Hyperopt: a Python library for model selection and hyperparameter optimization

COMPUTATIONAL SCIENCE&DISCOVERY

Hyperopt: a Python library for model selection and hyperparameter optimization

James Bergstra[1], **Brent Komer**[1], **Chris Eliasmith**[1], **Dan Yamins**[2] **and David D Cox**[3]

[1] University of Waterloo, Canada
[2] Massachusetts Institute of Technology, US
[3] Harvard University, US
E-mail: james.bergstra@uwaterloo.ca

Received 16 March 2014, revised 28 August 2014
Accepted for publication 4 May 2015
Published 28 July 2015

Computational Science & Discovery **8** (2015) 014008
doi:10.1088/1749-4699/8/1/014008

Abstract

Sequential model-based optimization (also known as Bayesian optimization) is one of the most efficient methods (per function evaluation) of function minimization. This efficiency makes it appropriate for optimizing the hyperparameters of machine learning algorithms that are slow to train. The Hyperopt library provides algorithms and parallelization infrastructure for performing hyperparameter optimization (model selection) in Python. This paper presents an introductory tutorial on the usage of the Hyperopt library, including the description of search spaces, minimization (in serial and parallel), and the analysis of the results collected in the course of minimization. This paper also gives an overview of Hyperopt-Sklearn, a software project that provides automatic algorithm configuration of the Scikit-learn machine learning library. Following Auto-Weka, we take the view that the choice of classifier and even the choice of preprocessing module can be taken together to represent a *single large hyperparameter optimization problem*. We use Hyperopt to define a search space that encompasses many standard components (e.g. SVM, RF, KNN, PCA, TFIDF) and common patterns of composing them together. We demonstrate, using search algorithms in Hyperopt and standard benchmarking data sets (MNIST, 20-newsgroups, convex shapes), that searching this space is practical and effective. In particular, we improve on best-known scores for the model space for both MNIST and convex shapes. The paper closes with some discussion of ongoing and future work.

Keywords: Python, Bayesian optimization, machine learning, Scikit-learn

Introduction

Sequential model-based optimization (SMBO, also known as Bayesian optimization) is a general technique for function optimization that includes some of the most call-efficient (in terms of function evaluations) optimization methods currently available. Originally developed for experiment design SMBO methods are generally applicable to scenarios in which a user wishes to minimize some scalar-valued function $f(x)$ that is costly to evaluate, often in terms of time or money. Compared with standard optimization strategies such as conjugate gradient descent methods, model-based optimization algorithms invest more time between function evaluations in order to reduce the number of function evaluations overall.

The advantages of SMBO are that it:

- leverages smoothness without analytic gradient,
- handles real-valued, discrete and conditional variables,
- handles parallel evaluations of $f(x)$,
- copes with hundreds of variables, even with a budget of just a few hundred function evaluations.

Many widely-used machine learning algorithms take a significant amount of time to train from data. At the same time, these same algorithms must be configured prior to training. Most implementations of machine learning algorithms have a set of configuration variables that the user can set which have various effects on how the training is done. Often there is no configuration that is optimal for all problem domains, so the best configuration will depend on the particular application. These configuration variables are called *hyperparameters*. Implementations of many of these algorithms can be found in the Scikit-learn project. Reference [Ped11] Scikit-learn is a open-source machine learning library written in Python that is quite popular amoung computational scientists. It boasts an easy to use interface and contains many tools for machine learning, including classification, preprocessing, clustering and regression. Many of these algorithms can be useful out-of-the-box, but to get the best performance the hyperparameters need to be tuned to the particular problem domain in which the algorithm is to be used. For example, support vector machines (SVMs) have hyperparameters that include the regularization strength (often C), the scaling of input data (and more generally, the preprocessing of input data), the choice of similarity kernel and the various parameters that are specific to that kernel choice. Decision trees (Dtree) are another machine learning algorithm with hyperparameters related to the heuristic for creating internal nodes, and the pruning strategy for the tree after (or during) training. Neural networks are a classic type of machine learning algorithm but they have so many hyperparameters that they have been considered too troublesome for inclusion in the Scikit-learn library.

Hyperparameter optimization is the act of searching the space of possible configuration variables for a training algorithm in order to find a set of variables that allows the algorithm to achieve more desirable results. Hyperparameters generally have a significant effect on the success of machine learning algorithms. A poorly-configured SVM may perform no better than chance, while a well-configured one may achieve state-of-the-art prediction accuracy. To experts and non-experts alike, adjusting hyperparameters to optimize end-to-end performance can be a tedious and difficult task. Hyperparameters come in many varieties—continuous-valued ones with and without bounds, discrete ones that are either ordered or not, and conditional ones that do not even always apply (e.g., the parameters of an optional preprocessing stage). Because of this variety, conventional continuous and combinatorial optimization algorithms either do not directly apply, or else operate without leveraging valuable structure in the configuration space. Common practice for the optimization of hyperparameters is (a) for algorithm developers to tune them by hand on representative problems to get good rules of thumb and default values and (b) for algorithm users to tune them manually for their particular prediction problems perhaps with the assistance of (multi-resolution) grid search. However, when dealing with more than a few hyperparameters (e.g. 5) this standard practice of manual search with grid refinement is not guaranteed to work well; in such cases even random search has been shown to be competitive with domain experts [BB12].

Hyperopt [Hyperopt] provides algorithms and software infrastructure for carrying out hyperparameter optimization for machine learning algorithms. Hyperopt provides an optimization interface that distinguishes a *configuration space* and an *evaluation function* that assigns real-valued *loss values* to points within the configuration space. Unlike the standard minimization interfaces provided by scientific programming libraries, Hyperopt's `fmin` interface requires users to specify the configuration space as a probability distribution. Specifying a probability distribution rather than just bounds and hard constraints allows domain experts to encode more of their intuitions regarding which values are plausible for various hyperparameters. Like SciPy's `optimize.minimize` interface, Hyperopt makes the SMBO algorithm itself an interchangeable component so that any search algorithm can be applied to any search problem. Currently three algorithms are provided—random search, simulated annealing, and Tree-of-Parzen-Estimators (TPE) algorithm introduced in [Ber11]—and more algorithms are planned (including [SMAC], and Gaussian-process-based [Sno12] and [Ber14]).

We are motivated to make hyperparameter optimization more reliable for four reasons:

Reproducibile research. Hyperopt formalizes the practice of model evaluation, so that benchmarking experiments can be reproduced at later dates, and by different people.

Empowering users. Learning algorithm designers can deliver flexible fully-configurable implementations to non-experts (e.g. deep learning systems), so long as they also provide a corresponding Hyperopt driver.

Designing better algorithms. As algorithm designers, we appreciate Hyperopt's capacity to find successful configurations that we might not have considered.

Fuzz testing. As algorithm designers, we appreciate Hyperopt's capacity to find failure modes via configurations that we had not considered.

This paper describes the usage and architecture of Hyperopt, for both sequential and parallel optimization of expensive functions.

Hyperopt can in principle be used for any SMBO problem (e.g. [Ber14]), but our development and testing efforts have focused on the optimization of hyperparameters for deep neural networks [hp-dbn], convolutional neural networks for object recognition [hp-convnet], and algorithms within the Scikit-learn library ([Kom14] and this paper).

Alternative software packages to Hyperopt include primarily Spearmint and SMAC. [Spearmint] provides Gaussian-Process Optimization as a Python package. The original spearmint code exists at https://github.com/JasperSnoek/spearmint, while an updated version has been recently released under a non-commercial license at https://github.com/HIPS/Spearmint. Reference [SMAC] is a Java package that provides the SMAC (same name) algorithm, which is similar to Gaussian-Process Optimization except that regression forests provide the engine for regression rather than Gaussian Processes. SMAC was developed for configuration SAT solvers, but has been used for algorithm configuration more generally and for machine learning hyperparameters in particular (e.g. [Egg13]).

The article is organized as follows:

- Introduction to Hyperopt.
- Introduction to configuration spaces.
- How to analyze the search with the trials object.
- Parallel evaluation with Hyperopt using a cluster.
- Introduction to Hyperopt-Sklearn.
- Example usage of Hyperopt-Sklearn.
- Empirical evalutation of Hyperopt-Sklearn.
- Discussion of results.
- Ongoing and future work.

Portions of this article have been presented previously as [Ber13b] and [Kom14].

Getting started with hyperopt

This section introduces basic usage of the `hyperopt.fmin` function, which is Hyperopt's basic optimization driver. We will look at how to write an objective function that `fmin` can optimize, and how to describe a configuration space that `fmin` can search.

Hyperopt shoulders the responsibility of finding the best value of a scalar-valued, possibly-stochastic function over a set of possible arguments to that function. Whereas most optimization packages assume that these inputs are drawn from a vector space, Hyperopt encourages you, the user, to describe your configuration space in more detail. Hyperopt is typically aimed at very difficult search settings, especially ones with many hyperparameters and a small budget for function evaluations. By providing more information about where your function is defined, and where you think the best values are, you allow algorithms in Hyperopt to search more efficiently.

The way to use Hyperopt is to describe:

- the objective function to minimize,
- the space over which to search,
- a trials database (optional),
- the search algorithm to use (optional).

This section will explain how to describe the objective function, configuration space and optimization algorithm. Later, the section Trial results: more than just the loss will explain how to use the trials database to analyze the results of a search and the section Parallel Evaluation with a Cluster will explain how to use parallel computation to search faster.

Step 1: define an objective function

Hyperopt provides a few levels of increasing flexibility/complexity when it comes to specifying an objective function to minimize. In the simplest case, an objective function is a Python function that accepts a single argument that stands for x (which can be an arbitrary object), and returns a single scalar value that represents the *loss* $(f(x))$ incurred by that argument.

So for a trivial example, if we want to minimize a quadratic function $q(x, y) := x^2 + y^2$ then we could define our objective `q` as follows:

```
def q (args) :
    x, y = args
    return x ** 2 + y ** 2
```

Although Hyperopt accepts objective functions that are more complex in both the arguments they accept and their return value, we will use this simple calling and return convention for the next few sections that introduce configuration spaces, optimization algorithms, and basic usage of the `fmin` interface. Later, as we explain how to use the Trials object to analyze search results, and how to search in parallel with a cluster, we will introduce different calling and return conventions.

Step 2: define a configuration space

A *configuration space* object describes the domain over which Hyperopt is allowed to search. If we want to search q over values of $x \in [0, 1]$, and values of $y \in \mathbb{R}$, then we can write our search space as:

```
from hyperopt import hp

space = [hp.uniform('x', 0, 1), hp.normal('y', 0, 1)]
```

Note that for both x *and* y we have specified not only the hard bound constraints, but also we have given Hyperopt an idea of what range of values for y to prioritize.

Step 3: choose a search algorithm

Assigning the `algo` keyword argument to `hyperopt.fmin` is recommended way to choose a search algorithm. Currently supported search algorithms are random search (`hyperopt.rand.suggest`), annealing (`hyperopt.anneal.suggest`), and TPE (`hyperopt.tpe.suggest`). There is an experimental Gaussian-process-based search algorithm available as well, which can be downloaded separately from https://github.com/hyperopt/hyperopt-gpsmbo. For example, to use random search on our search problem we can type:

```
from hyperopt import hp, fmin, rand, tpe, space_eval
best = fmin(q, space, algo=rand.suggest)
print space_eval(space, best)
```

Search algorithms can be complicated, and so they may have their own internal configuration parameters (hyper-hyperparameters) that control how they optimize the function at hand. The reason hyperopt exists is that hyper-hyperparameter defaults are more reliable than the default values for machine learning algorithm hyperparameters, but hyper-hyperparameters still exist. Hyperopt's search algorithms are created by global functions that use extra keyword arguments to override default hyper-hyperparameters values. For example, we can configure the TPE algorithm to transition from random sampling to guided search after ten initial jobs like this:

```
from functools import partial
from hyperopt import hp, fmin, tpe
algo = partial(tpe.suggest, n_startup_jobs=10)
best = fmin(q, space, algo=algo)
print space_eval(space, best)
```

To summarize, these are the steps to using Hyperopt: (1) implement an objective function that maps configuration points to a real-valued loss value, (2) define a configuration space of valid configuration points, and then (3) call `fmin` to search the space to optimize the objective function. The remainder of the paper describes (a) how to describe more elaborate configuration spaces, especially ones that enable more efficient search by expressing *conditional variables*, (b) how to analyze the results of a search as stored in a `Trials` object and (c) how to use a cluster of computers to search in parallel.

The API for actually implementing new search algorithms is beyond the scope of this article, but the interested reader is invited to study the source code of the `anneal` algorithm (anneal.py). This highly-documented search algorithm is meant primarily as an introduction to implementing search algorithms.

Configuration spaces

Part of what makes Hyperopt a good fit for optimizing machine learning hyperparameters is that it can optimize over general Python objects, not just e.g. vector spaces. Consider the simple function `w` below, which optimizes over dictionaries with `'type'` and either `'x'` or `'y'` keys:

```
def w(pos):
    if pos['use_var'] == 'x':
        return pos['x'] ** 2
    else:
        return math.exp(pos['y'])
```

To be efficient about optimizing `w` we must be able to (a) describe the kinds of dictionaries that `w` requires and (b) correctly associate `w`'s return value to the elements of `pos` that actually contributed to that return value. Hyperopt's configuration space description objects address both of these requirements. This section describes the nature of configuration space description objects, and how the description language can be extended with new expressions, and how the `choice` expression supports the creation of *conditional variables* that support efficient evaluation of structured search spaces of the sort we need to optimize `w`.

Configuration space primitives

A search space is a stochastic expression that always evaluates to a valid input argument for your objective function. A search space consists of nested function expressions. The stochastic expressions are the hyperparameters. (Random search is implemented by simply sampling these stochastic expressions.)

The stochastic expressions currently recognized by Hyperopt's optimization algorithms are in the `hyperopt.hp` module. The simplest kind of search spaces are ones that are not nested at all. For example, to optimize the simple function `q` (defined above) on the interval [0, 1], we could type `fmin(q, space=hp.uniform('a', 0, 1))`.

The first argument to `hp.uniform` here is the *label*. Each of the hyperparameters in a configuration space must be labeled like this with a unique string. The other hyperparameter distributions at our disposal as modelers are as follows:

`hp.choice(label, options)`. Returns one of the options, which should be a list or tuple. The elements of `options` can themselves be [nested] stochastic expressions. In this case, the stochastic choices that only appear in some of the options become *conditional* parameters.

`hp.pchoice(label, p_options)`. Return one of the `option` terms listed in `p_options`, a list of pairs `(prob, option)` in which the sum of all `prob` elements should sum to 1. The `pchoice` lets a user bias random search to choose some options more often than others.

`hp.uniform(label, low, high)`. Draws uniformly between `low` and `high`. When optimizing, this variable is constrained to a two-sided interval.

`hp.quniform(label, low, high, q)`. Drawn by `round(uniform(low, high) / q) * q`, Suitable for a discrete value with respect to which the objective is still somewhat smooth.

`hp.loguniform(label, low, high)`. Drawn by `exp(uniform(low, high))`. When optimizing, this variable is constrained to the interval $[e^{\text{low}}, e^{\text{high}}]$.

`hp.qloguniform(label, low, high, q)`. Drawn by `round(exp(uniform (low, high)) / q) * q`. Suitable for a discrete variable with respect to which the objective is smooth and gets smoother with the increasing size of the value.

`hp.normal(label, mu, sigma)`. Draws a normally-distributed real value. When optimizing, this is an unconstrained variable.

`hp.qnormal(label, mu, sigma, q)`. Drawn by `round(normal(mu, sigma) / q) * q`. Suitable for a discrete variable that probably takes a value around `mu`, but is technically unbounded.

`hp.lognormal(label, mu, sigma)`. Drawn by `exp(normal(mu, sigma))`. When optimizing, this variable is constrained to be positive.

`hp.qlognormal(label, mu, sigma, q)`. Drawn by `round(exp(normal(mu, sigma)) / q) * q`. Suitable for a discrete variable with respect to which the objective is smooth and gets smoother with the size of the variable, which is non-negative.

`hp.randint(label, upper)`. Returns a random integer in the range [0, *upper*). In contrast to `quniform` optimization algorithms should assume *no* additional correlation in the loss function between nearby integer values, as compared with more distant integer values (e.g. random seeds).

Structure in configuration spaces

Search spaces can also include lists and dictionaries. Using these containers make it possible for a search space to include multiple variables (hyperparameters). The following code fragment illustrates the syntax:

```
from hyperopt import hp

list_space = [
    hp.uniform('a', 0, 1),
    hp.loguniform('b', 0, 1)]
tuple_space = (
    hp.uniform('a', 0, 1),
    hp.loguniform('b', 0, 1))

dict_space = {
    'a': hp.uniform('a', 0, 1),
    'b': hp.loguniform('b', 0, 1)}
```

There should be no functional difference between using list and tuple syntax to describe a sequence of elements in a configuration space, but both syntaxes are supported for everyone's convenience.

Creating list, tuple and dictionary spaces as illustrated above is just one example of nesting. Each of these container types can be nested to form deeper configuration structures:

```
nested_space = [
    [  {'case': 1, 'a': hp.uniform('a', 0, 1)},
      {'case': 2, 'b': hp.loguniform('b', 0, 1)}],
    'extra literal string',
    hp.randint('r', 10)  ]
```

There is no requirement that list elements have some kind of similarity, each element can be any valid configuration expression. Note that Python values (e.g. numbers, strings and objects) can be embedded in the configuration space. These values will be treated as constants from the point of view of the optimization algorithms, but they will be included in the configuration argument objects passed to the objective function.

Sampling from a configuration space

The previous few code fragments have defined various configuration spaces. These spaces are not objective function arguments yet, they are simply a description of *how to sample* objective function arguments. You can use the routines in hyperopt.pyll.stochasticto sample values from these configuration spaces.

```
from hyperopt.pyll.stochasti import sample

print sample(list_space)
#  =>  [0.13, .235]

print sample(nested_space)
#  =>  [[{'case': 1, 'a', 0.12`}, {'case': 2, 'b': 2.3}],
```

(Continued.)

```
#       ’extra_literal_string’,
#       3]
```

Note that the labels of the random configuration variables have no bearing on the sampled values themselves, the labels are only used internally by the optimization algorithms. Later when we look at the `trials` parameter to `fmin` we will see that the labels are used for analyzing search results too. For now though, simply note that the labels are not for the objective function.

Deterministic expressions in configuration spaces

It is also possible to include deterministic expressions within the description of a configuration space. For example, we can write

```
from hyperopt.pyll import scope

def foo(x):
    return str(x) * 3

expr_space = {
    ’a’: 1 + hp.uniform(’a’, 0, 1),
    ’b’: scope.minimum(hp.loguniform(’b’, 0, 1), 10),
    ’c’: scope.call(foo, args=(hp.randint(’c’, 5),)),
    }
```

The `hyperopt.pyll` submodule implements an expression language that stores this logic in a symbolic representation. Significant processing can be carried out by these intermediate expressions. In fact, when you call `fmin(f, space)`, your arguments are quickly combined into a single objective-and-configuration evaluation graph of the form: `scope.call(f, space)`. Feel free to move computations between these intermediate functions and the final objective function as you see fit in your application.

You can add new functions to the `scope`object with the `define`decorator:

```
from hyperopt.pyll import scope

@scope.define
def foo(x):
    return str(x) * 3

# -- This will print "000"; foo is called as usual.
print foo(0)

expr_space = {
    ’a’: 1 + hp.uniform(’a’, 0, 1),
    ’b’: scope.minimum(hp.loguniform(’b’, 0, 1), 10),
    ’c’: scope.foo(hp.randint(’cbase’, 5)), }

# -- This will draw a sample by running foo(x)
# on a random integer x.
print sample(expr_space)
```

Note that functions used in configuration space descriptions must be serializable (with a pickle module) in order to be compatible with parallel search (discussed below).

Defining conditional variables with `choice` *and* `pchoice`

Having introduced nested configuration spaces, it is worth coming back to the `hp.choice` and `hp.pchoice` hyperparameter types. An `hp.choice(label, options)` hyperparameter *chooses* one of the options that you provide, where the `options` must be a list. We can use `choice` to define an appropriate configuration space for the `w`objective function (introduced in section configuration spaces).

```
w_space = hp.choice('case', [
      {'use_var': 'x', 'x': hp.normal('x', 0, 1)},
      {'use_var': 'y', 'y': hp.uniform('y', 1, 3)}])

print sample(w_space)
# ==>  {'use_var': 'x', 'x': -0.89}

print sample(w_space)
# ==>  {'use_var': 'y', 'y': 2.63}
```

Recall that in `w`, the `'y'` key of the configuration is not used when the `'use_var'` value is `'x'`. Similarly, the `'x'` key of the configuration is not used when the `'use_var'` value is `'y'`. The use of `choice` in the `w_space` search space reflects the conditional usage of keys `'x'` and `'y'` in the `w` function. We have used the `choice` variable to define a space that never has more variables than is necessary.

The choice variable here plays more than a cosmetic role; it can make optimization much more efficient. In terms of `w` and `w_space`, the choice node prevents `y` for being *blamed* (in terms of the logic of the search algorithm) for poor performance when `'use_var'` is `'x'`, or *credited* for good performance when `'use_var'` is `'x'`. The choice variable creates a special node in the expression graph that prevents the conditionally unnecessary part of the expression graph from being evaluated at all. During optimization, similar special-case logic prevents any association between the return value of the objective function and irrelevant hyperparameters (ones that were not chosen, and hence not involved in the creation of the configuration passed to the objective function).

The `hp.pchoice` hyperparameter constructor is similar to `choice` except that we can provide a list of probabilities corresponding to the options, so that random sampling chooses some of the options more often than others.

```
w_space_with_probs = hp.pchoice('case', [
    (0.8, {'use_var': 'x',
           'x': hp.normal('x', 0, 1)}),
    (0.2, {'use_var': 'y',
           'y': hp.uniform('y', 1, 3)})])
```

Using the `w_space_with_probs` configuration space expresses to `fmin` that we believe the first case (using `'x'`) is five times as likely to yield an optimal configuration than the second case. If your objective function only uses a subset of the configuration space on any given evaluation, then you should use choice or pchoice hyperparameter variables to communicate that pattern of inter-dependencies to fmin.

Sharing a configuration variable across choice branches

When using choice variables to divide a configuration space into many mutually exclusive possibilities, it can be natural to re-use some configuration variables across a few of those possible branches. Hyperopt's configuration space supports this in a natural way, by allowing the objects to appear in multiple places within a nested configuration expression. For example, if we wanted to add a randint choice to the returned dictionary that did not depend on the 'use_var' value, we could do it like this:

```
c = hp.randint('c', 10)

w_space_c = hp.choice('case', [
    {'use_var': 'x',
    'x': hp.normal('x', 0, 1),
    'c': c},
    {'use_var': 'y',
    'y': hp.uniform('y', 1, 3),
    'c': c}])
```

Optimization algorithms in Hyperopt would see that c is used regardless of the outcome of the choice value, so they would correctly associate c with all evaluations of the objective function.

Configuration example: sklearn **classifiers**

To see how we can use these mechanisms to describe a more realistic configuration space, let's look at how one might describe a set of classification algorithms in [sklearn]. This example is done without using the Hyperopt-Sklearn project, to indicate how Hyperopt can be used in general.

```
from hyperopt import hp
from hyperopt.pyll import scope
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier\
    as DTree

scope.define(GaussianNB)
scope.define(SVC)
scope.define(DTree, name='DTree')

C = hp.lognormal('svm_C', 0, 1)
space = hp.pchoice('estimator', [
    (0.1, scope.GaussianNB()),
    (0.2, scope.SVC(C=C, kernel='linear')),
    (0.3, scope.SVC(C=C, kernel='rbf',
```

(Continued.)

```
      width=hp.lognormal(’svm_rbf_width’, 0, 1),
      )),
    (0.4, scope.DTree(
      criterion=hp.choice(’dtree_criterion’,
        [’gini’, ’entropy’]),
      max_depth=hp.choice(’dtree_max_depth’,
        [None, hp.qlognormal(’dtree_max_depth_N’,
          2, 2, 1)],
    ])
```

This example illustrates nesting, the use of custom expression types, the use of `pchoice` to indicate independence among configuration branches, several numeric hyperparameters, a discrete hyperparameter (the Dtree criterion), and a specification of our prior preference among the four possible classifiers. At the top level we have a `pchoice` between four Scikit-learn algorithms: Naive Bayes (NB), a SVM using a linear kernel, an SVM using a Radial Basis Function (`’rbf’`) kernel, and a Dtree. The result of evaluating the configuration space is actually a Scikit-learn estimator corresponding to one of the three possible branches of the top-level choice. Note that the example uses the same C variable for both types of SVM kernel. This is a technique for injecting domain knowledge to assist with search; if each of the SVMs prefers roughly the same value of C then this will buy us some search efficiency, but it may hurt search efficiency if the two SVMs require very different values of C. Note also that the hyperparameters all have unique names; it is tempting to think they should be named automatically by their path to the root of the configuration space, but the configuration space is not a tree (consider the `C` above). These names are also invaluable in analyzing the results of search after `fmin` has been called, as we will see in the next section, on the `Trials` object.

The trials object

The `fmin` function returns the best result found during search, but can also be useful to analyze all of the trials evaluated during search. Pass a `trials` argument to `fmin` to retain access to all of the points accessed during search. In this case the call to `fmin` proceeds as before, but by passing in a trials object directly, we can inspect all of the return values that were calculated during the experiment.

```
from hyperopt import (hp, fmin, space_eval,
  Trials)
trials = Trials()
best = fmin(q, space, trials=trials)
print trials.trials
```

Information about all of the points evaluated during the search can be accessed via attributes of the `trials` object. The `.trials` attribute of a Trials object (`trials.trials` here) is a list with an element for every function evaluation made by `fmin`. Each element is a dictionary with at least keys:

`’tid’`: value of type int trial identifier of the trial within the search `'results'`: value of type dict dict with `'loss'`, `'status'`, and other information returned by the objective function (see below for details)

`'misc'` value of dict with keys `'idxs'` and `'vals'` compressed representation of hyperparameter values

This trials object can be pickled, analyzed with your own code, or passed to Hyperopt's plotting routines (described below).

Trial results: more than just the loss

Often when evaluating a long-running function, there is more to save after it has run than a single floating point loss value. For example there may be statistics of what happened during the function evaluation, or it might be expedient to pre-compute results to have them ready if the trial in question turns out to be the best-performing one.

Hyperopt supports saving extra information alongside the trial loss. To use this mechanism, an objective function must return a dictionary instead of a float. The returned dictionary must have keys `'loss'` and `'status'`. The status should be either `STATUS_OK` or `STATUS_FAIL` depending on whether the loss was computed successfully or not. If the status is `STATUS_OK`, then the loss must be the objective function value for the trial. Writing a quadratic `f(x)` function in this dictionary-returning style, it might look like:

```
import time
from hyperopt import fmin, Trials
from hyperopt import STATUS_OK, STATUS_FAIL

def f(x):
    try:
        return {'loss': x ** 2,
                'time': time.time(),
                'status': STATUS_OK }
    except Exception, e:
        return {'status': STATUS_FAIL,
                'time': time.time(),
                'exception': str(e)}
trials = Trials()
fmin(f, space=hp.uniform('x', -10, 10),
    trials=trials)
print trials.trials[0]['results']
print trials.argmin
```

An objective function can use just about any keys to store auxiliary information, but there are a few special keys that are interpreted by Hyperopt routines:

`'loss_variance'`: type float variance in a stochastic objective function,

`'true_loss'`: type float if you pre-compute a test error for a validation error loss, store it here so that Hyperopt plotting routines can find it,

`'true_loss_variance'`: type float variance in test error estimator,

`'attachments'`: type dict short (string) keys with potentially long (string) values.

The `'attachments'` mechanism is primarily useful for reducing data transfer times when using the `MongoTrials` trials object (discussed below) in the context of parallel function evaluation. In that case, any strings longer than a few megabytes actually *have* to be placed in the attachments because of limitations in certain versions of the mongodb database format. Another important consideration when using `MongoTrials` is that the entire dictionary returned from the objective function must be JSON-compatible. JSON allows for only strings, numbers, dictionaries, lists, tuples, and date-times.

HINT: to store NumPy arrays, serialize them to a string, and consider storing them as attachments.

Parallel evaluation with a cluster

Hyperopt has been designed to make use of a cluster of computers for faster search. Of course, parallel evaluation of trials sits at odds with SMBO. Evaluating trials in parallel means that efficiency per function evaluation will suffer (to an extent that is difficult to assess *a priori*), but the improvement in efficiency as a function of wall time can make the sacrifice worthwhile.

Hyperopt supports parallel search via a special trials type called `MongoTrials`. To set up a parallel search process, use `MongoTrials` instead of `Trials` in the `fmin` call:

```
from hyperopt import fmin
from hyperopt.mongo import MongoTrials
trials = MongoTrials('mongo://host:port/fmin_db/')
best = fmin(q, space, trials=trials)
```

When we construct a `MongoTrials` object, we must specify a running *mongod* database [mongodb] for inter-process communication between the `fmin` producer-process and *worker* processes, which act as the consumers in a producer-consumer processing model. If you simply type the code fragment above, you may find that it either crashes (if no mongod is found) or hangs (if no worker processes are connected to the same database). When used with `MongoTrials` the `fmin` call simply enqueues configurations and waits until they are evaluated. If no workers are running, `fmin` will block after enqueing one trial. To run `fmin` with `MongoTrials` requires that you:

(1) ensure that mongod is running on the specified host and port,

(2) choose a database name to use for a *particular fmin call*, and

(3) start one or more *hyperopt-mongo-worker* processes.

There is a generic *hyperopt-mongo-worker* script in Hyperopt's `scripts` subdirectory that can be run from a command line like this:

```
hyperopt-mongo-worker —mongo=host:port/db
```

To evaluate multiple trial points in parallel, simply start multiple scripts in this way that all work on the same database.

Note that mongodb databases persist until they are deleted, and `fmin` will never delete things from mongodb. If you call `fmin` using a particular database one day, stop the search, and start it again later, then `fmin` will continue where it left off.

HINT: results in a MongoTrials database can be visualized in real-time by querying the Mongo database, or by creating a `MongoTrials` object and calling `MongoTrials.refresh()`.

The ctrl object for realtime communication with MongoDB

When running a search in parallel, you may wish to provide your objective function with a handle to the mongodb database used by the search. This mechanism makes it possible for objective functions to:

- update the database with partial results,
- to communicate with concurrent processes and
- even to enqueue new configuration points.

This is an advanced usage of Hyperopt, but it is supported via syntax like the following:

```
from hyperopt import pyll

@hyperopt.fmin_pass_expr_memo_ctrl
def realtime_objective(expr, memo, ctrl):
    config = pyll.rec_eval(expr, memo=memo)
    # .. config is a configuration point
    # .. ctrl can be used to interact with database
    return {'loss': f(config),
            'status': STATUS_OK, ...}
```

The `fmin_pass_expr_memo_ctrl` decorator tells `fmin` to use a different calling convention for the objective function, in which internal objects `expr`, `memo` and `ctrl` are exposed to the objective function. The `expr` the configuration space, the `memo` is a dictionary mapping nodes in the configuration space description graph to values for those nodes (most importantly, values for the hyperparameters). The recursive evaluation function `rec_eval` computes the configuration point from the values in the `memo` dictionary. The `config` object produced by `rec_eval` is what would normally have been passed as the argument to the objective function. The `ctrl` object is an instance of `hyperopt.Ctrl`, and it can be used to to communicate with the trials object being used by `fmin`. It is possible to use a `ctrl` object with a (sequential) `Trials` object, but it is most useful when used with `MongoTrials`.

To summarize, Hyperopt can be used both purely sequentially, as well as *broadly sequentially* with multiple current candidates under evaluation at a time. In the parallel case, mongodb is used for inter-process communication and doubles as a persistent storage mechanism for post-hoc analysis. Parallel search can be done with the same objective functions as the ones used for sequential search, but users wishing to take advantage of asynchronous evaluation in the parallel case can do so by using a lower-level calling convention for their objective function.

Case study: Hyperopt-Sklearn

Relative to DBNs and convnets, algorithms such as SVMs and Random Forests (RFs) have a small-enough number of hyperparameters that manual tuning and grid or random search often provides satisfactory results. Taking a step back though, there is often no particular reason to use either an SVM or an RF when they are both computationally viable. A model-agnostic practitioner may simply prefer to go with the one that provides greater accuracy. In this light, *the choice of classifier can be seen as hyperparameter* alongside the C-value in the SVM and the max-tree-depth of the RF. Indeed the choice and configuration of *preprocessing* components may likewise be seen as part of the model selection / hyperparameter optimization problem. The combinatorial space of preprocessing options and various classifiers is difficult to do manually, even with the assistance of a compute cluster performing grid or random search.

The Auto-Weka project [Tho13] was the first to show that an entire library of machine learning approaches (Weka [Hal09]) can be searched within the scope of a single run of hyperparameter tuning. However, Weka is a GPL-licensed Java library, and was not written with scalability in mind, so we feel there is a need for alternatives to Auto-Weka. Scikit-learn [Ped11] is another library of machine learning algorithms. Is written in Python (with many modules in C for greater speed), and is BSD-licensed. Scikit-learn is widely used in the scientific Python community and supports many machine learning application areas.

In the following sections we introduce Hyperopt-Sklearn: a project that brings the benefits of automatic algorithm configuration to users of Python and Scikit-learn. Hyperopt-Sklearn uses Hyperopt to describe a search space over possible configurations of Scikit-learn components, including preprocessing and classification modules. The next section describes our configuration space of 6 classifiers and 5 preprocessing modules that encompasses a strong set of classification systems for dense and sparse feature classification (of images and text). This is followed by experimental evidence that search over this space is viable, meaningful, and effective. The final sections present a discussion of the results, and directions for future work.

Scikit-learn model selection as a search problem

Model selection is the process of estimating which machine learning model performs best from among a possibly infinite set of possibilities. As an optimization problem, the search domain is the set of valid assignments to the configuration parameters (hyperparameters) of the machine learning model, and the objective function is typically cross-validation, the negative degree of success on held-out examples. Practitioners usually address this optimization by hand, by grid search, or by random search. In this paper we discuss solving it with the Hyperopt optimization library. The basic approach is to set up a search space with random variable hyperparameters, use Scikit-learn to implement the objective function that performs model training and model validation, and use Hyperopt to optimize the hyperparamters.

Scikit-learn includes many algorithms for classification (classifiers), as well as many algorithms for preprocessing data into the vectors expected by classification algorithms. Classifiers include for example, K-Neighbors, SVM and RF algorithms. Preprocessing algorithms include things like component-wise Z-scaling (Normalizer) and principle components analysis (PCA). A full classification algorithm typically includes a series of preprocessing steps followed by a classifier. For this reason, Scikit-learn provides a *pipeline* data structure to represent and use a sequence of preprocessing steps and a classifier as if they were just one component (typically with an API similar to the classifier). Although Hyperopt-Sklearn does not formally use Scikit-learn's pipeline object, it provides related functionality.

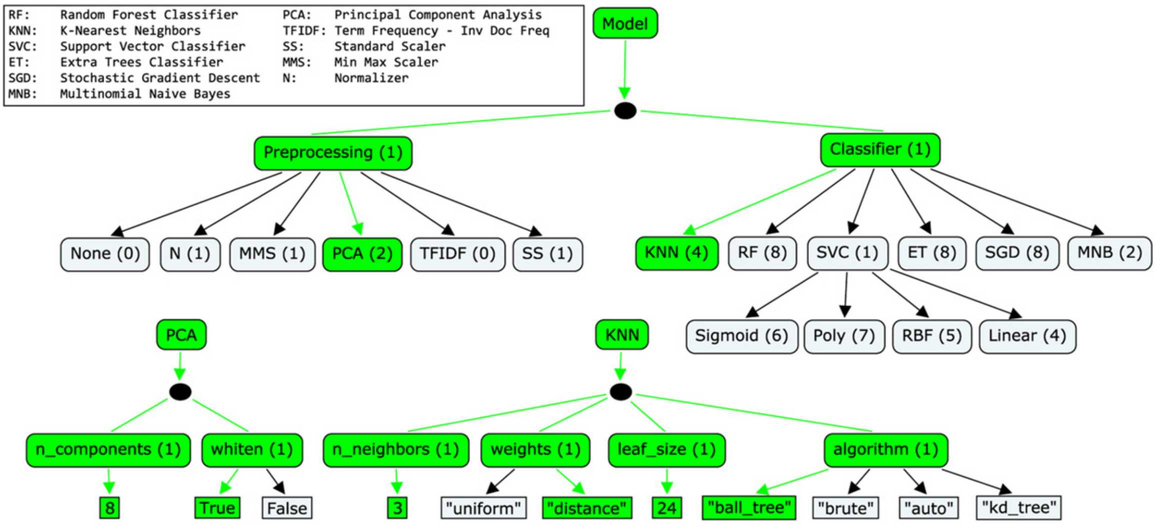

Figure 1. Hyeropt-Sklearns full search space ('any classifier') consists of a (preprocessing, classsifier) pair. There are six possible preprocessing modules and six possible classifiers. Choosing a model within this configuration space means choosing paths in an ancestral sampling process. The highlighted green edges and nodes represent a (PCA, K-Nearest Neighbor) model. The number of active hyperparameters in a model is the sum of parenthetical numbers in the selected boxes. For the PCA + KNN combination, seven hyperparameters are activated.

Hyperopt-Sklearn provides a parameterization of a *search space* over pipelines, that is, of sequences of preprocessing steps and classifiers.

The configuration space we provide includes six preprocessing algorithms and seven classification algorithms. The full search space is illustrated in figure 1. The preprocessing algorithms were (by class name, followed by n. hyperparameters + n. unused hyperparameters): `PCA(2)`, `StandardScaler(2)`, `MinMaxScaler(1)`, `Normalizer(1)`, `None`, and `TF-IDF(0+9)`. The first four preprocessing algorithms were for dense features. PCA performed whitening or non-whitening PCO. The `StandardScaler`, `MinMaxScaler`, and `Normalizer` did various feature-wise affine transforms to map numeric input features onto values near 0 and with roughly unit variance. The `TF-IDF` preprocessing module performed feature extraction from text data. The classification algorithms were (by class name (used + unused hyperparameters)): `SVC(23)`, `KNN(4+5)`, `RandomForest(8)`, `ExtraTrees(8)`, `SGD(8+4)`, and `MultinomialNB(2)` . The `SVC` module is a fork of LibSVM, and our wrapper has 23 hyperparameters because we treated each possible kernel as a different classifier, with its own set of hyperparameters: Linear(4), RBF(5), Polynomial(7) and Sigmoid(6). In total, our parameterization has 65 hyperparameters: 6 for preprocessing and 53 for classification. The search space includes 15 boolean variables, 14 categorical, 17 discrete, and 19 real-valued variables.

Although the total number of hyperparameters is large, the number of *active* hyperparameters describing any one model is much smaller: a model consisting of `PCA` and a `RandomForest` for example, would have only 12 active hyperparameters (1 for the choice of preprocessing, two internal to PCA, 1 for the choice of classifier and eight internal to the RF). Hyperopt description language allows us to differentiate between *conditional* hyperparameters (which must always be assigned) and *non-conditional* hyperparameters (which may remain unassigned when they would be unused). We make use of this mechanism extensively so that

Hyperopt's search algorithms do not waste time learning by trial and error that e.g. RF hyperparameters have no effect on SVM performance. Even internally within classifiers, there are instances of conditional parameters: `KNN` has conditional parameters depending on the distance metric, and `LinearSVC` has three binary parameters (`loss`, `penalty` and `dual`) that admit only four valid joint assignments. We also included a blacklist of (preprocessing, classifier) pairs that did not work together, e.g. PCA and MinMaxScaler were incompatible with MultinomialNB, TF-IDF could only be used for text data, and the tree-based classifiers were not compatible with the sparse features produced by the TF-IDF preprocessor. Allowing for a ten-way discretization of real-valued hyperparameters, and taking these conditional hyperparameters into account, a grid search of our search space would still require an infeasible number of evalutions (on the order of 10^{12}).

Finally, the search space becomes an optimization problem when we also define a scalar-valued search *objective*. Hyperopt-Sklearn uses Scikit-learn's *score* method on *validation data* to define the search criterion. For classifiers, this is the so-called 'Zero-One Loss': the number of correct label predictions among data that has been withheld from the data set used for training (and also from the data used for testing *after* the model selection search process).

Example usage

Following Scikit-learn's convention, Hyperopt-Sklearn provides an `Estimator` class with a `fit` method and a `predict` method. The `fit` method of this class performs hyperparameter optimization, and after it has completed, the `predict` method applies the best model to test data. Each evaluation during optimization performs training on a large fraction of the training set, estimates test set accuracy on a validation set and returns that validation set score to the optimizer. At the end of search, the best configuration is retrained on the whole data set to produce the classifier that handles subsequent `predict` calls.

One of the important goals of Hyperopt-Sklearn is that it is easy to learn and to use. To facilitate this, the syntax for fitting a classifier to data and making predictions is very similar to Scikit-learn. Here is the simplest example of using this software.

```
from hpsklearn import HyperoptEstimator
# Load data ({train,test}_{data,label})
# Create the estimator object
estim = HyperoptEstimator()
# Search the space of classifiers and preprocessing
# steps and their respective hyperparameters in
# Scikit-Learn to fit a model to the data
estim.fit(train_data, train_label)
# Make a prediction using the optimized model
prediction = estim.predict(unknown_data)
# Report the accuracy of the classifier
# on a given set of data
score = estim.score(test_data, test_label)
# Return instances of the classifier and
# preprocessing steps
model = estim.best_model()
```

The `HyperoptEstimator` object contains the information of what space to search as well as how to search it. It can be configured to use a variety of hyperparameter search algorithms and also supports using a combination of algorithms. Any algorithm that supports the same interface as the algorithms in Hyperopt can be used here. This is also where you, the user, can specify the maximum number of function evaluations you would like to be run as well as a timeout (in seconds) for each run.

```
from hpsklearn import HyperoptEstimator
from hyperopt import tpe
estim = HyperoptEstimator(algo=tpe.suggest,
                          max_evals=150,
                          trial_timeout=60)
```

All of the components available to the user can be found in the `components.py` file. A complete working example of using Hyperopt-Sklearn to find a model for the 20 newsgroups data set is shown below.

```
from hpsklearn import HyperoptEstimator, tfidf, \
                      any_sparse_classifier
from sklearn.datasets import fetch_20newsgroups
from hyperopt import tpe
import numpy as np
# Download data and split training and test sets
train = fetch_20newsgroups(subset=’train’)
test = fetch_20newsgroups(subset=’test’)
X_train = train.data
y_train = train.target
X_test = test.data
y_test = test.target
estim = HyperoptEstimator(
         classifier=any_sparse_classifier(’clf’),
         preprocessing=[tfidf(’tfidf’)],
         algo=tpe.suggest,
         trial_timeout=180)
estim.fit(X_train, y_train)
print(estim.score(X_test, y_test))
print(estim.best_model())
```

Experiments

We conducted experiments on three data sets to establish that Hyperopt-Sklearn can find accurate models on a range of data sets in a reasonable amount of time. Results were collected on three data sets: MNIST, 20-newsgroups and convex shapes. MNIST is a well-known data set of 70 K 28 × 28 greyscale images of hand-drawn digits [Lec98]. 20-newsgroups is a 20-way classification data set of 20 K newsgroup messages ([Mit96], we did not remove the headers for our experiments). convex shapes is a binary classification task of distinguishing pictures of convex white-colored regions in small (32 × 32) black-and-white images [Lar07].

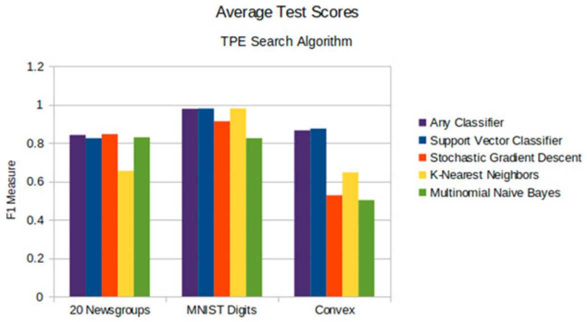

Figure 2. For each data set, searching the full configuration space (‘any classifier’) delivered performance approximately on par with a search that was restricted to the best classifier type. (Best viewed in color.)

Table 1. Hyperopt-Sklearn scores relative to selections from literature on the three data sets used in our experiments. On MNIST, Hyperopt-Sklearn is one of the best-scoring methods that does not use image-specific domain knowledge (these scores and others may be found at http://yann.lecun.com/exdb/mnist/). On 20 newsgroups, Hyperopt-Sklearn is competitive with similar approaches from the literature (scores taken from [Gua09]). In the 20 newsgroups data set, the score reported for Hyperopt-Sklearn is the weighted-average F1 score provided by Scikit-learn. The other approaches shown here use the macro-average F1 score. On convex shapes, Hyperopt-Sklearn outperforms previous automatic algorithm configuration approaches [Egg13] and manual tuning [Lar07].

MNIST		20 newsgroups		Convex shapes	
Approach	Accuracy	Approach	F-Score	Approach	Accuracy
Committee of convnets	99.8%	CFC	0.928	**hyperopt-sklearn**	**88.7%**
hyperopt-sklearn	**98.7%**	**hyperopt-sklearn**	**0.856**	hp-dbnet	84.6%
libSVM grid search	98.6%	SVMTorch	0.848	dbn-3	81.4%
Boosted trees	98.5%	LibSVM	0.843		

Figure 2 shows that there was no penalty for searching broadly. We performed optimization runs of up to 300 function evaluations searching the entire space, and compared the quality of solution with specialized searches of specific classifier types (including best known classifiers).

Discussion

Table 1 lists the test set scores of the best models found by cross-validation, as well as some points of reference from previous work. Hyperopt-Sklearn’s scores are relatively good on each data set, indicating that with Hyperopt-Sklearn’s parameterization, Hyperopt’s optimization algorithms are competitive with human experts.

The model with the best performance on the MNIST Digits data set uses deep artificial neural networks. Small receptive fields of convolutional winner-take-all neurons build up the large network. Each neural column becomes an expert on inputs preprocessed in different ways, and the average prediction of 35 deep neural columns to come up with a single final prediction [Cir12]. This model is much more advanced than those available in Scikit-learn. The previously best known model in the Scikit-learn search space is a radial-basis SVM on centered data that scores 98.6%, and Hyperopt-Sklearn matches that performance [MNIST].

The CFC model that performed quite well on the 20 newsgroups document classification data set is a Class–Feature–Centroid classifier. Centroid approaches are typically inferior to an SVM, due to the centroids found during training being far from the optimal location. The CFC method reported here uses a centroid built from the inter-class term index and the inner-class term index. It uses a novel combination of these indices along with a denormalized cosine measure to calculate the similarity score between the centroid and a text vector [Gua09]. This style of model is not currently implemented in Hyperopt-Sklearn, and our experiments suggest that existing Hyperopt-Sklearn components cannot be assembled to match its level of performance. Perhaps when it is implemented, Hyperopt may find a set of parameters that provides even greater classification accuracy.

On the convex shapes data set, our Hyperopt-Sklearn experiments revealed a more accurate model than was previously believed to exist in any search space, let alone a search space of such standard components. This result underscores the difficulty and importance of hyperparameter search.

Ongoing and future work

Hyperopt is the subject of ongoing and planned future work in the algorithms that it provides, the domains that it covers, and the technology that it builds on.

Related Bayesian optimization software such as Frank Hutter *et al*'s [SMAC], and Jasper Snoek's [Spearmint] implement state-of-the-art algorithms that are different from the TPE algorithm currently implemented in Hyperopt. Questions about which of these algorithms performs best in which circumstances, and over what search budgets remain topics of active research. One of the first technical milestones on the road to answering those research questions is to make each of those algorithms applicable to common search problems.

Hyperopt was developed to support research into deep learning [Ber11] and computer vision [Ber13a]. Corresponding projects [hp-dbn] and [hp-convnet] have been made public on Github to illustrate how Hyperopt can be used to define and optimize large-scale hyperparameter optimization problems.

With regards to implementation decisions in Hyperopt, several people have asked about the possibility of using IPython instead of mongodb to support parallelism. This would allow us to build on IPython's cluster management interface, and relax the constraint that objective function results be JSON-compatible. If anyone implements this functionality, a pull request to Hyperopt's master branch would be most welcome.

Hyperopt-Sklearn provides many opportunities for future work: more classifiers and preprocessing modules could be included in the search space, and there are more ways to combine even the existing components. Other types of data require different preprocessing, and other prediction problems exist beyond classification. In expanding the search space, care must be taken to ensure that the benefits of new models outweigh the greater difficulty of searching a larger space. There are some parameters that Scikit-learn exposes that are more implementation

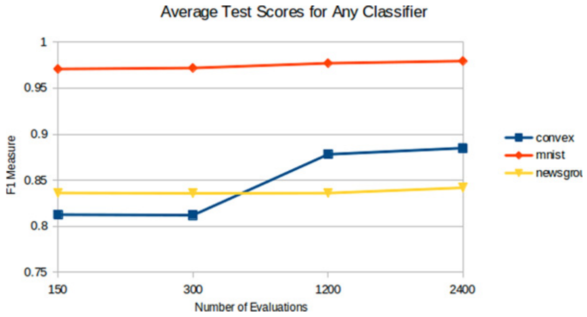

Figure 3. Using Hyperopts Anneal search algorithm, increasing the number of function evaluations from 150 to 2400 lead to a modest improvement in accuracy on 20 newsgroups and MNIST, and a more dramatic improvement on convex shapes. We capped evaluations to 5 min each so 300 evaluations took between 12 and 24 h of wall time.

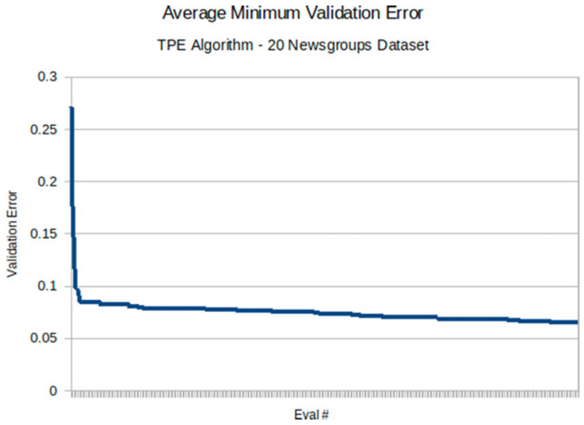

Figure 4. TPE makes gradual progress on 20 newsgroups over 300 iterations. The results are averaged over nine trials.

details than actual hyperparameters that affect the fit (such as `algorithm` and `leaf_size` in the KNN model). Care should be taken to identify these parameters in each model and they may need to be treated differently during exploration.

It is possible for a user to add their own classifier to the search space as long as it fits the Scikit-learn interface. This currently requires some understanding of how Hyperopt-Sklearn's code is structured and it would be nice to improve the support for this so minimal effort is required by the user. The user may also specify alternate scoring methods besides just accuracy and F-measure, as there can be cases where these are not best suited to the particular problem.

We have shown here that Hyperopt's random search, annealing search, and TPE algorithms make Hyperopt-Sklearn viable, but the slow convergence in e.g. Figures 3 and 4 suggests that other optimization algorithms might be more call-efficient. The development of Bayesian optimization algorithms is an active research area, and we look forward to looking at how other search algorithms interact with Hyperopt-Sklearn's search spaces. Hyperparameter optimization opens up a new art of matching the parameterization of search spaces to the strengths of search algorithms.

Computational wall time spent on search is of great practical importance, and Hyperopt-Sklearn currently spends a significant amount of time evaluating points that are un-promising. Techniques for recognizing bad performers early could speed up search enormously [Dom14, Swe14]. Relatedly, Hyperopt-Sklearn currently lacks support for K-fold cross-validation. In that setting, it will be crucial to follow SMAC in the use of racing algorithms to skip un-necessary folds.

Summary and further reading

Hyperopt is a Python library for SMBO that has been designed to meet the needs of machine learning researchers performing hyperparameter optimization. It provides a flexible and powerful language for describing search spaces, and supports scheduling asynchronous function evaluations for evaluation by multiple processes and computers. It is BSD-licensed and available for download from PyPI and Github. Further documentation is available at [http://jaberg.github.com/hyperopt].

Acknowledgments

Thanks to Nicolas Pinto for some influential design advice, Hristijan Bogoevski for early drafts of a hyperopt-to-scikit-learn bridge and to many users who have contributed feedback. This project has been supported by the Rowland Institute of Harvard, the National Science Foundation (IIS 0963668), the NSERC Banting Fellowship program, the NSERC Engage Program and by D-Wave Systems.

References

[BB12] Bergstra J and Bengio Y 2012 Random search for hyperparameter optimization *J. Mach. Learn. Res.* **13** 281–305

[Brochu10] Brochu E 2010 Interactive Bayesian optimization: learning parameters for graphics and animation *PhD Thesis* University of British Columbia

[Ber11] Bergstra J, Bardenet R, Bengio Y and Kéggl B 2011 Algorithms for hyperparameter optimization *NIPS* **24** 2546–54

[Ber13a] Bergstra J, Yamins D and Cox D D 2013 Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures *Proc. ICML*

[Ber13b] Bergstra J, Yamins D and Cox D D 2013 Hyperopt: a Python library for optimizing the hyperparameters of machine learning algorithms *Proc. SciPy 2013* pp 13–20

[Ber14] Bergstra J, Komer B, Eliasmith C and Warde-Farley D 2014 Preliminary evaluation of hyperopt algorithms on HPOLib *ICML AutoML Workshop*

[Cir12] Ciresan D, Meier U and Schmidhuber J 2012 Multi-column deep neural networks for image classification *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* pp 3642–9

[Dom14] Domhan T, Springenberg T and Hutter F 2014 Extrapolating learning curves of deep neural networks *ICML AutoML Workshop*

[Egg13] Eggensperger K, Feurer M, Hutter F, Bergstra J, Snoek J, Hoos H and Leyton-Brown K 2013 Towards an empirical foundation for assessing Bayesian optimization of hyperparameters *NIPS Workshop on Bayesian Optimization in Theory and Practice*

[Gua09] Guan H, Zhou J and Guo M 2009 A class-feature-centroid classifier for text categorization *Proc. 18th Int. Conf. on World Wide Web* pp 201–10 ACM

[Hal09] Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P and Witten I H 2009 The WEKA data mining software: an update *ACM SIGKDD Explorations Newsletter* **11** 10–18

[Hut11] Hutter F, Hoos H and Leyton-Brown K 2011 Sequential model-based optimization for general algorithm configuration LION-5 *Extended Version as UBC Tech Report* TR-2010-10

[Kom14] Komer B, Bergstra J and Eliasmith C 2014 Hyperopt-Sklearn: automatic hyperparameter configuration for Scikit-learn *Proc. SciPy 2014* (forthcoming)

[Lar07] Larochelle H, Erhan D, Courville A, Bergstra J and Bengio Y 2007 An empirical evaluation of deep architectures on problems with many factors of variation *ICML* pp 473–80

[Lec98] LeCun Y, Bottou L, Bengio Y and Haffner P 1998 Gradient-based learning applied to document recognition *Proc. IEEE* 86 2278–324

[Mit96] Mitchell T 1996 *Newsgroups Data Set* **20** (http://qwone.com/jason/20Newsgroups/)

[MNIST] The MNIST Database of handwritten digits: http://yann.lecun.com/exdb/mnist/

[Ped11] Pedregosa F *et al* 2011 Scikit-learn: machine learning in Python *J. Mach. Learn. Res.* **12** 2825–30

[Sno12] Snoek J, Larochelle H and Adams R P 2012 Practical Bayesian optimization of machine learning algorithms *Proc. NIPS*

[Swe14] Swersky K, Snoek J and Adams R P 2014 Freeze-thaw Bayesian optimization arXiv:1406.3896

[Tho13] Thornton C, Hutter F, Hoos H H and Leyton-Brown K 2013 Auto-WEKA: automated selection and hyper-parameter optimization of classification algorithms *Proc. KDD*

[Hyperopt] http://jaberg.github.com/hyperopt

[hp-dbn] http://github.com/jaberg/hyperopt-dbn

[hp-sklearn] http://github.com/jaberg/hyperopt-sklearn

[hp-convnet] http://github.com/jaberg/hyperopt-convnet

[mongodb] http://mongodb.org

[sklearn] http://scikit-learn.org

[Spearmint] http://cs.toronto.edu/~jasper/software.html

[SMAC] http://cs.ubc.ca/labs/beta/Projects/SMAC/#software