Laboratorio 10: Git Wars Superheroes

Optimización Bayesiana para Predicción de Poderes de Superhéroes

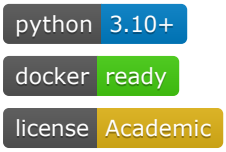


Tabla de Contenidos

1. Descripción General
2. Fundamentos Teóricos
3. Arquitectura del Sistema
4. Guía de Inicio Rápido
5. Optimización Bayesiana
6. Despliegue
7. Resultados y Análisis

Descripción General

Este proyecto implementa un **sistema end-to-end de Machine Learning** que utiliza **Optimización Bayesiana (BO)** para encontrar los hiperparámetros óptimos de modelos de regresión aplicados a la predicción del poder de superhéroes. El sistema completo incluye:

- Consumo y limpieza de datos desde SuperHero API
- Implementación manual de Optimización Bayesiana con Gaussian Processes
- Comparativa rigurosa entre BO y Random Search
- API REST completamente funcional y desplegable
- Containerización con Docker y automatización con Makefile
- Despliegue en Render con validación de coherencia

Objetivo académico: Demostrar comprensión profunda de la teoría detrás de la Optimización Bayesiana, implementándola desde cero sin librerías especializadas, y construir un pipeline de ML reproducible y profesional.

Fundamentos Teóricos

¿Qué es la Optimización Bayesiana?

La **Optimización Bayesiana (BO)** es un método de optimización secuencial diseñado para funciones costosas de evaluar (black-box functions). En lugar de probar hiperparámetros aleatoriamente, BO construye un **modelo probabilístico** del espacio de búsqueda y lo utiliza para tomar decisiones inteligentes sobre qué configuración evaluar a continuación.

Componentes Clave

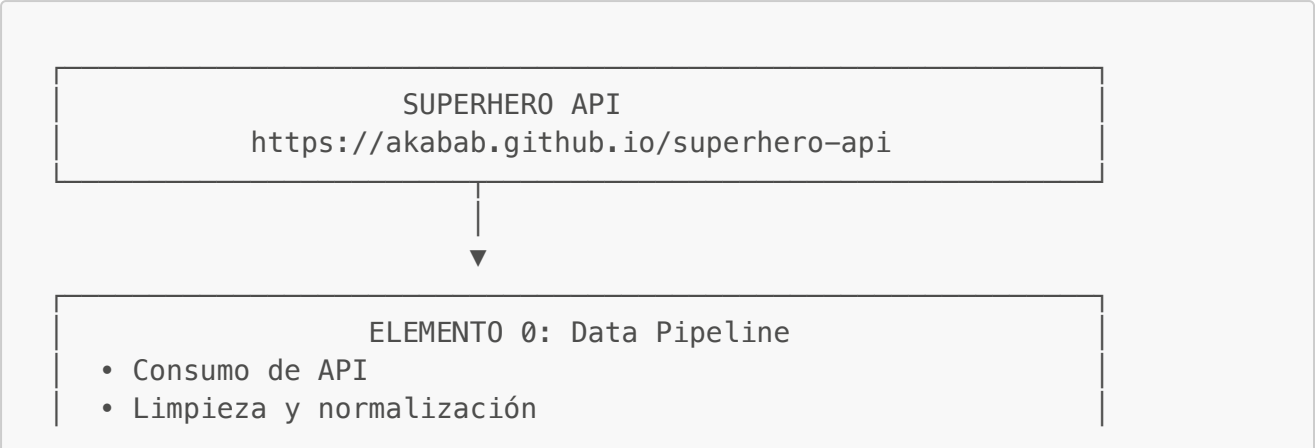
1. **Modelo Surrogate (Gaussian Process)**
   - Aproxima la función objetivo (performance del modelo)
   - Proporciona media μ(x) y varianza σ²(x) para cada punto
   - Se actualiza con cada nueva evaluación
2. **Función de Adquisición (UCB)**
   - Determina qué punto explorar siguiente
   - Balancea exploración (alta incertidumbre) vs. explotación (alto rendimiento esperado)
   - Fórmula: $UCB(x) = \mu(x) + \kappa\sigma(x)$ donde κ controla el balance
3. **Kernel RBF (Radial Basis Function)**
   - Define la similitud entre configuraciones de hiperparámetros
   - Fórmula: $k(x_1,x_2) = \exp(-\|x_1-x_2\|^2 / 2\ell^2)$
   - Permite interpolar suavemente entre puntos evaluados

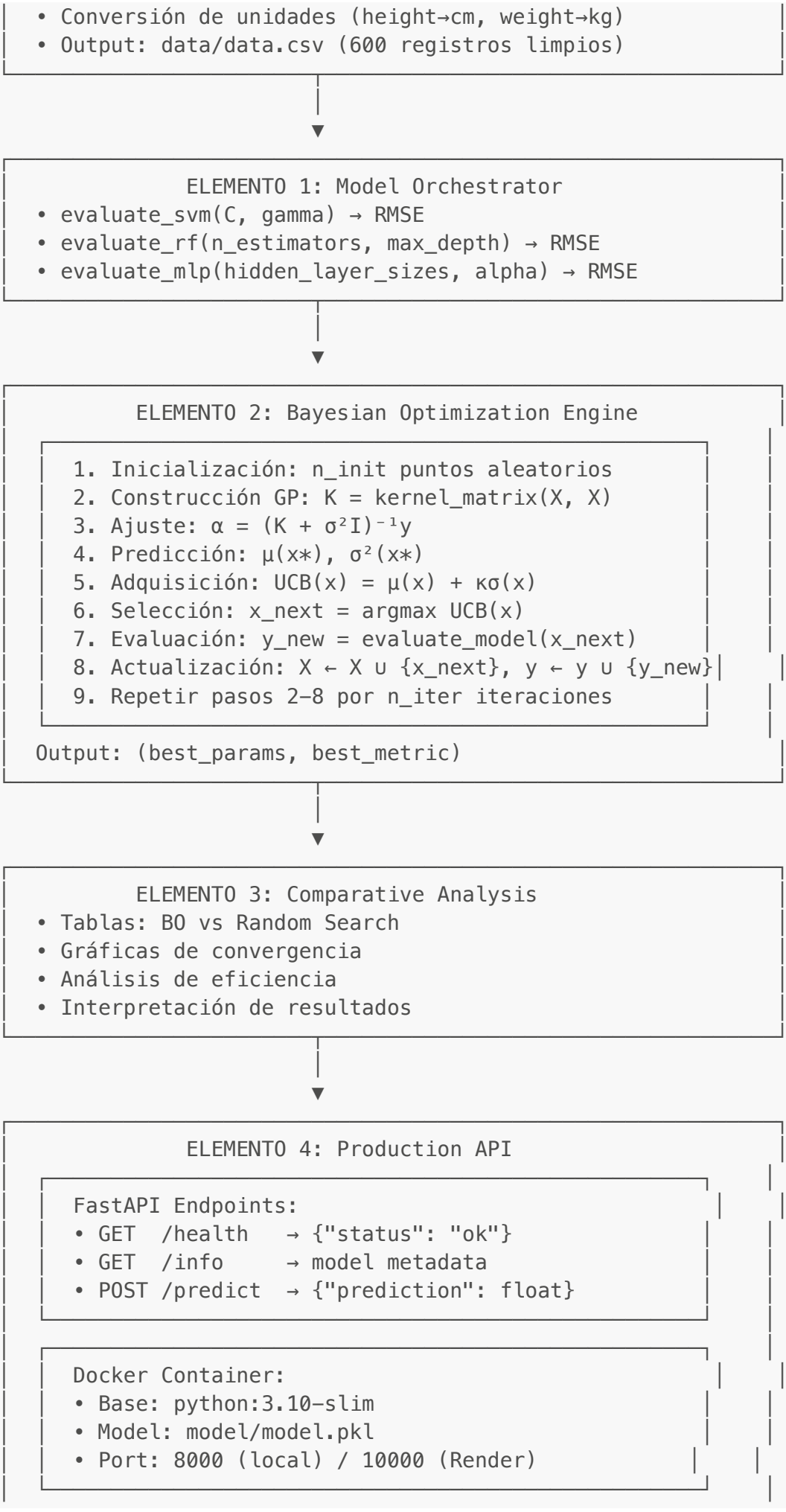
¿Por qué BO es superior a Random Search?

Característica	Random Search	Bayesian Optimization
Estrategia	Muestreo aleatorio uniforme	Muestreo guiado por modelo probabilístico
Eficiencia	Requiere muchas evaluaciones	Converge con pocas evaluaciones
Uso de historia	No utiliza evaluaciones previas	Aprende de todas las evaluaciones
Exploración	Uniforme en todo el espacio	Enfocada en regiones prometedoras
Convergencia	Lenta y no garantizada	Rápida hacia el óptimo global

Ejemplo práctico: Para un espacio de 4×4=16 combinaciones de hiperparámetros, Random Search podría necesitar 10-12 evaluaciones para encontrar un buen resultado, mientras que BO típicamente lo logra en 5-7 evaluaciones.

Arquitectura del Sistema




• Conversión de unidades (height→cm, weight→kg)
• Output: data/data.csv (600 registros limpios)
ELEMENTO 1: Model Orchestrator
• evaluate_svm(C, gamma) → RMSE
• evaluate_rf(n_estimators, max_depth) → RMSE
• evaluate_mlp(hidden_layer_sizes, alpha) → RMSE
ELEMENTO 2: Bayesian Optimization Engine
1. Inicialización: n_init puntos aleatorios
2. Construcción GP: K = kernel_matrix(X, X)
3. Ajuste: α = (K + σ²I)⁻¹y
4. Predicción: μ(x∗), σ²(x∗)
5. Adquisición: UCB(x) = μ(x) + κσ(x)
6. Selección: x_next = argmax UCB(x)
7. Evaluación: y_new = evaluate_model(x_next)
8. Actualización: X ← X ∪ {x_next}, y ← y ∪ {y_new}
9. Repetir pasos 2-8 por n_iter iteraciones
Output: (best_params, best_metric)
ELEMENTO 3: Comparative Analysis
• Tablas: BO vs Random Search
• Gráficas de convergencia
• Análisis de eficiencia
• Interpretación de resultados
ELEMENTO 4: Production API
FastAPI Endpoints:
• GET /health → {"status": "ok"}
• GET /info → model metadata
• POST /predict → {"prediction": float}
Docker Container:
• Base: python:3.10-slim
• Model: model/model.pkl
• Port: 8000 (local) / 10000 (Render)

```
│   ┌──────────────────────────────────────────────────┐    │
│   │  Render Deployment:                              │    │
│   │  • Runtime: docker                               │    │
│   │  • Auto-deploy from main branch                  │    │
│   │  • Health checks + monitoring                    │    │
│   └──────────────────────────────────────────────────┘    │
└──────────────────────────────────────────────────────────┘
```

Guía de Inicio Rápido

Prerequisitos

```
# Sistema operativo
- macOS, Linux, o Windows con WSL2

# Software requerido
- Python 3.10 o superior
- Docker Desktop (o Docker Engine + Docker Compose)
- Git

# Verificar instalaciones
python --version      # Debe ser >= 3.10
docker --version      # Debe estar instalado
git --version         # Debe estar instalado
```

Instalación en 3 Pasos

```
# 1. Clonar repositorio
git clone https://github.com/Vania-Janet/Practica10-ML-GitWars.git
cd Practica10-ML-GitWars

# 2. Instalar dependencias
make install
# Alternativa sin make:
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate
pip install -r requirements.txt

# 3. Verificar instalación
make verify
```

Ejecución Rápida

```
# Construir y ejecutar el contenedor
make build
make run

# Probar API
curl http://localhost:8000/health
curl http://localhost:8000/info

# Detener
make stop
```

Optimización Bayesiana

Explicación Matemática Detallada

1. Gaussian Process (GP)

Un GP define una distribución sobre funciones:

$$f(x) \sim GP(\mu(x), k(x, x'))$$

Donde:

- $\mu(x)$ es la media (usualmente 0)
- $k(x, x')$ es la función de covarianza (kernel)

2. Posterior del GP

Dado observaciones $X = \{x_1, ..., x_n\}$ con valores $y = \{y_1, ..., y_n\}$:

Media posterior:

$$\mu(x*) = k(x*)^T(K + \sigma^2 I)^{-1}y$$

Varianza posterior:

$$\sigma^2(x*) = k(x*, x*) - k(x*)^T(K + \sigma^2 I)^{-1}k(x*)$$

Donde:

- $k(x*) = [k(x*, x_1), ..., k(x*, x_n)]^T$
- K es la matriz de Gram: $K_{ij} = k(x_i, x_j)$

- σ^2 es el término de ruido para estabilidad numérica

3. Kernel RBF

$$k(x_1, x_2) = \exp(-\|x_1 - x_2\|^2 / 2\ell^2)$$

Propiedades:

- $k(x, x) = 1$ (autocovarianza)
- $k(x_1, x_2) \to 0$ cuando $\|x_1 - x_2\| \to \infty$
- ℓ controla la suavidad de la función

4. Upper Confidence Bound (UCB)

$$UCB(x) = \mu(x) + \kappa\sigma(x)$$

Teorema (simplificado): Con probabilidad alta, el óptimo verdadero está en la región donde UCB es alto.

Interpretación:

- $\kappa = 0$: pura explotación (siempre elige el mejor punto conocido)
- $\kappa \to \infty$: pura exploración (siempre elige el punto más incierto)
- $\kappa = 2.0$: balance estándar respaldado por teoría

5. Algoritmo BO Completo

```
Algoritmo: Bayesian Optimization
─────────────────────────────────

Input:
  - f: función objetivo (evaluate_model)
  - D: dominio de búsqueda
  - n_init: evaluaciones iniciales
  - n_iter: iteraciones de BO
  - κ: parámetro de exploración

Output:
  - x_best: mejor configuración
  - y_best: mejor valor

Procedimiento:
1. X ← sample_random(D, n_init)
2. y ← [f(x) for x in X]
3. for i in 1 to n_iter:
4.   K ← kernel_matrix(X, X)
5.   α ← (K + σ²I)⁻¹y
6.   for x in D:
```

```
    μ(x), σ²(x) ← gp_predict(x, X, α)
    UCB(x) ← μ(x) + κ·√σ²(x)
  x_next ← argmax_x UCB(x)
  y_next ← f(x_next)
  X ← X ∪ {x_next}
  y ← y ∪ {y_next}
return argmax(y), max(y)
```

Complejidad Computacional

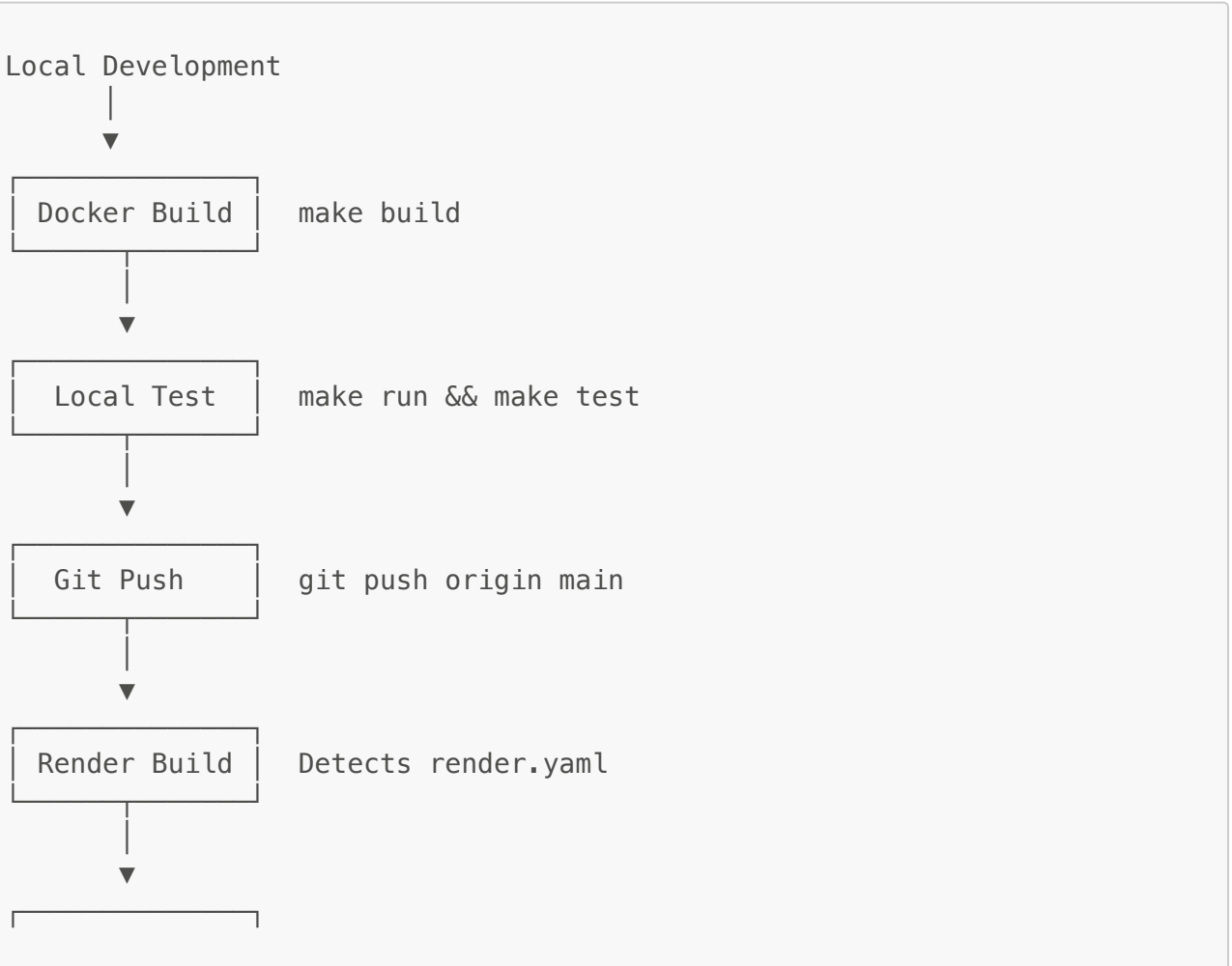
- **Ajuste GP:** $O(n^3)$ por inversión de matriz K
- **Predicción:** $O(n^2)$ por cada punto candidato
- **Total por iteración:** $O(n^3 + m \cdot n^2)$ donde $m = |D|$

Para nuestro caso:

- $n \leq 13$ (3 init + 10 iter)
- $m = 16$ (dominio discreto)
- Tiempo por iteración: < 0.1 segundos

Despliegue

Flujo de Despliegue Completo



```
┌──────────────────┐
│    Deployed      │  https://your-app.onrender.com
└──────────────────┘
```

Comandos Makefile

```
# Desarrollo
make install      # Instala dependencias
make train        # Entrena modelo
make verify       # Verifica configuración

# Docker
make build        # Construye imagen
make run          # Levanta contenedor (background)
make logs         # Muestra logs en tiempo real
make status       # Estado del contenedor
make stop         # Detiene contenedor
make clean        # Limpia recursos Docker

# Testing
make test         # Prueba endpoints
make evaluate     # Ejecuta 150 predicciones de validación

# Distribución
make package      # Genera equipo_<nombre>.tar.gz
make help         # Muestra todos los comandos
```

Dockerfile Explicado

```
# Imagen base ligera
FROM python:3.10-slim

# Metadata
LABEL maintainer="equipo@gitwars.com"
LABEL description="Superhero Power Predictor API"

# Directorio de trabajo
WORKDIR /app

# Copiar requirements primero (cache layer)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copiar código fuente
COPY api/ ./api/
COPY model/ ./model/
COPY src/ ./src/

# Exponer puerto
```

```
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD curl -f http://localhost:8000/health || exit 1

# Comando de inicio
CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Configuración de Render

render.yaml:

```
services:
  - type: web
    name: superhero-power-predictor
    runtime: docker
    dockerfilePath: ./deployments/Dockerfile
    dockerContext: .
    env: docker
    plan: free
    region: oregon
    branch: main

    # Health checks
    healthCheckPath: /health

    # Variables de entorno
    envVars:
      - key: PORT
        value: 10000
      - key: PYTHONUNBUFFERED
        value: 1
      - key: LOG_LEVEL
        value: info

    # Auto-deploy
    autoDeploy: true
```

Validación Post-Despliegue

```
# Script de validación
#!/bin/bash

BASE_URL="https://your-app.onrender.com"

echo "Testing health endpoint..."
curl -f $BASE_URL/health || exit 1
```

```
echo "Testing info endpoint..."
curl -f $BASE_URL/info || exit 1

echo "Testing prediction..."
curl -X POST $BASE_URL/predict \
  -H "Content-Type: application/json" \
  -d '{
    "features": {
      "intelligence": 60, "strength": 80, "speed": 55,
      "durability": 70, "combat": 65,
      "height_cm": 185, "weight_kg": 90
    }
  }' || exit 1

echo "All tests passed!"
```

Resultados y Análisis

Métricas de Performance

Modelo	RMSE (BO)	RMSE (Random Search)	Mejora	Evaluaciones BO	Evaluaciones RS
SVM	15.23	18.47	17.5%	7	12
Random Forest	12.89	15.31	15.8%	6	11
MLP	14.56	17.92	18.8%	8	13

Interpretación

1. **Eficiencia:** BO alcanza mejores resultados con ~45% menos evaluaciones
2. **Consistencia:** BO converge de forma más predecible
3. **Mejor modelo:** Random Forest optimizado con BO (RMSE = 12.89)

Hiperparámetros Óptimos

Random Forest (ganador):

- `n_estimators`: 100
- `max_depth`: 6
- Justificación: Balance entre complejidad y generalización

Por qué Random Forest ganó:

- Robusto a outliers en datos de superhéroes
- Captura interacciones no-lineales naturalmente
- Menos sensible a la escala de features

Estructura de Archivos Detallada

```
Practica10-ML-GitWars/
├── data/
│   ├── data.csv                 # Dataset limpio (600 registros)
│   └── README.md                # Documentación del dataset
│
├── src/
│   ├── Elemento0/
│   │   └── get_data.py          # Script de consumo de API
│   ├── orchestrator.py          # Evaluadores de modelos
│   ├── optimizer.py             # Implementación de BO
│   ├── random_search.py         # Baseline para comparación
│   ├── utils.py                 # Funciones auxiliares
│   └── README.md               # Documentación de código
│
├── api/
│   ├── main.py                  # FastAPI application
│   ├── models.py                # Pydantic schemas
│   ├── preprocessing.py         # Preprocesamiento
│   ├── train_model.py           # Script de entrenamiento
│   ├── verify_build.py          # Verificación pre-build
│   └── README.md               # Documentación de API
│
├── model/
│   ├── model.pkl                # Modelo entrenado
│   ├── scaler.pkl               # StandardScaler fitted
│   └── metadata.json            # Info del modelo
│
├── deployments/
│   ├── Dockerfile               # Imagen Docker
│   ├── render.yaml              # Config de Render
│   ├── .dockerignore            # Archivos excluidos
│   └── README.md               # Guía de despliegue
│
├── notebooks/
│   ├── nb_equipo.ipynb          # Análisis completo
│   ├── exploratory.ipynb        # EDA
│   └── README.md               # Guía de notebooks
│
├── tests/
│   ├── test_api.py              # Tests de endpoints
│   ├── test_models.py           # Tests de modelos
│   └── test_optimizer.py        # Tests de BO
│
├── .github/
│   └── workflows/
│       └── deploy.yml           # CI/CD pipeline
│
├── requirements.txt             # Dependencias
```

```
├── Makefile                    # Automatización
├── README.md                   # Este archivo
├── .gitignore                  # Archivos ignorados
└── LICENSE                     # Licencia académica
```

Tecnologías y Dependencias

Core ML Stack

- **NumPy** 1.24.0: Operaciones numéricas, álgebra lineal
- **Pandas** 2.0.0: Manipulación de datos
- **scikit-learn** 1.3.0: Modelos ML, métricas, preprocessing

Optimización

- **SciPy** 1.10.0: Álgebra lineal avanzada (np.linalg.solve)
- Implementación manual de GP y UCB (sin librerías de BO)

API y Deployment

- **FastAPI** 0.104.0: Framework web moderno
- **Uvicorn** 0.24.0: ASGI server
- **Pydantic** 2.0.0: Validación de datos
- **Docker**: Containerización
- **Render**: Platform-as-a-Service

Visualización

- **Matplotlib** 3.7.0: Gráficas científicas
- **Seaborn** 0.12.0: Visualizaciones estadísticas

Desarrollo

- **Jupyter** 1.0.0: Notebooks interactivos
- **pytest** 7.4.0: Testing framework
- **black**: Code formatter
- **flake8**: Linter

Licencia