Total Arrivals as a Feature
total_arrivals_df_prophet = df.groupby('Date')['Total_Arrivals'].sum().reset_index()
total_arrivals_df_prophet = total_arrivals_df_prophet.rename(columns={'Date': 'ds', 'Total_Arrivals': 'y'})
model_prophet = Prophet(seasonality_mode='additive')
model_prophet.fit(total_arrivals_df_prophet)
future_prophet = pd.DataFrame({'ds': df_encoded['Date'].unique()})
forecast_prophet = model_prophet.predict(future_prophet)
forecast_subset = forecast_prophet[['ds', 'yhat']].copy()
forecast_subset = forecast_subset.rename(columns={'ds': 'Date', 'yhat': 'Forecasted_Total_Arrivals'})
df_merged_features = pd.merge(df_encoded, forecast_subset, on='Date', how='left')

# Drop rows with NaN values created by the lag feature or missing forecasts
df_final = df_merged_features.dropna()

print("Feature engineering complete. Final dataset shape:", df_final.shape)
display(df_final.head())
```

```
INFO:prophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override this.
INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpj2rvtpv3/_vh0aol3.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpj2rvtpv3/l7_0g398.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.12/dist-packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=23451', '
23:43:51 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
23:43:51 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
Feature engineering complete. Final dataset shape: (4498, 34)
```

	Year	Month	District	Tourist_Nights	Tripadvisor_Reviews	GoogleTrends_Index	Avg_Temperature_C	Rainfall_mm	Sunshine_Hours	Eve
12	2011	january	Ampara	7493	52	28	26.0	205.0	190.0	
13	2011	february	Ampara	7854	55	29	27.0	115.0	220.0	
14	2011	march	Ampara	8129	57	30	28.0	70.0	250.0	
15	2011	april	Ampara	6537	43	26	29.0	75.0	240.0	
16	2011	may	Ampara	5822	35	23	30.0	55.0	230.0	

5 rows × 34 columns

5. Data Splitting

This section splits the prepared data into training and testing sets. A time-based split is used to ensure that the models are trained on historical data and evaluated on a future period (2019), excluding the anomaly period of 2020–2021 from the training data.

```
# We will train the model on data before the major disruption to learn stable patterns.
# We define our training data from before 2019.
# The test data will be from a "normal" period (2019) to validate performance.
train_df = df_final[df_final['Year'] < 2019].copy()
test_df = df_final[df_final['Year'] == 2019].copy()
```

```
# Define the features (X) and the target (y)
# Our goal is to predict 'Tourist_Nights'
target = 'Tourist_Nights'
features = [col for col in df_final.columns if col not in ['Year', 'Month', 'District', 'Date', target]]

X_train = train_df[features]
y_train = train_df[target]
X_test = test_df[features]
y_test = test_df[target]

print(f"Training data shape: {X_train.shape}")
print(f"Testing data shape: {X_test.shape}")
```

```
Training data shape: (2398, 29)
Testing data shape: (300, 29)
```

6. Base Model Training (Random Forest)

This section initializes and trains a Random Forest Regressor model on the training data as a baseline model. Its initial performance is then evaluated using standard metrics like MAE, RMSE, and R^2.

```
# Initialize and train the Random Forest model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1)
rf_model.fit(X_train, y_train)

# Make predictions on the test set
rf_preds = rf_model.predict(X_test)

# Evaluate the base model
rf_mae = mean_absolute_error(y_test, rf_preds)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_preds))
rf_r2 = r2_score(y_test, rf_preds)

print("--- Base Model (Random Forest) Performance ---")
print(f"Mean Absolute Error (MAE): {rf_mae:,.0f}")
print(f"Root Mean Squared Error (RMSE): {rf_rmse:,.0f}")
print(f"R-squared (R²): {rf_r2:.4f}")
```

```
--- Base Model (Random Forest) Performance ---
Mean Absolute Error (MAE): 7,359
Root Mean Squared Error (RMSE): 14,866
R-squared (R²): 0.9793
```

7. Ship-Readiness Checklist: Add Scale-Aware Metrics (for Base Model)

This section calculates and reports scale-aware performance metrics, sMAPE (Symmetric Mean Absolute Percentage Error) and WAPE (Weighted Absolute Percentage Error), for the base Random Forest model to provide a more robust assessment that is less sensitive to the scale of the target variable.

```
# Define sMAPE and WAPE functions
def smape(y,yhat): return np.mean(2*np.abs(yhat-y)/(np.abs(y)+np.abs(yhat)+1e-8))*100
def wape(y,yhat):  return np.sum(np.abs(yhat-y))/(np.sum(np.abs(y))+1e-8)

# Assuming rf_preds are the predictions from the Random Forest model on the test set
print("--- Random Forest Robust Model Performance Metrics (Ship-readiness Check) ---")
print("sMAPE:", smape(y_test, rf_preds))
print("WAPE :", wape(y_test, rf_preds))
```

```
--- Random Forest Robust Model Performance Metrics (Ship-readiness Check) ---
sMAPE: 16.884569211410703
WAPE : 0.11120878780991246
```

8. Meta Model Training (Initial XGBoost)

This section initializes and trains an initial XGBoost Regressor model. XGBoost is an advanced gradient boosting model often used for its performance. Its initial performance is evaluated on the test set using standard metrics.

```
# Initialize and train the XGBoost Regressor model
xgb_model = xgb.XGBRegressor(objective='reg:squarederror',
                             n_estimators=1000,
                             learning_rate=0.05,
                             random_state=42,
                             n_jobs=-1)

xgb_model.fit(X_train, y_train)

# Make predictions on the test set
xgb_preds = xgb_model.predict(X_test)

# Evaluate the initial XGBoost model
xgb_mae = mean_absolute_error(y_test, xgb_preds)
xgb_rmse = np.sqrt(mean_squared_error(y_test, xgb_preds))
xgb_r2 = r2_score(y_test, xgb_preds)

print("\n--- Initial Meta Model (XGBoost) Performance ---")
print(f"Mean Absolute Error (MAE): {xgb_mae:,.0f}")
print(f"Root Mean Squared Error (RMSE): {xgb_rmse:,.0f}")
print(f"R-squared (R²): {xgb_r2:.4f}")
```

```
--- Initial Meta Model (XGBoost) Performance ---
Mean Absolute Error (MAE): 7,971
Root Mean Squared Error (RMSE): 16,989
R-squared (R²): 0.9730
```

9. Model Tuning (XGBoost GridSearchCV)

This section focuses on tuning the hyperparameters of the XGBoost model using GridSearchCV with TimeSeriesSplit for time-aware cross-validation. This systematic search aims to find the best combination of hyperparameters to improve model performance.

```
# Define the parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [100, 300, 500],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'subsample': [0.7, 0.9],
    'colsample_bytree': [0.7, 0.9],
    'gamma': [0, 0.1]
}

print("Parameter grid defined for XGBoost tuning.")
print(param_grid)
```

```
Parameter grid defined for XGBoost tuning.
{'n_estimators': [100, 300, 500], 'learning_rate': [0.01, 0.05, 0.1], 'max_depth': [3, 5, 7], 'subsample': [0.7, 0.9], 'colsample_bytree
```

```
# Initialize GridSearchCV
from sklearn.model_selection import GridSearchCV, TimeSeriesSplit

# Instantiate an XGBoost Regressor
xgb_model_tune = xgb.XGBRegressor(objective='reg:squarederror', random_state=42, n_jobs=-1)

# Instantiate GridSearchCV
# Using TimeSeriesSplit for time-aware cross-validation
tscv = TimeSeriesSplit(n_splits=3)

grid_search = GridSearchCV(
    estimator=xgb_model_tune,
    param_grid=param_grid,
    cv=tscv,  # Using TimeSeriesSplit for cross-validation
    scoring='neg_mean_absolute_error', # Using negative MAE as the scoring metric
    n_jobs=-1, # Use all available cores
    verbose=2 # Display progress
)
```

```
print("GridSearchCV object initialized with TimeSeriesSplit.")
```

```
GridSearchCV object initialized with TimeSeriesSplit.
```

```
# Perform Grid Search
print("Starting GridSearchCV fit...")
grid_search.fit(X_train, y_train)
print("GridSearchCV fit complete.")

# Print the best hyperparameters found
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)
```

```
Starting GridSearchCV fit...
Fitting 3 folds for each of 216 candidates, totalling 648 fits
GridSearchCV fit complete.

Best hyperparameters found by GridSearchCV:
{'colsample_bytree': 0.9, 'gamma': 0, 'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500, 'subsample': 0.9}
```

10. Evaluate the Tuned Model (XGBoost)

This section evaluates the performance of the XGBoost model with the best hyperparameters found by GridSearchCV on the test set using standard metrics (MAE, RMSE, R^2).

```
# Get the best estimator from GridSearchCV
best_xgb_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
tuned_xgb_preds = best_xgb_model.predict(X_test)

# Evaluate the tuned model
tuned_xgb_mae = mean_absolute_error(y_test, tuned_xgb_preds)
tuned_xgb_rmse = np.sqrt(mean_squared_error(y_test, tuned_xgb_preds))
tuned_xgb_r2 = r2_score(y_test, tuned_xgb_preds)

print("\n--- Tuned XGBoost Model Performance (on Test Set) ---")
print(f"Mean Absolute Error (MAE): {tuned_xgb_mae:,.0f}")
print(f"Root Mean Squared Error (RMSE): {tuned_xgb_rmse:,.0f}")
print(f"R-squared (R²): {tuned_xgb_r2:.4f}")
```

```
--- Tuned XGBoost Model Performance (on Test Set) ---
Mean Absolute Error (MAE): 7,712
Root Mean Squared Error (RMSE): 15,874
R-squared (R²): 0.9764
```

11. Calculate Scale-Aware Metrics for the Tuned XGBoost Model

This section calculates and reports the scale-aware metrics (sMAPE and WAPE) for the tuned XGBoost model to provide a complete performance assessment that is independent of the target variable's scale.

```
# Calculate scale-aware metrics for the tuned XGBoost model
tuned_xgb_smape = smape(y_test, tuned_xgb_preds)
tuned_xgb_wape = wape(y_test, tuned_xgb_preds)

print("\n--- Tuned XGBoost Robust Model Performance Metrics ---")
print(f"sMAPE: {tuned_xgb_smape:.4f}")
print(f"WAPE : {tuned_xgb_wape:.4f}")
```

```
--- Tuned XGBoost Robust Model Performance Metrics ---
sMAPE: 23.5324
WAPE : 0.1166
```

12. Compare Performance Metrics of All Models (RF, Initial XGBoost, Tuned XGBoost)

This section provides a comparative analysis of the performance metrics (standard and scale-aware) for the initial Random Forest model, the initial XGBoost model, and the tuned XGBoost model on the test set to determine the best-performing model among these options.

```
print("\n--- Overall Model Performance Comparison (RF, Initial XGBoost, Tuned XGBoost) ---")

print("\nStandard Metrics (MAE, RMSE, R²):")
print(f"  Random Forest (Initial): MAE={rf_mae:,.0f}, RMSE={rf_rmse:,.0f}, R²={rf_r2:.4f}")
print(f"  XGBoost (Initial): MAE={xgb_mae:,.0f}, RMSE={xgb_rmse:,.0f}, R²={xgb_r2:.4f}")
print(f"  XGBoost (Tuned): MAE={tuned_xgb_mae:,.0f}, RMSE={tuned_xgb_rmse:,.0f}, R²={tuned_xgb_r2:.4f}")

print("\nScale-Aware Metrics (sMAPE, WAPE):")
print(f"  Random Forest (Initial): sMAPE={smape(y_test, rf_preds):.4f}, WAPE={wape(y_test, rf_preds):.4f}")
print(f"  XGBoost (Initial): sMAPE={smape(y_test, xgb_preds):.4f}, WAPE={wape(y_test, xgb_preds):.4f}") # Add sMAPE/WAPE for initial XGBoost
print(f"  XGBoost (Tuned): sMAPE={tuned_xgb_smape:.4f}, WAPE={tuned_xgb_wape:.4f}")

print("\nAnalysis and Summary:")
print("Based on the metrics on the 2019 test set:")
print("- Compare MAE, RMSE, R², sMAPE, and WAPE across the three models to identify the best performer.")
print("- Note if tuning improved XGBoost's performance compared to the initial XGBoost model.")
print("- Conclude which model is the champion based on the chosen evaluation metrics.")
```

```
--- Overall Model Performance Comparison (RF, Initial XGBoost, Tuned XGBoost) ---

Standard Metrics (MAE, RMSE, R²):
  Random Forest (Initial): MAE=7,359, RMSE=14,866, R²=0.9793
  XGBoost (Initial): MAE=7,971, RMSE=16,989, R²=0.9730
  XGBoost (Tuned): MAE=7,712, RMSE=15,874, R²=0.9764

Scale-Aware Metrics (sMAPE, WAPE):
  Random Forest (Initial): sMAPE=16.8846, WAPE=0.1112
  XGBoost (Initial): sMAPE=16.7993, WAPE=0.1205
  XGBoost (Tuned): sMAPE=23.5324, WAPE=0.1166

Analysis and Summary:
Based on the metrics on the 2019 test set:
- Compare MAE, RMSE, R², sMAPE, and WAPE across the three models to identify the best performer.
- Note if tuning improved XGBoost's performance compared to the initial XGBoost model.
- Conclude which model is the champion based on the chosen evaluation metrics.
```

13. Benchmarking Against a Naive Model

This section benchmarks the performance of the champion model (identified in the previous step, likely Random Forest or Tuned XGBoost based on metrics) against a simple seasonal naive baseline (predicting based on the previous year's value) using scale-aware metrics (sMAPE and WAPE). This helps assess if the more complex model provides significant value over a basic approach and provides a check for overfitting.

```
# Implement a simple seasonal naive benchmark (e.g., predict the same as 12 months ago)
# We can use the 'Tourist_Nights_Lag_12' feature from the test set as the naive prediction
# Ensure X_test contains 'Tourist_Nights_Lag_12'
if 'Tourist_Nights_Lag_12' in X_test.columns:
    naive_preds = X_test['Tourist_Nights_Lag_12']

    # Evaluate the naive model using scale-aware metrics
    naive_smape = smape(y_test, naive_preds)
    naive_wape = wape(y_test, naive_preds)

    print("--- Naive Model Performance (Seasonal Naive - Lag 12) ---")
    print(f"sMAPE: {naive_smape:.4f}")
    print(f"WAPE : {naive_wape:.4f}")
else:
    print("Skipping Seasonal Naive Benchmark: 'Tourist_Nights_Lag_12' not found in test data features.")
    naive_smape = np.nan
    naive_wape = np.nan

# Compare with the champion model (based on Step 12 analysis, let's assume Random Forest for now)
```

```
# You would replace rf_preds and the metric variables below with the actual champion model's results
print("\n--- Performance Comparison: Champion Model vs. Naive ---")
print(f"Champion Model sMAPE: {smape(y_test, rf_preds):.4f}") # Replace rf_preds with champion preds
print(f"Naive (Lag 12) sMAPE: {naive_smape:.4f}")
print(f"Champion Model WAPE : {wape(y_test, rf_preds):.4f}") # Replace rf_preds with champion preds
print(f"Naive (Lag 12) WAPE : {naive_wape:.4f}")

print("\nAnalysis and Conclusion:")
print("- Compare the champion model's performance metrics with the Naive benchmark.")
print("- A significant improvement (e.g., >10% lower WAPE/sMAPE) over the naive model suggests the champion model is capturing more complex
```

```
--- Naive Model Performance (Seasonal Naive - Lag 12) ---
sMAPE: 57.0245
WAPE : 0.5689

--- Performance Comparison: Champion Model vs. Naive ---
Champion Model sMAPE: 16.8846
Naive (Lag 12) sMAPE: 57.0245
Champion Model WAPE : 0.1112
Naive (Lag 12) WAPE : 0.5689

Analysis and Conclusion:
- Compare the champion model's performance metrics with the Naive benchmark.
- A significant improvement (e.g., >10% lower WAPE/sMAPE) over the naive model suggests the champion model is capturing more complex pat
```

14. Implement District-Specific Models (using features including Prophet forecast)

This section implements the strategy of training separate forecasting models for each district. Random Forest models are trained individually for each unique district using the prepared data, including the forecasted total arrivals and other engineered features. This approach aims to capture district-specific patterns for potentially better local predictions.

```
# Identify the unique districts present in the training data
unique_districts_train = train_df['District'].unique()
print(f"Unique districts in training data: {len(unique_districts_train)}")

# Create an empty dictionary to store the trained models
district_models = {}

# Define features list based on X_train columns
district_features = X_train.columns.tolist()

# Iterate through each unique district in the training data
for district in unique_districts_train:
    print(f"\nTraining model for {district}...")

    # Filter training data for the specific district
    district_train_df = train_df[train_df['District'] == district].copy()

    X_train_district = district_train_df[district_features]
    y_train_district = district_train_df[target]

    if X_train_district.empty:
        print(f"Skipping {district}: No training data available.")
        continue

    # Initialize a Random Forest Regressor model
    district_model = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1)

    # Train the model
    district_model.fit(X_train_district, y_train_district)

    # Store the trained model
    district_models[district] = district_model
    print(f"Model trained for {district}.")

print(f"\nFinished training models for {len(district_models)} districts.")
```

```
Model trained for Jaffna.
```

```
Model trained for Jaffna.

Training model for Kalutara...
Model trained for Kalutara.

Training model for Kandy...
Model trained for Kandy.

Training model for Kegalle...
Model trained for Kegalle.

Training model for Kilinochchi...
Model trained for Kilinochchi.

Training model for Kurunegala...
Model trained for Kurunegala.

Training model for Mannar...
Model trained for Mannar.

Training model for Matale...
Model trained for Matale.

Training model for Matara...
Model trained for Matara.

Training model for Monaragala...
Model trained for Monaragala.

Training model for Mullaitivu...
Model trained for Mullaitivu.

Training model for Nuwara Eliya...
Model trained for Nuwara Eliya.

Training model for Polonnaruwa...
Model trained for Polonnaruwa.

Training model for Puttalam...
Model trained for Puttalam.

Training model for Ratnapura...
Model trained for Ratnapura.

Training model for Trincomalee...
Model trained for Trincomalee.

Training model for Vavuniya...
Model trained for Vavuniya.

Finished training models for 25 districts.
```

15. Model Evaluation (District-Specific)

This section evaluates the performance of each trained district-specific model on the test set (2019 data). Accuracy metrics such as MAE, RMSE, sMAPE, and WAPE are calculated and reported for each individual district, providing a detailed view of how well each district's tourist nights are predicted.

```
print("\n--- District-Level Model Performance Metrics (Random Forest) ---")

district_performance = {}

# Define features list based on X_test columns for consistency
district_features_test = X_test.columns.tolist()

# Iterate through the trained district models
for district, model in district_models.items():
    print(f"\nEvaluating performance for {district}:")

    # Retrieve the test data for the current district
    district_test_df = test_df[test_df['District'] == district].copy()

    if district_test_df.empty:
        print(f"No test data available for {district}. Skipping evaluation.")
        district_performance[district] = "No test data"
        continue

    # Define features (X) and target (y) for this district's test data
```

```
        X_test_district = district_test_df[district_features_test]
        y_test_district = district_test_df[target]

        # Make predictions using the trained model
        district_preds = model.predict(X_test_district)

        # Calculate performance metrics
        mae = mean_absolute_error(y_test_district, district_preds)
        rmse = np.sqrt(mean_squared_error(y_test_district, district_preds))
        smape_val = smape(y_test_district, district_preds)
        wape_val = wape(y_test_district, district_preds)

        # Store and print metrics
        district_performance[district] = {
            'MAE': mae,
            'RMSE': rmse,
            'sMAPE': smape_val,
            'WAPE': wape_val
        }

        print(f"  MAE: {mae:,.0f}")
        print(f"  RMSE: {rmse:,.0f}")
        print(f"  sMAPE: {smape_val:.4f}")
        print(f"  WAPE : {wape_val:.4f}")

    # You can optionally convert district_performance dictionary to a DataFrame for easier analysis
    # district_performance_df = pd.DataFrame.from_dict(district_performance, orient='index')
    # print("\nSummary DataFrame of District Performance:")
    # display(district_performance_df)
```

```
  WAPE : 0.1583

 Evaluating performance for Vavuniya:
  MAE: 361
  RMSE: 403
  sMAPE: 17.1749
  WAPE : 0.1218
```

16. Benchmarking (District-Specific vs. Baselines)

This section compares the aggregate performance of the district-specific models against relevant baselines: a Seasonal Naive model and a Prophet proportional model. This comparison uses scale-aware metrics (sMAPE and WAPE) to demonstrate the value added by the district-specific modeling approach and the engineered features.

```
print("\n--- Benchmarking District-Specific Models vs. Baselines ---")

# Calculate aggregate performance metrics for district-specific RF models on the test set
# Aggregate actuals and predictions from district models on the test set
district_rf_preds_all = []
district_actuals_all = []

for district in district_models.keys(): # Iterate through districts for which models were trained
    district_test_df = test_df[test_df['District'] == district].copy()

    if district_test_df.empty:
        continue

    district_features_test = X_test.columns.tolist()
    X_test_district = district_test_df[district_features_test]
    y_test_district = district_test_df[target]

    # Ensure the model exists for the district before predicting
    if district in district_models:
        district_model = district_models[district]
        preds = district_model.predict(X_test_district)
        district_rf_preds_all.extend(preds)
        district_actuals_all.extend(y_test_district)

# Convert lists to numpy arrays for metric calculation
district_rf_preds_all = np.array(district_rf_preds_all)
district_actuals_all = np.array(district_actuals_all)

district_rf_smape = smape(district_actuals_all, district_rf_preds_all)
district_rf_wape = wape(district_actuals_all, district_rf_preds_all)

print("--- Average District-Specific Random Forest Model Performance (on Test Set) ---")
print(f"sMAPE: {district_rf_smape:.4f}")
print(f"WAPE : {district_rf_wape:.4f}")

# Calculate and print Seasonal Naive Benchmark performance
# Need to ensure test_df_prop is available or recalculate it for naive predictions
if 'Tourist_Nights_Lag_12' in test_df.columns:
    naive_preds = test_df['Tourist_Nights_Lag_12'] # Corrected from test_test_df
    naive_smape = smape(test_df['Tourist_Nights'], naive_preds)
    naive_wape = wape(test_df['Tourist_Nights'], naive_preds)

    print("\n--- Seasonal Naive Benchmark Performance (on Test Set) ---")
    print(f"sMAPE: {naive_smape:.4f}")
    print(f"WAPE : {naive_wape:.4f}")
else:
    print("\nSkipping Seasonal Naive Benchmark: 'Tourist_Nights_Lag_12' not found in test data features.")
    naive_smape = np.nan
    naive_wape = np.nan


# Calculate and print Prophet Proportional Naive Baseline performance
# Need to ensure test_df_prop is available or recalculate it for Prophet predictions
# Recalculate test_df_prop to be safe in this restructured notebook
total_arrivals_df_prophet = df.groupby('Date')['Total_Arrivals'].sum().reset_index()
total_arrivals_df_prophet = total_arrivals_df_prophet.rename(columns={'Date': 'ds', 'Total_Arrivals': 'y'}) # Corrected syntax
model_prophet = Prophet(seasonality_mode='additive')
model_prophet.fit(total_arrivals_df_prophet)
future_prophet_all = pd.DataFrame({'ds': df['Date'].unique()})
forecast_prophet_all = model_prophet.predict(future_prophet_all)
```