ML_V7

May 22, 2025

[1]:
```
# ========================================================================
# Importación de librerías necesarias
# ========================================================================
import os
import re  # Import the regular expression module

import pandas as pd
import numpy as np
import math

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

[2]:
```
# 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
```

[3]:
```
# 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
```

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

[5]:
```
# ============================================================================
# 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␣
 ↪'ML'
db_path = os.path.join(base_path, "DB_ML")
fig_path = os.path.join(base_path, "Figuras_ML")
model_path = os.path.join(base_path, "Modelos_ML")

# 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\2.ML\DB_ML\
design_DB_preprocessed_5000_Uniforme.csv
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\2.ML\Figura
s_ML\5000_MOT_Uniforme
C:\Users\s00244\Documents\GitHub\MotorDesignDataDriven\Notebooks_TFM\2.ML\Modelo
s_ML\5000_MOT_Uniforme
Archivo cargado exitosamente.
```

[6]:
```
# ============================================================================
# 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')

# 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
```

```python
cols_to_keep = variances[variances > threshold].index
y = y[cols_to_keep]

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

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

```
Primeras filas de X:
```

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

```
Primeras filas de y (P):
```

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

	p8::BSP_Mu	p9::BSP_Irms	p10::MSP_n	p11::UWP_Mu
0	90.763855	10.070335	18223.3200	86.138150
1	87.076820	7.558135	5737.1406	88.799880
2	83.929474	7.553457	4325.1235	83.402340
3	87.040310	7.554095	6293.4336	91.343490
4	89.363690	7.554099	5615.5110	91.807846

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

Variancia de cada columna de salida:
p1::W           2.409097e-02
p2::Tnom        1.733798e-33
p3::nnom        0.000000e+00
p4::GFF         1.216293e+02
p5::BSP_T       5.526305e-02
p6::BSP_n       2.691714e+07
p7::BSP_Pm      1.675008e+04
```

```
p8::BSP_Mu      7.538927e+00
p9::BSP_Irms    2.199128e+01
p10::MSP_n      3.204081e+07
p11::UWP_Mu     8.609559e+00
dtype: float64

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

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

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

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

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

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

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

	p9::BSP_Irms	p10::MSP_n	p11::UWP_Mu
1089	15.081463	15475.8150	89.26957
2943	10.061653	4290.1313	82.64902
485	17.622501	5139.3590	90.48702

```
2181      15.106777    6506.3423      87.61828
3311       7.553548    7994.2230      89.99379
```

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

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

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

[10]:
```
# ============================================================================
# 5. DEFINICIÓN DE PIPELINES PARA LOS MODELOS
# ============================================================================
# Se establecen los modelos a probar:
#   - PLS (Partial Least Squares)
#   - Regresión Lineal
#   - Kriging (GPR)
#   - SVR (Support Vector Regression), envuelto en MultiOutputRegressor para␣
 ↪salida multivariable
#   - Random Forest
#   - Artificial Neural Network (ANN), mediante un Multi-layer Perceptron␣
 ↪regressor
#   - Artificial Neural Network (ANN), mediante Keras
```

```
# Pipeline para PLS Regression
pipeline_pls = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', PLSRegression())
])

# Pipeline para Regresión Lineal
pipeline_lr = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', LinearRegression())
])

# Pipeline para Kriging (GPR)
# Se utiliza la suma de un RBF (para modelar la parte suave) y un WhiteKernel␣
 ↪(para el ruido).
kernel = RBF(length_scale=1.0) + WhiteKernel(noise_level=1.0)
# Definir el GaussianProcessRegressor con el kernel anterior y random_state␣
 ↪para reproducibilidad.
gpr = GaussianProcessRegressor(kernel=kernel, random_state=42,␣
 ↪n_restarts_optimizer=10)
# Envolver el GPR en un MultiOutputRegressor, de modo que el pipeline se pueda␣
 ↪aplicar sobre múltiples salidas.
multi_gpr = MultiOutputRegressor(gpr)

pipeline_gpr = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', multi_gpr)
])

# Pipeline para SVR: se utiliza MultiOutputRegressor ya que SVR no soporta␣
 ↪multi-output
pipeline_svr = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', MultiOutputRegressor(SVR()))
])

# Pipeline para Random Forest (RandomForestRegressor maneja multi-output)
pipeline_rf = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', RandomForestRegressor(random_state=42))
])

# Pipeline para Artificial Neural Network (Multi-layer Perceptron regressor)
#8,64,64
pipeline_ann = Pipeline([
    ('preprocessing', data_pipeline),
    ('model', MLPRegressor(hidden_layer_sizes=(100,50),
```

```
                            activation="relu",
                            solver='adam',
                            random_state=42,
                            max_iter=500))
])

# Pipeline para Artificial Neural Network (Keras)
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, permitiendo variar algunos␣
 ↪hiperparámetros.
# @tf.function(reduce_retracing=True)
def ANN_K_model(hidden_layer_size=50, hidden_layer_size_2=3):
    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))  # La salida produce n_out predicciones
    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=50,
    hidden_layer_size_2=3,
    epochs=100,  # valor por defecto
    random_state=42,
    verbose=0
)

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

# Se agrupan los pipelines en un diccionario para iterar y evaluar fácilmente
pipelines = {
    'PLS': pipeline_pls,
    'LR': pipeline_lr,
    'GPR': pipeline_gpr,
    'SVR': pipeline_svr,
    'RF': pipeline_rf,
    'ANN': pipeline_ann,
```

```
    'ANN-K': pipeline_ann_keras
}


# Ejemplo de estructura de uno de los modelos:
# Visualizar el pipeline
set_config(display="diagram")
display(pipeline_pls)
```

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

[11]:
```
# ============================================================================
# 6. VALIDACIÓN CRUZADA: EVALUACIÓN INICIAL DE MODELOS
# ============================================================================
evalInicial_start = time.time()
# Se utilizan 5 particiones (KFold) para evaluar cada modelo mediante Cross␣
 ↪Validation.
# las métricas (MSE y R²) usando las variables de salida escaladas.
cv = KFold(n_splits=5, shuffle=True, random_state=42)

# Diccionario para guardar las métricas de cada modelo
metrics_results = {}

print("\nEvaluación de modelos mediante validación cruzada (sobre y escalado):")

for name, pipe in pipelines.items():
    # Se realizan predicciones en validación cruzada usando y_train_scaled
    y_pred_cv = cross_val_predict(pipe, X_train, y_train_scaled, cv=cv)

    # Cálculo de las métricas por cada columna
    mse_columns = mean_squared_error(y_train_scaled, y_pred_cv,␣
 ↪multioutput='raw_values')
    r2_columns = r2_score(y_train_scaled, y_pred_cv, multioutput='raw_values')

    # Métricas globales (promedio)
    mse_avg = np.mean(mse_columns)
    r2_avg = np.mean(r2_columns)

   # Se asocian las métricas a cada etiqueta de la variable de salida usando el␣
 ↪mismo orden
    metrics_results[name] = {
        'mse_columns': dict(zip(y_train.columns, mse_columns)),
        'r2_columns': dict(zip(y_train.columns, r2_columns)),
        'mse_avg': mse_avg,
        'r2_avg': r2_avg
```

```
    }

    print(f"\nModelo {name}:")
    print("  MSE por columna:")
    for col, mse in metrics_results[name]['mse_columns'].items():
        print(f"    {col}: {mse:.6g}")
    print("  R2 por columna:")
    for col, r2 in metrics_results[name]['r2_columns'].items():
        print(f"    {col}: {r2:.6g}")
    print(f"  MSE promedio: {mse_avg:.4f}")
    print(f"  R2 promedio: {r2_avg:.4f}")

evalInicial_end = time.time()
evalInicial = evalInicial_end - evalInicial_start  # Tiempo transcurrido en␣
 ↪segundos
print(f"Tiempo de computación de la evaluación inicial es: {evalInicial:.2f}␣
 ↪segundos")
```

```
Evaluación de modelos mediante validación cruzada (sobre y escalado):

Modelo PLS:
  MSE por columna:
    p1::W: 0.161802
    p4::GFF: 0.994505
    p5::BSP_T: 0.330181
    p6::BSP_n: 0.29229
    p7::BSP_Pm: 0.143453
    p8::BSP_Mu: 0.480728
    p9::BSP_Irms: 0.250014
    p10::MSP_n: 0.31413
    p11::UWP_Mu: 0.945905
  R2 por columna:
    p1::W: 0.838198
    p4::GFF: 0.00549516
    p5::BSP_T: 0.669819
    p6::BSP_n: 0.70771
    p7::BSP_Pm: 0.856547
    p8::BSP_Mu: 0.519272
    p9::BSP_Irms: 0.749986
    p10::MSP_n: 0.68587
    p11::UWP_Mu: 0.0540949
  MSE promedio: 0.4348
  R2 promedio: 0.5652

Modelo LR:
  MSE por columna:
    p1::W: 0.0250213
```

```
    p4::GFF: 0.199223
    p5::BSP_T: 0.147077
    p6::BSP_n: 0.193906
    p7::BSP_Pm: 0.0393824
    p8::BSP_Mu: 0.243755
    p9::BSP_Irms: 0.00970522
    p10::MSP_n: 0.180276
    p11::UWP_Mu: 0.553176
  R2 por columna:
    p1::W: 0.974979
    p4::GFF: 0.800777
    p5::BSP_T: 0.852923
    p6::BSP_n: 0.806094
    p7::BSP_Pm: 0.960618
    p8::BSP_Mu: 0.756245
    p9::BSP_Irms: 0.990295
    p10::MSP_n: 0.819724
    p11::UWP_Mu: 0.446824
  MSE promedio: 0.1768
  R2 promedio: 0.8232

Modelo GPR:
  MSE por columna:
    p1::W: 0.00584305
    p4::GFF: 0.029877
    p5::BSP_T: 0.00666313
    p6::BSP_n: 0.0230031
    p7::BSP_Pm: 0.0142329
    p8::BSP_Mu: 0.0272831
    p9::BSP_Irms: 0.0100823
    p10::MSP_n: 0.0222077
    p11::UWP_Mu: 0.0315854
  R2 por columna:
    p1::W: 0.994157
    p4::GFF: 0.970123
    p5::BSP_T: 0.993337
    p6::BSP_n: 0.976997
    p7::BSP_Pm: 0.985767
    p8::BSP_Mu: 0.972717
    p9::BSP_Irms: 0.989918
    p10::MSP_n: 0.977792
    p11::UWP_Mu: 0.968415
  MSE promedio: 0.0190
  R2 promedio: 0.9810

Modelo SVR:
  MSE por columna:
    p1::W: 0.0111863
```

```
    p4::GFF: 0.0421387
    p5::BSP_T: 0.0144433
    p6::BSP_n: 0.0416182
    p7::BSP_Pm: 0.0173359
    p8::BSP_Mu: 0.04392
    p9::BSP_Irms: 0.0150402
    p10::MSP_n: 0.0387766
    p11::UWP_Mu: 0.0706417
  R2 por columna:
    p1::W: 0.988814
    p4::GFF: 0.957861
    p5::BSP_T: 0.985557
    p6::BSP_n: 0.958382
    p7::BSP_Pm: 0.982664
    p8::BSP_Mu: 0.95608
    p9::BSP_Irms: 0.98496
    p10::MSP_n: 0.961223
    p11::UWP_Mu: 0.929358
  MSE promedio: 0.0328
  R2 promedio: 0.9672

Modelo RF:
  MSE por columna:
    p1::W: 0.0894283
    p4::GFF: 0.329621
    p5::BSP_T: 0.0760568
    p6::BSP_n: 0.0653513
    p7::BSP_Pm: 0.0426185
    p8::BSP_Mu: 0.118425
    p9::BSP_Irms: 0.0387571
    p10::MSP_n: 0.0645744
    p11::UWP_Mu: 0.256301
  R2 por columna:
    p1::W: 0.910572
    p4::GFF: 0.670379
    p5::BSP_T: 0.923943
    p6::BSP_n: 0.934649
    p7::BSP_Pm: 0.957381
    p8::BSP_Mu: 0.881575
    p9::BSP_Irms: 0.961243
    p10::MSP_n: 0.935426
    p11::UWP_Mu: 0.743699
  MSE promedio: 0.1201
  R2 promedio: 0.8799

Modelo ANN:
  MSE por columna:
    p1::W: 0.00961657
```

```
    p4::GFF: 0.034631
    p5::BSP_T: 0.013713
    p6::BSP_n: 0.0199575
    p7::BSP_Pm: 0.0168245
    p8::BSP_Mu: 0.0278616
    p9::BSP_Irms: 0.0134479
    p10::MSP_n: 0.0191995
    p11::UWP_Mu: 0.0329539
  R2 por columna:
    p1::W: 0.990383
    p4::GFF: 0.965369
    p5::BSP_T: 0.986287
    p6::BSP_n: 0.980043
    p7::BSP_Pm: 0.983175
    p8::BSP_Mu: 0.972138
    p9::BSP_Irms: 0.986552
    p10::MSP_n: 0.980801
    p11::UWP_Mu: 0.967046
  MSE promedio: 0.0209
  R2 promedio: 0.9791

Modelo ANN-K:
  MSE por columna:
    p1::W: 0.20895
    p4::GFF: 0.292982
    p5::BSP_T: 0.109969
    p6::BSP_n: 0.0723461
    p7::BSP_Pm: 0.051204
    p8::BSP_Mu: 0.627061
    p9::BSP_Irms: 0.0998334
    p10::MSP_n: 0.0795597
    p11::UWP_Mu: 0.41723
  R2 por columna:
    p1::W: 0.79105
    p4::GFF: 0.707018
    p5::BSP_T: 0.890031
    p6::BSP_n: 0.927654
    p7::BSP_Pm: 0.948796
    p8::BSP_Mu: 0.372939
    p9::BSP_Irms: 0.900167
    p10::MSP_n: 0.92044
    p11::UWP_Mu: 0.58277
  MSE promedio: 0.2177
  R2 promedio: 0.7823
Tiempo de computación de la evaluación inicial es: 5712.68 segundos
```

[12]:
```
# ============================================================================
# 7. REPRESENTACIÓN DE RESULTADOS DE LA VALIDACIÓN CRUZADA
# ============================================================================
# Visualización mejorada para validación cruzada:
# 1. Crear un DataFrame resumen con las métricas promedio para cada modelo
summary_cv = pd.DataFrame({
    'Modelo': list(metrics_results.keys()),
    'R2_promedio': [metrics_results[m]['r2_avg'] for m in metrics_results],
    'MSE_promedio': [metrics_results[m]['mse_avg'] for m in metrics_results]
})

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

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

# Gráfico de MSE Promedio
bars2 = ax[1].bar(summary_cv['Modelo'], summary_cv['MSE_promedio'],␣
 ↪color='salmon')
ax[1].set_title('MSE Promedio (Validación Cruzada)')
ax[1].set_xlabel('Modelo')
ax[1].set_ylabel('MSE Promedio')
for bar in bars2:
    yval = bar.get_height()
    ax[1].text(bar.get_x() + bar.get_width()/2, yval + yval*0.01, f'{yval:.
 ↪1f}', ha='center', va='bottom')

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

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

# Gráfico para R2:
```

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

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

[13]:
```
# ============================================================================
# 8. HIPERPARAMETRIZACIÓN DE LOS MODELOS
# ============================================================================
# Se definirá una búsqueda en grilla (GridSearchCV) para encontrar los mejores␣
 ↪parámetros.
# Para cada modelo se define un espacio de búsqueda. Se omite LR por no tener␣
 ↪muchos
# hiperparámetros.

# Diccionario para guardar GridSearchCV ajustado para cada modelo
```

```
gridsearch_results = {}

# Medir el tiempo de entrenamiento (computación)
start_time_hiperparameters = time.time()    # Tiempo de inicio
```

[14]:
```
# ============================================================================
# Paso 8.1: PLS - Hiperparámetros
# ============================================================================
# Medimos el tiempo de ejecución
start_time_PLS = time.time()

# Parámetros para PLS: solo se ajusta el número de componentes
param_grid_pls = {
    'model__n_components': np.arange(1, min(len(X.columns), 20)),
    # Escalar o no la salida interna de PLS (True por defecto)
    'model__scale': [True, False],
    # Número máximo de iteraciones para el algoritmo NIPALS
    'model__max_iter': [500, 1000, 2000],
    # Tolerancia de convergencia
    'model__tol': [1e-06, 1e-05, 1e-04, 1e-03]
}
#gs_pls = GridSearchCV(pipeline_pls, param_grid=param_grid_pls, cv=cv,␣
 ↪scoring='r2', n_jobs=-1)
gs_pls = GridSearchCV(pipeline_pls,
                      param_grid=param_grid_pls,
                      cv=cv,
                      scoring='r2',
                      n_jobs=-1)
gs_pls.fit(X_train, y_train_scaled)
gridsearch_results['PLS'] = gs_pls

print("=== Optimización de PLS ===")
print("Mejores parámetros para PLS:")
print(gs_pls.best_params_)
print(f"Mejor R2 en validación: {gs_pls.best_score_:.4f}")
print("\n")

end_time_PLS = time.time()     # Tiempo de fin
elapsed_time = end_time_PLS - start_time_PLS  # Tiempo transcurrido en segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{elapsed_time:.2f} segundos")
```

```
=== Optimización de PLS ===
Mejores parámetros para PLS:
{'model__max_iter': 500, 'model__n_components': np.int64(7), 'model__scale':
False, 'model__tol': 1e-06}
Mejor R2 en validación: 0.8227
```

Tiempo de computación del entrenamiento de hiperparámetros: 4.92 segundos

[15]:
```
# ================================================================================
# Paso 8.2: Regresión Lineal (LR) - Hiperparámetros
# ================================================================================
# Medimos el tiempo de ejecución
start_time_LR = time.time()

# Para LinearRegression, se optimiza el parámetro "fit_intercept"
param_grid_lr = {
    # Ajustar o no el intercepto
    'model__fit_intercept': [True, False],
    # Forzar coeficientes positivos (disponible en sklearn 1.1)
    'model__positive': [True, False],
    # Copiar X antes de procesar
    'model__copy_X': [True, False]
}
gs_lr = GridSearchCV(pipeline_lr,
                     param_grid = param_grid_lr,
                     cv=cv,
                     scoring='r2',
                     n_jobs=-1)
gs_lr.fit(X_train, y_train_scaled)
gridsearch_results['LR'] = gs_lr

print("=== Optimización de LR ===")
print("Mejores parámetros para LR:")
print(gs_lr.best_params_)
print(f"Mejor R2 en validación: {gs_lr.best_score_:.4f}")
print("\n")

end_time_LR = time.time()     # Tiempo de fin
elapsed_time = end_time_LR - start_time_LR  # Tiempo transcurrido en segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{elapsed_time:.2f} segundos")
```

```
=== Optimización de LR ===
Mejores parámetros para LR:
{'model__copy_X': True, 'model__fit_intercept': True, 'model__positive': False}
Mejor R2 en validación: 0.8227
```

Tiempo de computación del entrenamiento de hiperparámetros: 0.18 segundos

[16]:
```
# ================================================================================
# Paso 8.3: KRIGING (GPR MULTISALIDA) - Hiperparámetros
# ================================================================================
```

```
# Medimos el tiempo de ejecución
start_time_GPR = time.time()

param_grid_gpr = {
    # Longitud de escala del componente RBF (se usa escala logarítmica)
    "model__estimator__kernel__k1__length_scale": Real(1e-2, 1e3,␣
 ↪prior="log-uniform"),
    # Nivel de ruido del componente WhiteKernel (escala logarítmica)
    "model__estimator__kernel__k2__noise_level": Real(1e-8, 1e+2,␣
 ↪prior="log-uniform"),
    # Número de reinicios del optimizador
    # "model__estimator__n_restarts_optimizer": Integer(10),
    # Parámetro alpha, que añade ruido en la diagonal de la matriz de covarianza
    "model__estimator__alpha": Real(1e-10, 1e-1, prior="log-uniform"),
    # Opción para normalizar la salida (bool)
    "model__estimator__normalize_y": Categorical([True, False])
}

# Configurar la optimización bayesiana con BayesSearchCV
gs_gpr = BayesSearchCV(estimator=pipeline_gpr,
                       search_spaces=param_grid_gpr,
                       n_iter=2,           # número de iteraciones de búsqueda
                       cv=cv,                # validación cruzada de 5 pliegues
                       scoring="r2",
                       random_state=42,
                       n_jobs=-1)
# Ejecutar la búsqueda sobre los datos escalados
gs_gpr.fit(X_train, y_train_scaled)
gridsearch_results['GPR'] = gs_gpr

print("=== Optimización de GPR ===")
print("Mejores parámetros para LR:")
print(gs_gpr.best_params_)
print(f"Mejor R2 en validación: {gs_gpr.best_score_:.4f}")
print("\n")

end_time_GPR = time.time()     # Tiempo de fin
elapsed_time = end_time_GPR - start_time_GPR  # Tiempo transcurrido en segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{elapsed_time:.2f} segundos")
```

```
=== Optimización de GPR ===
Mejores parámetros para LR:
OrderedDict({'model__estimator__alpha': 0.0034394990986855454,
'model__estimator__kernel__k1__length_scale': 260.9614680853858,
'model__estimator__kernel__k2__noise_level': 1.0816857261904209e-05,
'model__estimator__normalize_y': False})
```

```
Mejor R2 en validación: 0.9810


Tiempo de computación del entrenamiento de hiperparámetros: 7119.70 segundos
```

[17]:
```python
# ============================================================================
# Paso 8.4: Support Vector Regression (SVR) - Hiperparámetros
# ============================================================================
# Medimos el tiempo de ejecución
start_time_SVR = time.time()

# Parámetros para SVR: se ajustan C y epsilon
param_grid_svr = {
    # Penalización de la función de pérdida
    'model__estimator__C': [0.1, 1, 10, 100],
    # Zona de insensibilidad
    'model__estimator__epsilon': [0.001, 0.01, 0.1, 0.5, 1.0],
    # Tipo de kernel
    'model__estimator__kernel': ['rbf', 'linear', 'poly'],
    # Coeficiente gamma para kernels RBF y poly
    'model__estimator__gamma': ['scale', 'auto']
}
gs_svr = GridSearchCV(pipeline_svr,
                      param_grid=param_grid_svr,
                      cv=cv,
                      scoring='r2',
                      n_jobs=-1)
gs_svr.fit(X_train, y_train_scaled)
gridsearch_results['SVR'] = gs_svr

print("=== Optimización de SVR ===")
print("Mejores parámetros para SVR:")
print(gs_svr.best_params_)
print(f"Mejor R2 en validación: {gs_svr.best_score_:.4f}")
print("\n")

end_time_SVR = time.time()     # Tiempo de fin
elapsed_time = end_time_SVR - start_time_SVR  # Tiempo transcurrido en segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
  ↪{elapsed_time:.2f} segundos")
```

```
=== Optimización de SVR ===
Mejores parámetros para SVR:
{'model__estimator__C': 10, 'model__estimator__epsilon': 0.001,
'model__estimator__gamma': 'scale', 'model__estimator__kernel': 'rbf'}
Mejor R2 en validación: 0.9837
```

```
Tiempo de computación del entrenamiento de hiperparámetros: 4709.85 segundos
```

[18]:
```python
# ============================================================================
# Paso 8.5: Random Forest (RF) - Hiperparámetros
# ============================================================================
# Medimos el tiempo de ejecución
start_time_RF = time.time()

# Definición del grid de hiperparámetros para RandomForestRegressor
param_grid_rf = {
    # Número de árboles en el bosque
    'model__n_estimators': [50, 100, 200, 300],
    # Profundidad máxima de cada árbol
    'model__max_depth': [None, 10, 20, 30, 50],
    # Número mínimo de muestras para dividir un nodo
    'model__min_samples_split': [2, 5, 10, 20],
    # Número mínimo de muestras en cada hoja
    'model__min_samples_leaf': [1, 2, 4, 6],
    # Número de características a considerar al buscar la mejor división
    'model__max_features': ['auto', 'sqrt', 'log2']
}
gs_rf = GridSearchCV(pipeline_rf,
                     param_grid=param_grid_rf,
                     cv=cv,
                     scoring='r2',
                     n_jobs=-1)

gs_rf.fit(X_train, y_train_scaled)
gridsearch_results['RF'] = gs_rf

print("=== Optimización de RF ===")
print("Mejores parámetros para Random Forest:")
print(gs_rf.best_params_)
print(f"Mejor R2 en validación: {gs_rf.best_score_:.4f}")
print("\n")

end_time_RF = time.time()     # Tiempo de fin
elapsed_time = end_time_RF - start_time_RF  # Tiempo transcurrido en segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
  ↪{elapsed_time:.2f} segundos")
```

```
=== Optimización de RF ===
Mejores parámetros para Random Forest:
{'model__max_depth': None, 'model__max_features': 'log2',
'model__min_samples_leaf': 1, 'model__min_samples_split': 2,
'model__n_estimators': 300}
Mejor R2 en validación: 0.8772
```

```
Tiempo de computación del entrenamiento de hiperparámetros: 312.17 segundos
```

[19]:
```python
# ================================================================================
# Paso 8.6: Artificial Neural Network (ANN) - Hiperparámetros
# ================================================================================
# Medimos el tiempo de ejecución
start_time_ANN = time.time()

'''
# Parámetros para Artificial Neural Network:
param_grid_ann = {
    'model__hidden_layer_sizes': [(50,), (100,), (200,), (300,)],
    'model__max_iter': [250, 500, 750, 1000]
}
'''

param_grid_ann = {
    # Tamaños de capa oculta: 1 o 2 capas, varias configuraciones
    'model__hidden_layer_sizes': [
        (50,), (100,), (50,50), (100,50), (100,100)
    ],
    # Función de activación
    'model__activation': ['relu', 'tanh', 'logistic'],
    # Algoritmo de optimización
    'model__solver': ['adam', 'lbfgs', 'sgd'],
    # Tasa de aprendizaje inicial (solo para 'sgd' y 'adam')
    'model__learning_rate_init': [1e-4, 1e-3, 1e-2],
    # Parámetro de regularización L2
    'model__alpha': [1e-5, 1e-4, 1e-3],
    # Número máximo de iteraciones
    'model__max_iter': [200, 500, 1000]
}

gs_ann = GridSearchCV(pipeline_ann,
                      param_grid=param_grid_ann,
                      cv=cv,
                      scoring='r2',
                      n_jobs=-1)

gs_ann.fit(X_train, y_train_scaled)
gridsearch_results['ANN'] = gs_ann

print("=== Optimización de ANN ===")
print("Mejores parámetros para Artificial Neural Network:")
print(gs_ann.best_params_)
print(f"Mejor R2 en validación: {gs_ann.best_score_:.4f}")
```

```
print("\n")

end_time_ANN = time.time()      # Tiempo de fin
elapsed_time = end_time_ANN - start_time_ANN  # Tiempo transcurrido en segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{elapsed_time:.2f} segundos")
```

```
=== Optimización de ANN ===
Mejores parámetros para Artificial Neural Network:
{'model__activation': 'logistic', 'model__alpha': 0.0001,
'model__hidden_layer_sizes': (100, 100), 'model__learning_rate_init': 0.0001,
'model__max_iter': 1000, 'model__solver': 'lbfgs'}
Mejor R2 en validación: 0.9866


Tiempo de computación del entrenamiento de hiperparámetros: 5316.10 segundos
```

[20]:
```
# ============================================================================
# Paso 8.7: Artificial Neural Network mediante Keras(ANN-K) - Hiperparámetros
# ============================================================================
# Medimos el tiempo de ejecución
start_time_ANN_K = time.time()

# Parámetros para Artificial Neural Network:
'''
param_grid_ann_k = {
    'model__hidden_layer_size': [50, 100, 200, 300],  # Número de neuronas en␣
 ↪la primera capa oculta
    'model__hidden_layer_size_2': [3, 5, 10],           # Número de neuronas en␣
 ↪la segunda capa oculta
    'model__epochs': [250, 500, 750, 1000]              # Número de épocas de␣
 ↪entrenamiento
}

param_grid_ann_k = {
    'model__hidden_layer_size': [50, 100, 200, 300],  # Número de neuronas en␣
 ↪la primera capa oculta
    'model__hidden_layer_size_2': [3, 5, 10, 15],           # Número de␣
 ↪neuronas en la segunda capa oculta
    'model__epochs': [100, 200, 300]               # Número de épocas de␣
 ↪entrenamiento
}
'''
# Definición del grid de hiperparámetros para el wrapper KerasRegressor
param_grid_ann_k = {
    # Tamaño de la primera capa oculta
    'model__hidden_layer_size': [50, 100, 200],
```

```
    # Tamaño de la segunda capa oculta
    'model__hidden_layer_size_2': [3, 10, 20],
    # Número de épocas
    'model__epochs': [100, 300, 500],
    # Tamaño del batch
    'model__batch_size': [16, 32]
}

gs_ann_k = GridSearchCV(pipeline_ann_keras,
                        param_grid=param_grid_ann_k,
                        cv=cv,
                        scoring='r2',
                        n_jobs=-1)

gs_ann_k.fit(X_train, y_train_scaled)
gridsearch_results['ANN-K'] = gs_ann_k

print("=== Optimización de ANN-K ===")
print("Mejores parámetros para Artificial Neural Network mediante Keras:")
print(gs_ann_k.best_params_)
print(f"Mejor R2 en validación: {gs_ann_k.best_score_:.4f}")

end_time_ANN_K = time.time()     # Tiempo de fin
elapsed_time = end_time_ANN_K - start_time_ANN_K  # Tiempo transcurrido en␣
 ↪segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{elapsed_time:.2f} segundos")
```

```
=== Optimización de ANN-K ===
Mejores parámetros para Artificial Neural Network mediante Keras:
{'model__batch_size': 16, 'model__epochs': 500, 'model__hidden_layer_size': 100,
'model__hidden_layer_size_2': 10}
Mejor R2 en validación: 0.9836
Tiempo de computación del entrenamiento de hiperparámetros: 1872.76 segundos
```

[21]:
```
# ============================================================================
end_time_hiperparameters = time.time()     # Tiempo de fin
elapsed_time = end_time_hiperparameters - start_time_hiperparameters  # Tiempo␣
 ↪transcurrido en segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{elapsed_time:.2f} segundos")
```

```
Tiempo de computación del entrenamiento de hiperparámetros: 19335.77 segundos
```

[22]:
```
# ============================================================================
# 9. EVALUACIÓN DE LOS MODELOS AJUSTADOS SOBRE EL CONJUNTO DE TEST
# ============================================================================
evalFinal_start = time.time()
```

```
# Se evalúan los modelos con mejor hiperparametrización sobre el conjunto de test.
# Se calcularán las métricas finales para cada modelo.

final_metrics = {}

# Creamos un diccionario que asocie cada modelo con su GridSearchCV ajustado (o el pipeline original en el caso de LR)
final_models = {
    'PLS': gs_pls.best_estimator_,
    'LR': gs_lr.best_estimator_,
    'GPR': gs_gpr.best_estimator_,
    'SVR': gs_svr.best_estimator_,
    'RF': gs_rf.best_estimator_,
    'ANN': gs_ann.best_estimator_,
    'ANN-K': gs_ann_k.best_estimator_
}

print("\nEvaluación final de los modelos en el conjunto de test:")
for name, model in final_models.items():
    # Entrenamos el modelo con X_train y y_train_scaled y predecimos sobre X_test
    model.fit(X_train, y_train_scaled)
    y_pred = model.predict(X_test)

    mse_columns = mean_squared_error(y_test_scaled, y_pred, multioutput='raw_values')
    r2_columns = r2_score(y_test_scaled, y_pred, multioutput='raw_values')
    mse_avg = np.mean(mse_columns)
    r2_avg  = np.mean(r2_columns)

    final_metrics[name] = {
        'mse_columns': dict(zip(y_test.columns, mse_columns)),
        'r2_columns': dict(zip(y_test.columns, r2_columns)),
        'mse_avg': mse_avg,
        'r2_avg': r2_avg
    }

    print(f"\nModelo {name}:")
    print("  MSE por columna:")
    for col, mse in final_metrics[name]['mse_columns'].items():
        print(f"    {col}: {mse:.6g}")
    print("  R2 por columna:")
    for col, r2 in final_metrics[name]['r2_columns'].items():
        print(f"    {col}: {r2:.6g}")
    print(f"  MSE promedio: {mse_avg:.4f}")
    print(f"  R2 promedio: {r2_avg:.4f}")
```

```
evalFinal_end = time.time()
evalFinal = evalFinal_end - evalFinal_start  # Tiempo transcurrido en segundos
print(f"Tiempo de computación de la evaluación final es: {evalFinal:.2f}␣
 ↪segundos")
```

```
Evaluación final de los modelos en el conjunto de test:

Modelo PLS:
  MSE por columna:
    p1::W: 0.0588025
    p4::GFF: 0.209985
    p5::BSP_T: 0.198094
    p6::BSP_n: 0.225535
    p7::BSP_Pm: 0.0380766
    p8::BSP_Mu: 0.274666
    p9::BSP_Irms: 0.00743104
    p10::MSP_n: 0.213535
    p11::UWP_Mu: 0.472903
  R2 por columna:
    p1::W: 0.940773
    p4::GFF: 0.783554
    p5::BSP_T: 0.805885
    p6::BSP_n: 0.758764
    p7::BSP_Pm: 0.96252
    p8::BSP_Mu: 0.722572
    p9::BSP_Irms: 0.992538
    p10::MSP_n: 0.771376
    p11::UWP_Mu: 0.479903
  MSE promedio: 0.1888
  R2 promedio: 0.8020

Modelo LR:
  MSE por columna:
    p1::W: 0.0581387
    p4::GFF: 0.210022
    p5::BSP_T: 0.198151
    p6::BSP_n: 0.225707
    p7::BSP_Pm: 0.0380832
    p8::BSP_Mu: 0.27547
    p9::BSP_Irms: 0.00740015
    p10::MSP_n: 0.213603
    p11::UWP_Mu: 0.472973
  R2 por columna:
    p1::W: 0.941442
    p4::GFF: 0.783515
    p5::BSP_T: 0.80583
```

```
    p6::BSP_n: 0.75858
    p7::BSP_Pm: 0.962514
    p8::BSP_Mu: 0.721761
    p9::BSP_Irms: 0.992569
    p10::MSP_n: 0.771303
    p11::UWP_Mu: 0.479826
  MSE promedio: 0.1888
  R2 promedio: 0.8019

Modelo GPR:
  MSE por columna:
    p1::W: 0.0387676
    p4::GFF: 0.0532277
    p5::BSP_T: 0.0504869
    p6::BSP_n: 0.0369202
    p7::BSP_Pm: 0.00992826
    p8::BSP_Mu: 0.0537823
    p9::BSP_Irms: 0.00779623
    p10::MSP_n: 0.0358397
    p11::UWP_Mu: 0.0782774
  R2 por columna:
    p1::W: 0.960953
    p4::GFF: 0.945134
    p5::BSP_T: 0.950527
    p6::BSP_n: 0.96051
    p7::BSP_Pm: 0.990227
    p8::BSP_Mu: 0.945677
    p9::BSP_Irms: 0.992171
    p10::MSP_n: 0.961628
    p11::UWP_Mu: 0.913911
  MSE promedio: 0.0406
  R2 promedio: 0.9579

Modelo SVR:
  MSE por columna:
    p1::W: 0.0337514
    p4::GFF: 0.0563322
    p5::BSP_T: 0.0397366
    p6::BSP_n: 0.0290929
    p7::BSP_Pm: 0.00847176
    p8::BSP_Mu: 0.0564957
    p9::BSP_Irms: 0.00689287
    p10::MSP_n: 0.0280925
    p11::UWP_Mu: 0.0744723
  R2 por columna:
    p1::W: 0.966005
    p4::GFF: 0.941934
    p5::BSP_T: 0.961062
```

```
    p6::BSP_n: 0.968882
    p7::BSP_Pm: 0.991661
    p8::BSP_Mu: 0.942936
    p9::BSP_Irms: 0.993078
    p10::MSP_n: 0.969922
    p11::UWP_Mu: 0.918096
  MSE promedio: 0.0370
  R2 promedio: 0.9615

Modelo RF:
  MSE por columna:
    p1::W: 0.0959427
    p4::GFF: 0.30554
    p5::BSP_T: 0.087751
    p6::BSP_n: 0.0677377
    p7::BSP_Pm: 0.0311735
    p8::BSP_Mu: 0.126089
    p9::BSP_Irms: 0.0295933
    p10::MSP_n: 0.0653962
    p11::UWP_Mu: 0.23294
  R2 por columna:
    p1::W: 0.903365
    p4::GFF: 0.685059
    p5::BSP_T: 0.914012
    p6::BSP_n: 0.927547
    p7::BSP_Pm: 0.969315
    p8::BSP_Mu: 0.872644
    p9::BSP_Irms: 0.970282
    p10::MSP_n: 0.929983
    p11::UWP_Mu: 0.743814
  MSE promedio: 0.1158
  R2 promedio: 0.8796

Modelo ANN:
  MSE por columna:
    p1::W: 0.0422098
    p4::GFF: 0.0551518
    p5::BSP_T: 0.0577526
    p6::BSP_n: 0.0348281
    p7::BSP_Pm: 0.0113981
    p8::BSP_Mu: 0.0575455
    p9::BSP_Irms: 0.00944789
    p10::MSP_n: 0.034416
    p11::UWP_Mu: 0.0758777
  R2 por columna:
    p1::W: 0.957486
    p4::GFF: 0.943151
    p5::BSP_T: 0.943407
```

```
    p6::BSP_n: 0.962747
    p7::BSP_Pm: 0.988781
    p8::BSP_Mu: 0.941876
    p9::BSP_Irms: 0.990512
    p10::MSP_n: 0.963152
    p11::UWP_Mu: 0.91655
  MSE promedio: 0.0421
  R2 promedio: 0.9564

Modelo ANN-K:
  MSE por columna:
    p1::W: 0.040933
    p4::GFF: 0.0604569
    p5::BSP_T: 0.0619702
    p6::BSP_n: 0.0355571
    p7::BSP_Pm: 0.0129243
    p8::BSP_Mu: 0.0627088
    p9::BSP_Irms: 0.00990742
    p10::MSP_n: 0.0348376
    p11::UWP_Mu: 0.0702568
  R2 por columna:
    p1::W: 0.958772
    p4::GFF: 0.937683
    p5::BSP_T: 0.939275
    p6::BSP_n: 0.961968
    p7::BSP_Pm: 0.987278
    p8::BSP_Mu: 0.936661
    p9::BSP_Irms: 0.990051
    p10::MSP_n: 0.962701
    p11::UWP_Mu: 0.922732
  MSE promedio: 0.0433
  R2 promedio: 0.9552
Tiempo de computación de la evaluación final es: 1363.44 segundos
```

[23]:
```python
# ============================================================================
# 9.1. GUARDAMOS LOS HIPERPARÁMETROS EN UN FICHERO.
# ============================================================================
import json

# Extraer los mejores hiperparámetros de cada búsqueda
best_hyperparams = {
    model_name: gs.best_params_
    for model_name, gs in gridsearch_results.items()
}

# Convertir valores numpy a tipos nativos para que json.dump no falle
# (opcional si quieres asegurarte de que todo sea serializable)
```

```python
for params in best_hyperparams.values():
    for k, v in params.items():
        # si es numpy scalar, convertir a Python nativo
        if hasattr(v, "item"):
            params[k] = v.item()

# Ruta al fichero donde se almacenarán los hiperparámetros
hyperparams_file = os.path.join(modelo_path, "hyperparameters.json")

# Guardar en formato JSON con indentación para lectura
with open(hyperparams_file, "w") as f:
    json.dump(best_hyperparams, f, indent=4)

print(f"Hiperparámetros de cada modelo guardados en: {hyperparams_file}")

# Leer y pintar el JSON con indentación
with open(hyperparams_file, "r") as f:
    hyperparams = json.load(f)

print("Contenido de hyperparameters.json:")
print(json.dumps(hyperparams, indent=4))
```

```
Hiperparámetros de cada modelo guardados en: C:\Users\s00244\Documents\GitHub\Mo
torDesignDataDriven\Notebooks_TFM\2.ML\Modelos_ML\5000_MOT_Uniforme\hyperparamet
ers.json
Contenido de hyperparameters.json:
{
    "PLS": {
        "model__max_iter": 500,
        "model__n_components": 7,
        "model__scale": false,
        "model__tol": 1e-06
    },
    "LR": {
        "model__copy_X": true,
        "model__fit_intercept": true,
        "model__positive": false
    },
    "GPR": {
        "model__estimator__alpha": 0.0034394990986855454,
        "model__estimator__kernel__k1__length_scale": 260.9614680853858,
        "model__estimator__kernel__k2__noise_level": 1.0816857261904209e-05,
        "model__estimator__normalize_y": false
    },
    "SVR": {
        "model__estimator__C": 10,
        "model__estimator__epsilon": 0.001,
        "model__estimator__gamma": "scale",
```

```
        "model__estimator__kernel": "rbf"
    },
    "RF": {
        "model__max_depth": null,
        "model__max_features": "log2",
        "model__min_samples_leaf": 1,
        "model__min_samples_split": 2,
        "model__n_estimators": 300
    },
    "ANN": {
        "model__activation": "logistic",
        "model__alpha": 0.0001,
        "model__hidden_layer_sizes": [
            100,
            100
        ],
        "model__learning_rate_init": 0.0001,
        "model__max_iter": 1000,
        "model__solver": "lbfgs"
    },
    "ANN-K": {
        "model__batch_size": 16,
        "model__epochs": 500,
        "model__hidden_layer_size": 100,
        "model__hidden_layer_size_2": 10
    }
}
```

[24]:
```
# ================================================================================
# 9.2. REPRESENTACIÓN DE RESULTADOS DE LA VALIDACIÓN CRUZADA
# ================================================================================
# Visualización mejorada para validación cruzada:
# 1. Crear un DataFrame resumen con las métricas promedio para cada modelo
summary_cv = pd.DataFrame({
    'Modelo': list(final_metrics.keys()),
    'R2_promedio': [final_metrics[m]['r2_avg'] for m in final_metrics],
    'MSE_promedio': [final_metrics[m]['mse_avg'] for m in final_metrics]
})

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

# Gráfico de R2 Promedio
bars1 = ax[0].bar(summary_cv['Modelo'], summary_cv['R2_promedio'],␣
 ↪color='skyblue')
ax[0].set_title(r'$R^2$ Promedio (Validación Cruzada)')
ax[0].set_xlabel('Modelo')
```

```
ax[0].set_ylabel(r'$R^2$ Promedio')
ax[0].set_ylim([0, 1])
for bar in bars1:
    yval = bar.get_height()
    ax[0].text(bar.get_x() + bar.get_width()/2, yval + 0.01, f'{yval:.3f}',␣
 ↪ha='center', va='bottom')

# Gráfico de MSE Promedio
bars2 = ax[1].bar(summary_cv['Modelo'], summary_cv['MSE_promedio'],␣
 ↪color='salmon')
ax[1].set_title('MSE Promedio (Validación Cruzada)')
ax[1].set_xlabel('Modelo')
ax[1].set_ylabel('MSE Promedio')
for bar in bars2:
    yval = bar.get_height()
    ax[1].text(bar.get_x() + bar.get_width()/2, yval + yval*0.01, f'{yval:.
 ↪1f}', ha='center', va='bottom')

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

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

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

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

[25]:
```
# Tiempo de fin
end_time_program = time.time()
program_time = end_time_program - start_time_program  # Tiempo transcurrido en␣
 ↪segundos
print(f"Tiempo de computación del entrenamiento de hiperparámetros:␣
 ↪{program_time:.2f} segundos")
```

```
Tiempo de computación del entrenamiento de hiperparámetros: 26425.44 segundos
```

[ ]:
```
```