## Contents

1 Quick setup ........................................... 3

2 Developer Guide ...................................... 5

3 Contributors ......................................... 7

4 Indices and tables .................................... 9
  4.1 Installing Optunity ................................. 9
  4.2 User Guide ........................................ 12
  4.3 Examples .......................................... 24
  4.4 Notebooks ......................................... 70
  4.5 Using Other Environments ...................... 70
  4.6 Optunity API ..................................... 77

Bibliography ............................................. 123

Python Module Index ................................. 125
Optunity is a library containing various optimizers for hyperparameter tuning. Hyperparameter tuning is a recurrent problem in many machine learning tasks, both supervised and unsupervised. This package provides several distinct approaches to solve such problems including some helpful facilities such as cross-validation and a plethora of score functions.

### Getting started

- *Installation*
- *User guide*
- *Report a problem*

### Obtaining Optunity

- at PyPI (releases)
- at GitHub (development)

From an optimization point of view, the tuning problem can be considered as follows: the objective function is non-convex, non-smooth and typically expensive to evaluate. Tuning examples include optimizing regularization or kernel parameters.

The figure below shows an example response surface, in which we optimized the hyperparameters of an SVM with RBF kernel. This specific example is available at *Optimization response surface.*
The Optunity library is implemented in Python and allows straightforward integration in other machine learning environments. Optunity is currently also supported in R, MATLAB, GNU Octave and Java through Jython.

If you have any problems, comments or suggestions you can get in touch with us at gitter: Optunity is free software, using a BSD-style license.

As a simple example of Optunity’s features, the code below demonstrates how to tune an SVM with RBF kernel using Optunity and scikit-learn. This involves optimizing the hyperparameters $\gamma$ and $C$:

```python
import optunity
import optunity.metrics
import sklearn.svm

# score function: twice iterated 10-fold cross-validated accuracy
@optunity.cross_validated(x=data, y=labels, num_folds=10, num_iter=2)
def svm_auc(x_train, y_train, x_test, y_test, logC, logGamma):
    model = sklearn.svm.SVC(C=10 ** logC, gamma=10 ** logGamma).fit(x_train, y_train)
    decision_values = model.decision_function(x_test)
    return optunity.metrics.roc_auc(y_test, decision_values)

# perform tuning
hps, _, _ = optunity.maximize(svm_auc, num_evals=200, logC=[-5, 2], logGamma=[-5, 1])

# train model on the full training set with tuned hyperparameters
optimal_model = sklearn.svm.SVC(C=10 ** hps['logC'], gamma=10 ** hps['logGamma']).
                      fit(data, labels)
```

For more examples, please see our Examples.
Issue the following commands to get started on Linux:

```
  git clone https://github.com/claesenm/optunity.git
  export PYTHONPATH=$PYTHONPATH:$pwd/optunity/
```

Afterwards, importing `optunity` should work in Python:

```
  python -c 'import optunity'
```

For a proper installation, run the following:

```
  python optunity/setup.py install
```

or, if you have pip:

```
  pip install optunity
```

Installation may require superuser privileges.
CHAPTER 2

Developer Guide

Optunity API
Optunity is developed at the STADIUS lab of the dept. of electrical engineering at KU Leuven (ESAT). The main contributors to Optunity are:

**Marc Claesen**
- Python package
- framework design & implementation
- solver implementation
- communication protocol design & implementation
- MATLAB wrapper
- Octave wrapper
- Python, MATLAB and Octave examples

**Jaak Simm**
- communication protocol design
- R wrapper
- R examples

**Vilen Jumutc**
- Julia wrapper
Installing Optunity

The source can be obtained from git at http://git.optunity.net (recommended), releases can be obtained from http://releases.optunity.net. Optunity is compatible with Python 2.7 and 3.x. Note that you must have Python installed before Optunity can be used (all Linux distributions have this, but Windows requires explicit installation).

Obtaining the latest version of Optunity using git can be done as follows:

```
git clone https://github.com/claesenm/optunity.git
```

Installation instructions per environment:

- Installing Optunity for Python
- Installing Optunity for MATLAB
- Installing Optunity for Octave
- Installing Optunity for R
- Installing Optunity for Julia
- Installing Optunity for Java

If you encounter any difficulties during installation, please open an issue at https://github.com/claesenm/optunity/issues.
Note: Optunity has soft dependencies on NumPy and [DEAP2012] for the CMA-ES solver. If these Python libraries are unavailable, the CMA-ES solver will be unavailable.

Installing Optunity for Python

If you want to use Optunity in another environment, this is not required. Optunity can be installed as a typical Python package, for example:

- Add Optunity’s root directory (the one you cloned/extracted) to your PYTHONPATH environment variable. Note that this will only affect your current shell. To make it permanent, add this line to your .bashrc (Linux) or adjust the PYTHONPATH variable manually (Windows).

  Extending PYTHONPATH in Linux (not permanent):

  ```bash
  export PYTHONPATH=$PYTHONPATH:/path/to/optunity/
  ```

  Extending PYTHONPATH in Windows (not permanent):

  ```bash
  set PYTHONPATH=%PYTHONPATH%;/path/to/optunity
  ```

- Install using Optunity’ setup script:

  ```bash
  python optunity/setup.py install [--home=/desired/installation/directory/] 
  ```

- Install using pip:

  ```bash
  pip install optunity [-t /desired/installation/directory/]
  ```

After these steps, you should be able to import optunity module:

```bash
python -c 'import optunity'
```

Installing Optunity for MATLAB

To install Optunity for MATLAB, you must add `<optunity>/wrappers/matlab/` and its subdirectories to your MATLAB path. You can set the path in Preferences -> Path or using the following commands:

```matlab
addpath(genpath('/path/to/optunity/wrappers/matlab/'));
savepath
```

After these steps, you should be able to run the examples in `<optunity>/wrappers/matlab/optunity_example.m`:

```matlab
optunity_example
```

Warning: The MATLAB wrapper requires the entire directory structure of Optunity to remain as is. If you only copy the `<optunity>/wrappers/matlab` subdirectory it will not work.
Installing Optunity for Octave

Optunity requires sockets for communication. In Octave, please install the sockets package first (available at Octave-Forge) and at http://octave.sourceforge.net/sockets/.

To install Optunity for Octave, you must add `<optunity>/wrappers/octave/` and its subdirectories to your Octave path:

```octave
addpath(genpath('/path/to/optunity/wrappers/octave/'));
savepath
```

After these steps, you should be able to run the examples in `<optunity>/wrappers/octave/optunity/example/optunity_example.m`:

```octave
optunity_example
```

**Warning:** The Octave wrapper requires the entire directory structure of Optunity to remain as is. If you only copy the `<optunity>/wrappers/octave` subdirectory it will not work.

Installing Optunity for R

First install all dependencies. In R, issue the following command:

```r
install.packages(c("rjson", "ROCR", "enrichvs", "plyr"))
```

Subsequently, clone the git repository and then issue the following commands:

```bash
cd optunity/wrappers
R CMD build R/
R CMD INSTALL optunity_<version number>.tar.gz
```

Installing Optunity for Julia

First install all dependencies. In Julia console issue the following command:

```julia
Pkg.add("PyCall")
```

After this essential package is installed please issue the following command:

```julia
include("<optunity>/wrappers/julia/optunity.jl")
```

where `<optunity>` stands for the root of the working copy of Optunity.

Installing Optunity for Java

Optunity is available for Java through Jython (v2.7+). To use Optunity via Jython the Python package must be installed first (see above).
User Guide

Optunity provides a variety of solvers for hyperparameter tuning problems. A tuning problem is specified by an objective function that provides a score for some tuple of hyperparameters. Specifying the objective function must be done by the user. The software offers a diverse set of solvers to optimize the objective function. A solver determines a good set of hyperparameters.

Jump to

- Solver overview
- Cross-validation
- Quality metrics
- Domain constraints
- Structured search spaces

Optunity consists of a set of core functions that are offered in each environment, which we will now discuss briefly. Clicking on a function will take you to its Python API documentation. If you are using a different environment, you can still get the general idea on the Python pages. To dive into code details straight away, please consult the Optunity API.

If you want to learn by example, please consult our Examples which use various features of Optunity to cover a wide variety of tuning tasks. In case of confusion, we provide a list of basic Terminology.

In the rest of this section we will discuss the main API functions. We will start with very simple functions that offer basic functionality which should meet the needs of most use cases. Subsequently we will introduce the expert interface functions which have more bells and whistles that can be configured.

A variety of solvers is available, discussed in more detail Solver overview. Optunity additionally provides Cross-validation and several Quality metrics.

Simple interface

For beginning users, we offer a set of functions with simple arguments. These functions should be enough for most of your needs. In case these functions are insufficient, please refer to the expert functions listed below or to submodules.

- `optunity.maximize()`: maximizes the objective function. Adheres to a prespecified upper bound on the number of function evaluations. The solution will be within given box constraints. Optunity determines the best solver and its configuration for you.

- `optunity.minimize()`: minimizes the objective function. Adheres to a prespecified upper bound on the number of function evaluations. The solution will be within given box constraints. Optunity determines the best solver and its configuration for you.

- `optunity.maximize_structured()`: maximizes the objective function with a structured search space. This function extends the functionality of `optunity.maximize` by allowing you to specify a structured search space, that is a search space where the existence of some hyperparameters are contingent upon some discrete choices. Adheres to a prespecified upper bound on the number of function evaluations. The solution will be within given box constraints. Optunity determines the best solver and its configuration for you.

- `optunity.minimize_structured()`: minimizes the objective function with a structured search space. This function extends the functionality of `optunity.minimize` by allowing you to specify a structured search space, that is a search space where the existence of some hyperparameters are contingent upon some
discrete choices. Adheres to a prespecified upper bound on the number of function evaluations. The solution will be within given box constraints. Optunity determines the best solver and its configuration for you.

- **optunity.manual()**: prints a basic manual (general or solver specific) Prints a basic manual of Optunity and a list of all registered solvers. If a solver name is specified, its manual will be shown.

  Note: You may alternatively consult the solver API documentation at Solvers for more details.

- **optunity.cross_validated()**: decorator to perform k-fold cross-validation Wrap a function with cross-validation functionality. The way cross-validation is performed is highly configurable, including support for strata and clusters. More details are available here.

**Expert interface**

The following functions are recommended for more advanced use of Optunity. This part of the API allows you to fully configure every detail about the provided solvers. In addition to more control in configuration, you can also perform parallel function evaluations via this interface (turned off by default due to problems in IPython).

- **optunity.suggest_solver()**: suggests a solver and its configuration Suggests a solver and configuration for a given tuning problem, based on the permitted number of function evaluations and box constraints.

- **optunity.make_solver()**: constructs one of Optunity’s registered solvers. See the solver-specific manuals for more information per solver.

- **optunity.optimize()**: optimizes an objective function with given solver Some solvers are capable of vector evaluations. By default, the optimization is done through sequential function evaluations but this can be parallelized by specifying an appropriate pmap argument (cfr. optunity.parallel.pmap()).

More details about our solvers can be found here. To learn about adding constraints to the objective function, please consult this page.

**Solver overview**

We currently recommend using Particle Swarm Optimization (our default). Based on our experience this is the most reliable solver across different learning algorithms. If you consistently get great results with a solver/algorithm combination, we are happy to hear about your experiences.

You can specify which solver you want to use in optunity.maximize() and optunity.minimize(), but only limited configuration is possible. If you want to specify detailed settings for a solver, you can use the expert interface, specifically optunity.make_solver() in combination with optunity.optimize().

The following solvers are available in Optunity:

**Grid Search**

This solver is implemented in optunity.solvers.GridSearch. It as available in optunity.make_solver() as ‘grid search’.

Grid search is an undirected search method that consists of testing a predefined set of values per hyperparameter. A search grid is constructed that is the Cartesian product of these sets of values.
Grid search can be used to tune a limited number of hyperparameters, as the size of the search grid (number of evaluations) increases exponentially in terms of the number of hyperparameters. We generally recommend the use of directed solvers over grid search, as they waste less time exploring uninteresting regions.

When you use choose this solver in `optunity.maximize()` or `optunity.minimize()`, an equal number of values is determined per hyperparameter (spread uniformly). The number of test values per hyperparameter is determined based on the number of permitted evaluations. If you want to specify the grid points manually, this is possible via `optunity.make_solver()` and `optunity.optimize()`.

**Example**

Assume we have two hyperparameters $x$ and $y$. For $x$ we want to test the following values $\{0, 1, 2, 3\}$ and for $y$ we will try $\{-10, -5, 0, 5, 10\}$. The search grid consists of all possible pairs ($4 \times 5 = 20$ in this case), e.g.:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-10</td>
</tr>
<tr>
<td>0</td>
<td>-5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>-10</td>
</tr>
<tr>
<td>1</td>
<td>-5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>-10</td>
</tr>
<tr>
<td>2</td>
<td>-5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>-10</td>
</tr>
<tr>
<td>3</td>
<td>-5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

**Random Search**

This solver is implemented in `optunity.solvers.RandomSearch`. It as available in `optunity.make_solver()` as ‘random search’.

This strategy consists of testing a predefined number of randomly sampled hyperparameter tuples. Sampling is done uniform at random within specified box constraints.

This solver implements the search strategy described in [RAND].

**Particle Swarm Optimization**

This solver is implemented in `optunity.solvers.ParticleSwarm`. It as available in `optunity.make_solver()` as ‘particle swarm’.

Particle swarm optimization (PSO) is a heuristic optimization technique. It simulates a set of particles (candidate solutions) that are moving around in the search-space [PSO2010], [PSO2002].
In the context of hyperparameter search, the position of a particle represents a set of hyperparameters and its movement is influenced by the goodness of the objective function value.

PSO is an iterative algorithm:

1. **Initialization**: a set of particles is initialized with random positions and initial velocities. The initialization step is essentially equivalent to Random Search.

2. **Iteration**: every particle’s position is updated based on its velocity, the particle’s historically best position and the entire swarm’s historical optimum.

   
   ```
   1 for k ← 1 to n_gen do
   2     for i ← 1 to n_part do
   3         \vec{v}_i ← \vec{v}_i + \phi_1 \cdot (\vec{p}_i - \vec{x}_i) + \phi_2 \cdot (\vec{p}_g - \vec{x}_i)
   4         \vec{x}_i ← \vec{x}_i + \vec{v}_i
   5         if \( f(\vec{x}_i) \) is better than \( f(\vec{p}_i) \) then \vec{p}_i ← \vec{x}_i
   6     end
   7     \vec{p}_g ← \text{best}(\{\vec{p}_i \mid i = 1 : n_part\})
   end
   ```

   
   Fig. 4.1: Particle swarm iterations:

   - \( \vec{x}_i \) is a particle’s position,
   - \( \vec{v}_i \) its velocity,
   - \( \vec{p}_i \) its historically best position,
   - \( \vec{p}_g \) the swarm’s optimum,
   - \( \phi_1 \) and \( \phi_2 \) are vectors of uniformly sampled values in \((0, \phi_1)\) and \((0, \phi_2)\), respectively.

PSO has 5 parameters that can be configured (see `optunity.solvers.ParticleSwarm`):

- `num_particles`: the number of particles to use
- `num_generations`: the number of generations (iterations)
- `phi1`: the impact of each particle’s historical best on its movement
- `phi2`: the impact of the swarm’s optimum on the movement of each particle
- `max_speed`: an upper bound for \( \vec{v}_i \)

The number of function evaluations that will be performed is `num_particles * num_generations`. A high number of particles focuses on global, undirected search (just like Random Search), whereas a high number of generations leads to more localized search since all particles will have time to converge.

Bibliographic references:

**Nelder-Mead simplex**

This solver is implemented in `optunity.solvers.NelderMead`. It is available in `optunity.make_solver()` as ‘nelder-mead’.

This is a heuristic, nonlinear optimization method based on the concept of a simplex, originally introduced by Nelder and Mead [NELDERMEAD]. We implemented the version as described on Wikipedia.

This method requires an initial starting point \( x_0 \). It is a good local search method, but will get stuck in bad regions when a poor starting point is specified.
**CMA-ES**

This solver is implemented in `optunity.solvers.CMA_ES`. It is available in `optunity.make_solver()` as 'cma-es'.

CMA-ES stands for Covariance Matrix Adaptation Evolutionary Strategy. This is an evolutionary strategy for continuous function optimization. It can dynamically adapt its search resolution per hyperparameter, allowing for efficient searches on different scales. More information is available in [HANSEN2001].

Optunity’s implementation of this solver is done using the DEAP toolbox [DEAP2012]. This, in turn, requires NumPy. Both dependencies must be met to use this solver.

Bibliographic references:

**Tree-structured Parzen Estimator**

This solver is implemented in `optunity.solvers.TPE`. It is available in `optunity.make_solver()` as ‘TPE’.

The Tree-structured Parzen Estimator (TPE) is a sequential model-based optimization (SMBO) approach. SMBO methods sequentially construct models to approximate the performance of hyperparameters based on historical measurements, and then subsequently choose new hyperparameters to test based on this model.

The TPE approach models $P(x|y)$ and $P(y)$ where $x$ represents hyperparameters and $y$ the associated quality score. $P(x|y)$ is modeled by transforming the generative process of hyperparameters, replacing the distributions of the configuration prior with non-parametric densities. In this solver, Optunity only supports uniform priors within given box constraints. For more exotic search spaces, please refer to [Hyperopt]. This optimization approach is described in detail in [TPE2011] and [TPE2013].

Optunity provides the TPE solver as is implemented in [Hyperopt]. This solver is only available if Hyperopt is installed, which in turn requires NumPy. Both dependencies must be met to use this solver.

**Sobol sequences**

A Sobol sequence is a low discrepancy quasi-random sequence. Sobol sequences were designed to cover the unit hypercube with lower discrepancy than completely random sampling (e.g. Random Search). Optunity supports Sobol sequences in up to 40 dimensions (e.g. 40 hyperparameters).

The figures below show the differences between a Sobol sequence and sampling uniformly at random. These figures can be recreated using the code in `bin/examples/python/sobol_vs_random.py`.

The mathematical details on Sobol sequences are available in the following papers: [SOBOL], [SOBOL2], [ANTONOV], [BRATLEY], [FOX].

Optunity’s default solver is Particle Swarm Optimization.

**Grid Search, Random Search and Sobol sequences** are completely undirected algorithms and consequently not very efficient. Of these three, **Sobol sequences** is most efficient as uses a low-discrepancy quasirandom sequence.

**Nelder-Mead simplex** works well for objective functions that are smooth, unimodal and not too noisy (it is good for local search when you have a good idea about optimal regions for your hyperparameters).

For general searches, **Particle Swarm Optimization** and **CMA-ES** are most robust. Finally, the **Tree-structured Parzen Estimator** solver exposes Hyperopt’s TPE solver in Optunity’s familiar API.
Fig. 4.2: 200 points sampled in 2D with a Sobol sequence.

Fig. 4.3: 200 points sampled in 2D uniformly at random.
Cross-validation

Optunity offers a simple interface to k-fold cross-validation. This is a statistical approach to measure a model’s generalization performance. In the context of hyperparameter search, cross-validation is used to estimate the performance of a hyperparameter tuple. The cross-validation routines we offer are optional and can be replaced by comparable routines from other packages or some other method to estimate generalization performance.

The fold generation procedure in Optunity allows for iterated cross-validation and is aware of both strata (data instances that must be spread across folds) and clusters (sets of instances that must assigned to a single fold). Please refer to `optunity.cross_validated()` for implementation and API details.

We will build examples step by step. The basic setup is a `train` and `predict` function along with some `data` to construct folds over:

```python
from __future__ import print_function
import optunity as opt

def train(x, y, filler=''):  
    print(filler + 'Training data:')
    for instance, label in zip(x, y):
        print(filler + str(instance) + ' ' + str(label))

def predict(x, filler=''):  
    print(filler + 'Testing data:')
    for instance in x:
        print(filler + str(instance))

data = list(range(9))
labels = [0] * 9
```

The recommended way to perform cross-validation is using the `optunity.cross_validation.cross_validated()` function decorator. To use it, you must specify an objective function. This function should contain the logic that is placed in the inner loop in cross-validation (e.g. train a model, predict test set, compute score), with the following signature: `f(x_train, y_train, x_test, y_test, hyperpar_1, hyperpar_2, ...)` (argument names are important):

- `x_train`: training data
- `y_train`: training labels (optional)
- `x_test`: test data
- `y_test`: test labels (optional)
- `hyperparameter names`: the hyperparameters that must be optimized

The `cross_validated` decorator takes care of generating folds, partitioning the data, iterating over folds and aggregating partial results. After decoration, the arguments `x_train`, `y_train`, `x_test` and `y_test` will be bound (e.g. the decorated function does not take these as arguments). The decorated function will have hyperparameters as (keyword) arguments and returns a cross-validation result.

A simple code example:

```python
@opt.cross_validated(x=data, y=labels, num_folds=3)
def cved(x_train, y_train, x_test, y_test):
    train(x_train, y_train)
    predict(x_test)
    return 0.0

cved()
```
Reusing folds

If you want to compare different aspects of the learning approach (learning algorithms, score function, ...), it is a good idea to use the same cross-validation folds. This is very easy by using the `cross_validated` decorator without syntactic sugar. Let's say we want to compare an SVM with RBF kernel and polynomial kernel with the same cross-validation configuration:

```python
import sklearn.svm as svm

def svm_rbf(x_train, y_train, x_test, y_test, C, gamma):
    model = svm.SVC(kernel='rbf', C=C, gamma=gamma).fit(x_train, y_train)
    y_pred = model.predict(x_test)
    return opt.score_functions.accuracy(y_test, y_pred)

def svm_poly(x_train, y_train, x_test, y_test, C, d):
    model = svm.SVC(kernel='poly', C=C, degree=d).fit(x_train, y_train)
    y_pred = model.predict(x_test)
    return opt.score_functions.accuracy(y_test, y_pred)

cv_decorator = opt.cross_validated(x=data, y=labels, num_folds=3)
svm_rbf_cv = cv_decorator(svm_rbf)
svm_poly_cv = cv_decorator(svm_poly)
```

In this example, the function `svm_rbf_cv` takes keyword arguments `C` and `gamma` while `svm_poly_cv` takes `C` and `d`. Both perform cross-validation on the same data, using the same folds.

Nested cross-validation

Nested cross-validation is a commonly used approach to estimate the generalization performance of a modeling process which includes model selection internally. A good summary is provided here.

Nested cv consists of two cross-validation procedures wrapped around each other. The inner cv is used for model selection, the outer cv estimates generalization performance.

This can be done in a straightforward manner using Optunity:

```python
@opt.cross_validated(x=data, y=labels, num_folds=3)
def nested_cv(x_train, y_train, x_test, y_test):
    @opt.cross_validated(x=x_train, y=y_train, num_folds=3)
def inner_cv(x_train, y_train, x_test, y_test):
        train(x_train, y_train, '...')
        predict(x_test, '...')
        return 0.0

    inner_cv()
    predict(x_test)
    return 0.0

nested_cv()
```

The inner `optunity.cross_validated()` decorator has access to the train and test folds generated by the outer procedure (`x_train` and `x_test`). For notational simplicity we assume a problem without labels here.

**Note:** The inner folds are regenerated in every iteration (since we are redefining `inner_cv` each time). The inner folds will therefore be different each time. The outer folds remain static, unless `regenerate_folds=True` is
A complete example of nested cross-validation is available in *Basic: nested cross-validation*.

### Aggregators

Optunity’s cross-validation implementation allows you to specify an *aggregator*. This is the function that will be used to compute the cross-validation result based on the results of individual folds. The default function is *mean*. You can specify any function to compute another measure if desired (for instance *min*, *max*, ...).

### Computing multiple performance measures during cross-validation

Sometimes it is desired to compute multiple performance measures using cross-validation. This is particularly useful for nested cross-validation. This is possible in Optunity by letting the wrapped function return multiple scores and using the *optunity.cross_validation.list_mean()* aggregator:

```python
@optunity.cross_validated(x=data, y=labels, num_folds=3,
                           aggregator=optunity.cross_validation.list_mean)
def f(x_train, y_train, x_test, y_test):
    model = train(x_train, y_train)
    predictions = model.predict(x_test)
    score_1 = score_1(y_test, predictions)
    score_2 = score_2(y_test, predictions)
    return score_1, score_2
```

For even more flexibility, you can use *optunity.cross_validation.identity()* as aggregator, which will return a list of return values for every cross-validation fold.

### Quality metrics

Quality metrics (score/loss functions) are used to quantify the performance of a given model. Score functions are typically maximized (e.g. accuracy, concordance, ...) whereas loss functions should be minimized (e.g. mean squared error, error rate, ...). Optunity provides common score/loss functions for your convenience.

We use the following calling convention:

- *y* (iterable): the true labels/function values
- *yhat* (iterable): the predicted labels/function values
- we assume *y* and *yhat* are of the same length (though we do not assert this).
- potential parameters of the score function must be added by keyword

All functions listed here are available in *optunity.metrics*.

### Score functions

Score functions are typically maximized (e.g. *optunity.maximize()*).
Classification

<table>
<thead>
<tr>
<th>Score</th>
<th>Associated Optunity function</th>
</tr>
</thead>
<tbody>
<tr>
<td>accuracy</td>
<td>accuracy()</td>
</tr>
<tr>
<td>area under ROC curve</td>
<td>roc_auc()</td>
</tr>
<tr>
<td>area under PR curve</td>
<td>pr_auc()</td>
</tr>
<tr>
<td>$F_\beta$</td>
<td>fbeta()</td>
</tr>
<tr>
<td>precision/PPV</td>
<td>precision()</td>
</tr>
<tr>
<td>recall/sensitivity</td>
<td>recall()</td>
</tr>
<tr>
<td>specificity/NPV</td>
<td>npv()</td>
</tr>
<tr>
<td>PU score</td>
<td>pu_score()</td>
</tr>
</tbody>
</table>

Regression

<table>
<thead>
<tr>
<th>Score</th>
<th>Associated Optunity function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R squared</td>
<td>r_squared()</td>
</tr>
</tbody>
</table>

Loss functions

Loss functions are typically minimized (e.g. optunity.minimize()).

Classification

<table>
<thead>
<tr>
<th>Score</th>
<th>Associated Optunity function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brier score</td>
<td>brier()</td>
</tr>
<tr>
<td>error rate</td>
<td>error_rate()</td>
</tr>
<tr>
<td>log loss</td>
<td>logloss()</td>
</tr>
</tbody>
</table>

Regression

<table>
<thead>
<tr>
<th>Score</th>
<th>Associated Optunity function</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean squared error</td>
<td>mse()</td>
</tr>
<tr>
<td>absolute error</td>
<td>absolute_error()</td>
</tr>
</tbody>
</table>

Domain constraints

Optunity supports domain constraints on the objective function. Domain constraints are used to enforce solvers to remain within a prespecified search space. Most solvers that Optunity provides are implicitly unconstrained (cfr. Solver overview), though hyperparameters are usually constrained in some way (ex: regularization coefficients must be positive).

A set of simple constraints and facilities to use them are provided in Domain constraints. Specifically, the following constraints are provided:

- $lb_{foc}$: assigns a lower bound (open or closed)
- $ub_{foc}$: assigns an upper bound (open or closed)
- $range_{foc}(oc)$: creates a range constraint (e.g. $A < x < B$)
All of the above constraints apply to a specific hyperparameter. Multidimensional constraints are possible, but you would need to implement them yourself (see Implementing custom constraints).

Note that the functions `optunity.maximize()` and `optunity.minimize()` wrap explicit box constraints around the objective function prior to starting the solving process. The expert function `optunity.optimize()` does not do this for you, which allows more flexibility at the price of verbosity.

Constraint violations in Optunity raise a `ConstraintViolation` exception by default. The usual way we handle these exceptions is by returning a certain (typically bad) default function value (using the `optunity.constraints.violations_defaulted()` decorator). This will cause solvers to stop searching in the infeasible region.

To add a series of constraints, we recommend using the `optunity.wrap_constraints()` function. This function takes care of assigning default values on constraint violations if desired.

### Implementing custom constraints

Constraints are implemented as a binary functions, which yield false in case of a constraint violation. You can design your own constraint according to this principle. For instance, assume we have a binary function with two arguments x and y:

```python
def f(x, y):
    return x + y
```

Optunity provides all univariate constraints you need, but lets say we want to constrain the domain to be the unit circle in x,y-space. We can do this using the following constraint:

```python
constraint = lambda x, y: (x ** 2 + y ** 2) <= 1.0
```

To constrain `f`, we use `optunity.wrap_constraints()`:

```python
fc = optunity.wrap_constraints(f, custom=[constraint])
```

The constrained function `fc(x, y)` will yield `x + y` if the arguments are within the unit circle, or raise a `ConstraintViolation` exception otherwise.

### Structured search spaces

Some hyperparameter optimization problems have a hierarchical nature, comprising discrete choices and depending on the choice additional hyperparameters may exist. A common example is optimizing a kernel, without choosing a certain family of kernel functions in advance (e.g. polynomial, RBF, linear, ...).

Optunity provides the functions `optunity.maximize_structured()` and `optunity.minimize_structured()` for such structured search spaces. Structured search spaces can be specified as nested dictionaries, which generalize the standard way of specifying box constraints:

- hyperparameters within box constraints: specified as dictionary entries, where `key=parameter name` and `value=box constraints (list)`.

- discrete choices: specified as a dictionary, where each entry represents a choice, that is `key=option name` and `value` has two:
  - a new dictionary of conditional hyperparameters, following the same rules
  - `None`, to indicate a choice which doesn’t imply further hyperparameterization
Structured search spaces can be nested to form any graph-like search space. It’s worth noting that the addition of discrete choices naturally generalizes Optunity’s search space definition in `optunity.minimize()` and `optunity.maximize()`, since box constraints are specified as keyword arguments there, so Python’s `kwargs` to these functions effectively follows the exact same structure, e.g.:

```python
_ = optunity.minimize(fun, num_evals=1, A=[0, 1], B=[-1, 2])
# kwargs = {A: [0, 1], B: [-1, 2]}
```

**Example: SVM kernel hyperparameterization**

Suppose we want to optimize the kernel, choosing from the following options:

- linear kernel $\kappa_{\text{linear}}(u, v) = u^T v$: no additional hyperparameters
- polynomial kernel $\kappa_{\text{poly}}(u, v) = (u^T v + \text{coef0})^{\text{degree}}$: 2 hyperparameters (degree and coef0)
- RBF kernel $\kappa_{\text{RBF}}(u, v) = \exp(-\gamma \|u - v\|^2)$: 1 hyperparameter (gamma)

When we put this all together, the SVM kernel search space can be defined as follows (Python syntax):

```python
search = {
    'kernel': {
        'linear': None,
        'rbf': {'gamma': [gamma_min, gamma_max]},
        'poly': {'degree': [degree_min, degree_max],
                  'coef0': [coef0_min, coef0_max]}
    }
}
```

The structure of the search space directly matches the hyperparameterization of every kernel function. We use `None` in the linear kernel as there are no additional hyperparameters. The hyperparameters of the RBF and polynomial kernel follow Optunity’s default syntax based on dictionaries.

Optunity also supports nested choices, for example an outer choice for the learning algorithm (e.g. SVM, naive Bayes, ... ) and an inner choice for the SVM kernel function. This is illustrated in the following notebook: `sklearn: automated learning method selection and tuning`.

**Terminology**

To avoid confusion, here is some basic terminology we will use consistently throughout the documentation:

- **hyperparameters**  User-specified parameters for a given machine learning approach. These will serve as optimization variables in our context.

  Example: kernel parameters.

- **score**  Some measure which quantifies the quality of a certain modeling approach (model type + hyperparameters).

  Example: cross-validated accuracy of a classifier.

- **objective function**  The function that must be optimized. The arguments to this function are hyperparameters and the result is some score measure of a model constructed using these hyperparameters. Optunity can minimize and maximize, depending on the requirements.

  Example: accuracy of an SVM classifier as a function of kernel and regularization parameters.

- **box constraints**  Every hyperparameter of the tuning problem must be within a prespecified interval. The optimal solution will be within the hyperrectangle (box) specified by the ranges.

- **solver**  A strategy to optimize hyperparameters, such as `grid search`. 

---

**Optunity Documentation, Release 1.1.0**

---

4.2. User Guide 23
**train-predict-score (TPS) chain**  A sequence of code which trains a model, uses it to predict an independent test set and then computes some score measure based on the predictions. TPS chains must be specified by the user as they depend entirely on the method that is being tuned and the evaluation criteria.

**Examples**

Here you can find a variety of examples to illustrate how Optunity can be used in tandem with other machine learning software. We have split the examples per language and library.

To contribute examples, please send us a pull request on Github.

**Python**

The following examples are available as IPython notebooks in the `OPTUNITY/notebooks` folder:

**Basic: cross-validation**

This notebook explores the main elements of Optunity’s cross-validation facilities, including:

- standard cross-validation
- using strata and clusters while constructing folds
- using different aggregators

We recommend perusing the related documentation for more details.

Nested cross-validation is available as a separate notebook.

```python
import optunity
import optunity.cross_validation

# We start by generating some toy data containing 6 instances which we will partition into folds.

data = list(range(6))
labels = [True] * 3 + [False] * 3

# Standard cross-validation

def f(x_train, y_train, x_test, y_test):
    print("\n")
    print("train data:\t" + str(x_train) + "\ttrain labels:\t" + str(y_train))
    print("test data:\t" + str(x_test) + "\ttest labels:\t" + str(y_test))
    return 0.0
```

24 Chapter 4. Indices and tables
We start with 2 folds, which leads to equally sized train and test partitions.

```python
f_2folds = optunity.cross_validated(x=data, y=labels, num_folds=2)(f)
print("using 2 folds")
f_2folds()
```

```
using 2 folds

train data: [4, 2, 0]   train labels: [False, True, True]
test data: [3, 1, 5]    test labels: [False, True, False]
train data: [3, 1, 5]   train labels: [False, True, False]
test data: [4, 2, 0]    test labels: [False, True, True]
```

```
0.0
```

# f_2folds as defined above would typically be written using decorator syntax as follows:
# we don’t do that in these examples so we can reuse the toy objective function

```python
@optunity.cross_validated(x=data, y=labels, num_folds=2)
def f_2folds(x_train, y_train, x_test, y_test):
    print("")
    print("train data:\t" + str(x_train) + "\t train labels:\t" + str(y_train))
    print("test data:\t" + str(x_test) + "\t test labels:\t" + str(y_test))
    return 0.0
```

If we use three folds instead of 2, we get 3 iterations in which the training set is twice the size of the test set.

```python
f_3folds = optunity.cross_validated(x=data, y=labels, num_folds=3)(f)
print("using 3 folds")
f_3folds()
```

```
using 3 folds

train data: [2, 1, 3, 0]   train labels: [True, True, False, True]
test data: [5, 4]          test labels: [False, False]
train data: [5, 4, 3, 0]   train labels: [False, False, False, True]
test data: [2, 1]          test labels: [True, True]
train data: [5, 4, 2, 1]   train labels: [False, False, True, True]
test data: [3, 0]          test labels: [False, True]
```

```
0.0
```

If we do two iterations of 3-fold cross-validation (denoted by 2x3 fold), two sets of folds are generated and evaluated.

```python
f_2x3folds = optunity.cross_validated(x=data, y=labels, num_folds=3, num_iter=2)(f)
print("using 2x3 folds")
f_2x3folds()
```

```
using 2x3 folds

train data: [4, 1, 5, 3]   train labels: [False, True, False, False]
test data: [0, 2]          test labels: [True, True]
```
Using strata and clusters

Strata are defined as sets of instances that should be spread out across folds as much as possible (e.g. stratify patients by age). Clusters are sets of instances that must be put in a single fold (e.g. cluster measurements of the same patient).

Optunity allows you to specify strata and/or clusters that must be accounted for while construct cross-validation folds. Not all instances have to belong to a stratum or clusters.

**Strata**

We start by illustrating strata. Strata are specified as a list of lists of instances indices. Each list defines one stratum. We will reuse the toy data and objective function specified above. We will create 2 strata with 2 instances each. These instances will be spread across folds. We create two strata: \{0, 1\} and \{2, 3\}.

```python
strata = [[0, 1], [2, 3]]
f_stratified = optunity.cross_validated(x=data, y=labels, strata=strata, num_folds=3)(f)
f_stratified()
```

**Clusters**

Clusters work similarly, except that now instances within a cluster are guaranteed to be placed within a single fold. The way to specify clusters is identical to strata. We create two clusters: \{0, 1\} and \{2, 3\}. These pairs will always
Strata and clusters

Strata and clusters can be used together. Let's say we have the following configuration:

- 1 stratum: \{0, 1, 2\}
- 2 clusters: \{0, 3\}, \{4, 5\}

In this particular example, instances 1 and 2 will inevitably end up in a single fold, even though they are part of one stratum. This happens because the total data set has size 6, and 4 instances are already in clusters.

Aggregators

Aggregators are used to combine the scores per fold into a single result. The default approach used in cross-validation is to take the mean of all scores. In some cases, we might be interested in worst-case or best-case performance, the spread, ...

Opunity allows passing a custom callable to be used as aggregator.

The default aggregation in Optunity is to compute the mean across folds.
```python
@optunity.cross_validated(x=data, num_folds=3)
def f(x_train, x_test):
    result = x_test[0]
    print(result)
    return result
f(1)
```

```
4
1
2
2.3333333333333335
```

This can be replaced by any function, e.g. min or max.

```python
@optunity.cross_validated(x=data, num_folds=3, aggregator=max)
def fmax(x_train, x_test):
    result = x_test[0]
    print(result)
    return result
fmax(1)
```

```
2
5
1
```

```python
@optunity.cross_validated(x=data, num_folds=3, aggregator=min)
def fmin(x_train, x_test):
    result = x_test[0]
    print(result)
    return result
fmin(1)
```

```
3
4
5
3
```

**Retaining intermediate results**

Often, it may be useful to retain all intermediate results, not just the final aggregated data. This is made possible via `optunity.cross_validation.mean_and_list` aggregator. This aggregator computes the mean for internal use in cross-validation, but also returns a list of lists containing the full evaluation results.

```python
@optunity.cross_validated(x=data, num_folds=3,
aggregator=optunity.cross_validation.mean_and_list)
def f_full(x_train, x_test, coeff):
```
return x_test[0] * coeff

# evaluate f
mean_score, all_scores = f_full(1.0)
print(mean_score)
print(all_scores)

2.33333333333
[0.0, 2.0, 5.0]

Note that a cross-validation based on the mean_and_list aggregator essentially returns a tuple of results. If the result is iterable, all solvers in Optunity use the first element as the objective function value. You can let the cross-validation procedure return other useful statistics too, which you can access from the solver trace.

```python
opt_coeff, info, _ = optunity.minimize(f_full, coeff=[0, 1], num_evals=10)
print(opt_coeff)
print("call log")
for args, val in zip(info.call_log['args']['coeff'], info.call_log['values']):
    print(str(args) + ' ' + str(val))
```

```python
{'coeff': 0.15771484375}
call log
0.34521484375 (0.8055013020833334, [0.0, 0.6904296875, 1.72607421875])
0.47021484375 (1.09716796875, [0.0, 0.9404296875, 2.35107421875])
0.97021484375 (2.2638346354166665, [0.0, 1.9404296875, 4.85107421875])
0.72021484375 (1.680513020833333, [0.0, 1.4404296875, 3.60107421875])
0.22021484375 (0.5138346354166666, [0.0, 0.4404296875, 1.10107421875])
0.15771484375 (0.3680013020833333, [0.0, 0.3154296875, 0.78857421875])
0.65771484375 (1.53466796875, [0.0, 1.3154296875, 3.28857421875])
0.90771484375 (2.1180013020833335, [0.0, 1.8154296875, 4.53857421875])
0.40771484375 (0.9513346354166666, [0.0, 0.8154296875, 2.03857421875])
0.28271484375 (0.65966796875, [0.0, 0.5654296875, 1.41357421875])
```

**Cross-validation with scikit-learn**

In this example we will show how to use cross-validation methods that are provided by scikit-learn in conjunction with Optunity. To do this we provide Optunity with the folds that scikit-learn produces in a specific format.

In supervised learning datasets often have unbalanced labels. When performing cross-validation with unbalanced data it is good practice to preserve the percentage of samples for each class across folds. To achieve this label balance we will use StratifiedKFold.

```python
data = list(range(20))
labels = [1 if i%4==0 else 0 for i in range(20)]

@optunity.cross_validated(x=data, y=labels, num_folds=5)
def unbalanced_folds(x_train, y_train, x_test, y_test):
    print("")
    print("train data:\t" + str(x_train) + "\ntrain labels:\t" + str(y_train)) + '\n'
    print("test data:\t" + str(x_test) + "\ntest labels:\t" + str(y_test)) + '\n'
    return 0.0

unbalanced_folds()
```

4.3. Examples
Notice above how the test label sets have a varying number of positive samples, some have none, some have one, and some have two.

```python
from sklearn.cross_validation import StratifiedKFold

stratified_5folds = StratifiedKFold(labels, n_folds=5)
folds = [[list(test) for train, test in stratified_5folds]]

@optunity.cross_validated(x=data, y=labels, folds=folds, num_folds=5)
def balanced_folds(x_train, y_train, x_test, y_test):
    print(""
    print("train data:\t" + str(x_train) + "\ntrain labels:\t" + str(y_train) + "\n"
    print("test data:\t" + str(x_test) + "\ntest labels:\t" + str(y_test) + "\n"
    return 0.0

balanced_folds()
```
test labels: [1, 0, 0, 0]  

train data: [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]  
train labels: [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]  

test data: [4, 5, 6, 7]  
test labels: [1, 0, 0, 0]  

train data: [0, 1, 2, 3, 4, 5, 6, 7, 12, 13, 14, 15, 16, 17, 18, 19]  
train labels: [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]  

test data: [8, 9, 10, 11]  
test labels: [1, 0, 0, 0]  

train data: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 16, 17, 18, 19]  
train labels: [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]  

test data: [12, 13, 14, 15]  
test labels: [1, 0, 0, 0]  

train data: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]  
train labels: [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]  

test data: [16, 17, 18, 19]  
test labels: [1, 0, 0, 0]  

0.0  

Now all of our train sets have four positive samples and our test sets have one positive sample.

To use predetermined folds, place a list of the test sample idices into a list. And then insert that list into another list. Why so many nested lists? Because you can perform multiple cross-validation runs by setting num_iter appropriately and then append num_iter lists of test samples to the outer most list. Note that the test samples for a given fold are the idices that you provide and then the train samples for that fold are all of the indices from all other test sets joined together. If not done carefully this may lead to duplicated samples in a train set and also samples that fall in both train and test sets of a fold if a datapoint is in multiple folds’ test sets.

data = list(range(6))  
labels = [True] * 3 + [False] * 3  
fold1 = [[0, 3], [1, 4], [2, 5]]  
fold2 = [[0, 5], [1, 4], [0, 3]]  
# notice what happens when the indices are not unique  
folds = [fold1, fold2]  

@optunity.cross_validated(x=data, y=labels, folds=folds, num_folds=3, num_iter=2)  
def multiple_iters(x_train, y_train, x_test, y_test):  
    print("")  
    print("train data: \t" + str(x_train) + " train labels: \t" + str(y_train))  
    print("test data: \t" + str(x_test) + " test labels: \t" + str(y_test))  
    return 0.0  

multiple_iters()
Basic: nested cross-validation

In this notebook we will briefly illustrate how to use Optunity for nested cross-validation.

Nested cross-validation is used to reliably estimate generalization performance of a learning pipeline (which may involve preprocessing, tuning, model selection, ...). Before starting this tutorial, we recommend making sure you are reliable with basic cross-validation in Optunity.

We will use a scikit-learn SVM to illustrate the key concepts on the MNIST data set.

```python
import optunity
import optunity.cross_validation
import optunity.metrics
import numpy as np
import sklearn.svm

We load the digits data set and will construct models to distinguish digits 6 from and 8.

```python
from sklearn.datasets import load_digits
digits = load_digits()
n = digits.data.shape[0]

positive_digit = 6
negative_digit = 8

positive_idx = [i for i in range(n) if digits.target[i] == positive_digit]
negative_idx = [i for i in range(n) if digits.target[i] == negative_digit]

# add some noise to the data to make it a little challenging
original_data = digits.data[positive_idx + negative_idx, ...]
data = original_data + 5 * np.random.randn(original_data.shape[0], original_data.shape[1])
labels = [True] * len(positive_idx) + [False] * len(negative_idx)
```

The basic nested cross-validation scheme involves two cross-validation routines:

- outer cross-validation: to estimate the generalization performance of the learning pipeline. We will use 5folds.
• inner cross-validation: to use while optimizing hyperparameters. We will use twice iterated 10-fold cross-validation.

Here, we have to take into account that we need to stratify the data based on the label, to ensure we don’t run into situations where only one label is available in the train or testing splits. To do this, we use the `strata_by_labels` utility function.

We will use an SVM with RBF kernel and optimize gamma on an exponential grid $10^{-5} < \gamma < 10^1$ and $0 < C < 10$ on a linear grid.

```python
# outer cross-validation to estimate performance of whole pipeline
@optunity.cross_validated(x=data, y=labels, num_folds=5,
strata=optunity.cross_validation.strata_by_labels(labels))
def nested_cv(x_train, y_train, x_test, y_test):
    # inner cross-validation to estimate performance of a set of hyperparameters
    @optunity.cross_validated(x=x_train, y=y_train, num_folds=10, num_iter=2,
strata=optunity.cross_validation.strata_by_labels(y_train))
    def inner_cv(x_train, y_train, x_test, y_test, C, logGamma):
        # note that the x_train, ... variables in this function are not the same
        # as within nested_cv!
        model = sklearn.svm.SVC(C=C, gamma=10 ** logGamma).fit(x_train, y_train)
        predictions = model.decision_function(x_test)
        return optunity.metrics.roc_auc(y_test, predictions)

    hpars, info, _ = optunity.maximize(inner_cv, num_evals=100,
C=[0, 10], logGamma=[-5, 1])

    print('Hyperparameters: ' + str(hpars))
    print('Cross-validated AUROC after tuning: %1.3f' % info.optimum)
    model = sklearn.svm.SVC(C=hpars['C'], gamma=10 ** hpars['logGamma']).fit(x_train, y_train)
    predictions = model.decision_function(x_test)
    return optunity.metrics.roc_auc(y_test, predictions)

    auc = nested_cv()
    print('')
    print('Nested AUROC: %1.3f' % auc)
```

Hyperparameters: {'logGamma': -3.8679410473451057, 'C': 0.6162109374999996}
Cross-validated AUROC after tuning: 1.000

Hyperparameters: {'logGamma': -4.535231399331072, 'C': 0.4839113474508706}
Cross-validated AUROC after tuning: 0.999

Hyperparameters: {'logGamma': -4.0821875, 'C': 1.5395986549905802}
Cross-validated AUROC after tuning: 1.000

Hyperparameters: {'logGamma': -3.078125, 'C': 6.015625}
Cross-validated AUROC after tuning: 1.000

Hyperparameters: {'logGamma': -4.630859375, 'C': 3.173828125}
Cross-validated AUROC after tuning: 1.000

Nested AUROC: 0.999

If you want to explicitly retain statistics from the inner cross-validation procedure, such as the ones we printed below, we can do so by returning tuples in the outer cross-validation and using the `identity` aggregator.

4.3. Examples
# outer cross-validation to estimate performance of whole pipeline
@optunity.cross_validated(x=data, y=labels, num_folds=5,
    strata=optunity.cross_validation.strata_by_labels(labels),
    aggregator=optunity.cross_validation.identity)
def nested_cv(x_train, y_train, x_test, y_test):
    
    # inner cross-validation to estimate performance of a set of hyperparameters
    @optunity.cross_validated(x=x_train, y=y_train, num_folds=10, num_iter=2,
    strata=optunity.cross_validation.strata_by_labels(y_train))
def inner_cv(x_train, y_train, x_test, y_test, C, logGamma):
        # note that the x_train, ... variables in this function are not the same
        # as within nested_cv!
        model = sklearn.svm.SVC(C=C, gamma=10 ** logGamma).fit(x_train, y_train)
        predictions = model.decision_function(x_test)
        return optunity.metrics.roc_auc(y_test, predictions)
        
        hpars, info, _ = optunity.maximize(inner_cv, num_evals=100,
                                C=[0, 10], logGamma=[-5, 1])
        model = sklearn.svm.SVC(C=hpars['C'], gamma=10 ** hpars['logGamma']).fit(x_train, y_train)
        predictions = model.decision_function(x_test)
        
        # return AUROC, optimized hyperparameters and AUROC during hyperparameter search
        return optunity.metrics.roc_auc(y_test, predictions), hpars, info.optimum
        
    nested_cv_result = nested_cv()

    We can then process the results like this:

    aucs, hpars, optima = zip(*nested_cv_result)
    print("AUCs: " + str(aucs))
    print('')
    print("hpars: " + \n".join(map(str, hpars)))
    print('')
    print("optima: " + str(optima))
    mean_auc = sum(aucs) / len(aucs)
    print('')
    print("Mean AUC \$1.3f\%1.3f\%1.3f\%

AUCs: (0.9992063492063492, 1.0, 1.0, 0.9976190476190476, 0.9984126984126984)

hpars: {'logGamma': -3.5753515625, 'C': 3.9048828125000004}
{'logGamma': -2.6765234375, 'C': 6.9193359375000005}
{'logGamma': -3.0538671875, 'C': 2.2935546875}
{'logGamma': -3.593515625, 'C': 4.4136718749999995}
{'logGamma': -3.337747403818736, 'C': 4.367953383541078}

optima: (0.9995032051282051, 0.9985177917320774, 0.9994871794871795, 0.9995238095238095, 0.9995032051282051)

Mean AUC 0.999
Basic: minimizing a simple function

In this example we will use various Optunity solvers to minimize the following simple 2d parabola:

\[ f(x, y) = (x - x_{off})^2 + (y - y_{off})^2, \]

where \( x_{off} \) and \( y_{off} \) are randomly determined offsets.

```python
def create_objective_function():
    xoff = random.random()
yoff = random.random()
def f(x, y):
    return (x - xoff)**2 + (y - yoff)**2
return f
```

We start with the necessary imports.

```python
# comment this line when running the notebook yourself
%matplotlib inline
import math
import optunity
import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

First we check which solvers are available. This is available via `optunity.available_solvers`.

```python
solvers = optunity.available_solvers()
print('Available solvers: ' + ', '.join(solvers))
```

```
Available solvers: particle swarm, tpe, sobol, nelder-mead, random search, cma-es, grid search
```

To run an experiment, we start by generating an objective function.

```python
f = create_objective_function()
```

Now we use every available solver to optimize the objective function within the box \( x \in [-5, 5] \) and \( y \in [-5, 5] \) and save the results.

```python
logs = {}
for solver in solvers:
    pars, details, _ = optunity.minimize(f, num_evals=100, x=[-5, 5], y=[-5, 5],
                                          solver_name=solver)
    logs[solver] = np.array([details.call_log['args']['x'],
                              details.call_log['args']['y']])
```

Finally, lets look at the results, that is the trace of each solver along with contours of the objective function.

```python
# make sure different traces are somewhat visually separable
colors = ['r', 'g', 'b', 'y', 'k', 'y', 'r', 'g']
markers = ['x', '+', 'o', 's', 'p', 'x', '+', 'o']

# compute contours of the objective function
delta = 0.025
x = np.arange(-5.0, 5.0, delta)
```

4.3. Examples
Now let's see the performance of the solvers across 100 repeated experiments. We will do 100 experiments for each solver and then report the resulting statistics. This may take a while to run.

```python
optima = dict([(s, []) for s in solvers])
for i in range(100):
    f = create_objective_function()
    for solver in solvers:
        pars, details, _ = optunity.minimize(f, num_evals=100, x=[-5, 5], y=[-5, 5], solver_name=solver)
        # the above line can be parallelized by adding `pmap=optunity.pmap`
        # however this is incompatible with IPython
        optima[solver].append(details.optimum)
        logs[solver] = np.array([details.call_log['args']['x'], details.call_log['args']['y']])
```

```python
from collections import OrderedDict
```
```python
log_optima = OrderedDict()
means = OrderedDict()
std = OrderedDict()
for k, v in optima.items():
    log_optima[k] = [-math.log10(val) for val in v]
    means[k] = sum(log_optima[k]) / len(v)
    std[k] = np.std(log_optima[k])
plt.barh(np.arange(len(means)), means.values(), height=0.8, xerr=std.values(), alpha=0.5)
plt.xlabel('number of correct digits')
plt.yticks(np.arange(len(means))+0.4, list(means.keys()))
plt.tight_layout()
plt.show()```

**Basic: Sobol sequences**

In this example we will show the difference between a 2-d Sobol sequence and sampling uniformly at random in 2 dimensions. We will use the `sobol` and `random-search` solvers. The Sobol sequence has lower discrepancy, i.e., the generated samples are spread out better in the sampling space.

This example requires matplotlib to generate figures.

```
# uncomment the following line when you run this notebook interactively
%matplotlib inline
import matplotlib.pyplot as plt
import optunity
import random
```
of samples.

```python
def f(x1, x2):
    return 0.0
```

We run the random search solver and Sobol sequence solver.

```python
__, info_random, __ = optunity.minimize(f, num_evals=200, x1=[0, 1], x2=[0, 2], solver__name='random search')
plt.plot(info_random.call_log['args']['x1'], info_random.call_log['args']['x2'], 'bo')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Random search')
plt.show()

__, info_sobol, __ = optunity.minimize(f, num_evals=200, x1=[0, 1], x2=[0, 2], solver__name='sobol')
plt.plot(info_sobol.call_log['args']['x1'], info_sobol.call_log['args']['x2'], 'ro')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Sobol sequence')
plt.show()
```
Optimization response surface

In this example we will use Optunity to optimize hyperparameters for a support vector machine classifier (SVC) in scikit-learn. We will construct a simple toy data set to illustrate that the response surface can be highly irregular for even simple learning tasks (i.e., the response surface is non-smooth, non-convex and has many local minima).

```python
import optunity
import optunity.metrics

# comment this line if you are running the notebook
import sklearn.svm
import numpy as np
import math
import pandas

%matplotlib inline
from matplotlib import pylab as plt
from mpl_toolkits.mplot3d import Axes3D

Create a 2-D toy data set.

```python
npos = 200
nneg = 200

delta = 2 * math.pi / npos
radius = 2
circle = np.array([(radius * math.sin(i * delta),
                    radius * math.cos(i * delta))
                   for i in range(npos)]

neg = np.random.randn(nneg, 2)
```
pos = np.random.randn(npos, 2) + circle
data = np.vstack((neg, pos))
labels = np.array([False] * nneg + [True] * npos)

Let's have a look at our 2D data set.

plt.plot(neg[:,0], neg[:,1], 'ro')
plt.plot(pos[:,0], pos[:,1], 'bo')

In order to use Optunity to optimize hyperparameters, we start by defining the objective function. We will use 5-fold cross-validated area under the ROC curve. We will regenerate folds for every iteration, which helps to minimize systematic bias originating from the fold partitioning.

We start by defining the objective function `svm_rbf_tuned_auroc()`, which accepts $C$ and $\gamma$ as arguments.

```python
@optunity.cross_validated(x=data, y=labels, num_folds=5, regenerate_folds=True)
def svm_rbf_tuned_auroc(x_train, y_train, x_test, y_test, logC, logGamma):
    model = sklearn.svm.SVC(C=10 ** logC, gamma=10 ** logGamma).fit(x_train, y_train)
    decision_values = model.decision_function(x_test)
    auc = optunity.metrics.roc_auc(y_test, decision_values)
    return auc
```

Now we can use Optunity to find the hyperparameters that maximize AUROC.

```python
optimal_rbf_pars, info, _ = optunity.maximize(svm_rbf_tuned_auroc, num_evals=300,
                                           logC=[-4, 2], logGamma=[-5, 0])
# when running this outside of IPython we can parallelize via optunity.pmap
# optimal_rbf_pars, _, _ = optunity.maximize(svm_rbf_tuned_auroc, 150, C=[0, 10],
#                                           gamma=[0, 0.1], pmap=optunity.pmap)
```
print("Optimal parameters: " + str(optimal_rbf_pars))
print("AUROC of tuned SVM with RBF kernel: \$1.3f" % info.optimum)

Optimal parameters: {'logGamma': -1.7262599731822696, 'logC': 0.5460942232689681}
AUROC of tuned SVM with RBF kernel: 0.825

We can turn the call log into a pandas dataframe to efficiently inspect the solver trace.

df = optunity.call_log2dataframe(info.call_log)

The past function evaluations indicate that the response surface is filled with local minima, caused by finite sample effects. To see this, we can make surface plots.

cutoff = 0.5
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xs=df[df.value > cutoff]['logC'],
          ys=df[df.value > cutoff]['logGamma'],
          zs=df[df.value > cutoff]['value'])
ax.set_xlabel('logC')
ax.set_ylabel('logGamma')
ax.set_zlabel('AUROC')

<matplotlib.text.Text at 0x7f1f26cbed50>

The above plot shows the particles converge directly towards the optimum. At this granularity, the response surface appears smooth.

However, a more detailed analysis reveals this is not the case, as shown subsequently: - showing the sub trace with score up to 90% of the optimum - showing the sub trace with score up to 95% of the optimum - showing the sub trace with score up to 99% of the optimum

cutoff = 0.9 * info.optimum
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

4.3. Examples 41
cutoff = 0.95 * info.optimum
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xs=df[df.value > cutoff]['logC'],
            ys=df[df.value > cutoff]['logGamma'],
            zs=df[df.value > cutoff]['value'])
ax.set_xlabel('logC')
ax.set_ylabel('logGamma')
ax.set_zlabel('AUROC')
Let’s further examine the area close to the optimum, that is the 95% region. We will examine a 50x50 grid in this region.

\[
\begin{align*}
\text{minlogc} &= \min(df[df.value > \text{cutoff}]['logC']) \\
\text{maxlogc} &= \max(df[df.value > \text{cutoff}]['logC']) \\
\text{minloggamma} &= \min(df[df.value > \text{cutoff}]['logGamma']) \\
\text{maxloggamma} &= \max(df[df.value > \text{cutoff}]['logGamma']) \\
\_\text{, info\_new, \_} &= \text{optunity.maximize(svm\_rbf\_tuned\_auroc, num\_evals=2500,} \\
&\quad \text{logC=[minlogc, maxlogc],} \\
&\quad \text{logGamma=[minloggamma, maxloggamma],} \\
&\quad \text{solver\_name='grid search')} \\
\end{align*}
\]

Make a new data frame of the call log for easy manipulation.

\[
\text{df\_new} = \text{optunity.call\_log2dataframe(info\_new.call\_log)}
\]

Determine the region in which we will do a standard grid search to obtain contours.

\[
\begin{align*}
\text{reshape} &= \lambda x: \text{np.reshape}(x, (50, 50)) \\
\text{logcs} &= \text{reshape}(\text{df\_new}['logC']) \\
\text{loggammas} &= \text{reshape}(\text{df\_new}['logGamma']) \\
\text{values} &= \text{reshape}(\text{df\_new}['value'])
\end{align*}
\]

Now, let’s look at a contour plot to see whether or not the response surface is smooth.

\[
\begin{align*}
\text{levels} &= \text{np.arange(0.97, 1.0, 0.005)} \ast \text{info\_new.optimum} \\
\text{CS} &= \text{plt.contour(logcs, loggammas, values, levels=levels)} \\
\text{plt.clabel(CS, inline=1, fontsize=10)} \\
\text{plt.title('Contours of SVM tuning response surface')} \\
\text{plt.xlabel('log C')} \\
\text{plt.ylabel('log Gamma')}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>479</td>
<td>DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future</td>
</tr>
<tr>
<td>483</td>
<td>DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future</td>
</tr>
<tr>
<td>485</td>
<td>DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future</td>
</tr>
<tr>
<td>487</td>
<td>DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future</td>
</tr>
<tr>
<td>491</td>
<td>DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future</td>
</tr>
<tr>
<td>493</td>
<td>DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future</td>
</tr>
<tr>
<td>497</td>
<td>DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future</td>
</tr>
</tbody>
</table>

Non integer number instead of an integer will result in an error in the future
nlc.append(np.r_[xy2, lc[I[1]]:I[0] + 1, xy1])
/usr/lib/pymodules/python2.7/matplotlib/contour.py:479: DeprecationWarning: using a non-integer number instead of an integer will result in an error in the future
nlc.append(np.r_[xy2, lc[I[1]]:I[0] + 1, xy1])

<matplotlib.text.Text at 0x7f1f26ac3090>

Clearly, this response surface is filled with local minima. This is a general observation in automated hyperparameter optimization, and is one of the key reasons we need robust solvers. If there were no local minima, a simple gradient-descent-like solver would do the trick.

**OpenCV: optical character recognition**

We will use OpenCV ([http://www.opencv.org/](http://www.opencv.org/)) for optical character recognition (OCR) using support vector machine (SVM) classifiers. This example is based on OpenCV’s digit tutorial (available in `OPENCV_ROOT/samples/python2/digits.py`).

We start with the necessary imports.

```python
import cv2
import numpy as np
import optunity
import optunity.metrics

# comment the line below when running the notebook yourself
%matplotlib inline
import matplotlib.pyplot as plt

# finally, we import some convenience functions to avoid bloating this tutorial
```

4.3. Examples
# these are available in the `OPTUNITY/notebooks` folder
import opencv_utility_functions as util

Load the data, which is available in the OPTUNITY/notebooks directory. This data file is a direct copy from OpenCV's example.

digits, labels = util.load_digits("opencv-digits.png")

loading "opencv-digits.png" ...

We preprocess the data and make a train/test split.

```
rand = np.random.RandomState(100)
shuffle = rand.permutation(len(digits))
digits, labels = digits[shuffle], labels[shuffle]

digits2 = map(util.deskew, digits)
samples = util.preprocess_hog(digits2)

train_n = int(0.9*len(samples))

digits_train, digits_test = np.split(digits2, [train_n])
samples_train, samples_test = np.split(samples, [train_n])
labels_train, labels_test = np.split(labels, [train_n])

test_set_img = util.mosaic(25, digits[train_n:])
plt.title('Test set digits')
plt.imshow(test_set_img, cmap='binary_r')
```

Now, it's time to construct classifiers. We will use a SVM classifier with an RBF kernel, i.e. \( \kappa(u, v) = \exp(-\gamma ||u - v||) \).
Such an SVM has two hyperparameters that must be optimized, namely the misclassification penalty $C$ and kernel parameter $\gamma$.

We start with an SVM with default parameters, which in this case means: $C = 1$ and $\gamma = 0.5$.

```python
model_default = util.SVM()
model_default.train(samples_train, labels_train)
vis_default, err_default = util.evaluate_model(model_default, digits_test, samples_test, labels_test)
plt.title('unoptimized SVM test set performance')
plt.imshow(vis_default)
```

Next, we will construct a model with optimized hyperparameters. First we need to build Optunity’s objective function. We will use 5-fold cross-validated error rate as loss function, which we will minimize.

```python
@optunity.cross_validated(x=samples_train, y=labels_train, num_folds=5)
def svm_error_rate(x_train, y_train, x_test, y_test, C, gamma):
    model = util.SVM(C=C, gamma=gamma)
```
We will use Optunity’s default solver to optimize the error rate given $0 < C < 5$ and $0 < \gamma < 10$ and up to 50 function evaluations. This may take a while.

```python
optimal_parameters, details, _ = optunity.minimize(svm_error_rate, num_evals=50,
                                                C=[0, 5], gamma=[0, 10])
# the above line can be parallelized by adding `pmap=optunity.pmap`
# however this is incompatible with IPython
print("Optimal parameters: C=%.3f, gamma=%.3f" % (optimal_parameters['C'], optimal_parameters['gamma']))
print("Cross-validated error rate: %.3f" % details.optimum)
```

Optimal parameters: C=1.798, gamma=6.072
Cross-validated error rate: 0.025

Finally, we train a model with the optimized parameters and determine its test set performance.

```python
model_opt = util.SVM(**optimal_parameters)
model_opt.train(samples_train, labels_train)
vis_opt, err_opt = util.evaluate_model(model_opt, digits_test, samples_test, labels_test)
plt.title('optimized SVM test set performance')
plt.imshow(vis_opt)
```

error: 2.80 %
confusion matrix:
```
[[50  0  0  0  0  0  0  0  0  0]
 [ 0 45  0  1  0  0  0  0  0  0]
 [ 0  0 44  1  0  0  0  1  1  1]
 [ 0  0  0 49  0  0  1  0  0  0]
 [ 0  0  1  0 53  0  2  0  0  0]
 [ 0  0  0  0  0 49  0  0  0  0]
 [ 0  0  0  0  0  0 48  0  0  0]
 [ 0  0  2  0  0  0  0 56  0  0]
 [ 0  1  0  0  0  1  0  0 51  0]
 [ 0  0  0  0  0  0  1  0  0 41]]
```

<matplotlib.image.AxesImage at 0x7fd3ee4da050>
Reduction in error rate by optimizing hyperparameters: 33.3%

sklearn: automated learning method selection and tuning

In this tutorial we will show how to use Optunity in combination with sklearn to classify the digit recognition data set available in sklearn. The cool part is that we will use Optunity to choose the best approach from a set of available learning algorithms and optimize hyperparameters in one go. We will use the following learning algorithms:

- k-nearest neighbour
- SVM
- Naive Bayes
- Random Forest

For simplicity, we will focus on a binary classification task, namely digit 3 versus digit 9. We start with the necessary imports and create the data set.

```python
import optunity
import optunity.metrics
import numpy as np

# k nearest neighbours
from sklearn.neighbors import KNeighborsClassifier
# support vector machine classifier
from sklearn.svm import SVC
# Naive Bayes
from sklearn.naive_bayes import GaussianNB
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
digits = load_digits()
n = digits.data.shape[0]

positive_digit = 3
negative_digit = 9

positive_idx = [i for i in range(n) if digits.target[i] == positive_digit]
negative_idx = [i for i in range(n) if digits.target[i] == negative_digit]

# add some noise to the data to make it a little challenging
original_data = digits.data[positive_idx + negative_idx, ...]
data = original_data + 5 * np.random.randn(original_data.shape[0], original_data.shape[1])
labels = [True] * len(positive_idx) + [False] * len(negative_idx)

For the SVM model we will let Optunity optimize the kernel family, choosing from linear, polynomial and RBF. We start by creating a convenience function for SVM training that handles this:

def train_svm(data, labels, kernel, C, gamma, degree, coef0):
    """A generic SVM training function, with arguments based on the chosen kernel.""
    if kernel == 'linear':
        model = SVC(kernel=kernel, C=C)
    elif kernel == 'poly':
        model = SVC(kernel=kernel, C=C, degree=degree, coef0=coef0)
    elif kernel == 'rbf':
        model = SVC(kernel=kernel, C=C, gamma=gamma)
    else:
        raise ArgumentError("Unknown kernel function: \$s\" % kernel)
    model.fit(data, labels)
    return model

Every learning algorithm has its own hyperparameters:

- k-NN: $1 < n_{neighbors} < 5$ the number of neighbours to use
- SVM: kernel family and misclassification penalty, we will make the penalty contingent on the family. Per kernel family, we have different hyperparameters:
  - polynomial kernel: $0 < C < 50$, $2 < degree < 5$ and $0 < coef0 < 1$
  - linear kernel: $0 < C < 2$,
  - RBF kernel: $0 < C < 10$ and $0 < gamma < 1$
- naive Bayes: no hyperparameters
- random forest:
  - $10 < n_{estimators} < 30$: number of trees in the forest
  - $5 < max_{features} < 20$: number of features to consider for each split

This translates into the following search space:

```
search = {'algorithm': {'k-nn': {'n_neighbors': [1, 5]},
                        'SVM': {'kernel': {'linear': {'C': [0, 2]},
                                          'rbf': {'gamma': [0, 1], 'C': [0, 10]},
                                          'poly': {'degree': [2, 5], 'C': [0, 50]},
                         ...
```
We also need an objective function that can properly orchestrate everything. We will choose the best model based on area under the ROC curve in 5-fold cross-validation.

```python
@optunity.cross_validated(x=data, y=labels, num_folds=5)
def performance(x_train, y_train, x_test, y_test, algorithm, n_neighbors=None, n_estimators=None, max_features=None, kernel=None, C=None, gamma=None, degree=None, coef0=None):
    # fit the model
    if algorithm == 'k-nn':
        model = KNeighborsClassifier(n_neighbors=int(n_neighbors))
        model.fit(x_train, y_train)
    elif algorithm == 'SVM':
        model = train_svm(x_train, y_train, kernel, C, gamma, degree, coef0)
    elif algorithm == 'naive-bayes':
        model = GaussianNB()
        model.fit(x_train, y_train)
    elif algorithm == 'random-forest':
        model = RandomForestClassifier(n_estimators=int(n_estimators),
                                        max_features=int(max_features))
        model.fit(x_train, y_train)
    else:
        raise ArgumentError('Unknown algorithm: %s' % algorithm)

    # predict the test set
    if algorithm == 'SVM':
        predictions = model.decision_function(x_test)
    else:
        predictions = model.predict_proba(x_test)[:, 1]
    return optunity.metrics.roc_auc(y_test, predictions, positive=True)
```

Let's do a simple test run of this fancy objective function.

```python
performance(algorithm='k-nn', n_neighbors=3)
```

0.9547920006472639

Seems okay! Now we can let Optunity do its magic with a budget of 300 tries.

```python
optimal_configuration, info, _ = optunity.maximize_structured(performance,
                                                              search_space=search,
                                                              num_evals=300)

print(optimal_configuration)
print(info.optimum)
```

```json
{  'kernel': 'poly',  'C': 38.5498046875,  'algorithm': 'SVM',  'degree': 3.88525390625,  'n_neighbors': None,  'n_estimators': None,  'max_features': None,  'coef0': 0.71826171875,  'gamma': None}
0.979302949566
```
Finally, let's make the results a little bit more readable. All dictionary items in `optimal_configuration` with value `None` can be removed.

```python
solution = dict((k, v) for k, v in optimal_configuration.items() if v is not None)
print('Solution
========

').join(map(lambda x: f'{x[0]}   {str(x[1])}', solution.items()))
```

```
Solution
========
kernel   poly
C        38.5498046875
degree   0.71826171875
algorithm SVM
```

**sklearn: SVM classification**

In this example we will use Optunity to optimize hyperparameters for a support vector machine classifier (SVC) in scikit-learn. We will learn a model to distinguish digits 8 and 9 in the MNIST data set in two settings:

- tune SVM with RBF kernel
- tune SVM with RBF, polynomial or linear kernel, that is choose the kernel function and its hyperparameters at once

```python
import optunity
import optunity.metrics

# comment this line if you are running the notebook
import sklearn.svm
import numpy as np

Create the data set: we use the MNIST data set and will build models to distinguish digits 8 and 9.

```python
from sklearn.datasets import load_digits
digits = load_digits()
n = digits.data.shape[0]

positive_digit = 8
negative_digit = 9

positive_idx = [i for i in range(n) if digits.target[i] == positive_digit]
negative_idx = [i for i in range(n) if digits.target[i] == negative_digit]

# add some noise to the data to make it a little challenging
original_data = digits.data[positive_idx + negative_idx, ...]
data = original_data + 5 * np.random.randn(original_data.shape[0], original_data.shape[1])
labels = [True] * len(positive_idx) + [False] * len(negative_idx)
```

First, let's see the performance of an SVC with default hyperparameters.

```python
# compute area under ROC curve of default parameters
@optunity.cross_validated(x=data, y=labels, num_folds=5)
def svm_default_auroc(x_train, y_train, x_test, y_test):
    model = sklearn.svm.SVC().fit(x_train, y_train)
    decision_values = model.decision_function(x_test)
```
Tune SVC with RBF kernel

In order to use Optunity to optimize hyperparameters, we start by defining the objective function. We will use 5-fold cross-validated area under the ROC curve. For now, let's restrict ourselves to the RBF kernel and optimize $C$ and $\gamma$.

We start by defining the objective function `svm_rbf_tuned_auroc()`, which accepts $C$ and $\gamma$ as arguments.

```python
# we will make the cross-validation decorator once, so we can reuse it later for the other tuning task
# by reusing the decorator, we get the same folds etc.
cv_decorator = optunity.cross_validated(x=data, y=labels, num_folds=5)

def svm_rbf_tuned_auroc(x_train, y_train, x_test, y_test, C, logGamma):
    model = sklearn.svm.SVC(C=C, gamma=10 ** logGamma).fit(x_train, y_train)
    decision_values = model.decision_function(x_test)
    auc = optunity.metrics.roc_auc(y_test, decision_values)
    return auc

svm_rbf_tuned_auroc = cv_decorator(svm_rbf_tuned_auroc)
# this is equivalent to the more common syntax below
# @optunity.cross_validated(x=data, y=labels, num_folds=5)
# def svm_rbf_tuned_auroc...

svm_rbf_tuned_auroc(C=1.0, logGamma=0.0)
```

Now we can use Optunity to find the hyperparameters that maximize AUROC.

```python
optimal_rbf_pars, info, _ = optunity.maximize(svm_rbf_tuned_auroc, num_evals=150, C=[0, 10], logGamma=[-5, 0])
# when running this outside of IPython we can parallelize via optunity.pmap
# optimal_rbf_pars, _, _ = optunity.maximize(svm_rbf_tuned_auroc, 150, C=[0, 10],
# gamma=[0, 0.1], pmap=optunity.pmap)

print("Optimal parameters: " + str(optimal_rbf_pars))
print("AUROC of tuned SVM with RBF kernel: %1.3f" % info.optimum)
```

Optimal parameters: {
'logGamma': -3.0716796875000005, 'C': 3.3025997497032007}
AUROC of tuned SVM with RBF kernel: 0.987

We can turn the call log into a pandas dataframe to efficiently inspect the solver trace.

```python
import pandas
df = optunity.call_log2dataframe(info.call_log)
```

Lets look at the best 20 sets of hyperparameters to make sure the results are somewhat stable.
Tune SVC without deciding the kernel in advance

In the previous part we choose to use an RBF kernel. Even though the RBF kernel is known to work well for a large variety of problems (and yielded good accuracy here), our choice was somewhat arbitrary.

We will now use Optunity’s conditional hyperparameter optimization feature to optimize over all kernel functions and their associated hyperparameters at once. This requires us to define the search space.

```
space = {'kernel': {'linear': {'C': [0, 2]},
                   'rbf': {'logGamma': [-5, 0], 'C': [0, 10]},
                   'poly': {'degree': [2, 5], 'C': [0, 5], 'coef0': [0, 2]}}
```

We will also have to modify the objective function to cope with conditional hyperparameters. The reason we need to do this explicitly is because scikit-learn doesn’t like dealing with `None` values for irrelevant hyperparameters (e.g. degree when using an RBF kernel). Optunity will set all irrelevant hyperparameters in a given set to `None`.

```
def train_model(x_train, y_train, kernel, C, logGamma, degree, coef0):
    """A generic SVM training function, with arguments based on the chosen kernel.""
    if kernel == 'linear':
        model = sklearn.svm.SVC(kernel=kernel, C=C)
    elif kernel == 'poly':
        model = sklearn.svm.SVC(kernel=kernel, C=C, degree=degree, coef0=coef0)
    elif kernel == 'rbf':
        model = sklearn.svm.SVC(kernel=kernel, C=C, gamma=10 ** logGamma)
    else:
        raise ArgumentError("Unknown kernel function: $s" % kernel)
    model.fit(x_train, y_train)
    return model
```

```
def svm_tuned_auroc(x_train, y_train, x_test, y_test, kernel='linear', C=0, logGamma=0, degree=0, coef0=0):
    model = train_model(x_train, y_train, kernel, C, logGamma, degree, coef0)
    decision_values = model.decision_function(x_test)
    return optunity.metrics.roc_auc(y_test, decision_values)
```

Now we are ready to go and optimize both kernel function and associated hyperparameters!

```
optimal_svm_pars, info, _ = optunity.maximize_structured(svm_tuned_auroc, space, num_evals=150)
p
```
sklearn: SVM regression

In this example we will show how to use Optunity to tune hyperparameters for support vector regression, more specifically:

- measure empirical improvements through nested cross-validation
- optimizing hyperparameters for a given family of kernel functions
- determining the optimal model without choosing the kernel in advance

```python
import math
import itertools
import optunity
import optunity.metrics
import sklearn.svm
```

We start by creating the data set. We use sklearn’s diabetes data.

```python
from sklearn.datasets import load_diabetes
diabetes = load_diabetes()
n = diabetes.data.shape[0]
data = diabetes.data
targets = diabetes.target
```

**Nested cross-validation**

Nested cross-validation is used to estimate generalization performance of a full learning pipeline, which includes optimizing hyperparameters. We will use three folds in the outer loop.

When using default hyperparameters, there is no need for an inner cross-validation procedure. However, if we want to include tuning in the learning pipeline, the inner loop is used to determine generalization performance with optimized hyperparameters.

We start by measuring generalization performance with default hyperparameters.

```python
# we explicitly generate the outer_cv decorator so we can use it twice
outer_cv = optunity.cross_validated(x=data, y=targets, num_folds=3)

def compute_mse_standard(x_train, y_train, x_test, y_test):
    """Computes MSE of an SVR with RBF kernel and default hyperparameters."
    model = sklearn.svm.SVR().fit(x_train, y_train)
predictions = model.predict(x_test)
return optunity.metrics.mse(y_test, predictions)

# wrap with outer cross-validation
compute_mse_standard = outer_cv(compute_mse_standard)
```

`compute_mse_standard()` returns a three-fold cross-validation estimate of MSE for an SVR with default hyperparameters.

```python
compute_mse_standard()
```

6190.481497665955
We will create a function that returns MSE based on optimized hyperparameters, where we choose a polynomial kernel in advance.

```python
def compute_mse_poly_tuned(x_train, y_train, x_test, y_test):
    """Computes MSE of an SVR with RBF kernel and optimized hyperparameters."""

    # define objective function for tuning
    @optunity.cross_validated(x=x_train, y=y_train, num_iter=2, num_folds=5)
    def tune_cv(x_train, y_train, x_test, y_test, C, degree, coef0):
        model = sklearn.svm.SVR(C=C, degree=degree, coef0=coef0, kernel='poly').fit(x_train, y_train)
        predictions = model.predict(x_test)
        return optunity.metrics.mse(y_test, predictions)

    # optimize parameters
    optimal_pars, _, _ = optunity.minimize(tune_cv, 150, C=[1000, 20000], degree=[2, 5], coef0=[0, 1])
    print("optimal hyperparameters: " + str(optimal_pars))
    tuned_model = sklearn.svm.SVR(kernel='poly', **optimal_pars).fit(x_train, y_train)
    predictions = tuned_model.predict(x_test)
    return optunity.metrics.mse(y_test, predictions)

# wrap with outer cross-validation
compute_mse_poly_tuned = outer_cv(compute_mse_poly_tuned)
```

`compute_mse_poly_tuned()` returns a three-fold cross-validation estimate of MSE for an SVR with RBF kernel with tuned hyperparameters $1000 < C < 20000$, $2 < \text{degree} < 5$ and $0 < \text{coef0} < 1$ with a budget of 150 function evaluations. Each tuple of hyperparameters is evaluated using twice-iterated 5-fold cross-validation.

```python
compute_mse_poly_tuned()
```

```text
optimal hyperparameters: {'C': 12078.673881034498, 'coef0': 0.5011052085197018, 'degree': 4.608902814634181}
optimal hyperparameters: {'C': 14391.165364583334, 'coef0': 0.17313151041666666, 'degree': 2.35826171875}
optimal hyperparameters: {'C': 11713.456382191061, 'coef0': 0.49836486667796476, 'degree': 4.616077904035152}
```

3047.035965991627

The polynomial kernel yields pretty good results when optimized, but maybe we can do even better with an RBF kernel.

```python
def compute_mse_rbf_tuned(x_train, y_train, x_test, y_test):
    """Computes MSE of an SVR with RBF kernel and optimized hyperparameters."""

    # define objective function for tuning
    @optunity.cross_validated(x=x_train, y=y_train, num_iter=2, num_folds=5)
    def tune_cv(x_train, y_train, x_test, y_test, C, gamma):
        model = sklearn.svm.SVR(C=C, gamma=gamma).fit(x_train, y_train)
        predictions = model.predict(x_test)
        return optunity.metrics.mse(y_test, predictions)

    # optimize parameters
    optimal_pars, _, _ = optunity.minimize(tune_cv, 150, C=[1, 100], gamma=[0, 50])
    print("optimal hyperparameters: " + str(optimal_pars))
```

```text
optimal hyperparameters: {'C': 1.4391165364583334, 'gamma': 0.017313151041666666}
```

56 Chapter 4. Indices and tables
tuned_model = sklearn.svm.SVR(**optimal_pars).fit(x_train, y_train)
predictions = tuned_model.predict(x_test)
return optunity.metrics.mse(y_test, predictions)

# wrap with outer cross-validation
compute_mse_rbf_tuned = outer_cv(compute_mse_rbf_tuned)

compute_mse_rbf_tuned() returns a three-fold cross-validation estimate of MSE for an SVR with RBF kernel with tuned hyperparameters \(1 < C < 100\) and \(0 < \gamma < 5\) with a budget of 150 function evaluations. Each tuple of hyperparameters is evaluated using twice-iterated 5-fold cross-validation.

optimal hyperparameters: {'C': 21.654003906250026, 'gamma': 16.536188056152554}
optimal hyperparameters: {'C': 80.89867187499999, 'gamma': 3.2346692538501784}
optimal hyperparameters: {'C': 19.35431640625002, 'gamma': 22.083848774716085}

2990.8572696483493

Woop! Seems like an RBF kernel is a good choice. An optimized RBF kernel leads to a 50% reduction in MSE compared to the default configuration.

**Determining the kernel family during tuning**

In the previous part we’ve seen that the choice of kernel and its parameters significantly impact performance. However, testing every kernel family separately is cumbersome. It’s better to let Optunity do the work for us.

Optunity can optimize conditional search spaces, here the kernel family and depending on which family the hyperparameterization (\(\gamma, \text{ degree}, \text{ coef0}, \ldots\)). We start by defining the search space (we will try the linear, polynomial and RBF kernel).

```python
space = {'kernel': {'linear': {'C': [0, 100]},
                 'rbf': {'gamma': [0, 50], 'C': [1, 100]},
                 'poly': {'degree': [2, 5], 'C': [1000, 20000], 'coef0': [0, 1]}
                }
}
```

Now we do nested cross-validation again.

```python
def compute_mse_all_tuned(x_train, y_train, x_test, y_test):
    """Computes MSE of an SVR with RBF kernel and optimized hyperparameters."""

    # define objective function for tuning
    @optunity.cross_validated(x=x_train, y=y_train, num_iter=2, num_folds=5)
def tune_cv(x_train, y_train, x_test, y_test, kernel, C, gamma, degree, coef0):
        if kernel == 'linear':
            model = sklearn.svm.SVR(kernel=kernel, C=C)
        elif kernel == 'poly':
            model = sklearn.svm.SVR(kernel=kernel, C=C, degree=degree, coef0=coef0)
        elif kernel == 'rbf':
            model = sklearn.svm.SVR(kernel=kernel, C=C, gamma=gamma)
        else:
            raise ArgumentError("Unknown kernel function: \$s\" % kernel)

        model.fit(x_train, y_train)
```
predictions = model.predict(x_test)
return optunity.metrics.mse(y_test, predictions)

# optimize parameters
optimal_pars, _, _ = optunity.minimize_structured(tune_cv, num_evals=150, search_space=space)

# remove hyperparameters with None value from optimal pars
for k, v in optimal_pars.items():
    if v is None: del optimal_pars[k]
print("optimal hyperparameters: " + str(optimal_pars))
tuned_model = sklearn.svm.SVR(**optimal_pars).fit(x_train, y_train)
predictions = tuned_model.predict(x_test)
return optunity.metrics.mse(y_test, predictions)

# wrap with outer cross-validation
compute_mse_all_tuned = outer_cv(compute_mse_all_tuned)

And now the kernel family will be optimized along with its hyperparameterization.

compute_mse_all_tuned()

optimal hyperparameters: {'kernel': 'rbf', 'C': 33.70116043112164, 'gamma': 16.3231735448437}
optimal hyperparameters: {'kernel': 'rbf', 'C': 58.11404170763237, 'gamma': 26.45349823062099}
optimal hyperparameters: {'kernel': 'poly', 'C': 14964.421875843143, 'coef0': 0.5127175861493205, 'degree': 4.045210787998622}

3107.625560844859

It looks like the RBF and polynomial kernel are competitive for this problem.

Additional examples:

**OpenCV**

**Optimizing a simple 2D parabola**

In this example, we will use Optunity to optimize a very simple function, namely a two-dimensional parabola.

More specifically, the objective function is \( f(x, y) = -x^2 - y^2 \).

The full code in Python:

```python
import optunity

def f(x, y):
    return -x**2 - y**2

optimal_pars, details, _ = optunity.maximize(f, num_evals=100, x=[-5, 5], y=[-5, 5])
```

For such simple functions we would use different solvers in practice, but the main idea remains.

To get a basic understanding of the way various solvers in Optunity work, we can optimize this function with all solvers and plot the resulting call logs. The source code for this example is available in `bin/examples/python/parabola.py`.
Below are plots of the traces and precision of various solvers in optimizing a 2D parabola. These results are averaged over 200 runs, in each run the solvers got a budget of 100 function evaluations with the box $x = (-5, 5), y = -(5, 5)$.

Note that both Nelder-Mead simplex and CMA-ES started from an initial solution close to the optimal one. The results on this toy problem do not generalize directly to a real tuning problem.

Fig. 4.4: Solver traces showing the evolution of their search for a single run on the 2D parabola.

scikit-learn

Support vector machine classification (SVC)

You can find an executable version of this example in `bin/examples/python/sklearn/svc.py` in your Optunity release.

In this example, we will train an SVC with RBF kernel using scikit-learn. In this case, we have to tune two hyperparameters: $C$ and $gamma$. We will use twice iterated 10-fold cross-validation to test a pair of hyperparameters.

In this example, we will use `optunity.maximize()`.

```python
import optunity
import optunity.metrics
import sklearn.svm

# score function: twice iterated 10-fold cross-validated accuracy
@optunity.cross_validated(x=data, y=labels, num_folds=10, num_iter=2)
def svm_auc(x_train, y_train, x_test, y_test, logC, logGamma):
    model = sklearn.svm.SVC(C=10 ** logC, gamma=10 ** logGamma).fit(x_train, y_train)
    decision_values = model.decision_function(x_test)
    return optunity.metrics.roc_auc(y_test, decision_values)

# perform tuning
hps, _, _ = optunity.maximize(svm_auc, num_evals=200, logC=[-5, 2], logGamma=[-5, 1])

# train model on the full training set with tuned hyperparameters
optimal_model = sklearn.svm.SVC(C=10 ** hps['logC'], gamma=10 ** hps['logGamma']).fit(data, labels)
```
Support vector machine regression (SVR)

You can find an executable version of this example in `bin/examples/python/sklearn/svc.py` in your Optunity release.

In this example, we will train an SVC with RBF kernel using scikit-learn. In this case, we have to tune two hyperparameters: \( C \) and \( \gamma \). We will use twice iterated 10-fold cross-validation to test a pair of hyperparameters.

In this example, we will use `optunity.maximize()`.

Spark

Logistic regression with Spark and MLlib

In this example, we will train a linear logistic regression model using Spark and MLlib. In this case, we have to tune one hyperparameter: \( \text{regParam} \) for L2 regularization. We will use 5-fold cross-validation to find optimal hyperparameters.

In this example, we will use `optunity.maximize()`, which by default uses Particle Swarms. Assumes the data is located in file `sample_svm_data.txt`, change it if necessary.

```python
# To run Spark with ipython use:
# IPYTHON=1 .bin/pyspark

from pyspark.mllib.classification import LogisticRegressionWithSGD
from pyspark.mllib.regression import LabeledPoint
from numpy import array

import optunity

# Load and parse the data
def parsePoint(line):
```
values = [float(x) for x in line.split(' ')]
return LabeledPoint(values[0], values[1:])

data = sc.textFile("sample_svm_data.txt")
parsedData = data.map(parsePoint).cache()

# cross-validation using optunity
@optunity.cross_validated(x=parsedData, num_folds=5, num_iter=1)
def logistic_l2_accuracy(x_train, x_test, regParam):
    # cache data to get reasonable speeds for methods like LogisticRegression and SVM
    xc = x_train.cache()
    # training logistic regression with L2 regularization
    model = LogisticRegressionWithSGD.train(xc, regParam=regParam, regType="l2")
    # making prediction on x_test
    yhat = x_test.map(lambda p: (p.label, model.predict(p.features)))
    # returning accuracy on x_test
    return yhat.filter(lambda (v, p): v == p).count() / float(x_test.count())

# using default maximize (particle swarm) with 10 evaluations, regularization between 0 and 10
optimal_pars, _, _ = optunity.maximize(logistic_l2_accuracy, num_evals=10,
    regParam=[0, 10])

# training model with all data for the best parameters
model = LogisticRegressionWithSGD.train(parsedData, regType="l2", **optimal_pars)

# prediction (in real application you would use here newData instead of parsedData)
yhat = parsedData.map(lambda p: (p.label, model.predict(p.features)))

Theano

These examples use Optunity in combination with Theano. Theano is available at https://github.com/Theano/Theano.

Logistic regression

In this example we will use Theano to train logistic regression models on a simple two-dimensional data set. We will use Optunity to tune the degree of regularization and step sizes (learning rate). This example requires Theano and NumPy.

We start with the necessary imports:

```python
import numpy
from numpy.random import multivariate_normal
rng = numpy.random

import theano
import theano.tensor as T

import optunity
import optunity.metrics
```

The next step is defining our data set. We will generate a random 2-dimensional data set. The generative procedure for the targets is as follows: $1 + 2 \times x_1 + 3 \times x_2 + \text{a noise term}$. We assign binary class labels based on whether or not the target value is higher than the mean target:
The next thing we need is a training function for LR models, based on Theano’s example:

```python
training_steps = 2000

def train_lr(x_train, y_train, regularization=0.01, step=0.1):
    x = T.matrix("x")
    y = T.vector("y")
    w = theano.shared(rng.randn(feats), name="w")
    b = theano.shared(0., name="b")

    # Construct Theano expression graph
    p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))  # Probability that target
    prediction = p_1

    xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1)  # Cross-entropy loss

    cost = xent.mean() + regularization * (w ** 2).sum()  # The cost to minimize

    gw, gb = T.grad(cost, [w, b])  # Compute the gradient of
    # (we shall return to these in a following section of
    # this tutorial)

    train = theano.function(
        inputs=[x, y],
        outputs=[prediction, xent],
        updates=((w, w - step * gw), (b, b - step * gb)))

    predict = theano.function(inputs=[x], outputs=prediction)

    for i in range(training_steps):
        train(x_train, y_train)
    return predict, w, b
```

Now that we know how to train, we can define a modeling strategy with default and tuned hyperparameters:

```python
def lr_untuned(x_train, y_train, x_test, y_test):
    predict, w, b = train_lr(x_train, y_train)
    yhat = predict(x_test)
    loss = optunity.metrics.logloss(y_test, yhat)
    brier = optunity.metrics.brier(y_test, yhat)
    return loss, brier

def lr_tuned(x_train, y_train, x_test, y_test):
    @optunity.cross_validated(x=x_train, y=y_train, num_folds=3)
    def inner_cv(x_train, y_train, x_test, y_test, regularization, step):
        predict, w, b = train_lr(x_train, y_train, regularization=regularization, step=step)
        yhat = predict(x_test)
        loss = optunity.metrics.logloss(y_test, yhat)
        brier = optunity.metrics.brier(y_test, yhat)
        return loss, brier

    optunity.minimize(inner_cv, regularization=0.01, step=0.1)

    predict, w, b = train_lr(x_train, y_train, regularization=regularization, step=step)
    yhat = predict(x_test)
    loss = optunity.metrics.logloss(y_test, yhat)
    brier = optunity.metrics.brier(y_test, yhat)
    return loss, brier
```
predict, _, _ = train_lr(x_train, y_train,
    regularization=regularization, step=step)

    yhat = predict(x_test)
    return optunity.metrics.logloss(y_test, yhat)

pars, _, _ = optunity.minimize(inner_cv, num_evals=50,
    regularization=[0.001, 0.05],
    step=[0.01, 0.2])

predict, w, b = train_lr(x_train, y_train, **pars)
    yhat = predict(x_test)
    loss = optunity.metrics.logloss(y_test, yhat)
    brier = optunity.metrics.brier(y_test, yhat)
    return loss, brier

Note that both modeling functions (train, predict, score) return two score measures (log loss and Brier score). We
will evaluate both modeling approaches using cross-validation and report both performance measures (see Cross-
validation). The cross-validation decorator:

outer_cv = optunity.cross_validated(x=data, y=labels, num_folds=3,
    aggregator=optunity.cross_validation.list_mean)

lr_untuned = outer_cv(lr_untuned)
lr_tuned = outer_cv(lr_tuned)

At this point, lr_untuned and lr_tuned will return a 3-fold cross-validation estimate of [logloss, Brier] when evaluated.

Full code

This example is available in detail in <optunity>/bin/examples/python/theano/logistic_regression.py. Typical output
of this script will look like:

true model: 1 + 2 * x1 + 3 * x2

evaluating untuned LR model
+ model: -0.18 + 1.679 * x1 + 2.045 * x2
  ++ log loss in test fold: 0.08921125198
  ++ Brier loss in test fold: 0.0786225946458

+ model: -0.36 + 1.449 * x1 + 2.247 * x2
  ++ log loss in test fold: 0.08217097905
  ++ Brier loss in test fold: 0.070741583014

+ model: -0.48 + 1.443 * x1 + 2.187 * x2
  ++ log loss in test fold: 0.10545356515
  ++ Brier loss in test fold: 0.0941325050801

evaluating tuned LR model
+ model: -0.66 + 2.354 * x1 + 3.441 * x2
  ++ log loss in test fold: 0.07508872472
  ++ Brier loss in test fold: 0.0718020866519

+ model: -0.44 + 2.648 * x1 + 3.817 * x2
  ++ log loss in test fold: 0.0718891792875
  ++ Brier loss in test fold: 0.0638209513581

+ model: -0.45 + 2.689 * x1 + 3.858 * x2
  ++ log loss in test fold: 0.06380803593

4.3. Examples 63
MATLAB

LS-SV MLab

These examples show use-cases of Optunity in combination with the LS-SV MLab software, available at http://www.esat.kuleuven.be/sista/lssvmlab/.

Least-squares SVM classification

You can find a MATLAB script for this example in `<optunity>/wrappers/matlab/example_lssvmlab/demo_classification.m` in your Optunity release.

In this example, we will perform nonlinear classification using LS-SVM with RBF kernel using the LS-SV MLab toolbox. In this case, we have to tune two hyperparameters: `gam` (regularization) and `sig2` (kernel bandwidth). We will 10-fold cross-validation to test a pair of hyperparameters.

In this example, we will use `optunity.minimize()` to minimize misclassification rate.

Start off by defining the objective function, we will generate a double helix for classification:

```matlab
r = 0.5;
hold = 1;

helix = @(x, theta, n) [x + r*cos(theta + linspace(0, 4*pi, n)); ...
    r*sin(theta + linspace(0, 4*pi, n)); linspace(0, h, n)]';

noise = 0.1;
ntr = 200;
pos = helix(0, 0, ntr);
neg = helix(0, pi, ntr);
X = [pos; neg] + noise*randn(2*ntr, 3);
Y = [ones(ntr, 1); -1*ones(ntr, 1)];
```

Now that we have training data, we can use LS-SV MLab and Optunity for tuning. In this example we use LS-SV MLab’s (verbose) functional API.

Tuning with LS-SV MLab’s built in procedure (in this case a combination of coupled simulated annealing and Nelder-Mead simplex):

```matlab
[lssvm_gam, lssvm_sig2] = tunelssvm((X,Y,'c',[],[],'RBF_kernel',), 'simplex',...
    'leaveoneoutlssvm', {'misclass'});
[alpha_lssvm, b_lssvm] = trainlssvm((X,Y,type,lssvm_gam,lssvm_sig2,'RBF_kernel',));
```

To do the same with Optunity, we must first define an objective function, let's say `demo_classification_misclass.m`:
function [ misclass ] = demo_classification_misclass( x_train, y_train, x_test, y_test, pars )

% train model
[alpha,b] = trainlssvm({x_train,y_train,'classification',pars.gam,pars.sig2,'RBF_kernel'});

% predict test data
Yt = simlssvm({x_train,y_train,'classification',pars.gam,pars.sig2,'RBF_kernel','preprocess'},(alpha,b),x_test);

% compute misclassification rate
misclass = sum(Yt ~= y_test)/numel(Yt);
end

The objective function seems quite involved but it’s essentially training a model, predicting test data and evaluating. The code to perform the actual tuning (we use 10-fold cross-validation):

```matlab
obj_fun = optunity.cross_validate(@demo_classification_misclass, X, 'y', Y, 'num_folds', 10);
opt_pars = optunity.minimize(obj_fun, 100, 'gam', [1, 100], 'sig2', [0.01, 2]);
[alpha_optunity,b_optunity] = trainlssvm({X,Y,type,opt_pars.gam,opt_pars.sig2,'RBF_kernel'});
```

Now we have two LS-SVM models, one tuned with LS-SVM’s own algorithms and one using Optunity. For this particular problem, their performance is close to equivalent. The figures below shows test set predictions for both models.

Fig. 4.6: Test set predictions of LS-SVM after tuning with LS-SVMlab.

Least-squares SVM regression

You can find a MATLAB script for this example in `<optunity>/wrappers/matlab/example_lssvmlab/demo_regression.m` in your Optunity release.
In this example, we will perform nonlinear regression using LS-SVM with RBF kernel using the LS-SVMlab toolbox. In this case, we have to tune two hyperparameters: \( \text{gam} \) (regularization) and \( \text{sig}^2 \) (kernel bandwidth). We will use 10-fold cross-validation to test a pair of hyperparameters.

In this example, we will use \texttt{optunity.minimize()} to minimize mean squared error (MSE).

Start off by defining the objective function:

\[
\text{fun} = \theta(X) \cdot \text{sinc}(X) + 0.03 \cdot \sin(15\pi X)
\]

This is a simple sinc function with a high frequency sine wave imposed. Now let's generate some noisy data:

\[
\begin{align*}
X &= \text{randn}(200, 1); \\
Y &= \text{fun}(X) + 0.1 \cdot \text{randn}(|X|, 1);
\end{align*}
\]

Now that we have the function we want to estimate, and data to use for it, we can use LS-SVMlab and Optunity for tuning. In this example we use LS-SVMlab's (verbose) functional API.

Tuning with LS-SVMlab's built in procedure (in this case a combination of coupled simulated annealing and Nelder-Mead simplex):

\[
\begin{align*}
\{\text{lssvm\_gam, lssvm\_sig}^2\} &= \text{tunelssvm}((X,Y, 'f', [], [], 'RBF\_kernel'), ... \\
&\quad \text{'simplex', 'leaveoneoutlssvm', ['mse']}); \\
\text{type} &= \text{'function estimation'}; \\
\{\alpha\_lssvm, b\_lssvm\} &= \text{trainlssvm}((X,Y,\text{type},\text{lssvm\_gam},\text{lssvm\_sig}^2, 'RBF\_kernel'));
\end{align*}
\]

To do the same with Optunity, we must first define an objective function, let's say \texttt{demo\_regression\_mse.m}:

\[
\begin{align*}
\text{fun} &= \theta(X) \cdot \text{sinc}(X) + 0.03 \cdot \sin(15\pi X); \\
\end{align*}
\]

This is a simple sinc function with a high frequency sine wave imposed. Now let's generate some noisy data:

\[
\begin{align*}
X &= \text{randn}(200, 1); \\
Y &= \text{fun}(X) + 0.1 \cdot \text{randn}(|X|, 1);
\end{align*}
\]

Now that we have the function we want to estimate, and data to use for it, we can use LS-SVMlab and Optunity for tuning. In this example we use LS-SVMlab's (verbose) functional API.

Tuning with LS-SVMlab's built in procedure (in this case a combination of coupled simulated annealing and Nelder-Mead simplex):

\[
\begin{align*}
\{\text{lssvm\_gam, lssvm\_sig}^2\} &= \text{tunelssvm}((X,Y, 'f', [], [], 'RBF\_kernel'), ... \\
&\quad \text{'simplex', 'leaveoneoutlssvm', ['mse']}); \\
\text{type} &= \text{'function estimation'}; \\
\{\alpha\_lssvm, b\_lssvm\} &= \text{trainlssvm}((X,Y,\text{type},\text{lssvm\_gam},\text{lssvm\_sig}^2, 'RBF\_kernel'));
\end{align*}
\]

To do the same with Optunity, we must first define an objective function, let's say \texttt{demo\_regression\_mse.m}:

\[
\begin{align*}
\text{fun} &= \theta(X) \cdot \text{sinc}(X) + 0.03 \cdot \sin(15\pi X); \\
\end{align*}
\]
The objective function seems quite involved but it’s essentially training a model, predicting test data and evaluating. The code to perform the actual tuning (we use 10-fold cross-validation):

```
obj_fun = optunity.cross_validate(@demo_regression_mse, X, 'y', Y, 'num_folds', 10);
opt_pars = optunity.minimize(obj_fun, 100, 'gam', [1, 30], 'sig2', [0.01, 1]);
[alpha_optunity, b_optunity] = trainlssvm({X, Y, type, ...
    opt_pars.gam, opt_pars.sig2, 'RBF_kernel'});
```

Now we have two LS-SVM models, one tuned with LS-SVM’s own algorithms and one using Optunity. For this particular problem, their performance is close to equivalent. The figure below shows test set predictions.

![Test set predictions and the true function.](image)

**R**

**Optimizing a simple 2D parabola**

In this example, we will use Optunity in R to maximize a very simple function, namely a two-dimensional parabola. More specifically, the objective function is \( f(x, y) = -x^2 - y^2 \).

The full code in R:

```
library(optunity)

f <- function(x, y) -x^2 - y^2
opt <- particle_swarm(f, x=c(-5, 5), y=c(-5, 5))
```
In this simple example we used particle swarms optimization with default settings (number of particles 5, number of generations 10).

An example with 10 particles and 15 generations:

```r
opt <- particle_swarm(f, x=c(-5, 5), y=c(-5, 5), num_particles=10, num_generations=15)
```

In addition to `particle_swarm` the R interface has `grid_search`, `random_search`, `nelder_mead`. For examples with them use R’s internal help, e.g. `?random_search`.

**Ridge Regression**

In this example, we will train a (linear) ridge regression. In this case, we have to tune one hyperparameters: `logC` (regularization). We will use twice iterated 5-fold cross-validation to test a hyperparameter and minimize mean squared error (`mean_se`):

```r
library(optunity)
## artificial data
N <- 50
x <- matrix(runif(N*5), N, 5)
y <- x[,1] + 0.5*x[,2] + 0.1*runif(N)

## ridge regression
regr <- function(x, y, xtest, ytest, logC) {
  ## regularization matrix
  C = diag(x=exp(logC), ncol(x))
  beta = solve(t(x) %*% x + C, t(x) %*% y)
  xtest %*% beta
}

## cv
sv <- cv.setup(x, y, score=mean_se, num_folds = 5, num_iter = 2)
res <- cv.particle_swarm(sv, regr, logC = c(-5, 5), maximize = FALSE)

## optimal value for logC:
res$solution$logC
```

Here we used default settings for `cv.particle_swarm`, see `?cv.particle_swarm` in R for details.

**SVM (e1071)**

In this example, we will train an SVM with RBF kernel and tune its hyperparameters, i.e. `cost` and `gamma` in log-scale. We will use particle swarms to maximize AUC of Precision-Recall (`auc_pr`) of twice iterated 5-fold cross-validation:

```r
library(optunity)
library(e1071)

## artificial data
N <- 100
x <- matrix(runif(N*5), N, 5)
y <- x[,1] + 0.5*x[,2]^2 + 0.1*runif(N) + sin(x[,3]) > 1.0

## SVM train and predict
svm.rbf <- function(x, y, xtest, ytest, logCost, logGamma) {
  m <- svm(x, y, kernel="radial", cost=exp(logCost), gamma=exp(logGamma),
           type="C-classification", probability=TRUE)
  yscore <- attr(predict(m, xtest, probability= T), "probabilities")[,"TRUE"]
}
```

---

**68 Chapter 4. Indices and tables**
Here we used default settings for `cv.particle_swarm`, see `?cv.particle_swarm` in R for details.

### xgboost

In this example, we will train a xgboost. There is only one hyper-parameter `max.depth`, which takes integer values. Therefore, we will use grid search to find `max.depth` that maximizes AUC-ROC in twice iterated 5-fold cross-validation:

```r
library(optunity)
library(xgboost)

data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test

learn_xg <- function(x, y, xtest, ytest, max.depth) {
    ## train xgboost:
    bst <- xgboost(data = x, label = y, max.depth = round(max.depth),
                    eta = 1, nthread = 2, nround = 2, objective = "binary:logistic")
    predict(bst, xtest)
}

## using grid search to find max.depth maximizing AUC-ROC
## grid for max.depth is 2:10
cv <- cv.setup(x = train$data, y = train$label, score=auc_roc, num_folds = 5, num_iter = 2)

## running cross-validation
res.grid <- cv.grid_search(cv, learn_xg, max.depth = 2:10, maximize=TRUE)

## best result:
res.grid$solution$max.depth

## train xgboost with the best max.depth
xbest <- xgboost(data = train$data, label = train$label,
                 max.depth = res.grid$solution$max.depth,
                 eta = 1, nthread = 2, nround = 2, objective = "binary:logistic")

## check AUC-ROC of the best model
pred <- predict(xbest, test$data)
auc_roc(test$label, pred)
```

Here we used grid of 2:10 for `max.depth`.
Notebooks

Here you can find a variety of notebooks to illustrate various features of Optunity. All of these examples are available in the OPTUNITY/notebooks folder.

To contribute examples, please send us a pull request on Github.

Using Other Environments

MATLAB

In this page we briefly discuss the MATLAB wrapper, which provides most of Optunity’s functionality. For a general overview, we recommend reading the User Guide.

For installation instructions, please refer to Installing Optunity.

Whenever the Python API requires a dictionary, we use a MATLAB struct. As MATLAB has no keyword arguments, we use varargs with the convention of \texttt{<name>, <value>}. Please refer to each function’s help for details on how to use it.

\textbf{Warning:} Since MATLAB does not have keyword arguments, every objective function you define should accept a single struct argument, whose fields represent hyperparameters.

Example, to model a function \( f(x, y) = x + y \):

\begin{verbatim}
f = @(pars) pars.x + pars.y;
\end{verbatim}

For MATLAB, the following main features are provided:

- Manual
- Optimizing hyperparameters
- Cross-validation

The file \texttt{<optunity>/wrappers/matlab/optunity_example.m} contains code for all the functionality that is available in MATLAB.

\textbf{Note:} If you experience any issues with the MATLAB wrapper, create a global variable \texttt{DEBUG_OPTUNITY} and set its value to true. This will show all debugging output:

\begin{verbatim}
global DEBUG_OPTUNITY
DEBUG_OPTUNITY=true;
\end{verbatim}

Please submit this output as an issue at https://github.com/claesenm/optunity/issues/new to obtain help.

If MATLAB hangs while using Optunity there is a communication issue. This should not occur, if you encounter this please file an issue at https://github.com/claesenm/optunity/issues/new. Currently the only way out of this is to kill MATLAB.

70 Chapter 4. Indices and tables
Manual

To obtain the manual of all solvers and a list of available solver names, use `optunity.manual()`. If you specify a solver name, its manual will be printed out.

You can verify whether or not a certain solver, for instance `cma-es`, is available like this:

```python
solvers = optunity.manual();
available = any(arrayfun(@(x) strcmp(x, 'cma-es'), solvers));
```

Optimizing hyperparameters

The MATLAB wrapper offers `optunity.maximize()`, `optunity.minimize()` and `optunity.optimize()`. These provide the same functionality as their Python equivalents.

The following code fragment shows how to optimize a simple function $f$ with Random Search within the box $-4 < x < 4$ and $-5 < y < 5$ and 200 evaluations:

```python
offx = rand();
offy = rand();
f = @(pars) - (offx+pars.x)^2 - (offy+pars.y)^2;
[max_solution, max_details, max_solver] = optunity.maximize(f, 200, ...
    'solver_name', 'random search', 'x', [-4, 4], 'y', [-5, 5]);
```

If you want to use `optunity.optimize()`, you must create the solver in advance using `optunity.make_solver()`. This will return an `optunity.Solver` object or return an error message:

```python
f = @(pars) pars.x + pars.y
rnd_solver = optunity.make_solver('random search', 'x', [-5, 5], ...
    'y', [-5, 5], 'num_evals', 400);
[rnd_solution, rnd_details] = optunity.optimize(rnd_solver, f);
```

Differences between Python and MATLAB version of `optimize`

The MATLAB version has an extra argument `constraints` where you can specify domain constraints (the MATLAB wrapper has no equivalent of `optunity.wrap_constraints()`). Constraints are communicated as a struct with the correct field names and corresponding values (more info at Domain constraints).

As an example, to add constraints $x < 3$ and $y \geq 0$, we use the following code:

```python
constraints = struct('ub_o', struct('x', 3), 'lb_c', struct('y', 0));
```

Information about the solvers can be obtained at Solver overview. To learn about the specific parameter names for each solver, check out the optunity.solvers.

Cross-validation

Two functions are provided for cross-validation:

- `optunity.generate_folds()`: generates a set of cross-validation folds
- `optunity.cross_validate()`: function decorator, to perform cross-validation with specified function

Both functions can deal with strata and clusters. You can specify these as a cell array of vectors of indices.
strata = {[1,2,3], [6,7,8,9]};
folds = optunity.generate_folds(20, 'num_folds', 10, 'num_iter', 2, 'strata', strata);

Cross-validation folds are returned as a matrix of \( num_{\text{instances}} \times num_{\text{iter}} \) with entries ranging from 1 to \( num_{\text{folds}} \) to indicate the fold each instance belongs to per iteration.

\texttt{optunity.cross_validate()} requires a function handle as its first argument. This is the function that will be decorated, which must have the following first arguments: \texttt{x_train} and \texttt{x_test} (if unsupervised) or \texttt{x_train}, \texttt{y_train}, \texttt{x_test}, \texttt{y_test}.

As an example, assume we have a function \texttt{optunity_cv_fun(x_train, x_test, pars)}:

```matlab
function [ result ] = cv_fun( x_train, x_test, pars )
disp('training set:');
disp(x_train');
disp('test set:');
disp(x_test');
result = -pars.x^2 - pars.y^2;
end
```

This must be decorated with cross-validation, for instance:

```matlab
x = (1:10)';
cvf = optunity.cross_validate(@optunity_cv_fun, x);
% evaluate the function: this will return a cross-validation result
performance = cvf(struct('x',1,'y',2));
```

\textbf{Warning:} After decorating with cross-validation, the objective function should have a single argument, namely a struct of hyperparameters.

### GNU Octave

In this page we briefly discuss the Octave wrapper, which provides most of Optunity’s functionality. For a general overview, we recommend reading the \textit{User Guide}.

For installation instructions, please refer to \textit{Installing Optunity}.

Whenever the Python API requires a dictionary, we use an Octave \textit{struct}. As Octave has no keyword arguments, we use \textit{varargs} with the convention of \texttt{<name>, <value>}. Please refer to each function’s help for details on how to use it.

\textbf{Warning:} Since Octave does not have keyword arguments, every objective function you define should accept a single struct argument, whose fields represent hyperparameters.

Example, to model a function \( f(x, y) = x + y \):

```matlab
f = @(pars) pars.x + pars.y;
```

For octave, the following main features are provided:

- \textit{Manual}
- \textit{Optimizing hyperparameters}
- \textit{Cross-validation}
The file `<optunity>/wrappers/octave/optunity/example/optunity_example.m` contains code for all the functionality that is available in octave.

**Note:** If you experience any issues with the octave wrapper, create a global variable `DEBUG_OPTUNITY` and set its value to true. This will show all debugging output:

```octave
global DEBUG_OPTUNITY
DEBUG_OPTUNITY=true;
```

Please submit this output as an issue at https://github.com/claesenm/optunity/issues/new to obtain help.

If octave hangs while using Optunity there is a communication issue. This should not occur, if you encounter this please file an issue at https://github.com/claesenm/optunity/issues/new. Currently the only way out of this is to kill octave.

**Manual**

To obtain the manual of all solvers and a list of available solver names, use `manual()`. If you specify a solver name, its manual will be printed out.

You can verify whether or not a certain solver, for instance `cma-es`, is available like this:

```octave
solvers = manual();
available = any(arrayfun(@(x) strcmp(x, 'cma-es'), solvers));
```

**Optimizing hyperparameters**

The octave wrapper offers `maximize()`, `minimize()` and `optimize()`. These provide the same functionality as their Python equivalents.

The following code fragment shows how to optimize a simple function $f$ with Random Search within the box $-4 < x < 4$ and $-5 < y < 5$ and 200 evaluations:

```octave
offx = rand();
offy = rand();
f = @(pars) - (offx+pars.x)^2 - (offy+pars.y)^2;
[max_solution, max_details, max_solver] = maximize(f, 200, ...
    'solver_name', 'random search', 'x', [-4, 4], 'y', [-5, 5]);
```

If you want to use `optimize()`, you must create the solver in advance using `make_solver()`. This will return a `Solver` object or return an error message:

```octave
f = @(pars) pars.x + pars.y
rnd_solver = make_solver('random search', 'x', [-5, 5], ...
    'y', [-5, 5], 'num_evals', 400);
[rnd_solution, rnd_details] = optimize(rnd_solver, f);
```

**Differences between Python and octave version of optimize**

The octave version has an extra argument `constraints` where you can specify domain constraints (the octave wrapper has no equivalent of `optunity.wrap_constraints()`). Constraints are communicated as a struct with the correct field names and corresponding values (more info at `Domain constraints`).

4.5. Using Other Environments
As an example, to add constraints \( x < 3 \) and \( y \geq 0 \), we use the following code:

\[
\text{constraints} = \text{struct('ub_o', struct('x', 3), 'lb_c', struct('y', 0));}
\]

Information about the solvers can be obtained at Solver overview. To learn about the specific parameter names for each solver, check out the optunity.solvers.

**Cross-validation**

Two functions are provided for cross-validation:

- `generate_folds()`: generates a set of cross-validation folds
- `cross_validate()`: function decorator, to perform cross-validation with specified function

Both functions can deal with strata and clusters. You can specify these as a cell array of vectors of indices:

\[
\text{strata} = \{ [1,2,3], [6,7,8,9] \};
\]
\[
\text{folds} = \text{generate_folds}(20, 'num_folds', 10, 'num_iter', 2, 'strata', \text{strata});
\]

Cross-validation folds are returned as a matrix of `num_instances \times num_iter` with entries ranging from 1 to `num_folds` to indicate the fold each instance belongs to per iteration.

`cross_validate()` requires a function handle as its first argument. This is the function that will be decorated, which must have the following first arguments: \( x_{\text{train}} \) and \( x_{\text{test}} \) (if unsupervised) or \( x_{\text{train}}, y_{\text{train}}, x_{\text{test}}, y_{\text{test}} \).

As an example, assume we have a function `optunity_cv_fun(x_{\text{train}}, x_{\text{test}}, \text{pars})`:

```matlab
function [ result ] = optunity_cv_fun( x_train, x_test, pars )
    disp('training set:');
    disp(x_train);
    disp('test set:');
    disp(x_test);
    result = -pars.x^2 - pars.y^2;
end
```

This must be decorated with cross-validation, for instance:

```matlab
x = (1:10)';
\text{cvf} = \text{cross_validate}(\text{@optunity_cv_fun}, x);
\% evaluate the function: this will \textbf{return} a cross-validation result
\text{performance} = \text{cvf}(\text{struct('x',1,'y',2)});
```

**Warning**: After decorating with cross-validation, the objective function should have a single argument, namely a struct of hyperparameters.

**Julia**

In this page we briefly discuss the Julia wrapper, which provides most of Optunity’s functionality. For a general overview, we recommend reading the User Guide.

For installation instructions, please refer to Installing Optunity and in particular to Installing Optunity for Julia.
Manual

For Julia the following main functions are available:

**minimize** \((f \mid \text{name}_\text{args})\)

Perform minimization of the function \(f\) based on the provided \text{name}_\text{args} parameters.

**Parameters**

- \(f\) – minimized function (can be an anonymous or any multi-argument function, or a function accepting \text{Dict})
- \text{name}_\text{args} – various arguments can be provided, such as \text{num_evals}, \text{solver_name}, but mandatory named arguments are related to the hard constraint on the domain of the function \(f\) (for instance \(x=[-5,5]\)).

**Returns** \(\text{vars}:\text{Dict} , \text{details}:\text{Dict}\)

If \(f\) is provided in the anonymous or multi-argument form then the order of the provided hard constraints matter. For instance if one defines:

\[
f = (x,y,z) \rightarrow (x-1)^2 + (y-2)^2 + (z+3)^4
\]

or:

\[
f(x,y,z) = (x-1)^2 + (y-2)^2 + (z+3)^4
\]

then we strictly require to provide among other named arguments the following hard constraints:

\[
x=[lb,ub], \ y=[lb,ub], \ z=[lb,ub]
\]

If one provides a function accepting \text{Dict} then the order of the provided hard constraints does not matter.

**maximize** \((f \mid \text{name}_\text{args})\)

Perform maximization of the function \(f\) based on the provided \text{name}_\text{args} parameters.

**Parameters**

- \(f\) – minimized function (can be an anonymous or any multi-argument function, or a function accepting \text{Dict})
- \text{name}_\text{args} – various arguments can be provided, such as \text{num_evals}, \text{solver_name}, but mandatory named arguments are related to the hard constraint on the domain of the function \(f\) (for instance \(x=[-5,5]\)).

**Examples**

```julia
using Base.Test

vars, details = minimize((x,y,z) -> (x-1)^2 + (y-2)^2 + (z+3)^4, x=[-5,5], y=[-5,5], z=[-5,5])
@test_approx_eq_eps vars"x" 1.0 1.
@test_approx_eq_eps vars"y" 2.0 1.
@test_approx_eq_eps vars"z" -3.0 1.

testit(x,y,z) = (x-1)^2 + (y-2)^2 + (z+3)^4

vars, details = minimize(testit, num_evals=10000, solver_name="grid search", x=[-5,5], y=[-5,5], z=[-5,5])
```
R

In this page we briefly discuss the R wrapper, which provides most of Optunity’s functionality. For a general overview, we recommend reading the User Guide.

For installation instructions, please refer to Installing Optunity. To use the package has to be loaded like all R packages by:

```r
library(optunity)
```

**Manual**

All functions in R wrapper have documentation available in R’s help system, e.g., `?cv.particle_swarm` gives help for cross-validation (CV) using particle swarms. To see the list of available functions type `optunity::<TAB><TAB>` in R’s command line.

For R following main functions are available:

- `cv.setup` for setting up CV, specifying data and the number of folds
- `cv.particle_swarm`, `cv.nelder_mead`, `cv.grid_search`, `cv.random_search` for optimizing hyperparameters in CV setting.
- `auc_roc` and `auc_pr` for calculating AUC of ROC and precision-recall curve.
- `early_rie` and `early_bedroc` for early discovery metrics.
- `mean_se` and `mean_ae` for regression loss functions, mean squared error and mean absolute error.
- `particle_swarm`, `nelder_mead`, `grid_search`, `random_search` for minimizing (or maximizing) regular functionals.
- `cv.run` performs a cross-validation evaluation for the given values of the hyperparameters (no optimization).

General workflow is to first create a CV object using `cv.setup` and then use it to run CV, by using functions like `cv.particle_swarms`, `cv.random_search` etc.

**Examples**

Please see the example pages, Ridge Regression and SVM (e1071). The R help pages for each function contain examples showing how to use them.
Optunity API

optunity submodules

Key features

Solvers

The optunity.solvers package bundles all solvers and related logic available in Optunity.

Module contents

Module to take care of registering solvers for use in the main Optunity API.

Main classes in this module:

- GridSearch
- RandomSearch
- NelderMead
- ParticleSwarm
- CMA_ES
- TPE
- Sobol
- BayesOpt

**Warning:** CMA_ES require DEAP and NumPy.

**Warning:** TPE require Hyperopt and NumPy.

**Warning:** BayesOpt require BayesOpt and NumPy.

*Module author: Marc Claesen*

Submodules

This package contains the following submodules:
**optunity.solvers.CMAES module**

```python
class optunity.solvers.CMAES.CMA_ES(num_generations, sigma=1.0, Lambda=None, **kwargs)
    Bases: optunity.solvers.util.Solver
```

Please refer to *CMA-ES* for details about this algorithm.

This solver uses an implementation available in the DEAP library [*DEAP2012*].

**Warning:** This solver has dependencies on DEAP and NumPy and will be unavailable if these are not met.

```python
lambda_
maximize(f, pmap=<built-in function map>)
    Maximizes f.

Parameters
```
• f (callable) – the objective function
• pmap (callable) – the map() function to use

```
Returns
```
• the arguments which optimize f
• an optional solver report, can be None

```python
minimize(f, pmap=<built-in function map>)
    Minimizes f.

Parameters
```
• f (callable) – the objective function
• pmap (callable) – the map() function to use

```
Returns
```
• the arguments which optimize f
• an optional solver report, can be None

```python
num_generations
optimize(f, maximize=True, pmap=<built-in function map>)
    Optimizes f.

Parameters
```
• f (callable) – the objective function
• maximize (boolean) – do we want to maximizes?
```

**Warning:** This solver is not explicitly constrained. The box constraints that are given are used for initialization, but solver may leave the specified region during iterations. If this is unacceptable, you must manually constrain the domain of the objective function prior to using this solver (cfr. *Domain constraints*).
- **pmap (callable)** – the map() function to use

**Returns**
- the arguments which optimize $f$
- an optional solver report, can be None

**sigma**

**start**

Returns the starting point for CMA-ES.

**static suggest_from_seed (num_evals, **kwargs)**

Verify that we can effectively make a solver. The doctest has to be skipped from automated builds, because DEAP may not be available and yet we want documentation to be generated.

```python
>>> s = CMA_ES.suggest_from_seed(30, x=1.0, y=-1.0, z=2.0)
>>> solver = CMA_ES(**s)
```

**optunity.solvers.GridSearch module**

**class optunity.solvers.GridSearch.GridSearch (**kwargs)**

Bases: `optunity.solvers.util.Solver`

Please refer to Grid Search for more information about this algorithm.

Exhaustive search over the Cartesian product of parameter tuples. Returns $x$ (the tuple which maximizes $f$) and its score $f(x)$.

```python
>>> s = GridSearch(x=[1,2,3], y=[-1,0,1])
>>> best_pars, _ = s.optimize(lambda x, y: x*y)
>>> best_pars['x']
3
>>> best_pars['y']
1
```

Initializes the solver with a tuple indicating parameter values.

```python
>>> s = GridSearch(x=[1,2], y=[3,4])
>>> s.parameter_tuples['x']
[1, 2]
>>> s.parameter_tuples['y']
[3, 4]
```

**static assign_grid_points (lb, ub, density)**

Assigns equally spaced grid points with given density in [ub, lb]. The bounds are always used. density must be at least 2.

**Parameters**
- **lb (float)** – lower bound of resulting grid
- **ub (float)** – upper bound of resulting grid
- **density (int)** – number of points to use

```python
>>> s = GridSearch.assign_grid_points(1.0, 2.0, 3)
>>> s
[1.0, 1.5, 2.0]
```

desc_brief = ‘finds optimal parameter values on a predefined grid’
desc_full = ['Retrieves the best parameter tuple on a predefined grid.', 'This function requires the grid to be specified via the Cartesian product of grid values.', 'The number of evaluations is the product of the length of all value vectors.']['maximize'](f, pmap=<built-in function map>)
Maximizes f.

Parameters

- f (callable) – the objective function
- pmap (callable) – the map() function to use

Returns

- the arguments which optimize f
- an optional solver report, can be None

minimize(f, pmap=<built-in function map>)
Minimizes f.

Parameters

- f (callable) – the objective function
- pmap (callable) – the map() function to use

Returns

- the arguments which optimize f
- an optional solver report, can be None

name = ‘grid search’

optimize(f, maximize=True, pmap=<built-in function map>)
Optimizes f.

Parameters

- f (callable) – the objective function
- maximize (boolean) – do we want to maximize?
- pmap (callable) – the map() function to use

Returns

- the arguments which optimize f
- an optional solver report, can be None

parameter_tuples
Returns the possible values of every parameter.

static suggest_from_box(num_evals, **kwargs)
Creates a GridSearch solver that uses less than num_evals evaluations within given bounds (lb, ub). The bounds are first tightened, resulting in new bounds covering 99% of the area. The resulting solver will use an equally spaced grid with the same number of points in every dimension. The amount of points that is used is per dimension is the nth root of num_evals, rounded down, where n is the number of hyperparameters.

```python
>>> s = GridSearch.suggest_from_box(30, x=[0, 1], y=[-1, 0], z=[-1, 1])
>>> s['x']
[0.005, 0.5, 0.995]
>>> s['y']
[-0.995, -0.5, -0.005]
>>> s['z']
[-0.99, 0.0, 0.99]
```
Verify that we can effectively make a solver from box.

```python
>>> s = GridSearch.suggest_from_box(30, x=[0, 1], y=[-1, 0], z=[-1, 1])
>>> solver = GridSearch(**s)
```

**optunity.solvers.NelderMead module**

```python
s = NelderMead(x=1, y=1, xtol=1e-8)
>>> best_pars, _ = s.optimize(lambda x, y: -x**2 - y**2)
>>> [math.fabs(best_pars['x']) < 1e-8, math.fabs(best_pars['y']) < 1e-8]
[True, True]
```

Initializes the solver with a tuple indicating parameter values.

```python
s = NelderMead(x=1, ftol=2)
>>> s.start
{'x': 1}
>>> s.ftol
2
```

**Warning:** This solver is not explicitly constrained. The box constraints that are given are used for initialization, but solver may leave the specified region during iterations. If this is unacceptable, you must manually constrain the domain of the objective function prior to using this solver (cfr. Domain constraints).

**desc_brief** = ‘simplex method for unconstrained optimization’

**desc_full** = ['Simplex method for unconstrained optimization', 'The simplex algorithm is a simple way to optimize a fairly general function. It works well when the objective function is smooth and the number of parameters is not too high.', 'In this context, the algorithm is a generalization of the classical simplex method popular for constrained optimization problems.', '- **ftol** :: accuracy up to which to optimize the function (default 1e-4)']

**ftol**

Returns the tolerance.

**max_iter**

Returns the maximum number of iterations.

**maximize** (f, pmap=<built-in function map>)

Maximizes f.

Parameters

- **f** (callable) – the objective function
- **pmap** (callable) – the map() function to use

Returns

- the arguments which optimize f
- an optional solver report, can be None

**minimize** (f, pmap=<built-in function map>)

Minimizes f.
Parameters

• \( f \) (callable) – the objective function
• \( pmap \) (callable) – the map() function to use

Returns

• the arguments which optimize \( f \)
• an optional solver report, can be None

name = 'nelder-mead'

\[\text{optimize}(f, \text{maximize=True}, pmap=<\text{built-in function map}>)\]

Optimizes \( f \).

Parameters

• \( f \) (callable) – the objective function
• \( \text{maximize} \) (boolean) – do we want to maximizes?
• \( pmap \) (callable) – the map() function to use

Returns

• the arguments which optimize \( f \)
• an optional solver report, can be None

static \text{reflect}(x0, xn1, alpha)
static \text{scale}(vertex, coeff)
static \text{simplex_center}(vertices)
static \text{sort_vertices}(vertices, values)

\text{start}

Returns the starting point.

static \text{suggest_from_seed}(num_evals, **kwargs)

Verify that we can effectively make a solver.

\[\text{NelderMead.suggest_from_seed}({30, \text{x}=1.0, \text{y}=-1.0, \text{z}=2.0})\]
\[\text{NelderMead(**s)}\]

\text{optunity.solvers.ParticleSwarm module}

class optunity.solvers.ParticleSwarm.ParticleSwarm(num_particles, num_generations,
max_speed=None, phi1=1.5, phi2=2.0, **kwargs)

Bases: optunity.solvers.util.Solver

Please refer to \text{Particle Swarm Optimization} for details on this algorithm.

Initializes a PSO solver.

Parameters

• \text{num_particles} (int) – number of particles to use
• \text{num_generations} (int) – number of generations to use
• max_speed (float or None) – maximum velocity of each particle
• phi1 (float) – parameter used in updating position based on local best
• phi2 (float) – parameter used in updating position based on global best
• kwargs (‘name’: [lb, ub], ..) – box constraints for each hyperparameter

The number of function evaluations it will perform is num_particles*num_generations. The search space is rescaled to the unit hypercube before the solving process begins.

```python
>>> solver = ParticleSwarm(num_particles=10, num_generations=5, x=[-1, 1], y=[0, 2])
>>> solver.bounds['x']
[-1, 1]
>>> solver.bounds['y']
[0, 2]
>>> solver.num_particles
10
>>> solver.num_generations
5
```

**Warning:** This solver is not explicitly constrained. The box constraints that are given are used for initialization, but solver may leave the specified region during iterations. If this is unacceptable, you must manually constrain the domain of the objective function prior to using this solver (cfr. *Domain constraints*).

class Particle (position, speed, best, fitness, best_fitness)
  Constructs a Particle.

  clone ()
    Clones this Particle.

ParticleSwarm.bounds
ParticleSwarm.desc_brief = ‘particle swarm optimization’
ParticleSwarm.desc_full = ['Maximizes the function using particle swarm optimization.', 'This is a two-phase approach:', '1. ... constraints are lists [lb, ub]']
ParticleSwarm.generate ()
  Generate a new Particle.

ParticleSwarm.max_speed

ParticleSwarm.maximize (f, pmap=<built-in function map>)
  Maximizes f.

  Parameters
    • f (callable) – the objective function
    • pmap (callable) – the map() function to use

  Returns
    • the arguments which optimize f
    • an optional solver report, can be None

ParticleSwarm.minimize (f, pmap=<built-in function map>)
  Minimizes f.
Parameters

- \( f(\text{callable}) \) – the objective function
- \( \text{pmap}(\text{callable}) \) – the map() function to use

Returns

- the arguments which optimize \( f \)
- an optional solver report, can be None

ParticleSwarm.name = 'particle swarm'
ParticleSwarm.num_generations
ParticleSwarm.num_particles
ParticleSwarm.optimize \((f, \text{maximize=True, pmap=<built-in function map>})\)

Optimizes \( f \).

Parameters

- \( f(\text{callable}) \) – the objective function
- \( \text{maximize}(\text{boolean}) \) – do we want to maximizes?
- \( \text{pmap}(\text{callable}) \) – the map() function to use

Returns

- the arguments which optimize \( f \)
- an optional solver report, can be None

ParticleSwarm.particle2dict \((\text{particle})\)
ParticleSwarm.phi1
ParticleSwarm.phi2
ParticleSwarm.smax
ParticleSwarm.smin
ParticleSwarm.sobolseed

static ParticleSwarm.suggest_from_box \((\text{num_evals}, **\text{kwargs})\)

Create a configuration for a ParticleSwarm solver.

Parameters

- \( \text{num_evals}(\text{int}) \) – number of permitted function evaluations
- \( \text{kwargs}((\text{\text{'param': [lb, ub], ..) - box constraints}}\)

```python
>>> config = ParticleSwarm.suggest_from_box(200, x=[-1, 1], y=[0, 1])
>>> config['x']
[-1, 1]
>>> config['y']
[0, 1]
>>> config['num_particles'] > 0
True
>>> config['num_generations'] > 0
True
>>> solver = ParticleSwarm(**config)
>>> solver.bounds['x']
[-1, 1]
>>> solver.bounds['y']
[0, 1]
```
ParticleSwarm. **updateParticle** *(part, best, phi1, phi2)*

Update the particle.

**optunity.solvers.RandomSearch module**

class **optunity.solvers.RandomSearch.RandomSearch**(num_evals, **kwargs)

**Bases:** `optunity.solvers.util.Solver`

Please refer to Random Search for more details about this algorithm.

Initializes the solver with bounds and a number of allowed evaluations. **kwargs** must be a dictionary of parameter-bound pairs representing the box constraints. Bounds are a 2-element list: [lower_bound, upper_bound].

```python
>>> s = RandomSearch(x=[0, 1], y=[-1, 2], num_evals=50)
>>> s.bounds['x']
[0, 1]
>>> s.bounds['y']
[-1, 2]
>>> s.num_evals
50
```

**bounds**

Returns a dictionary containing the box constraints.

**desc_brief** = ‘random parameter tuples sampled uniformly within box constraints’


**lower**

Returns the lower bound of par.

**maximize** *(f, pmap=<built-in function map>)*

Maximizes f.

**Parameters**

- **f** *(callable)* – the objective function
- **pmap** *(callable)* – the map() function to use

**Returns**

- the arguments which optimize f
- an optional solver report, can be None

**minimize** *(f, pmap=<built-in function map>)*

Minimizes f.

**Parameters**

- **f** *(callable)* – the objective function
- **pmap** *(callable)* – the map() function to use

**Returns**

- the arguments which optimize f
- an optional solver report, can be None
num_evals
Returns the number of evaluations this solver may do.

optimize (f, maximize=True, pmap=<built-in function map>)
Optimizes f.

Parameters
• f (callable) – the objective function
• maximize (boolean) – do we want to maximize?
• pmap (callable) – the map() function to use

Returns
• the arguments which optimize f
• an optional solver report, can be None

static suggest_from_box (num_evals, **kwargs)
Creates a RandomSearch solver that uses num_evals evaluations within given bounds (lb, ub). The bounds are first tightened, resulting in new bounds covering 99% of the area.

```
>>> s = RandomSearch.suggest_from_box(30, x=[0, 1], y=[-1, 0], z=[-1, 1])
>>> s['x']
[0.005, 0.995]
>>> s['y']
[-0.995, -0.005]
>>> s['z']
[-0.99, 0.99]
>>> s['num_evals']
30
```

Verify that we can effectively make a solver from box.

```
>>> s = RandomSearch.suggest_from_box(30, x=[0, 1], y=[-1, 0], z=[-1, 1])
>>> solver = RandomSearch(**s)
```

upper
Returns the upper bound of par.

optunity.solvers.Sobol module
class optunity.solvers.Sobol
Sobol (num_evals, seed=None, skip=None, **kwargs)
Bases: optunity.solvers.util.Solver
Please refer to Sobol sequences for details on this algorithm.

Initializes a Sobol sequence solver.

Parameters
• num_evals (int) – number of evaluations to use
• skip (int or None) – the number of initial elements of the sequence to skip, if None a random skip is generated
• kwargs ({'name': [lb, ub], ..}) – box constraints for each hyperparameter

The search space is rescaled to the unit hypercube before the solving process begins.
static bitwise_xor(a, b)
    Returns the bitwise_xor of a and b as a bitstring.

    Parameters
    • a (int) – first number
    • b (int) – second number

    >>> Sobol.bitwise_xor(13, 17)
    28
    >>> Sobol.bitwise_xor(31, 5)
    26

bounds

desc_brief = ‘sample the search space using a Sobol sequence’
desc_full = ['Generates a Sobol sequence of points to sample in the search space.', '‘A Sobol sequence is a low discrepancy sequence.’, 'This sampling method should be preferred over sampling uniformly at random."

static i4_bit_hi1(n)
    Returns the position of the high 1 bit base 2 in an integer.

    Parameters n (int) – the integer to be measured
    Returns (int) the number of bits base 2

    This was taken from http://people.sc.fsu.edu/~jburkardt/py_src/sobol/sobol.html Licensing:

    This code is distributed under the MIT license.

Modified: 22 February 2011

Author: Original MATLAB version by John Burkardt. PYTHON version by Corrado Chisari

static i4_bit_lo0(n)
    Returns the position of the low 0 bit base 2 in an integer.

    Parameters n (int) – the integer to be measured
    Returns (int) the number of bits base 2

    This was taken from http://people.sc.fsu.edu/~jburkardt/py_src/sobol/sobol.html Licensing:

    This code is distributed under the MIT license.

Modified: 22 February 2011

Author: Original MATLAB version by John Burkardt. PYTHON version by Corrado Chisari

static i4_sobol(dim_num, seed)
    Generates a new quasi-random Sobol vector with each call.

    Parameters
    • dim_num (int) – number of dimensions of the Sobol vector
    • seed (int) – the seed to use to generate the Sobol vector

    Returns the next quasirandom vector and the next seed to use

    This was taken from http://people.sc.fsu.edu/~jburkardt/py_src/sobol/sobol.html Licensing:

    This code is distributed under the MIT license.
static i4_sobol_generate (m, n, skip)
Generates a Sobol sequence.

Parameters
• m(int) – the number of dimensions (our implementation supports up to 40)
• n(int) – the length of the sequence to generate
• skip(int) – the number of initial elements in the sequence to skip

Returns a list of length n containing m-dimensional points of the Sobol sequence

static maxdim()
The maximum dimensionality that we currently support.

maximize (f, pmap=<built-in function map>)
Maximizes f.

Parameters
• f(callable) – the objective function
• pmap(callable) – the map() function to use

Returns
• the arguments which optimize f
• an optional solver report, can be None

minimize (f, pmap=<built-in function map>)
Minimizes f.

Parameters
• f(callable) – the objective function
• pmap(callable) – the map() function to use

Returns
• the arguments which optimize f
• an optional solver report, can be None

name = 'sobol'
num_evals
optimize (f, maximize=True, pmap=<built-in function map>)
Optimizes f.

Parameters
• f(callable) – the objective function
• maximize(boolean) – do we want to maximizes?
• pmap(callable) – the map() function to use

Returns
• the arguments which optimize f
• an optional solver report, can be None

```
skip
```

```
static suggest_from_box(num_evals, **kwargs)
Create a configuration for a Sobol solver.

Parameters

• num_evals (int) – number of permitted function evaluations
• kwargs (**kwargs) – box constraints
```

Verify that we can effectively make a solver from box.

```
>>> s = Sobol.suggest_from_box(30, x=[0, 1], y=[-1, 0], z=[-1, 1])
>>> solver = Sobol(**s)
```

```
optunity.solvers.TPE module
```

class optunity.solvers.TPE.TPE(num_evals=100, seed=None, **kwargs)

Bases: optunity.solvers.util.Solver

This solver implements the Tree-structured Parzen Estimator, as described in [TPE2011]. This solver uses Hyperopt in the back-end and exposes the TPE estimator with uniform priors.

Please refer to *Tree-structured Parzen Estimator* for details about this algorithm.

Initialize the TPE solver.

```
Parameters

• num_evals (int) – number of permitted function evaluations
• seed (double) – the random seed to be used
• kwargs (**kwargs) – box constraints for each hyperparameter

bounds

maximize (f, pmap=<built-in function map>)
Maximizes f.

Parameters

• f (callable) – the objective function
• pmap (callable) – the map() function to use

Returns

• the arguments which optimize f
• an optional solver report, can be None

minimize (f, pmap=<built-in function map>)
Minimizes f.

Parameters

• f (callable) – the objective function
• pmap (callable) – the map() function to use

Returns
• the arguments which optimize \( f \)
• an optional solver report, can be None

num_evals

\texttt{optimize}( f, \texttt{maximize=True, pmap=<built-in function map>})
Optimizes \( f \).

\textbf{Parameters}

• \( f \) (\texttt{callable}) – the objective function
• \texttt{maximize} (\texttt{boolean}) – do we want to maximizes?
• \texttt{pmap} (\texttt{callable}) – the map() function to use

\textbf{Returns}

• the arguments which optimize \( f \)
• an optional solver report, can be None

seed

\texttt{static suggest_from_box}(num_evals, **kwargs)
Verify that we can effectively make a solver from box.

```
>>> s = TPE.suggest_from_box(30, x=[0, 1], y=[-1, 0], z=[-1, 1])
>>> solver = TPE(**s)
```

\texttt{optunity.solvers.solver\_registry module}

Module to take care of registering solvers for use in the main Optunity API.

Main functions in this module:

• \texttt{register\_solver()}
• \texttt{manual()}
• \texttt{get()}

\textit{Module author: Marc Claesen}

\texttt{optunity.solvers.solver\_registry.get}(solver\_name)
Returns the class of the solver registered under given name.

\textbf{Parameters} solver\_name – name of the solver

\textbf{Returns} the solver class

\texttt{optunity.solvers.solver\_registry.manual()}
Returns the general manual of Optunity, with a brief introduction of all registered solvers.

\textbf{Returns} the manual as a list of strings (lines)

\texttt{optunity.solvers.solver\_registry.register\_solver}(name, desc\_brief, desc\_full)
Class decorator to register a \texttt{optunity.solvers.Solver} subclass in the registry. Registered solvers will be available through Optunity’s main API functions, e.g. \texttt{optunity.make\_solver()} and \texttt{optunity.manual()}.

\textbf{Parameters}

• \texttt{name} – name to register the solver with
• **desc_brief** – one-line description of the solver

• **desc_full** – extensive description and manual of the solver returns a list of strings representing manual lines

**Returns** a class decorator to register solvers in the solver registry

The resulting class decorator attaches attributes to the class before registering:

- `_name` the name using which the solver is registered
- `_desc_full` extensive description and manual as list of strings (lines)
- `_desc_brief` attribute with one-line description

These attributes will be available as class properties.

```python
optunity.solvers.solver_registry.solver_names()
```

Returns a list of all registered solvers.

### optunity.solvers.util module

**class** `optunity.solvers.util.Solver`

**Bases:** `abc.SolverBase`

Base class of all Optunity solvers.

**maximize**(*f*, *pmap=<built-in function map>*)

Maximizes *f*.

**Parameters**

- **f** (*callable*) – the objective function
- **pmap** (*callable*) – the map() function to use

**Returns**

- the arguments which optimize *f*
- an optional solver report, can be None

**minimize**(*f*, *pmap=<built-in function map>*)

Minimizes *f*.

**Parameters**

- **f** (*callable*) – the objective function
- **pmap** (*callable*) – the map() function to use

**Returns**

- the arguments which optimize *f*
- an optional solver report, can be None

**optimize**(*f*, **maximize=** *True*, **pmap=** *<built-in function map>*)

Optimizes *f*.

**Parameters**

- **f** (*callable*) – the objective function
- **maximize** (*boolean*) – do we want to maximizes?
• **pmap** (*callable*) – the map() function to use

**Returns**

• the arguments which optimize \( f \)
• an optional solver report, can be None

**class** optunity.solvers.util.ThreadSafeQueue(*lst=None*)

**Bases:** object

Initializes a new object.

**Parameters**

**lst** (*list or None*) – initial content

**append** (*value*)

Acquires lock and appends value to the content.

```python
>>> q1 = ThreadSafeQueue()
>>> q1
[]
>>> q1.append(1)
[1]
```

**content**

**copy()**

Makes a deep copy of this ThreadSafeQueue.

```python
>>> q1 = ThreadSafeQueue([1,2,3])
>>> q2 = q1.copy()
>>> q2.append(4)
>>> q1
[1, 2, 3]
>>> q2
[1, 2, 3, 4]
```

**lock**

**optunity.solvers.util.scale_unit_to_bounds** (*seq, bounds*)

Scales all elements in seq (unit hypercube) to the box constraints in bounds.

**Parameters**

• **seq** (*iterable of \([lb, ub]\) pairs*) – the sequence in the unit hypercube to scale

• **bounds** – bounds to scale to

**Returns**

a list of scaled elements of *seq*

```python
>>> scale_unit_to_bounds([0.0, 0.5, 0.5, 1.0], [[-1.0, 2.0], [-2.0, 0.0], [0.0, 3.0], [0.0, 2.0]])
[-1.0, -1.0, 1.5, 2.0]
```

**optunity.solvers.util.score** (*value*)

General wrapper around objective function evaluations to get the score.

**Parameters**

**value** – output of the objective function

**Returns**

the score

If value is a scalar, it is returned immediately. If value is iterable, its first element is returned.
Shrinks the bounds. The new bounds will cover the fraction coverage.

```python
>>> [round(x, 3) for x in shrink_bounds([0, 1], coverage=0.99)]
[0.005, 0.995]
```

Generates a random uniform sample between bounds.

**Parameters**

- `bounds (dict {"name": [lb, ub], ..}` – the bounds we must adhere to

**Cross-validation**

This module contains various provisions for cross-validation.

The main functions in this module are:

- `cross_validated()`
- `generate_folds()`
- `strata_by_labels()`
- `random_permutation()`
- `mean()`
- `identity()`
- `list_mean()`

**Module author:** Marc Claesen

**optunity.cross_validation.select(collection, indices)**

Selects the subset specified by indices from collection.

```python
>>> select([0, 1, 2, 3, 4], [1, 3])
[1, 3]
```

**optunity.cross_validation.random_permutation(data)**

Returns a list containing a random permutation of `r` elements out of data.

**Parameters**

- `data` – an iterable containing the elements to permute over

**Returns**

returns a list containing permuted entries of `data`

**optunity.cross_validation.cross_validated(x, num_folds=10, y=None, strata=None, folds=None, num_iter=1, regenerate_folds=False, clusters=None, aggregator=<function mean>)**

Function decorator to perform cross-validation as configured.

**Parameters**

- `x` – data to be used for cross-validation
- `num_folds` – number of cross-validation folds (default 10)
- `y` – (optional) labels to be used for cross-validation. If specified, len(labels) must equal len(x)
- `strata` – (optional) strata to account for when generating folds. Strata signify instances that must be spread across folds. Not every instance must be in a stratum. Specify strata as a list of lists of instance indices.
• **folds** – (optional) prespecified cross-validation folds to be used (list of lists (iterations) of lists (folds)).

• **num_iter** – (optional) number of iterations to use (default 1)

• **regenerate_folds** – (optional) whether or not to regenerate folds on every evaluation (default false)

• **clusters** – (optional) clusters to account for when generating folds. Clusters signify instances that must be assigned to the same fold. Not every instance must be in a cluster. Specify clusters as a list of lists of instance indices.

• **aggregator** – function to aggregate scores of different folds (default: mean)

Returns a `cross_validated_callable` with the proper configuration.

This resulting decorator must be used on a function with the following signature (+ potential other arguments):

**Parameters**

• **x_train** *(iterable)* – training data

• **y_train** *(iterable)* – training labels (optional)

• **x_test** *(iterable)* – testing data

• **y_test** *(iterable)* – testing labels (optional)

`y_train` and `y_test` must be available if the `y` argument to this function is not None.

These 4 keyword arguments will be bound upon decoration. Further arguments will remain free (e.g. hyperparameter names).

```python
def f(x_train, x_test, a):
    return x_test[0] + a
```

The number of folds must be less than or equal to the size of the data.

```python
>>> data = list(range(5))
>>> @cross_validated(x=data, num_folds=5, folds=[[i] for i in range(5)], aggregator=identity)
... def f(x_train, x_test, a):
...     return x_test[0] + a
... f(a=1)
[1, 2, 3, 4, 5]
>>> f(1)
[1, 2, 3, 4, 5]
>>> f(a=2)
[2, 3, 4, 5, 6]
```

The number of labels (if specified) must match the number of data instances.

```python
>>> data = list(range(5))
>>> labels = list(range(3))
>>> @cross_validated(x=data, y=labels, num_folds=2)
... def f(x_train, x_test, a):
...     return x_test[0] + a
AssertionError
```

94 Chapter 4. Indices and tables
Optunity Documentation, Release 1.1.0

optunity.cross_validation.generate_folds (num_rows, num_folds=10, strata=None, clusters=None)

Generates folds for a given number of rows.

Parameters

• `num_rows` – number of data instances
• `num_folds` – number of folds to use (default 10)
• `strata` – (optional) list of lists to indicate different sampling strata. Not all rows must be in a stratum. The number of rows per stratum must be larger than or equal to num_folds.
• `clusters` – (optional) list of lists indicating clustered instances. Clustered instances must be placed in a single fold to avoid information leaks.

Returns a list of folds, each fold is a list of instance indices

```python
>>> folds = generate_folds(num_rows=6, num_folds=2, clusters=[[1, 2]], strata=[[3, 4]])
>>> len(folds)
2
>>> i1 = [idx for idx, fold in enumerate(folds) if 1 in fold]
>>> i2 = [idx for idx, fold in enumerate(folds) if 2 in fold]
>>> i1 == i2
True
>>> i3 = [idx for idx, fold in enumerate(folds) if 3 in fold]
>>> i4 = [idx for idx, fold in enumerate(folds) if 4 in fold]
>>> i3 == i4
False
```

Warning: Instances in strata are not necessarily spread out over all folds. Some folds may already be full due to clusters. This effect should be negligible.

optunity.cross_validation.strata_by_labels(labels)

Constructs a list of strata (lists) based on unique values of labels.

Parameters `labels` – iterable, identical values will end up in identical strata

Returns the strata, as a list of lists

optunity.cross_validation.mean(x)
optunity.cross_validation.identity(x)
optunity.cross_validation.list_mean(list_of_measures)

Computes means of consequent elements in given list.

Parameters `list_of_measures` (list) – a list of tuples to compute means from

Returns a list containing the means

This function can be used as an aggregator in `cross_validated()`, when multiple performance measures are being returned by the wrapped function.

```python
>>> list_mean([(1, 4), (2, 5), (3, 6)])
[2.0, 5.0]
```

optunity.cross_validation.mean_and_list(x)

Returns a tuple, the first element is the mean of x and the second is x itself.

This function can be used as an aggregator in `cross_validated()`.
>>> mean_and_list([1,2,3])
(2.0, [1, 2, 3])

Quality metrics

optunity.metrics.absolute_error(y, yhat)

Returns the maximal absolute error between y and yhat.

Parameters

• y – true function values
• yhat – predicted function values

Lower is better.

>>> absolute_error([0,1,2,3], [0,0,1,1])
2.0

optunity.metrics.accuracy(y, yhat)

Returns the accuracy. Higher is better.

Parameters

• y – true function values
• yhat – predicted function values

optunity.metrics.auc(curve)

Computes the area under the specified curve.

Parameters curve ([(x, y), ..]) – a curve, specified as a list of (x, y) tuples

See also:

optunity.score_functions.compute_curve()

optunity.metrics.brier(y, yhat, positive=True)

Returns the Brier score between y and yhat.

Parameters

• y – true function values
• yhat – predicted function values

Returns

$$\frac{1}{n} \sum_{i=1}^{n} [ (\hat{y} - y)^2 ]$$

yhat must be a vector of probabilities, e.g. elements in [0, 1]

Lower is better.

Note: This loss function should only be used for probabilistic models.

optunity.metrics.compute_curve(ys, decision_values, xfun, yfun, positive=True, pre-sorted=False)

Computes a curve based on contingency tables at different decision values.
Parameters

- `ys(iterable)` – true labels
- `decision_values(iterable)` – decision values
- `positive` – positive label
- `xfun(callable)` – function to compute x values, based on contingency tables
- `yfun(callable)` – function to compute y values, based on contingency tables
- `presorted(bool)` – whether or not ys and yhat are already sorted

Returns

the resulting curve, as a list of (x, y)-tuples

```python
optunity.metrics.contingency_table(ys, yhats, positive=True)
```
Computes a contingency table for given predictions.

Parameters

- `ys(iterable)` – true labels
- `yhats(iterable)` – predicted labels
- `positive` – the positive label

Returns

TP, FP, TN, FN

```python
>>> ys = [True, True, True, True, True, False]
>>> yhats = [True, True, False, False, False, True]
>>> tab = contingency_table(ys, yhats, 1)
>>> print(tab)
(2, 1, 0, 3)
```

```python
optunity.metrics.contingency_tables(ys, decision_values, positive=True, presorted=False)
```
Computes contingency tables for every unique decision value.

Parameters

- `ys(iterable)` – true labels
- `decision_values(iterable)` – decision values (higher = stronger positive)
- `positive` – the positive label
- `presorted(bool)` – whether or not ys and yhat are already sorted

Returns

a list of contingency tables (TP, FP, TN, FN) and the corresponding thresholds.

Contingency tables are built based on decision `decision_value >= threshold`.

The first contingency table corresponds with a (potentially unseen) threshold that yields all negatives.

```python
>>> y = [0, 0, 0, 0, 1, 1, 1, 1]
>>> d = [2, 2, 1, 1, 1, 2, 3, 3]
>>> tables, thresholds = contingency_tables(y, d, 1)
>>> print(tables)
[(0, 0, 4, 4), (2, 0, 4, 2), (3, 2, 2, 1), (4, 4, 0, 0)]
>>> print(thresholds)
[None, 3, 2, 1]
```

```python
optunity.metrics.error_rate(y, yhat)
```
Returns the error rate (lower is better).

Parameters
• \( y \) – true function values
• \( \text{yhat} \) – predicted function values

```python
>>> error_rate([0, 0, 1, 1], [0, 0, 0, 1])
0.25
```

call optunity.metrics.fbeta\((y, \text{yhat}, \beta, \text{positive}=\text{True})\)

Returns the \( F_\beta \)-score.

**Parameters**

• \( y \) – true function values
• \( \text{yhat} \) – predicted function values
• \( \beta \) \((\text{float} \ (\text{positive}))\) – the value for \( \beta \) to be used
• \( \text{positive} \) – the positive label

**Returns**

\[
(1 + \beta^2) \frac{\text{precision} \cdot \text{recall}}{\beta^2 \ast \text{precision} \ast \text{recall}}
\]


call optunity.metrics.logloss\((y, \text{yhat})\)

Returns the log loss between labels and predictions.

**Parameters**

• \( y \) – true function values
• \( \text{yhat} \) – predicted function values

**Returns**

\[
-\frac{1}{n} \sum_{i=1}^{n} [y \ast \log(\hat{y}) + (1 - y) \ast \log(1 - \hat{y})]
\]

\( y \) must be a binary vector, e.g. elements in \{True, False\} \( \text{yhat} \) must be a vector of probabilities, e.g. elements in \([0, 1]\)

Lower is better.

**Note:** This loss function should only be used for probabilistic models.


call optunity.metrics.mse\((y, \text{yhat})\)

Returns the mean squared error between \( y \) and \( \text{yhat} \).

**Parameters**

• \( y \) – true function values
• \( \text{yhat} \) – predicted function values

**Returns**

\[
\frac{1}{n} \sum_{i=1}^{n} [(\hat{y} - y)^2]
\]

Lower is better.
optunity.metrics.npv(y, yhat, positive=True)

Returns the negative predictive value (higher is better).

**Parameters**

- `y` – true function values
- `yhat` – predicted function values
- `positive` – the positive label

**Returns** number of true negative predictions / number of negative predictions

optunity.metrics.pr_auc(ys, yhat, positive=True, presorted=False, return_curve=False)

Computes the area under the precision-recall curve (higher is better).

**Parameters**

- `y` – true function values
- `yhat` – predicted function values
- `positive` – the positive label
- `presorted` (bool) – whether or not `ys` and `yhat` are already sorted
- `return_curve` (bool) – whether or not the curve should be returned

```python
>>> pr_auc([0, 0, 1, 1], [0, 0, 1, 1], 1)
1.0
```

```python
>>> round(pr_auc([0, 0, 1, 1], [0, 1, 1, 2], 1), 2)
0.92
```

**Note:** Precision is undefined at recall = 0. In this case, we set precision equal to the precision that was obtained at the lowest non-zero recall.

optunity.metrics.precision(y, yhat, positive=True)

Returns the precision (higher is better).

**Parameters**

- `y` – true function values
- `yhat` – predicted function values
- `positive` – the positive label

**Returns** number of true positive predictions / number of positive predictions

optunity.metrics.pu_score(y, yhat)

Returns a score used for PU learning as introduced in [LEE2003].

**Parameters**

- `y` – true function values
- `yhat` – predicted function values
Returns
\[
P(\hat{y} = 1 | y = 1)^2
\]
\[
P(\hat{y} = 1)
\]

\(y\) and \(y_{\text{hat}}\) must be boolean vectors.

Higher is better.

\texttt{optunity.metrics.r_squared}(y, y_{\text{hat}})

Returns the R squared statistic, also known as coefficient of determination (higher is better).

\textbf{Parameters}
- \(y\) – true function values
- \(y_{\text{hat}}\) – predicted function values
- \texttt{positive} – the positive label

\textbf{Returns}
\[
R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} = 1 - \frac{\sum_i (y_i - y_{\text{hat}_i})^2}{\sum_i (y_i - \text{mean}(y))^2}
\]

\texttt{optunity.metrics.recall}(y, y_{\text{hat}}, \texttt{positive=True})

Returns the recall (higher is better).

\textbf{Parameters}
- \(y\) – true function values
- \(y_{\text{hat}}\) – predicted function values
- \texttt{positive} – the positive label

\textbf{Returns} number of true positive predictions / number of true positives

\texttt{optunity.metrics.roc_auc}(y, y_{\text{hat}}, \texttt{positive=True, presorted=False, return_curve=False})

Computes the area under the receiver operating characteristic curve (higher is better).

\textbf{Parameters}
- \(y\) – true function values
- \(y_{\text{hat}}\) – predicted function values
- \texttt{positive} – the positive label
- \texttt{presorted} (\texttt{bool}) – whether or not \(y\) and \(y_{\text{hat}}\) are already sorted
- \texttt{return_curve} (\texttt{bool}) – whether or not the curve should be returned

\begin{verbatim}
>>> roc_auc([[0, 0, 1, 1], [0, 0, 1, 1], 1])
1.0

>>> roc_auc([[0,0,1,1], [0,1,1,2], 1])
0.875
\end{verbatim}
Domain constraints

All functionality related to domain constraints on objective function.

Main features in this module:

- `constrained()`
- `wrap_constraints()`

Module author: Marc Claesen

```
exception optunity.constraints.ConstraintViolation(constraint, *args, **kwargs)
    Bases: exceptions.Exception

    Thrown when constraints are not met.
    
    args
    constraint
    kwargs
    message

optunity.constraints.constr_lb_c(field, bounds, *args, **kwargs)
    Models args.field >= bounds.

optunity.constraints.constr_lb_o(field, bounds, *args, **kwargs)
    Models args.field > bounds.

optunity.constraints.constr_range_cc(field, bounds, *args, **kwargs)
    Models args.field in [bounds[0], bounds[1]].

optunity.constraints.constr_range_co(field, bounds, *args, **kwargs)
    Models args.field in [bounds[0), bounds[1)).

optunity.constraints.constr_range_oo(field, bounds, *args, **kwargs)
    Models args.field in (bounds[0), bounds[1]].

optunity.constraints.constr_range_co(field, bounds, *args, **kwargs)
    Models args.field in (bounds[0], bounds[1]).

optunity.constraints.constr_ub_c(field, bounds, *args, **kwargs)
    Models args.field <= bounds.

optunity.constraints.constr_ub_o(field, bounds, *args, **kwargs)
    Models args.field < bounds.

optunity.constraints.constrained(constraints)
    Decorator that puts constraints on the domain of f.
```

```
>>> @constrained([lambda x: x > 0])
... def f(x): return x+1
>>> f(1)
2
>>> f(0)
Traceback (most recent call last):
...
ConstraintViolation
>>> len(f.constraints)
1
```
optunity.constraints.violations_defaulted(default)
Decorator to default function value when a ConstraintViolation occurs.

>>> @violations_defaulted("foobar")
... @constrained([lambda x: x > 0])
... def f(x):
...    return x+1
>>> f(1)
2
>>> f(0)
'foobar'

optunity.constraints.wrap_constraints(f, default=None, ub_o=None, ub_c=None, lb_o=None, lb_c=None, range_oo=None, range_co=None, range_oc=None, range_cc=None, custom=None)
Decorates f with given input domain constraints.

Parameters

- **f(constiable)** – the function that will be constrained
- **default(number)** – function value to default to in case of constraint violations
- **ub_o(dict)** – open upper bound constraints, e.g. \( x < c \)
- **ub_c(dict)** – closed upper bound constraints, e.g. \( x \leq c \)
- **lb_o(dict)** – open lower bound constraints, e.g. \( x > c \)
- **lb_c(dict)** – closed lower bound constraints, e.g. \( x \geq c \)
- **range_oo** (dict with 2-element lists as values ([lb, ub])) – range constraints (open lb and open ub) \( lb < x < ub \)
- **range_co** (dict with 2-element lists as values ([lb, ub])) – range constraints (closed lb and open ub) \( lb \leq x < ub \)
- **range_oc** (dict with 2-element lists as values ([lb, ub])) – range constraints (open lb and closed ub) \( lb < x \leq ub \)
- **range_cc** (dict with 2-element lists as values ([lb, ub])) – range constraints (closed lb and closed ub) \( lb \leq x \leq ub \)
- **custom(list of constraints)** – custom, user-defined constraints

*custom constraints are binary functions that yield False in case of violations.

>>> def f(x):
...    return x
... fc = wrap_constraints(f, default=-1, range_oc=('x': [0, 1]))
>>> fc(x=0.5)
0.5
>>> fc(x=1)
1
>>> fc(x=5)
-1
>>> fc(x=0)
-1

We can define any custom constraint that we want. For instance, assume we have a binary function with arguments \( x \) and \( y \), and we want to make sure that the provided values remain within the unit circle.
Function decorators

A variety of useful function decorators for logging and more.

Main features in this module:

- `logged()`
- `max_evals()`

Module author: Marc Claesen

```python
class optunity.functions.Args(*args, **kwargs):
    Bases: object

    Class to model arguments to a function evaluation. Objects of this class are hashable and can be used as dict keys.

    Arguments and keyword arguments are stored in a frozenset.

    keys()
        Returns a list of argument names.

    parameters
        Returns the internal representation.

    values()
        Returns a list of argument values.

class optunity.functions.CallLog
    Bases: object

    Thread-safe call log.

    The call log is an ordered dictionary containing all previous function calls. Its keys are dictionaries representing the arguments and its values are the function values. As dictionaries can’t be used as keys in dictionaries, a custom internal representation is used.

    Initialize an empty CallLog.

    data
        Access internal data after obtaining lock.

    delete(*args, **kwargs)

    static from_dict(d)
        Converts given dict to a valid call log used by logged functions.
```

```python
>>> def f(x, y):
...     return x + y

>>> circle_constraint = lambda x, y: (x ** 2 + y ** 2) <= 1

>>> fc = wrap_constraints(f, default=1234, custom=[circle_constraint])

>>> fc(0.0, 0.0)
0.0
>>> fc(1.0, 0.0)
1.0
>>> fc(0.5, 0.5)
1.0
>>> fc(1, 0.5)
1234
```
Given dictionary must have the following structure: `{'args': {'argname': []}, 'values': []}

```python
>>> log = CallLog.from_dict({'args': {'x': [1, 2]}, 'values': [2, 3]})
>>> print(log)
{'x': 1} --> 2
{'x': 2} --> 3
```

**get** (*args, **kwargs*)

Returns the result of given evaluation or None if not previously done.

**insert**(value, *args, **kwargs)

**items**()

**keys**()

**lock**

**to_dict**()

Returns given call_log into a dictionary.

The result is a dict with the following structure: `{'args': {'argname': []}, 'values': []}

```python
>>> call_log = CallLog()
>>> call_log.insert(3, x=1, y=2)
>>> d = call_log.to_dict()
>>> d['args']['x'][1]
>>> d['args']['y'][2]
>>> d['values'][3]
```

**update**(other)

**values**()

**exception** optunity.functions.MaximunEvaluationsException (*max_evals*)

Bases: exceptions.Exception

Raised when the maximum number of function evaluations are used.

**args**

**max_evals**

- Returns the maximum number of evaluations that was permitted.

**message**

optunity.functions.call_log2dataframe (log)

Converts a call log into a pandas data frame. This function errors if you don’t have pandas available.

**Parameters** log – call log to be converted, as returned by e.g. optunity.minimize

**Returns** a pandas data frame capturing the same information as the call log

optunity.functions.logged (f)

Decorator that logs unique calls to f. The call log can always be retrieved using f.call_log. Decorating a function that is already being logged has no effect.

The call log is an instance of CallLog.
```python
>>> @logged
def f(x):
    return x+1

>>> a, b, c = f(1), f(1), f(2)

>>> print(f.call_log)
{'pos_0': 1} --> 2
{'pos_0': 2} --> 3

logged as inner decorator:
```}

```python
>>> from .constraints import constrained

>>> @logged
... @constrained([lambda x: x > 1])
... def f2(x):
    return x+1

>>> len(f2.call_log)
0

>>> f2(2)
3

>>> print(f2.call_log)
{'pos_0': 2} --> 3

logged as outer decorator:
```}

```python
>>> from .constraints import constrained

>>> @constrained([lambda x: x > 1])
... @logged
... def f3(x):
    return x+1

>>> len(f3.call_log)
0

>>> f3(2)
3

>>> print(f3.call_log)
{'pos_0': 2} --> 3
```

```python
>>> @logged
def f(x):
    return 1

>>> f(1)
1

>>> print(f.call_log)
{'pos_0': 1} --> 1

>>> @logged
... @wraps(f)
... def f2(x):
    return f(x)

>>> print(f2.call_log)
{'pos_0': 1} --> 1
```

```python
optunity.functions.max_evals(max_evals)

Decorator to enforce a maximum number of function evaluations.

Throws a MaximumEvaluationsException during evaluations after the maximum is reached. Adds a field `f.num_evals` which tracks the number of evaluations that have been performed.

```
optunity.functions.negated(f)

Decorator to negate f such that f'(x) = -f(x).

optunity.functions.static_key_order(keys)

Decorator to fix the key order for use in function evaluations.

A fixed key order allows the function to be evaluated with a list of unnamed arguments rather than kwargs.

```python
>>> @static_key_order(['foo', 'bar'])
... def f(bar, foo):
...     return bar + 2 * foo

f(3, 5)
```

optunity.functions.wraps(obj, attr_names=('__module__', '__name__', '__doc__'))

Safe version of , that can deal with missing attributes such as missing __module__ and __name__.

### Structured search spaces

Functionality to deal with structured search spaces, in which the existence of some hyperparameters is contingent on some discrete choice(s).

For more information on the syntax and modalities of defining structured search spaces, please refer to Structured search spaces.

A search space is defined as a dictionary mapping strings to nodes. Each node is one of the following:

1. A new sub search space, that is a new dictionary with the same structure.
2. 2-element list or tuple, containing (lb, ub) for the associated hyperparameter.
3. None, to indicate a terminal node that has no numeric value associated to it.

Internally, structured search spaces are encoded as a vector, based on the following rules:

- a standard hyperparameter with box constraints is a single dimension
- when a choice must be made, this is encoded as a single dimension (regardless of the amount of options), with values in the range [0, num_choices]
- choice options without additional hyperparameters (i.e., None nodes) do not require further coding

For example, consider an SVM kernel family, choosing between linear and RBF. This is specified as follows:

```python
search = {'kernel': {'linear': None,
                     'rbf': {'gamma': [0, 3]}
                 }
}
```

In vector format, this is encoded as a vector with 2 entries \([kernel \in [0, 2], gamma \in [0, 3]]\).

The main API of this module is the SearchTree class.

*Module author: Marc Claesen*

```python
class optunity.search_spaces.Node(key, value)
    Bases: object

    Models a node within a search space.

    Nodes can be internal or terminal, and may or may not be choices. A choice is a node that models a discrete choice out of \(k > 1\) options.
```
choice
Determines whether this node is a choice.

A choice is a node that models a discrete choice out of \( k > 1 \) options.

determines whether or not this Node is terminal.

A terminal node has a non-dictionary value (numeric, list or None).

class optunity.search_spaces.Options(cases)
Bases: object

cases

class optunity.search_spaces.SearchTree(d)
Bases: object

Tree structure to model a search space.

Fairly elaborate unit test.

```python
>>> space = {'a': {'b0': {'c0': {'d0': {'e0': [0, 10], 'e1': [-2, -1]},
... 'd1': {'e2': [-3, -1]},
... 'd2': None,
... 'c1': [0.0, 1.0],
... 'b1': {'c2': [-2.0, -1.0]},
... 'b2': None,
... }}
... }
```

```python
>>> tree = SearchTree(space)
>>> b = tree.to_box()
>>> print(b['a'] == [0.0, 3.0] and ...
... b['a|b0|c0'] == [0.0, 3.0] and ...
... b['a|b0|c1'] == [0.0, 1.0] and ...
... b['a|b1|c2'] == [-2.0, -1.0] and ...
... b['a|b0|c0|d0|e0'] == [0, 10] and ...
... b['a|b0|c0|d0|e1'] == [-2, -1] and ...
... b['a|b0|c0|d1|e2'] == [-3, -1])
True
```

```python
>>> d = tree.decode({'a': 2.5})
>>> d['a'] == 'b2'
True
```

```python
>>> d = tree.decode({'a': 1.5, 'a|b1|c2': -1.5})
>>> print(d['a'] == 'b1' and ...
... d['c2'] == -1.5)
True
```

```python
>>> d = tree.decode({'a': 0.5, 'a|b0|c0': 1.7, 'a|b0|c0|d1|e2': -1.2})
>>> print(d['a'] == 'b0' and ...
... d['c0'] == 'd1' and ...
... d['e2'] == -1.2)
True
```
```
>>> d = tree.decode({'a': 0.5, 'a|b0|c0': 2.7})
>>> print(d['a'] == 'b0' and ...
...   d['c0'] == 'd2'))
True

>>> d = tree.decode({'a': 0.5, 'a|b0|c0': 0.7, 'a|b0|c0|d0|e0': 2.3, ...
...   'a|b0|c0|d0|e1': -1.5})
>>> print(d['a'] == 'b0' and ...
...   d['c0'] == 'd0' and ...
...   d['e0'] == 2.3 and ...
...   d['e1'] == -1.5)
True
```

### content

**decode** (*vd*)

Decodes a vector representation (as a dictionary) into a result dictionary.

**Parameters**

- **vd** *(dict)* – vector representation

**Returns**

dict containing the decoded representation.

- Choices are given as key:value pairs.
- Active hyperparameters have numeric values.
- Inactive hyperparameters have value None.

**to_box** ()

Creates a set of box constraints to define the given search space.

**Returns**

a set of box constraints (e.g. dictionary, with box constraint values)

Use these box constraints to initialize a solver, which will then implicitly work in the vector representation of the search space.

To evaluate such vector representations, decorate the objective function with `optunity.search_spaces.SearchTree.decode()` of the same object.

**vectorcontent**

**vectordict**

**wrap_decoder** (*f*)

Wraps a function to automatically decode arguments based on given SearchTree.

Use in conjunction with `optunity.search_spaces.SearchTree.to_box()`.

### Standalone process

#### Standalone implementation

This module is for developer use only. It is used to start a communication session.

**Warning:** Importing this module should only be done when launching Optunity as a subprocess to access functionality from non-Python environments. Do not import this module as a Python user.
This module implements several use cases to provide the main Optunity API in other environments. It communicates with the external environment using JSON messages.

We will discuss the use cases as if they are ordinary function calls. Parameter names are keys in the JSON root message.

This standalone subprocess can communicate through stdin/stdout or sockets. To use sockets:

- **standalone as client**: specify the port as first commandline argument and host as second (omitting host will imply localhost).
  
  ```
  python -m optunity.standalone <PORT> <HOST>
  ```

- **standalone as server**: launch with ‘server’ as first command line argument. The port number that is being listened on will be printed on stdout.
  
  ```
  python -m optunity.standalone server
  ```

### Requesting manuals

Emulates `optunity.manual()`.

Request:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>manual</td>
<td>name of the solver whose manual we want or empty string</td>
<td>no</td>
</tr>
</tbody>
</table>

Examples:

```json
{"manual": "}
{"manual": "grid search"}
```

Reply:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>manual</td>
<td>the manual that was requested</td>
<td>list of strings</td>
</tr>
<tr>
<td>solver_names</td>
<td>the name of the solver whose manual was requested or a list of all registered solvers</td>
<td>list of strings</td>
</tr>
</tbody>
</table>

Manuals are returned as a list of strings, each string is a line of the manual.

### Generate cross-validation folds

Use the Python library to generate k-fold cross-validation folds. This requires one message back and forth.

Emulates `optunity.generate_folds()`.

Request:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate_folds</td>
<td>dictionary:</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>- <strong>num_instances</strong> number of instances to consider</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- <strong>num_folds</strong> number of folds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- <strong>num_iter</strong> number of iterations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- <strong>strata</strong> to account for in fold generation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- <strong>clusters</strong> to account for in fold generation</td>
<td></td>
</tr>
</tbody>
</table>

```
Strata and clusters are sent as list (strata/clusters) of lists (instance indices).

**Note:** strata and cluster indices must be 0-based

Example:

```json
{"generate_folds": {"num_instances": 10, "num_folds": 2, "num_iter": 5, "strata": [[[1, 2], [3, 4]], "clusters": [[5, 6], [7, 8]]}}
```

Reply:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>folds</td>
<td>the resulting folds</td>
<td>list (iterations) of lists (folds) of lists (instances)</td>
</tr>
</tbody>
</table>

The inner lists contain the instance indices per fold (0-based indexing).

Example:

```json
{"folds": [[[2, 3, 0, 5, 8], [1, 4, 7, 6, 9]], [[2, 4, 7, 8, 0], [1, 3, 6, 9, 5]], [[2, 4, 9, 7, 8], [1, 3, 0, 5, 6]], [[1, 3, 7, 6, 5], [2, 4, 9, 8, 0]], [[2, 3, 5, 8, 0], [1, 4, 6, 9, 7]]]}
```

### Maximize

Emulates `optunity.maximize()`.

Using the simple maximize functionality involves sending an initial message with the arguments to `optunity.maximize()` as shown below.

Subsequently, the solver will send objective function evaluation requests sequentially. These requests can be for scalar or vector evaluations (details below).

When the solution is found, or the maximum number of evaluations is reached a final message is sent with all details.

Setup request:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximize</td>
<td>dictionary:</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>• num_evals number of permitted function evaluations</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>• box constraints dictionary</td>
<td>no</td>
</tr>
<tr>
<td>call_log</td>
<td>a call log of previous function evaluations</td>
<td>yes</td>
</tr>
<tr>
<td>constraints</td>
<td>domain constraints on the objective function</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>• ub_{oc} upper bound (open/closed)</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>• lb_{oc} lower bound (open/closed)</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>• range_{oc}{oc} interval bounds</td>
<td>yes</td>
</tr>
<tr>
<td>default</td>
<td>number, default function value if constraints are violated</td>
<td>yes</td>
</tr>
</tbody>
</table>
After the initial setup message, Optunity will send objective function evaluation requests. These request may be for a scalar or a vector evaluation, and look like this:

**Scalar evaluation request**: the message is a dictionary containing the hyperparameter names as keys and their associated values as values.

An example request to evaluate \( f(x, y) \) in \((x=1, y=2)\):

\[
\{ "x": 1, "y": 2 \}
\]

**Vector evaluation request**: the message is a list of dictionaries with the same form as the dictionary of a scalar evaluation request.

An example request to evaluate \( f(x, y) \) in \((x=1, y=2)\) and \((x=2, y=3)\):

\[
[\{ "x": 1, "y": 2 \}, \{ "x": 2, "y": 3 \}]
\]

The replies to evaluation requests are simple:

- scalar request: dictionary with key `value` and value the objective function value
- vector request: dictionary with key `values` and value a list of function values

**Note**: Results of vector evaluations must be returned in the same order as the request.

When a solution is found, Optunity will send a final message with all necessary information and then exit. This final message contains the following:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>solution</td>
<td>the optimal hyperparameters</td>
<td>dictionary</td>
</tr>
<tr>
<td>details</td>
<td>various details about the solving process</td>
<td>dictionary</td>
</tr>
<tr>
<td></td>
<td>* <strong>optimum</strong>: ( f(\text{solution}) )</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>* <strong>stats</strong>: number of evaluations and wall clock time</td>
<td>dictionary</td>
</tr>
<tr>
<td></td>
<td>* <strong>call_log</strong>: record of all function evaluations</td>
<td>dictionary</td>
</tr>
<tr>
<td></td>
<td>* <strong>report</strong>: optional solver report</td>
<td>optional</td>
</tr>
<tr>
<td>solver</td>
<td>information about the solver that was used</td>
<td>dictionary</td>
</tr>
</tbody>
</table>

**Minimize**

Identical to maximize (above) except that the initial message has the key `minimize` instead of `maximize`.

Emulates `optunity.minimize()`.

**Make solver**

Attempt to instantiate a solver from given configuration. This serves as a sanity-check.

Emulates `optunity.make_solver()`.
Example:

```json
{"make_solver": {"x": [1, 2], "y": [2, 3], "solver_name": "grid search"}}
```

Optunity replies with one of two things:

- `{"success": true}`: the solver was correctly instantiated
- `{"error_msg": "...": instantiating the solver failed

**Optimize**

Using the optimize functionality involves sending an initial message with the arguments to `optunity.optimize()` as shown below.

Subsequently, the solver will send objective function evaluation requests sequentially. These requests can be for scalar or vector evaluations (details below).

When the solution is found, or the maximum number of evaluations is reached a final message is sent with all details.

Emulates `optunity.optimize()`.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimize</td>
<td>dictionary</td>
<td>no</td>
</tr>
<tr>
<td>solver</td>
<td>dictionary</td>
<td>no</td>
</tr>
<tr>
<td>call_log</td>
<td>a call log of previous function evaluations</td>
<td>yes</td>
</tr>
<tr>
<td>constraints</td>
<td>domain constraints on the objective function</td>
<td>yes</td>
</tr>
<tr>
<td>default</td>
<td>number, default function value if constraints are violated</td>
<td>yes</td>
</tr>
</tbody>
</table>
After the initial setup message, Optunity will send objective function evaluation requests. These request may be for a scalar or a vector evaluation, and look like this:

**Scalar evaluation request:** the message is a dictionary containing the hyperparameter names as keys and their associated values as values.

An example request to evaluate \( f(x, y) \) in \((x=1, y=2)\):

\[
\{"x": 1, "y": 2\}
\]

**Vector evaluation request:** the message is a list of dictionaries with the same form as the dictionary of a scalar evaluation request.

An example request to evaluate \( f(x, y) \) in \((x=1, y=2)\) and \((x=2, y=3)\):

\[
[\{"x": 1, "y": 2\}, \{"x": 2, "y": 3\}]
\]

The replies to evaluation requests are simple:

- scalar request: dictionary with key *value* and value the objective function value
- vector request: dictionary with key *values* and value a list of function values

**Note:** Results of vector evaluations must be returned in the same order as the request.

When a solution is found, Optunity will send a final message with all necessary information and then exit. This final message contains the following:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>solution</td>
<td>the optimal hyperparameters</td>
<td>dictionary</td>
</tr>
<tr>
<td>details</td>
<td>various details about the solving process</td>
<td>dictionary</td>
</tr>
<tr>
<td></td>
<td>• optimum: ( f(\text{solution}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• stats: number of evaluations and wall clock time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• call_log: record of all function evaluations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• report: optional solver report</td>
<td></td>
</tr>
</tbody>
</table>

**Module author:** Marc Claesen

```python
optunity.standalone.fold_request(cv_opts)
```

Computes k-fold cross-validation folds. Emulates `optunity.cross_validated()`.

```python
optunity.standalone.main()
```

```python
optunity.standalone.make_solver(solver_config)
```

```python
optunity.standalone.manual_request(solver_name)
```

Emulates `optunity.manual()`.

```python
optunity.standalone.max_or_min(solve_fun, kwargs, constraints, default, call_log)
```

Emulates `optunity.maximize()` and `optunity.minimize()`.

```python
optunity.standalone.optimize(solver_config, constraints, default, call_log, maximize, max_evals)
```

Emulates `optunity.optimize()`.
optunity.standalone.prepare_fun(mgr, constraints, default, call_log)

Creates the objective function and wraps it with domain constraints and an existing call log, if applicable.

Communication

class optunity.communication.EvalManager(max_vectorized=100, replacements={})
    Bases: object

    Constructs an EvalManager object.

    Parameters
        • max_vectorized (int) – the maximum size of a vector evaluation larger vectorizations will be chunked
        • replacements (dict) – a mapping of original:replacement keyword names

    add_to_queue(**kwargs)

    cv

    flush_queue()

    get(number)

    max_vectorized

    pipe_eval(**kwargs)

    pmap(f, *args)

    Performs vector evaluations through pipes.

    Parameters
        • f (callable) – the objective function (pipelined_function_eval)
        • args (iterables) – function arguments

    The vector evaluation is sent in chunks of size self.max_vectorized.

    processed_semaphore

    queue

    queue_lock

    replacements

    Keys that must be replaced: keys are current values, values are what must be sent through the pipe.

    semaphore

    vectorized

optunity.communication.accept_server_connection(server_socket)

optunity.communication.json_decode(data)

    Decodes given JSON string and returns its data.

    >>> orig = {1: 2, 'a': [1,2]}
    >>> data = json_decode(json_encode(orig))
    >>> data[str(1)]
    2
    >>> data['a']
    [1, 2]
Optunity Documentation, Release 1.1.0

```python
optunity.communication.json_encode(data)
    Encodes given data in a JSON string.

optunity.communication.make_piped_function(mgr)

optunity.communication.open_server_socket()

optunity.communication.open_socket(port, host='localhost')
    Opens a socket to host:port and reconfigures internal channels.

optunity.communication.receive()
    Reads data from channel.

optunity.communication.send(data)
    Writes data to channel and flushes.
```

Optunity

Provides

1. Routines to efficiently optimize hyperparameters
2. Function decorators to implicitly log evaluations, constrain the domain and more.
3. Facilities for k-fold cross-validation to estimate generalization performance.

Available modules

- **solvers** contains all officially supported solvers
- **functions** a variety of useful function decorators for hyperparameter tuning
- **cross_validation** k-fold cross-validation

Available subpackages

- **tests** regression test suite
- **solvers** solver implementations and auxiliary functions

Utilities

- `__version__` Optunity version string
- `__revision__` Optunity revision string
- `__author__` Main authors of the package

```
optunity.manual(solver_name=None)
    Prints the manual of requested solver.

    Parameters solver_name – (optional) name of the solver to request a manual from. If none is specified, a general manual is printed.

    Raises KeyError if solver_name is not registered.
```

```
optunity.maximize(f, num_evals=50, solver_name=None, pmap=<built-in function map>, **kwargs)
    Basic function maximization routine. Maximizes f within the given box constraints.
```

4.6. Optunity API 115
Parameters

- \( f \) – the function to be maximized
- \( \text{num_evals} \) – number of permitted function evaluations
- \( \text{solver}_\text{name}(\text{string}) \) – name of the solver to use (optional)
- \( \text{pmap}(\text{callable}) \) – the map function to use
- \( \text{kwargs} \) – box constraints, a dict of the following form {'parameter_name':
  [lower_bound, upper_bound], ...}

Returns retrieved maximum, extra information and solver info

This function will implicitly choose an appropriate solver and its initialization based on \( \text{num_evals} \) and the box constraints.

```
optunity.minimize(f, num_evals=50, solver_name=None, pmap=<built-in function map>, **kwargs)
```

Basic function minimization routine. Minimizes \( f \) within the given box constraints.

Parameters

- \( f \) – the function to be minimized
- \( \text{num_evals} \) – number of permitted function evaluations
- \( \text{solver}_\text{name}(\text{string}) \) – name of the solver to use (optional)
- \( \text{pmap}(\text{callable}) \) – the map function to use
- \( \text{kwargs} \) – box constraints, a dict of the following form {'parameter_name':
  [lower_bound, upper_bound], ...}

Returns retrieved minimum, extra information and solver info

This function will implicitly choose an appropriate solver and its initialization based on \( \text{num_evals} \) and the box constraints.

```
optunity.optimize(solver, func, maximize=True, max_evals=0, pmap=<built-in function map>, decoder=None)
```

Optimizes func with given solver.

Parameters

- \( \text{solver} \) – the solver to be used, for instance a result from \( \text{optunity.make_solver()} \)
- \( \text{func}(\text{callable}) \) – the objective function
- \( \text{maximize}(\text{bool}) \) – maximize or minimize?
- \( \text{max_evals}(\text{int}) \) – maximum number of permitted function evaluations
- \( \text{pmap}(\text{function}) \) – the map() function to use, to vectorize use \( \text{optunity.pmap()} \)

Returns the solution and a namedtuple with further details.

Result details includes the following:

- \( \text{optimum} \) optimal function value \( f(\text{solution}) \)
- \( \text{stats} \) statistics about the solving process
- \( \text{call_log} \) the call log
- \( \text{report} \) solver report, can be None

Statistics gathered while solving a problem:

- \( \text{num_evals} \) number of function evaluations
**time** wall clock time needed to solve

**optunity.wrap_call_log**(f, call_dict)
Wraps an existing call log (as dictionary) around f.

This allows you to communicate known function values to solvers. (currently available solvers do not use this info)

**optunity.wrap_constraints**(f, default=None, ub_o=None, ub_c=None, lb_o=None, lb_c=None, range_oo=None, range_co=None, range_oc=None, range_cc=None, custom=None)
Decorates f with given input domain constraints.

*Parameters*

- **f**(callable) – the function that will be constrained
- **default**(number) – function value to default to in case of constraint violations
- **ub_o**(dict) – open upper bound constraints, e.g. \( x < c \)
- **ub_c**(dict) – closed upper bound constraints, e.g. \( x \leq c \)
- **lb_o**(dict) – open lower bound constraints, e.g. \( x > c \)
- **lb_c**(dict) – closed lower bound constraints, e.g. \( x \geq c \)
- **range_oo**(dict with 2-element lists as values ([lb, ub])) – range constraints (open lb and open ub) \( lb < x < ub \)
- **range_co**(dict with 2-element lists as values ([lb, ub])) – range constraints (closed lb and open ub) \( lb \leq x < ub \)
- **range_oc**(dict with 2-element lists as values ([lb, ub])) – range constraints (open lb and closed ub) \( lb < x \leq ub \)
- **range_cc**(dict with 2-element lists as values ([lb, ub])) – range constraints (closed lb and closed ub) \( lb \leq x \leq ub \)
- **custom**(list of constraints) – custom, user-defined constraints

*custom constraints are binary functions that yield False in case of violations.*

```python
>>> def f(x):
...     return x
>>> fc = wrap_constraints(f, default=-1, range_oc={'x': [0, 1]})
>>> fc(x=0.5)
0.5
>>> fc(x=1)
1
>>> fc(x=5)
-1
>>> fc(x=0)
-1
```

We can define any custom constraint that we want. For instance, assume we have a binary function with arguments \( x \) and \( y \), and we want to make sure that the provided values remain within the unit circle.

```python
>>> def f(x, y):
...     return x + y
>>> circle_constraint = lambda x, y: (x ** 2 + y ** 2) <= 1
>>> fc = wrap_constraints(f, default=1234, custom=[circle_constraint])
>>> fc(0.0, 0.0)
0.0
```
optunity.make_solver(solver_name, *args, **kwargs)

Creates a Solver from given parameters.

Parameters

- **solver_name** *(string)* – the solver to instantiate
- **args** – positional arguments to solver constructor.
- **kwargs** – keyword arguments to solver constructor.

Use *optunity.manual()* to get a list of registered solvers. For constructor arguments per solver, please refer to *Solver overview*.

Raises *KeyError* if

- *solver_name* is not registered
- *args* and *kwargs* are invalid to instantiate the solver.

optunity.suggest_solver(num_evals=50, solver_name=None, **kwargs)

optunity.cross_validated(x, num_folds=10, y=None, strata=None, folds=None, num_iter=1, regenerate_folds=False, clusters=None, aggregator=<function mean>)

Function decorator to perform cross-validation as configured.

Parameters

- **x** – data to be used for cross-validation
- **num_folds** – number of cross-validation folds (default 10)
- **y** – (optional) labels to be used for cross-validation. If specified, len(labels) must equal len(x)
- **strata** – (optional) strata to account for when generating folds. Strata signify instances that must be spread across folds. Not every instance must be in a stratum. Specify strata as a list of lists of instance indices.
- **folds** – (optional) prespecified cross-validation folds to be used (list of lists (iterations) of lists (folds)).
- **num_iter** – (optional) number of iterations to use (default 1)
- **regenerate_folds** – (optional) whether or not to regenerate folds on every evaluation (default false)
- **clusters** – (optional) clusters to account for when generating folds. Clusters signify instances that must be assigned to the same fold. Not every instance must be in a cluster. Specify clusters as a list of lists of instance indices.
- **aggregator** – function to aggregate scores of different folds (default: mean)

Returns a *cross_validated_callable* with the proper configuration.

This resulting decorator must be used on a function with the following signature (+ potential other arguments):

Parameters
• `x_train(iterable)` – training data
• `y_train(iterable)` – training labels (optional)
• `x_test(iterable)` – testing data
• `y_test(iterable)` – testing labels (optional)

`y_train` and `y_test` must be available of the `y` argument to this function is not `None`.

These 4 keyword arguments will be bound upon decoration. Further arguments will remain free (e.g. hyperparameter names).

```python
>>> data = list(range(5))
>>> @cross_validated(x=data, num_folds=5, folds=[[i] for i in range(5)],
                  aggregator=identity)
... def f(x_train, x_test, a):
...     return x_test[0] + a
>>> f(a=1)
[1, 2, 3, 4, 5]
>>> f(1)
[1, 2, 3, 4, 5]
>>> f(a=2)
[2, 3, 4, 5, 6]
```

The number of folds must be less than or equal to the size of the data.

```python
>>> data = list(range(5))
>>> @cross_validated(x=data, num_folds=6)
... def f(x_train, x_test, a):
...     return x_test[0] + a
AssertionError
```

The number of labels (if specified) must match the number of data instances.

```python
>>> data = list(range(5))
>>> labels = list(range(3))
>>> @cross_validated(x=data, y=labels, num_folds=2)
... def f(x_train, x_test, a):
...     return x_test[0] + a
AssertionError
```

`optunity.generate_folds(num_rows, num_folds=10, strata=None, clusters=None)`

Generates folds for a given number of rows.

**Parameters**

• `num_rows` – number of data instances
• `num_folds` – number of folds to use (default 10)
• `strata` – (optional) list of lists to indicate different sampling strata. Not all rows must be in a stratum. The number of rows per stratum must be larger than or equal to `num_folds`.
• `clusters` – (optional) list of lists indicating clustered instances. Clustered instances must be placed in a single fold to avoid information leaks.

**Returns** a list of folds, each fold is a list of instance indices

```python
>>> folds = generate_folds(num_rows=6, num_folds=2, clusters=[[1, 2]], strata=[[3, 4]])
>>> len(folds)
```
```
>>> i1 = [idx for idx, fold in enumerate(folds) if 1 in fold]
>>> i2 = [idx for idx, fold in enumerate(folds) if 2 in fold]
>>> i1 == i2
True
```  
```
>>> i3 = [idx for idx, fold in enumerate(folds) if 3 in fold]
>>> i4 = [idx for idx, fold in enumerate(folds) if 4 in fold]
>>> i3 == i4
False
```  
**Warning:** Instances in strata are not necessarily spread out over all folds. Some folds may already be full due to clusters. This effect should be negligible.

```
optunity.pmap(f, *args, **kwargs)
```

Parallel map using multiprocessing.

**Parameters**

- `f` – the callable
- `args` – arguments to f, as iterables

**Returns** a list containing the results

**Warning:** This function will not work in IPython: https://github.com/claesenm/optunity/issues/8.

**Warning:** Python’s multiprocessing library is incompatible with Jython.

```
optunity.available_solvers()
```

Returns a list of all available solvers.

These can be used in `optunity.make_solver()`.

```
optunity.call_log2dataframe(log)
```

Converts a call log into a pandas data frame. This function errors if you don’t have pandas available.

**Parameters** `log` – call log to be converted, as returned by e.g. `optunity.minimize`

**Returns** a pandas data frame capturing the same information as the call log

```
optunity.maximize_structured(f, search_space, num_evals=50, pmap=<built-in function map>)
```

Basic function maximization routine. Maximizes `f` within the given box constraints.

**Parameters**

- `f` – the function to be maximized
- `search_space` – the search space (see Structured search spaces for details)
- `num_evals` – number of permitted function evaluations
- `pmap` (callable) – the map function to use

**Returns** retrieved maximum, extra information and solver info

This function will implicitly choose an appropriate solver and its initialization based on `num_evals` and the box constraints.
optunity.minimize_structured(f, search_space, num_evals=50, pmap=<built-in function map>)

Basic function minimization routine. Minimizes f within the given box constraints.

**Parameters**

- **f** – the function to be maximized
- **search_space** – the search space (see *Structured search spaces* for details)
- **num_evals** – number of permitted function evaluations
- **pmap** (*callable*) – the map function to use

**Returns** retrieved maximum, extra information and solver info

This function will implicitly choose an appropriate solver and its initialization based on num_evals and the box constraints.


Python Module Index

optimized

optunity.communication, 114
optunity.constraints, 101
optunity.cross_validation, 93
optunity.functions, 103
optunity.metrics, 96
optunity.search_spaces, 106
optunity.solvers, 77
optunity.solvers.CMAES, 78
optunity.solvers.GridSearch, 79
optunity.solvers.NelderMead, 81
optunity.solvers.ParticleSwarm, 82
optunity.solvers.RandomSearch, 85
optunity.solvers.Sobol, 86
optunity.solvers.solver_registry, 90
optunity.solvers.TPE, 89
optunity.solvers.util, 91
optunity.standalone, 108

O

optunity, 115
Index

A
absolute_error() (in module optunity.metrics), 96
accept_server_connection() (in module optunity.communication), 114
accuracy() (in module optunity.metrics), 96
add_to_queue() (optunity.communication.EvalManager method), 114
append() (optunity.solvers.util.ThreadSafeQueue method), 92
Args (class in optunity.functions), 103
args (optunity.constraints.ConstraintViolation attribute), 101
args (optunity.functions.MaximumEvaluationsException attribute), 104
assign_grid_points() (optunity.solvers.GridSearch.GridSearch static method), 79
auc() (in module optunity.metrics), 96
available_solvers() (in module optunity), 120

B
bitwise_xor() (optunity.solvers.Sobol.Sobol static method), 86
bounds (optunity.solvers.ParticleSwarm.ParticleSwarm attribute), 83
bounds (optunity.solvers.RandomSearch.RandomSearch attribute), 85
bounds (optunity.solvers.Sobol.Sobol attribute), 87
bound constraints, 23
brier() (in module optunity.metrics), 96
call_log2dataframe() (in module optunity), 120
call_log2dataframe() (in module optunity.functions), 104
CallLog (class in optunity.functions), 103
cases (optunity.search_spaces.Options attribute), 107
choice (optunity.search_spaces.Node attribute), 106
cloned() (optunity.solvers.ParticleSwarm.ParticleSwarm.ParticleSwarm attribute), 83
CMA_ES (class in optunity.solvers.CMAES), 78
compute_curve() (in module optunity.metrics), 96
constr_lb_c() (in module optunity.constraints), 101
constr_lb_o() (in module optunity.constraints), 101
constr_range_cc() (in module optunity.constraints), 101
constr_range_co() (in module optunity.constraints), 101
constr_range_oc() (in module optunity.constraints), 101
constr_range_oo() (in module optunity.constraints), 101
constr_ub_c() (in module optunity.constraints), 101
constr_ub_o() (in module optunity.constraints), 101
constrained() (in module optunity.constraints), 101
constraint (optunity.constraints.ConstraintViolation attribute), 101
ConstraintViolation, 101
call_log2dataframe() (in module optunity.functions), 104
decode() (optunity.search_spaces.SearchTree method), 108
data (optunity.solvers.util.ThreadSafeQueue attribute), 92
data (optunity.functions.CallLog attribute), 103
data (optunity.solvers.util.ThreadSafeQueue method), 92
data (optunity.solvers.ParticleSwarm.ParticleSwarm method), 83
contingency_table() (in module optunity.metrics), 97
contingency_tables() (in module optunity.metrics), 97
copy() (optunity.solvers.util.ThreadSafeQueue method), 92
cross_validated() (in module optunity), 118
cross_validated() (in module optunity.cross_validation), 118
cv (optunity.communication.EvalManager attribute), 114
data (optunity.functions.CallLog method), 103
desc_brief (optunity.solvers.GridSearch.GridSearch attribute), 79
desc_brief (optunity.solvers.NelderMead.NelderMead attribute), 81
desc_brief (optunity.solvers.ParticleSwarm.ParticleSwarm attribute), 83

127
| A | desc_brief (optunity.solvers.RandomSearch.RandomSearch attribute), 85 |
| B | desc_brief (optunity.solvers.Sobol.Sobol attribute), 87 |
| C | desc_full (optunity.solvers.GridSearch.GridSearch attribute), 79 |
| D | desc_full (optunity.solvers.NelderMead.NelderMead attribute), 81 |
| E | desc_full (optunity.solvers.ParticleSwarm.ParticleSwarm attribute), 83 |
| F | desc_full (optunity.solvers.RandomSearch.RandomSearch attribute), 85 |
| G | desc_full (optunity.solvers.Sobol.Sobol attribute), 87 |
| H | error_rate() (in module optunity.metrics), 97 |
| I | EvalManager (class in optunity.communication), 114 |
| J | fbeta() (in module optunity.metrics), 98 |
| K | flush_queue() (optunity.communication.EvalManager method), 114 |
| L | fold_request() (in module optunity.standalone), 113 |
| M | from_dict() (optunity.functions.CallLog static method), 103 |
| N | get() (optunity.communication.EvalManager method), 114 |
| O | get() (optunity.functions.CallLog method), 104 |
| P | get() (optunity.solvers.ParticleSwarm.ParticleSwarm method), 83 |
| Q | generate() (optunity.solvers.ParticleSwarm.ParticleSwarm method), 83 |
| R | generate() (optunity.solvers.solver_registry), 90 |
| S | generate_folds() (in module optunity), 119 |
| T | generate_folds() (in module optunity.cross_validation), 94 |
| U | get() (optunity.solvers.solver_registry), 90 |
| V | get() (optunity.solvers.solver_registry), 90 |
| W | get() (optunity.solvers.solver_registry), 90 |
| X | get() (optunity.solvers.solver_registry), 90 |
| Y | get() (optunity.solvers.solver_registry), 90 |
| Z | get() (optunity.solvers.solver_registry), 90 |
maximize() (optunity.solvers.ParticleSwarm.ParticleSwarm method), 83
maximize() (optunity.solvers.RandomSearch.RandomSearch method), 85
maximize() (optunity.solvers.Sobol.Sobol method), 88
maximize() (optunity.solvers.TPE.TPE method), 89
maximize() (optunity.solvers.util.Solver method), 91
maximize_structured() (in module optunity), 120
MaximumEvaluationsException, 104
mean() (in module optunity.cross_validation), 95
mean_and_list() (in module optunity.cross_validation), 95
message (optunity.constraints.ConstraintViolation attribute), 101
message (optunity.functions.MaximumEvaluationsException attribute), 104
minimize() (built-in function), 75
minimize() (in module optunity), 116
minimize() (optunity.solvers.CMAES.CMA_ES method), 78
minimize() (optunity.solvers.GridSearch.GridSearch method), 80
minimize() (optunity.solvers.NelderMead.NelderMead method), 81
minimize() (optunity.solvers.ParticleSwarm.ParticleSwarm method), 83
minimize() (optunity.solvers.RandomSearch.RandomSearch method), 85
minimize() (optunity.solvers.Sobol.Sobol method), 88
minimize() (optunity.solvers.TPE.TPE method), 89
minimize() (optunity.solvers.util.Solver method), 91
minimize_structured() (in module optunity), 120
mse() (in module optunity.metrics), 98

name (optunity.solvers.GridSearch.GridSearch attribute), 80
name (optunity.solvers.NelderMead.NelderMead attribute), 82
name (optunity.solvers.ParticleSwarm.ParticleSwarm attribute), 84
name (optunity.solvers.RandomSearch.RandomSearch attribute), 85
name (optunity.solvers.Sobol.Sobol attribute), 88
negated() (in module optunity.functions), 106
NelderMead (class in optunity.solvers.NelderMead), 81
Node (class in optunity.search_spaces), 106
npv() (in module optunity.metrics), 99
nth_root() (in module optunity.search_spaces), 106
num_evals (optunity.solvers.GridSearch.GridSearch attribute), 81
num_evals (optunity.solvers.RandomSearch.RandomSearch attribute), 85
num_evals (optunity.solvers.Sobol.Sobol attribute), 88
num_evals (optunity.solvers.TPE.TPE attribute), 90

O
objective function, 23
open_server_socket() (in module optunity.communication), 115
open_socket() (in module optunity.communication), 115
optimize() (in module optunity), 116
optimize() (in module optunity.standalone), 113
optimize() (optunity.solvers.CMAES.CMA_ES method), 78
optimize() (optunity.solvers.GridSearch.GridSearch method), 80
optimize() (optunity.solvers.NelderMead.NelderMead method), 82
optimize() (optunity.solvers.ParticleSwarm.ParticleSwarm method), 84
optimize() (optunity.solvers.RandomSearch.RandomSearch method), 86
optimize() (optunity.solvers.Sobol.Sobol method), 88
optimize() (optunity.solvers.TPE.TPE method), 90
optimize() (optunity.solvers.util.Solver method), 91
Options (class in optunity.search_spaces), 107
optunity (module), 115
optunity.communication (module), 114
optunity.constraints (module), 101
optunity.cross_validation (module), 114
optunity.functions (module), 103
optunity.metrics (module), 96
optunity.search_spaces (module), 106
optunity.solvers (module), 77
optunity.solvers.CMAES (module), 78
optunity.solvers.GridSearch (module), 79
optunity.solvers.NelderMead (module), 81
optunity.solvers.ParticleSwarm (module), 82
optunity.solvers.RandomSearch (module), 85
optunity.solvers.Sobol (module), 86
optunity.solvers.solver_registry (module), 90
optunity.solvers.TPE (module), 89
optunity.solvers.util (module), 91
optunity.standalone (module), 108

P
parameter_tuples (optunity.solvers.GridSearch.GridSearch attribute), 80
parameters (optunity.functions.Args attribute), 103
to_box() (optunity.search_spaces.SearchTree method), 108

to_dict() (optunity.functions.CallLog method), 104
TPE (class in optunity.solvers.TPE), 89
train-predict-score (TPS) chain, 24

U
uniform_in_bounds() (in module optunity.solvers.util), 93
update() (optunity.functions.CallLog method), 104
updateParticle() (optunity.solvers.ParticleSwarm.ParticleSwarm method), 85
upper (optunity.solvers.RandomSearch.RandomSearch attribute), 86

V
value (optunity.search_spaces.Node attribute), 107
values() (optunity.functions.Args method), 103
values() (optunity.functions.CallLog method), 104
vectorcontent (optunity.search_spaces.SearchTree attribute), 108
vectordict (optunity.search_spaces.SearchTree attribute), 108
vectorized (optunity.communication.EvalManager attribute), 114
violations_defaulted() (in module optunity.constraints), 101

W
wrap_call_log() (in module optunity), 117
wrap_constraints() (in module optunity), 117
wrap_constraints() (in module optunity.constraints), 102
wrap_decoder() (optunity.search_spaces.SearchTree method), 108
wraps() (in module optunity.functions), 106