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_600_Uniforme.csv")
print(data_file)

# Ruta al archivo de las figuras
figure_path = os.path.join(fig_path, "600_MOT_Uniforme")
print(figure_path)

# Ruta al archivo de los modelos
modelo_path = os.path.join(model_path, "600_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_600_Uniforme.csv
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Figur
as_DBG\600_MOT_Uniforme
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Model
os_DBG\600_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           2.486359e-02
p2::Tnom        7.720431e-34
p3::nnom        0.000000e+00
p4::GFF         1.212433e+02
p5::BSP_T       5.009069e-02
p6::BSP_n       2.475225e+07
p7::BSP_Pm      1.659710e+04
p8::BSP_Mu      6.835560e+00
p9::BSP_Irms    2.181205e+01
p10::MSP_n      2.931535e+07
p11::UWP_Mu     9.860026e+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']
```

[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\600_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	56.878938	25.937002	13.961210	3.361205	11.962285	4.533674	25	
1	58.454009	21.423947	38.705283	2.266803	10.635381	2.074551	15	
2	56.373816	32.891454	10.584881	2.339747	7.331313	4.501579	14	
3	54.103051	22.013876	22.342279	2.675007	6.615436	3.942007	5	
4	51.490359	26.690447	34.891264	2.776343	7.550254	3.352386	5	

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

[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\600_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([5.71125229e-01, 8.79473809e+01, 8.32889831e-01, 2.13809505e+03,
       2.34814684e+02, 7.49467030e+01, 1.50991042e+01, 5.39096256e+03,
       9.29880215e+01])
```

	x1::OSD	x2::Dint	x3::L	x4::tm	x5::hs2	x6::wt	x7::Nt	\
0	56.878938	25.937002	13.961210	3.361205	11.962285	4.533674	25	
1	58.454009	21.423947	38.705283	2.266803	10.635381	2.074551	15	
2	56.373816	32.891454	10.584881	2.339747	7.331313	4.501579	14	
3	54.103051	22.013876	22.342279	2.675007	6.615436	3.942007	5	
4	51.490359	26.690447	34.891264	2.776343	7.550254	3.352386	5	
5	52.862254	29.133671	29.763748	3.239311	7.093637	2.164290	11	
6	56.029122	31.724857	32.839381	3.477797	7.334359	2.986544	20	
7	50.852398	25.292360	29.933904	2.649398	7.531686	3.442395	5	
8	55.850300	23.737541	26.075296	3.220673	9.228778	3.501741	10	
9	54.365467	23.726527	23.447882	3.309582	5.092144	3.208230	16	
10	59.894188	25.210708	26.303353	2.161891	6.482994	4.670316	29	
11	56.344079	27.270916	15.632514	2.306139	5.239036	3.431562	8	
12	59.475657	26.971752	32.627522	2.130080	8.810644	2.943810	8	
13	56.826489	22.308518	19.527462	2.793726	8.730450	4.147115	29	
14	58.526727	27.841164	30.514026	3.065691	7.765252	3.283206	21	

	x8::Nh	m1::Drot	m2::Dsh	…	m6::GFF	p1::W	p4::GFF	\
0	6	24.937002	10.804021	…	95.078576	0.571125	87.947381	
1	5	20.423947	9.769213	…	45.598472	0.918258	51.016506	
2	8	31.891454	17.814401	…	103.531285	0.468527	109.725892	
3	3	21.013876	9.374183	…	26.072461	0.494489	29.135620	
4	9	25.690447	12.511919	…	43.391863	0.685863	50.855219	
5	8	28.133671	13.331143	…	69.240955	0.721728	84.088153	
6	3	30.724857	14.705018	…	46.170296	0.765447	59.241048	
7	5	24.292360	11.767175	…	26.328322	0.548512	29.998370	
8	3	22.737541	9.514041	…	25.895229	0.573754	28.088617	
9	6	22.726527	9.328355	…	177.620619	0.689849	126.203670	
10	6	24.210708	12.683866	…	302.327245	0.967357	123.616269	
11	7	26.270916	13.866947	…	85.975343	0.488516	96.746961	
12	4	25.971752	14.005376	…	23.099210	0.758720	25.915206	
13	7	21.308518	9.347204	…	238.759231	0.802051	131.573742	
14	4	26.841164	12.755163	…	73.190140	0.840594	80.992559	

	p5::BSP_T	p6::BSP_n	p7::BSP_Pm	p8::BSP_Mu	p9::BSP_Irms	\
0	0.832890	2138.095050	234.814684	74.946703	15.099104	
1	0.885106	3415.071208	294.230652	81.823136	12.588950	
2	0.679041	6891.664345	477.956119	87.822102	20.134776	
3	0.126783	18875.543141	233.679373	91.055474	7.550785	
4	0.723746	8813.116972	666.991170	92.678976	22.655243	
5	1.081437	5384.715849	572.167845	88.394197	20.140841	
6	0.999424	1762.917782	202.985392	80.659168	7.552715	
7	0.315148	11419.152178	381.391615	91.396695	12.586725	
8	0.314910	6768.833058	217.648157	89.148581	7.552212	

```
9    0.806369   2913.842811  337.582449   83.912980     15.101929
10   1.197414  -1229.837087  200.704731   72.318266     15.096310
11   0.427093  11154.943247  507.754517   92.533944     17.620176
12   0.468483   5914.294867  299.661156   89.963246     10.071284
13   1.053067     41.481184  188.945142   72.121168     17.614894
14   1.115498   2004.177610  235.947938   78.850995     10.068246

      p10::MSP_n  p11::UWP_Mu
0    5390.962555    92.988021
1    3770.742950    89.854650
2    9383.670037    91.902708
3   19806.403331    83.711824
4    9910.830172    89.399904
5    5672.768549    85.812237
6    1932.158692    86.627378
7   12246.929896    89.457747
8    7645.504450    89.948917
9    4764.481154    93.526863
10   2701.186509    95.770214
11  13327.326360    92.953446
12   6634.162569    88.467659
13   5306.064338    96.613952
14   2429.310420    91.867621

[15 rows x 23 columns]

Base de datos de 10,000 motores guardada en: C:\Users\s00244\Documents\GitHub\Mo
torDesignDataDriven\Notebooks_TFM\4.DBG\Modelos_DBG
```

[19]:
```
# ============================================================================
# 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  56.878938  25.937002  13.961210  3.361205  11.962285  4.533674      25
1  58.454009  21.423947  38.705283  2.266803  10.635381  2.074551      15
```

2	56.373816	32.891454	10.584881	2.339747	7.331313	4.501579	14
3	54.103051	22.013876	22.342279	2.675007	6.615436	3.942007	5
4	51.490359	26.690447	34.891264	2.776343	7.550254	3.352386	5
5	52.862254	29.133671	29.763748	3.239311	7.093637	2.164290	11
6	56.029122	31.724857	32.839381	3.477797	7.334359	2.986544	20
7	50.852398	25.292360	29.933904	2.649398	7.531686	3.442395	5
8	55.850300	23.737541	26.075296	3.220673	9.228778	3.501741	10
9	54.365467	23.726527	23.447882	3.309582	5.092144	3.208230	16
10	59.894188	25.210708	26.303353	2.161891	6.482994	4.670316	29
11	56.344079	27.270916	15.632514	2.306139	5.239036	3.431562	8
12	59.475657	26.971752	32.627522	2.130080	8.810644	2.943810	8
13	56.826489	22.308518	19.527462	2.793726	8.730450	4.147115	29
14	58.526727	27.841164	30.514026	3.065691	7.765252	3.283206	21

	x8::Nh	m1::Drot	m2::Dsh	…	p5::BSP_T	p6::BSP_n	p7::BSP_Pm	\
0	6	24.937002	10.804021	…	0.832890	2138.095050	234.814684	
1	5	20.423947	9.769213	…	0.885106	3415.071208	294.230652	
2	8	31.891454	17.814401	…	0.679041	6891.664345	477.956119	
3	3	21.013876	9.374183	…	0.126783	18875.543141	233.679373	
4	9	25.690447	12.511919	…	0.723746	8813.116972	666.991170	
5	8	28.133671	13.331143	…	1.081437	5384.715849	572.167845	
6	3	30.724857	14.705018	…	0.999424	1762.917782	202.985392	
7	5	24.292360	11.767175	…	0.315148	11419.152178	381.391615	
8	3	22.737541	9.514041	…	0.314910	6768.833058	217.648157	
9	6	22.726527	9.328355	…	0.806369	2913.842811	337.582449	
10	6	24.210708	12.683866	…	1.197414	-1229.837087	200.704731	
11	7	26.270916	13.866947	…	0.427093	11154.943247	507.754517	
12	4	25.971752	14.005376	…	0.468483	5914.294867	299.661156	
13	7	21.308518	9.347204	…	1.053067	41.481184	188.945142	
14	4	26.841164	12.755163	…	1.115498	2004.177610	235.947938	

	p8::BSP_Mu	p9::BSP_Irms	p10::MSP_n	p11::UWP_Mu	p12::BSP_wPOT	\
0	74.946703	15.099104	5390.962555	92.988021	411.143952	
1	81.823136	12.588950	3770.742950	89.854650	320.422560	
2	87.822102	20.134776	9383.670037	91.902708	1020.124505	
3	91.055474	7.550785	19806.403331	83.711824	472.567720	
4	92.678976	22.655243	9910.830172	89.399904	972.483848	
5	88.394197	20.140841	5672.768549	85.812237	792.774444	
6	80.659168	7.552715	1932.158692	86.627378	265.185384	
7	91.396695	12.586725	12246.929896	89.457747	695.320133	
8	89.148581	7.552212	7645.504450	89.948917	379.340563	
9	83.912980	15.101929	4764.481154	93.526863	489.357215	
10	72.318266	15.096310	2701.186509	95.770214	207.477478	
11	92.533944	17.620176	13327.326360	92.953446	1039.382142	
12	89.963246	10.071284	6634.162569	88.467659	394.956413	
13	72.121168	17.614894	5306.064338	96.613952	235.577431	
14	78.850995	10.068246	2429.310420	91.867621	280.692001	

```
    p13::BSP_kt  Valid
0      0.055162  False
1      0.070308  False
2      0.033725  False
3      0.016791  False
4      0.031946   True
5      0.053694  False
6      0.132326  False
7      0.025038  False
8      0.041698  False
9      0.053395  False
10     0.079318  False
11     0.024239  False
12     0.046517  False
13     0.059783  False
14     0.110794  False

[15 rows x 26 columns]
```

```
Base de datos de 10,000 motores guardada en: C:\Users\s00244\Documents\GitHub\Mo
torDesignDataDriven\Notebooks_TFM\4.DBG\Modelos_DBG\600_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: 551
```

[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 'p9::UWP_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	\
8660	51.149526	28.090465	29.343576	3.013235	6.860024	4.134233	5	
6505	48.148948	24.275162	30.384555	2.103520	7.951496	3.970023	5	
1795	59.281772	27.307231	23.221081	2.437038	7.674955	4.045654	6	
4	51.490359	26.690447	34.891264	2.776343	7.550254	3.352386	5	
3216	56.185359	29.527419	26.451127	2.772208	6.046473	3.881202	6	
8447	59.581185	24.333391	33.684722	2.733235	9.423389	4.100979	5	
2947	59.339395	32.259439	28.953560	3.480808	8.217510	4.296542	5	
6557	56.933517	23.286026	32.917931	2.487137	8.943066	3.577294	5	
2324	56.218475	28.030163	33.518655	3.377907	7.139038	4.693076	6	
1596	56.671206	23.077045	37.081514	2.439254	9.186895	4.309262	5	

	x8::Nh	m1::Drot	m2::Dsh	…	p5::BSP_T	p6::BSP_n	p7::BSP_Pm	\
8660	8	27.090465	13.038147	…	0.583563	9936.826958	605.241488	
6505	9	23.275162	12.132362	…	0.553650	10886.616728	634.910391	
1795	8	26.307231	13.631089	…	0.530363	10225.693406	580.681581	
4	9	25.690447	12.511919	…	0.723746	8813.116972	666.991170	
3216	6	28.527419	14.546597	…	0.513026	8752.117932	461.134723	
8447	9	23.333391	10.914523	…	0.620283	9606.793306	633.948693	
2947	8	31.259439	15.080840	…	0.699102	8650.082128	611.603083	
6557	9	22.286026	10.658602	…	0.575441	10433.387739	629.811484	
2324	7	27.030163	12.265731	…	0.706948	6953.481691	525.641799	
1596	8	22.077045	10.605096	…	0.567790	9343.335636	568.397102	

	p8::BSP_Mu	p9::BSP_Irms	p10::MSP_n	p11::UWP_Mu	p12::BSP_wPOT	\
8660	93.013118	20.136280	10971.572578	90.977685	1006.759296	
6505	92.864665	22.653061	12917.473207	90.992896	1126.572313	
1795	92.754363	20.136599	12011.010748	92.016838	889.453278	
4	92.678976	22.655243	9910.830172	89.399904	972.483848	
3216	92.648044	15.102927	9555.779982	91.618661	714.378006	
8447	92.641357	22.653618	11508.591752	91.407396	746.218191	
2947	92.635885	20.137254	9401.334707	88.191362	834.817552	
6557	92.578788	22.654578	12479.215848	91.275715	815.209558	
2324	92.501984	17.617916	7811.868954	92.018229	664.981131	
1596	92.493561	20.135608	11215.440880	91.721106	675.135676	

	p13::BSP_kt	Valid
8660	0.028981	True
6505	0.024440	True
1795	0.026338	True
4	0.031946	True
3216	0.033969	True
8447	0.027381	True
2947	0.034717	True

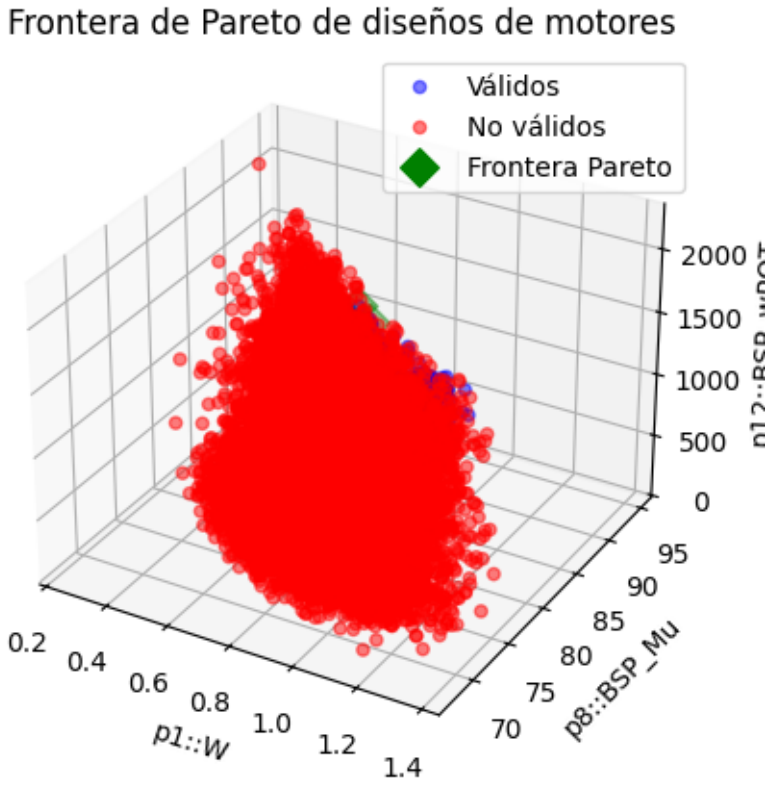
```
6557     0.025401   True
2324     0.040127   True
1596     0.028198   True

[10 rows x 26 columns]
```

[16]:
```python
# ============================================================================
# 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: 15
Figura guardada en: C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Noteb
```

ooks_TFM\4.DBG\Figuras_DBG\600_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            55.977614
x2::Dint           28.594854
x3::L              14.630009
x4::tm              3.322105
x5::hs2             9.107869
x6::wt              3.925814
x7::Nt                    17
x8::Nh                     4
m1::Drot           27.594854
m2::Dsh            12.780685
m3::he              4.58351
m4::Rmag           12.966901
m5::Rs             23.405297
m6::GFF            51.312103
p1::W               0.448055
p4::GFF            56.380343
```

```
p5::BSP_T           0.509027
p6::BSP_n         4885.25023
p7::BSP_Pm          255.628
p8::BSP_Mu         86.358103
p9::BSP_Irms       10.067735
p10::MSP_n       6445.331387
p11::UWP_Mu        91.670918
p12::BSP_wPOT      570.52758
p13::BSP_kt          0.05056
Valid                   True
Name: 1947, dtype: object

Motor más eficiente:
x1::OSD             51.149526
x2::Dint            28.090465
x3::L               29.343576
x4::tm               3.013235
x5::hs2              6.860024
x6::wt               4.134233
x7::Nt                      5
x8::Nh                      8
m1::Drot            27.090465
m2::Dsh             13.038147
m3::he               4.669506
m4::Rmag            12.791924
m5::Rs              20.905257
m6::GFF             47.495858
p1::W                0.601178
p4::GFF             56.314016
p5::BSP_T            0.583563
p6::BSP_n         9936.826958
p7::BSP_Pm         605.241488
p8::BSP_Mu          93.013118
p9::BSP_Irms         20.13628
p10::MSP_n       10971.572578
p11::UWP_Mu         90.977685
p12::BSP_wPOT     1006.759296
p13::BSP_kt          0.028981
Valid                   True
Name: 8660, dtype: object

Motor más eficiente y liviano (score compuesto):
x1::OSD                48.148948
x2::Dint               24.275162
x3::L                  30.384555
x4::tm                   2.10352
x5::hs2                7.951496
x6::wt                 3.970023
```

```
x7::Nt                          5
x8::Nh                          9
m1::Drot                23.275162
m2::Dsh                 12.132362
m3::he                   3.985397
m4::Rmag                11.111701
m5::Rs                  20.089077
m6::GFF                 51.762827
p1::W                    0.563577
p4::GFF                 56.611095
p5::BSP_T                0.55365
p6::BSP_n           10886.616728
p7::BSP_Pm            634.910391
p8::BSP_Mu             92.864665
p9::BSP_Irms           22.653061
p10::MSP_n          12917.473207
p11::UWP_Mu            90.992896
p12::BSP_wPOT        1126.572313
p13::BSP_kt              0.02444
Valid                       True
p1::W_norm              0.214016
p8::BSP_Mu_norm          0.98146
p12::BSP_wPOT_norm      0.901872
composite_score         1.767444
Name: 6505, dtype: object
```

[18]:
```
# ============================================================================
# Paso 7: 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                  48.148948
x2::Dint                 24.275162
x3::L                    30.384555
x4::tm                     2.10352
x5::hs2                   7.951496
x6::wt                    3.970023
x7::Nt                           5
x8::Nh                           9
m1::Drot                 23.275162
m2::Dsh                  12.132362
m3::he                    3.985397
m4::Rmag                 11.111701
m5::Rs                   20.089077
m6::GFF                  51.762827
p1::W                     0.563577
p4::GFF                  56.611095
p5::BSP_T                  0.55365
p6::BSP_n              10886.616728
p7::BSP_Pm               634.910391
p8::BSP_Mu                92.864665
p9::BSP_Irms              22.653061
p10::MSP_n             12917.473207
p11::UWP_Mu               90.992896
p12::BSP_wPOT           1126.572313
p13::BSP_kt                0.02444
Valid                         True
p1::W_norm                0.785984
p8::BSP_Mu_norm            0.98146
p12::BSP_wPOT_norm        0.901872
composite_score           2.669316
Name: 6505, dtype: object
El motor óptimo se ha guardado en: C:\Users\s00244\Documents\GitHub\MotorDesignD
ataDriven\Notebooks_TFM\4.DBG\Modelos_DBG
```

```
[ ]:
```