MOP_V13

May 22, 2025

0.1 3. Modelo de Optimización y Prognosis (MOP)

0.1.1 3.1. Librerías

[1]:
```
# ============================================================================
# Paso 0: Importar librerías y definir funciones auxiliares
# ============================================================================

# Librerías necesarias
import os
import glob
import re  # Import the regular expression module

import pandas as pd
import numpy as np
import math
from math import ceil

import matplotlib
# matplotlib.use('TKAgg')
import matplotlib.pyplot as plt
from matplotlib.ticker import ScalarFormatter
import seaborn as sns

import time
import warnings
warnings.filterwarnings("ignore")

# Para guardar y cargar modelos
import joblib

# Librerías de preprocesado y modelado de scikit-learn
from sklearn.model_selection import train_test_split, KFold, cross_val_predict,␣
 ↪GridSearchCV, cross_val_score
from sklearn import model_selection
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.pipeline import Pipeline
from sklearn import set_config
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.cross_decomposition import PLSRegression
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, ConstantKernel␣
 ↪as C
from sklearn.svm import SVR
from sklearn.multioutput import MultiOutputRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.neural_network import MLPRegressor

import keras
from keras.layers import Dense
from keras.models import Sequential

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

from scikeras.wrappers import KerasRegressor
from sklearn.base import BaseEstimator, RegressorMixin



from skopt import BayesSearchCV
from skopt.space import Real, Integer, Categorical
```

[2]:
```
# Clase auxiliar que convierte un diccionario en un objeto con atributos.
class TagBunch:
    def __init__(self, d):
        self.__dict__.update(d)

# Monkey-patch: asignar __sklearn_tags__ al wrapper para evitar el error
# Definición del wrapper personalizado para KerasRegressor
class MyKerasRegressorWrapper(BaseEstimator, RegressorMixin):
    def __init__(self, model, hidden_layer_size=50, hidden_layer_size_2=3,␣
 ↪epochs=100, **kwargs):
        """
        model: función que construye el modelo (por ejemplo, create_model)
        hidden_layer_size, hidden_layer_size_2, epochs: parámetros a pasar a la␣
 ↪función
        kwargs: otros parámetros (como batch_size, verbose, etc.)
        """
        self.model = model
        self.hidden_layer_size = hidden_layer_size
        self.hidden_layer_size_2 = hidden_layer_size_2
```

```
        self.epochs = epochs
        self.kwargs = kwargs
        self.estimator_ = None  # Se llenará al entrenar

    def fit(self, X, y, **fit_params):
        # Se crea la instancia interna de KerasRegressor usando scikeras.
        self.estimator_ = KerasRegressor(
            model=self.model,
            hidden_layer_size=self.hidden_layer_size,
            hidden_layer_size_2=self.hidden_layer_size_2,
            epochs=self.epochs,
            **self.kwargs
        )
        self.estimator_.fit(X, y, **fit_params)
        return self

    def predict(self, X):
        return self.estimator_.predict(X)

    def score(self, X, y):
        return self.estimator_.score(X, y)

    def get_params(self, deep=True):
        params = {
            "model": self.model,
            "hidden_layer_size": self.hidden_layer_size,
            "hidden_layer_size_2": self.hidden_layer_size_2,
            "epochs": self.epochs,
        }
        params.update(self.kwargs)
        return params

    def set_params(self, **parameters):
        for key, value in parameters.items():
            setattr(self, key, value)
        return self

    def __sklearn_tags__(self):
        # NUEVO: Devolver un objeto TagBunch en lugar de un dict.
        return TagBunch({
            "requires_fit": True,
            "X_types": ["2darray"],
            "preserves_dtype": [np.float64],
            "allow_nan": False,
            "requires_y": True,
        })
```

```
    def __sklearn_is_fitted__(self):
        return self.estimator_ is not None
```

[3]:
```
# Función para limpiar nombres de archivo inválidos
def clean_filename(name):
    return re.sub(r'[\\/*?:"<>|]', "_", name)

# Función para calcular la correlación p_ij según la fórmula del paper:
#   p_ij = (1/(N-1)) * Σ[(ŷ(k) - _ŷ) * (x_j(k) - _xj)] / ( _ŷ  _xj)
def compute_corr(y, x):
    N = len(y)
    mean_y = np.mean(y)
    mean_x = np.mean(x)
    std_y = np.std(y, ddof=1)
    std_x = np.std(x, ddof=1)
    return np.sum((y - mean_y) * (x - mean_x)) / ((N - 1) * std_y * std_x)

# Función para dibujar un mapa de calor con seaborn
def plot_heatmap(matrix, col_labels, row_labels, title, ax=None):
    """
    Dibuja un mapa de calor de 'matrix' usando seaborn.
    'col_labels' y 'row_labels' definen las etiquetas de columnas y filas.
    Si se proporciona 'ax', se dibuja en ese subplot; de lo contrario, se crea␣
 ↪uno nuevo.
    """
    if ax is None:
        fig, ax = plt.subplots()
    sns.heatmap(matrix, annot=True, fmt=".2f", xticklabels=col_labels,
                yticklabels=row_labels, cmap="viridis", ax=ax)
    ax.set_title(title)
    return ax

# Función auxiliar para calcular el Coefficient of Prognosis (CoP)
def compute_CoP(y_true, y_pred):
    N = len(y_true)
    mean_y = np.mean(y_true)
    mean_y_pred = np.mean(y_pred)
    std_y = np.std(y_true, ddof=1)
    std_y_pred = np.std(y_pred, ddof=1)
    denominator = (N - 1) * std_y * std_y_pred
    if denominator == 0:
        return np.nan
    else:
        return (np.sum((y_true - mean_y) * (y_pred - mean_y_pred)) /␣
 ↪denominator) ** 2

# Función para calcular el diccionario de CoP para cada salida y cada modelo
```

```python
def compute_cop_results(metricas, outputs):
    """
    Genera un diccionario de CoP con la estructura:
      { output1: { 'PLS': cop_value, 'LR': cop_value, ... },
        output2: { 'PLS': cop_value, ... },
        ... }
    Se asume que 'metricas' tiene, para cada modelo, una lista de valores de CoP
    en el mismo orden que 'outputs'.
    """
    cop_results = {}
    for j, output in enumerate(outputs):
        cop_results[output] = {}
        for model_name in metricas.keys():
            cop_results[output][model_name] = metricas[model_name]['CoP'][j]
    return cop_results

# Función para graficar los CoP en subplots y guardar la figura (Paso 8)
def plot_cop_subplots(cop_results, outputs, mejores_modelos, figure_path,
                      filename='CoP_para_cada_modelo.png'):
    """
    Dibuja un subplot por variable de salida mostrando únicamente las barras␣
 ↪con CoP
    y acrónimos en el eje x. Resalta con borde rojo el mejor modelo.
    """
    n_out = len(outputs)
    ncols = 3
    nrows = int(np.ceil(n_out / ncols))
    fig, axes = plt.subplots(nrows, ncols, figsize=(12, 6 * nrows))
    axes = axes.flatten()
    for i, output in enumerate(outputs):
        model_names = list(cop_results[output].keys())
        cop_vals = [cop_results[output][m] for m in model_names]
        acronyms = model_names
        ax = axes[i]
        bars = ax.bar(range(len(model_names)), cop_vals)
        # configurar acrónimos como xticklabels
        ax.set_xticks(range(len(model_names)))
        ax.set_xticklabels(acronyms, fontsize=10)
        # resaltar mejor modelo
        best = mejores_modelos[output]
        if best in model_names:
            best_idx = model_names.index(best)
            bars[best_idx].set_edgecolor('red')
            bars[best_idx].set_linewidth(2)
        ax.set_title(f'CoP para {output}', fontsize=12)
        ax.set_ylabel('CoP')
        ax.set_ylim(0, 1)
```

```
    # eliminar ejes vacíos
    for j in range(i+1, len(axes)):
        fig.delaxes(axes[j])
    plt.tight_layout()
    out_file = os.path.join(figure_path, filename)
    plt.savefig(out_file, dpi=1000)
    plt.close()
    print(f'Figura CoP guardada en {out_file}')
```

[4]:
```
# Tiempo de inicio del programa
start_time_program = time.time()
```

[5]:
```
# ================================================================================
# Paso 1: Definir rutas, cargar datos y configurar directorios
# ================================================================================
base_path = os.getcwd()  # Se asume que el notebook se ejecuta desde la carpeta␣
 ↪'MOP'
db_path = os.path.join(base_path, "DB_MOP")
fig_path = os.path.join(base_path, "Figuras_MOP")
model_path = os.path.join(base_path, "Modelos_MOP")

# Ruta al archivo de la base de datos
data_file = os.path.join(db_path, "design_DB_preprocessed_5000_Uniforme.csv")
print(data_file)

# Ruta al archivo de las figuras
figure_path = os.path.join(fig_path, "5000_MOT_Uniforme")
print(figure_path)

# Ruta al archivo de los modelos
modelo_path = os.path.join(model_path, "5000_MOT_Uniforme")
print(modelo_path)

# Lectura del archivo CSV
try:
    df = pd.read_csv(data_file)
    print("Archivo cargado exitosamente.")
except FileNotFoundError:
    print("Error: Archivo no encontrado. Revisa la ruta del archivo.")
except pd.errors.ParserError:
    print("Error: Problema al analizar el archivo CSV. Revisa el formato del␣
 ↪archivo.")
except Exception as e:
    print(f"Ocurrió un error inesperado: {e}")

# Función para limpiar nombres de archivo inválidos
def clean_filename(name):
```

```
    return re.sub(r'[\\/*?:"<>|]', "_", name)
```

```
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\3.MOP\DB_MO
P\design_DB_preprocessed_5000_Uniforme.csv
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\3.MOP\Figur
as_MOP\5000_MOT_Uniforme
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\3.MOP\Model
os_MOP\5000_MOT_Uniforme
Archivo cargado exitosamente.
```

[6]:
```
# ================================================================================
# Paso 2: Preprocesar datos: separar columnas en X, M, P y convertir a numérico
# ================================================================================
# Se separan las columnas según prefijos:
#   - Variables 'x' (inputs principales)
#   - Variables 'm' (otras características del motor)
#   - Variables 'p' (salidas: parámetros a predecir)
X_cols = [col for col in df.columns if col.startswith('x')]
M_cols = [col for col in df.columns if col.startswith('m')]
P_cols = [col for col in df.columns if col.startswith('p')]

# Se crea el DataFrame de características y del target. En este ejemplo se usa␣
 ↪X (inputs)
# y P (salidas), pero se pueden incluir también las M si así se requiere.
X = df[X_cols].copy()
M = df[M_cols].copy()
P = df[P_cols].copy()
y = df[P_cols].copy()  # Usamos las columnas p para las predicciones

# Convertir todas las columnas a tipo numérico en caso de haber algún dato no␣
 ↪numérico
for col in X.columns:
    X[col] = pd.to_numeric(X[col], errors='coerce')
for col in M.columns:
    M[col] = pd.to_numeric(M[col], errors='coerce')
for col in P.columns:
    P[col] = pd.to_numeric(P[col], errors='coerce')
for col in y.columns:
    y[col] = pd.to_numeric(y[col], errors='coerce')

# Concatena las matrices X y M
X_M = pd.concat([X, M], axis=1)

print("\nPrimeras filas de X:")
display(X.head())
print("\nPrimeras filas de y (P):")
display(y.head())
```

```
print("Columnas de salida originales:", y.columns.tolist())

# Definir un umbral para la varianza
threshold = 1e-8  # Este umbral puede ajustarse según la precisión deseada

# Calcular la varianza de cada columna del DataFrame y
variances = y.var()
print("\nVariancia de cada columna de salida:")
print(variances)

# Seleccionar aquellas columnas cuya varianza es mayor que el umbral
cols_to_keep = variances[variances > threshold].index
y = y[cols_to_keep]

# Filtrar las filas del DataFrame y para eliminar aquellas que contienen NaN
y = y.dropna()  # Se eliminan todas las filas con al menos un valor NaN en y
# Actualizar X para que quede alineado con los índices de y
X = X.loc[y.index]

print("\nColumnas de salida tras eliminar las constantes o casi constantes:")
print(y.columns.tolist())

model_names = ['PLS','LR','GPR','SVR','RF','ANN','ANN-K']
salidas = y.columns.tolist()
entradas = X.columns.tolist()
```

```
Primeras filas de X:
   x1::OSD  x2::Dint      x3::L    x4::tm    x5::hs2    x6::wt  x7::Nt  x8::Nh
0    48.60   27.8640  14.800000  2.780311   6.312467  4.392325       6       4
1    59.40   24.0560  29.200000  2.121244  10.249868  2.569301      12       3
2    54.72   32.0528  22.960001  2.456926   7.797124  2.123813      18       3
3    48.84   21.9616  25.120000  3.032072   6.972909  2.557345      14       3
4    59.76   27.1024  29.680002  3.249535   8.141503  4.802138      10       3

Primeras filas de y (P):
      p1::W  p2::Tnom  p3::nnom    p4::GFF  p5::BSP_T   p6::BSP_n  p7::BSP_Pm  \
0  0.322074      0.11    3960.0  40.082718   0.170606  17113.2340   305.74252
1  0.674799      0.11    3960.0  24.675780   0.412852   4913.5480   212.43124
2  0.535554      0.11    3960.0  42.652370   0.538189   3806.5370   214.53262
3  0.487619      0.11    3960.0  57.017277   0.380920   5161.0967   205.87508
4  0.749844      0.11    3960.0  37.444870   0.429127   4961.4146   222.95651

   p8::BSP_Mu  p9::BSP_Irms  p10::MSP_n  p11::UWP_Mu
```

```
0   90.763855     10.070335  18223.3200    86.138150
1   87.076820      7.558135   5737.1406    88.799880
2   83.929474      7.553457   4325.1235    83.402340
3   87.040310      7.554095   6293.4336    91.343490
4   89.363690      7.554099   5615.5110    91.807846
```

```
Columnas de salida originales: ['p1::W', 'p2::Tnom', 'p3::nnom', 'p4::GFF',
'p5::BSP_T', 'p6::BSP_n', 'p7::BSP_Pm', 'p8::BSP_Mu', 'p9::BSP_Irms',
'p10::MSP_n', 'p11::UWP_Mu']

Variancia de cada columna de salida:
p1::W           2.409097e-02
p2::Tnom        1.733798e-33
p3::nnom        0.000000e+00
p4::GFF         1.216293e+02
p5::BSP_T       5.526305e-02
p6::BSP_n       2.691714e+07
p7::BSP_Pm      1.675008e+04
p8::BSP_Mu      7.538927e+00
p9::BSP_Irms    2.199128e+01
p10::MSP_n      3.204081e+07
p11::UWP_Mu     8.609559e+00
dtype: float64

Columnas de salida tras eliminar las constantes o casi constantes:
['p1::W', 'p4::GFF', 'p5::BSP_T', 'p6::BSP_n', 'p7::BSP_Pm', 'p8::BSP_Mu',
'p9::BSP_Irms', 'p10::MSP_n', 'p11::UWP_Mu']
```

[7]:
```
# ================================================================================
# 3. DIVISIÓN DE LOS DATOS EN ENTRENAMIENTO Y TEST
# ================================================================================
# Se separa el conjunto de datos en entrenamiento (80%) y test (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
 ↪random_state=42)
print(f"\nTamaño conjunto entrenamiento: {X_train.shape}, test: {X_test.shape}")

display(X_train.head())
display(y_train.head())
```

```
Tamaño conjunto entrenamiento: (3008, 8), test: (753, 8)
```

	x1::OSD	x2::Dint	x3::L	x4::tm	x5::hs2	x6::wt	x7::Nt	\
1089	55.489346	27.316868	10.849793	2.187989	9.253901	4.002112	10	
2943	58.628390	36.216133	29.793587	2.323664	6.696508	2.822144	11	
485	58.006080	27.873138	39.821440	2.109099	9.197280	4.695756	8	
2181	59.283650	22.919085	24.532866	2.569501	14.360030	2.136430	14	
3311	55.225384	22.993841	21.608246	2.596443	10.905984	4.641170	12	

```
      x8::Nh
1089       6
2943       4
485        7
2181       6
3311       3

         p1::W    p4::GFF  p5::BSP_T   p6::BSP_n  p7::BSP_Pm  p8::BSP_Mu  \
1089  0.368133  52.472054   0.307818  12249.8440   394.86935    90.61840
2943  0.738758  42.728900   0.732744   3915.3108   300.43277    87.74775
485   0.954001  54.171997   1.099048   4131.3220   475.48224    89.40208
2181  0.605616  33.351067   0.638103   5247.9966   350.68127    84.63588
3311  0.501348  33.346000   0.300204   6547.3223   205.83038    88.26435

      p9::BSP_Irms  p10::MSP_n  p11::UWP_Mu
1089     15.081463  15475.8150     89.26957
2943     10.061653   4290.1313     82.64902
485      17.622501   5139.3590     90.48702
2181     15.106777   6506.3423     87.61828
3311      7.553548   7994.2230     89.99379
```

[8]:
```
# ============================================================================
# 3.1. ESCALADO DE LA VARIABLE OBJETIVO (y)
# ============================================================================
# Dado que los modelos son sensibles al escalado y se deben evaluar en el mismo␣
 ↪espacio,
# se escala la variable de salida utilizando StandardScaler.
target_scaler = StandardScaler()
y_train_scaled = target_scaler.fit_transform(y_train)
y_test_scaled  = target_scaler.transform(y_test)
```

[9]:
```
# ============================================================================
# 4. CREACIÓN DEL PIPELINE DE PREPROCESAMIENTO
# ============================================================================
# Se define un pipeline para el preprocesado de datos que aplica:
#   a) Escalado (StandardScaler)
#   b) Análisis PCA (se retiene el 95% de la varianza)
'''
data_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95, random_state=42)),
])
'''
data_pipeline = Pipeline([
    ('scaler', StandardScaler())
])
# Visualizar el pipeline
set_config(display="diagram")
```

```
display(data_pipeline)
```

```
Pipeline(steps=[('scaler', StandardScaler())])
```

[10]:
```
# ============================================================================
# 4.1. Leer el archivo de hiperparámetros de cada modelo.
# ============================================================================
import json

# Supongamos que el JSON está en la raíz del proyecto y se llama␣
 ↪'hiperparametros_MOP.json'
params_file = os.path.join(modelo_path, "hiperparametros_MOP.json")
try:
    with open(params_file, "r") as f:
        hiperparametros = json.load(f)
    print(f"Hiperparámetros cargados desde {params_file}")
except FileNotFoundError:
    print(f"No se encontró el archivo de hiperparámetros: {params_file}")
    param_grids = {}

print("Contenido de hiperparametros_MOP.json:")
print(json.dumps(hiperparametros, indent=4))

# Asegurarnos de tener diccionario con cada modelo
hiperparametros = {
    "PLS":    hiperparametros.get("PLS", {}),
    "LR":     hiperparametros.get("LR", {}),
    "GPR":    hiperparametros.get("GPR", {}),
    "SVR":    hiperparametros.get("SVR", {}),
    "RF":     hiperparametros.get("RF", {}),
    "ANN":    hiperparametros.get("ANN", {}),
    "ANN-K":  hiperparametros.get("ANN-K", {}),
}
```

```
Hiperparámetros cargados desde C:\Users\s00244\Documents\GitHub\MotorDesignDataD
riven\Notebooks_TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\hiperparametros_MOP.json
Contenido de hiperparametros_MOP.json:
{
    "PLS": {
        "model__max_iter": 500,
        "model__n_components": 7,
        "model__scale": false,
        "model__tol": 1e-06
    },
    "LR": {
        "model__copy_X": true,
        "model__fit_intercept": true,
        "model__positive": false
```

```
    },
    "GPR": {
        "model__estimator__alpha": 0.0034394990986855454,
        "model__estimator__kernel__k1__length_scale": 260.9614680853858,
        "model__estimator__kernel__k2__noise_level": 1.0816857261904209e-05,
        "model__estimator__normalize_y": false
    },
    "SVR": {
        "model__estimator__C": 10,
        "model__estimator__epsilon": 0.001,
        "model__estimator__gamma": "scale",
        "model__estimator__kernel": "rbf"
    },
    "RF": {
        "model__max_depth": null,
        "model__max_features": "log2",
        "model__min_samples_leaf": 1,
        "model__min_samples_split": 2,
        "model__n_estimators": 300
    },
    "ANN": {
        "model__activation": "logistic",
        "model__alpha": 0.0001,
        "model__hidden_layer_sizes": [
            100,
            100
        ],
        "model__learning_rate_init": 0.0001,
        "model__max_iter": 1000,
        "model__solver": "lbfgs"
    },
    "ANN-K": {
        "model__batch_size": 16,
        "model__epochs": 500,
        "model__hidden_layer_size": 100,
        "model__hidden_layer_size_2": 10
    }
}
```

[11]:
```
# ============================================================================
# 5. DEFINICIÓN DE PIPELINES PARA LOS MODELOS
# ============================================================================
# Se establecen los modelos a probar:
#   - PLS (Partial Least Squares)
#   - Regresión Lineal
#   - Kriging (GPR)
```

```
#   - SVR (Support Vector Regression), envuelto en MultiOutputRegressor para␣
 ↪salida multivariable
#   - Random Forest
#   - Artificial Neural Network (ANN), mediante un Multi-layer Perceptron␣
 ↪regressor
#   - Artificial Neural Network (ANN), mediante Keras

# ============================================================================
# Paso 5.1: PLS - Pipeline con Hiperparámetros
# ============================================================================
p = hiperparametros['PLS'].get('model__n_components')
pls = PLSRegression(n_components=p) if p is not None else PLSRegression()
pipeline_pls = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', pls)
])

# ============================================================================
# Paso 5.2: RL - Pipeline con Hiperparámetros
# ============================================================================
fi = hiperparametros['LR'].get('model__fit_intercept')
lr = LinearRegression(fit_intercept=fi)
pipeline_lr = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', lr)
])

# ============================================================================
# Paso 5.3: GPR - Pipeline con Hiperparámetros
# ============================================================================
gp_par = hiperparametros['GPR']
# kernel parámetros
lscale = gp_par.get('model__estimator__kernel__k1__length_scale')
nlevel = gp_par.get('model__estimator__kernel__k2__noise_level')
alpha  = gp_par.get('model__estimator__alpha')
norm_y = gp_par.get('model__estimator__normalize_y')

kernel = RBF(length_scale=lscale) + WhiteKernel(noise_level=nlevel)
gpr = GaussianProcessRegressor(kernel=kernel,
                               alpha=alpha,
                               normalize_y=norm_y,
                               random_state=42,
                               n_restarts_optimizer=10)
pipeline_gpr = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', MultiOutputRegressor(gpr))
])
```

```
# ============================================================================
# Paso 5.4: SVR - Pipeline con Hiperparámetros
# ============================================================================
# Se utiliza MultiOutputRegressor ya que SVR no soporta multi-output
svr_par = hiperparametros['SVR']
C       = svr_par.get('model__estimator__C')
eps     = svr_par.get('model__estimator__epsilon')
svr = SVR(C=C, epsilon=eps)
pipeline_svr = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', MultiOutputRegressor(svr))
])

# ============================================================================
# Paso 5.5: RF - Pipeline con Hiperparámetros
# ============================================================================
# RandomForestRegressor maneja multi-output
rf_par = hiperparametros['RF']
n_est = rf_par.get('model__n_estimators')
m_depth = rf_par.get('model__max_depth')
min_ss  = rf_par.get('model__min_samples_split')
rf = RandomForestRegressor(n_estimators=n_est,
                           max_depth=m_depth,
                           min_samples_split=min_ss,
                           random_state=42)
pipeline_rf = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', rf)
])

# ============================================================================
# Paso 5.6: ANN - Pipeline con Hiperparámetros
# ============================================================================
# Multi-layer Perceptron regressor
ann_par = hiperparametros['ANN']
hls = ann_par.get('model__hidden_layer_sizes')
mi  = ann_par.get('model__max_iter')
ann = MLPRegressor(hidden_layer_sizes=hls,
                   max_iter=mi,
                   activation='relu',
                   solver='adam',
                   random_state=42)
pipeline_ann = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', ann)
])
```

```
# ============================================================================
# Paso 5.7: ANN-K - Pipeline con Hiperparámetros
# ============================================================================
akk_par = hiperparametros['ANN-K']
h1 = akk_par.get('model__hidden_layer_size')
h2 = akk_par.get('model__hidden_layer_size_2')
ep = akk_par.get('model__epochs')

n_cols = X.shape[1]
n_out = y.shape[1]  # El modelo debe producir n_out salidas

# Definir la función que crea el modelo Keras
# @tf.function(reduce_retracing=True)
def ANN_K_model(hidden_layer_size=h1, hidden_layer_size_2=h2):
    model = Sequential()
    model.add(Dense(hidden_layer_size, activation='relu',␣
 ↪input_shape=(n_cols,)))
    model.add(Dense(hidden_layer_size_2, activation='relu'))
    model.add(Dense(n_out))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# Envolver el modelo en KerasRegressor para utilizarlo con scikit-learn
my_keras_reg = MyKerasRegressorWrapper(
    model=ANN_K_model,
    hidden_layer_size=h1,
    hidden_layer_size_2=h2,
    epochs=ep,
    random_state=42,
    verbose=0
)

# Crear el pipeline para la ANN-K; se incluirá la etapa de preprocesamiento y␣
 ↪el modelo Keras
pipeline_ann_keras = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', my_keras_reg)
])

# ============================================================================
# Paso 5.8: ANN-K - Se agrupan los distintos pipelines
# ============================================================================
# Se agrupan los pipelines en un diccionario para iterar y evaluar fácilmente

pipelines = {
    'PLS':   pipeline_pls,
```

```
    'LR':    pipeline_lr,
    'GPR':   pipeline_gpr,
    'SVR':   pipeline_svr,
    'RF':    pipeline_rf,
    'ANN':   pipeline_ann,
    'ANN-K': pipeline_ann_keras
}
'''
pipelines = {
    'PLS':   pipeline_pls,
    'LR':    pipeline_lr,
    'GPR':   pipeline_gpr,
    'SVR':   pipeline_svr,
    'RF':    pipeline_rf,
    'ANN':   pipeline_ann
    #'ANN-K': pipeline_ann_keras
}
'''
# Ejemplo de estructura de uno de los modelos:
# Visualizar el pipeline
set_config(display="diagram")
display(pipeline_pls)
display(pipeline_ann_keras)
```

```
Pipeline(steps=[('preprocessing',
                 Pipeline(steps=[('scaler', StandardScaler())])),
                ('model', PLSRegression(n_components=7))])
```

```
Pipeline(steps=[('preprocessing',
                 Pipeline(steps=[('scaler', StandardScaler())])),
                ('model',
                 MyKerasRegressorWrapper(epochs=500, hidden_layer_size=100,
                                         hidden_layer_size_2=10,
                                         model=<function ANN_K_model at␣
↪0x000001E7791071A0>,
                                         random_state=42, verbose=0))])
```

[12]:
```
# ============================================================================
# Paso 6. VALIDACIÓN CRUZADA: EVALUACIÓN DE MODELOS CON HIPERPARÁMETROS
# ============================================================================
# Se utilizan 5 particiones (KFold) para evaluar cada modelo mediante Cross␣
↪Validation.
# las métricas (R², MSE, RMSE, MAE y CoP) para cada variable de salida por␣
↪modelo.
cv = KFold(n_splits=5, shuffle=True, random_state=42)
metrics_results = {}
trained_estimators = {}
```

```python
print("Evaluación por validación cruzada (y escalado):")
for name, pipe in pipelines.items():
    # 1) Preparamos el Grid: convertimos valores simples en listas
    raw_grid = hiperparametros.get(name, {})
    grid = {
        param: (val if isinstance(val, (list, tuple, np.ndarray)) else [val])
        for param, val in raw_grid.items()
    }
    if grid:
        estimator = GridSearchCV(pipe,
                                 param_grid=grid,
                                 cv=cv,
                                 scoring="r2",
                                 refit=True,
                                 n_jobs=-1)
    else:
        estimator = pipe

    # 2) Evaluamos con cross_val_predict fuera de muestra
    y_pred_cv = cross_val_predict(estimator, X_train, y_train_scaled, cv=cv)

    # 3) Calculamos métricas
    mse_cols  = mean_squared_error(y_train_scaled, y_pred_cv,␣
 ↪multioutput="raw_values")
    rmse_cols = np.sqrt(mse_cols)
    mae_cols  = mean_absolute_error(y_train_scaled, y_pred_cv,␣
 ↪multioutput="raw_values")
    r2_cols   = r2_score(y_train_scaled, y_pred_cv, multioutput="raw_values")
    cop_cols  = np.array([
        compute_CoP(y_train_scaled[:, j], y_pred_cv[:, j])
        for j in range(y_train_scaled.shape[1])
    ])

    metrics_results[name] = {
        "mse_columns":  dict(zip(y_train.columns, mse_cols)),
        "rmse_columns": dict(zip(y_train.columns, rmse_cols)),
        "mae_columns":  dict(zip(y_train.columns, mae_cols)),
        "r2_columns":   dict(zip(y_train.columns, r2_cols)),
        "cop_columns":  dict(zip(y_train.columns, cop_cols)),
        "mse_avg":      mse_cols.mean(),
        "rmse_avg":     rmse_cols.mean(),
        "mae_avg":      mae_cols.mean(),
        "r2_avg":       r2_cols.mean(),
        "cop_avg":      cop_cols.mean(),
    }

    print(f"\nModelo {name}:")
```

```
    print(f"  R² promedio:  {metrics_results[name]['r2_avg']:.4f}")
    print(f"  MSE promedio: {metrics_results[name]['mse_avg']:.4f}")
    print(f"  RMSE promedio:{metrics_results[name]['rmse_avg']:.4f}")
    print(f"  MAE promedio: {metrics_results[name]['mae_avg']:.4f}")
    print(f"  CoP promedio: {metrics_results[name]['cop_avg']:.4f}")

    # 4) Ahora, refit completo sobre todo el set de entrenamiento
    if hasattr(estimator, "fit"):
        estimator.fit(X_train, y_train_scaled)
    # Guardamos el objeto (GridSearchCV o Pipeline ya ajustado)
    trained_estimators[name] = estimator
```

```
Evaluación por validación cruzada (y escalado):

Modelo PLS:
  R² promedio:  0.8231
  MSE promedio: 0.1769
  RMSE promedio:0.3766
  MAE promedio: 0.2643
  CoP promedio: 0.8231

Modelo LR:
  R² promedio:  0.8232
  MSE promedio: 0.1768
  RMSE promedio:0.3764
  MAE promedio: 0.2640
  CoP promedio: 0.8232

Modelo GPR:
  R² promedio:  0.9811
  MSE promedio: 0.0189
  RMSE promedio:0.1325
  MAE promedio: 0.0519
  CoP promedio: 0.9811

Modelo SVR:
  R² promedio:  0.9838
  MSE promedio: 0.0162
  RMSE promedio:0.1229
  MAE promedio: 0.0399
  CoP promedio: 0.9840

Modelo RF:
  R² promedio:  0.8771
  MSE promedio: 0.1229
  RMSE promedio:0.3237
  MAE promedio: 0.2339
  CoP promedio: 0.9097
```

```
Modelo ANN:
  R² promedio:  0.9855
  MSE promedio: 0.0145
  RMSE promedio:0.1180
  MAE promedio: 0.0460
  CoP promedio: 0.9855

Modelo ANN-K:
  R² promedio:  0.9837
  MSE promedio: 0.0163
  RMSE promedio:0.1255
  MAE promedio: 0.0615
  CoP promedio: 0.9839
```

[13]:
```python
# ============================================================================
# 6.1. REPRESENTACIÓN DE RESULTADOS DE LA VALIDACIÓN CRUZADA
# ============================================================================
# Visualización mejorada para validación cruzada:
summary_cv = pd.DataFrame({
    'Modelo': list(metrics_results.keys()),
    'R2_promedio': [metrics_results[m]['r2_avg'] for m in metrics_results],
    'RMSE_promedio': [metrics_results[m]['rmse_avg'] for m in metrics_results],
    'MSE_promedio': [metrics_results[m]['mse_avg'] for m in metrics_results],
    'MAE_promedio': [metrics_results[m]['mae_avg'] for m in metrics_results],
    'CoP_promedio': [metrics_results[m]['cop_avg'] for m in metrics_results]
})

# 2. Gráficos de barras para los promedios de R2 y MSE
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

# Gráfico de R2 Promedio
bars1 = ax[0].bar(summary_cv['Modelo'], summary_cv['R2_promedio'],␣
 ↪color='skyblue')
ax[0].set_title(r'$R^2$ Promedio (Validación Cruzada)')
ax[0].set_xlabel('Modelo')
ax[0].set_ylabel(r'$R^2$ Promedio')
ax[0].set_ylim([0, 1])
for bar in bars1:
    yval = bar.get_height()
    ax[0].text(bar.get_x() + bar.get_width()/2, yval + 0.01, f'{yval:.3f}',␣
 ↪ha='center', va='bottom')

# Gráfico de MSE Promedio
bars2 = ax[1].bar(summary_cv['Modelo'], summary_cv['MSE_promedio'],␣
 ↪color='salmon')
ax[1].set_title('MSE Promedio (Validación Cruzada)')
```

```
ax[1].set_xlabel('Modelo')
ax[1].set_ylabel('MSE Promedio')
for bar in bars2:
    yval = bar.get_height()
    ax[1].text(bar.get_x() + bar.get_width()/2, yval + yval*0.01, f'{yval:.
 ↪3f}', ha='center', va='bottom')

plt.title('Resumen con las métricas promedio')
plt.tight_layout()
# Guardar la figura en la carpeta 'figure_path'
figure_file = os.path.join(figure_path, "Resumen con las métricas promedio.png")
plt.savefig(figure_file, dpi=1080)
plt.close()

# 3. Gráficos de líneas para comparar el desempeño por cada columna de salida.

# Gráfico para R2:
plt.figure(figsize=(8,5))
for model_name, metrics in metrics_results.items():
    # Extraemos la lista de nombres de columnas y sus valores de R2
    columns = list(metrics['r2_columns'].keys())
    r2_values = list(metrics['r2_columns'].values())
    # Se usa range(len(columns)) para el eje x y luego se asignan los ticks
    plt.plot(range(len(columns)), r2_values, marker='o', label=model_name)
plt.xlabel('Columna de salida')
plt.ylabel(r'$R^2$')
plt.title(r'$R^2$ por columna en Validación Cruzada')
plt.xticks(range(len(columns)), columns, rotation=45)
plt.legend()
plt.grid(True)
# Guardar la figura en la carpeta 'figure_path'
figure_file = os.path.join(figure_path, 'R2 por columna en Validación Cruzada.
 ↪png')
plt.savefig(figure_file, dpi=1080)
plt.close()

# Gráfico para RMSE:
plt.figure(figsize=(8,5))
for model_name, metrics in metrics_results.items():
    columns = list(metrics['rmse_columns'].keys())
    mse_values = list(metrics['rmse_columns'].values())
    plt.plot(range(len(columns)), mse_values, marker='o', label=model_name)
plt.xlabel('Columna de salida')
plt.ylabel('RMSE')
plt.title('RMSE por columna en Validación Cruzada')
plt.xticks(range(len(columns)), columns, rotation=45)
plt.legend()
```

```
plt.grid(True)
# Guardar la figura en la carpeta 'figure_path'
figure_file = os.path.join(figure_path, "RMSE por columna en Validación Cruzada.
 ↪png")
plt.savefig(figure_file, dpi=1080)
plt.close()

# Gráfico para MSE:
plt.figure(figsize=(8,5))
for model_name, metrics in metrics_results.items():
    columns = list(metrics['mse_columns'].keys())
    mse_values = list(metrics['mse_columns'].values())
    plt.plot(range(len(columns)), mse_values, marker='o', label=model_name)
plt.xlabel('Columna de salida')
plt.ylabel('MSE')
plt.title('MSE por columna en Validación Cruzada')
plt.xticks(range(len(columns)), columns, rotation=45)
plt.legend()
plt.grid(True)
# Guardar la figura en la carpeta 'figure_path'
figure_file = os.path.join(figure_path, "MSE por columna en Validación Cruzada.
 ↪png")
plt.savefig(figure_file, dpi=1080)
plt.close()

# Gráfico para MAE:
plt.figure(figsize=(8,5))
for model_name, metrics in metrics_results.items():
    columns = list(metrics['mae_columns'].keys())
    mse_values = list(metrics['mae_columns'].values())
    plt.plot(range(len(columns)), mse_values, marker='o', label=model_name)
plt.xlabel('Columna de salida')
plt.ylabel('MAE')
plt.title('MAE por columna en Validación Cruzada')
plt.xticks(range(len(columns)), columns, rotation=45)
plt.legend()
plt.grid(True)
# Guardar la figura en la carpeta 'figure_path'
figure_file = os.path.join(figure_path, "MAE por columna en Validación Cruzada.
 ↪png")
plt.savefig(figure_file, dpi=1080)
plt.close()

# Gráfico para CoP:
plt.figure(figsize=(8,5))
for model_name, metrics in metrics_results.items():
    columns = list(metrics['cop_columns'].keys())
```

```
    mse_values = list(metrics['cop_columns'].values())
    plt.plot(range(len(columns)), mse_values, marker='o', label=model_name)
plt.xlabel('Columna de salida')
plt.ylabel('CoP')
plt.title('CoP por columna en Validación Cruzada')
plt.xticks(range(len(columns)), columns, rotation=45)
plt.legend()
plt.grid(True)
# Guardar la figura en la carpeta 'figure_path'
figure_file = os.path.join(figure_path, "CoP por columna en Validación Cruzada.
 ↪png")
plt.savefig(figure_file, dpi=1000)
plt.close()
```

[14]:
```
# ================================================================================
# Paso 7: Seleccionar el mejor modelo (mayor CoP) para cada variable de salida
# ================================================================================
best_models_per_output = {}
for output in y_train.columns:
    # Buscar qué modelo tiene la CoP más alta para esta salida
    best_model = max(
        metrics_results.items(),
        key=lambda item: item[1]['cop_columns'][output]
    )[0]
    best_models_per_output[output] = {
        'model_name': best_model,
        'cop':  metrics_results[best_model]['cop_columns'][output],
        'r2':   metrics_results[best_model]['r2_columns'][output],
        'mse':  metrics_results[best_model]['mse_columns'][output]
    }
```

[15]:
```
# ================================================================================
# Paso 7.1: Representar el CoP obtenido para cada variable de salida y modelo.
# ================================================================================
# Imprimir por pantalla los valores de CoP para cada variable de salida y cada␣
 ↪modelo
print("Valores de CoP por variable de salida y modelo:")
for output in y_train.columns:
    print(f"\nSalida {output}:")
    for model_name in metrics_results:
        cop_val = metrics_results[model_name]['cop_columns'][output]
        print(f"  {model_name}: CoP = {cop_val:.4f}")

# Creamos el diccionario de CoP para pasar a la función de plot
cop_results = {
    output: {m: metrics_results[m]['cop_columns'][output] for m in␣
 ↪metrics_results}
```

```
    for output in y_train.columns
}
# Llamamos a la función ya definida para subplots de CoP
plot_cop_subplots(cop_results,
                  outputs=y_train.columns,
                  mejores_modelos={o: best_models_per_output[o]['model_name']␣
 ↪for o in y_train.columns},
                  figure_path=figure_path)
```

```
Valores de CoP por variable de salida y modelo:

Salida p1::W:
  PLS: CoP = 0.9746
  LR: CoP = 0.9750
  GPR: CoP = 0.9945
  SVR: CoP = 0.9950
  RF: CoP = 0.9357
  ANN: CoP = 0.9943
  ANN-K: CoP = 0.9928

Salida p4::GFF:
  PLS: CoP = 0.8007
  LR: CoP = 0.8008
  GPR: CoP = 0.9702
  SVR: CoP = 0.9734
  RF: CoP = 0.7537
  ANN: CoP = 0.9726
  ANN-K: CoP = 0.9702

Salida p5::BSP_T:
  PLS: CoP = 0.8527
  LR: CoP = 0.8529
  GPR: CoP = 0.9933
  SVR: CoP = 0.9941
  RF: CoP = 0.9429
  ANN: CoP = 0.9922
  ANN-K: CoP = 0.9897

Salida p6::BSP_n:
  PLS: CoP = 0.8058
  LR: CoP = 0.8061
  GPR: CoP = 0.9770
  SVR: CoP = 0.9818
  RF: CoP = 0.9492
  ANN: CoP = 0.9841
  ANN-K: CoP = 0.9855

Salida p7::BSP_Pm:
```

```
  PLS: CoP = 0.9606
  LR: CoP = 0.9606
  GPR: CoP = 0.9858
  SVR: CoP = 0.9876
  RF: CoP = 0.9702
  ANN: CoP = 0.9857
  ANN-K: CoP = 0.9841

Salida p8::BSP_Mu:
  PLS: CoP = 0.7563
  LR: CoP = 0.7562
  GPR: CoP = 0.9728
  SVR: CoP = 0.9776
  RF: CoP = 0.8917
  ANN: CoP = 0.9828
  ANN-K: CoP = 0.9785

Salida p9::BSP_Irms:
  PLS: CoP = 0.9903
  LR: CoP = 0.9903
  GPR: CoP = 0.9899
  SVR: CoP = 0.9901
  RF: CoP = 0.9734
  ANN: CoP = 0.9890
  ANN-K: CoP = 0.9883

Salida p10::MSP_n:
  PLS: CoP = 0.8195
  LR: CoP = 0.8197
  GPR: CoP = 0.9778
  SVR: CoP = 0.9827
  RF: CoP = 0.9510
  ANN: CoP = 0.9842
  ANN-K: CoP = 0.9854

Salida p11::UWP_Mu:
  PLS: CoP = 0.4478
  LR: CoP = 0.4468
  GPR: CoP = 0.9689
  SVR: CoP = 0.9736
  RF: CoP = 0.8191
  ANN: CoP = 0.9844
  ANN-K: CoP = 0.9802
Figura CoP guardada en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\No
tebooks_TFM\3.MOP\Figuras_MOP\5000_MOT_Uniforme\CoP_para_cada_modelo.png
```

[16]:
```python
# ============================================================================
# Definición de un wrapper para desescalar la predicción del target
# ============================================================================
from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.metrics import r2_score

class DescaledRegressor(BaseEstimator, RegressorMixin):
    """
    Wrapper para un modelo cuya salida se entrenó sobre y escalado y que,
    al predecir, se desescala automáticamente usando el target_scaler.
    """
    def __init__(self, estimator, target_scaler):
        self.estimator = estimator  # Modelo previamente entrenado (pipeline)
        self.target_scaler = target_scaler  # Escalador entrenado sobre y_train

    def predict(self, X):
        # Se predice en la escala del target (y escalado)
        y_pred_scaled = self.estimator.predict(X)
        # Se aplica la transformación inversa para recuperar la escala original
        return self.target_scaler.inverse_transform(y_pred_scaled)

    def fit(self, X, y):
        # Aunque el modelo ya esté entrenado, este método permite reentrenarlo
        y_scaled = self.target_scaler.transform(y)
        self.estimator.fit(X, y_scaled)
        return self

    def score(self, X, y):
        # Calcula R² usando las predicciones ya desescaladas
        y_pred = self.predict(X)
        return r2_score(y, y_pred)

class SingleOutputDescaledRegressor(BaseEstimator, RegressorMixin):
    """
    Wrapper para obtener la predicción de un modelo multioutput
    para una variable de salida particular y desescalarla usando el
    target_scaler. Se utiliza el índice de la columna deseada.
    """
    def __init__(self, estimator, target_scaler, col_index):
        self.estimator = estimator            # Modelo multioutput previamente␣
↪entrenado
        self.target_scaler = target_scaler    # Escalador entrenado sobre␣
↪y_train
        self.col_index = col_index            # Índice de la variable de salida

    def predict(self, X):
```

```
        # Se predice con el modelo multioutput; se obtiene la predicción en␣
↪escala (2D array)
        y_pred_scaled = self.estimator.predict(X)
        # Se extrae la predicción para la columna de interés
        single_pred_scaled = y_pred_scaled[:, self.col_index]
        # Se recuperan los parámetros del escalador para la columna
        scale_val = self.target_scaler.scale_[self.col_index]
        mean_val = self.target_scaler.mean_[self.col_index]
        # Desescalar manualmente: valor original = valor escalado * escala +␣
↪media
        y_pred_original = single_pred_scaled * scale_val + mean_val
        return y_pred_original

    def fit(self, X, y):
        # (Opcional) Si se desea reentrenar el modelo, se transforma y y se␣
↪ajusta
        y_scaled = self.target_scaler.transform(y)
        self.estimator.fit(X, y_scaled)
        return self

    def score(self, X, y):
        from sklearn.metrics import r2_score
        y_pred = self.predict(X)
        return r2_score(y, y_pred)

class UnifiedDescaledRegressor(BaseEstimator, RegressorMixin):
    """
    Modelo que encapsula un diccionario de modelos individuales (por variable␣
↪de salida).
    Cada modelo (del tipo SingleOutputDescaledRegressor) se utiliza para␣
↪predecir su variable
    de salida correspondiente y se realiza la transformación inversa para␣
↪retornar el valor original.
    """
    def __init__(self, models):
        """
        :param models: diccionario con llave = etiqueta de salida y valor =␣
↪SingleOutputDescaledRegressor.
        """
        self.models = models
        # Se conserva el orden de salida en función de las claves del␣
↪diccionario;
        # se asume que estas claves son exactamente las mismas que aparecen en␣
↪y_test.
        self.output_columns = list(models.keys())
```

```
    def predict(self, X):
        preds = []
        # Se predice para cada variable en el orden de self.output_columns
        for col in self.output_columns:
            model = self.models[col]
            pred = model.predict(X)  # cada predicción es un array de forma␣
 ↪(n_samples,)
            preds.append(pred)
        # Se combinan las predicciones columna a columna para formar un array␣
 ↪(n_samples, n_targets)
        return np.column_stack(preds)

    def score(self, X, y):
        from sklearn.metrics import r2_score
        y_pred = self.predict(X)
        return r2_score(y, y_pred, multioutput='uniform_average')
```

[17]:
```
# ================================================================================
# Paso 8: Seleccionar el mejor modelo (mayor R²) para cada variable de salida
# ================================================================================
best_models_per_output = {}
for output in y_train.columns:
    # Elegimos el modelo que maximiza el R² para esta salida
    best_mod = max(
        pipelines.keys(),
        key=lambda m: metrics_results[m]['r2_columns'][output]
    )
    best_models_per_output[output] = {
        'model_name': best_mod,
        'r2': metrics_results[best_mod]['r2_columns'][output],
        'mse': metrics_results[best_mod]['mse_columns'][output],
        'rmse': metrics_results[best_mod]['rmse_columns'][output]
    }
print("Mejores modelos por variable de salida (R²):")
for out, info in best_models_per_output.items():
    print(f"{out}: Modelo = {info['model_name']}, R² = {info['r2']:.3f}")
```

```
Mejores modelos por variable de salida (R²):
p1::W: Modelo = SVR, R² = 0.995
p4::GFF: Modelo = SVR, R² = 0.973
p5::BSP_T: Modelo = SVR, R² = 0.994
p6::BSP_n: Modelo = ANN-K, R² = 0.985
p7::BSP_Pm: Modelo = SVR, R² = 0.988
p8::BSP_Mu: Modelo = ANN, R² = 0.983
p9::BSP_Irms: Modelo = LR, R² = 0.990
p10::MSP_n: Modelo = ANN-K, R² = 0.985
p11::UWP_Mu: Modelo = ANN, R² = 0.984
```

[18]:
```
#==============================================================================
# Paso 8.1: Guardamos cada modelo de forma independiente.
#==============================================================================
for name, est in trained_estimators.items():
    final_model = est.best_estimator_ if hasattr(est, "best_estimator_") else␣
 ↪est
    filepath = os.path.join(modelo_path, f"{clean_filename(name)}.joblib")
    joblib.dump(final_model, filepath)
    print(f"Guardado {name} en {filepath}")
```

```
Guardado PLS en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks
_TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\PLS.joblib
Guardado LR en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_
TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\LR.joblib
Guardado GPR en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks
_TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\GPR.joblib
Guardado SVR en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks
_TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\SVR.joblib
Guardado RF en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_
TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\RF.joblib
Guardado ANN en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks
_TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\ANN.joblib
Guardado ANN-K en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Noteboo
ks_TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\ANN-K.joblib
```

[19]:
```
# ==============================================================================
# Paso 8.2: ALMACENAMIENTO DEL MODELO FINAL "DESESCADO"
# ==============================================================================
descaled_models = {}
for idx, output in enumerate(y_train.columns):
    best_name = best_models_per_output[output]['model_name']
    # 1) Ruta al modelo ya guardado
    path = os.path.join(modelo_path, f"{clean_filename(best_name)}.joblib")
    # 2) Cargar el pipeline que SÍ está ajustado
    fitted_pipeline = joblib.load(path)
    # 3) Envolverlo en SingleOutputDescaledRegressor
    descaled_models[output] = SingleOutputDescaledRegressor(
        estimator=fitted_pipeline,
        target_scaler=target_scaler,
        col_index=idx
    )

# 4) Crear y guardar el modelo unificado
unified_model = UnifiedDescaledRegressor(descaled_models)
unif_path = os.path.join(modelo_path, "MOP_descaled_unified.joblib")
joblib.dump(unified_model, unif_path)
print(f"Modelo unificado desescalado guardado en {unif_path}")
```

```
Modelo unificado desescalado guardado en C:\Users\s00244\Documents\GitHub\MotorD
esignDataDriven\Notebooks_TFM\3.MOP\Modelos_MOP\5000_MOT_Uniforme\MOP_descaled_u
nified.joblib
```

[20]:
```python
# ================================================================================
# Paso 8.3: CARGAR EL MODELO GUARDADO Y REALIZAR PREDICCIONES
# ================================================================================
loaded_unified = joblib.load(unif_path)
```

[21]:
```python
# ================================================================================
# PASO 8.4: CALCULAR LAS PREDICCIONES DE CADA MODELO
# ================================================================================
y_pred_test = loaded_unified.predict(X_test)
df_pred = pd.DataFrame(
    y_pred_test,
    index=X_test.index,
    columns=y_test.columns
)
print("Predicciones realizadas para X_test.")

# Mostrar una comparación de predicciones vs. valores reales, utilizando las␣
 ↪etiquetas
print("\nEjemplo de predicciones (valores reales vs. predichos):")
for col in y_test.columns:
    print(f"\nColumna: {col}")
    comp_df = pd.DataFrame({
        "Real": y_test[col],
        "Predicho": df_pred[col]
    })
    print(comp_df.head())
```

```
Predicciones realizadas para X_test.

Ejemplo de predicciones (valores reales vs. predichos):

Columna: p1::W
          Real  Predicho
1553  0.516093  0.516356
2986  0.439500  0.439389
220   0.701446  0.703187
2965  0.662162  0.663059
1971  0.581912  0.581516

Columna: p4::GFF
           Real   Predicho
1553  36.878147  36.419003
2986  37.621853  37.447729
220   43.636204  43.746485
```

```
2965  34.102028  34.578724
1971  45.624140  46.137530

Columna: p5::BSP_T
          Real  Predicho
1553  0.414659  0.416042
2986  0.330263  0.330572
220   0.708157  0.712013
2965  0.418287  0.418361
1971  0.626960  0.624207

Columna: p6::BSP_n
            Real      Predicho
1553  10460.6840  10276.944869
2986   6445.2510   6281.278663
220    5098.5620   5394.655905
2965   8612.3780   8489.178553
1971   3813.9675   3599.725165

Columna: p7::BSP_Pm
           Real    Predicho
1553  454.23386  454.347938
2986  222.90920  224.044513
220   378.09915  380.129502
2965  377.24774  377.986217
1971  250.40652  251.529064

Columna: p8::BSP_Mu
           Real   Predicho
1553  89.800910  89.725520
2986  89.025635  88.868842
220   90.660515  90.640631
2965  87.769660  87.701913
1971  82.303090  82.488511

Columna: p9::BSP_Irms
           Real   Predicho
1553  15.106918  15.055372
2986   7.552780   7.537300
220   12.587974  12.595587
2965  12.587929  12.559873
1971  12.589243  12.630047

Columna: p10::MSP_n
            Real      Predicho
1553  11283.0860  11032.088333
2986   7195.7750   7079.114283
220    5643.4854   5937.470313
```

```
2965   8977.9040   9078.477253
1971   6303.4717   6030.979681

Columna: p11::UWP_Mu
           Real    Predicho
1553  87.366745  87.329436
2986  89.442310  89.405564
220   87.649370  87.482574
2965  82.228450  81.975338
1971  91.269240  91.534131
```

[22]:
```
# ============================================================================
# Paso 9: Calcular y representar mapas de calor de p   (correlación entre␣
 ↪predicción y variables de entrada)
# ============================================================================
# Se usa la función compute_corr(y, x) definida en la plantilla para calcular␣
 ↪p_ij:
# p_ij = (1/(N-1)) * Σ[(ŷ(k) - _ŷ)*(x_j(k) - _xj)] / (_ŷ  _xj)
#
# Se decide calcular p_ij usando el modelo seleccionado (el mejor para cada␣
 ↪salida).
# Para cada variable de salida, se tomarán las predicciones del mejor modelo y␣
 ↪se calculará la correlación
# entre esas predicciones y cada una de las variables de entrada.

# Inicializar la matriz de pij donde filas corresponden a cada variable de␣
 ↪entrada y columnas a cada salida.
n_inputs  = X_train.shape[1]
n_outputs = y_train.shape[1]
# Inicializar matriz p_ij
pij_matrix = np.zeros((n_inputs, n_outputs))

for j, output in enumerate(y_train.columns):
    model_name = best_models_per_output[output]['model_name']
    # Predecir con el modelo (ya ajustado) sobre X_train
    y_pred_scaled = trained_estimators[model_name].predict(X_train)
    # Si es multioutput extraemos la j-ésima columna
    y_pred_j = y_pred_scaled[:, j] if y_pred_scaled.ndim == 2 else y_pred_scaled
    for i, input_var in enumerate(X_train.columns):
        pij_matrix[i, j] = compute_corr(y_pred_j, X_train[input_var])

# Generar el mapa de calor y guardarlo
pij_heatmap_file = os.path.join(figure_path,␣
 ↪"Heatmap_pij_prediccion_vs_entradas.png")
ax = plot_heatmap(
    pij_matrix,
    col_labels=y_train.columns,
```

```
    row_labels=X_train.columns,
    title="p_ij: Correlación Predicción vs Entradas"
)
plt.savefig(pij_heatmap_file, dpi=1000)
plt.close()

# Imprimir por pantalla la ruta del archivo generado
print(f'Heatmap p_ij guardado en {pij_heatmap_file}')
```

```
Heatmap p_ij guardado en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\
Notebooks_TFM\3.MOP\Figuras_MOP\5000_MOT_Uniforme\Heatmap_pij_prediccion_vs_entr
adas.png
```

[23]:
```
# ================================================================================
# Paso 10: Calcular y representar el mapa de calor de la correlación de Pearson␣
 ↪(entradas vs salidas)
# ================================================================================
corr_mat = pd.concat([X, y], axis=1).corr()
corr_Xy   = corr_mat.loc[entradas, salidas].values
plot_heatmap(corr_Xy, col_labels=salidas, row_labels=entradas,
             title='Correlación Pearson X vs Y')
plt.savefig(os.path.join(figure_path, 'Correlacion_X_Y.png'), dpi=1000)
plt.close()
print('Mapa de calor de correlación Pearson guardado.')
```

```
Mapa de calor de correlación Pearson guardado.
```

[24]:
```
# ================================================================================
# Paso 13: REPRESENTACIÓN DE RESULTADOS: GRÁFICO DE PREDICCIÓN VS REAL
# ================================================================================
# Número de variables de salida a graficar (usando las etiquetas de y_test)
num_vars = len(y_test.columns)
n_cols = 3  # Número deseado de columnas en el subplot (se puede ajustar)
n_rows = math.ceil(num_vars / n_cols)  # Número de filas

plt.figure(figsize=(5 * n_cols, 4 * n_rows))

for idx, col in enumerate(y_test.columns):
    ax = plt.subplot(n_rows, n_cols, idx + 1)

    # Graficar scatter de valores reales vs. predichos para la variable 'col'
    ax.scatter(y_test[col], df_pred[col], alpha=0.7)

    # Definir los límites para la línea de identidad (diagonal)
    min_val = min(y_test[col].min(), df_pred[col].min())
    max_val = max(y_test[col].max(), df_pred[col].max())
    ax.plot([min_val, max_val], [min_val, max_val], 'r--')
```

```
    # Recuperar la información del modelo escogido para esa variable
    chosen_model = best_models_per_output[col]['model_name']
    r2_val = best_models_per_output[col]['r2']
    mse_val = best_models_per_output[col]['mse']
    rmse_val = best_models_per_output[col]['rmse']

    # Añadir el texto con el modelo escogido, R² y MSE en la esquina superior␣
 ↪izquierda del subplot
    ax.text(0.05, 0.95,
            f"Modelo: {chosen_model}\nR2: {r2_val:.3f}\nMSE: {mse_val:.
 ↪3g}\nRMSE: {rmse_val:.3g}",
            transform=ax.transAxes,
            fontsize=10,
            verticalalignment='top',
            bbox=dict(facecolor='white', alpha=0.6))

    ax.set_xlabel("Valor Real")
    ax.set_ylabel("Valor Predicho")
    ax.set_title(col)

plt.title('Comparacion_FEA_vs_Predicciones')
plt.tight_layout()
# Guardar la figura en la carpeta 'figure_path'
figure_file = os.path.join(figure_path, "Comparacion_FEA_vs_Predicciones.png")
plt.savefig(figure_file, dpi=1000)
plt.close()
```

[25]:
```
# ============================================================================
# Paso 14. REPRESENTACIÓN DE RESULTADOS: GRÁFICO DE PREDICCIÓN VS REAL (con␣
 ↪métricas desescaladas)
# ============================================================================

# 2) Calcular métricas sobre valores desescalados
mse_des  = mean_squared_error(y_test, y_pred_test, multioutput="raw_values")
rmse_des = np.sqrt(mse_des)
mae_des  = mean_absolute_error(y_test, y_pred_test, multioutput="raw_values")
r2_des   = r2_score(y_test, y_pred_test, multioutput="raw_values")
cop_des  = np.array([
    compute_CoP(y_test.iloc[:, j].values, y_pred_test[:, j])
    for j in range(y_test.shape[1])
])

# 3) Imprimir métricas detalladas
print("Métricas de test en escala original (desescalado):")
for j, col in enumerate(y_test.columns):
    print(f"{col}: RMSE = {rmse_des[j]:.3f}, MAE = {mae_des[j]:.3f}, R² =␣
 ↪{r2_des[j]:.3f}, CoP = {cop_des[j]:.3f}")
```

```
print(f"\nPromedios: RMSE = {rmse_des.mean():.3f}, MAE = {mae_des.mean():.3f},␣
 ↪R² = {r2_des.mean():.3f}, CoP = {cop_des.mean():.3f}\n")

# 4) Gráfico scatter predicho vs real con anotaciones de modelo y R²/MSE/RMSE␣
 ↪originales
num_vars = len(y_test.columns)
n_cols   = 3
n_rows   = math.ceil(num_vars / n_cols)
plt.figure(figsize=(5 * n_cols, 4 * n_rows))

for idx, col in enumerate(y_test.columns):
    ax = plt.subplot(n_rows, n_cols, idx + 1)
    ax.scatter(y_test[col], df_pred[col], alpha=0.7)

    # Línea identidad
    min_val = min(y_test[col].min(), df_pred[col].min())
    max_val = max(y_test[col].max(), df_pred[col].max())
    ax.plot([min_val, max_val], [min_val, max_val], 'r--')

    # Recuperar info del mejor modelo para esta salida
    chosen = best_models_per_output[col]['model_name']
    r2_val = r2_des[list(y_test.columns).index(col)]  # R2 desescalado
    mse_val = mse_des[list(y_test.columns).index(col)]  # MSE desescalado
    rmse_val = rmse_des[list(y_test.columns).index(col)]  # RMSE desescalado
    cop_val  = cop_des[list(y_test.columns).index(col)]   # CoP desescalado

    ax.text(0.05, 0.95,
            f"Modelo: {chosen}\nR²_des: {r2_val:.3f}\nMSE_des: {mse_val:.3g}\n"
            f"RMSE_des: {rmse_val:.3f}\nCoP_des: {cop_val:.3f}",
            transform=ax.transAxes, fontsize=9, verticalalignment='top',
            bbox=dict(facecolor='white', alpha=0.6))

    ax.set_xlabel("Valor Real")
    ax.set_ylabel("Valor Predicho")
    ax.set_title(col)

plt.suptitle('Comparación Real vs Predicción (valores desescalados)', y=1.02)
plt.tight_layout()

# 5) Guardar figura
figure_file = os.path.join(figure_path,␣
 ↪"Comparacion_FEA_vs_Predicciones_Desescalado.png")
plt.savefig(figure_file, dpi=1000, bbox_inches='tight')
plt.close()

print(f'Figura guardada en {figure_file}')
```

```
Métricas de test en escala original (desescalado):
p1::W: RMSE = 0.029, MAE = 0.004, R² = 0.966, CoP = 0.966
p4::GFF: RMSE = 2.624, MAE = 0.609, R² = 0.942, CoP = 0.943
p5::BSP_T: RMSE = 0.047, MAE = 0.007, R² = 0.961, CoP = 0.962
p6::BSP_n: RMSE = 984.609, MAE = 303.191, R² = 0.962, CoP = 0.963
p7::BSP_Pm: RMSE = 11.886, MAE = 2.543, R² = 0.992, CoP = 0.992
p8::BSP_Mu: RMSE = 0.697, MAE = 0.202, R² = 0.935, CoP = 0.937
p9::BSP_Irms: RMSE = 0.403, MAE = 0.066, R² = 0.993, CoP = 0.993
p10::MSP_n: RMSE = 1063.388, MAE = 318.726, R² = 0.963, CoP = 0.964
p11::UWP_Mu: RMSE = 0.791, MAE = 0.252, R² = 0.921, CoP = 0.925

Promedios: RMSE = 229.386, MAE = 69.511, R² = 0.959, CoP = 0.960

Figura guardada en C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebo
oks_TFM\3.MOP\Figuras_MOP\5000_MOT_Uniforme\Comparacion_FEA_vs_Predicciones_Dese
scalado.png
```

[26]:
```python
# ================================================================================
# Fin del código
# ================================================================================
# Tiempo de fin
end_time_program = time.time()
program_time = end_time_program - start_time_program  # Tiempo transcurrido en␣
 ↪segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{program_time:.2f} segundos")
print("Ejecución completada.")
```

```
Tiempo de computación del entrenamiento de hiperparámetros: 20724.03 segundos
Ejecución completada.
```

[ ]:
```python

```