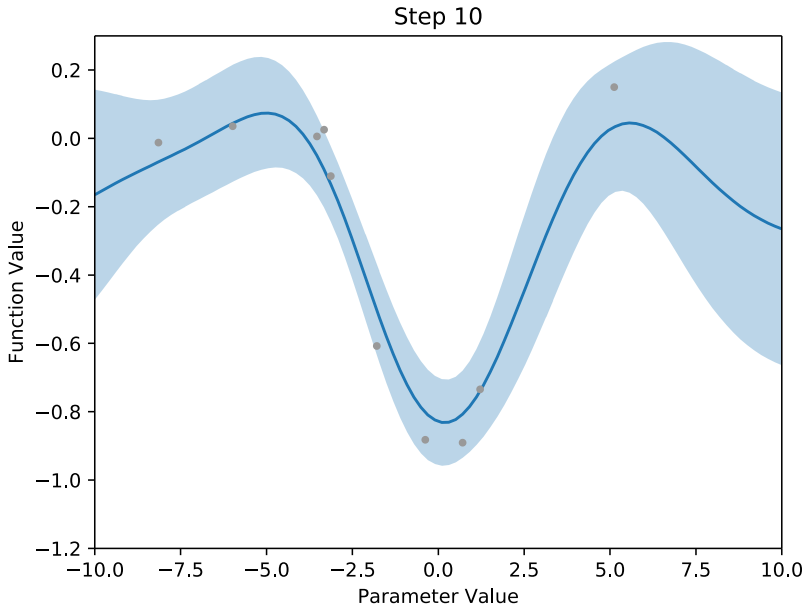
Brendan Hasz
Data Scientist at
Github
LinkedIn
Email
Resume

Bayesian Hyperparameter Optimization using Gaussian Processes

28 Mar 2019 - Tags: bayesian, prediction, and optimization

View on Github



Most machine learning models have several hyperparameters - values which can be tuned to way the learning process for that algorithms works. Changing these hyperparameters usuall different predictive performance of the algorithm. For example, we might want to find the lea which gives us the highest R^2 value. Unfortunately, one combination of settings isn't univers even for the same algorithm: the best hyperparameters for one dataset are likely to be differe best hyperparameters for another!

Manually searching for the best combination of hyperparameters for your model and dataset lot of time and effort. One automated method which works pretty well is trying out a bunch o combinations of hyperparameters and using the combination which gives the best results. H there's an even more efficient way to optimize the hyperparameters: using Bayesian optimiza

With Bayesian optimization, we use a "surrogate" model to estimate the performance of our p algorithm as a function of the hyperparameter values. This surrogate model is then used to s hyperparameter combination to try. Here we'll use a Gaussian process as the surrogate mode are other alternatives such as random forests and tree Parzen estimators.

In this post we'll first build a Python class for opimizing an(y) expensive function (in our cas validated predictive performance), and then a function which uses that class to find the optir hyperparameters for any sklearn estimator. Finally, we'll put it to use and find the optimal hyperparameters for CatBoost which allow us to best predict diabetes disease progression!

Outline

The Math
Bayesian Optimization
- The code
- Minimizing a 1D Function
- Minimizing a 2D Function

- Hyperparameter Optimization

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

- Optimizing One Hyperparameter
- Optimizing Multiple Hyperparameters
- Conclusion

First, let's import the packages we'll use, and set some settings.

```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt

from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import RobustScaler

# Plot settings
%matplotlib inline
%config InlineBackend.figure_format = 'svg'
mpl.rcParams['image.cmap'] = 'viridis'
```

We'll also be using CatBoost for prediction, so let's install and import that as well.

```
!pip install -q catboost
from catboost import CatBoostRegressor
```

A more complete version of the code we'll be working with here is available on my GitHub. W
import the two relevant components from that repository which this post shows how to build

```
!pip install git+http://github.com/brendanhasz/dsutils.git
from dsutils.optimization import GaussianProcessOptimizer
from dsutils.optimization import optimize_params
```

The Math

Bayesian optimization uses a surrogate model to estimate the function to be optimized. We'll
Gaussian process because it gives us not just an estimate of the function, but also informatio
uncertain that estimate is. We need this uncertainty information because when attempting t
best parameter combination, we want to strike a balance between exploration and exploitatic
we want to explore areas of hyperparameter-space which we haven't tried yet (i.e., where the
highly uncertain as to the performance score), but we also want to exploit areas where we're
that the score is good, because those areas are more likely to be where the *actual* best parame
combination lies.

A Gaussian process models the dependent variable $\mathbf{y}$ (in our case, the cross-validated perfor
being drawn from a N-dimensional multivariate normal distribution:

$$\mathbf{y} \sim \mathcal{N}(\mu,\ \Sigma)$$

We'll just normalize the data such that it has a mean of 0, and use $mu = 0$. The covariance m
that normal distribution is an $N \times N$ matrix defined by the independent variables x (in our
hyperparameter values):

$$\Sigma_{i,j} = K(x_i, x_j)$$

Where K is some "kernel" function. We'll use the rational quadratic kernel (though there are
different options):

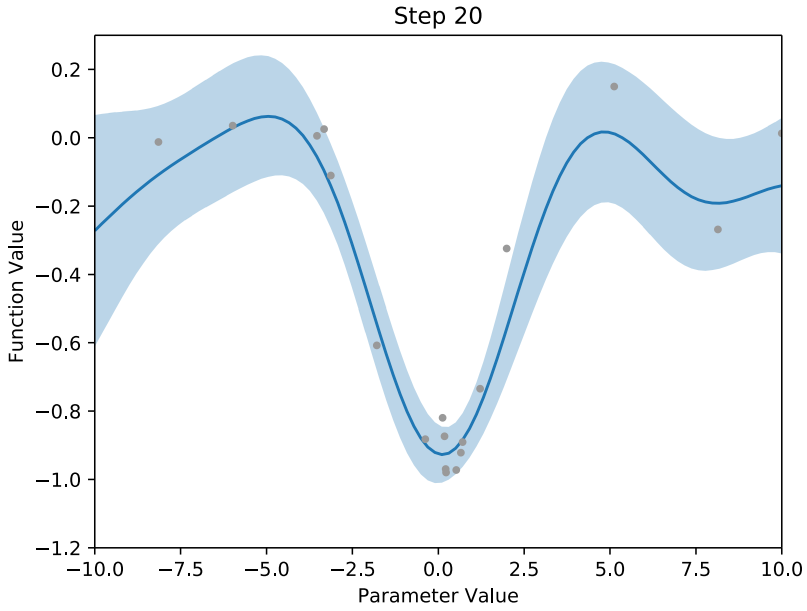
$$K(x_i, x_j) = \sigma^2 \left(1 + \frac{(x_i - x_j)^2}{2\alpha\ell}\right)^{-\alpha}$$

σ, α, and ℓ are parameters which are optimized during the fitting of the Gaussian proces

a more intuitive and less math-in-your-face overview of Gaussian processes, take a look a
ious post on them.

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

If we've already sampled some points (hyperparameters and the scores they resulted in), ther Gaussian process to that data, and get an estimate of our model's performance as a function c hyperparameters. For example, in the image below, the cross-validated error of the algorithm axis, and a hyperparameter value is on the x-axis. The gray dots show the error for hyperpara which have been tried. Also shown is the Gaussian process's estimate as to the error over all hyperparameter combinations (blue line) and its 1 sigma confidence interval (shaded area).



But really where the Gaussian process comes in handy is when we want to figure out what c of hyperparameters we should try next! To determine what point to sample next, we can find hyperparameters which maximize an "acquisition function". The acquisition function is a fu when evaluated at $\mathbf{x}$ (some combination of hyperparameters), tells us how advantagous it w evaluate our expensive function f at that point. That way, we can use the acquisition functio next combination of hyperparameters we should try!

There are a few different functions which we could use for the acquisition function, includin probability of improvement, the expected improvement, and the upper confidence bound. He the expected improvement as our acquisition function.

The expected improvement for some combination of hyperparameters $\mathbf{x}$ is, like the name su average improvement that we expect from that combination of hyperparameters over the be found so far ($\hat{\mathbf{x}}$):

$$EI(\mathbf{x}) = \mathbb{E}[\max(0,\ f(\mathbf{x}) - f(\hat{\mathbf{x}}))]$$

where $f(\mathbf{x})$ is our Gaussian process' estimation as to our model's performance with hyperpa combination $\mathbf{x}$, and $f(\hat{\mathbf{x}})$ is the best performance that we've actually acheived so far.

How do we actually compute that? The Gaussian process is modeling the probability of our m performance (f) as a function of the hyperparameters ($\mathbf{x}$), and so at any value of $\mathbf{x}$, it gives us distribution for its prediction of the value of f:

$$p(f(\mathbf{x})|\mathbf{x})\ \sim\ \mathcal{N}(\mu(\mathbf{x}), \sigma(\mathbf{x}))$$

So, we can integrate that probability distribution (times the improvement magnitude) above t our current best performance value. This gives us how much of an improvement we can exp hyperparameters $\mathbf{x}$ over our current best, $\hat{\mathbf{x}}$.

$$\begin{aligned} EI(\mathbf{x}) &= \int_{f(\hat{\mathbf{x}})}^{\infty} p(f(\mathbf{x})|\mathbf{x})(f(\mathbf{x}) - f(\hat{\mathbf{x}}))df(\mathbf{x}) \\ &= \int_{f(\hat{\mathbf{x}})}^{\infty} \mathcal{N}(f(\mathbf{x})|\mu(\mathbf{x}), \sigma(\mathbf{x}))\ (f(\mathbf{x}) - f(\hat{\mathbf{x}}))\ df(\mathbf{x}) \end{aligned}$$

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

Brendan Hasz
Data Scientist at C.H. Robinson
Github
Email
LinkedIn
Resume

After "applying some tedious integration by parts" (haha, Jones et al., 1998), we have an analy solution:

$$EI(\mathbf{x}) = (\mu(\mathbf{x}) - f(\hat{\mathbf{x}}))\Phi(z) + \sigma(\mathbf{x})\phi(z)$$

where Φ is the cumulative distribution function of the standard normal distribution, ϕ is the density function of the standard normal, and z is the scaled difference between the Gaussian mean and our current best performance:

$$z = \frac{\mu(\mathbf{x}) - f(\hat{\mathbf{x}})}{\sigma(\mathbf{x})}$$

Using our Gaussian process to estimate $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$, we can then search for the $\mathbf{x}$ which give us the best expected improvement if we take our next sample there:

$$\mathbf{x}_{t+1} = \arg\max_{\mathbf{x}_{t+1}} EI(\mathbf{x}_{t+1})$$

For more information, Brochu et al., 2010 is a great tutorial on Bayesian optimization, which i intro to Gaussian processes and info about several different types of acquisition functions. Bu math - on to the code!

Bayesian Optimization

Bayesian optimization isn't specific to finding hyperparameters - it lets you optimize *any* exp function. That includes, say, the parameters of a simulation which takes a long time, or the c of a scientific research study, or the appearance of a website during an A/B test. Before we wr optimize our hyperparameters, we'll build a system which can optimize any expensive functi Bayesian optimization.

The code

To optimize our expensive function, let's build a python class which we can use to keep track hyperparameter combinations we've sampled, the resulting error, and then suggest the next of hyperparameters that we should try based on that information.

When initializing the optimizer, the user should define the lower and upper bounds of the par optimize (so our optimizer can search some confined space!). Also, the user should be able to whether they want to maximize or minimize the function (we'll do that via a `minimize` arg).

For example, to create an optimizer which finds the maximum of a function with 2 parameter float between 0 and 1, and the second an integer between 10 and 100, the user should be able t

```
gpo = GaussianProcessOptimizer([0, 10], [1, 100],
                               minimize=False)
```

So to start, let's build the class definition and the constructor (which defines how one initializ as above):

```
class GaussianProcessOptimizer():
    """Bayesian function optimizer which uses a Gaussian process
    to model the expensive function, and expected improvement as
    the acquisition function.
    """

    def __init__(self, lb, ub, minimize=True):
        """Gaussian process-based optimizer

        Parameters
        ----------
        lb : list
            Lower bound for each parameter.
        ub : list
            Upper bound for each parameter.
        minimize : bool
            Whether to minimize (True) or maximize (False).
        """

        # Store parameters
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

Brendan Hasz

```
        self.lb = lb
        self.ub = ub
        self.minimize = minimize
        self.db = [ub[i]-lb[i] for i in range(self.num_dims)]
        self.bounds = [(lb[i], ub[i]) for i in range(self.num_dims)]

        # Gaussian process to use for estimating the function
        self.gp = GaussianProcessRegressor(
            kernel=RationalQuadratic()+WhiteKernel(),
            alpha=0.0,
            n_restarts_optimizer=10,
        )

        # Keep track of points so far, and the best
        self.x = []
        self.y = []
        self.opt_x = None
        if minimize:
            self.opt_y = np.inf
        else:
            self.opt_y = -np.inf
```

The class will also need a function to add points to the history, and the resulting function val corresponding to that point. This function should allow multiple y values per x value, becaus use it for estimating the cross-validated scores, we'll have a different score for each fold! Also should keep track of the point corresponding to the best y value which has been sampled so f this function to the `GaussianProcessOptimizer` class:

```
def add_point(self, x, y):
    """Add a point to the history of sampled points."""

    # Append to sample record
    if isinstance(y, list): #repeated x values
        for ty in y:
            self.x.append(x)
            self.y.append(ty)
        ty = np.array(y).mean()
    else:
        self.x.append(x)
        self.y.append(y)
        ty = y

    # Store best point so far
    if self.minimize:
        if ty < self.opt_y:
            self.opt_y = ty
            self.opt_x = x
    else:
        if ty > self.opt_y:
            self.opt_y = ty
            self.opt_x = x
```

The meat of the optimizer is its ability to fit a Gaussian process to the sampled points, so we'l private function to do that fitting. We'll normalize the y-values, which will cause our Gaussia assume that, in areas of parameter space which we have not explored yet, the y-values are ro to the mean of the y-values for the points which we have sampled. The way in which we nor need to be kept track of, so that we can *un*-normalize the predictions of the Gaussian process

We'll also normalize the x-values so that we have to worry less about the exact value of the le parameter of the Gaussian process (see my previous post on Gaussian processes to see why tl pain). Let's add to the `GaussianProcessOptimizer` class a method to fit the Gaussian process:

```
def _fit_gp(self, step=None):
    """Fit the Gaussian process to data."""

    # Normalize y
    self._y_mean = y.mean()
    self._y_std = y.std()
    y = (y-self._y_mean)/self._y_std

    # Normalize x
    for iD in range(self.num_dims):
        x[:, iD] = (x[:, iD]-self.lb[iD]) / self.db[iD]

    # Fit the Gaussian process
    self.gp = self.gp.fit(x, y)
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

Our optimizer will also need to be able use the Gaussian process to predict the y-values (e.g. t validated performance) for a given x-value (e.g. the hyperparameter values). We need to norm new x values in the same way we did when fitting the Gaussian process (above), and un-norm predicted y-values as discussed above. Let's another private method to the class which perfo prediction:

```
def _pred_gp(self, x, return_std=False):
    """Predict y with the Gaussian process."""

    # Normalize x
    for iD in range(self.num_dims):
        x[:, iD] = (x[:, iD] - self.lb[iD]) / self.db[iD]

    # Predict y
    y, y_std = self.gp.predict(x, return_std=True)

    # Convert y back to true scale
    y = y*self._y_std+self._y_mean
    y_std = y_std*self._y_std

    # Return std dev if requested
    if return_std:
        return y, y_std
    else:
        return y
```

The last key peice of the optimizer is the acquisition function. Let's add one last private meth `GaussianProcessOptimizer` class which uses the Gaussian process' estimate and uncertainty as t values to compute the expected improvement:

```
def _expected_improvement(self, x):
    """Compute the expected improvement at x."""

    # Predict performance at x
    mu, sigma = self._pred_gp(x.reshape(-1, self.num_dims),
                              return_std=True)

    # Compute and return expected improvement
    flip = np.power(-1, self.minimize)
    z = flip*(mu-self.opt_y)/sigma
    return flip*(mu-self.opt_y)*norm.cdf(z) + sigma*norm.pdf(z)
```

Now that our optimizer is able to fit the Gaussian process to our data, we need it to be able to s points to sample. The easiest method of suggesting points is just generating a random point t the parameter bounds:

```
def random_point(self, get_dict=False):
    """Get a random point within the bounds."""
    return np.random.uniform(self.lb, self.ub)
```

But what we really want is for the optimizer to suggest a point which is either likely to have a value, or a point for which we're highly uncertain as to the y-values in that area of parameter add to the `GaussianProcessOptimizer` class a method which first fits a Gaussian process to the po so far, uses that fit Gaussian process to compute the expected improvement, and then searche values which result in the best expected improvement:

```
def next_point(self):
    """Get the point with the highest expected improvement."""

    # Fit the Gaussian process to samples so far
    self._fit_gp()

    # Find x with greatest expected improvement
    x = self.random_point()
    best_score = np.inf
    n_restarts = 10
    for iR in range(n_restarts):

        # Maximize expected improvement
        res = minimize(lambda x: -self._expected_improvement(x),
                       self.random_point(),
                       method='L-BFGS-B',
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages


```
                        bounds=self.bounds)

        # Keep x if it's the best so far
        if res.fun < best_score:
            best_score = res.fun
            x = res.x

    # Return x with highest expected improvement
    return x
```

After sampling as many points as we have time to sample, we'll want the optimizer to spit ou which results in the best y-value. There's two ways we could do this. The simplest way woul optimizer to simply search the history of sampled points to find the point with the highest y- However, a different method would be to use the Gaussian process to find the point with the b y-value. We'll add to the `GaussianProcessOptimizer` class a function which takes a `expected` argum allows the user to choose between the two options. When `expected=True` (the default), the opim search for the point which has the best y-value, as predicted by the Gaussian process. When the optimizer object will simply return the point in the sample history which had the highest

```
def best_point(self, expected=True):
    """Get the best point (expected or actual)"""

    # Return best point which was actually sampled
    if not expected:
        return self.x[self.y.index(min(self.y))]

    # Fit the Gaussian process to samples so far
    self._fit_gp()

    # Find x with greatest expected score
    flip = np.power(-1, self.minimize)
    rx = x.reshape(-1,self.num_dims)
    score_func = lambda x: flip*self._pred_gp(rx)
    x = self.random_point()
    best_score = np.inf
    n_restarts = 10
    for iR in range(n_restarts):

        # Maximize expected improvement
        res = minimize(score_func,
                       self.random_point(),
                       method='L-BFGS-B',
                       bounds=self.bounds)

        # Keep x if it's the best so far
        if res.fun < best_score:
            best_score = res.fun
            x = res.x

    # Return x with highest expected value
    return x
```

Finally, we want a way to visualize the Gaussian process' estimate of the expensive function' add to the optimizer class a method which plots the sampled points along with the Gaussian estimate of the function value in either one or two dimensions:

```
def plot_surface(self, x_dim=0, y_dim=None, res=100):
    """Plot the estimated surface of the function"""

    # Fit the Gaussian process to points so far
    self._fit_gp()

    # 1D plot
    if y_dim is None:

        # Predict y as a fn of x
        res = 100 #resolution
        x_pred = np.ones((res, self.num_dims))
        x_pred *= (np.array(self.bounds)
                   .mean(axis=1)
                   .reshape(-1, self.num_dims))
        x_pred[:,x_dim] = np.linspace(self.lb[x_dim],
                                      self.ub[x_dim], res)
        y_pred, y_err = self._pred_gp(x_pred, return_std=True)

        # Plot the Gaussian process' estimate of the function
        plot_err(x_pred[:, x_dim], y_pred, y_err)
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

Brendan Hasz

```
        # Plot the sampled points
        for iP in range(len(self.x)):
            plt.plot(self.x[iP][x_dim],
                     self.y[iP], '.', color='0.6')

    # 2D plot
    else:

        # Predict y as a fn of x
        res = 100 #resolution
        x_pred = np.ones((res*res, self.num_dims))
        x_pred *= (np.array(self.bounds)
                   .mean(axis=1)
                   .reshape(-1, self.num_dims))
        xp, yp = np.meshgrid(
            np.linspace(self.lb[x_dim], self.ub[x_dim], res),
            np.linspace(self.lb[y_dim], self.ub[y_dim], res))
        x_pred[:,x_dim] = xp.reshape(-1)
        x_pred[:,y_dim] = yp.reshape(-1)
        y_pred = self._pred_gp(x_pred, return_std=False)

        # Plot the Gaussian process
        plt.imshow(y_pred.reshape((res, res)), aspect='auto',
                   interpolation='bicubic', origin='lower',
                   extent=(self.lb[x_dim], self.ub[x_dim],
                           self.lb[y_dim], self.ub[y_dim]))
        plt.colorbar()

        # Plot the sampled points
        for iP in range(len(self.x)):
            plt.plot(self.x[iP][x_dim],
                     self.x[iP][y_dim],
                     '.', color='0.6')
```

Minimizing a 1D Function

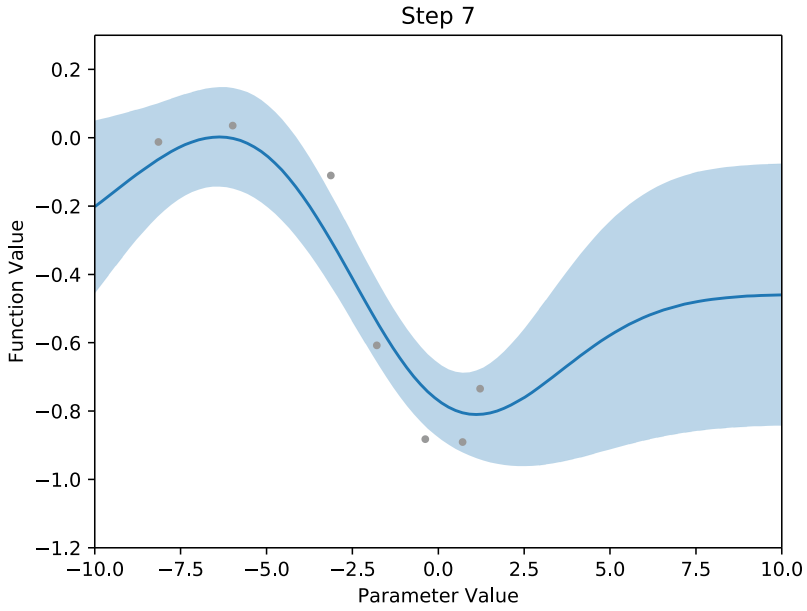
Now we can find the minimum (or maximum) of any expensive function with our Bayesian c
Supose we have some expensive function (the function below isn't expensive, of course, but i
demonstration purposes):

```
# Some 1D function
func = lambda x: -np.sin(x)/x + 0.1*np.random.randn()
```

We can use the optimizer class we just created to find the minimum of that function in as fev
possible. The optimizer lets us sample random points in the range:

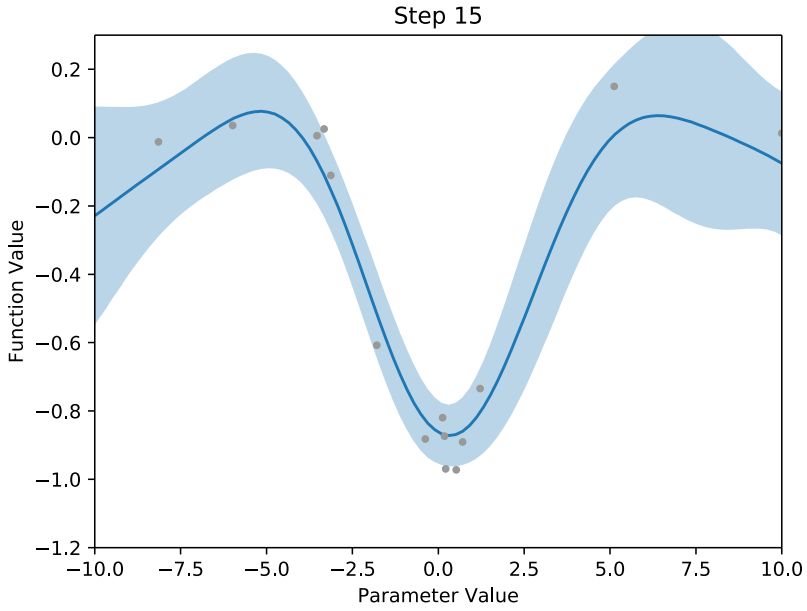
```
# Create an optimizer object
# (using the class we just created!)
gpo = GaussianProcessOptimizer([-10], [10])

# Randomly sample points
for i in range(10):

    # Get a new random point in the range
    new_params = gpo.random_point()

    # Compute the function value at that point
    # (this is usually the expensive part)
    value = func(new_params)

    # Store the outcome
    gpo.add_point(new_params, value)

    # Plot the function estimate
    gpo.plot_surface()
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages



But more importantly, it provides suggestions as to where to sample next, in order to most eff minimize the function:

```
# Find the minimum w/ as few steps as possible
for i in range(10):

    # Get a new random point in the range
    new_params = gpo.next_point()

    # Compute the function value at that point
    # (this is usually the expensive part)
    value = func(new_params)

    # Store the outcome
    gpo.add_point(new_params, value)

    # Plot the function estimate
    gpo.plot_surface()
```



ce how when sampling the points the optimizer suggests, the samples concentrate arour ...imum of the function, allowing us to better-explore promising areas of parameter space.

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

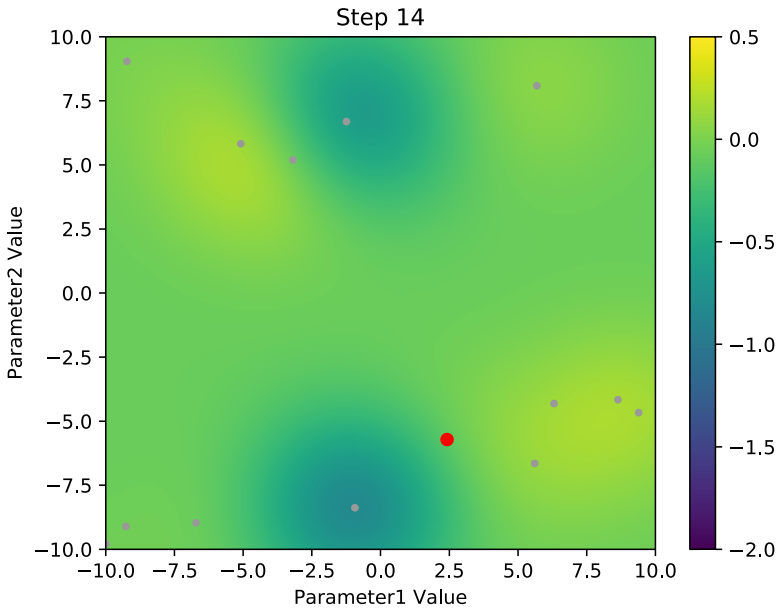
Minimizing a 2D Function

Our optimizer would only be moderately useful if it could only optimize one parameter at a ti
Gaussian processes can handle multiple dimensions, and therefore so can our optimizer. For
suppose we have an expensive-to-evaluate two-dimensional function (again, this isn't expen
for demonstration purposes):

```
# Some 2D function
func = lambda x: (-np.sin(x[0])/x[0]
                  - np.sin(x[1])/x[1]
                  + 0.1*np.random.randn())
```

We can create a 2D optimizer and make some random samples in exactly the same way as be

```
# Create an optimizer object
gpo = GaussianProcessOptimizer([-10, -10], [10, 10])

# Perform a few random samples
for i in range(40):

    # Get a new random point in the range
    new_params = gpo.random_point()

    # Compute the function value at that point
    value = func(new_params)

    # Store the outcome
    gpo.add_point(new_params, value)

    # Plot the surface
    gpo.plot_surface(x_dim=0, y_dim=1)
```

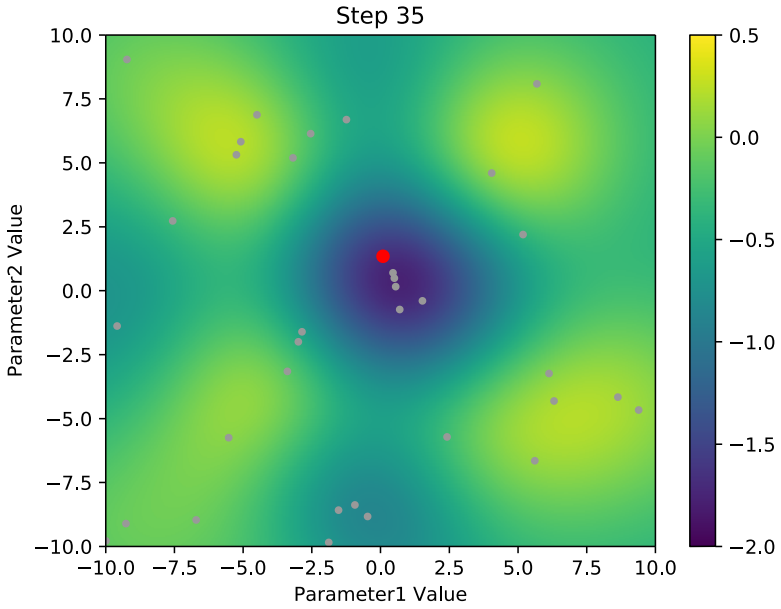


And we can also make directed samples in the same way as before:

```
# Sample directed by the optimizer
for i in range(10):

    # Point suggested by optimizer
    new_params = gpo.next_point()

    # Compute the function value at that point
    value = func(new_params)

    # Store the outcome
    gpo.add_point(new_params, value)
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

```
# Plot the surface
gpo.plot_surface(x_dim=0, y_dim=1)
```



Notice how the optimizer now focuses on promising areas, occasionally exploring the more u areas of parameter-space. Also keep in mind that the `GaussianProcessOptimizer` isn't limited to ju dimensions - in theory it can handle as many dimensions as you can throw at it. But of cours of dimensionality will catch up with you fairly quickly and you'll need an inordinate number to perform effective optimization.

Hyperparameter Optimization

Now that we have a Bayesian optimizer, we can create a function to find the hyperparameters machine learning model which optimize the cross-validated performance. All this function r and y data, the predictive model (in the form of an sklearn Estimator), and the hyperparamete The user should also be able to specify what metric to optimize (e.g. mean squared error or ac addition to whether that metric should be maximized or minimized. This is because some m better when lower, like mean squared error, while others are better when higher, like the coef determination. Users should also be able to specify how many samples to be taken, and how random samples to take before switching to Bayesian optimization. This function won't be a `GaussianProcessOptimizer`, but rather a stand-alone function which uses that class.

```
def optimize_params(X, y, model, bounds,
                    metric=make_scorer(mean_squared_error),
                    minimize=True,
                    n_splits=3,
                    shuffle=True,
                    max_evals=50,
                    n_random=5):
    """Optimize model parameters using cross-fold validation.

    Parameters
    ----------
    X : pandas DataFrame
        Independent variable values (features)
    y : pandas Series
        Dependent variable values (target)
    model : sklearn Estimator
        Predictive model to optimize
    bounds : dict
        Parameter bounds.
    metric : sklearn scorer
        Metric to use for evaluation.
    minimize : bool
        Whether to minimize ``metric``.
        If true, minimize; if false, maximize.
    n_splits : int
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

Brendan Hasz

```
    Number of cross-validation folds.
shuffle : bool
    Whether to shuffle samples before splitting into CV folds.
max_evals : int
    Max number of cross-validation evaluations to perform.
n_random : int
    Number of evaluations to use random parameter combinations
    before switching to Bayesian global optimization.

Returns
-------
opt_params : dict
    Optimal parameters.
optimizer : dsutils.optimization.GaussianProcessOptimizer
    Optimizer used to select the points.  Contains the history
    of all points which were sampled.
"""

# Collect info about parameters to optimize
Np = len(bounds) #number of parameters
step_params = [e for e in bounds]
steps = [e.split('__')[0] for e in step_params]
params = [e.split('__')[1] for e in step_params]
lb = [bounds[e][0] for e in step_params]
ub = [bounds[e][1] for e in step_params]

# Initialize the Gaussian process optimizer
gpo = GaussianProcessOptimizer(lb, ub, minimize=minimize)

# Create a cross-fold generator
kf = KFold(n_splits=n_splits, shuffle=shuffle)

# Search for optimal parameters
for i in range(max_evals):

    # Get next set of parameters to try
    if i < n_random:
        new_params = gpo.random_point()
    else:
        new_params = gpo.next_point()

    # Modify model to use new parameters
    for iP in range(Np):
        tP = {params[iP]: new_params[iP]}
        model.named_steps[steps[iP]].set_params(**tP)

    # Compute and store cross-validated metric
    scores = cross_val_score(model, X, y, cv=kf,
                             scoring=scorer, n_jobs=n_jobs)

    # Store parameters and scores
    gpo.add_point(new_params, scores)

# Return optimal parameters and the optimizer object used
opt_params = dict(zip(step_params, gpo.best_point()))
return opt_params, gpo
```

Optimizing One Hyperparameter

Let's try the hyperparameter optimizer out on some real data. We'll use the Diabetes dataset, a
predict the severity of the progression of patients' diabetes from variables such as age, sex, BI
pressure, and blood serum measurements. Sklearn comes packaged with the dataset, so we'll
sklearn:

```
# Load the diabetes dataset
from sklearn.datasets import load_diabetes
diabetes = load_diabetes()
X_data = pd.DataFrame(diabetes['data'])
y_data = pd.Series(diabetes['target'], X_data.index)
y_data = (y_data-y_data.mean())/y_data.std()
```

We'll build an sklearn data processing and prediction pipeline in the same way as one norma
n working with sklearn:

```
rediction pipeline
boost = Pipeline([
 ('scaler', RobustScaler()),
 ('imputer', SimpleImputer(strategy='mean')),
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages


```
    ('regressor', CatBoostRegressor(verbose=False,
                                    loss_function='RMSE',
                                    learning_rate=0.1))
])
```

However, we now want to find the optimal learning rate for our dataset. Using the function w above, we can specify the range of learning rates to search using the `bounds` parameter. The bo specified using a dictionary, where the values are a tuple containing the lower and upper bou keys are a string with the corresponding parameter name. Note that the parameter name is tl the step in the pipeline, and then the parameter name within that step which we want to opti separated by a double-underscore. We'll optimize CatBoost's learning rate to find the learning gives us the best predictive performance.

```
# Parameter bounds
bounds = {
    'regressor__learning_rate': [0.01, 0.9],
}
```

Now, all we have to do is call the `optimize_params` function to find the optimal learning rate!

```
# Find the optimal parameters
opt_params, gpo = optimize_params(
    X_data, y_data, catboost, bounds,
    max_evals=20, n_random=0, n_grid=20)
```

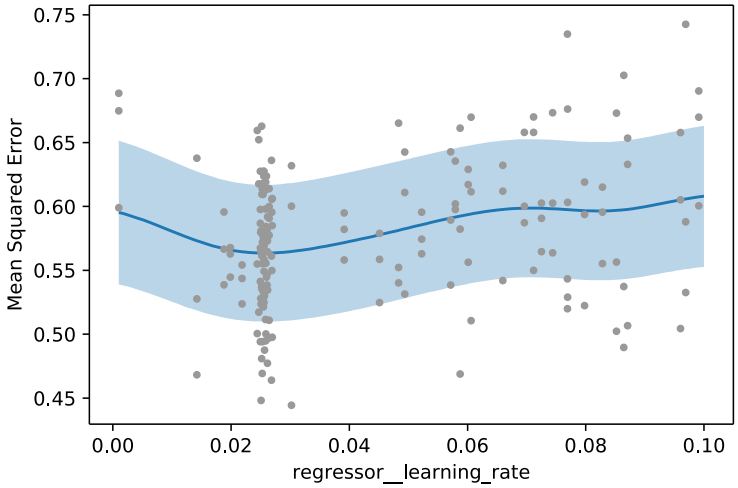
Now, we have the parameter value which the optimizer estimates is best for our dataset:

```
opt_params
```

```
{'regressor__learning_rate': 0.0263}
```

And we can also plot all the parameter values and their corresponding scores:

```
# Plot the score against the parameter value
gpo.plot_surface()
```



Optimizing Multiple Hyperparameters

Just like the Bayesian optimizer, our hyperparameter optimizer can handle multiple dimensi means we can optimize multiple parameters at the same time. This time let's try optimizing t CatBoost's hyperparameters: the learning rate *and* the leaf regularization parameter.

```
atBoost pipeline
boost = Pipeline([
 ('scaler', RobustScaler()),
 ('regressor', CatBoostRegressor(verbose=False,
                                 loss_function='RMSE',
```

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages


```
                                        learning_rate=0.1,
                                        l2_leaf_reg=2.0))
])

# Parameter bounds
bounds = {
    'regressor__learning_rate': [0.01, 0.5],
    'regressor__l2_leaf_reg': [0.01, 0.5],
}

# Find the optimal parameters
opt_params, gpo = optimize_params(
    X_data, y_data, catboost, bounds,
    max_evals=60, n_random=30)
```

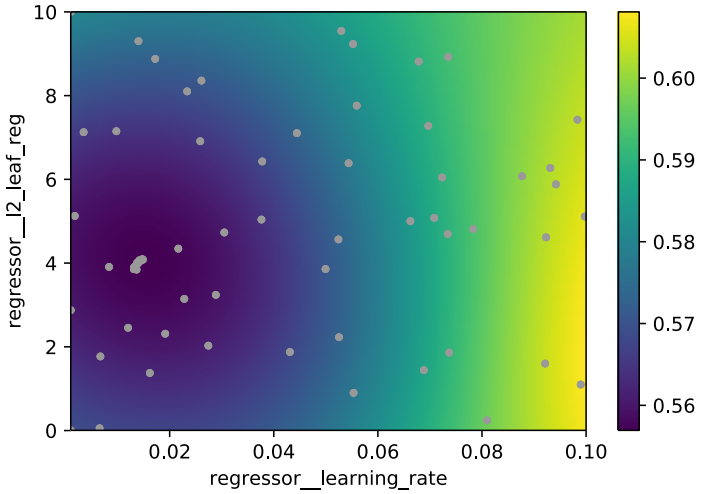
This gives us an estimate of the best *combination* of parameter values:

```
opt_params
```

```
{'regressor__learning_rate': 0.0174,
 'regressor__l2_leaf_reg': 3.92}
```

And the ability to view all the sampled combinations and the corresponding scores:

```
# Plot the score vs parameter values
gpo.plot_surface(x_dim=0, y_dim=1)
```



Conclusion

This sort of automatic parameter tuning is a huge time-saver when trying to find the parame
work best for your model and dataset. In practice, using a fancy Gaussian-process (or other) c
only marginally better than random sampling - in my experience random sampling usually g
about 70% of the way there.

There are a lot of great packages out there for either Bayesian optimization in general, and so
sklearn hyperparameter optimization specifically:

- HyperOpt
- skopt
- MOE
- HyperparameterHunter
- Spearmint
- BayesOpt

SMAC
HPOlib2
Optunity

ɔite the kitschy name, HyperparameterHunter specifically is great because it keeps track
previous parameter combinations and scores which you've run in the past. In theory, we coul

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages

one huge long run where we try all the different hyperparameters and then just pick the best practice, one usually ends up doing many shorter runs, tweaking things in between. So, it's n history of all your outcomes, instead of having to start the search from scratch each time you small change.

Also, Gaussian processes aren't the only surrogate models used to estimate the score as a fun hyperparameters. Other methods include random forests and tree Parzen estimators.

Lastly, keep in mind that you don't want to over-do it on the optimization (that is, search supe thoroughly). Especially if you're optimizing a large number of parameters at the same time, th to overfitting on your training set - even if you're using cross-validation! Using the Gaussian p estimate of the optimal parameter combination can help with this, because it smoothes the s landscape a bit, but just taking the parameter combination which gave you the best score is a overfitting.

ALSO ON **BRENDANHASZ.GITHUB.IO**

3 years ago • 2 comments
Representing Categorical Data ...

2 years ago • 3 comments
Bayesian Gaussian Mixture

Comments Community Privacy Policy 1 Login
Recommend 2 Tweet Share Sort by Best
Join the discussion…
LOG IN WITH
OR SIGN UP WITH DISQUS ?
Name

Thibault Putseys • 2 years ago • edited

Your smiling face pinned to the left of the screen is disturbing when reading about Bayesian optimization. Other than that, good article :)

Reply • Share ›

Hunter McGushion • 3 years ago

I suppose I'll take kitschy over pompous hahaha. Fantastic article, Brendan! Thanks for sharing your work and for mentioning HyperparameterHunter! I'd love to hear any of your thoughts on improving the library as I firmly believe it revolutionizes hyperparameter optimization, and I'm hoping to find others that are interested in contributing!
Thanks again for sharing your insights on hyperparameter optimization!

Reply • Share ›

Brendan Hasz Mod → Hunter McGushion • 3 years ago

Hahaha no offense intended about the name, I think it's a fantastic package!! I'll let you know if I have any thoughts about improving the library - so far it's been great and I'm just thrilled about not having to manually keep janky lists of scores + parameters haha. Especially when you're doing multiple optimization runs, it's such a pain to manually keep track of all the points the optimizer sampled and then feed them into the optimizer on the next run that I never even bother, so it's awesome that HyperparameterHunter supports keeping track of that!

Reply • Share ›

Hunter McGushion → Brendan Hasz • 3 years ago

None taken! I'll proudly admit the name is

RSS Feed
© 2021 Brendan Hasz
Hosted on GitHub Pages