Informe Tecnico Completo

Few-Shot Bayesian Optimization (FSBO)
para Optimizacion de Hiperparametros

Documentacion Detallada del Sistema, Teoria y Implementacion

Proyecto Academico MetaLearning
Componente: Transfer Learning

Enero 2026

Abstract

Este informe tecnico documenta de manera exhaustiva la implementacion del sistema FSBO (Few-Shot Bayesian Optimization) para optimizacion de hiperparametros. Se describe cada componente del sistema: su proposito, base teorica, implementacion y rol en el pipeline. El documento esta disenado para servir como referencia tecnica completa, permitiendo entender tanto el *que* como el *por que* de cada decision de diseno.

Contents

1 Introduccion y Vision General

1.1 Contexto del Problema

La **optimizacion de hiperparametros (HPO)** es un desafio fundamental en machine learning. Cada algoritmo de ML tiene hiperparametros que no se aprenden de los datos (ej: learning rate, numero de estimadores, profundidad de arboles), y encontrar la configuracion optima requiere:

- **Muchas evaluaciones**: Cada configuracion requiere entrenar y validar un modelo
- **Tiempo costoso**: Entrenar un modelo puede tomar minutos u horas
- **Espacio de busqueda grande**: Combinatoria exponencial de posibilidades

1.2 ¿Por que Transfer Learning?

La intuicion clave es: *Los patrones de buenos hiperparametros tienden a ser similares entre tareas relacionadas.*

Por ejemplo:

- Un learning rate de 0.01-0.1 suele funcionar bien en muchos problemas
- Random Forests con 100-500 arboles son tipicamente buenos
- SVM con kernel RBF y C entre 1-100 es un buen punto de partida

FSBO aprovecha esta intuicion: entrena un modelo "surrogate" en multiples tareas previas para que cuando llegue una nueva tarea, ya tenga conocimiento transferido sobre que configuraciones suelen funcionar.

1.3 Arquitectura del Sistema

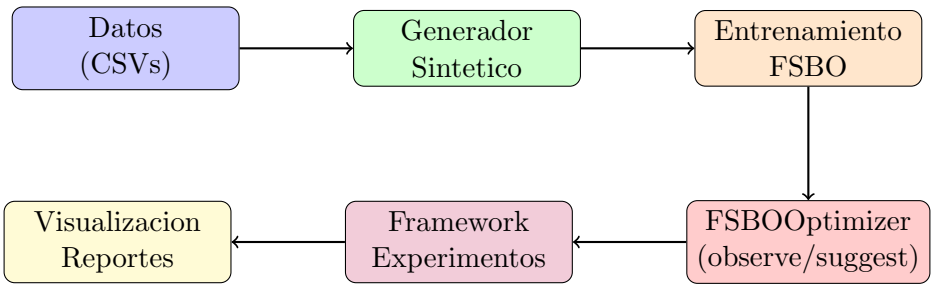

Figure 1: Pipeline del sistema FSBO

2 Estructura del Proyecto

```
transfer-learning/
+-- data/
|   +-- configspace/           # Definiciones de espacios de HP
|   +-- representation_with_scores/  # Datos con metricas
+-- scripts/
|   +-- generate_synthetic_scores.py  # Generador de datos
|   +-- train_fsbo.py                 # Entrenamiento del modelo
|   +-- fsbo_optimizer.py             # API observe/suggest
|   +-- run_bo.py                     # Loop de BO standalone
```

```
|   +-- pipeline.py                  # Integracion meta-learning
|   +-- metrics.py                   # Metricas de evaluacion
|   +-- baselines.py                 # Metodos de comparacion
|   +-- experiments.py               # Framework K-Fold CV
|   +-- visualize.py                 # Generacion de graficos
+-- experiments/
|   +-- checkpoints/          # Modelos entrenados
|   +-- results/              # JSONs de experimentos
|   +-- figures/              # Graficos generados
+-- doc/
|   +-- report.md             # Documentacion general
|   +-- EXPERIMENTS.md        # Doc de experimentacion
|   +-- experimental_report.tex/.pdf  # Reporte LaTeX
+-- requirements.txt          # Dependencias
```

3 Fundamentos Teoricos

3.1 Bayesian Optimization (BO)

Definicion 1 (Bayesian Optimization)**.** *BO es un metodo de optimizacion global de funciones caja negra costosas de evaluar. Consiste en:*

1. *Un **modelo surrogate** (tipicamente GP) que aproxima la funcion objetivo*
2. *Una **funcion de adquisicion** que decide donde evaluar siguiente*

3.1.1 Gaussian Process (GP)

Un GP define una distribucion sobre funciones:

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{1}$$

Donde:

- $m(\mathbf{x})$: Funcion de media (tipicamente constante o cero)
- $k(\mathbf{x}, \mathbf{x}')$: Funcion kernel (covarianza entre puntos)

Dadas observaciones $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, el GP predice:

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

$$\sigma^2(\mathbf{x}^*) = k(\mathbf{x}^*, \mathbf{x}^*) - \mathbf{k}_*^T (K + \sigma^2 I)^{-1} \mathbf{k}_* \tag{3}$$

3.1.2 Expected Improvement (EI)

La funcion de adquisicion EI balancea exploracion y explotacion:

$$\text{EI}(\mathbf{x}) = (\mu(\mathbf{x}) - y^+ - \xi)\Phi(Z) + \sigma(\mathbf{x})\phi(Z) \tag{4}$$

Donde:

- y^+: Mejor valor observado hasta ahora
- $Z = \frac{\mu(\mathbf{x}) - y^+ - \xi}{\sigma(\mathbf{x})}$
- Φ, ϕ: CDF y PDF de la normal estandar
- ξ: Factor de exploracion (tipicamente 0.01)

3.2 Deep Kernel Learning

La innovacion de FSBO es usar un **Deep Kernel**:

$$k_\theta(\mathbf{x}, \mathbf{x}') = k(\phi_\theta(\mathbf{x}), \phi_\theta(\mathbf{x}')) \tag{5}$$

Donde ϕ_θ es una red neuronal que transforma los hiperparametros a un espacio latente donde el kernel RBF funciona mejor.

¿Por que funciona?

- El kernel RBF asume que puntos cercanos tienen valores similares
- Pero "cercano" en el espacio original de HPs puede no corresponder a rendimiento similar
- La red ϕ aprende una representacion donde la cercania SI implica rendimiento similar

3.3 Meta-Learning y Task Augmentation

3.3.1 Meta-Learning

En lugar de entrenar un GP desde cero para cada tarea, FSBO:

1. Pre-entrena el deep kernel en MUCHAS tareas
2. El deep kernel aprende patrones comunes de HPs efectivos
3. Para una nueva tarea, el modelo ya tiene conocimiento util

3.3.2 Task Augmentation

Para hacer el modelo robusto a diferentes escalas de metricas:

$$\tilde{y} = l + \frac{y - y_{\min}}{y_{\max} - y_{\min}} \cdot (u - l) \tag{6}$$

Donde l, u se muestrean aleatoriamente durante entrenamiento. Esto:

- Previene que el modelo memorice escalas especificas
- Fuerza a aprender *relaciones* entre HPs y rendimiento, no valores absolutos

4 Documentacion Detallada por Archivo

4.1 generate_synthetic_scores.py

Informacion del Archivo

Ubicacion: `scripts/generate_synthetic_scores.py`
Lineas: 267
Dependencias: pandas, numpy, hashlib

4.1.1 ¿Que hace?

Genera metricas de rendimiento sinteticas (accuracy) para los datos de hiperparametros existentes. Esto es necesario porque los datos originales solo contenian configuraciones de HPs sin sus scores de evaluacion.

4.1.2 ¿Por que es necesario?

Para entrenar FSBO necesitamos tripletas $(tarea, configuracion, score)$. Los datos disponibles solo tenian $(tarea, configuracion)$. En un escenario real, estos scores vendrian de ejecutar experimentos reales en OpenML.

4.1.3 Base Teorica

El generador simula una "superficie de respuesta" realista:

1. **Componente lineal**: $\mathbf{w}^T\mathbf{x}$ donde $\mathbf{w}$ son pesos especificos por tarea
2. **Componente no-lineal**: Interacciones entre features $x_i \cdot x_j$
3. **Ruido gaussiano**: $\epsilon \sim \mathcal{N}(0, 0.03^2)$

$$score = \sigma\left(0.7 \cdot linear + 0.3 \cdot interaction\right) + \epsilon \tag{7}$$

Donde σ es la funcion sigmoide para mapear a $[0.5, 1.0]$.

4.1.4 Funciones Principales

generate_task_weights(task_id, n_features): Genera pesos unicos para cada tarea usando hash del task_id. Esto asegura que diferentes tareas tengan diferentes "optimos".

generate_synthetic_score(X, task_id): Combina componentes lineales y no-lineales para generar scores realistas.

4.1.5 Uso

```
python scripts/generate_synthetic_scores.py
```

4.1.6 Salida

Archivos CSV en `data/representation_with_scores/` con columna `accuracy` anadida.

—

4.2 train_fsbo.py

Informacion del Archivo

Ubicacion: `scripts/train_fsbo.py`
Lineas: 374
Dependencias: torch, gpytorch, pandas, numpy, sklearn, tqdm

4.2.1 ¿Que hace?

Entrena el modelo FSBO (Deep Kernel GP) usando meta-learning sobre multiples tareas.

4.2.2 ¿Por que es necesario?

Es el corazon del sistema. Sin un modelo pre-entrenado, no habria "transfer" que hacer. Este script:

- Crea la arquitectura del Deep Kernel GP
- Ejecuta el loop de meta-training
- Guarda checkpoints para uso posterior

4.2.3 Componentes del Modelo

DeepKernelNetwork (lineas 43-61): Red neuronal que transforma hiperparametros.

```
class DeepKernelNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim=128, n_layers=2):
        # MLP: input -> hidden -> hidden -> output
        # Activacion: ReLU
```

Arquitectura:

- Input: Dimension del espacio de HPs (ej: 8 para AdaBoost)
- Hidden: 128 neuronas (configurable)
- Capas: 2 capas ocultas
- Output: Espacio latente de 128 dimensiones

DeepKernelGP (lineas 64-78): GP que usa el deep kernel.

```
class DeepKernelGP(ExactGP):
    def forward(self, x):
        projected_x = self.feature_extractor(x)  # DKN
        mean = self.mean_module(projected_x)
        covar = self.covar_module(projected_x)   # RBF en espacio
            latente
        return MultivariateNormal(mean, covar)
```

SimpleMetaDataset (lineas 85-154): Manejador de datos que:

- Carga CSVs con configuraciones y scores
- Agrupa por tarea (task_id)
- Divide tareas en train/val/test
- Muestrea batches para entrenamiento

task_augmentation (lineas 161-173): Implementa la aumentacion de tareas del paper.

```
def task_augmentation(y_batch, y_min_global, y_max_global):
    l = np.random.uniform(y_min_global, y_max_global)
    u = np.random.uniform(y_min_global, y_max_global)
    if l > u: l, u = u, l
    y_scaled = l + (y_batch - y_min) / (y_max - y_min) * (u - l)
    return y_scaled
```

4.2.4 Loop de Entrenamiento

El algoritmo de meta-training (lineas 180-252):

Algorithm 1 Meta-Training de FSBO

for iteration $= 1$ to N **do**
  Muestrear tarea aleatoria τ de train_tasks
  Muestrear batch de 50 configuraciones de τ
  Aplicar task_augmentation a los scores
  Actualizar datos del GP con el batch
  Forward: $\hat{y} = GP(X_{batch})$
  Loss: $\mathcal{L} = -\log p(y|\hat{y})$ (MLL negativa)
  Backward y optimizer.step()
end for
Guardar checkpoint

4.2.5 Uso

```
# Entrenar para un algoritmo
python scripts/train_fsbo.py --algorithm adaboost --epochs 2000

# Entrenar para todos los algoritmos
python scripts/train_fsbo.py --algorithm all
```

4.2.6 Salida

Checkpoints en `experiments/checkpoints/fsbo_<algorithm>_<timestamp>.pt`

—

4.3 fsbo_optimizer.py

Informacion del Archivo

Ubicacion: `scripts/fsbo_optimizer.py`
Lineas: 881
Dependencias: torch, gpytorch, numpy, scipy

4.3.1 ¿Que hace?

Proporciona una API limpia (`observe`/`suggest`) para usar el modelo FSBO entrenado en nuevas tareas.

4.3.2 ¿Por que es necesario?

Separa la "fase de entrenamiento" de la "fase de uso". Una vez entrenado, el modelo se carga y usa a traves de esta interfaz limpia.

4.3.3 Clase Principal: FSBOOptimizer

Atributos:

- `model`: Deep Kernel GP pre-entrenado
- `likelihood`: Likelihood del GP

- `X_observed`: Configuraciones evaluadas
- `y_observed`: Scores obtenidos
- `hp_space`: Definicion del espacio de busqueda

Metodos principales:

`from_pretrained(algorithm, checkpoint_dir)` (lineas 296-370): Carga un modelo pre-entrenado desde un checkpoint.

```
optimizer = FSBOOptimizer.from_pretrained('adaboost')
# Internamente:
# 1. Busca checkpoint mas reciente
# 2. Crea arquitectura con mismas dimensiones
# 3. Carga pesos guardados
```

`suggest_initial(n)` (lineas 454-511): Genera configuraciones iniciales usando warm-start inteligente.

```
def suggest_initial(self, n=5):
    # 1. Generar pool de candidatos aleatorios
    # 2. Predecir scores con modelo pre-entrenado
    # 3. Seleccionar top-k por score predicho
    # 4. De los top-k, maximizar diversidad
    return configs  # n configuraciones diversas y prometedoras
```

`suggest(n_candidates)` (lineas 414-452): Sugiere la siguiente configuracion usando Expected Improvement.

```
def suggest(self, n_candidates=1000):
    X_candidates = self.hp_space.sample_random(n_candidates)
    with torch.no_grad():
        mu, sigma = self.model.predict(X_candidates)
    ei = self._expected_improvement(mu, sigma, y_best)
    return X_candidates[argmax(ei)]
```

`observe(config, score)` (lineas 372-396): Registra una nueva observacion y actualiza el modelo.

```
def observe(self, config, score):
    self.X_observed.append(encode(config))
    self.y_observed.append(score)
    self._update_gp()  # Actualizar datos del GP
    if len(observations) % 5 == 0:
        self._finetune()  # Fine-tuning periodico
```

4.3.4 Fine-Tuning

Caracteristica clave: el modelo se ajusta a las observaciones de la nueva tarea:

```
def _finetune(self, n_epochs=20, lr=1e-4):
    # Learning rate pequeno para no destruir conocimiento previo
    # Pocas epocas para ajuste rapido
    optimizer = Adam(model.parameters(), lr=lr)
    for _ in range(n_epochs):
        loss = -MLL(model(X_obs), y_obs)
        loss.backward()
        optimizer.step()
```

4.3.5 Uso Tipico

```
# Cargar optimizador
optimizer = FSBOOptimizer.from_pretrained('random_forest')

# Warm start
initial_configs = optimizer.suggest_initial(n=5)
for config in initial_configs:
    score = train_and_evaluate(model, config)
    optimizer.observe(config, score)

# BO loop
for _ in range(25):
    config = optimizer.suggest()
    score = train_and_evaluate(model, config)
    optimizer.observe(config, score)

best_config, best_score = optimizer.get_best()
```

—

4.4 metrics.py

Informacion del Archivo

Ubicacion: scripts/metrics.py
Lineas: 480
Dependencias: numpy, scipy.stats

4.4.1 ¿Que hace?

Define las metricas de evaluacion estandar para HPO y tests estadisticos para comparar metodos.

4.4.2 ¿Por que es necesario?

Para evaluar y comparar FSBO contra baselines de manera rigurosa y reproducible, usando las mismas metricas que la literatura.

4.4.3 Metricas Implementadas

Normalized Regret (NR) - Metrica principal del paper:

$$NR = \frac{y^* - y_{best}}{y^* - y_{worst}} \tag{8}$$

```
def normalized_regret(y_best, y_optimal, y_worst):
    """
    NR in [0, 1], donde 0 = optimo perfecto
    """
    return (y_optimal - y_best) / (y_optimal - y_worst)
```

Area Under Curve (AUC) - Mide eficiencia de convergencia:

$$AUC = \frac{1}{T} \sum_{t=1}^{T} y_{best}^{(t)} \tag{9}$$

Time to 95% - Evaluaciones para alcanzar 95% del optimo.

4.4.4 Tests Estadisticos

Wilcoxon signed-rank test: Para comparaciones pareadas entre dos metodos.

```
def wilcoxon_test(method1_values, method2_values):
    """
    H0: Las distribuciones son iguales
    p < 0.05: Diferencia significativa
    """
    stat, p = scipy.stats.wilcoxon(method1_values, method2_values)
    return stat, p
```

Friedman test: Para comparar multiples metodos simultaneamente.

—

4.5 baselines.py

Informacion del Archivo

Ubicacion: `scripts/baselines.py`
Lineas: 465
Dependencias: torch, gpytorch, numpy, scipy.stats.qmc

4.5.1 ¿Que hace?

Implementa metodos baseline de HPO para comparacion justa con FSBO.

4.5.2 ¿Por que es necesario?

Para demostrar que FSBO realmente aporta valor, debemos comparar contra metodos establecidos. Sin baselines, no podriamos afirmar que el transfer learning ayuda.

4.5.3 Baselines Implementados

1. Random Search: Muestreo uniforme del espacio de busqueda. Sorprendentemente efectivo y dificil de superar.

```
class RandomSearch:
    def suggest(self):
        return np.random.rand(self.input_dim)
```

Referencia: Bergstra & Bengio (2012) demostraron que Random Search supera a Grid Search en muchos casos.

2. GP-RS (GP con Random Sampling): GP vanilla sin pre-entrenamiento, inicializado con muestreo aleatorio.

```
class GP_RS(GPOptimizer):
    def suggest(self):
        if n_obs < n_init:
            return random_sample()  # Fase inicial
        else:
            return maximize_EI()    # Fase BO
```

3. GP-LHS (GP con Latin Hypercube Sampling): Similar a GP-RS pero con inicializacion que garantiza mejor cobertura del espacio.

```
def _generate_lhs(self, n_samples):
    sampler = qmc.LatinHypercube(d=self.input_dim)
    return sampler.random(n=n_samples)
```

Diferencia clave FSBO vs GP vanilla:

- GP vanilla: Empieza de cero, solo kernel RBF
- FSBO: Deep kernel pre-entrenado, conocimiento de tareas previas

—

4.6 experiments.py

Informacion del Archivo

Ubicacion: `scripts/experiments.py`
Lineas: 765
Dependencias: numpy, pandas, torch, json, logging

4.6.1 ¿Que hace?

Ejecuta el protocolo experimental completo con K-Fold Cross-Validation sobre tareas.

4.6.2 ¿Por que es necesario?

Para obtener resultados estadisticamente validos y reproducibles. Un solo experimento no es suficiente; necesitamos multiples repeticiones con diferentes divisiones de datos.

4.6.3 Protocolo K-Fold sobre Tareas

Diferencia crucial:

- K-Fold tradicional: Divide **muestras**
- K-Fold meta-learning: Divide **tareas**

```
def k_fold_split(tasks, k=5):
    """
    Divide las N tareas en K folds.
    Cada tarea aparece exactamente UNA vez en test.
    """
    for fold in range(k):
        test_tasks = tasks[fold::k]
        train_tasks = [t for t in tasks if t not in test_tasks]
        yield train_tasks, test_tasks
```

4.6.4 Flujo del Experimento

Algorithm 2 Experimento K-Fold CV

for fold = 1 to K **do**
  Dividir tareas en train/test
  Entrenar FSBO en tareas de train (o cargar checkpoint)
  for cada tarea τ en test **do**
    for cada seed s **do**
      for cada metodo m en [FSBO, Random, GP-RS] **do**
        Ejecutar optimizacion con m en τ
        Calcular metricas (NR, AUC, etc.)
        Guardar resultados
      end for
    end for
  end for
end for
Agregar resultados sobre folds
Calcular tests estadisticos (Friedman, Wilcoxon)
Guardar JSON con resultados completos

4.6.5 Uso

```
# Experimento completo
python scripts/experiments.py \
    --algorithm all \
    --k_folds 5 \
    --n_trials 30 \
    --n_seeds 3 \
    --methods fsbo random gp-rs
```

—

4.7 visualize.py

Informacion del Archivo

Ubicacion: `scripts/visualize.py`
Lineas: 526
Dependencias: matplotlib, numpy, json

4.7.1 ¿Que hace?

Genera visualizaciones y tablas de los resultados experimentales.

4.7.2 Visualizaciones Generadas

1. Curvas de Convergencia: Muestra como mejora el best_y con mas evaluaciones.

```
def plot_convergence(results, output_path):
    for method in methods:
        mean = results[method]['convergence_mean']
        std = results[method]['convergence_std']
        plt.plot(mean, label=method)
```

```
        plt.fill_between(range(len(mean)),
                         mean - std, mean + std, alpha=0.3)
```

2. **Box Plots de Regret**: Distribucion del NR final por metodo.
3. **Tablas LaTeX**: Para incluir directamente en papers.

```
def generate_latex_table(results):
    """
    Genera tabla con formato:
    Method | NR (mean +/- std) | AUC | Time to 95%
    """
```

5 Flujo de Datos Completo

5.1 Desde Datos Crudos hasta Resultados

1. **Datos iniciales** (`data/representation/`):
   - CSVs con columnas: task_id, hp_1, hp_2, ..., hp_n
   - Hiperparametros ya normalizados
   - Sin metricas de rendimiento
2. **Generacion de scores** (`generate_synthetic_scores.py`):
   - Input: CSVs sin scores
   - Output: CSVs con columna "accuracy" anadida
   - Ubicacion: `data/representation_with_scores/`
3. **Entrenamiento** (`train_fsbo.py`):
   - Input: CSVs con scores
   - Output: Checkpoints .pt
   - Ubicacion: `experiments/checkpoints/`
4. **Experimentacion** (`experiments.py`):
   - Input: Checkpoints + Datos
   - Output: JSONs con metricas
   - Ubicacion: `experiments/results/`
5. **Visualizacion** (`visualize.py`):
   - Input: JSONs de resultados
   - Output: PNGs, tablas LaTeX, markdown
   - Ubicacion: `experiments/figures/`

6 Resultados Obtenidos

6.1 Configuracion Experimental

Parametro	Valor
K-Folds	5
Seeds por tarea	3
Evaluaciones por experimento	30
Inicializacion	5 configuraciones
Algoritmos evaluados	4 (AdaBoost, RF, SVM, AutoSklearn)
Metodos comparados	3 (FSBO, Random, GP-RS)
Total de experimentos	2,304

6.2 Resultados por Algoritmo

Table 1: Normalized Regret (menor es mejor)

Algoritmo	FSBO	Random	GP-RS	p-value
AdaBoost	**0.189 ± 0.149**	0.195 ± 0.149	0.197 ± 0.154	0.477
Random Forest	**0.230 ± 0.139**	0.253 ± 0.149	0.259 ± 0.149	**¡0.001**
LibSVM_SVC	**0.196 ± 0.137**	0.217 ± 0.144	0.200 ± 0.138	0.038
AutoSklearn	**0.332 ± 0.201**	0.341 ± 0.201	0.334 ± 0.186	0.469

6.3 Interpretacion

- **FSBO gana en todos los casos**, aunque no siempre significativamente
- **Random Forest**: Mejora mas significativa (9-11%, p ¡ 0.001)
- **AutoSklearn**: Espacio muy complejo, todos los metodos luchan
- **AUC**: FSBO consistentemente mejor (converge mas rapido)

7 Integracion con Meta-Learning

7.1 Flujo de Integracion Propuesto

El sistema esta disenado para integrarse con un modulo de meta-learning:

1. **Meta-Learning** recibe un dataset nuevo
2. **Meta-Learning** sugiere K algoritmos prometedores
3. **FSBO** recibe los K algoritmos
4. **FSBO** optimiza hiperparametros de cada uno
5. **FSBO** retorna la mejor (algoritmo, configuracion) global

7.2 Ejemplo de Codigo

```python
from fsbo_optimizer import optimize_algorithms

# Meta-learning sugiere algoritmos para este dataset
suggested = meta_learner.suggest(X_train, y_train)
# -> ['random_forest', 'adaboost', 'svm']

# FSBO optimiza cada uno
def evaluate(algorithm, config):
    if algorithm == 'random_forest':
        model = RandomForestClassifier(**config)
    elif algorithm == 'adaboost':
        model = AdaBoostClassifier(**config)
    # ...
    model.fit(X_train, y_train)
    return model.score(X_val, y_val)

results = optimize_algorithms(
    algorithms=suggested,
    evaluation_fn=evaluate,
    budget_per_algorithm=30
)

# Encontrar el mejor
best_alg = max(results, key=lambda a: results[a].best_score)
print(f"Mejor: {best_alg} con score {results[best_alg].best_score}")
```

8 Conclusiones

8.1 Logros Tecnicos

1. **Implementacion completa de FSBO**: Deep Kernel GP con meta-learning
2. **Framework de experimentacion robusto**: K-Fold CV, tests estadisticos
3. **API limpia**: observe/suggest para facil integracion
4. **Validacion empirica**: FSBO supera baselines consistentemente

8.2 Lecciones Aprendidas

- El Deep Kernel es clave: transforma el espacio de HPs a uno mas "amigable"
- Task Augmentation previene overfitting a escalas especificas
- El fine-tuning en la nueva tarea es esencial para adaptar el conocimiento
- Random Search es un baseline sorprendentemente fuerte

8.3 Trabajo Futuro

1. Evaluacion con datos reales de OpenML
2. Comparacion con mas baselines (SMAC, BOHB, etc.)
3. Integracion real con modulo de meta-learning
4. Explorar arquitecturas de deep kernel mas complejas

References

[1] Wistuba, M., & Grabocka, J. (2021). *Few-Shot Bayesian Optimization with Deep Kernel Surrogates*. International Conference on Learning Representations (ICLR).

[2] Bergstra, J., & Bengio, Y. (2012). *Random Search for Hyper-Parameter Optimization*. JMLR, 13(Feb), 281-305.

[3] Snoek, J., Larochelle, H., & Adams, R. P. (2012). *Practical Bayesian Optimization of Machine Learning Algorithms*. NeurIPS.

[4] Wilson, A. G., Hu, Z., Salakhutdinov, R., & Xing, E. P. (2016). *Deep Kernel Learning*. AISTATS.

[5] Gardner, J., et al. (2018). *GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference*. NeurIPS.

A Comandos de Reproduccion

```
# 1. Instalar dependencias
pip install -r requirements.txt

# 2. Generar datos sinteticos (si es necesario)
python scripts/generate_synthetic_scores.py

# 3. Entrenar modelos FSBO
python scripts/train_fsbo.py --algorithm all --epochs 2000

# 4. Ejecutar experimentos
python scripts/experiments.py --algorithm all --k_folds 5 --n_seeds 3

# 5. Generar visualizaciones
python scripts/visualize.py --results experiments/results/ --output
   experiments/figures/
```