# example of bayesian optimization with scikit-optimize

from numpy import mean

from sklearn.datasets import make\_blobs

from sklearn.model\_selection import cross\_val\_score

from sklearn.neighbors import KNeighborsClassifier

!pip install scikit\_optimize

from skopt.space import Integer

from skopt.utils import use\_named\_args

from skopt import gp\_minimize

# generate 2d classification dataset

X, y = make\_blobs(n\_samples=500, centers=3, n\_features=2)

# define the model

model = KNeighborsClassifier()

# define the space of hyperparameters to search

search\_space = [Integer(1, 5, name='n\_neighbors'), Integer(1, 2, name='p')]

# define the function used to evaluate a given configuration

@use\_named\_args(search\_space)

def evaluate\_model(\*\*params):

  model.set\_params(\*\*params)

  # calculate 5-fold cross validation

  result = cross\_val\_score(model, X, y, cv=5, n\_jobs=-1, scoring='accuracy')

 # calculate the mean of the scores

  estimate = mean(result)

  return 1.0 - estimate

# perform optimization

result = gp\_minimize(evaluate\_model, search\_space)

# summarizing finding:

print('Best Accuracy: %.3f' % (1.0 - result.fun))

print('Best Parameters: n\_neighbors=%d, p=%d' % (result.x[0], result.x[1]))

**OUTPUT**

**Best Accuracy: 0.998**

**Best Parameters: n\_neighbors=3, p=1**

**COMFIRMANDO**

from sklearn.datasets import make\_blobs

from sklearn.neighbors import KNeighborsClassifier

# generate 2d classification dataset

X, y = make\_blobs(n\_samples=500, centers=3, n\_features=2)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30)

# define the model

model = KNeighborsClassifier(n\_neighbors=3, p=1)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

result = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(result)

**OUTPUT**

Confusion Matrix:

[[50 0 0]

[ 0 50 0]

[ 0 0 50]]

result1 = classification\_report(y\_test, y\_pred)

print("Classification Report:",)

print (result1)

result2 = accuracy\_score(y\_test,y\_pred)

print("Accuracy:",result2)

Classification Report:

precision recall f1-score support

0 1.00 1.00 1.00 50

1 1.00 1.00 1.00 50

2 1.00 1.00 1.00 50

accuracy 1.00 150

macro avg 1.00 1.00 1.00 150

weighted avg 1.00 1.00 1.00 150

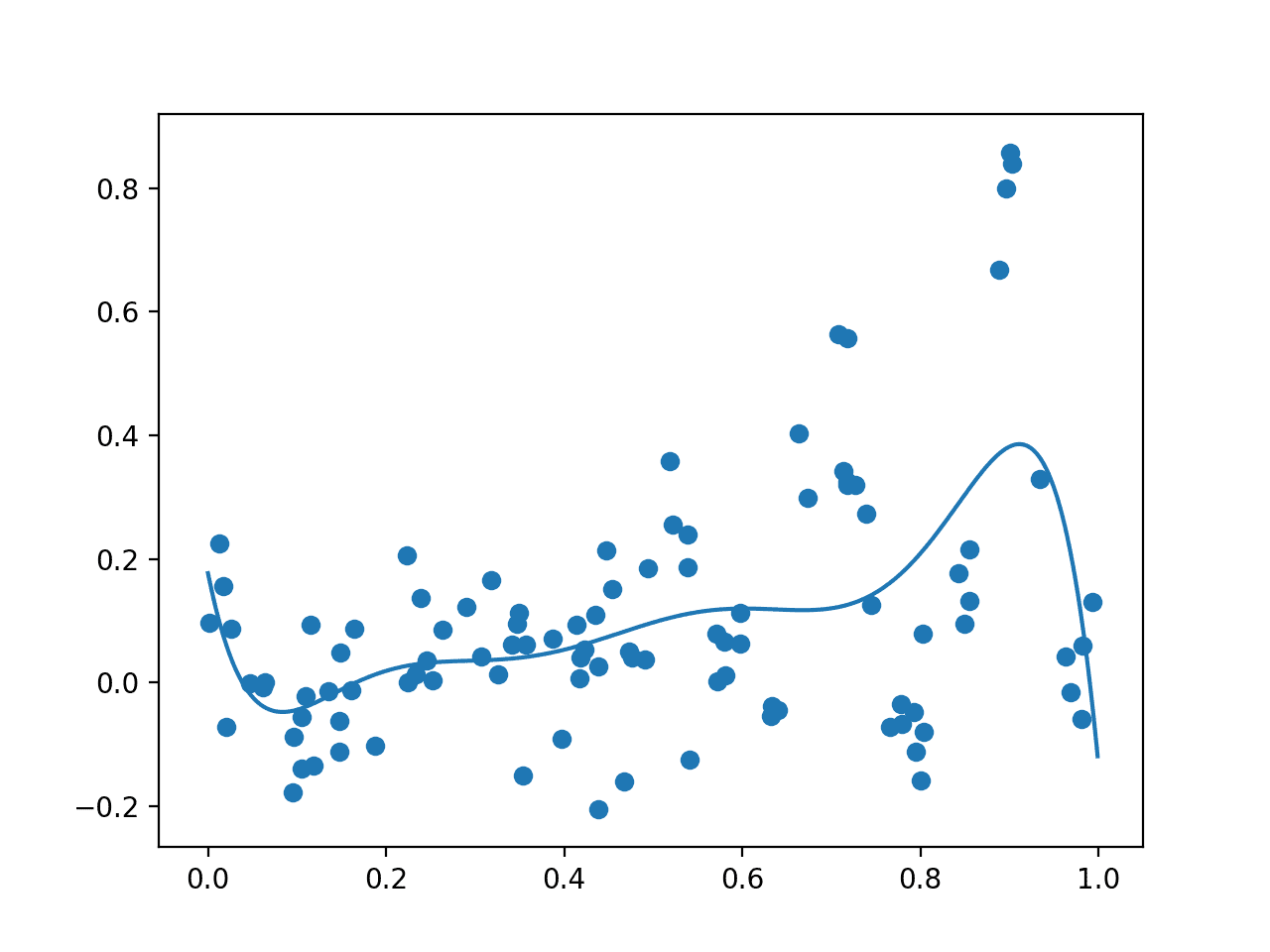
Accuracy: 1.0

|  |  |
| --- | --- |
|  | # example of bayesian optimization for a 1d function from scratch  from math import sin  from math import pi  from numpy import arange  from numpy import vstack  from numpy import argmax  from numpy import asarray  from numpy.random import normal  from numpy.random import random  from scipy.stats import norm  from sklearn.gaussian\_process import GaussianProcessRegressor  from warnings import catch\_warnings  from warnings import simplefilter  from matplotlib import pyplot    # objective function  def objective(x, noise=0.1):  noise = normal(loc=0, scale=noise)  return (x\*\*2 \* sin(5 \* pi \* x)\*\*6.0) + noise    # surrogate or approximation for the objective function  def surrogate(model, X):  # catch any warning generated when making a prediction  with catch\_warnings():  # ignore generated warnings  simplefilter("ignore")  return model.predict(X, return\_std=True)    # probability of improvement acquisition function  def acquisition(X, Xsamples, model):  # calculate the best surrogate score found so far  yhat, \_ = surrogate(model, X)  best = max(yhat)  # calculate mean and stdev via surrogate function  mu, std = surrogate(model, Xsamples)  mu = mu[:, 0]  # calculate the probability of improvement  probs = norm.cdf((mu - best) / (std+1E-9))  return probs    # optimize the acquisition function  def opt\_acquisition(X, y, model):  # random search, generate random samples  Xsamples = random(100)  Xsamples = Xsamples.reshape(len(Xsamples), 1)  # calculate the acquisition function for each sample  scores = acquisition(X, Xsamples, model)  # locate the index of the largest scores  ix = argmax(scores)  return Xsamples[ix, 0]    # plot real observations vs surrogate function  def plot(X, y, model):  # scatter plot of inputs and real objective function  pyplot.scatter(X, y)  # line plot of surrogate function across domain  Xsamples = asarray(arange(0, 1, 0.001))  Xsamples = Xsamples.reshape(len(Xsamples), 1)  ysamples, \_ = surrogate(model, Xsamples)  pyplot.plot(Xsamples, ysamples)  # show the plot  pyplot.show()    # sample the domain sparsely with noise  X = random(100)  y = asarray([objective(x) for x in X])  # reshape into rows and cols  X = X.reshape(len(X), 1)  y = y.reshape(len(y), 1)  # define the model  model = GaussianProcessRegressor()  # fit the model  model.fit(X, y)  # plot before hand  plot(X, y, model)  # perform the optimization process  for i in range(100):  # select the next point to sample  x = opt\_acquisition(X, y, model)  # sample the point  actual = objective(x)  # summarize the finding  est, \_ = surrogate(model, [[x]])  print('>x=%.3f, f()=%3f, actual=%.3f' % (x, est, actual))  # add the data to the dataset  X = vstack((X, [[x]]))  y = vstack((y, [[actual]]))  # update the model  model.fit(X, y)    # plot all samples and the final surrogate function  plot(X, y, model)  # best result  ix = argmax(y)  print('Best Result: x=%.3f, y=%.3f' % (X[ix], y[ix])) |

Running the example first creates an initial random sample of the search space and evaluation of the results. Then a GP model is fit on this data.

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

A plot is created showing the raw observations as dots and the surrogate function across the entire domain. In this case, the initial sample has a good spread across the domain and the surrogate function has a bias towards the part of the domain where we know the optima is located.



Plot of Initial Sample (dots) and Surrogate Function Across the Domain (line).

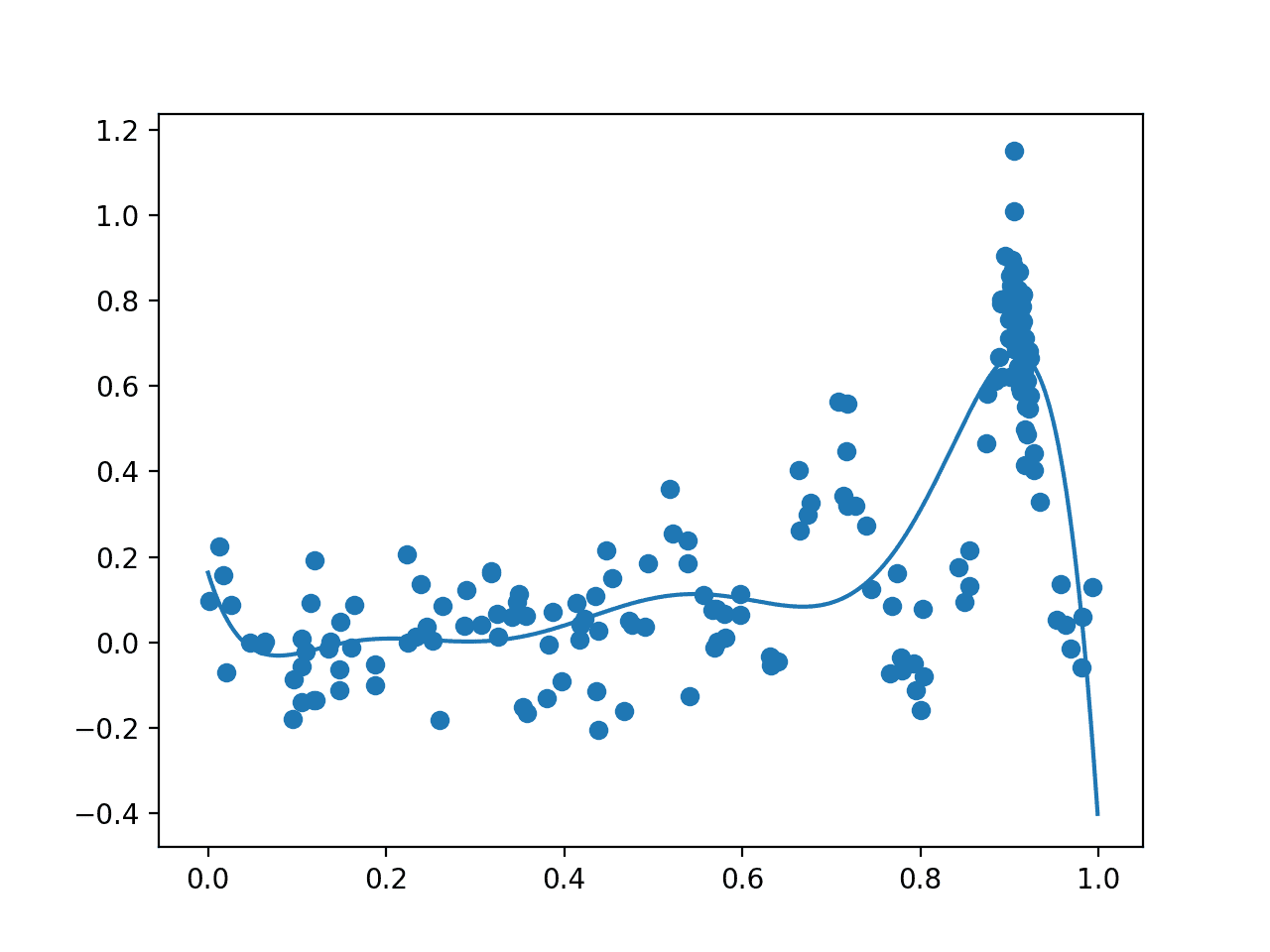
The algorithm then iterates for 100 cycles, selecting samples, evaluating them, and adding them to the dataset to update the surrogate function, and over again.

Each cycle reports the selected input value, the estimated score from the surrogate function, and the actual score. Ideally, these scores would get closer and closer as the algorithm converges on one area of the search space.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ...  >x=0.922, f()=0.661501, actual=0.682  >x=0.895, f()=0.661668, actual=0.905  >x=0.928, f()=0.648008, actual=0.403  >x=0.908, f()=0.674864, actual=0.750  >x=0.436, f()=0.071377, actual=-0.115 |

Next, a final plot is created with the same form as the prior plot.

This time, all 200 samples evaluated during the optimization task are plotted. We would expect an overabundance of sampling around the known optima, and this is what we see, with may dots around 0.9. We also see that the surrogate function has a stronger representation of the underlying target domain.



Plot of All Samples (dots) and Surrogate Function Across the Domain (line) after Bayesian Optimization.

Finally, the best input and its objective function score are reported.

We know the optima has an input of 0.9 and an output of 0.810 if there was no sampling noise.

Given the sampling noise, the optimization algorithm gets close in this case, suggesting an input of 0.905.

|  |  |
| --- | --- |
| 1 | Best Result: x=0.905, y=1.150 |

**Hyperparameter Tuning With Bayesian Optimization**

It can be a useful exercise to implement Bayesian Optimization to learn how it works.

In practice, when using Bayesian Optimization on a project, it is a good idea to use a standard implementation provided in an open-source library. This is to both avoid bugs and to leverage a wider range of configuration options and speed improvements.

Two popular libraries for Bayesian Optimization include [Scikit-Optimize](https://github.com/scikit-optimize/scikit-optimize) and [HyperOpt](https://github.com/hyperopt/hyperopt). In machine learning, these libraries are often used to tune the hyperparameters of algorithms.

Hyperparameter tuning is a good fit for Bayesian Optimization because the evaluation function is computationally expensive (e.g. training models for each set of hyperparameters) and noisy (e.g. noise in training data and stochastic learning algorithms).

In this section, we will take a brief look at how to use the Scikit-Optimize library to optimize the hyperparameters of a k-nearest neighbor classifier for a simple test classification problem. This will provide a useful template that you can use on your own projects.

The [Scikit-Optimize project](https://github.com/scikit-optimize/scikit-optimize) is designed to provide access to Bayesian Optimization for applications that use SciPy and NumPy, or applications that use scikit-learn machine learning algorithms.

First, the library must be installed, which can be achieved easily using pip; for example:

|  |  |
| --- | --- |
| 1 | sudo pip install scikit-optimize |

It is also assumed that you have [scikit-learn](https://scikit-learn.org/stable/index.html) installed for this example.

Once installed, there are two ways that scikit-optimize can be used to optimize the hyperparameters of a scikit-learn algorithm. The first is to perform the optimization directly on a search space, and the second is to use the BayesSearchCV class, a sibling of the scikit-learn native classes for random and grid searching.

In this example, will use the simpler approach of optimizing the hyperparameters directly.

The first step is to prepare the data and define the model. We will use a simple test classification problem via the [make\_blobs() function](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html) with 500 examples, each with two features and three class labels. We will then use a [KNeighborsClassifier algorithm](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html).

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...  # generate 2d classification dataset  X, y = make\_blobs(n\_samples=500, centers=3, n\_features=2)  # define the model  model = KNeighborsClassifier() |

Next, we must define the search space.

In this case, we will tune the number of neighbors (*n\_neighbors*) and the shape of the neighborhood function (*p*). This requires ranges be defined for a given data type. In this case, they are Integers, defined with the min, max, and the name of the parameter to the scikit-learn model. For your algorithm, you can just as easily optimize *Real()* and *Categorical()* data types.

|  |  |
| --- | --- |
| 1  2  3 | ...  # define the space of hyperparameters to search  search\_space = [Integer(1, 5, name='n\_neighbors'), Integer(1, 2, name='p')] |

Next, we need to define a function that will be used to evaluate a given set of hyperparameters. We want to minimize this function, therefore smaller values returned must indicate a better performing model.

We can use the *use\_named\_args()* decorator from the scikit-optimize project on the function definition that allows the function to be called directly with a specific set of parameters from the search space.

As such, our custom function will take the hyperparameter values as arguments, which can be provided to the model directly in order to configure it. We can define these arguments generically in python using the *\*\*params* argument to the function, then pass them to the model via the [set\_params(\*\*) function](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html" \l "sklearn.neighbors.KNeighborsClassifier.set_params).

Now that the model is configured, we can evaluate it. In this case, we will use 5-fold cross-validation on our dataset and evaluate the accuracy for each fold. We can then report the performance of the model as one minus the mean accuracy across these folds. This means that a perfect model with an accuracy of 1.0 will return a value of 0.0 (1.0 – mean accuracy).

This function is defined after we have loaded the dataset and defined the model so that both the dataset and model are in scope and can be used directly.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | # define the function used to evaluate a given configuration  @use\_named\_args(search\_space)  def evaluate\_model(\*\*params):  # something  model.set\_params(\*\*params)  # calculate 5-fold cross validation  result = cross\_val\_score(model, X, y, cv=5, n\_jobs=-1, scoring='accuracy')  # calculate the mean of the scores  estimate = mean(result)  return 1.0 - estimate |

Next, we can perform the optimization.

This is achieved by calling the [gp\_minimize() function](https://scikit-optimize.github.io/" \l "skopt.gp_minimize) with the name of the objective function and the defined search space.

By default, this function will use a ‘*gp\_hedge*‘ acquisition function that tries to figure out the best strategy, but this can be configured via the *acq\_func* argument. The optimization will also run for 100 iterations by default, but this can be controlled via the *n\_calls* argument.

|  |  |
| --- | --- |
| 1  2  3 | ...  # perform optimization  result = gp\_minimize(evaluate\_model, search\_space) |

Once run, we can access the best score via the “fun” property and the best set of hyperparameters via the “*x*” array property.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # summarizing finding:  print('Best Accuracy: %.3f' % (1.0 - result.fun))  print('Best Parameters: n\_neighbors=%d, p=%d' % (result.x[0], result.x[1])) |

Tying this all together, the complete example is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | # example of bayesian optimization with scikit-optimize  from numpy import mean  from sklearn.datasets import make\_blobs  from sklearn.model\_selection import cross\_val\_score  from sklearn.neighbors import KNeighborsClassifier  from skopt.space import Integer  from skopt.utils import use\_named\_args  from skopt import gp\_minimize    # generate 2d classification dataset  X, y = make\_blobs(n\_samples=500, centers=3, n\_features=2)  # define the model  model = KNeighborsClassifier()  # define the space of hyperparameters to search  search\_space = [Integer(1, 5, name='n\_neighbors'), Integer(1, 2, name='p')]    # define the function used to evaluate a given configuration  @use\_named\_args(search\_space)  def evaluate\_model(\*\*params):    model.set\_params(\*\*params)  # calculate 5-fold cross validation  result = cross\_val\_score(model, X, y, cv=5, n\_jobs=-1, scoring='accuracy')    # calculate the mean of the scores  estimate = mean(result)  return 1.0 - estimate    # perform optimization  result = gp\_minimize(evaluate\_model, search\_space)  # summarizing finding:  print('Best Accuracy: %.3f' % (1.0 - result.fun))  print('Best Parameters: n\_neighbors=%d, p=%d' % (result.x[0], result.x[1])) |

Running the example executes the hyperparameter tuning using Bayesian Optimization.

The code may report many warning messages, such as:

|  |  |
| --- | --- |
| 1 | UserWarning: The objective has been evaluated at this point before. |

This is to be expected and is caused by the same hyperparameter configuration being evaluated more than once.

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, the model achieved about 97% accuracy via mean 5-fold cross-validation with 3 neighbors and a p-value of 2.

|  |  |
| --- | --- |
| 1  2 | Best Accuracy: 0.976  Best Parameters: n\_neighbors=3, p=2 |