Solar Energy Generation Prediction - Full Walkthrough / Guide

This document provides a complete, step-by-step walkthrough of the solar energy prediction pipeline built for predicting hourly solar capacity factors across 29 European countries. It covers everything from raw data exploration and cleaning, through feature engineering and merging, to training two machine learning models and analysing their performance through diagnostic visualisations.

Table of Contents

Datasets Used

Dataset	Source	Temporal Range	Resolution	Description
EMHIRES Solar PV	European Commission Joint Research Centre	1986 - 2015	Hourly	Solar capacity factors for 29 European countries
NASA POWER Hourly Weather	NASA POWER API	2001 - 2015	Hourly	Irradiance, Temperature, and Wind Speed at each country's geographic centroid

> The EMHIRES dataset provides the **target variable** (Capacity_Factor), while the NASA POWER dataset provides the **meteorological features** that drive solar generation.

Models Trained

Model	Type	Purpose
Linear Regression	Parametric, linear	Interpretable baseline model
Random Forest Regressor	Non-parametric, ensemble	Non-linear model for improved accuracy

Target Variable

`Capacity_Factor` - A value between 0.0 and 1.0 representing how efficiently a solar panel converts available sunlight into electricity at a given hour. A value of 0.0 means zero output (nighttime), while 1.0 means maximum theoretical output.

Pipeline Overview

The end-to-end pipeline follows this sequence:

1. Import Libraries - Load all necessary Python packages.
2. Load and Explore the Dataset - Read the EMHIRES CSV and understand its structure.
3. Visualise the Raw Data - Plot a single day's solar cycle to build intuition.
4. NASA POWER Data Fetching (Reference) - Document how the weather data was collected (not executed here).
5. Data Cleaning and Quality Checks - Verify data integrity before processing.
6. Data Structuralization and Temporal Validation - Reshape the dataset, create timestamps, extract features.
7. Master Dataset Integration and Memory Optimization - Merge generation data with NASA weather data.
8. Categorical Encoding (One-Hot Encoding) - Convert country labels into numerical features.
9. Training and Evaluation (Linear Regression) - Train the first model and measure performance.
10. Model Export (Linear Regression) - Save the trained model to disk.
11. Analysis and Visualisation (Linear Regression) - Detailed graphical analysis of model performance.
12. Training and Evaluation (Random Forest Regressor) - Train the second model and measure performance.
13. Model Export (Random Forest Regressor) - Save the trained model to disk.
14. Analysis and Visualisation (Random Forest Regressor) - Detailed graphical analysis of model performance.
15. Custom Prediction Example - Demonstrate how to use a trained model for inference.

1. Importing Libraries

All required libraries are loaded upfront. These cover data manipulation (`pandas`, `numpy`), visualisation (`matplotlib`, `seaborn`), machine learning (`scikit-learn`), model persistence (`joblib`), and memory management (`gc`).

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import seaborn as sns
import numpy as np
import gc
import joblib
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

Library	Role
`pandas`	Tabular data manipulation, CSV I/O, datetime handling
`numpy`	Numerical operations, array mathematics
`matplotlib`	Low-level plotting and figure configuration
`seaborn`	Statistical visualisation built on matplotlib
`scikit-learn`	Model training, evaluation metrics, data splitting
`joblib`	Serialising trained models to `.pkl` files
`gc`	Explicit garbage collection for memory management

> These libraries form the standard toolkit for a tabular machine learning pipeline in Python. No deep learning frameworks are required for this task because the relationship between weather conditions and solar output is well-captured by classical regression methods.

2. Loading and Exploring the EMHIRES Dataset

The EMHIRES PV dataset (`EMHIRESPV_TSh_CF_Country_19862015.csv`) contains hourly solar capacity factors for 29 European countries spanning 1986 to 2015. Each column represents a country (using its two-letter ISO code), and each row represents one hour of data. The values range from 0.0 (no solar production) to 1.0 (maximum theoretical output).

The dataset is structured in **wide format** - every country is a separate column. This is convenient for quick country-level lookups but must be reshaped before machine learning.

```
df = pd.read_csv('EMHIRESPV_TSh_CF_Country_19862015.csv')
print(f"Dataset Shape: {df.shape}")
print(f"Countries: {list(df.columns)}")
print(f"Total Hours Recorded: {len(df):,}")
df.head(15)
```

What the Raw Data Looks Like

Property	Value
Rows	262,968 (one per hour from Jan 1986 to Dec 2015)
Columns	29 (one per country)
Format	Wide - each country is a separate column
Value Range	0.0 to 1.0

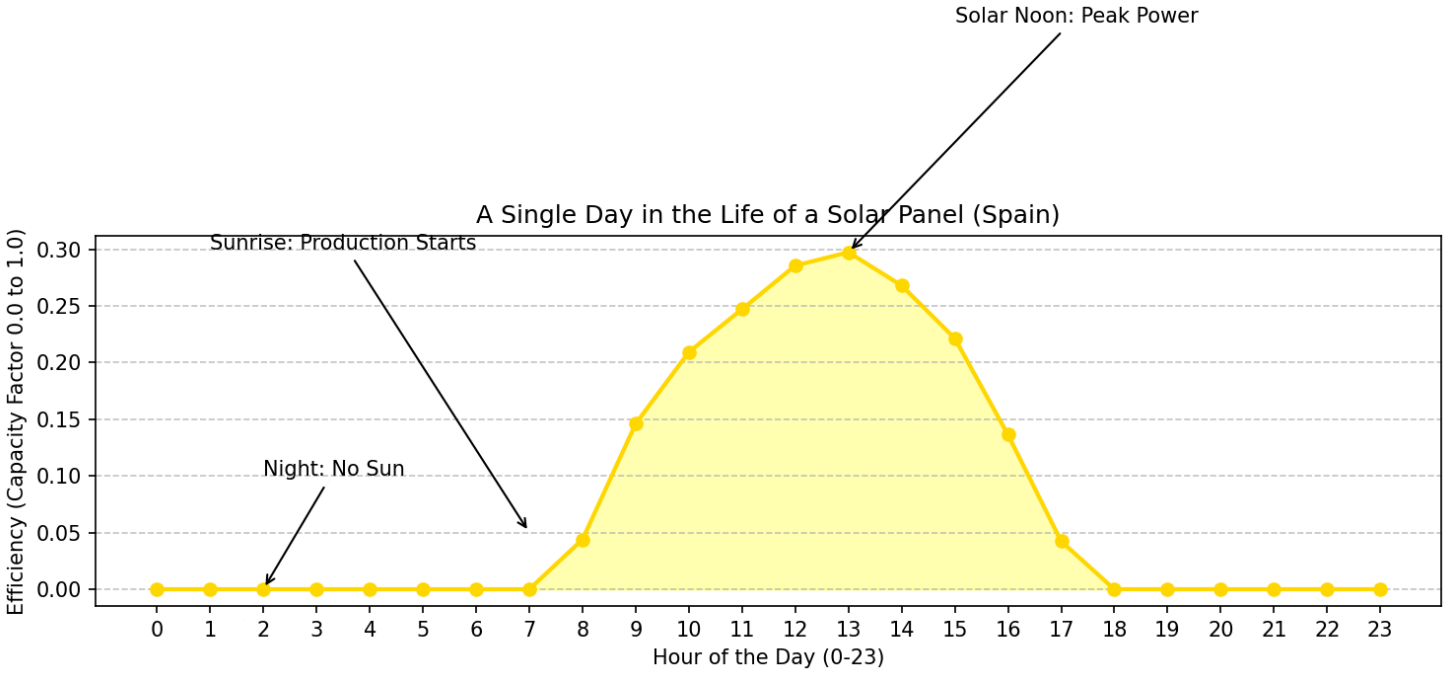
The 29 countries represented are: AT, BE, BG, CH, CY, CZ, DE, DK, EE, EL, ES, FI, FR, HR, HU, IE, IT, LT, LU, LV, NL, NO, PL, PT, RO, SE, SI, SK, UK.

> The wide format is ideal for quick visual inspection and country-level comparisons, but machine learning models require a single target column. This necessitates reshaping (melting) the data in a later step.

3. Visualising the Raw Data - A Single Day's Solar Cycle

Before any transformation, it is important to understand what the raw data looks like. The plot below shows 24 hours of solar capacity factor data for Spain (ES). This reveals the natural "heartbeat" of solar production: zero output at night, a ramp-up at sunrise, a peak at solar noon, and a ramp-down at sunset.

```
one_day = df['ES'].iloc[0:24]

plt.figure(figsize=(10, 6))
plt.plot(one_day, marker='o', color='gold', linewidth=2, label='Solar Efficiency')
plt.fill_between(range(24), one_day, color='yellow', alpha=0.3)

plt.annotate('Night: No Sun', xy=(2, 0), xytext=(2, 0.1), arrowprops=dict(arrowstyle='->'))
plt.annotate('Sunrise: Production Starts', xy=(7, 0.05), xytext=(1, 0.3), arrowprops=dict(arrowstyle='->'))
plt.annotate('Solar Noon: Peak Power', xy=(13, one_day.max()), xytext=(15, 0.5), arrowprops=dict(arrowstyle='->'))

plt.title('A Single Day in the Life of a Solar Panel (Spain)')
plt.ylabel('Efficiency (Capacity Factor 0.0 to 1.0)')
plt.xlabel('Hour of the Day (0-23)')
plt.xticks(range(0, 24))
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



Interpreting the Solar Lifecycle

The plot above illustrates the fundamental physics that drive the data:

> **The Zero Line (Nighttime):** From hour 0 to approximately hour 7, and again from hour 18 to 23, the capacity factor is 0.00. No solar radiation reaches the panels during these hours.

> **The Ramping Phase (Sunrise):** Starting around hour 7, there is a sharp ramp-up. This is the moment the sun crosses the horizon and irradiance begins converting into electrical current.

> **Solar Noon (The Peak):** The curve peaks around hour 13. This is the point of maximum capacity factor, when the sun is at its highest angle in the sky, providing the most intense irradiance.

> **The Ramping Phase (Sunset):** After the peak, production gradually ramps down as the sun descends, eventually returning to zero.

This daily cycle is the core pattern that the machine learning models will learn to predict. Every row in the dataset represents one snapshot from this repeating cycle, and the model's job is to estimate where in the cycle each snapshot falls based on the time-of-day, season, and local weather conditions.

4. NASA POWER Weather Data (Reference Only)

The weather data used in this pipeline was fetched from the NASA POWER API. The script (`nasa.py`) was used to collect hourly Irradiance, Temperature, and Wind Speed for all 29 EMHIRES countries from 2001 to 2015. The fetched data is saved as `nasa_weather_master.csv`.

> This script is provided for reference only and is not meant to be executed in the notebook. The fetch process takes approximately 20+ minutes due to the sheer amount of data and API rate limiting. The resulting CSV file is already included in the project and will be loaded directly in the next steps.

Geographic Centroids

For each of the 29 countries, the approximate geographic centroid was used as the point location for querying the NASA POWER API. This is a simplification - in reality, solar infrastructure is distributed across each country - but centroids provide a reasonable national-average approximation for the meteorological conditions.

Country	Latitude	Longitude	Country	Latitude	Longitude
AT	47.51	14.55	LT	55.16	23.88
BE	50.50	4.47	LU	49.81	6.12
BG	42.73	25.48	LV	56.87	24.60
CH	46.81	8.22	NL	52.13	5.29
CY	35.12	33.42	NO	60.47	8.46
CZ	49.81	15.47	PL	51.91	19.14
DE	51.16	10.45	PT	39.39	-8.22
DK	56.26	9.50	RO	45.94	24.96
EE	58.59	25.01	SE	60.12	18.64
EL	39.07	21.82	SI	46.15	14.99
ES	40.46	-3.74	SK	48.66	19.69
FI	61.92	25.74	UK	55.37	-3.43
FR	46.22	2.21			
HR	45.10	15.20			
HU	47.16	19.50			
IE	53.41	-8.24			
IT	41.87	12.56			

The Fetch Script

```python
import pandas as pd
import requests
import time

countries = {
    'AT': {'lat': 47.51, 'lon': 14.55}, 'BE': {'lat': 50.50, 'lon': 4.47},
    'BG': {'lat': 42.73, 'lon': 25.48}, 'CH': {'lat': 46.81, 'lon': 8.22},
    'CY': {'lat': 35.12, 'lon': 33.42}, 'CZ': {'lat': 49.81, 'lon': 15.47},
    'DE': {'lat': 51.16, 'lon': 10.45}, 'DK': {'lat': 56.26, 'lon': 9.50},
    'EE': {'lat': 58.59, 'lon': 25.01}, 'EL': {'lat': 39.07, 'lon': 21.82},
    'ES': {'lat': 40.46, 'lon': -3.74}, 'FI': {'lat': 61.92, 'lon': 25.74},
    'FR': {'lat': 46.22, 'lon': 2.21},  'HR': {'lat': 45.10, 'lon': 15.20},
    'HU': {'lat': 47.16, 'lon': 19.50}, 'IE': {'lat': 53.41, 'lon': -8.24},
    'IT': {'lat': 41.87, 'lon': 12.56}, 'LT': {'lat': 55.16, 'lon': 23.88},
    'LU': {'lat': 49.81, 'lon': 6.12},  'LV': {'lat': 56.87, 'lon': 24.60},
    'NL': {'lat': 52.13, 'lon': 5.29},  'NO': {'lat': 60.47, 'lon': 8.46},
    'PL': {'lat': 51.91, 'lon': 19.14}, 'PT': {'lat': 39.39, 'lon': -8.22},
    'RO': {'lat': 45.94, 'lon': 24.96}, 'SE': {'lat': 60.12, 'lon': 18.64},
    'SI': {'lat': 46.15, 'lon': 14.99}, 'SK': {'lat': 48.66, 'lon': 19.69},
    'UK': {'lat': 55.37, 'lon': -3.43}
}

all_data = []

for code, coords in countries.items():
    for year in range(2001, 2016):
        url = (f"https://power.larc.nasa.gov/api/temporal/hourly/point?"
               f"parameters=ALLSKY_SFC_SW_DWN,T2M,WS2M&community=RE&longitude=
{coords['lon']}"
               f"&latitude={coords['lat']}&start={year}0101&end=
{year}1231&format=JSON")
        try:
            r = requests.get(url, timeout=60)
            if r.status_code == 200:
                json_data = r.json()['properties']['parameter']
                df_year = pd.DataFrame(json_data)
                df_year['Country'] = code
                all_data.append(df_year)
            time.sleep(0.5)
        except Exception as e:
            print(f"Error on {year}: {e}")

```

```
if all_data:
    final_weather = pd.concat(all_data)
    final_weather.index = pd.to_datetime(final_weather.index, format='%Y%m%d%H')
    final_weather.index.name = 'Timestamp'
    final_weather = final_weather.rename(columns={
        'ALLSKY_SFC_SW_DWN': 'Irradiance',
        'T2M': 'Temperature',
        'WS2M': 'Wind_Speed'
    })
    final_weather.to_csv('nasa_weather_master.csv')
```

What the Script Does

- Iterates over all 29 countries and all years from 2001 to 2015.
- For each country-year combination, it sends a GET request to the NASA POWER Hourly API.
- The API returns three parameters:

API Parameter	Renamed To	Description
`ALLSKY_SFC_SW_DWN`	`Irradiance`	Total solar irradiance at the Earth's surface (W/m^2)
`T2M`	`Temperature`	Air temperature at 2 metres above ground (Celsius)
`WS2M`	`Wind_Speed`	Wind speed at 2 metres above ground (m/s)

- All responses are concatenated into a single DataFrame, timestamps are parsed from the index, columns are renamed for consistency, and the result is saved as `nasa_weather_master.csv`.
- A 0.5-second delay is inserted between requests to respect the API's rate limit.

Why These Three Variables?

- **Irradiance** is the primary driver of solar generation. Without sunlight, there is no output.
- **Temperature** affects panel efficiency. Photovoltaic cells lose efficiency at higher temperatures due to increased electron-hole recombination in the semiconductor material.
- **Wind Speed** provides a cooling effect on panels, partially offsetting temperature losses. It also correlates with atmospheric clarity (cloud cover patterns).

5. Data Cleaning and Quality Checks

Before any transformation, the raw data is inspected for common issues:

- **NASA Sentinel Values (-999):** The NASA POWER API uses -999 as a placeholder when sensor data is unavailable or corrupted.
- **NaN/Null Values:** Standard missing data indicators in pandas.

```
nasa_flags = (df == -999).sum().sum()
null_values = df.isnull().sum().sum()

print(f"NASA -999 Flags Found: {nasa_flags}")
print(f"Empty/NaN Cells Found: {null_values}")
```

Results

Check	Count
NASA -999 Flags	0
NaN/Null Values	0

Interpretation

Both checks return zero. This indicates that the EMHIRES dataset has been pre-cleaned by the European Commission's Joint Research Centre. Satellite sensor gaps and transmission errors have already been handled upstream, ensuring a continuous and complete time series.

> The NASA weather data will be checked separately after the merge step, as that dataset may still contain -999 sentinel values from periods of satellite downtime.

6. Data Structuralization and Temporal Validation

This step transforms the raw wide-format EMHIRES data into a machine-learning-ready long format. Three key operations are performed.

Dataset Reshaping (Melting)

The initial dataset has countries as columns. `pd.melt()` transforms this into a long format where:

- A `Country` column holds the two-letter ISO code.
- A `Capacity_Factor` column holds the solar efficiency value.

This is required because the model needs a single target column (y) and country as a categorical feature.

Temporal Alignment and Truncation

- **Timestamping:** Every row is mapped to a real-world calendar starting January 1, 1986, at hourly frequency. This accounts for leap years and variable month lengths.
- **Range Truncation (2001-2015):** The data is cropped to start from 2001. This aligns the generation data with the availability window of NASA's hourly solar irradiance data, which only provides reliable hourly resolution from 2001 onward.

Feature Extraction

- **Hour (0-23):** Extracted from the timestamp to give the model diurnal context (time of day).
- **Month (1-12):** Extracted to provide seasonal context.

```
start_date = '1986-01-01 00:00:00'
timestamps = pd.date_range(start=start_date, periods=len(df), freq='h')
df['Timestamp'] = timestamps

df_long = pd.melt(df,
                  id_vars=['Timestamp'],
                  var_name='Country',
                  value_name='Capacity_Factor')

df_long = df_long[df_long['Timestamp'].dt.year >= 2001]

df_long['Hour'] = df_long['Timestamp'].dt.hour
df_long['Month'] = df_long['Timestamp'].dt.month
```

Why Truncate to 2001-2015?

The EMHIRES dataset begins in 1986, but the NASA POWER Hourly API only provides reliable hourly-resolution meteorological data from 2001 onward. If the earlier years (1986-2000) were kept, the merge step would discard them anyway because there would be no matching weather records. Truncating early avoids carrying unnecessary data through the pipeline, reducing memory usage and processing time.

Why Extract Hour and Month?

Solar generation is fundamentally governed by two temporal cycles:

- **Diurnal cycle (Hour):** The sun rises, peaks, and sets every day. Hour captures where in the daily cycle each observation falls.
- **Seasonal cycle (Month):** Day length and solar angle change throughout the year. Month captures where in the annual cycle each observation falls.

Without these features, the model would have no concept of "time of day" or "season" and would only know the instantaneous weather conditions - missing the temporal context that drives solar output patterns.

7. Master Dataset Integration and Memory Optimization

This is the most critical and resource-intensive step. The goal is to fuse 15 years of hourly power generation data with localized meteorological data, creating a single feature matrix of approximately 3.8 million rows.

Why Sequence Matters

The order of operations is not arbitrary - incorrect sequencing causes catastrophic failures:

> **The Cartesian Risk:** If One-Hot Encoding is applied *before* the merge, the categorical `Country` column is destroyed. Attempting to merge weather data using only `Timestamp` (without the location key) would cause a Cartesian product explosion, multiplying 3.8 million rows into hundreds of millions and instantly crashing the runtime.

> **Memory Bloat:** Default Python float types (`float64`) consume double the RAM needed for the precision levels required in atmospheric modelling.

Operations Performed

Operation	Purpose
Precision Downcasting	Converts NASA float64 columns to float32, reducing memory footprint by ~50%
Composite Key Merge (Inner Join)	Joins on `['Timestamp', 'Country']` to match generation with local weather
Post-Merge Cleaning	Removes rows with NASA sentinel values (-999.0) or negative irradiance
Active Garbage Collection	Explicitly deletes intermediate DataFrames and triggers `gc.collect()`

```
df_weather = pd.read_csv('nasa_weather_master.csv')
df_weather['Timestamp'] = pd.to_datetime(df_weather['Timestamp'])

float_cols = df_weather.select_dtypes(include=['float64']).columns
df_weather[float_cols] = df_weather[float_cols].astype('float32')

df_master = pd.merge(df_long, df_weather, on=['Timestamp', 'Country'], how='inner')

null_counts = df_master[['Irradiance', 'Temperature', 'Wind_Speed']].isnull().sum()
sentinel_counts = (df_master[['Irradiance', 'Temperature', 'Wind_Speed']] == -999.0).sum()

```

```
df_master = df_master[
    (df_master['Irradiance'] >= 0) &
    (df_master['Temperature'] != -999.0) &
    (df_master['Wind_Speed'] != -999.0)
]

del df_weather
gc.collect()
```

Why Inner Join?

An inner join is used because only rows that exist in both datasets are valid for training. If a timestamp-country pair exists in the generation data but not in the weather data (or vice versa), the row would have missing features and could not be used for prediction. The inner join automatically discards these incomplete pairs.

Why Filter Negative Irradiance?

Solar irradiance is a physical quantity that cannot be negative. A negative value indicates a sensor error or data processing artefact. Similarly, -999.0 is the NASA POWER API's explicit sentinel for "data unavailable." Both types of invalid readings are removed to prevent the model from learning incorrect relationships.

8. Categorical Encoding (One-Hot Encoding)

The Problem

Machine learning algorithms, including Linear Regression, cannot interpret string labels like "AT" or "ES". The `Country` column must be converted into a numerical representation.

The Solution

One-Hot Encoding transforms the single `Country` column into 29 binary columns (one per country). For any given row, exactly one country column is set to 1 and the remaining 28 are set to 0.

Why Not Label Encoding?

Label Encoding assigns ordinal integers to categories (e.g., AT=0, BE=1, BG=2, ...). This implies a mathematical ordering between countries: the model would "think" that BG (2) is between AT (0) and DE (6), or that UK (28) is 28 times more than AT (0). This is meaningless for geographic categories. One-Hot Encoding avoids this by treating each country as an independent binary flag.

Why This Is Done After Merging

Encoding is performed *after* the merge to preserve the `Country` column as a usable merge key. If encoding were done before merging, the string-based `Country` key would no longer exist, making it impossible to join generation data with weather data by location.

Impact

This turns a generic global model into a location-aware forecast system. The model learns separate baseline weights for each country, capturing geographic differences in solar panel infrastructure, latitude, and local climate patterns.

```
df_master = pd.get_dummies(df_master, columns=['Country'], prefix='Country', dtype=int)
df_master.to_csv('merged_encoded.csv', index=False)
```

Final Dataset Shape

Property	Value
Rows	3,812,688
Columns	36
Numeric Features	Hour, Month, Irradiance, Temperature, Wind_Speed
Binary Features	29 country columns (Country_AT through Country_UK)
Target	Capacity_Factor
Non-Feature	Timestamp (dropped before training)

The encoded dataset is exported to `merged_encoded.csv` so that subsequent model training steps can load the fully prepared data directly without repeating the cleaning and merging process.

9. Training and Evaluation - Linear Regression

Objective

Train a Linear Regression model to predict the solar capacity factor from meteorological and temporal features, and quantify its accuracy using standard regression metrics.

Feature Engineering

- **X (Features):** All columns except `Timestamp` (non-numeric) and `Capacity_Factor` (the target). This gives the model: Hour, Month, Irradiance, Temperature, Wind_Speed, and 29 one-hot-encoded country columns - 34 features in total.
- **y (Target):** The `Capacity_Factor` column.

Data Partitioning

The dataset is split into:

Partition	Proportion	Rows	Purpose
Training	80%	3,050,150	Used to learn patterns from historical data
Testing	20%	762,538	Held out to validate predictive accuracy on unseen data

`random_state=67` ensures reproducibility across all runs.

Evaluation Metrics

> **MAE (Mean Absolute Error):** The average of the absolute differences between predicted and actual values. Expressed in the same units as the target.
>
> **RMSE (Root Mean Squared Error):** The square root of the average of squared differences. Like MAE but penalises larger errors more heavily.
>
> **R-Squared (R2):** The proportion of variance in the target that the model explains. A value of 1.0 is perfect; 0.0 means the model is no better than predicting the mean.

```
X = df_master.drop(columns=['Timestamp', 'Capacity_Factor'])
y = df_master['Capacity_Factor']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=67)

lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

lr_train_pred = lr_model.predict(X_train)
lr_test_pred = lr_model.predict(X_test)
```

Results

Metric	Training Set	Testing Set
Mean Absolute Error (MAE)	0.0532	0.0534
Root Mean Squared Error (RMSE)	0.0843	0.0845
R-Squared (R2)	0.7885	0.7880

Interpreting the Linear Regression Results

Linear Regression fits a straight-line relationship between features and the target. Its strength is simplicity and interpretability, but it struggles with non-linear patterns inherent in solar data (e.g., the sharp on/off transition at sunrise and sunset).

Key observations:

- The training and testing metrics are nearly identical, indicating no overfitting. The model generalises well to unseen data.
- The R2 value of 0.7880 means the model explains approximately 78.8% of the variance in solar generation. The remaining 21.2% is attributable to non-linear effects and complex weather interactions that a linear model cannot capture.
- An MAE of 0.0534 means the average prediction is off by about 5.3 percentage points of capacity factor - reasonable for a baseline, but there is room for improvement.

---

10. Exporting the Linear Regression Model

The trained model is serialised using `joblib` and saved as `solar_model_lr.pkl`. This allows the model to be loaded and used for predictions without retraining.

```
joblib.dump(lr_model, 'solar_model_lr.pkl')
```

> `joblib` is preferred over Python's built-in `pickle` for scikit-learn models because it handles large NumPy arrays more efficiently, resulting in smaller file sizes and faster load times.

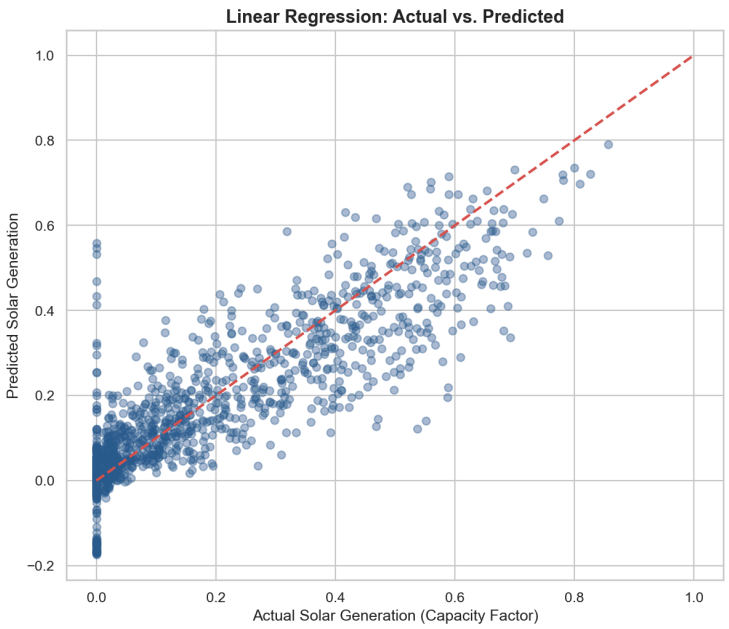
11. Analysis and Visualisation - Linear Regression

The following visualisations provide a detailed diagnostic of the Linear Regression model's behaviour. Two figures are generated, each containing two plots.

Figure 1 - Model Accuracy and Error Analysis

- **Actual vs. Predicted Scatter Plot:** Points along the diagonal line indicate perfect predictions. Spread away from the line shows prediction error.
- **Error Distribution (Residuals):** A histogram of prediction errors. A symmetric, narrow bell curve centred at zero indicates a well-calibrated model.

```
df_analysis_lr = pd.read_csv('merged_encoded.csv')
df_analysis_lr['Timestamp'] = pd.to_datetime(df_analysis_lr['Timestamp'])

X = df_analysis_lr.drop(columns=['Timestamp', 'Capacity_Factor'])
y = df_analysis_lr['Capacity_Factor']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=67)

model_lr = joblib.load('solar_model_lr.pkl')

sns.set_theme(style="whitegrid")
fig1, axes1 = plt.subplots(1, 2, figsize=(16, 7))

sample_indices = np.random.choice(len(y_test), size=2000, replace=False)
y_test_sample = y_test.iloc[sample_indices]
y_pred_sample = model_lr.predict(X_test.iloc[sample_indices])

axes1[0].scatter(y_test_sample, y_pred_sample, alpha=0.4, color='#2b5c8f')
axes1[0].plot([0, 1], [0, 1], '--', color='#d9534f', linewidth=2)
axes1[0].set_title('Linear Regression: Actual vs. Predicted', fontsize=14,
fontweight='bold')
axes1[0].set_xlabel('Actual Solar Generation (Capacity Factor)')
axes1[0].set_ylabel('Predicted Solar Generation')

errors = y_test_sample - y_pred_sample
sns.histplot(errors, bins=50, kde=True, ax=axes1[1], color='#5cb85c')
axes1[1].set_title('Error Distribution (Residuals)', fontsize=14,
fontweight='bold')
axes1[1].set_xlabel('Prediction Error')
axes1[1].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```


Linear Regression: Actual vs. Predicted
Predicted Solar Generation
Actual Solar Generation (Capacity Factor)
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0

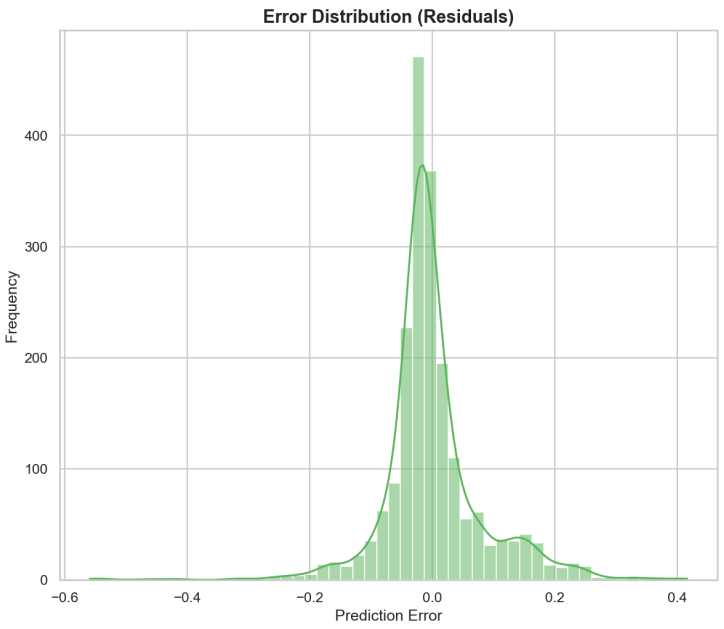
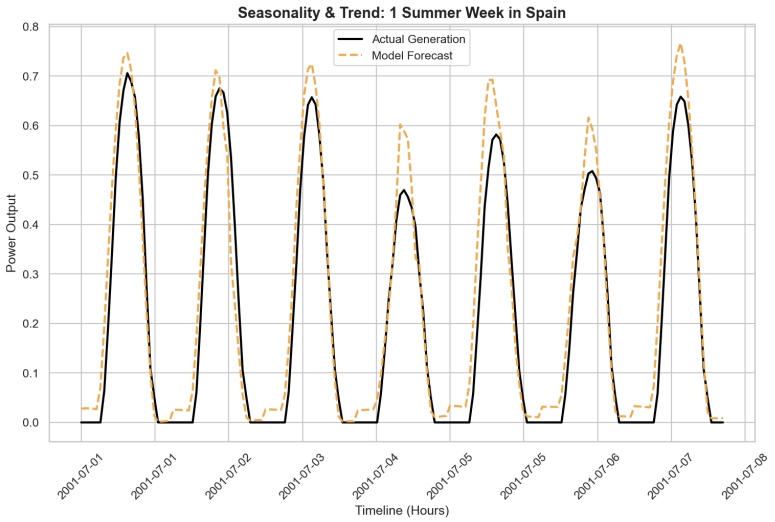
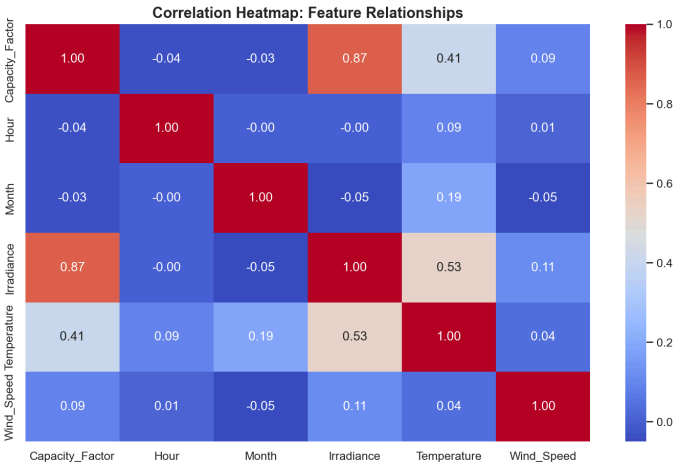

Error Distribution (Residuals)
Frequency
Prediction Error
400
300
200
100
0
−0.6
−0.4
−0.2
0.0
0.2
0.4

Figure 2 - Trends and Feature Relationships

- **Seasonality and Trends (1 Summer Week in Spain):** Overlays actual generation against model forecasts for 168 consecutive hours (1 week) in July for Spain. This tests whether the model captures diurnal cycling.
- **Correlation Heatmap:** Shows pairwise Pearson correlations between the core numerical features, revealing which variables are most strongly associated with the target.

```
fig2, axes2 = plt.subplots(1, 2, figsize=(20, 7))

spain_summer = df_analysis_lr[
    (df_analysis_lr['Country_ES'] == 1) & (df_analysis_lr['Month'] == 7)
].head(168)
X_spain = spain_summer.drop(columns=['Timestamp', 'Capacity_Factor'])
y_true_spain = spain_summer['Capacity_Factor']
y_pred_spain = model_lr.predict(X_spain)

axes2[0].plot(spain_summer['Timestamp'], y_true_spain, label='Actual Generation',
              color='black', linewidth=2)
axes2[0].plot(spain_summer['Timestamp'], y_pred_spain, label='Model Forecast',
              color='#f0ad4e', linestyle='dashed', linewidth=2)
axes2[0].set_title('Seasonality & Trend: 1 Summer Week in Spain', fontsize=14, fontweight='bold')
axes2[0].set_xlabel('Timeline (Hours)')
axes2[0].set_ylabel('Power Output')
axes2[0].xaxis.set_major_locator(ticker.MaxNLocator(nbins=10))
axes2[0].tick_params(axis='x', rotation=45)
axes2[0].legend()

cols_to_check = ['Capacity_Factor', 'Hour', 'Month', 'Irradiance', 'Temperature', 'Wind_Speed']
corr_matrix = df_analysis_lr[cols_to_check].corr()

sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", ax=axes2[1], cbar=True)
axes2[1].set_title('Correlation Heatmap: Feature Relationships', fontsize=14, fontweight='bold')

plt.tight_layout()
plt.show()
```





Interpreting the Linear Regression Visualisations

Actual vs. Predicted: The scatter should cluster along the red dashed diagonal. Any systematic deviation from this line (e.g., fanning out at higher values) indicates the model struggles in certain capacity factor ranges.

Error Distribution: A tight, symmetric bell curve centred at 0 means the model's errors are small and unbiased. Skewness or heavy tails would indicate systematic over- or under-prediction.

Seasonality Plot: If the dashed forecast line closely tracks the solid actual line over the week, the model has successfully learned the diurnal (daily sunrise-sunset) cycle. Gaps between the lines highlight where the linear assumption breaks down.

Correlation Heatmap: Irradiance is expected to have the strongest positive correlation with Capacity_Factor, since solar generation is directly driven by incoming sunlight. Temperature and Wind_Speed provide supplementary information about atmospheric conditions.

12. Training and Evaluation - Random Forest Regressor

Objective

Train a Random Forest Regressor (RFR) to predict solar capacity factor and compare its performance against the Linear Regression baseline.

Why Random Forest?

Random Forest is an ensemble method that builds multiple decision trees and averages their predictions. Unlike Linear Regression, it can capture non-linear relationships - such as the sharp transition between zero output (night) and active generation (day) - without requiring explicit feature engineering.

Hyperparameters

Parameter	Value	Purpose
`n_estimators`	100	Number of individual decision trees in the forest
`max_depth`	12	Maximum depth of each tree, preventing overfitting on noise
`random_state`	67	Ensures reproducibility
`n_jobs`	-1	Utilises all available CPU cores for parallel training

Why These Specific Values?

- **100 trees** provides a good balance between accuracy and training time. Increasing beyond 100 yields diminishing returns on this dataset.
- **max_depth=12** constrains each tree's complexity. Without this limit, trees could grow until every training sample is perfectly memorised (overfitting). A depth of 12 allows sufficient complexity to capture non-linear weather-generation relationships while keeping generalisation error low.
- **n_jobs=-1** is essential for handling 3.8 million training rows. Training 100 trees sequentially on a dataset this large would take significantly longer.

Data Partitioning

The same 80/20 train-test split with `random_state=67` is used to ensure a fair comparison with Linear Regression.

```
rfr_model = RandomForestRegressor(n_estimators=100, random_state=67, n_jobs=-1, max_depth=12)
rfr_model.fit(X_train, y_train)

rfr_train_pred = rfr_model.predict(X_train)
rfr_test_pred = rfr_model.predict(X_test)
```

Results

Metric	Training Set	Testing Set
Mean Absolute Error (MAE)	0.0278	0.0280
Root Mean Squared Error (RMSE)	0.0543	0.0547
R-Squared (R2)	0.9123	0.9112

Interpreting the Random Forest Results

The Random Forest Regressor significantly outperforms Linear Regression on this dataset due to its ability to model non-linear patterns.

Key observations:

- **Higher R2 (0.9112 vs. 0.7880):** The RFR explains 91.1% of variance in solar generation, compared to 78.8% for Linear Regression. This is a substantial improvement.
- **Lower MAE (0.0280 vs. 0.0534):** The average prediction error dropped by nearly half, from 5.3 percentage points to 2.8 percentage points of capacity factor.
- **Lower RMSE (0.0547 vs. 0.0845):** Large errors are also reduced, as shown by the RMSE improvement.
- **Train vs. Test Gap:** The gap between training and testing metrics is minimal (0.0011 in R2), indicating that the `max_depth=12` constraint successfully prevents overfitting.

> The Random Forest effectively learns decision boundaries like "if Irradiance > X and Hour is between 8 and 17, then the capacity factor is approximately Y" - rules that a linear model cannot express.

Model Comparison

Metric	Linear Regression	Random Forest	Improvement
MAE (Test)	0.0534	0.0280	47.6% lower
RMSE (Test)	0.0845	0.0547	35.3% lower
R2 (Test)	0.7880	0.9112	+12.3 points

---

13. Exporting the Random Forest Model

The trained Random Forest model is serialised and saved as `solar_model_rfr.pkl`.

```
1   joblib.dump(rfr_model, 'solar_model_rfr.pkl')
```

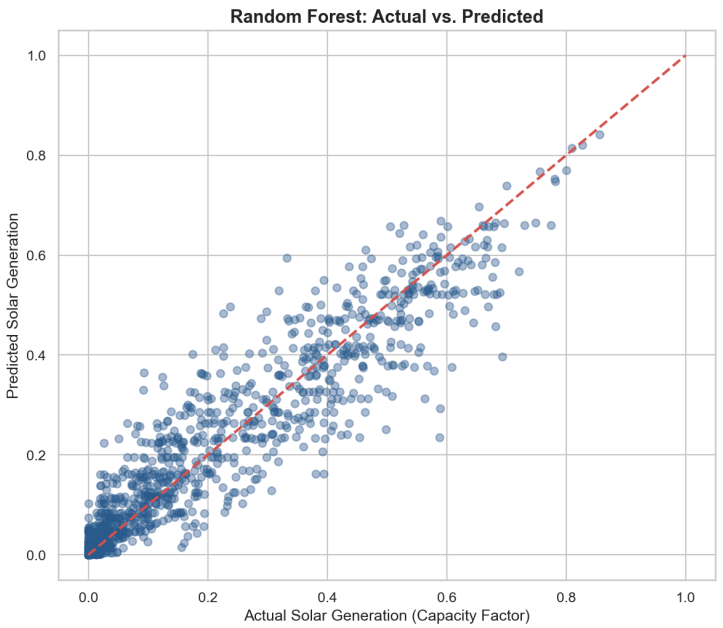
14. Analysis and Visualisation - Random Forest Regressor

The same four diagnostic plots are generated for the Random Forest model to enable direct comparison with Linear Regression.

Figure 1 - Model Accuracy and Error Analysis

- Actual vs. Predicted: Should show tighter clustering along the diagonal compared to Linear Regression.
- Error Distribution: Should show a narrower and more sharply peaked bell curve.

```
df_analysis_rfr = pd.read_csv('merged_encoded.csv')
df_analysis_rfr['Timestamp'] = pd.to_datetime(df_analysis_rfr['Timestamp'])

X = df_analysis_rfr.drop(columns=['Timestamp', 'Capacity_Factor'])
y = df_analysis_rfr['Capacity_Factor']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=67)

model_rfr = joblib.load('solar_model_rfr.pkl')

sns.set_theme(style="whitegrid")
fig1, axes1 = plt.subplots(1, 2, figsize=(16, 7))

sample_indices = np.random.choice(len(y_test), size=2000, replace=False)
y_test_sample = y_test.iloc[sample_indices]
y_pred_sample = model_rfr.predict(X_test.iloc[sample_indices])

axes1[0].scatter(y_test_sample, y_pred_sample, alpha=0.4, color='#2b5c8f')
axes1[0].plot([0, 1], [0, 1], '--', color='#d9534f', linewidth=2)
axes1[0].set_title('Random Forest: Actual vs. Predicted', fontsize=14,
fontweight='bold')
axes1[0].set_xlabel('Actual Solar Generation (Capacity Factor)')
axes1[0].set_ylabel('Predicted Solar Generation')

errors = y_test_sample - y_pred_sample
sns.histplot(errors, bins=50, kde=True, ax=axes1[1], color='#5cb85c')
axes1[1].set_title('Error Distribution (Residuals)', fontsize=14,
fontweight='bold')
axes1[1].set_xlabel('Prediction Error')
axes1[1].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```


Random Forest: Actual vs. Predicted
Predicted Solar Generation
Actual Solar Generation (Capacity Factor)
0.0
0.2
0.4
0.6
0.8
1.0

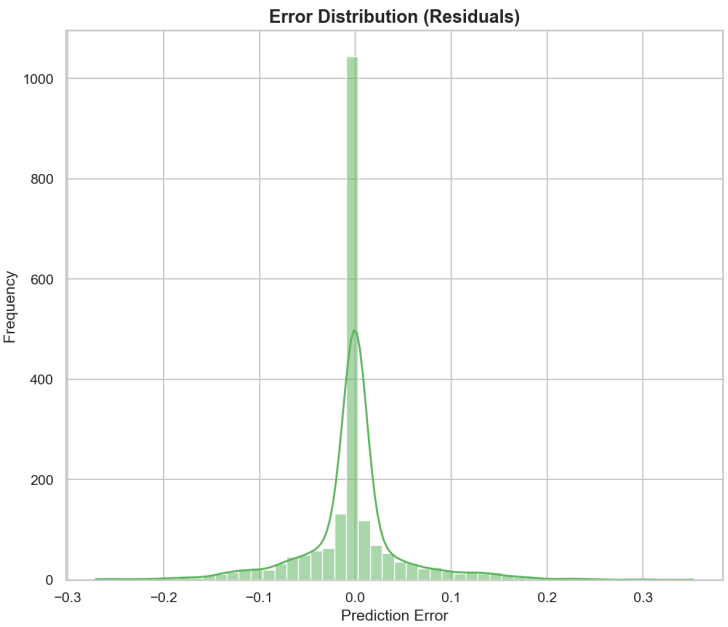
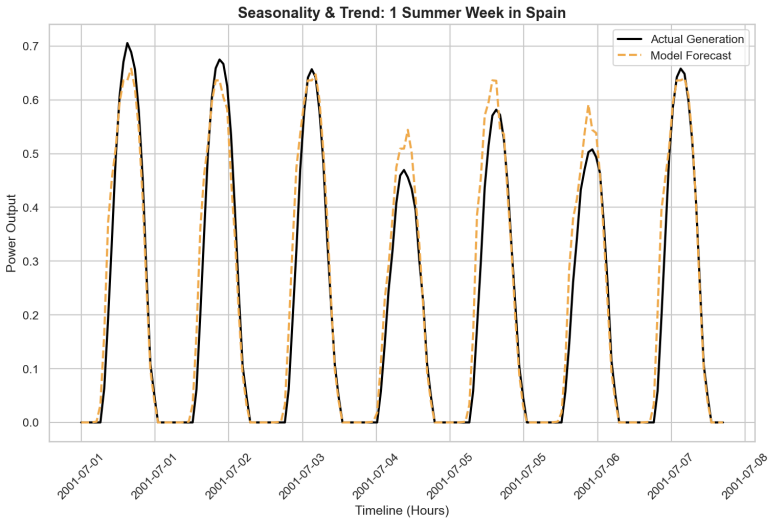
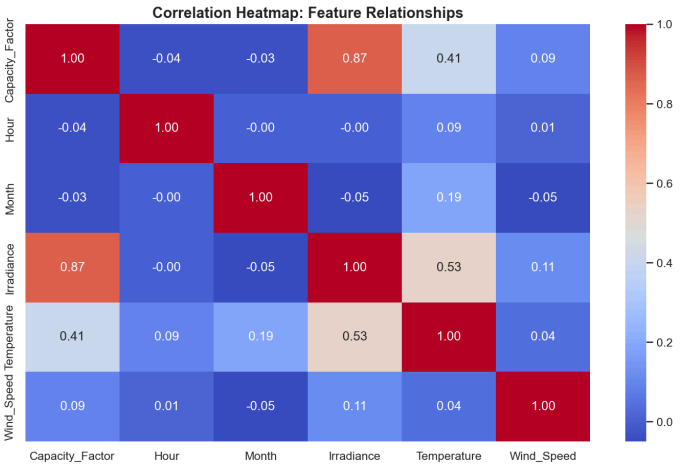

Error Distribution (Residuals)
Frequency
Prediction Error
0
200
400
600
800
1000
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3

Figure 2 - Trends and Feature Relationships

- **Seasonality (1 Summer Week in Spain):** The RFR forecast should track the actual daily peaks and troughs more closely.
- **Correlation Heatmap:** Identical to the LR version since it reflects data properties, not model properties.

```
fig2, axes2 = plt.subplots(1, 2, figsize=(20, 7))

spain_summer = df_analysis_rfr[
    (df_analysis_rfr['Country_ES'] == 1) & (df_analysis_rfr['Month'] == 7)
].head(168)
X_spain = spain_summer.drop(columns=['Timestamp', 'Capacity_Factor'])
y_true_spain = spain_summer['Capacity_Factor']
y_pred_spain = model_rfr.predict(X_spain)

axes2[0].plot(spain_summer['Timestamp'], y_true_spain, label='Actual Generation',
              color='black', linewidth=2)
axes2[0].plot(spain_summer['Timestamp'], y_pred_spain, label='Model Forecast',
              color='#f0ad4e', linestyle='dashed', linewidth=2)
axes2[0].set_title('Seasonality & Trend: 1 Summer Week in Spain', fontsize=14, fontweight='bold')
axes2[0].set_xlabel('Timeline (Hours)')
axes2[0].set_ylabel('Power Output')
axes2[0].xaxis.set_major_locator(ticker.MaxNLocator(nbins=10))
axes2[0].tick_params(axis='x', rotation=45)
axes2[0].legend()

cols_to_check = ['Capacity_Factor', 'Hour', 'Month', 'Irradiance', 'Temperature', 'Wind_Speed']
corr_matrix = df_analysis_rfr[cols_to_check].corr()

sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", ax=axes2[1], cbar=True)
axes2[1].set_title('Correlation Heatmap: Feature Relationships', fontsize=14, fontweight='bold')

plt.tight_layout()
plt.show()
```





Interpreting the Random Forest Visualisations

Actual vs. Predicted: Compared to the Linear Regression scatter, the Random Forest points should hug the diagonal more tightly, particularly at the extremes (very low and very high capacity factors). This reflects the model's ability to handle non-linear relationships.

Error Distribution: The residual histogram should be narrower and more sharply peaked at zero. The reduced spread indicates lower prediction error across the board.

Seasonality Plot: The Random Forest forecast line should overlay the actual generation curve more precisely, especially at the daily peaks and the sharp sunrise/sunset transitions that Linear Regression tends to smooth over.

Correlation Heatmap: This is data-dependent and remains identical across models. It confirms that Irradiance is the dominant predictor of solar output.

15. Custom Prediction Example

To demonstrate practical usage, the trained models are used to predict solar generation for a custom scenario. The process involves:

1. Creating an empty row with all 34 feature columns set to 0.
2. Filling in the desired weather conditions and time of day.
3. Setting the appropriate country column to 1 (simulating One-Hot Encoding).
4. Passing the row to both models for prediction.

Simulated Scenario

Parameter	Value	Description
Hour	13	1:00 PM - near solar noon
Month	7	July - peak summer
Irradiance	850.0 W/m^2	Strong sunlight
Temperature	32.0 C	Hot summer day
Wind_Speed	2.5 m/s	Light breeze
Country	ES (Spain)	Southern European location

```python
test_input = {
    'Hour': 13,
    'Month': 7,
    'Irradiance': 850.0,
    'Temperature': 32.0,
    'Wind_Speed': 2.5,
    'Country': 'Country_ES'
}

sample_row = pd.DataFrame(0, index=[0], columns=X.columns)

sample_row['Hour'] = test_input['Hour']
sample_row['Month'] = test_input['Month']
sample_row['Irradiance'] = test_input['Irradiance']
sample_row['Temperature'] = test_input['Temperature']
sample_row['Wind_Speed'] = test_input['Wind_Speed']

if test_input['Country'] in sample_row.columns:
    sample_row[test_input['Country']] = 1

lr_prediction = lr_model.predict(sample_row)[0]
rfr_prediction = rfr_model.predict(sample_row)[0]
```

Why This Scenario?

Spain at 1 PM in July with 850 W/m^2 irradiance represents a near-ideal solar generation scenario. This tests whether the models produce realistic high-output predictions. A well-trained model should predict a capacity factor in the range of 0.4 to 0.7 for these conditions, reflecting the practical efficiency limitations of real-world photovoltaic systems (which never reach the theoretical maximum of 1.0 due to inverter losses, panel degradation, and other factors).

16. Summary and Key Takeaways

This walkthrough covered the complete solar energy prediction pipeline:

Stage	Description
Data Loading	Loaded the EMHIRES PV capacity factor dataset (29 countries, 1986-2015)
Visualisation	Plotted a single day's solar cycle to build intuition about the data
NASA Data Reference	Documented the weather data collection process via the NASA POWER API
Data Cleaning	Verified the absence of null values and sentinel flags in the raw data
Restructuring	Melted the wide format data into long format, created timestamps, extracted Hour and Month features
Merging	Fused generation data with NASA weather data using a composite Timestamp+Country key
Encoding	Applied One-Hot Encoding to convert country labels into 29 binary features
Linear Regression	Trained and evaluated a linear model as the baseline
Random Forest	Trained and evaluated a non-linear ensemble model for comparison
Analysis	Generated diagnostic visualisations (scatter plots, residuals, seasonality, correlation heatmaps) for both models
Model Export	Saved both trained models as `.pkl` files for deployment

Final Model Comparison

Metric	Linear Regression	Random Forest Regressor
MAE (Test)	0.0534	0.0280
RMSE (Test)	0.0845	0.0547
R2 (Test)	0.7880	0.9112

> The Random Forest Regressor consistently outperforms Linear Regression on this dataset because solar generation exhibits non-linear dependencies on time-of-day, weather conditions, and geographic location. The linear model provides a useful interpretable baseline, while the Random Forest captures the complex interactions needed for accurate forecasting.

Why Both Models?

Keeping both models is deliberate:

- Linear Regression serves as an interpretable baseline. Its coefficients can be directly inspected to understand the marginal effect of each feature (e.g., "each additional W/m^2 of irradiance increases capacity factor by X"). This is valuable for explainability and stakeholder communication.
- Random Forest Regressor serves as the production-grade model. Its superior accuracy makes it the better choice for deployment in forecasting systems where prediction precision matters more than interpretability.

Comparing the two models also validates that the dataset contains genuine non-linear patterns. If both models performed identically, it would suggest the relationships are purely linear - and that would be a surprising (and incorrect) finding for solar physics.