

weather-prediction

May 15, 2025

```
[ ]: %pip install torch tqdm matplotlib seaborn tqdm
```

```
[2]: import requests
import datetime
import time
import json
import csv
import pandas as pd
import numpy as np
from pprint import pprint
import seaborn as sns
from tqdm import tqdm

import matplotlib.pyplot as plt

from dateutil.relativedelta import relativedelta

from sklearn.model_selection import train_test_split, TimeSeriesSplit, RandomizedSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
```

1 Obtener los datos de la API de AEMET

Para descargar los datos de la API de AEMET, primero debemos registrarnos y obtener una clave de acceso. Una vez que tengamos la clave, podemos utilizar la biblioteca `requests` para realizar solicitudes a la API.

Vamos a intentar descargar los datos 5 veces con un delay de 5 segundos entre cada intento. Si después de 5 intentos no conseguimos los datos, se lanzará una excepción. Hacemos esto porque la API de AEMET nos ha dado problemas de conexión en multiples ocasiones.

```
[ ]: API_KEY = "eyJhbGciOiJIUzI1NiJ9.
    ↪eyJzdWIiOiJvc2N1cm9tZXRLb3JvQGdtYWlsLmNvbSIsImp0aSI6ImQ4YjQ3YmVjLWQ5MWMtNGRhMy1hNjNjLTRlZjI
    ↪w_x6_x8-lfdn9L3Hho-j0FgQBoZnqj4qumE5yBb_FGg"
download_data = False
output_file = "datos_tiempo.json"
```

```
[4]: def fetch_interval_data(api_key, station, start_dt, end_dt, max_retries=5,
    ↪delay=5):
    """
    Para el intervalo de fechas [start_dt, end_dt], construye la URL,
    llama a la API de AEMET, luego usa la URL "datos" para recuperar los datos
    meteorológicos reales con reintentos.
    """
    start_str = start_dt.strftime("%Y-%m-%dT00:00:00UTC")
    end_str = end_dt.strftime("%Y-%m-%dT23:59:59UTC")

    url = (
        f"https://opendata.aemet.es/opendata/api/valores/climatologicos/diarios/
    ↪datos/"
        f"fechaini/{start_str}/fechafin/{end_str}/estacion/{station}/"
        f"?api_key={api_key}"
    )

    for attempt in range(max_retries):
        print(f"Consulta de {start_str} a {end_str} (Intentos {attempt + 1}/
    ↪{max_retries})...")
        try:
            response = requests.get(url)
            response.raise_for_status() # Raise HTTPError for bad responses
    ↪(4xx or 5xx)

            initial_data = response.json()

            if initial_data.get("estado") != 200 or initial_data.
    ↪get("descripcion") != "exito":
                print(
                    f"La API ha reportado un error {start_str} - {end_str}:",
                    initial_data,
                )
                return []

            datos_url = initial_data.get("datos")
            if not datos_url:
                print("No se ha encontrado la clave 'datos' en la respuesta.")
                return []

            datos_response = requests.get(datos_url)
```

```

        datos_response.raise_for_status() # Raise HTTPError for bad
↳responses (4xx or 5xx)

        real_data = datos_response.json()
        return real_data

    except requests.exceptions.ConnectionError as e:
        print(f"Error de conexión: {e}. Reintentando en {delay} segundos...
↳")
        time.sleep(delay)
    except requests.exceptions.RequestException as e:
        print(f"Error en la solicitud: {e}")
        return [] # Or handle other request errors as needed
    except Exception as e:
        print(f"Ha ocurrido un error inesperado: {e}")
        return [] # Handle other potential errors

    print(f"Ha ocurrido un error al intentar recuperar los datos de {start_str}
↳a {end_str}.")
    return []

```

```

[5]: # A veces, en databricks, el obtener la información no es posible, si ejecutas
↳esta celda en local, es posible, y el resultado es el JSON que viene debajo
if download_data:
    # Obten esto del csv "diccionario24.csv"
    station = "3170Y"

    # Define la fecha de inicio y la fecha final de los datos que vamos a
↳recolectar para entrenar los modelos.
    # Se ha escogido la fecha de 2009 porque es la primera en la que la API
↳devuelve datos.
    global_start = datetime.datetime(2009, 1, 1)
    global_end = datetime.datetime(2025, 4, 28)
    merged_data = []

    current_date = global_start
    while current_date <= global_end:
        potential_end = current_date + relativedelta(months=+6) - datetime.
↳timedelta(days=1)
        segment_end = min(potential_end, global_end)

        interval_data = fetch_interval_data(API_KEY, station, current_date,
↳segment_end)
        if interval_data:
            if isinstance(interval_data, list):
                merged_data.extend(interval_data)

```

```

        else:
            merged_data.append(interval_data)
    else:
        print(
            f"No hay datos obtenidos para el periodo {current_date.date()}_
↪to "
            f"{segment_end.date()})."
        )

    current_date = current_date + relativedelta(months=+6)

    # Pausa para eliminar los errores de la API "too many requests"
    time.sleep(1)

    # Escribimos los datos en un archivo JSON
    try:
        with open(output_file, "w", encoding="utf-8") as f:
            json.dump(merged_data, f, indent=2, ensure_ascii=False)
        print(f"Datos juntados en {output_file}.")
    except Exception as e:
        print("Error juntando los datos:", e)

```

```

[6]: if not download_data:
    # Si no se ha podido descargar el JSON, lo bajamos de la nube
    !wget -O {output_file} https://4sd1h7eymb.ufs.sh/f/
↪k5bvuwMnPr07YbdYIPxUMQfASp0wD4IXkN6FoZCtHjm5ryWL

```

```

--2025-05-15 18:34:55--
https://4sd1h7eymb.ufs.sh/f/k5bvuwMnPr07YbdYIPxUMQfASp0wD4IXkN6FoZCtHjm5ryWL
Resolving 4sd1h7eymb.ufs.sh (4sd1h7eymb.ufs.sh)... 104.21.16.1, 104.21.32.1,
104.21.80.1, ...
Connecting to 4sd1h7eymb.ufs.sh (4sd1h7eymb.ufs.sh)|104.21.16.1|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 2444931 (2.3M) [application/json]
Saving to: 'datos_tiempo.json'

datos_tiempo.json  100%[=====>]    2.33M  5.87MB/s   in 0.4s

2025-05-15 18:34:55 (5.87 MB/s) - 'datos_tiempo.json' saved [2444931/2444931]

```

1.1 Convertir los datos de JSON a CSV

Convertimos los datos obtenidos de la API (en formato JSON) a un archivo CSV para poder trabajar con ellos más fácilmente. Utilizamos la biblioteca `csv`.

```
[7]: # Input JSON file
with open(output_file, 'r', encoding='utf-8') as json_file:
    input_json = json.load(json_file)

# Output CSV file
output_csv = output_file.replace('.json', '.csv')

# Get all possible fieldnames from all dictionaries in the JSON
all_fieldnames = set()
for item in input_json:
    all_fieldnames.update(item.keys())

# Convert JSON to CSV
with open(output_csv, mode='w', newline='', encoding='utf-8') as csv_file:
    writer = csv.DictWriter(
        csv_file,
        fieldnames=list(all_fieldnames),
        restval=-999, # Default value for missing fields
        extrasaction='ignore', # Ignore extra fields not in fieldnames
        delimiter=';' # Use semicolon as separator,
    )
    writer.writeheader()
    writer.writerows(input_json)

print(f"Los datos en formato JSON han sido convertidos a {output_csv}")
```

Los datos en formato JSON han sido convertidos a datos_tiempo.csv

```
[8]: # Reemplazar comas por puntos y punto y coma por comas para que luego al usar
      ↪ pandas no de problemas
with open(output_csv, 'r', encoding='utf-8') as file:
    data = file.read()
data = data.replace(',', '.')
data = data.replace('; ', ', ')
with open(output_csv, 'w', encoding='utf-8') as file:
    file.write(data)
```

1.2 Comprobar las fechas que faltan en los datos

Iteramos sobre las fechas de los datos y comprobamos si hay alguna fecha que falta. Si encontramos alguna fecha que falta, la añadimos a una lista y cuantos días faltan.

```
[9]: # Carga el CSV
data = pd.read_csv(output_csv)

data['fecha'] = pd.to_datetime(data['fecha'], errors='coerce')
data = data.sort_values(by='fecha')
data['gap'] = data['fecha'].diff().dt.days
```

```

gaps = data[data['gap'] > 1]

if not gaps.empty:
    print("Los huecos encontrados en estos datos son:")
    print(f"La suma de los días en los que no hay datos es de: {int(gaps['gap'].
    ↪sum())}/{len(data)} días: un {gaps['gap'].sum()/len(data)*100:.2f}%")
    print(gaps[['fecha', 'gap']])
else:
    print("No se han encontrado huecos en los datos.")

```

Los huecos encontrados en estos datos son:

La suma de los días en los que no hay datos es de: 734/5245 días: un 13.99%

| | fecha | gap |
|------|------------|-------|
| 65 | 2009-04-10 | 5.0 |
| 119 | 2009-06-04 | 2.0 |
| 224 | 2009-09-18 | 2.0 |
| 227 | 2009-09-22 | 2.0 |
| 270 | 2009-11-05 | 2.0 |
| 273 | 2009-11-09 | 2.0 |
| 408 | 2010-04-14 | 22.0 |
| 498 | 2010-07-15 | 3.0 |
| 532 | 2010-08-19 | 2.0 |
| 537 | 2010-08-28 | 5.0 |
| 569 | 2010-09-30 | 2.0 |
| 673 | 2011-01-19 | 8.0 |
| 803 | 2011-06-09 | 12.0 |
| 1375 | 2013-07-01 | 182.0 |
| 1433 | 2013-09-03 | 7.0 |
| 1759 | 2014-07-27 | 2.0 |
| 1761 | 2014-07-31 | 3.0 |
| 1826 | 2014-10-05 | 2.0 |
| 1827 | 2014-10-07 | 2.0 |
| 1855 | 2014-11-05 | 2.0 |
| 1871 | 2014-11-22 | 2.0 |
| 1967 | 2015-02-27 | 2.0 |
| 1999 | 2015-04-01 | 2.0 |
| 2057 | 2015-06-11 | 14.0 |
| 2167 | 2015-09-30 | 2.0 |
| 2172 | 2015-10-06 | 2.0 |
| 2185 | 2015-10-20 | 2.0 |
| 2194 | 2015-11-03 | 6.0 |
| 2198 | 2015-11-09 | 3.0 |
| 2200 | 2015-11-12 | 2.0 |
| 2201 | 2016-03-30 | 139.0 |
| 2263 | 2016-06-16 | 17.0 |
| 2319 | 2016-08-25 | 15.0 |
| 2399 | 2016-12-15 | 33.0 |
| 2400 | 2016-12-17 | 2.0 |

| | | |
|------|------------|------|
| 2401 | 2016-12-19 | 2.0 |
| 2408 | 2016-12-28 | 3.0 |
| 2409 | 2016-12-31 | 3.0 |
| 2420 | 2017-01-12 | 2.0 |
| 2436 | 2017-02-10 | 14.0 |
| 2477 | 2017-05-11 | 50.0 |
| 2488 | 2017-07-27 | 67.0 |
| 2501 | 2017-08-25 | 17.0 |
| 2512 | 2017-10-25 | 51.0 |
| 3203 | 2019-09-19 | 4.0 |
| 3220 | 2019-10-07 | 2.0 |
| 3221 | 2019-10-09 | 2.0 |
| 3304 | 2020-01-01 | 2.0 |
| 3597 | 2020-10-22 | 3.0 |

2 Transformar los datos y limpieza de los mismos

Los datos obtenidos de la API pueden contener información innecesaria o estar en un formato que no es adecuado para nuestro análisis. Por lo tanto, es importante limpiar y transformar los datos antes de utilizarlos. Esto puede incluir eliminar columnas innecesarias, cambiar el formato de las fechas, etc.

Empezamos eliminando las columnas que tengan un valor de -999 en cualquier columna (valor que hemos puesto si no existía el dato en pasos anteriores). También eliminamos las columnas que tengan o no un valor, es decir, que tengan un valor `None`.

Por último, eliminamos las columnas `nombre`, `provincia`, `indicativo`, `altitud` ya que no son necesarias para nuestro análisis y cambiamos el dato de la fecha para tener un día del año.

```
[10]: data = pd.read_csv(output_csv)

all_fields = data.columns.tolist()
print(f"Número total de días: {len(all_fields)}")
pprint(all_fields, compact=True)

print(f"Número total de filas: {len(data)} antes de eliminar las filas en las_
      ↳ que faltan algunos datos:")
data = data[~(data == -999).any(axis=1)]
data = data[~(data == "-999").any(axis=1)]
print(f"Número de filas que quedan después de eliminar las filas que contienen_
      ↳ un -999: {len(data)}")
data = data.dropna()
print(f"Número de filas que quedan después de eliminar las filas que contienen_
      ↳ un NaN: {len(data)}")

data = data.drop(columns=["nombre", "provincia", "indicativo", "altitud"],_
      ↳ axis=1)
```

```
# data = data[~(data == "Varias").any(axis=1)]
# print(f"Numeros de filas que quedan después de eliminar las filas que
↳ contienen 'Varias': {len(data)}")
# Convert 'fecha' to datetime if not already
data['fecha'] = pd.to_datetime(data['fecha'], errors='coerce')

data['day'] = data['fecha'].dt.dayofyear
```

Número total de días: 20

```
['hrMin', 'velmedia', 'horatmin', 'nombre', 'tmax', 'tmed', 'hrMedia',
 'indicativo', 'provincia', 'fecha', 'altitud', 'horaracha', 'prec', 'horatmax',
 'hrMax', 'horaHrMax', 'racha', 'dir', 'horaHrMin', 'tmin']
```

Número total de filas: 5245 antes de eliminar las filas en las que faltan algunos datos:

Número de filas que quedan después de eliminar las filas que contienen un -999: 5090

Número de filas que quedan después de eliminar las filas que contienen un NaN: 5090

También vamos a eliminar las columnas de tiempos, ya que solo nos interesa la temperatura media, alta y baja del día, no la hora en la que se ha registrado esa temperatura

```
[11]: data = data.drop(columns=['horatmin', 'horatmax', 'horaHrMax', 'horaracha',
↳ 'horaHrMin'], axis=1)
```

2.1 Generar los datos de los últimos N días

Generamos los datos de los últimos N días a partir de la fecha actual. Esto nos permitirá tener una visión más clara de cómo ha evolucionado la temperatura en los últimos días. Porque no solo nos interesa la temperatura que ha hecho el día de antes, queremos ver un poco la evolución de la temperatura en los últimos días. Por eso hemos creado columnas nuevas con los datos del enésimo día anterior: Si tenemos la columna `dir`, vamos a añadir las columnas `dir_1`, `dir_2`, `dir_3` que contendrán los datos de la dirección del viento de hace 1, 2 y 3 días respectivamente.

Esto lo vamos a hacer para todas las columnas excepto la fecha o el día.

```
[12]: def derive_nth_day_feature(df: pd.DataFrame, feature: str, N: int):
        col_name = f"{feature}_{N}"
        df[col_name] = df[feature].shift(N)
        return df

features = data.columns.tolist()
features.remove('fecha')
features.remove('day')

# Crear nuevas columnas para las temperaturas de los días anteriores
for feature in features:
    for N in range(1, 4):
        data = derive_nth_day_feature(data, feature, N)
```



```

# Filtra las columnas con datos del mismo día
# Mantener solo las columnas que terminen con _1, _2, _3
columns_to_keep = []
for col in data.columns:
    if '_' in col or col == 'fecha' or col == 'day' or col == 'tmed':
        columns_to_keep.append(col)
    elif any(col.endswith(f'_{i}') for i in range(1, 4)):
        columns_to_keep.append(col)

data = data[columns_to_keep]

data = data.dropna()

print(f"Forma final del dataset: {data.shape}")
print("Columnas en el dataset:")
pprint(data.columns.tolist(), compact=True)

```

Forma final del dataset: (5087, 33)

Columnas en el dataset:

```

['tmed', 'fecha', 'day', 'hrMin_1', 'hrMin_2', 'hrMin_3', 'velmedia_1',
 'velmedia_2', 'velmedia_3', 'tmax_1', 'tmax_2', 'tmax_3', 'tmed_1', 'tmed_2',
 'tmed_3', 'hrMedia_1', 'hrMedia_2', 'hrMedia_3', 'prec_1', 'prec_2', 'prec_3',
 'hrMax_1', 'hrMax_2', 'hrMax_3', 'racha_1', 'racha_2', 'racha_3', 'dir_1',
 'dir_2', 'dir_3', 'tmin_1', 'tmin_2', 'tmin_3']

```

```

[13]: # Guarda el CSV limpio
output_clean_csv = output_file.replace('.json', '-limpios.csv')
data.to_csv(output_clean_csv, index=False)
print(f"Cleanead data has been saved to {output_clean_csv}")

```

Cleanead data has been saved to datos_tiempo-limpios.csv

2.2 Comprobar cuales son los parámetros importantes

Comprobamos, entre las columnas que tenemos, cuales son los parámetros más importantes para nuestro análisis. Para ello, vamos a utilizar la biblioteca `pandas` y la función `corr()` para calcular la correlación entre las diferentes columnas. Esto nos permitirá ver qué columnas están más relacionadas entre sí y cuáles son las más importantes para nuestro análisis.

```

[14]: data = pd.read_csv(output_clean_csv, parse_dates=['fecha'], dayfirst=True)
data['fecha'] = pd.to_datetime(data['fecha'], format='%Y-%m-%d',
    ↪errors='coerce')
data = data.dropna(subset=['fecha'])
data = data.sort_values(by='fecha')

# Convertir la columna 'fecha' a un índice de fecha
data.set_index('fecha', inplace=True)

```

```

data = data.asfreq('D')

# Convertir las columnas tmin, racha tmax velmedia altitud prec tmed a float
columns_to_convert = data.columns.tolist()
for col in columns_to_convert:
    data[col] = pd.to_numeric(data[col], errors='coerce')

data.info()

```

```

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5926 entries, 2009-02-04 to 2025-04-26
Freq: D
Data columns (total 32 columns):

```

| # | Column | Non-Null Count | Dtype |
|----|------------|----------------|---------|
| 0 | tmed | 5087 non-null | float64 |
| 1 | day | 5087 non-null | float64 |
| 2 | hrMin_1 | 5087 non-null | float64 |
| 3 | hrMin_2 | 5087 non-null | float64 |
| 4 | hrMin_3 | 5087 non-null | float64 |
| 5 | velmedia_1 | 5087 non-null | float64 |
| 6 | velmedia_2 | 5087 non-null | float64 |
| 7 | velmedia_3 | 5087 non-null | float64 |
| 8 | tmax_1 | 5087 non-null | float64 |
| 9 | tmax_2 | 5087 non-null | float64 |
| 10 | tmax_3 | 5087 non-null | float64 |
| 11 | tmed_1 | 5087 non-null | float64 |
| 12 | tmed_2 | 5087 non-null | float64 |
| 13 | tmed_3 | 5087 non-null | float64 |
| 14 | hrMedia_1 | 5087 non-null | float64 |
| 15 | hrMedia_2 | 5087 non-null | float64 |
| 16 | hrMedia_3 | 5087 non-null | float64 |
| 17 | prec_1 | 5087 non-null | float64 |
| 18 | prec_2 | 5087 non-null | float64 |
| 19 | prec_3 | 5087 non-null | float64 |
| 20 | hrMax_1 | 5087 non-null | float64 |
| 21 | hrMax_2 | 5087 non-null | float64 |
| 22 | hrMax_3 | 5087 non-null | float64 |
| 23 | racha_1 | 5087 non-null | float64 |
| 24 | racha_2 | 5087 non-null | float64 |
| 25 | racha_3 | 5087 non-null | float64 |
| 26 | dir_1 | 5087 non-null | float64 |
| 27 | dir_2 | 5087 non-null | float64 |
| 28 | dir_3 | 5087 non-null | float64 |
| 29 | tmin_1 | 5087 non-null | float64 |
| 30 | tmin_2 | 5087 non-null | float64 |
| 31 | tmin_3 | 5087 non-null | float64 |

```
dtypes: float64(32)
memory usage: 1.5 MB
```

Con la función `corr()` podemos ver la correlación entre las diferentes columnas con respecto a la temperatura media `tmed`.

```
[15]: data.corr()[['tmed']].abs().sort_values('tmed')
```

```
[15]:
```

| | tmed |
|------------|----------|
| dir_1 | 0.000140 |
| dir_2 | 0.005786 |
| dir_3 | 0.009321 |
| prec_3 | 0.091201 |
| prec_2 | 0.092970 |
| prec_1 | 0.093921 |
| velmedia_1 | 0.147602 |
| velmedia_2 | 0.153739 |
| racha_1 | 0.176867 |
| racha_2 | 0.177169 |
| velmedia_3 | 0.181227 |
| racha_3 | 0.202118 |
| day | 0.207439 |
| hrMin_1 | 0.627331 |
| hrMin_3 | 0.631258 |
| hrMin_2 | 0.631863 |
| hrMax_3 | 0.679317 |
| hrMax_2 | 0.687778 |
| hrMax_1 | 0.702521 |
| hrMedia_3 | 0.707277 |
| hrMedia_2 | 0.709740 |
| hrMedia_1 | 0.711071 |
| tmin_3 | 0.846604 |
| tmin_2 | 0.866324 |
| tmax_3 | 0.892796 |
| tmin_1 | 0.902454 |
| tmed_3 | 0.907520 |
| tmax_2 | 0.916948 |
| tmed_2 | 0.930634 |
| tmax_1 | 0.948360 |
| tmed_1 | 0.965384 |
| tmed | 1.000000 |

Seleccionamos las columnas que tienen una correlación mayor a 0.5 con `tmed`

```
[16]: correlated_columns = data.corr()[['tmed']].abs().sort_values('tmed',
    ↪ascending=False)
correlated_columns = correlated_columns[correlated_columns['tmed'] > 0.5].index.
    ↪tolist()
```

```

correlated_columns.remove('tmed') # Remove 'tmed' itself from the list
pprint(f"Las columnas que tienen una correlación mayor a 0.5 con tmed son:␣
↳{correlated_columns}")
print()
non_correlated_columns = data.corr()[['tmed']].abs().sort_values('tmed',␣
↳ascending=False)
non_correlated_columns = non_correlated_columns[non_correlated_columns['tmed']␣
↳< 0.5].index.tolist()
pprint(f"Las columnas que tienen una correlación menor a 0.5 con tmed son:␣
↳{sorted(non_correlated_columns)}")

```

```

("Las columnas que tienen una correlación mayor a 0.5 con tmed son: ['tmed_1', "
 "'tmax_1', 'tmed_2', 'tmax_2', 'tmed_3', 'tmin_1', 'tmax_3', 'tmin_2', "
 "'tmin_3', 'hrMedia_1', 'hrMedia_2', 'hrMedia_3', 'hrMax_1', 'hrMax_2', "
 "'hrMax_3', 'hrMin_2', 'hrMin_3', 'hrMin_1']")

```

```

("Las columnas que tienen una correlación menor a 0.5 con tmed son: ['day', "
 "'dir_1', 'dir_2', 'dir_3', 'prec_1', 'prec_2', 'prec_3', 'racha_1', "
 "'racha_2', 'racha_3', 'velmedia_1', 'velmedia_2', 'velmedia_3']")

```

2.2.1 Resultados de la correlación

De los datos obtenidos de hacer la correlación con la temperatura media del día, podemos ver que los parámetros que menos influyen en esta son `day`, `dir`, `year`, `prec`, `racha` y `velmedia`. Por eso, descartaremos estos datos a la hora de entrenar nuestro modelo de predicción.

Por decirlo de otra manera, los parámetros que más influyen en la temperatura media del día son: `hrMin`, `hrMax`, `hrMedia`, `tmin` y `tmax`.

Filtramos los datos para quedarnos solo con estos parámetros y los guardamos en un nuevo dataframe. Para ver los resultados, vamos a crear algunas gráficas con los datos obtenidos. Vamos a utilizar la biblioteca `matplotlib` para crear las gráficas y ver cómo se comportan los datos.

2.2.2 Gráficas

La primera gráfica que vamos a crear es una matriz de correlación entre los diferentes parámetros. Esto nos permitirá ver qué parámetros están más relacionados entre sí y cuáles son los más importantes para nuestro análisis. Es otra manera de ver los resultados de la correlación que hemos hecho antes.

```

[17]: data_filtered = data[correlated_columns + ['tmed']].copy()
data_filtered = data_filtered.dropna()

# Visualizar las relaciones entre las variables
def plot_correlation_matrix(data):
    """
    Plots the correlation matrix of the given DataFrame.
    """
    plt.figure(figsize=(10, 8))

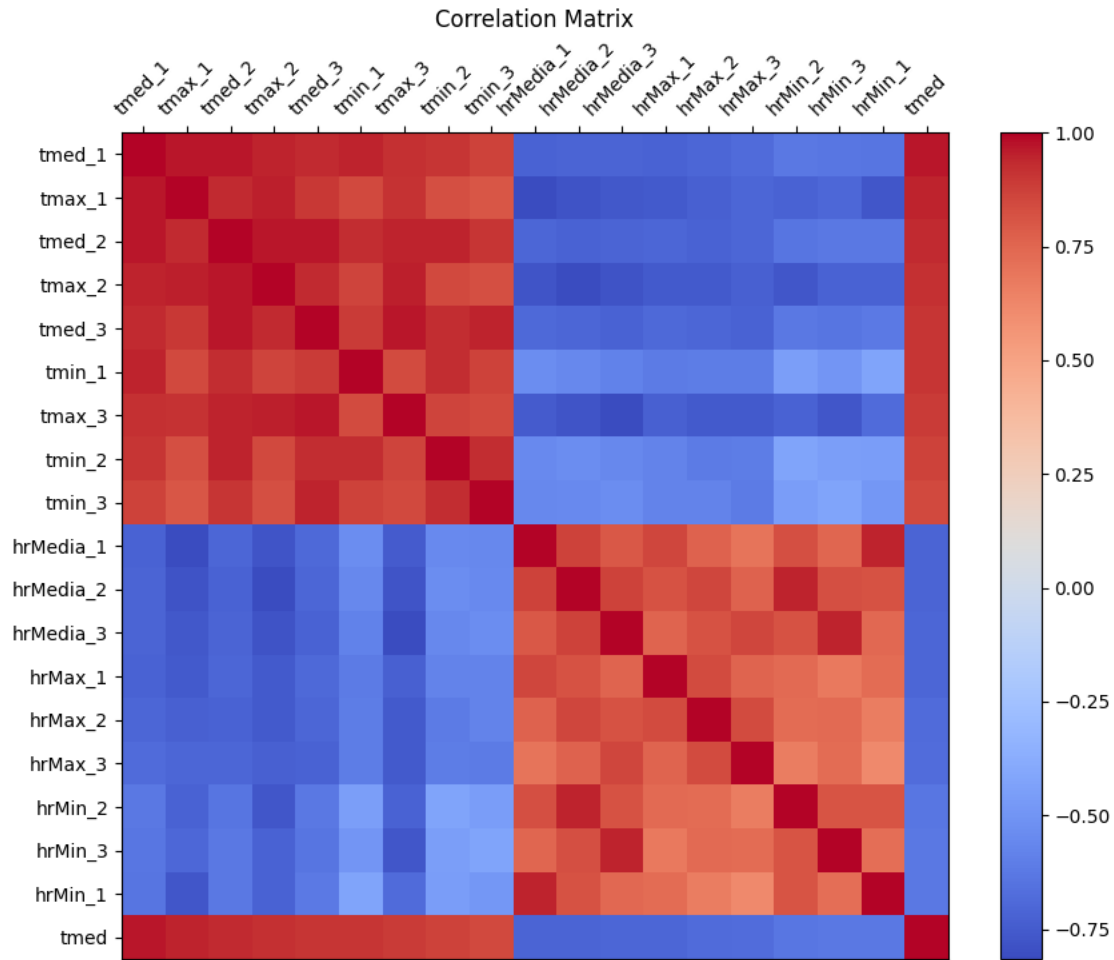
```

```

corr = data.corr()
plt.matshow(corr, cmap='coolwarm', fignum=1)
plt.colorbar()
plt.xticks(range(len(corr.columns)), corr.columns, rotation=45)
plt.yticks(range(len(corr.columns)), corr.columns)
plt.title('Correlation Matrix')
plt.show()

plot_correlation_matrix(data_filtered)

```



2.3 Visualización de la Relación entre Variables y Temperatura Media

En esta sección, creamos gráficos de dispersión para visualizar la relación entre cada variable y la temperatura media (tmed). Esto nos permite:

1. Identificar correlaciones visuales entre las variables predictoras y la temperatura media
2. Detectar posibles relaciones no lineales

3. Identificar patrones o clusters en los datos
4. Evaluar la dispersión y distribución de los puntos

Se genera una matriz de 7x3 subgráficos, donde cada uno representa la relación entre una variable predictora y la temperatura media. Los puntos más agrupados o que siguen una tendencia clara indican una mayor correlación con la temperatura media.

```
[18]: %matplotlib inline

# Creamos la figura donde van a ir los gráficos
plt.rcParams['figure.figsize'] = [16, 22]

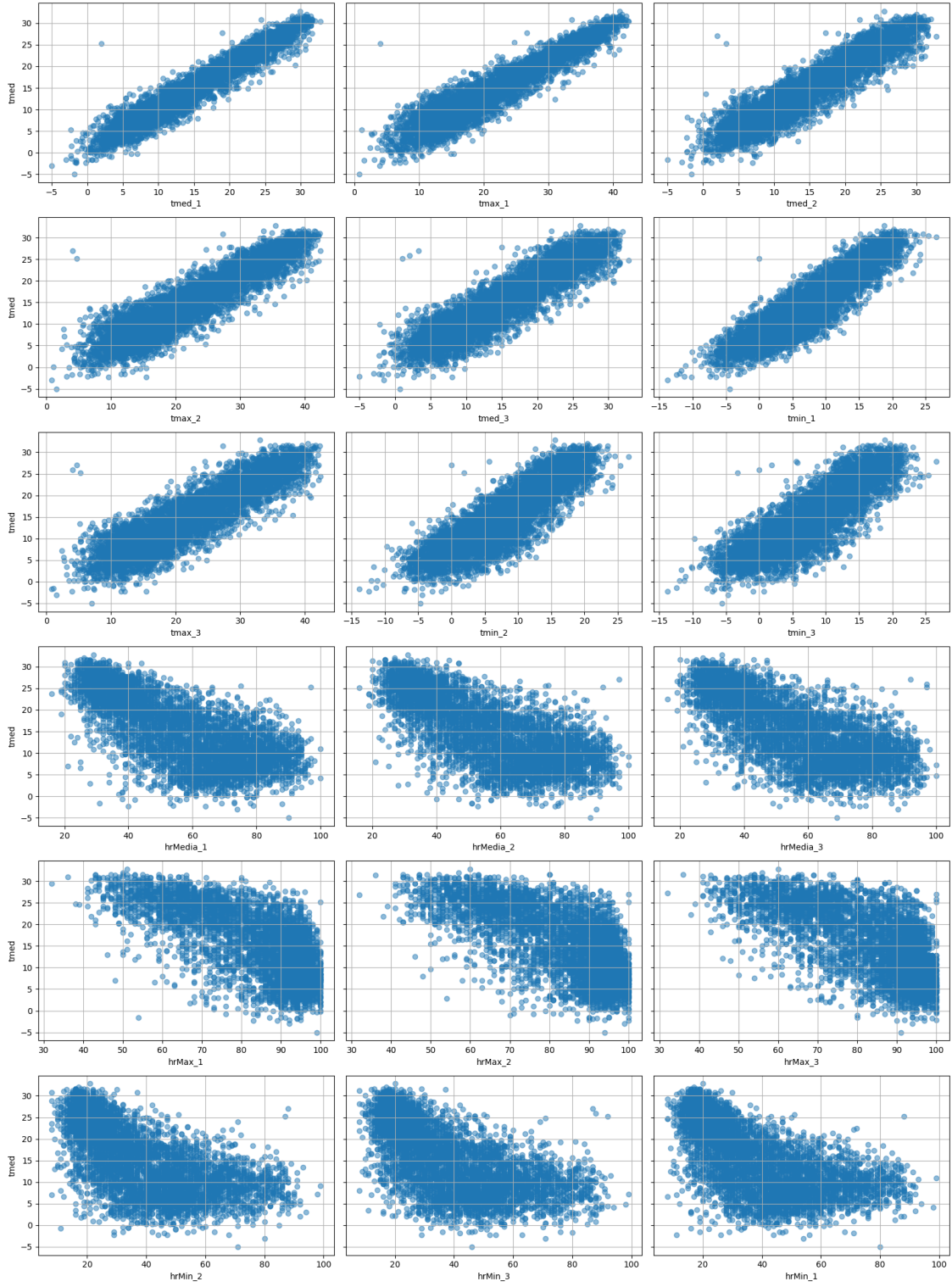
# Creamos las sub-plots
fig, axes = plt.subplots(nrows=6, ncols=3, sharey=True)

# Identifica las características a graficar contra tmed (variable objetivo)
features = data_filtered.columns.tolist()
axes_flat = axes.flatten()

# Crea las scatter plots
for idx, feature in enumerate(features):
    if feature == 'tmed':
        continue # Skip the target variable itself
    axes_flat[idx].scatter(data[feature], data['tmed'], alpha=0.5)
    axes_flat[idx].set_xlabel(feature)
    axes_flat[idx].set_ylabel('tmed' if idx % 3 == 0 else '')
    axes_flat[idx].grid(True)

plt.suptitle('Scatter Plots de cada una de las columnas vs temperatura media_↵
↵(tmed)', y=1.02, fontsize=16)
plt.tight_layout()
plt.show()
```

Scatter Plots de cada una de las columnas vs temperatura media (tmed)



Vamos a guardar las columnas que tienen una correlación mayor a 0.5 con tmed en una lista para poder utilizarlas más adelante en nuestro análisis en un archivo CSV que termina en `filtradas.csv`.

```
[19]: # Guardar los datos filtrados en un nuevo CSV
output_filtered_csv = output_file.replace('.json', '-filtradas.csv')
data_filtered.to_csv(output_filtered_csv, index=True)
print(f"Filtered data has been saved to {output_filtered_csv}")
```

Filtered data has been saved to `datos_tiempo-filtradas.csv`

3 Modelo de regresión lineal

Ahora que ya tenemos los datos filtrados, limpios, y solo las features que nos interesan, vamos a entrenar el modelo de predicción. Para ello, vamos a utilizar la biblioteca `scikit-learn` y el algoritmo de regresión lineal.

```
[20]: data = pd.read_csv(output_filtered_csv, parse_dates=['fecha'],
    ↪index_col='fecha')
data.columns
```

```
[20]: Index(['tmed_1', 'tmax_1', 'tmed_2', 'tmax_2', 'tmed_3', 'tmin_1', 'tmax_3',
    'tmin_2', 'tmin_3', 'hrMedia_1', 'hrMedia_2', 'hrMedia_3', 'hrMax_1',
    'hrMax_2', 'hrMax_3', 'hrMin_2', 'hrMin_3', 'hrMin_1', 'tmed'],
    dtype='object')
```

3.1 Dividir los datos en train, test y val

Para entrenar un modelo de machine learning, es importante dividir los datos en tres conjuntos: train, test y val. Esto nos permitirá entrenar el modelo con un conjunto de datos, validarlo con otro conjunto y probarlo con un tercer conjunto. De esta manera, podremos evaluar el rendimiento del modelo y evitar el sobreajuste. Las particiones que vamos a hacer son las siguientes: - Train: 70% - Test: 15% - Val: 15%

```
[21]: # Crea los datasets de entrenamiento, validación y test
X = data.drop('tmed', axis=1)
y = data['tmed']

# Divide los datos en conjuntos de entrenamiento y test
X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=0.15,
    ↪shuffle=False)

# Divide los datos de entrenamiento en conjuntos de entrenamiento y validación
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=0.
    ↪165, shuffle=False)

print(f"Training set size: {len(X_train)}")
print(f"Validation set size: {len(X_val)}")
```



```
print(f"Test set size: {len(X_test)}")
```

Training set size: 3609

Validation set size: 714

Test set size: 764

3.2 Entrenar el modelo

Para entrenar el modelo hemos creado una función que recibe los datos de train, test y val. El modelo se define dentro de la función.

```
[22]: import warnings
warnings.filterwarnings("ignore", category=UserWarning, module="sklearn")
warnings.filterwarnings("ignore", category=RuntimeWarning, module="sklearn")
warnings.filterwarnings("ignore", category=RuntimeWarning, module="scipy")

def train_with_hyperparameter_optimization(
    X_train, y_train, X_val, y_val, X_test, y_test
):
    # Como es una serie temporal, no se puede hacer shuffle
    # Se usa TimeSeriesSplit para hacer la validación cruzada
    tscv = TimeSeriesSplit(n_splits=5)

    # Escala las features
    scaler = StandardScaler()
    X_train_scaled = pd.DataFrame(
        scaler.fit_transform(X_train), columns=X_train.columns
    )
    X_val_scaled = pd.DataFrame(scaler.transform(X_val), columns=X_train.
↪columns)
    X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_train.
↪columns)

    # Define el modelo y sus hiperparámetros
    model_config = {
        "LinearRegression": {
            "model": LinearRegression(),
            "params": {
                "fit_intercept": [False, True],
                "copy_X": [True],
                "positive": [False],
                "n_jobs": [-1],
            },
        },
    }

    # Puntuador personalizado de RMSE (negativo para optimización)
```

```

rmse_scorer = make_scorer(
    lambda y, y_pred: -np.sqrt(mean_squared_error(y, y_pred))
)

results = {}

# Entrena y evalua el modelo
for model_name, model_info in model_config.items():
    with tqdm(total=1, desc=f"Training {model_name}") as pbar:
        # Haz una búsqueda Cross Validation aleatoria para reducir el
↪ tiempo de entrenamiento
        search = RandomizedSearchCV(
            estimator=model_info["model"],
            param_distributions=model_info["params"],
            n_iter=100,
            cv=tscv,
            scoring=rmse_scorer,
            n_jobs=-1,
            random_state=42,
            verbose=0,
        )

        # Fit model
        search.fit(X_train_scaled, y_train)
        pbar.update(1)

        # Obten el mejor modelo de la búsqueda
        # y haz predicciones en los conjuntos de entrenamiento, validación
↪ y test
        best_model = search.best_estimator_
        train_pred = best_model.predict(X_train_scaled)
        val_pred = best_model.predict(X_val_scaled)
        test_pred = best_model.predict(X_test_scaled)

        # Guarda los resultados
        results[model_name] = {
            "Train RMSE": np.sqrt(mean_squared_error(y_train, train_pred)),
            "Val RMSE": np.sqrt(mean_squared_error(y_val, val_pred)),
            "Test RMSE": np.sqrt(mean_squared_error(y_test, test_pred)),
            "Best Parameters": search.best_params_,
            "Coefficients": dict(zip(X_train.columns, best_model.coef_)),
            "Intercept": best_model.intercept_,
            "Model": best_model,
        }

return results, scaler

```

```
[23]: # Train the model
results, scaler = train_with_hyperparameter_optimization(
    X_train, y_train, X_val, y_val, X_test, y_test
)
```

```
Training LinearRegression:  0%|          | 0/1 [00:00<?,
?it/s]/Users/edu/Documents/Personal/proyectos/juan-
tiempo/.venv/lib/python3.12/site-packages/sklearn/linear_model/_base.py:279:
RuntimeWarning: divide by zero encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: overflow encountered
in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: invalid value
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: divide by zero
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: overflow encountered
in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: invalid value
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: divide by zero
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: overflow encountered
in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: invalid value
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: divide by zero
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: overflow encountered
```



```

in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: invalid value
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: divide by zero
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: overflow encountered
in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: invalid value
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: divide by zero
encountered in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: overflow encountered
in matmul
    return X @ coef_ + self.intercept_
/Users/edu/Documents/Personal/proyectos/juan-tiempo/.venv/lib/python3.12/site-
packages/sklearn/linear_model/_base.py:279: RuntimeWarning: invalid value
encountered in matmul
    return X @ coef_ + self.intercept_
Training LinearRegression: 100%|          | 1/1 [00:01<00:00, 1.55s/it]

```

```

[24]: # Print results
for model_name, metrics in results.items():
    print(f"\nResults for {model_name}:")
    print(f"Train RMSE: {metrics['Train RMSE']:.4f}")
    print(f"Validation RMSE: {metrics['Val RMSE']:.4f}")
    print(f"Test RMSE: {metrics['Test RMSE']:.4f}")
    print("\nBest Parameters:")
    print(metrics['Best Parameters'])

```

```

Results for LinearRegression:
Train RMSE: 1.8734
Validation RMSE: 1.8251
Test RMSE: 1.8672

```

```

Best Parameters:

```

```
{'positive': False, 'n_jobs': -1, 'fit_intercept': True, 'copy_X': True}
```

3.3 Evaluar el modelo de regresión lineal

```
[25]: import matplotlib.pyplot as plt
import seaborn as sns # Asegúrate de tener seaborn importado
import pandas as pd
import numpy as np
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

def evaluate_model(results, X_test, y_test, scaler):
    """
    Evalúa el modelo lineal entrenado, calcula métricas de rendimiento,
    y genera un gráfico de 2x2 visualizando real vs predicho, residuales,
    distribución de residuales, y serie temporal de predicción (si aplica).
    """
    # Obten el mejor modelo de los entrenados.
    # Asumiendo que existen las claves 'LinearRegression' y 'Model'
    model = results['LinearRegression']['Model']

    # Haz predicciones en el conjunto de test
    # Asegura que X_test sea un DataFrame antes de escalar
    if not isinstance(X_test, pd.DataFrame):
        # Asumiendo que los nombres de las características están almacenados
        X_test = pd.DataFrame(X_test,
                               columns=results['LinearRegression']['Features'])

    X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.
                               columns)
    y_pred = model.predict(X_test_scaled)

    # Asegura que y_test y y_pred sean series para facilitar la alineación si
    # el índice existe
    if isinstance(y_test, np.ndarray):
        # Intenta usar el índice de X_test si y_test era solo un array de numpy
        y_test = pd.Series(y_test, index=X_test.index)
        y_pred = pd.Series(y_pred, index=X_test.index)
    elif isinstance(y_test, pd.Series):
        y_pred = pd.Series(y_pred, index=y_test.index)

    # Calcula las métricas de rendimiento
    r2 = r2_score(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    # Añade un pequeño epsilon a y_test para evitar la división por cero en
    # MAPE para ceros exactos
```

```

mape = np.mean(np.abs((y_test - y_pred) / (y_test + 1e-8))) * 100

# Imprime las metricas
print("\nMétricas de Rendimiento del Modelo:")
print(f"Puntuación R²: {r2:.4f}")
print(f"Error Absoluto Medio: {mae:.4f}°C")
print(f"Raíz del Error Cuadrático Medio: {rmse:.4f}°C")
print(f"Error Porcentual Absoluto Medio: {mape:.2f}%")

# Calcula los residuales antes de graficar
residuals = y_test - y_pred

# Crea la figura con 4 subplots
fig, axes = plt.subplots(2, 2, figsize=(18, 12))

# Plot 1: Real vs Predicho (Arriba Izquierda) - Ahora un gráfico de
↳ dispersión según tu código original 1
# Personaliza los puntos del gráfico de dispersión para que sean azul
↳ semitransparente con tamaño=50
axes[0, 0].scatter(y_test, y_pred, alpha=0.5, c='blue', s=50)
# Añade la línea de predicción perfecta usando el min/max de y_test
axes[0, 0].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
                'r--', lw=2, label='Predicción Perfecta')
axes[0, 0].set_xlabel('Temperatura Real (°C)')
axes[0, 0].set_ylabel('Temperatura Predicha (°C)')
axes[0, 0].set_title('1. Temperatura Real vs Predicha')
axes[0, 0].legend()
axes[0, 0].grid(True)

# Plot 2: Residuales vs Predicho (Arriba Derecha)
axes[0, 1].scatter(y_pred, residuals, alpha=0.5)
axes[0, 1].axhline(y=0, color='r', linestyle='--')
axes[0, 1].set_xlabel('Temperatura Predicha (°C)')
axes[0, 1].set_ylabel('Residuales (°C)')
axes[0, 1].set_title('2. Residuales vs Valores Predichos')
axes[0, 1].grid(True)

# Plot 3: Distribución de Residuales (Abajo Izquierda)
sns.histplot(residuals, kde=True, ax=axes[1, 0])
axes[1, 0].set_xlabel('Residuales (°C)')
axes[1, 0].set_ylabel('Conteo')
axes[1, 0].set_title('3. Distribución de Residuales')
axes[1, 0].grid(True)

```

```

# Plot 4: Serie Temporal (Abajo Derecha) - Comprueba si el índice es
↳datetime
if isinstance(X_test.index, pd.DatetimeIndex):
    # Usa el índice de y_test/y_pred (que se alineó con el índice de X_test)
    axes[1, 1].plot(y_test.index, y_test, label='Real', alpha=0.7)
    axes[1, 1].plot(y_pred.index, y_pred, label='Predicho', alpha=0.7)
    axes[1, 1].set_xlabel('Fecha')
    axes[1, 1].set_ylabel('Temperatura (°C)')
    axes[1, 1].set_title('4. Temperatura Real vs Predicha a lo Largo del
↳Tiempo')
    axes[1, 1].legend()
    axes[1, 1].grid(True)
    # Opcional: Rota las etiquetas de fecha si se superponen
    fig.autofmt_xdate()
else:
    axes[1, 1].set_title("4. Gráfico de Serie Temporal No Disponible")
    axes[1, 1].text(0.5, 0.5, 'El índice no es Datetime',
↳horizontalalignment='center',
                                verticalalignment='center', transform=axes[1, 1].
↳transAxes)
    axes[1, 1].set_xticks([])
    axes[1, 1].set_yticks([])

plt.tight_layout()
plt.show()

return {
    'R2': r2,
    'MAE': mae,
    'RMSE': rmse,
    'MAPE': mape,
    'Residuals': residuals
}

# Usa la función
metrics = evaluate_model(results, X_test, y_test, scaler)

```

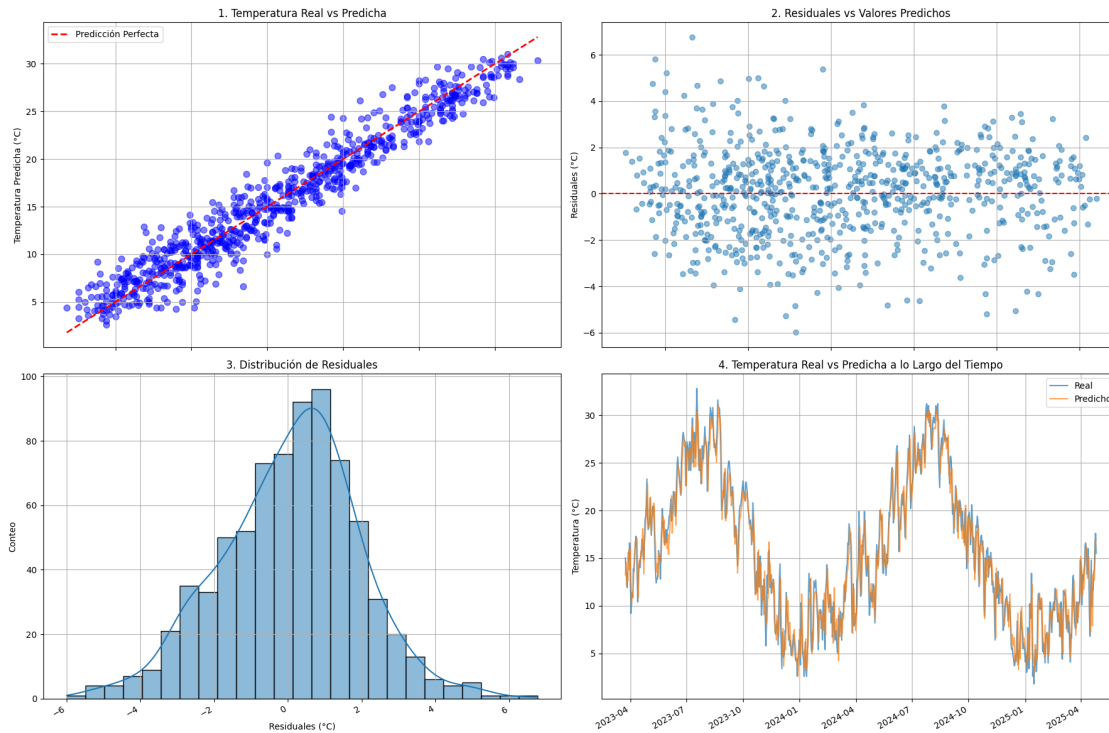
Métricas de Rendimiento del Modelo:

Puntuación R^2 : 0.9365

Error Absoluto Medio: 1.4724°C

Raíz del Error Cuadrático Medio: 1.8672°C

Error Porcentual Absoluto Medio: 13.29%



4 Redes de neuronas

Para entrenar una red neuronal, vamos a utilizar la biblioteca torch. Al igual que antes, vamos a dividir los datos en train, test y val. Para evitar problemas, vamos a volver a leer los datos de train, test y val desde el CSV que hemos creado antes.

```
[26]: data = pd.read_csv(output_filtered_csv, parse_dates=['fecha'],
    ↪ index_col='fecha')
X = data.drop('tmed', axis=1)
y = data['tmed']

# Dividir los datos en conjuntos de entrenamiento, validación y prueba
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
    ↪ random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
    ↪ random_state=42)
```

4.1 Generando los Dataset

Para trabajar con torch, vamos a crear un dataset y un dataloader. Vamos a crear un dataset que contenga los datos de train, test y val. Para ello, vamos a crear una clase que herede de `torch.utils.data.Dataset`.

```
[27]: # Clase de conjunto de datos personalizada
class TemperatureDataset(Dataset):
    def __init__(self, X, y, scaler=None):
        if scaler is None:
            self.scaler = StandardScaler()
            self.X = torch.FloatTensor(self.scaler.fit_transform(X))
        else:
            self.scaler = scaler
            self.X = torch.FloatTensor(self.scaler.transform(X))

        self.y = torch.FloatTensor(y.values.reshape(-1, 1))

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

4.2 Modelo de red neuronal

Vamos a crear un modelo de red neuronal que contenga:

- Una capa de entrada con `input_size` dimensiones
- Primera capa oculta con 64 neuronas, activación ReLU y dropout de 0.2
- Segunda capa oculta con 32 neuronas, activación ReLU y dropout de 0.2
- Tercera capa oculta con 16 neuronas y activación ReLU
- Capa de salida con 1 neurona para regresión

```
[28]: class TemperatureNN(nn.Module):
    def __init__(self, input_size):
        super(TemperatureNN, self).__init__()
        self.layer1 = nn.Linear(input_size, 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.relu1 = nn.LeakyReLU()
        self.dropout1 = nn.Dropout(0.3)

        self.layer2 = nn.Linear(128, 64)
        self.bn2 = nn.BatchNorm1d(64)
        self.relu2 = nn.LeakyReLU()
        self.dropout2 = nn.Dropout(0.2)

        self.layer3 = nn.Linear(64, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.relu3 = nn.LeakyReLU()
        self.dropout3 = nn.Dropout(0.1)

        self.layer4 = nn.Linear(64, 32)
        self.bn4 = nn.BatchNorm1d(32)
```

```

self.relu4 = nn.LeakyReLU()
self.dropout4 = nn.Dropout(0.1)

self.layer5 = nn.Linear(32, 16)
self.bn5 = nn.BatchNorm1d(16)
self.relu5 = nn.LeakyReLU()

self.output_layer = nn.Linear(16, 1)

def forward(self, x):
    x = self.layer1(x)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.dropout1(x)

    x = self.layer2(x)
    x = self.bn2(x)
    x = self.relu2(x)
    x = self.dropout2(x)

    x = self.layer3(x)
    x = self.bn3(x)
    x = self.relu3(x)
    x = self.dropout3(x)

    x = self.layer4(x)
    x = self.bn4(x)
    x = self.relu4(x)

    x = self.layer5(x)
    x = self.bn5(x)
    x = self.relu5(x)

    x = self.output_layer(x)
    return x

```

4.3 Entrenamiento del modelo

1. Inicialización:

- Se definen listas para almacenar las pérdidas de entrenamiento (`train_losses`) y validación (`val_losses`).

2. Bucle de entrenamiento:

- Para cada época, el modelo pasa por dos fases: entrenamiento y validación.

3. Fase de entrenamiento:

- El modelo se pone en modo de entrenamiento (`model.train()`).
- Para cada lote de datos en el `train_loader`:
 - Los datos se mueven al dispositivo (CPU/GPU).
 - Se realiza una predicción (`y_pred`) con el modelo.

- Se calcula la pérdida entre las predicciones y las etiquetas reales (`loss`).
 - Se realiza la retropropagación (`loss.backward()`) para calcular los gradientes.
 - Se actualizan los pesos del modelo con el optimizador (`optimizer.step()`).
 - La pérdida promedio de entrenamiento para la época se calcula y se almacena.
- Fase de validación:**
 - El modelo se pone en modo de evaluación (`model.eval()`).
 - Se desactiva el cálculo de gradientes para ahorrar memoria y acelerar el proceso (`torch.no_grad()`).
 - Para cada lote de datos en el `val_loader`:
 - Se calcula la pérdida entre las predicciones y las etiquetas reales.
 - La pérdida promedio de validación para la época se calcula y se almacena.
 - Ajuste del scheduler:**
 - El scheduler ajusta la tasa de aprendizaje en función de la pérdida de validación.
 - Retorno:**
 - Devuelve las listas de pérdidas de entrenamiento y validación para su análisis posterior.

```
[ ]: def train_model(model, train_loader, val_loader, criterion, optimizer,
                    num_epochs, device, scheduler):
    train_losses = []
    val_losses = []

    epoch_bar = tqdm(range(num_epochs), desc="Entrenando", unit="epoch")

    for epoch in epoch_bar:
        # Fase de entrenamiento
        model.train()
        train_loss = 0
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)

            optimizer.zero_grad()
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)

            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        train_loss = train_loss / len(train_loader)
        train_losses.append(train_loss)

        # Fase de validación
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
```

```

        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        y_pred = model(X_batch)
        loss = criterion(y_pred, y_batch)
        val_loss += loss.item()

    val_loss = val_loss / len(val_loader)
    val_losses.append(val_loss)

    scheduler.step(val_loss)

    epoch_bar.set_postfix(
        train_loss=train_loss, val_loss=val_loss
    )
    print("Entrenamiento completetado.")

    return train_losses, val_losses

```

```

[30]: # Función para evaluar el modelo
def evaluate_model(model, test_loader, criterion, device):
    model.eval()
    test_loss = 0
    predictions = []
    actuals = []

    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)
            test_loss += loss.item()

            predictions.extend(y_pred.cpu().numpy())
            actuals.extend(y_batch.cpu().numpy())

    test_loss = test_loss / len(test_loader)
    return test_loss, np.array(predictions), np.array(actuals)

```

4.3.1 Ejecución del entrenamiento

Para ejecutar el entrenamiento, se definen los hiperparámetros como la tasa de aprendizaje, el número de épocas y el tamaño del lote (`batch_size`). Luego, se crea una instancia del modelo y se define el optimizador y el scheduler. Finalmente, se llama a la función de entrenamiento con los datos de entrenamiento y validación.

```

[31]: def run_neural_network(X_train, X_val, X_test, y_train, y_val, y_test):
        batch_size = 32

```

```

learning_rate = 0.001
num_epochs = 50
device = torch.device("cuda" if torch.cuda.is_available() else "mps" if
↳torch.mps.is_available() else "cpu")
device = torch.device("cpu")
print(f"Usando dispositivo: {device}")
# Crear conjuntos de datos
train_dataset = TemperatureDataset(X_train, y_train)
val_dataset = TemperatureDataset(X_val, y_val, train_dataset.scaler)
test_dataset = TemperatureDataset(X_test, y_test, train_dataset.scaler)

# Crear cargadores de datos
train_loader = DataLoader(train_dataset, batch_size=batch_size,
↳shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)

# Inicializar modelo
model = TemperatureNN(input_size=X_train.shape[1]).to(device)

# Función de pérdida y optimizador
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.5,
    patience=10
)

print(f"Parámetros del modelo: {sum(p.numel() for p in model.
↳parameters())}")
# Entrenar el modelo
train_losses, val_losses = train_model(
    model, train_loader, val_loader, criterion, optimizer, num_epochs,
↳device, scheduler=scheduler
)
# print(f"Pérdidas de entrenamiento: {train_losses}")
# Evaluar en el conjunto de prueba
_, predictions, actuals = evaluate_model(
    model, test_loader, criterion, device
)

# Calcular métricas
mse = np.mean((predictions - actuals) ** 2)

```

```

rmse = np.sqrt(mse)
mae = np.mean(np.abs(predictions - actuals))
r2 = 1 - np.sum((actuals - predictions) ** 2) / np.sum(
    (actuals - np.mean(actuals)) ** 2
)

print('\nMétricas de Prueba:')
print(f'MSE: {mse:.4f}')
print(f'RMSE: {rmse:.4f}')
print(f'MAE: {mae:.4f}')
print(f'R²: {r2:.4f}')

# Graficar curvas de entrenamiento y predicción vs valores reales en un
↳ subplot 2x1
fig, axes = plt.subplots(2, 1, figsize=(12, 10))

# Primer subplot: curvas de pérdida
axes[0].plot(train_losses, label='Pérdida de Entrenamiento')
axes[0].plot(val_losses, label='Pérdida de Validación')
axes[0].set_xlabel('Época')
axes[0].set_ylabel('Pérdida')
axes[0].set_title('Pérdida de Entrenamiento y Validación')
axes[0].legend()
axes[0].grid(True)

# Segundo subplot: predicciones vs valores reales
axes[1].scatter(actuals, predictions.reshape(-1), alpha=0.5)
axes[1].plot([actuals.min(), actuals.max()],
             [actuals.min(), actuals.max()],
             'r--', lw=2)
axes[1].set_xlabel('Temperatura Real')
axes[1].set_ylabel('Temperatura Predicha')
axes[1].set_title('Temperatura Real vs Predicha')
axes[1].grid(True)

plt.tight_layout()
plt.show()

return train_losses, val_losses, train_dataset.scaler

```

4.4 Resultados del entrenamiento

```

[32]: # Ejecutar el entranamiento de la red neuronal
train_losses, val_losses, scaler = run_neural_network(X_train, X_val, X_test,
↳ y_train, y_val, y_test)

```

Usando dispositivo: cpu

Parámetros del modelo: 18081

Entrenando: 100% | 50/50 [00:14<00:00, 3.54epoch/s, train_loss=4.9, val_loss=4.1]

Entrenamiento completado.

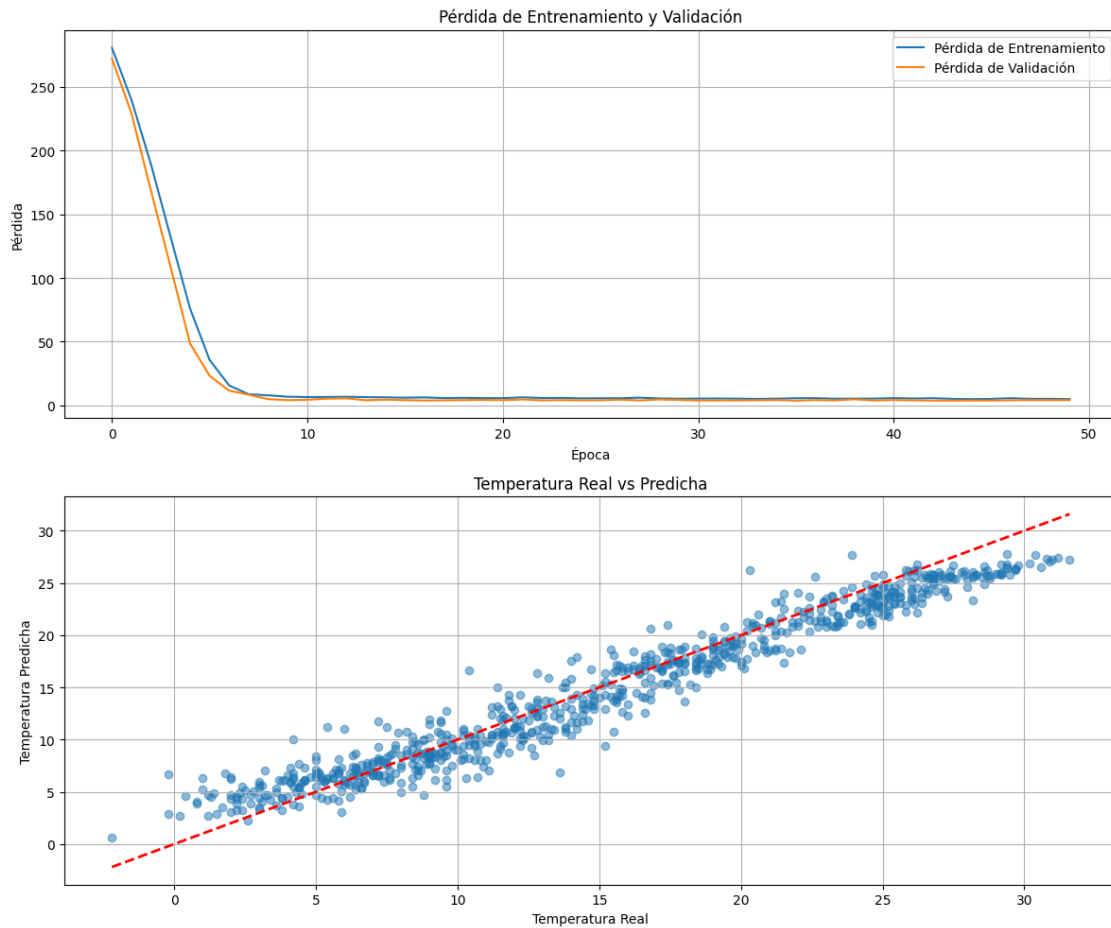
Métricas de Prueba:

MSE: 3.9283

RMSE: 1.9820

MAE: 1.6037

R^2 : 0.9389



Podemos observar que la pérdida de entrenamiento y validación disminuye con el tiempo, lo que indica que el modelo está aprendiendo.

También podemos ver que a partir de la época 30, el modelo se estabiliza y no mejora sustancialmente, por lo que podemos considerar que ha convergido. (si lo siguiéramos entrenar el modelo, podríamos acabar haciendo overfitting o sobreajustar el modelo a los datos de entrenamiento, lo que haría que el modelo no generalizara bien a nuevos datos).