Bayesian optimization with Scikit-Optimize

Gilles Louppe
@glouppe  🌼 NYU

PyData Amsterdam 2017
Pause café?

You have an espresso machine with many buttons and knobs to tweak.

Your task is to **brew the best cup of espresso** before dying of caffeine overdose.
\[ x^* = \arg \max_x f(x) \]

- \( f \) is a black box function, with no closed form nor gradients.
- \( f \) is expensive to evaluate.
- You may only have noisy observations of \( f \).
If you do not have these constraints, do not use Bayesian optimization.
Bayesian optimisation, step-by-step

Start with observations of the objective $f$:

$$\{(x_i, f(x_i)) | i = 1, \ldots, t\}$$
Bayesian optimisation, step-by-step

Build a probabilistic model for $f$ (typically, a Gaussian process).
Bayesian optimisation, step-by-step

Optimize a cheap utility function $u$ based on the posterior distribution for sampling the next point.

$$x_{t+1} = \arg \max_x u(x)$$
Bayesian optimisation, step-by-step

Evaluate $f$ and repeat.
What is Bayesian about Bayesian optimization?

- The unknown objective is considered as a *random function* (a stochastic process) on which we place a *prior* (here defined by a Gaussian process capturing our beliefs about the function behaviour).

- Function evaluations are treated as data and used to update the prior to form the *posterior* distribution over the objective function.

\[
P(A|B) = \frac{P(B|A)P(A)}{P(B)}
\]
Scikit-Optimize

A simple library for black-box optimization with a scipy.optimize interface.

Sprouted from a (never-ending) Scikit-Learn pull request.

https://scikit-optimize.github.io

pip install scikit-optimize
Objective function

- Takes a list of values as argument.
- Returns a scalar (the lower, the better)

```python
from sklearn.datasets import load_boston
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import cross_val_score

boston = load_boston()
X, y = boston.data, boston.target

def objective(params):
    # Unpack parameters
    learning_rate = 10. ** params[0]
    max_depth = params[1]
    min_samples_split = params[2]
    max_features = params[3]

    # Build regressor
    reg = GradientBoostingRegressor(
        n_estimators=50,
        learning_rate=learning_rate,
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        max_features=max_features,
        random_state=1)

    # The lower the better
    return -np.mean(cross_val_score(reg, X, y, cv=5, n_jobs=-1,
                                      scoring="neg_mean_absolute_error"))
```
Minimize

- Takes the objective function and the bounds of the parameter space.
- That’s it!
- The tuple contains all the results (final and intermediate).

```python
from skopt import gp_minimize

space = [(-5.0, 0.0), (1, 5), (2, 100), (1, X.shape[1])]  # learning_rate, max_depth, min_samples_split, max_features

r = gp_minimize(objective, space, n_calls=50, random_state=1)

r.x
[-0.96542858133704623, 5, 9, 7]

r.fun
2.8944939733595256
```
Visualization

- Convergence plot
- Pairwise partial dependence plot of the surrogate objective.
Acquisition function

Specifies the next sample $x$.

- Upper confidence bound
  
  $$UCB(x) = \mu_{GP}(x) + \kappa \sigma_{GP}(x)$$

- Probability of improvement
  
  $$PI(x) = P(f(x) \geq f(x_t^+) + \kappa)$$

- Expected improvement
  
  $$EI(x) = \mathbb{E}[f(x) - f(x_t^+)]$$

$\kappa$ provides a knob for controlling the exploration-exploitation trade-off.
Surrogate model

The probabilistic model for $f$ is usually built as a Gaussian Process.

Scikit-Optimize supports any Scikit-Learn regressor that can also return the variance of the predictions ($\text{return_std}=\text{True}$).

- Random forests / Extra-trees
- Gradient boosting

Tree-based optimization is fast and usually better on discontinuous high-dimensional spaces.

```python
from skopt import forest_minimize
r = forest_minimize(objective, space)

r.x
[-0.89303546067450945, 5, 33, 9]

r.fun
2.8175239612455112
```
Is this better than random?
Was this actually worth it?
Random search

- If nothing works, try random search.
- Works surprisingly well, often only slightly worse than “smart” algorithms.
Wait.

That’s nice, but how do I put my coffee maker into a Python function?
Ask and tell API

- Factor out the evaluation of the objective function.
- \texttt{ask}: query next point to evaluate
  \texttt{tell}: provide function value
- Resume long optimization by pickling/unpickling the \texttt{Optimizer} object.

```python
from skopt import Optimizer
opt = Optimizer(space, gp)

# Next query point
def objective(x):
    return x**2 + 3

next_x = opt.ask()
next_x
[-0.69339327310821286, 3, 74, 3]

r = opt.tell(next_x, objective(next_x))

# Tell value at x
r = opt.run(objective, n_iter=50)

r.x
[-0.867784969421451, 5, 37, 9]

r.fun
2.736335329599823
```
Beyond coffee (or ML)

Tuning high energy physics simulators to match experimental data (e.g., 1610.08328)
Good coffee is expensive and does not come with gradients. Bayesian optimisation can help.
Summary

- Bayesian optimisation is a principled approach for optimising an expensive function $f$.

- Scikit-Optimize provides an easy-to-use set of tools for this.
We want you!
Help us improve Scikit-Optimize

https://github.com/scikit-optimize/scikit-optimize