DBG_V5

May 22, 2025

[1]:
```
# Librerías necesarias
import os
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
from mpl_toolkits.mplot3d import Axes3D

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
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

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

# Para guardar y cargar modelos
import joblib
```

[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]:
```
# ============================================================================
# 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')
```

[4]:
```
# ============================================================================
# 1. CARGA DE DATOS Y PREPARACIÓN DEL DATAFRAME
# ============================================================================
# Definir las rutas base y de las carpetas
base_path = os.getcwd()  # Se asume que el notebook se ejecuta desde la carpeta␣
 ↪'DBG'
db_path = os.path.join(base_path, "DB_DBG")
fig_path = os.path.join(base_path, "Figuras_DBG")
model_path = os.path.join(base_path, "Modelos_DBG")

# Ruta al archivo de la base de datos
data_file = os.path.join(db_path, "design_DB_preprocessed_1000_Uniforme.csv")
print(data_file)

# Ruta al archivo de las figuras
figure_path = os.path.join(fig_path, "1000_MOT_Uniforme")
print(figure_path)

# Ruta al archivo de los modelos
modelo_path = os.path.join(model_path, "1000_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\4.DBG\DB_DB
G\design_DB_preprocessed_1000_Uniforme.csv
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Figur
as_DBG\1000_MOT_Uniforme
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Model
os_DBG\1000_MOT_Uniforme
Archivo cargado exitosamente.
```

[5]:
```
# ================================================================================
# 2. SEPARACIÓN DE VARIABLES
# ================================================================================
# 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')
```

```python
# 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]

features = list(X.columns)
outputs = [col for col in Y.columns]

print("\nColumnas de salida tras eliminar las constantes o casi constantes:")
print(Y.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           2.413512e-02
p2::Tnom        1.928477e-34
p3::nnom        0.000000e+00
p4::GFF         1.228099e+02
p5::BSP_T       5.206690e-02
p6::BSP_n       2.567653e+07
p7::BSP_Pm      1.719746e+04
p8::BSP_Mu      6.758214e+00
p9::BSP_Irms    2.226090e+01
p10::MSP_n      3.054785e+07
p11::UWP_Mu     1.010653e+01
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']
```

[6]:
```python
# ============================================================================
# Paso 3: Definir el modelo ANN_K para que pueda leerse
# ============================================================================
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_DBG.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 = {}

# 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", {}),
}

akk_par = hiperparametros['ANN-K']
bs = akk_par.get('model__batch_size')
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
)
```

```
Hiperparámetros cargados desde C:\Users\s00244\Documents\GitHub\MotorDesignDataD
riven\Notebooks_TFM\4.DBG\Modelos_DBG\1000_MOT_Uniforme\hiperparametros_DBG.json
```

[7]:
```
# ============================================================================
# Paso 4: Generar 10,000 nuevos motores a partir de los rangos de entrada
# ============================================================================
# Las restricciones (Boundaries B) se definen sobre las variables de X y de M.
# Definir la función check_boundaries escalable: se evalúan todas las␣
 ↪condiciones definidas en una lista.
def check_boundaries(row):
    boundaries = [
        lambda r: r['x1::OSD'] > r['x2::Dint'],  # Boundarie_1: x1 debe ser␣
 ↪mayor que x2
        lambda r: 45.0 < r['x1::OSD'] < 60.0,           # Boundarie_2: x1 debe␣
 ↪estar entre 45 y 60.
        lambda r: ((r['x2::Dint']-2*0.5)-2*r['x4::tm']-r['x2::Dint']/3.5) >= 8,␣
 ↪ # Boundarie_3: Dsh debe ser mayor 8 mm. Un eje muy esbelto puede flectar.
        lambda r: ((r['x1::OSD']/2)-(r['x2::Dint']+2*r['x5::hs2'])/2) >= 3.5, ␣
 ↪# Boundarie_4: he debe ser mayor 3.5 mm. Puede romper si es muy delgado.
        # Aquí se pueden agregar más condiciones según se requiera
    ]
    return all(condition(row) for condition in boundaries)

# Función para generar muestras considerando si la variable debe ser entera
def generate_samples(n_samples):
    data = {}
    for col in X_cols:
        # Si la variable es una de las que deben ser enteras, usar randint
        if col in ['x7::Nt', 'x8::Nh']:
            low = int(np.floor(X_min[col]))
            high = int(np.ceil(X_max[col]))
            # np.random.randint es exclusivo en el extremo superior, por lo que␣
 ↪se suma 1
            data[col] = np.random.randint(low=low, high=high+1, size=n_samples)
        else:
            data[col] = np.random.uniform(low=X_min[col], high=X_max[col],␣
 ↪size=n_samples)
    return pd.DataFrame(data)

# Guardamos los valores máximos y mínimos
X_min = df[features].min()
X_max = df[features].max()

desired_samples = 10000
valid_samples_list = []
# Generamos muestras en bloques; para aumentar la probabilidad de cumplir las␣
 ↪restricciones,
# se genera un bloque mayor al deseado
batch_size = int(desired_samples * 1.5)
```

```
# Acumular muestras válidas hasta obtener el número deseado
while sum(len(df_batch) for df_batch in valid_samples_list) < desired_samples:
    X_batch = generate_samples(batch_size)
    X_valid_batch = X_batch[X_batch.apply(check_boundaries, axis=1)]
    valid_samples_list.append(X_valid_batch)

# Concatenar todas las muestras válidas y truncar a desired_samples
valid_samples = pd.concat(valid_samples_list).reset_index(drop=True)
X_new = valid_samples.iloc[:desired_samples].copy()
print(f"Se generaron {len(X_new)} muestras de X que cumplen con las␣
 ↪restricciones de Boundaries B (objetivo: {desired_samples}).")
display(X_new.head())
```

```
Se generaron 10000 muestras de X que cumplen con las restricciones de Boundaries
B (objetivo: 10000).
```

	x1::OSD	x2::Dint	x3::L	x4::tm	x5::hs2	x6::wt	x7::Nt	\
0	59.109233	31.737597	31.857104	3.038212	8.100607	2.862901	19	
1	59.328662	25.186221	36.345354	3.184483	5.897683	2.917327	10	
2	56.160596	27.553930	15.929237	2.550499	8.694059	4.861692	29	
3	57.901684	31.469801	28.003841	2.484687	7.803684	4.070863	5	
4	57.872131	28.719523	22.667203	3.251350	6.544124	2.336454	20	

	x8::Nh
0	9
1	5
2	6
3	4
4	8

[8]:
```
# ============================================================================
# Paso 4.1: Generamos la matriz M de funciones de X
# ============================================================================
M_new = pd.DataFrame()
# Utilizamos los boundaries relevantes (se asume que B tiene al menos 'b11::g',␣
 ↪etc.)
M_new['m1::Drot'] = X_new['x2::Dint'] - 2 * 0.5
M_new['m2::Dsh'] = M_new['m1::Drot'] - 2 * X_new['x4::tm'] - X_new['x2::Dint'] /
 ↪ 3.5
M_new['m3::he'] = (X_new['x1::OSD'] / 2) - (X_new['x2::Dint'] + 2 * X_new['x5::
 ↪hs2']) / 2
M_new['m4::Rmag'] = (M_new['m1::Drot'] / 2) - 0.25 * X_new['x4::tm']
M_new['m5::Rs'] = (X_new['x2::Dint'] / 2) + X_new['x5::hs2']
# Calcular el Gross Fill Factor (GFF) como ejemplo (puede ajustarse según el␣
 ↪caso)
CS = 2 * X_new['x7::Nt'] * X_new['x8::Nh'] * np.pi * (0.51 / 2) ** 2
```

```
SS = (np.pi * M_new['m5::Rs']**2 - np.pi * (X_new['x2::Dint'] / 2)**2) / 12 -␣
 ↪X_new['x6::wt'] * X_new['x5::hs2']
M_new['m6::GFF'] = 100 * (CS / SS)
```

[9]:
```
# ================================================================================
# Paso 5: Cargar modelo final desescalado y predecir
# ================================================================================
model_filename = os.path.join(modelo_path, f"DBG_descaled_unified.joblib")
print(model_filename)
loaded_model = joblib.load(model_filename)

# Predicción en la escala original
y_pred = loaded_model.predict(X_new)
```

```
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Model
os_DBG\1000_MOT_Uniforme\DBG_descaled_unified.joblib
```

[10]:
```
# ================================================================================
# Paso 6: Escalado de datos
# ================================================================================
scaler_X = StandardScaler()
X_scaled = scaler_X.fit_transform(X_new)
scaler_Y = StandardScaler()
Y_scaled = scaler_Y.fit_transform(Y)

# Crear DataFrames escalados completos (para reentrenamiento final y␣
 ↪predicciones)
X_scaled_df = X_new
Y_scaled_df = Y
```

[11]:
```
# --------------------------------------------------------------------------------
# Definir una clase que encapsule el ensemble de los mejores modelos
# --------------------------------------------------------------------------------
class BestModelEnsemble:
    def __init__(self, model_dict, outputs):
        """
        model_dict: Diccionario que mapea cada variable de salida a una tupla␣
 ↪(modelo, índice)
                    donde 'modelo' es el mejor modelo para esa salida y␣
 ↪'índice' es la posición
                    de esa salida en el vector de predicción que produce ese␣
 ↪modelo.
        outputs: Lista de nombres de variables de salida, en el orden deseado.
        """
        self.model_dict = model_dict
        self.outputs = outputs
```

```
    def predict(self, X):
        """
        Realiza la predicción para cada variable de salida usando el modelo␣
↪asignado.
        Se espera que cada modelo tenga un método predict que devuelva un array␣
↪de
        dimensiones (n_samples, n_outputs_model). Si el modelo es univariable,␣
↪se asume
        que devuelve un array 1D.

        :param X: Datos de entrada (array o DataFrame) con la forma (n_samples,␣
↪n_features).
        :return: Array con la predicción para todas las variables de salida,␣
↪forma (n_samples, n_outputs).
        """
        n_samples = X.shape[0]
        n_outputs = len(self.outputs)
        preds = np.zeros((n_samples, n_outputs))

        # Iterar sobre cada variable de salida
        for output in self.outputs:
            model, idx = self.model_dict[output]
            model_pred = model.predict(X)
            # Si el modelo es univariable, model_pred es 1D; de lo contrario,␣
↪es 2D
            if model_pred.ndim == 1:
                preds[:, self.outputs.index(output)] = model_pred
            else:
                preds[:, self.outputs.index(output)] = model_pred[:, idx]
        return preds
```

[12]:
```
# ============================================================================
# Paso 7: Preprocesar los nuevos datos con el mismo escalador usado en␣
↪entrenamiento
# ============================================================================
# Convertir las predicciones a la escala original
preds_original = y_pred
display(preds_original[0])

# Combinar las predicciones en un DataFrame
df_predictions = pd.DataFrame(preds_original, columns=outputs)

# Combinar las variables de entrada originales y las salidas predichas
mot = pd.concat([X_new, M_new], axis=1)
motors = pd.concat([mot, df_predictions], axis=1)
display(motors.head(15))
```

```
array([   1.07234973,  121.68340057,    2.19548776, -899.79023511,
         455.8755681 ,   80.37661502,   22.62517856,  414.86152757,
          88.6456328 ])
```

	x1::OSD	x2::Dint	x3::L	x4::tm	x5::hs2	x6::wt	x7::Nt	\
0	59.109233	31.737597	31.857104	3.038212	8.100607	2.862901	19	
1	59.328662	25.186221	36.345354	3.184483	5.897683	2.917327	10	
2	56.160596	27.553930	15.929237	2.550499	8.694059	4.861692	29	
3	57.901684	31.469801	28.003841	2.484687	7.803684	4.070863	5	
4	57.872131	28.719523	22.667203	3.251350	6.544124	2.336454	20	
5	52.032954	25.642214	19.145558	3.351096	8.292027	4.708284	10	
6	59.022678	30.102536	20.725107	3.232948	8.300313	4.880363	26	
7	59.084247	23.506281	33.429104	2.176902	12.881647	3.738299	10	
8	59.755479	29.172891	20.501322	2.071435	7.684031	2.163283	30	
9	58.495452	23.956231	22.701931	3.328490	13.620656	2.710177	29	
10	54.892270	27.079011	32.339341	3.050077	8.790003	2.443443	8	
11	59.557619	29.147132	17.652638	3.129924	5.193074	2.318498	17	
12	55.110892	26.975004	33.937222	2.244937	6.909910	4.422841	11	
13	56.524556	32.208991	22.513839	3.210961	8.131219	3.047029	29	
14	57.114990	33.602851	32.270730	2.398497	6.025044	3.636245	15	

	x8::Nh	m1::Drot	m2::Dsh	…	m6::GFF	p1::W	p4::GFF	\
0	9	30.737597	15.593288	…	113.980694	1.072350	121.683401	
1	5	24.186221	10.621192	…	66.350372	0.949640	79.164804	
2	6	26.553930	13.580381	…	176.682147	0.663990	118.831479	
3	4	30.469801	16.509056	…	16.859131	0.633743	18.483142	
4	8	27.719523	13.011245	…	144.863821	0.831519	129.987080	
5	9	24.642214	10.613676	…	106.197637	0.551859	94.679272	
6	7	29.102536	14.035915	…	173.162736	0.833932	126.125566	
7	3	22.506281	11.436397	…	16.439126	0.696591	17.571305	
8	9	28.172891	15.694910	…	191.776137	0.950668	133.769191	
9	9	22.956231	9.454615	…	109.842356	0.909935	100.507478	
10	4	26.079011	12.241997	…	21.410224	0.635098	24.593303	
11	7	28.147132	13.559531	…	140.327589	0.680010	124.018366	
12	9	25.975004	13.777985	…	131.594558	0.920568	115.591728	
13	3	31.208991	15.584499	…	58.177074	0.626660	69.087639	
14	7	32.602851	18.205043	…	105.666676	0.935673	111.126049	

	p5::BSP_T	p6::BSP_n	p7::BSP_Pm	p8::BSP_Mu	p9::BSP_Irms	\
0	2.195488	-899.790235	455.875568	80.376615	22.625179	
1	0.786514	3929.156938	355.540030	89.221634	12.580649	
2	0.990222	3.527924	198.335769	77.244850	14.958926	
3	0.303060	9698.199115	304.440999	89.972861	10.057482	
4	1.267237	1118.107567	416.918003	80.546046	20.088639	
5	0.730050	6639.036286	525.634760	90.026268	22.623560	
6	1.472640	-1020.992272	258.078796	78.270538	17.490562	
7	0.391729	5171.823947	211.433877	87.473022	7.568995	
8	1.050829	-1929.044073	259.761539	75.457806	22.323881	

```
9    1.150322 -1405.642051  246.165757   73.158322     22.359127
10   0.446300  6355.251622  300.436897   88.765449     10.057775
11   0.843218  3236.366880  433.204842   85.452662     17.581354
12   1.467850  2610.183051  498.616377   87.017205     22.637501
13   0.942382   599.422861  187.331615   77.982601      7.592398
14   1.828527  1134.229564  426.674349   84.610137     17.626277

      p10::MSP_n  p11::UWP_Mu
0     414.861528    88.645633
1    4947.391460    92.228441
2    3205.291516    95.466488
3   10232.612990    87.166049
4    3684.173703    90.843749
5    9849.559652    93.973760
6    1889.461643    94.751823
7    6103.216045    88.430043
8    1083.331967    89.947109
9    1565.227176    91.293506
10   6908.425001    85.906202
11   5973.476339    91.379183
12   4313.447519    93.891017
13    779.947617    89.106183
14   2585.576073    89.809821

[15 rows x 23 columns]
```

[13]:
```
# ============================================================================
# Paso 7.1: Calculos derivados de las variables de salida (Ej: Densidad de potencia)
# ============================================================================
# Añadimos las columnas que queramos obtener como resultado de cálculos con las varables de salida.
motors['p12::BSP_wPOT'] = motors['p7::BSP_Pm']/motors['p1::W']
motors['p13::BSP_kt'] = motors['p5::BSP_T']/motors['p9::BSP_Irms']

display(motors.head(15))

# Guardar el DataFrame de los motores generados en formato CSV
model_file = os.path.join(modelo_path, "generated_motors.csv")
motors.to_csv(model_file, index=False)
print("Base de datos de 10,000 motores guardada en:", modelo_path)
```

```
     x1::OSD   x2::Dint      x3::L    x4::tm    x5::hs2    x6::wt  x7::Nt  \
0   59.109233  31.737597  31.857104  3.038212   8.100607  2.862901      19
1   59.328662  25.186221  36.345354  3.184483   5.897683  2.917327      10
2   56.160596  27.553930  15.929237  2.550499   8.694059  4.861692      29
3   57.901684  31.469801  28.003841  2.484687   7.803684  4.070863       5
4   57.872131  28.719523  22.667203  3.251350   6.544124  2.336454      20
```

```
5   52.032954  25.642214  19.145558  3.351096   8.292027  4.708284      10
6   59.022678  30.102536  20.725107  3.232948   8.300313  4.880363      26
7   59.084247  23.506281  33.429104  2.176902  12.881647  3.738299      10
8   59.755479  29.172891  20.501322  2.071435   7.684031  2.163283      30
9   58.495452  23.956231  22.701931  3.328490  13.620656  2.710177      29
10  54.892270  27.079011  32.339341  3.050077   8.790003  2.443443       8
11  59.557619  29.147132  17.652638  3.129924   5.193074  2.318498      17
12  55.110892  26.975004  33.937222  2.244937   6.909910  4.422841      11
13  56.524556  32.208991  22.513839  3.210961   8.131219  3.047029      29
14  57.114990  33.602851  32.270730  2.398497   6.025044  3.636245      15

    x8::Nh   m1::Drot    m2::Dsh  …     p4::GFF  p5::BSP_T    p6::BSP_n  \
0        9  30.737597  15.593288  …  121.683401   2.195488  -899.790235
1        5  24.186221  10.621192  …   79.164804   0.786514  3929.156938
2        6  26.553930  13.580381  …  118.831479   0.990222     3.527924
3        4  30.469801  16.509056  …   18.483142   0.303060  9698.199115
4        8  27.719523  13.011245  …  129.987080   1.267237  1118.107567
5        9  24.642214  10.613676  …   94.679272   0.730050  6639.036286
6        7  29.102536  14.035915  …  126.125566   1.472640 -1020.992272
7        3  22.506281  11.436397  …   17.571305   0.391729  5171.823947
8        9  28.172891  15.694910  …  133.769191   1.050829 -1929.044073
9        9  22.956231   9.454615  …  100.507478   1.150322 -1405.642051
10       4  26.079011  12.241997  …   24.593303   0.446300  6355.251622
11       7  28.147132  13.559531  …  124.018366   0.843218  3236.366880
12       9  25.975004  13.777985  …  115.591728   1.467850  2610.183051
13       3  31.208991  15.584499  …   69.087639   0.942382   599.422861
14       7  32.602851  18.205043  …  111.126049   1.828527  1134.229564

    p7::BSP_Pm  p8::BSP_Mu  p9::BSP_Irms    p10::MSP_n  p11::UWP_Mu  \
0   455.875568   80.376615     22.625179    414.861528    88.645633
1   355.540030   89.221634     12.580649   4947.391460    92.228441
2   198.335769   77.244850     14.958926   3205.291516    95.466488
3   304.440999   89.972861     10.057482  10232.612990    87.166049
4   416.918003   80.546046     20.088639   3684.173703    90.843749
5   525.634760   90.026268     22.623560   9849.559652    93.973760
6   258.078796   78.270538     17.490562   1889.461643    94.751823
7   211.433877   87.473022      7.568995   6103.216045    88.430043
8   259.761539   75.457806     22.323881   1083.331967    89.947109
9   246.165757   73.158322     22.359127   1565.227176    91.293506
10  300.436897   88.765449     10.057775   6908.425001    85.906202
11  433.204842   85.452662     17.581354   5973.476339    91.379183
12  498.616377   87.017205     22.637501   4313.447519    93.891017
13  187.331615   77.982601      7.592398    779.947617    89.106183
14  426.674349   84.610137     17.626277   2585.576073    89.809821

    p12::BSP_wPOT  p13::BSP_kt
0      425.118369     0.097037
1      374.394561     0.062518
```

```
2      298.703113     0.066196
3      480.385304     0.030133
4      501.393378     0.063082
5      952.479490     0.032269
6      309.472096     0.084196
7      303.526520     0.051754
8      273.241087     0.047072
9      270.531117     0.051448
10     473.056175     0.044374
11     637.057022     0.047961
12     541.639811     0.064842
13     298.936675     0.124122
14     456.007821     0.103739

[15 rows x 25 columns]
```

Base de datos de 10,000 motores guardada en: C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Modelos_DBG\1000_MOT_Uniforme

[14]:
```python
# ============================================================================
# Paso 8: Filtrar motores válidos según constraints definidos
# ============================================================================
def is_valid_motor(row):
    constraints = [
        lambda r: 0.15 <= r['p1::W'] <= 1,      # p1::W entre 0.15 y 1
        lambda r: r['p4::GFF'] >= 1 and r['p4::GFF'] <= 60,
        lambda r: r['p5::BSP_T'] >= 0.5,
        lambda r: r['p6::BSP_n'] >= 3000,
        lambda r: 85 <= r['p8::BSP_Mu'] <= 99,     # p7::BSP_Mu entre 50 y 99
        lambda r: r['p10::MSP_n'] >= 4000,
        lambda r: 80 <= r['p11::UWP_Mu'] <= 99,      # p9::UWP_Mu entre 90 y 99
        # Puedes agregar más restricciones aquí, por ejemplo:
        # lambda r: r['p4::GFF'] >= 1 and r['p4::GFF'] <= 100,
    ]
    return all(condition(row) for condition in constraints)

motors['Valid'] = motors.apply(is_valid_motor, axis=1)
valid_motors = motors[motors['Valid']]
print(f"Número de motores válidos: {len(valid_motors)}")
```

```
Número de motores válidos: 571
```

[15]:
```python
##### Ordenar los motores válidos por 'p9::UWP_Mu' de menor a mayor
sorted_motors = valid_motors.sort_values(by='p8::BSP_Mu', ascending=False)
print("Motores válidos ordenados por 'p8::BSP_Mu' (de menor a mayor):")
display(sorted_motors.head(10))
```

```
Motores válidos ordenados por 'p9::UWP_Mu' (de menor a mayor):

        x1::OSD   x2::Dint      x3::L    x4::tm   x5::hs2    x6::wt  x7::Nt  \
```

```
9184  57.041319  29.484630  28.754743  3.257452  6.863795  4.267748       5
4276  57.446000  31.921637  31.476593  3.399687  6.906188  4.365567       5
850   59.007096  31.636367  38.951430  2.376020  6.576349  4.471652       5
3754  59.780590  25.655533  34.595331  2.404886  7.381689  4.139312       5
9077  52.967728  29.487177  31.351064  2.271805  7.256960  4.774430       5
1422  58.132522  32.494782  31.662778  2.604776  7.214700  4.112501       5
2635  52.068865  29.659137  33.385578  2.770355  6.781876  4.271929       5
1426  55.798289  35.123574  30.489771  2.199016  6.523112  4.563930       5
4897  55.906188  27.554639  25.533559  2.946039  7.196884  2.971461       5
9153  59.663450  33.171298  35.794893  2.947976  7.366636  4.723034       5

      x8::Nh   m1::Drot    m2::Dsh  …  p5::BSP_T     p6::BSP_n  p7::BSP_Pm  \
9184       8  28.484630  13.545546  …   0.612439   9557.507427  609.910881
4276       8  30.921637  15.001795  …   0.742987   7995.338998  612.737091
850        8  30.636367  16.845365  …   0.901542   6565.322157  608.131774
3754       7  24.655533  12.515609  …   0.528697   9451.994441  525.863221
9077       8  28.487177  15.518659  …   0.663647   8468.919235  604.421710
1422       9  31.494782  17.001007  …   0.857206   7890.659177  684.488263
2635       8  28.659137  14.644387  …   0.718730   8128.310456  607.365852
1426       9  34.123574  19.690235  …   0.897241   7248.191995  681.576571
4897       9  26.554639  12.789808  …   0.543645  12098.613665  678.966370
9153       7  32.171298  16.797832  …   0.791267   6598.592032  538.945523

      p8::BSP_Mu  p9::BSP_Irms    p10::MSP_n  p11::UWP_Mu  p12::BSP_wPOT  \
9184   93.350519     20.145856  10518.805629    91.114797     858.285979
4276   93.063910     20.136600   8838.919783    89.370271     800.205731
850    93.056981     20.125626   7383.503463    89.622715     631.615362
3754   93.012863     17.620511  10383.646297    91.753311     598.423956
9077   92.972180     20.129219   9589.389682    90.609999     902.082436
1422   92.851507     22.662280   8786.698929    88.365753     867.618938
2635   92.835788     20.145975   9036.395427    89.830751     889.191876
1426   92.696555     22.625614   8346.271229    87.016142     935.780800
4897   92.617758     22.682453  13281.967004    90.109876    1088.201685
9153   92.607329     17.618577   7513.251613    88.066723     610.757966

      p13::BSP_kt  Valid
9184     0.030400   True
4276     0.036897   True
850      0.044796   True
3754     0.030005   True
9077     0.032969   True
1422     0.037825   True
2635     0.035676   True
1426     0.039656   True
4897     0.023968   True
9153     0.044911   True

[10 rows x 26 columns]
```

[16]:
```
# ============================================================================
# Paso 9: Calcular y representar la frontera de Pareto
# ============================================================================
# Objetivos: minimizar p1::W, maximizar p8::BSP_Mu y p9::UWP_Mu
def compute_pareto_front(df, objectives):
    is_dominated = np.zeros(len(df), dtype=bool)
    for i in range(len(df)):
        for j in range(len(df)):
            if i == j:
                continue
            dominates = True
            for obj, sense in objectives.items():
                if sense == 'min':
                    if df.iloc[j][obj] > df.iloc[i][obj]:
                        dominates = False
                        break
                elif sense == 'max':
                    if df.iloc[j][obj] < df.iloc[i][obj]:
                        dominates = False
                        break
            if dominates:
                is_dominated[i] = True
                break
    frontier = df[~is_dominated]
    return frontier

objectives = {'p1::W': 'min', 'p8::BSP_Mu': 'max', 'p12::BSP_wPOT': 'max'}
valid_motors_reset = valid_motors.reset_index(drop=True)
pareto_motors = compute_pareto_front(valid_motors_reset, objectives)
print(f"Número de motores en la frontera de Pareto: {len(pareto_motors)}")

# Representación 2D: eje X = p9, eje Y = p1
plt.figure(figsize=(12, 6))

# Motores no válidos en negro
plt.scatter(
    motors.loc[~motors['Valid'], 'p8::BSP_Mu'],
    motors.loc[~motors['Valid'], 'p1::W'],
    c='black', label='No válidos', alpha=0.6, edgecolors='none'
)

# Motores válidos (no dominados) en azul
plt.scatter(
    valid_motors['p8::BSP_Mu'],
    valid_motors['p1::W'],
    c='blue', label='Válidos', alpha=0.6, edgecolors='none'
)
```

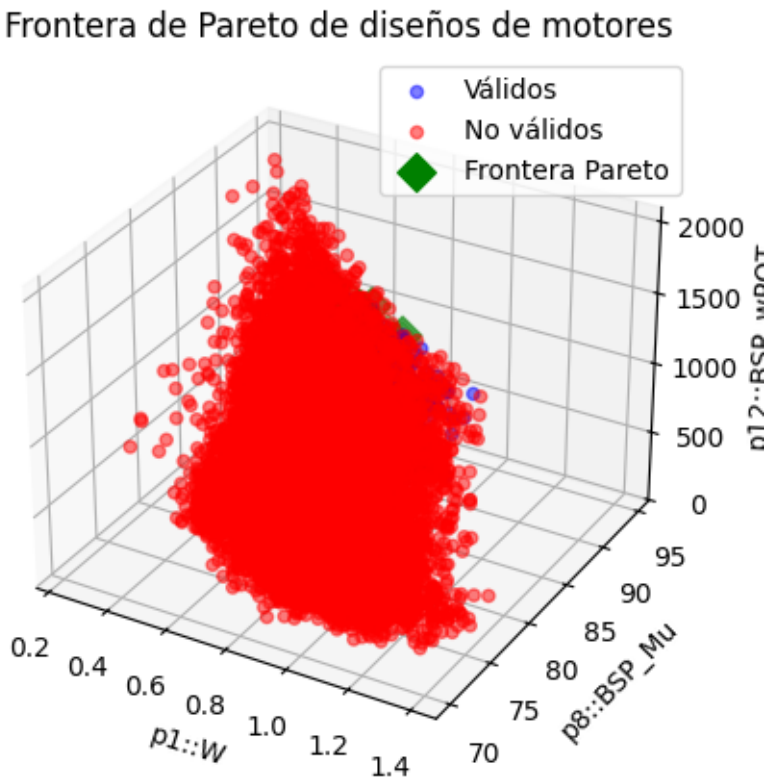
```
# Motores en la frontera de Pareto en rojo
plt.scatter(
    pareto_motors['p8::BSP_Mu'],
    pareto_motors['p1::W'],
    c='red', label='Frontera Pareto', s=60, marker='o', edgecolors='k'
)

plt.xlabel(r'p8::$\mu$')
plt.ylabel('p1::W')
plt.title('Frontera de Pareto en 2D (p8 vs p1)')
plt.legend()
plt.grid(True)
plt.tight_layout()

figure_file_2d = os.path.join(figure_path, "Pareto_frontier_2D.png")
plt.savefig(figure_file_2d, dpi=1000)
print("Figura guardada en:", figure_file_2d)
plt.close()

# Representación 3D de la frontera de Pareto
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(valid_motors['p1::W'], valid_motors['p8::BSP_Mu'], valid_motors['p12:
 ↪:BSP_wPOT'],
           c='blue', label='Válidos', alpha=0.5)
ax.scatter(motors[~motors['Valid']]['p1::W'], motors[~motors['Valid']]['p8::
 ↪BSP_Mu'], motors[~motors['Valid']]['p12::BSP_wPOT'],
           c='red', label='No válidos', alpha=0.5)
ax.scatter(pareto_motors['p1::W'], pareto_motors['p8::BSP_Mu'],␣
 ↪pareto_motors['p12::BSP_wPOT'],
           c='green', label='Frontera Pareto', s=100, marker='D')
ax.set_xlabel('p1::W')
ax.set_ylabel('p8::BSP_Mu')
ax.set_zlabel('p12::BSP_wPOT')
ax.legend()
plt.title('Frontera de Pareto de diseños de motores')
plt.show()
```

```
Número de motores en la frontera de Pareto: 13
Figura guardada en: C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Noteb
ooks_TFM\4.DBG\Figuras_DBG\1000_MOT_Uniforme\Pareto_frontier_2D.png
```



[17]:
```
# Si existen motores válidos, procedemos a la selección:
if len(valid_motors) > 0:
    # 1. Motor más liviano: mínimo de p1::W
    motor_liviano = valid_motors.loc[valid_motors['p1::W'].idxmin()]

    # 2. Motor más eficiente: máximo de p8::BSP_Mu (asumiendo que mayor p9::
 ↪UWP_Mu indica mayor eficiencia)
    motor_eficiente = valid_motors.loc[valid_motors['p8::BSP_Mu'].idxmax()]

    # 3. Motor más eficiente y liviano:
    # Se normalizan p1::W y p9::UWP_Mu en el subconjunto de motores válidos.
    vm = valid_motors.copy()
    # Normalizar p1::W (donde un menor valor es mejor, así que se invertirá)
    vm['p1::W_norm'] = (vm['p1::W'] - vm['p1::W'].min()) / (vm['p1::W'].max() -␣
 ↪vm['p1::W'].min())
    # Normalizar p8::BSP_Mu (mayor es mejor)
    vm['p8::BSP_Mu_norm'] = (vm['p8::BSP_Mu'] - vm['p8::BSP_Mu'].min()) /␣
 ↪(vm['p8::BSP_Mu'].max() - vm['p8::BSP_Mu'].min())
    # Normalizar p12::BSP_wPOT (mayor es mejor)
    vm['p12::BSP_wPOT_norm'] = (vm['p12::BSP_wPOT'] - vm['p12::BSP_wPOT'].
 ↪min()) / (vm['p12::BSP_wPOT'].max() - vm['p12::BSP_wPOT'].min())
```

```
    # Definir un score compuesto: se busca minimizar p1::W (por ello, usamos 1␣
 ↪- normalizado) y maximizar p8::BSP_Mu
    vm['composite_score'] = (1 - vm['p1::W_norm']) + vm['p8::BSP_Mu_norm']
    motor_eficiente_liviano = vm.loc[vm['composite_score'].idxmax()]

    # Mostrar las soluciones:
    print("\nMotor más liviano:")
    print(motor_liviano)

    print("\nMotor más eficiente:")
    print(motor_eficiente)

    print("\nMotor más eficiente y liviano (score compuesto):")
    print(motor_eficiente_liviano)

# Opcional: Guardar cada solución en un CSV separado
    #motor_liviano.to_frame().T.to_csv("motor_mas_liviano.csv", index=False)
    # motor_eficiente.to_frame().T.to_csv("motor_mas_eficiente.csv",␣
 ↪index=False)
    # motor_eficiente_liviano.to_frame().T.to_csv("motor_eficiente_y_liviano.
 ↪csv", index=False)
    # print("\nSoluciones guardadas en CSV.")
else:
    print("No se encontraron motores válidos. Verifique las constraints y el␣
 ↪escalado de los datos.")
```

```
Motor más liviano:
x1::OSD             57.89255
x2::Dint           29.469571
x3::L                 13.348
x4::tm              3.122518
x5::hs2             9.766914
x6::wt              2.942228
x7::Nt                    15
x8::Nh                     6
m1::Drot           28.469571
m2::Dsh            13.804658
m3::he              4.444576
m4::Rmag           13.454156
m5::Rs             24.501699
m6::GFF            51.362921
p1::W               0.469023
p4::GFF            58.436128
p5::BSP_T           0.578618
p6::BSP_n        6631.444627
```

```
p7::BSP_Pm        390.805603
p8::BSP_Mu         87.629198
p9::BSP_Irms        15.10486
p10::MSP_n       8105.883682
p11::UWP_Mu        90.511919
p12::BSP_wPOT     833.233806
p13::BSP_kt         0.038307
Valid                   True
Name: 7474, dtype: object

Motor más eficiente:
x1::OSD             57.041319
x2::Dint             29.48463
x3::L               28.754743
x4::tm               3.257452
x5::hs2              6.863795
x6::wt               4.267748
x7::Nt                      5
x8::Nh                      8
m1::Drot             28.48463
m2::Dsh             13.545546
m3::he               6.91455
m4::Rmag            13.427952
m5::Rs               21.60611
m6::GFF             45.367167
p1::W                0.710615
p4::GFF             53.606973
p5::BSP_T            0.612439
p6::BSP_n         9557.507427
p7::BSP_Pm         609.910881
p8::BSP_Mu          93.350519
p9::BSP_Irms        20.145856
p10::MSP_n       10518.805629
p11::UWP_Mu         91.114797
p12::BSP_wPOT      858.285979
p13::BSP_kt            0.0304
Valid                    True
Name: 9184, dtype: object

Motor más eficiente y liviano (score compuesto):
x1::OSD                  57.165948
x2::Dint                  34.97629
x3::L                    19.604869
x4::tm                    2.938846
x5::hs2                   6.859336
x6::wt                    4.624641
x7::Nt                           6
x8::Nh                           8
```

```
m1::Drot                   33.97629
m2::Dsh                   18.105372
m3::he                     4.235494
m4::Rmag                  16.253433
m5::Rs                    24.347481
m6::GFF                   45.181414
p1::W                      0.549823
p4::GFF                   55.091781
p5::BSP_T                  0.615264
p6::BSP_n               9356.209039
p7::BSP_Pm                603.97592
p8::BSP_Mu                92.536375
p9::BSP_Irms              20.136878
p10::MSP_n             10639.211197
p11::UWP_Mu               89.725893
p12::BSP_wPOT           1098.491564
p13::BSP_kt                0.030554
Valid                          True
p1::W_norm                 0.160186
p8::BSP_Mu_norm            0.902394
p12::BSP_wPOT_norm         0.790996
composite_score            1.742208
Name: 8649, dtype: object
```

[18]:
```
# ============================================================================
# Paso 10: Seleccionar el motor válido óptimo
# ============================================================================
# Se normalizan los objetivos y se define un score compuesto
valid_motors_comp = valid_motors.copy()
for col, sense in [('p1::W', 'min'), ('p8::BSP_Mu', 'max'), ('p12::BSP_wPOT',␣
 ↪'max')]:
    col_min = valid_motors_comp[col].min()
    col_max = valid_motors_comp[col].max()
    if sense == 'min':
        valid_motors_comp[col + '_norm'] = 1 - (valid_motors_comp[col] -␣
 ↪col_min) / (col_max - col_min)
    else:
        valid_motors_comp[col + '_norm'] = (valid_motors_comp[col] - col_min) /␣
 ↪(col_max - col_min)

valid_motors_comp['composite_score'] = (valid_motors_comp['p1::W_norm'] +
                                        valid_motors_comp['p8::BSP_Mu_norm'] +
                                        valid_motors_comp['p12::
 ↪BSP_wPOT_norm'])
optimal_motor = valid_motors_comp.loc[valid_motors_comp['composite_score'].
 ↪idxmax()]
print("Motor válido óptimo (según score compuesto):")
```

```
print(optimal_motor)

model_file = os.path.join(model_path, "optimal_motor.csv")
optimal_motor.to_frame().T.to_csv(model_file, index=False)
print("El motor óptimo se ha guardado en:", model_path)
```

```
Motor válido óptimo (según score compuesto):
x1::OSD                  54.956163
x2::Dint                 29.744712
x3::L                    18.937528
x4::tm                    2.533385
x5::hs2                   8.747622
x6::wt                    2.504537
x7::Nt                           7
x8::Nh                           9
m1::Drot                 28.744712
m2::Dsh                  15.179452
m3::he                    3.858104
m4::Rmag                  13.73901
m5::Rs                   23.619978
m6::GFF                  38.856007
p1::W                     0.495859
p4::GFF                  45.529899
p5::BSP_T                 0.540438
p6::BSP_n             11624.167634
p7::BSP_Pm              658.124721
p8::BSP_Mu               90.590176
p9::BSP_Irms             22.682105
p10::MSP_n            12790.473124
p11::UWP_Mu              87.780375
p12::BSP_wPOT          1327.242788
p13::BSP_kt               0.023827
Valid                         True
p1::W_norm                0.946798
p8::BSP_Mu_norm           0.669069
p12::BSP_wPOT_norm             1.0
composite_score           2.615867
Name: 447, dtype: object
El motor óptimo se ha guardado en: C:\Users\s00244\Documents\GitHub\MotorDesignD
ataDriven\Notebooks_TFM\4.DBG\Modelos_DBG
```

[ ]: