DBG_V5

May 22, 2025

[1]:
```python
# 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
```

```python
    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())
```

```python
    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]:

```python
# ================================================================================
# 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_5000_Uniforme.csv")
print(data_file)

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

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

# Lectura del archivo CSV
try:
    df = pd.read_csv(data_file)
    print("Archivo cargado exitosamente.")
except FileNotFoundError:
    print("Error: Archivo no encontrado. Revisa la ruta del archivo.")
except pd.errors.ParserError:
```

```
    print("Error: Problema al analizar el archivo CSV. Revisa el formato del␣
 ↪archivo.")
except Exception as e:
    print(f"Ocurrió un error inesperado: {e}")

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

```
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\DB_DB
G\design_DB_preprocessed_5000_Uniforme.csv
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Figur
as_DBG\5000_MOT_Uniforme
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\4.DBG\Model
os_DBG\5000_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           2.409097e-02
p2::Tnom        1.733798e-33
p3::nnom        0.000000e+00
p4::GFF         1.216293e+02
p5::BSP_T       5.526305e-02
p6::BSP_n       2.691714e+07
p7::BSP_Pm      1.675008e+04
p8::BSP_Mu      7.538927e+00
p9::BSP_Irms    2.199128e+01
p10::MSP_n      3.204081e+07
p11::UWP_Mu     8.609559e+00
dtype: float64

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

[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}")
```

```python
    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\5000_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  57.739013  25.618727  31.781983  2.443376  10.051552  4.021307      22
1  55.701469  37.016266  33.828646  2.348957   5.623211  2.081814      22
2  48.458063  21.942572  14.966768  2.864008   9.605803  3.540775      14
3  50.679253  21.359862  14.373353  2.082497   8.610180  3.164543      16
4  59.112750  25.544342  28.039716  2.566281  10.398713  4.541243      15

   x8::Nh
0       5
1       4
2       4
3       6
4       5
```

[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\5000_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([8.43545788e-01, 7.88012478e+01, 1.29385826e+00, 4.02428637e+03,
       2.39351825e+02, 7.57942494e+01, 1.25718784e+01, 5.13579431e+03,
       9.15424914e+01])
```

	x1::OSD	x2::Dint	x3::L	x4::tm	x5::hs2	x6::wt	x7::Nt	\
0	57.739013	25.618727	31.781983	2.443376	10.051552	4.021307	22	
1	55.701469	37.016266	33.828646	2.348957	5.623211	2.081814	22	
2	48.458063	21.942572	14.966768	2.864008	9.605803	3.540775	14	
3	50.679253	21.359862	14.373353	2.082497	8.610180	3.164543	16	
4	59.112750	25.544342	28.039716	2.566281	10.398713	4.541243	15	
5	58.475943	27.493545	12.576956	2.520047	11.186401	2.869536	25	
6	53.766832	21.735243	31.213946	3.241647	10.699640	2.374365	18	
7	46.576302	21.536620	20.353282	2.833550	6.754147	3.372813	26	
8	50.428177	25.736853	20.788893	2.788333	7.083470	3.921121	5	
9	57.811825	34.456591	17.238978	2.752249	5.728449	2.745969	7	
10	51.186176	23.894798	17.183530	2.369422	9.096225	2.599487	20	
11	57.500903	34.090783	37.661305	2.180233	6.026680	4.689462	12	
12	59.368120	21.442862	23.105132	2.998199	12.503771	2.041355	18	
13	51.013443	25.540723	24.699027	2.455742	6.790719	3.277956	10	
14	56.481669	26.011266	34.169712	2.418495	6.663797	4.619776	11	

	x8::Nh	m1::Drot	m2::Dsh	…	m6::GFF	p1::W	p4::GFF	\
0	5	24.618727	12.412338	…	84.089330	0.843546	78.801248	
1	4	36.016266	20.742276	…	70.407106	0.735548	70.059653	
2	4	20.942572	8.945250	…	50.478243	0.353519	54.232766	
3	6	20.359862	10.092051	…	97.302792	0.446476	92.226806	
4	5	24.544342	12.113396	…	60.525094	0.777579	63.702734	
5	4	26.493545	13.598153	…	50.329395	0.459175	55.499953	
6	5	20.735243	8.041881	…	56.181107	0.708497	59.801299	
7	6	20.536620	8.716200	…	233.943794	0.543727	93.607320	
8	4	24.736853	11.806800	…	24.695278	0.411649	27.707768	
9	8	33.456591	18.107352	…	51.373778	0.531055	69.570716	
10	9	22.894798	11.328869	…	133.909481	0.588717	99.781736	
11	5	33.090783	18.990092	…	69.970224	0.905060	75.193828	
12	7	20.442862	8.319933	…	60.139833	0.707649	62.586766	
13	7	24.540723	12.331890	…	81.203898	0.584561	90.714838	
14	3	25.011266	12.742486	…	51.423180	0.793178	58.067516	

	p5::BSP_T	p6::BSP_n	p7::BSP_Pm	p8::BSP_Mu	p9::BSP_Irms	\
0	1.293858	4024.286368	239.351825	75.794249	12.571878	
1	0.903996	6101.978426	265.829679	80.161631	10.022244	
2	0.299880	7634.670165	245.516346	87.487547	10.070279	
3	0.466765	12915.206269	300.919447	85.339741	15.031529	
4	0.932663	3001.326114	286.462955	83.530609	12.596004	
5	0.530852	4114.537314	227.906361	80.884929	10.045504	
6	0.879672	2991.585253	272.502250	80.009854	12.567975	
7	0.607276	22177.981906	261.534676	75.940039	14.983005	
8	0.179600	16574.423995	306.799497	90.456781	10.073510	

```
9     0.519127  11261.947498  617.229398   91.292699     20.029477
10    0.738627  12060.728537  377.747176   80.096391     22.531423
11    1.232478   1079.488441  332.063327   85.600423     12.582260
12    0.721756   3831.221385  308.390087   80.049994     17.566057
13    0.749373   8039.110383  447.182482   89.575871     17.546973
14    0.517558   4314.623945  218.257403   88.181153      7.558288

      p10::MSP_n  p11::UWP_Mu
0    5135.794315    91.542491
1    5854.738680    78.355673
2   10470.547406    89.191287
3   16189.051025    92.264251
4    4399.805860    92.267205
5    5345.701053    90.798200
6    3870.705916    91.086400
7   22958.907362    94.421584
8   17338.902076    86.828616
9   11688.173469    89.499920
10  14513.129125    93.436966
11   1504.634046    89.814511
12   6060.280397    91.401434
13   9472.631634    93.031207
14   4848.859266    92.784853

[15 rows x 23 columns]
```

[13]:

```python
# ==============================================================================
# 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  57.739013  25.618727  31.781983  2.443376  10.051552  4.021307      22
1  55.701469  37.016266  33.828646  2.348957   5.623211  2.081814      22
2  48.458063  21.942572  14.966768  2.864008   9.605803  3.540775      14
3  50.679253  21.359862  14.373353  2.082497   8.610180  3.164543      16
4  59.112750  25.544342  28.039716  2.566281  10.398713  4.541243      15
```

5	58.475943	27.493545	12.576956	2.520047	11.186401	2.869536	25
6	53.766832	21.735243	31.213946	3.241647	10.699640	2.374365	18
7	46.576302	21.536620	20.353282	2.833550	6.754147	3.372813	26
8	50.428177	25.736853	20.788893	2.788333	7.083470	3.921121	5
9	57.811825	34.456591	17.238978	2.752249	5.728449	2.745969	7
10	51.186176	23.894798	17.183530	2.369422	9.096225	2.599487	20
11	57.500903	34.090783	37.661305	2.180233	6.026680	4.689462	12
12	59.368120	21.442862	23.105132	2.998199	12.503771	2.041355	18
13	51.013443	25.540723	24.699027	2.455742	6.790719	3.277956	10
14	56.481669	26.011266	34.169712	2.418495	6.663797	4.619776	11

	x8::Nh	m1::Drot	m2::Dsh	…	p4::GFF	p5::BSP_T	p6::BSP_n	\
0	5	24.618727	12.412338	…	78.801248	1.293858	4024.286368	
1	4	36.016266	20.742276	…	70.059653	0.903996	6101.978426	
2	4	20.942572	8.945250	…	54.232766	0.299880	7634.670165	
3	6	20.359862	10.092051	…	92.226806	0.466765	12915.206269	
4	5	24.544342	12.113396	…	63.702734	0.932663	3001.326114	
5	4	26.493545	13.598153	…	55.499953	0.530852	4114.537314	
6	5	20.735243	8.041881	…	59.801299	0.879672	2991.585253	
7	6	20.536620	8.716200	…	93.607320	0.607276	22177.981906	
8	4	24.736853	11.806800	…	27.707768	0.179600	16574.423995	
9	8	33.456591	18.107352	…	69.570716	0.519127	11261.947498	
10	9	22.894798	11.328869	…	99.781736	0.738627	12060.728537	
11	5	33.090783	18.990092	…	75.193828	1.232478	1079.488441	
12	7	20.442862	8.319933	…	62.586766	0.721756	3831.221385	
13	7	24.540723	12.331890	…	90.714838	0.749373	8039.110383	
14	3	25.011266	12.742486	…	58.067516	0.517558	4314.623945	

	p7::BSP_Pm	p8::BSP_Mu	p9::BSP_Irms	p10::MSP_n	p11::UWP_Mu	\
0	239.351825	75.794249	12.571878	5135.794315	91.542491	
1	265.829679	80.161631	10.022244	5854.738680	78.355673	
2	245.516346	87.487547	10.070279	10470.547406	89.191287	
3	300.919447	85.339741	15.031529	16189.051025	92.264251	
4	286.462955	83.530609	12.596004	4399.805860	92.267205	
5	227.906361	80.884929	10.045504	5345.701053	90.798200	
6	272.502250	80.009854	12.567975	3870.705916	91.086400	
7	261.534676	75.940039	14.983005	22958.907362	94.421584	
8	306.799497	90.456781	10.073510	17338.902076	86.828616	
9	617.229398	91.292699	20.029477	11688.173469	89.499920	
10	377.747176	80.096391	22.531423	14513.129125	93.436966	
11	332.063327	85.600423	12.582260	1504.634046	89.814511	
12	308.390087	80.049994	17.566057	6060.280397	91.401434	
13	447.182482	89.575871	17.546973	9472.631634	93.031207	
14	218.257403	88.181153	7.558288	4848.859266	92.784853	

	p12::BSP_wPOT	p13::BSP_kt
0	283.744911	0.102917
1	361.403589	0.090199

```
2      694.492281     0.029779
3      673.987796     0.031052
4      368.403615     0.074044
5      496.339216     0.052845
6      384.620315     0.069993
7      481.004059     0.040531
8      745.294628     0.017829
9     1162.270103     0.025918
10     641.644882     0.032782
11     366.896419     0.097954
12     435.795475     0.041088
13     764.988636     0.042707
14     275.168384     0.068476

[15 rows x 25 columns]
```

```
Base de datos de 10,000 motores guardada en: C:\Users\s00244\Documents\GitHub\Mo
torDesignDataDriven\Notebooks_TFM\4.DBG\Modelos_DBG\5000_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: 530
```

[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 'p8::BSP_Mu' (de menor a mayor):

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

```
1615  59.387769  32.169998  21.937080  3.497787  8.672517  4.697763       5
8841  51.478880  32.302844  33.164056  2.417361  5.578714  4.716446       5
1424  50.600987  23.840088  34.426025  3.436487  8.084903  4.229597       5
383   55.049440  30.597256  37.355724  2.042843  7.212628  4.760848       5
9549  52.554025  29.686380  34.654113  2.547031  6.390001  3.861166       5
8900  57.029228  27.637400  36.601265  2.304205  8.857233  4.633507       5
9973  58.870163  30.488760  30.376425  2.362576  7.838134  3.547344       5
8991  59.021062  26.502408  39.311193  3.134490  6.863629  3.623829       5
6140  56.216132  31.396114  30.315207  3.116725  6.589711  4.820198       6
1100  55.555900  26.610658  31.260201  3.336658  9.704965  4.375744       5

      x8::Nh   m1::Drot    m2::Dsh  …  p5::BSP_T     p6::BSP_n  p7::BSP_Pm  \
1615       9  31.169998  14.982997  …   0.597715  11195.409748  663.343955
8841       7  31.302844  17.238739  …   0.668658   7397.856744  536.211651
1424       8  22.840088   9.155660  …   0.547782   9884.953831  579.771048
383        8  29.597256  16.769498  …   0.816518   6940.423795  601.153819
9549       7  28.686380  15.110495  …   0.645179   8129.719165  535.173052
8900       8  26.637400  14.132591  …   0.719547   7930.433873  596.647786
9973       9  29.488760  16.052533  …   0.748494   9375.757817  678.355971
8991       9  25.502408  11.661311  …   0.813600   8065.063887  656.615595
6140       7  30.396114  15.192346  …   0.738781   6833.613261  534.567880
1100       8  25.610658  11.334298  …   0.581282   9688.352815  592.832207

      p8::BSP_Mu  p9::BSP_Irms    p10::MSP_n  p11::UWP_Mu  p12::BSP_wPOT  \
1615   92.859160     22.605125  12396.350201    89.860818    1090.393920
8841   92.749580     17.599793   8070.047900    89.751990     797.205042
1424   92.592101     20.103088  11504.130172    91.398275     878.680266
383    92.575788     20.121891   7617.481491    89.302669     732.678578
9549   92.542233     17.591485   8808.791234    89.350714     762.478772
8900   92.525986     20.129900   8803.100306    90.060338     714.687571
9973   92.501290     22.590789  10017.301319    88.739036     885.507246
8991   92.489776     22.561825   8927.579788    91.017323     674.331372
6140   92.487292     17.586915   7560.701930    91.022111     729.432589
1100   92.474871     20.128094  10968.235522    89.739270     865.893383

      p13::BSP_kt  Valid
1615     0.026442   True
8841     0.037992   True
1424     0.027249   True
383      0.040579   True
9549     0.036676   True
8900     0.035745   True
9973     0.033133   True
8991     0.036061   True
6140     0.042007   True
1100     0.028879   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'
)
```

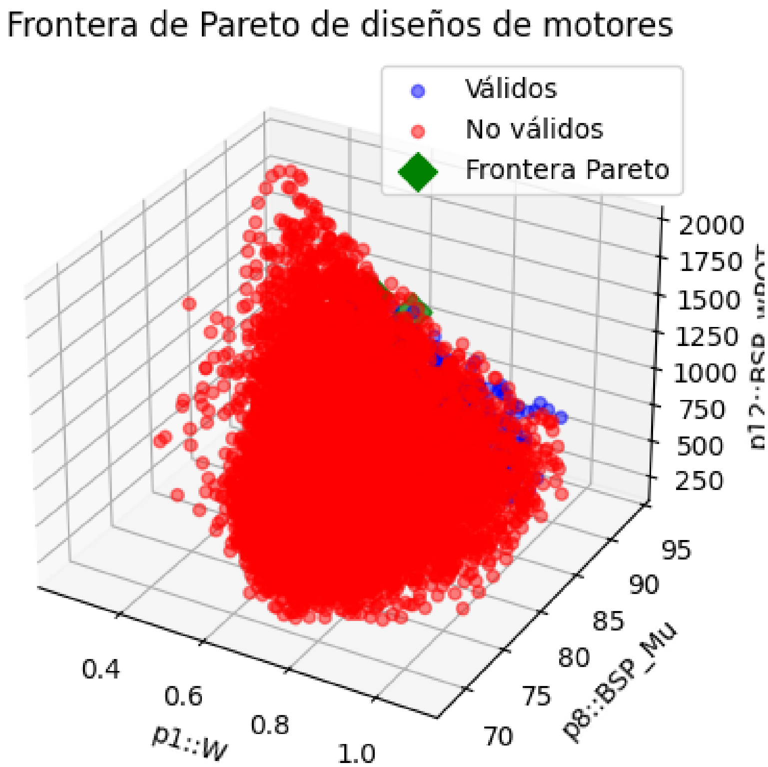
```python
# Motores en la frontera de Pareto en rojo
plt.scatter(
    pareto_motors['p8::BSP_Mu'],
    pareto_motors['p1::W'],
    c='red', label='Frontera Pareto', s=60, marker='o', edgecolors='k'
)

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: 14
Figura guardada en: C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Noteb
ooks_TFM\4.DBG\Figuras_DBG\5000_MOT_Uniforme\Pareto_frontier_2D.png
```



[17]:
```python
# 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           57.900295
x2::Dint          36.680621
x3::L             14.242181
x4::tm             2.096105
x5::hs2              6.5074
x6::wt              3.52096
x7::Nt                   20
x8::Nh                    3
m1::Drot          35.680621
m2::Dsh           21.008234
m3::he             4.102436
m4::Rmag          17.316284
m5::Rs            24.847711
m6::GFF           48.384891
p1::W              0.452416
p4::GFF           59.386588
p5::BSP_T          0.506027
p6::BSP_n       3737.274277
```

```
p7::BSP_Pm        211.375358
p8::BSP_Mu         85.130921
p9::BSP_Irms        7.525288
p10::MSP_n       4526.839019
p11::UWP_Mu        89.502812
p12::BSP_wPOT     467.214853
p13::BSP_kt         0.067244
Valid                   True
Name: 820, dtype: object

Motor más eficiente:
x1::OSD             59.387769
x2::Dint            32.169998
x3::L                21.93708
x4::tm               3.497787
x5::hs2              8.672517
x6::wt               4.697763
x7::Nt                      5
x8::Nh                      9
m1::Drot            31.169998
m2::Dsh             14.982997
m3::he               4.936369
m4::Rmag            14.710552
m5::Rs              24.757516
m6::GFF              35.36341
p1::W                0.608353
p4::GFF             42.142787
p5::BSP_T            0.597715
p6::BSP_n        11195.409748
p7::BSP_Pm         663.343955
p8::BSP_Mu           92.85916
p9::BSP_Irms        22.605125
p10::MSP_n       12396.350201
p11::UWP_Mu         89.860818
p12::BSP_wPOT      1090.39392
p13::BSP_kt          0.026442
Valid                    True
Name: 1615, dtype: object

Motor más eficiente y liviano (score compuesto):
x1::OSD                  53.531872
x2::Dint                 27.089571
x3::L                    21.479058
x4::tm                    2.081564
x5::hs2                   9.469086
x6::wt                    3.831137
x7::Nt                           6
x8::Nh                           9
```

```
m1::Drot                   26.089571
m2::Dsh                    14.186566
m3::he                      3.752064
m4::Rmag                   12.524395
m5::Rs                     23.013871
m6::GFF                     40.59213
p1::W                       0.512947
p4::GFF                    45.135671
p5::BSP_T                   0.541446
p6::BSP_n               10855.906628
p7::BSP_Pm                 617.63368
p8::BSP_Mu                 91.845708
p9::BSP_Irms               22.615686
p10::MSP_n              13048.107926
p11::UWP_Mu                 90.22314
p12::BSP_wPOT            1204.089531
p13::BSP_kt                 0.023941
Valid                           True
p1::W_norm                  0.116112
p8::BSP_Mu_norm             0.870718
p12::BSP_wPOT_norm          0.982633
composite_score             1.754606
Name: 2956, dtype: object
```

[18]:

```python
# ============================================================================
# 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):")
```

```python
print(optimal_motor)

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

```
Motor válido óptimo (según score compuesto):
x1::OSD                    53.531872
x2::Dint                   27.089571
x3::L                      21.479058
x4::tm                      2.081564
x5::hs2                     9.469086
x6::wt                      3.831137
x7::Nt                             6
x8::Nh                             9
m1::Drot                   26.089571
m2::Dsh                    14.186566
m3::he                      3.752064
m4::Rmag                   12.524395
m5::Rs                     23.013871
m6::GFF                     40.59213
p1::W                       0.512947
p4::GFF                    45.135671
p5::BSP_T                   0.541446
p6::BSP_n               10855.906628
p7::BSP_Pm                 617.63368
p8::BSP_Mu                 91.845708
p9::BSP_Irms               22.615686
p10::MSP_n              13048.107926
p11::UWP_Mu                 90.22314
p12::BSP_wPOT            1204.089531
p13::BSP_kt                 0.023941
Valid                           True
p1::W_norm                  0.883888
p8::BSP_Mu_norm             0.870718
p12::BSP_wPOT_norm          0.982633
composite_score             2.737238
Name: 2956, dtype: object
El motor óptimo se ha guardado en: C:\Users\s00244\Documents\GitHub\MotorDesignD
ataDriven\Notebooks_TFM\4.DBG\Modelos_DBG
```

[ ]:

```python

```