Chocolate

Release 0.6.0

May 16, 2020

Contents

Chocolate is a completely asynchronous optimisation framework relying solely on a database to share information between workers. Chocolate uses no master process for distributing tasks. Every task is completely independent and only gets its information from the database. Chocolate is thus ideal in controlled computing environments where it is hard to maintain a master process for the duration of the optimisation.

Chocolate has been designed and optimized for hyperparameter optimization where each function evaluation takes very long to complete and is difficult to parallelize.

Chocolate is licensed under the 3-Clause BSD License

CHAPTER 1

Basics

Let start with the very basics. Suppose you want to optimize the parameters of the himmelblau function $f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$ with Chocolate.

```
def himmelblau(x, y):
    return (x**2 + y - 11)**2 + (x + y**2 - 7)**2
```

You'd first have to define a *search space* for parameters x and y.

```
import chocolate as choco

space = {"x" : choco.uniform(-6, 6),
         "y" : choco.uniform(-6, 6)}
```

Next, you'd establish where the results should be saved. We have two database adaptors one *SQLiteConnection* (which we prefer) and one *MongoDBConnection* (which we also like, of course!).

```
conn = choco.SQLiteConnection("sqlite:///my_db.db")
```

Note: While the SQLite adaptor should be used when a common file system is available for all compute nodes, the MongoDB adaptor is more suited when compute nodes cannot share such file system (e.g. Amazon EC2 spot instances).

When this overwhelming task is done, you'd choose from our *sampling* or *search* algorithms the one that you prefer. We will use quasi random sampling because its ze best.

```
sampler = choco.QuasiRandom(conn, space, random_state=42, skip=0)
```

Now comes the funny part, using Chocolate, a process usually does one and only one evaluation. So you'd just have to ask the sampler: "Good morning! What's the next point I should evaluate?", do the evaluation an tell the sampler about it.

```
token, params = sampler.next()
loss = himmelblau(**params)
sampler.update(token, loss)
```

Ho yeah, the token is just a unique id we use to trace the parameters in the database. You sure can have a look at it.

```
>>> print(token)
{"_chocolate_id" : 0}
```

CHAPTER 2

Realistic Example

Lets see how one can optimize the hyper parameters of say a gradient boosting tree classifier using scikit-learn and Chocolate. First we'll do the necessary imports.

```
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split

import chocolate as choco
```

And we'll define our train function to return the negative of the F1 score as loss function for our Chocolate minimizer. Nothing fancy here.

```
def score_gbt(X, y, params):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    gbt = GradientBoostingClassifier(**params)
    gbt.fit(X_train, y_train)
    y_pred = gbt.predict(X_test)

    return -f1_score(y_test, y_pred)
```

Then we will load our dataset (or make it).

```
X, y = make_classification(n_samples=80000, random_state=1)
```

And just as in the *Basics* tutorial, we'll decide where the data is stored and the *search space* for the algorithm. We will optimize over a mix of continuous and discrete variables.

```
conn = choco.SQLiteConnection(url="sqlite:///db.db")
s = {"learning_rate" : choco.uniform(0.001, 0.1),
     "n_estimators"  : choco.quantized_uniform(25, 525, 25),
     "max_depth"     : choco.quantized_uniform(2, 10, 2),
     "subsample"     : choco.quantized_uniform(0.7, 1.05, 0.05)}
```

Finally, we will define our sampling algorithm,

```
sampler = choco.QuasiRandom(conn, s, seed=110, skip=3)
```

request a set of parameters to test,

```
token, params = sampler.next()
```

get the loss for that set,

```
loss = score_gbt(X, y, params)
```

and signify it to the database.

```
sampler.update(token, loss)
```

Running this ~20 line script a bunch of times in completely separate processes will explore the search space to find a good parameter set for your problem. As simple as that.

CHAPTER 3

Optimizing Over Multiple Models

Searching for a good configuration for multiple models at the same time is possible using conditional search spaces. A conditional search space is defined by a list of dictionaries each containing one or more non-*chocolate.Distribution* parameter.

3.1 Independent Parameter Search

Say we want to optimize the hyperparameters of SVMs with different kernels or even multiple types of SVMs. We would define the *search space* as a list of dictionaries, one for each model.

```
from sklearn.svm import SVC, LinearSVC
import chocolate as choco

space = [{"algo" : SVC, "kernel" : "rbf",
          "C" : choco.log(low=-2, high=10, base=10),
          "gamma" : choco.log(low=-9, high=3, base=10)},
         {"algo" : SVC, "kernel" : "poly",
          "C" : choco.log(low=-2, high=10, base=10),
          "gamma" : choco.log(low=-9, high=3, base=10),
          "degree" : choco.quantized_uniform(low=1, high=5, step=1),
          "coef0" : choco.uniform(low=-1, high=1)},
         {"algo" : LinearSVC,
          "C" : choco.log(low=-2, high=10, base=10),
          "penalty" : choco.choice(["l1", "l2"])}]
```

Lets now define the optimization function. Since we were able to directly define the classifier type as the parameter `"algo"` we can use that directly. Note that the F1 score has to be maximized, however, Chocolate always minimizes the loss. Thus, we shall return the negative of the F1 score.

```
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split

def score_svm(X, y, algo, **params):
```

(continues on next page)

(continued from previous page)

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    clf = algo(**params)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    return -f1_score(y_test, y_pred)
```

Just as in the simpler examples, we will load our dataset, make our connection and explore the configurations using one of the algorithm.

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=80000, random_state=1)

conn = choco.SQLiteConnection(url="sqlite:///db.db")
sampler = choco.QuasiRandom(conn, space, random_state=42, skip=0)

token, params = sampler.next()
loss = score_svm(X, y, **params)
sampler.update(token, loss)
```

And just like that multiple models are explored simultaneously.

3.2 Sharing Parameters Between Models

In the last example, all SVMs share parameter `C`. Some optimizers might take advantage of this information to optimize this parameters across all SVM types thus accelerating convergence to an optimal configuration. Furthermore, the `SVC` algorithm and parameter `gamma` are shared for `"rbf"` and `"poly"` kernels. We can rewrite the last search space using nested dictionaries to represent the multiple condition levels.

```
space = {"algo" : {SVC : {"gamma" : choco.log(low=-9, high=3, base=10)},
                          "kernel" : {"rbf" : None,
                                      "poly" : {"degree" : choco.quantized_
↪uniform(low=1, high=5, step=1),
                                                "coef0" : choco.uniform(low=-1,␣
↪high=1)}},
                   LinearSVC : {"penalty" : choco.choice(["l1", "l2"])}},
         "C" : choco.log(low=-2, high=10, base=10)}
```

We can still add complexity to the search space by combining multiple dictionaries at the top level, if for example a configuration does not name an `"algo"` parameter.

```
space = [{"algo" : {SVC : {"gamma" : choco.log(low=-9, high=3, base=10)},
                           "kernel" : {"rbf" : None,
                                       "poly" : {"degree" : choco.quantized_
↪uniform(low=1, high=5,  step=1),
                                                 "coef0" : choco.uniform(low=-1,␣
↪high=1)}},
                    LinearSVC : {"penalty" : choco.choice(["l1", "l2"])}},
          "C" : choco.log(low=-2, high=10, base=10)},

         {"type" : "an_other_optimizer", "param" : choco.uniform(low=-1, high=1)}]
```

The remaining of the exploration is identical to the previous section.

CHAPTER 4

Optimizing Over Multiple Objectives

Chocolate offers multi-objective optimization. This means you can optimize the precision and recall without averaging them in a f1 score or even the precision and inference time of a model! Lets go straight to how to do that. First, as always, import we import the necessary modules.

```
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import precision_score, recall_score
from sklearn.model_selection import train_test_split

import chocolate as choco
```

Note that we imported both the `sklearn.metrics.precision_score()` and `sklearn.metrics.recall_score()` metrics. The train function is almost identical to the *realistic* tutorial, except for the two losses.

```
def score_gbt(X, y, params):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    gbt = GradientBoostingClassifier(**params)
    gbt.fit(X_train, y_train)
    y_pred = gbt.predict(X_test)

    return -precision_score(y_test, y_pred), -recall_score(y_test, y_pred)
```

Is that it? Yes! This is the only modofication required to optimize over multiple objectives (in addition to using a multi-objective capable search algorithm).

Then we will load our dataset (or make it).

```
X, y = make_classification(n_samples=80000, random_state=1)
```

And just as in the *Basics* tutorial, we'll decide where the data is stored and the *search space* for the algorithm. We will optimize over a mix of continuous and discrete variables.

```
conn = choco.SQLiteConnection(url="sqlite:///db.db")
s = {"learning_rate" : choco.uniform(0.001, 0.1),
     "n_estimators"  : choco.quantized_uniform(25, 525, 25),
     "max_depth"     : choco.quantized_uniform(2, 10, 2),
     "subsample"     : choco.quantized_uniform(0.7, 1.05, 0.05)}
```

Finally, we will define our search algorithm, request a set of parameters to test, get the loss for that set and signify it to the database.

```
sampler = choco.MOCMAES(conn, s, mu=5)
token, params = sampler.next()
loss = score_gbt(X, y, params)
sampler.update(token, loss)
```

Once this script has run a couple of times, the results can be retrieved. Obviously, we cannot find THE ULTIMATE configuration in our database since multi-objective optimization is all about compromise. In fact, the result of the optimization is a Pareto front containing all non dominated compromises between the objectives. You can easily retrieve these compromises using the *results_as_dataframe()* method of your connection. To find the Pareto optimal solutions use *chocolate.mo.argsortNondominated()* function as follow.

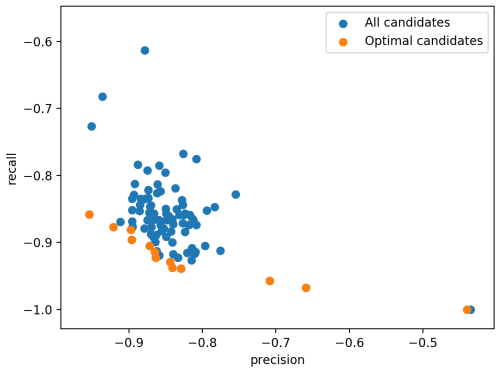
```
conn = choco.SQLiteConnection(url="sqlite:///db.db")
results = conn.results_as_dataframe()
losses = results.as_matrix(("_loss_0", "_loss_1"))
first_front = argsortNondominated(losses, len(losses), first_front_only=True)
```

This front can be plotted using matplotlib.

```
plt.scatter(losses[:, 0], losses[:, 1], label="All candidates")
plt.scatter(losses[first_front, 0], losses[first_front, 1], label="Optimal candidates
↪")
plt.xlabel("precision")
plt.ylabel("recall")
plt.legend()

plt.show()
```

And, we get this nice graph:



CHAPTER 5

Optimizing a Tensor Flow Model

Optimizing the hyperparameters of a TensorFlow model is no harder than any other optimization. The only difficulty would be the multiple levels where hyperparameters are set. For example, the learning rate is set in the training function while the number of neurons in a given layer is set while constructing the model.

Let say we want to optimize the hyperparameters of a convolutional neural network over bunch of parameters including the activation function per layer, the number of neurons in each layer and even the number of layers. First, we need a function that builds the model.

```
import tensorflow as tf
from tensorflow import layers

def cnn_model(inputs, targets, dropout_keep_prob, params):
    num_output = int(targets.get_shape()[1])
    net = inputs

    # Get the number of convolution layers from the parameter set
    for i in range(0, params["num_conv_layers"]):
        with tf.variable_scope("conv_{}".format(i)):
            # Create layer using input parameters
            net = layers.conv2d(net,
                                filters=params["conv_{}_num_outputs".format(i)],
                                kernel_size=params["conv_{}_kernel_size".format(i)],
                                strides=1,
                                padding="SAME",
                                activation=params["conv_{}_activation_fn".format(i)])

            net = layers.conv2d(net,
                                filters=params["conv_{}_num_outputs".format(i)],
                                kernel_size=params["conv_{}_kernel_size".format(i)],
                                strides=1,
                                padding="SAME",
                                activation=params["conv_{}_activation_fn".format(i)])

        with tf.variable_scope("mp_{}".format(i)):
```

(continues on next page)

(continued from previous page)

```
            net = layers.max_pooling2d(net,
                                       pool_size=params["mp_{}_kernel_size".
↪format(i)],
                                       strides=1,
                                       padding="VALID")

    # Dropout keep probability is set a train time.
    net = tf.nn.dropout(net, keep_prob=dropout_keep_prob)
    net = tf.contrib.layers.flatten(net)

    # Get the number of fully connectec layers from the parameter set
    for i in range(params["num_fc_layers"]):
        with tf.variable_scope("fc_{}".format(i)):
            # Create layer using input parameters
            net = tf.contrib.layers.fully_connected(net, params["fc_{}_num_outputs".
↪format(i)],
                                    activation_fn=params["fc_{}_activation_fn".
↪format(i)])

            net = tf.nn.dropout(net, keep_prob=dropout_keep_prob)

    with tf.variable_scope("output_layer"):
        net = tf.contrib.layers.fully_connected(net, num_output, activation_fn=tf.
↪identity)

    return net
```

Then, we need a function to train the model that also has parameters to optimize such as the learning rate, the decay rate and the dropout keep probability. (No, it is not the ideal train function, it is just a demo.)

```
def score_cnn(X, y, params):
    sess = tf.InteractiveSession()

    train_steps = 20
    num_classes = y.shape[1]

    X_ = tf.placeholder(tf.float32, shape=(None,) + X.shape[1:])
    y_ = tf.placeholder(tf.float32, shape=(None, num_classes))
    keep_prob_ = tf.placeholder(tf.float32)
    lr_ = tf.placeholder(tf.float32)

    logits = cnn_model(X_, y_, keep_prob_, params)

    loss_func = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,␣
↪labels=y_))
    optimizer_func = tf.train.AdamOptimizer(lr_).minimize(loss_func)

    predict = tf.argmax(logits, 1)
    correct_prediction = tf.equal(predict, tf.argmax(y_, 1))

    init = tf.global_variables_initializer()
    sess.run(init)

    lr_init = params["initial_learning_rate"]
    lr_decay = params["decay_learning_rate"]
    decay_steps = params["decay_steps"]
```

(continues on next page)

(continued from previous page)

```
    X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2)

    with sess.as_default():
        for step in range(train_steps):
            lr = lr_init * lr_decay ** (step / decay_steps)
            for i in range(0, X_train.shape[0], 128):
                feed_dict = {lr_: lr, X_: X_train[i:i+128], y_: y_train[i:i+128],
                             keep_prob_: params["dropout_keep_prob"]}

            _, train_loss = sess.run([optimizer_func, loss_func], feed_dict=feed_dict)
        valid_loss = 0
        for i in range(0, X_valid.shape[0], 128):
            feed_dict = {X_: X_valid[i:i+128], y_: y_valid[i:i+128], keep_prob_: 1.0}
            valid_loss += sess.run([loss_func], feed_dict=feed_dict)[0]
        valid_loss = valid_loss / (X_valid.shape[0]//128)

    return {"loss" : valid_loss}
```

The flexibility of the last pieces of code comes at a price; the number of parameters to set in the search space is quite large. The next table summarizes all the parameters that needs to be set with their type

Model	Type	**Training**	Type
num_conv_layers	integer	initial_learning_rate	float
conv_{i}_num_outputs	integer	decay_learning_rate	float
conv_{i}_kernel_size	integer	decay_steps	integer
conv_{i}_activation_fn	choice	dropout_keep_prob	float
mp_{i}_kernel_size	integer		
num_fc_layers	integer		
fc_{i}_num_outputs	integer		
fc_{i}_activation_fn	choice		

Since there are so many hyperparameters, lets just define a function that will creates the search space. The four training hyperparameters will sit a the top level of our space and the two defining the number of layers will constitute our conditions. All others will be set for these conditions.

```
import chocolate as choco

max_num_conv_layers = 8
max_num_fc_layers = 3

def create_space():
    space = {"initial_learning_rate" : choco.log(low=-5, high=-2, base=10),
             "decay_learning_rate" : choco.uniform(low=0.7, high=1.0),
             "decay_steps" : choco.quantized_log(low=2, high=4, step=1, base=10),
             "dropout_keep_prob" : choco.uniform(low=0.5, high=0.95)}

    num_conv_layer_cond = dict()
    for i in range(1, max_num_conv_layers):
        condition = dict()
        for j in range(i):
            condition["conv_{}_num_outputs".format(j)] = choco.quantized_log(low=3,␣
↪high=10, step=1, base=2)
            condition["conv_{}_kernel_size".format(j)] = choco.quantized_
↪uniform(low=1, high=7, step=1)
```

(continues on next page)

(continued from previous page)

```
            condition["conv_{}_activation_fn".format(j)] = choco.choice([tf.nn.relu,␣
↪tf.nn.elu, tf.nn.tanh])
            condition["mp_{}_kernel_size".format(j)] = choco.quantized_uniform(low=2,␣
↪high=5, step=1)

        num_conv_layer_cond[i] = condition

    space["num_conv_layers"] = num_conv_layer_cond

    num_fc_layer_cond = dict()
    for i in range(1, max_num_fc_layers):
        condition = dict()
        for j in range(i):
            condition["fc_{}_num_outputs".format(j)] = choco.quantized_log(low=3,␣
↪high=10, step=1, base=2)
            condition["fc_{}_activation_fn".format(j)] = choco.choice([tf.nn.relu, tf.
↪nn.elu, tf.nn.tanh])

        num_fc_layer_cond[i] = condition
    space["num_fc_layers"] = num_fc_layer_cond

    return space
```

Guess how large is the largest conditional branch of this search space. It has 36 parameters. 36 parameters is quite a lot to optimize by hand. The entire tree has 124 parameters! That is why we built Chocolate.

Ho yeah, I forgot about the last bit of code. The one that does the trick.

```
if __name__ == "__main__":
    X, y = some_dataset()

    space = create_space()
    conn = choco.SQLiteConnection(url="sqlite:///db.db")
    sampler = choco.Bayes(conn, space, random_state=42, skip=0)

    token, params = sampler.next()
    loss = score_cnn(X, y, params)
    sampler.update(token, loss)
```

Nha, there was absolutly nothing new here compared to the last tutorials.

CHAPTER 6

Retrieving Results

There is nothing easier than retrieving your results with Chocolate. Connections define a method *results_as_dataframe()* that takes care of loading the data from your database, transforming it back to your search space ranges and populating a pandas.DataFrame. This way you can use the powerful pandas and seaborn libraries to analyse you results and not miss anything. Here is how to get a nice pairwise plot of each parameter with the loss.

```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

from chocolate import SQLiteConnection

conn = SQLiteConnection("sqlite:///chocolate.db")
results = conn.results_as_dataframe()
results = pd.melt(results, id_vars=["loss"], value_name='value', var_name="variable")

sns.lmplot(x="value", y="loss", data=results, col="variable", col_wrap=3,␣
↪sharex=False)

plt.show()
```

And for those like me who are not patient enough to let the optimization finish, the method *results_as_dataframe()* is multiprocess-safe (thanks to our databases)!

CHAPTER 7

What Algorithm to Choose?

The choice of the sampling/search strategy depends strongly on the problem tackled. Ultimately, their are 4 aspects of the problem to look at:

- the time required to evaluate a model,
- the number of variables,
- the type of variable (continuous or discrete),
- the conditionality of the search space.

Chocolate proposes 5 algorithms with their own advantages and disadvantages:

- `Grid` sampling applies when all variables are discrete and the number of possibilities is low. A grid search will perform the exhaustive combinatorial search over all possibilities making the search extremely long even for medium sized problems.
- `Random` sampling is an alternative to grid search when the number of discrete parameters to optimize and the time required for each evaluation is high. When all parameters are discrete, random search will perform sampling without replacement making it an algorithm of choice when combinatorial exploration is not possible. With continuous parameters, it is preferable to use quasi random sampling.
- `QuasiRandom` sampling ensures a much more uniform exploration of the search space than traditional pseudo random. Thus, quasi random sampling is preferable when not all variables are discrete, the number of dimensions is high and the time required to evaluate a solution is high.
- `Bayes` search models the search space using gaussian process regression, which allows to have an estimate of the loss function and the uncertainty on that estimate at every point of the search space. Modeling the search space suffers from the curse of dimensionality, which makes this method more suitable when the number of dimensions is low. Moreover, since it models both the expected loss and uncertainty, this search algorithm converges in few steps on superior configurations, making it a good choice when the time to complete the evaluation of a parameter configuration is high.
- `CMAES` search is one of the most powerful black-box optimization algorithm. However, it requires a significant number of model evaluation (in the order of 10 to 50 times the number of dimensions) to converge to an optimal solution. This search method is more suitable when the time required for a model evaluation is relatively low.

- *MOCMAES* search is a multi-objective algorithm optimizing multiple tradeoffs simultaneously. To do that, MOCMAES employs μ CMAES algorithms. Thus requiring even more evaluation to converge to the optimal solution (in the order of μ times 10 to 50, times the number of dimensions). This search method is more suitable when the time required for a model evaluation is relatively low.

In addition to the 5 previous algorithms Chocolate proposes a wrapper that transforms the conditional search space problem in a multi-armed bandit problem.

- *ThompsonSampling* is a wrapper around any of the sampling/search algorithms that will allocate more resources to the exploration of the most promising subspaces. This method will help any of the algorithm in finding a superior solution in conditional search spaces.

Here is a table that resumes when to use each algorithm.

Algorithm	Time	Dimensions	Continuity	Conditions	Multi-objective
Grid	Low	Low	Discrete	Yes	No
Random	High	High	Discrete	Yes	No
QuasiRandom	High	High	Mixed	Yes	No
Bayes	High	Medium	Mixed	Yes	No
CMAES	Low	Low	Mixed	No	No
MOCMAES	Low	Low	Mixed	No	Yes
ThompsonSampling	–	–	–	Yes	–

CHAPTER 8

Cross-validating Optimization

More often than not, the optimized process results have some variability. To make the optimization process more robust each parameter set has to be evaluated more than once. Chocolate provides seemless cross-validation in the search algorithms. The cross-validation object, if provided, intercepts calls to the database and ensures every experiment is repeated a given number of times. Cross-validations, just like every other experiments, is done in parallel and asynchroneously. To use cross-validation simply create a cross-validation object and assign it to the search algorithm.

```
import numpy as np
import chocolate as choco

def evaluate(p1, p2):
    return p1 + p2 + np.random.randn()

if __name__ == "__main__":
    space = {"p1": choco.uniform(0, 10), "p2": choco.uniform(0, 5)}
    connection = choco.SQLiteConnection(url="sqlite:///cv.db")
    cv = choco.Repeat(repetitions=3, reduce=np.mean, rep_col="_repetition_id")
    s = choco.Grid(space, connection, crossvalidation=cv)

    token, params = s.next()
    loss = evaluate(**params)
    print(token, params, loss)
    s.update(token, loss)
```

The preceding script, if run a couple of times, will output the following tokens and parameters (with probably different parameters).

```
{'_repetition_id': 0, '_chocolate_id': 0} {'p1': 8.1935000833291518, 'p2': 4.
↪2668676560356529} 13.886112047266854
{'_repetition_id': 1, '_chocolate_id': 0} {'p1': 8.1935000833291518, 'p2': 4.
↪2668676560356529} 11.394347119228563
{'_repetition_id': 2, '_chocolate_id': 0} {'p1': 8.1935000833291518, 'p2': 4.
↪2668676560356529} 10.790294230308477
{'_repetition_id': 0, '_chocolate_id': 1} {'p1': 7.4031022047092732, 'p2': 0.
↪14633280691567885} 6.349087103521951
```

(continues on next page)

(continued from previous page)

```
{'_repetition_id': 1, '_chocolate_id': 1} {'p1': 7.4031022047092732, 'p2': 0.
↪14633280691567885} 6.269733948749414
{'_repetition_id': 2, '_chocolate_id': 1} {'p1': 7.4031022047092732, 'p2': 0.
↪14633280691567885} 6.895059981273982
{'_repetition_id': 0, '_chocolate_id': 2} {'p1': 2.4955760398088778, 'p2': 4.
↪4722460515061} 6.82570693646037
```

Note: The cross-validation is not responsible of shuffling your dataset. You must include this step in your script.

The cross-validation object wraps the connection to reduce the loss of experiments with same `"_chocolate_id"`. Thus, algorithms never see the repetitions, they only receive a single parameter set with the reduced loss. For the last example, the algorithms, when interrogating the database, will see the following parameter sets and losses.

```
{'p1': 8.1935000833291518, 'p2': 4.2668676560356529} 12.023584465601298
{'p1': 7.4031022047092732, 'p2': 0.14633280691567885} 6.5046270111817819
{'p1': 2.4955760398088778, 'p2': 4.4722460515061} 6.82570693646037
```

CHAPTER 9

Installation

Chocolate is installed using pip, unfortunately we don't have any PyPI package yet. Here is the line you have to type

```
pip install git+https://github.com/AIworx-Labs/chocolate@master
```

9.1 Dependencies

Chocolate has various dependencies. While the optimizers depends on NumPy, SciPy and Scikit-Learn, the SQLite database connection depends on dataset and filelock and the MongoDB database connection depends on PyMongo. Some utilities depend on pandas. All but PyMongo will be installed with Chocolate.

CHAPTER 10

Library Reference

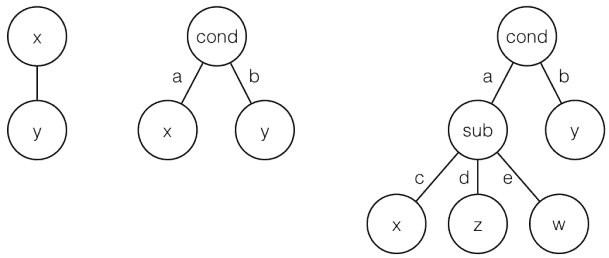
10.1 Search Space Representation

This module provides common building blocks to define a search space.

Search spaces are defined using dictionaries, where the keys are the parameter names and the values their distribution. For example, defining a two parameter search space is done as follow

```
space = {"x": uniform(-5, 5),
         "y": quantized_uniform(-2, 3, 0.5)}
```

A conditional search space can be seen as a tree, where each condition defines a subtree. For example, in the next figure, three search spaces are presented.



The left tree is the simple two parameter search space defined earlier. The middle tree defines a conditional search space with a single root condition. Two subspaces exist in this search space, one when the condition is *a* the other when the condition is *b*. Defining such a search space is done using a list of dictionaries as follow

```
space = [{"cond": "a", "x": uniform(-5, 5)},
         {"cond": "b", "y": quantized_uniform(-2, 3, 0.5)}]
```

The right most tree has two conditions one at its root and another one when the root condition is *a*. It has a total of four subspaces. Defining such a search space is done using a hierarchy of dictionaries as follow

```
space = [{"cond": "a", "sub": {"c": {"x": uniform(-5, 5)},
                               "d": {"z": log(-5, 5, 10)},
                               "e": {"w": quantized_log(-2, 7, 1, 10)}}},
         {"cond": "b", "y": quantized_uniform(-2, 3, 0.5)}
```

Note that lists can only be used at the root of conditional search spaces, sub-conditions must use the dictionary form. Moreover, it is not necessary to use the same parameter name for root conditions. For example, the following is a valid search space

```
space = [{"cond": "a", "x": uniform(-5, 5)},
         {"spam": "b", "y": quantized_uniform(-2, 3, 0.5)}]
```

The only restriction is that each search space must have a unique combination of conditional parameters and values, where conditional parameters have non-distribution values. Finally, one and only one subspace can be defined without condition as follow

```
space = [{"x": uniform(-5, 5)},
         {"cond": "b", "y": quantized_uniform(-2, 3, 0.5)}]
```

If two or more subspaces share the same conditional key (set of parameters and values) an `AssertionError` will be raised uppon building the search space specifying the erroneous key.

class chocolate.**Space**(*spaces*)

Representation of the search space.

Encapsulate a multidimentional search space defined on various distributions. Remind that order in standard python dictionary is undefined, thus the keys of the input dictionaries are `sorted()` and put in `OrderedDict`s for reproductibility.

Parameters **spaces** – A dictionary or list of dictionaries of parameter names to their distribution. When a list of multiple dictionaries is provided, the structuring elements of these items must define a set of unique choices. Structuring elements are defined using non-distribution values. See examples below.

Raises `AssertionError` – When two keys at the same level are equal.

An instance of a space is a callable object wich will return a valid parameter set provided a vector of numbers in the half-open uniform distribution $[0, 1)$.

The number of distinc dimensions can be queried with the `len()` function. When a list of dictionaries is provided, this choice constitute the first dimension and each subsequent conditional choice is also a dimension.

Examples

Here is how a simple search space can be defined and the parameters can be retrieved

```
In [2]: s = Space({"learning_rate": uniform(0.0005, 0.1),
                   "n_estimators" : quantized_uniform(1, 11, 1)})

In [3]: s([0.1, 0.7])
Out[3]: {'learning_rate': 0.01045, 'n_estimators': 8}
```

A one level conditional multidimentional search space is defined using a list of dictionaries. Here the choices are a SMV with linear kernel and a K-nearest neighbor as defined by the string values. Note the use of class names in the space definition.

```
In [2]: from sklearn.svm import SVC

In [3]: from sklearn.neighbors import KNeighborsClassifier

In [4]: s = Space([{"algo": SVC, "kernel": "linear",
                        "C": log(low=-3, high=5, base=10)},
                   {"algo": KNeighborsClassifier,
                        "n_neighbors": quantized_uniform(low=1, high=20, step=1)}
 ↪])
```

The number of dimensions of such search space can be retrieved with the `len()` function.

```
In [5]: len(s)
Out[5]: 3
```

As in the simple search space a valid parameter set can be retrieved by querying the space object with a vector of length equal to the full search space.

```
In [6]: s([0.1, 0.2, 0.3])
Out[6]:
{'C': 0.039810717055349734,
 'algo': sklearn.svm.classes.SVC,
 'kernel': 'linear'}

In [7]: s([0.6, 0.2, 0.3])
Out[7]:
{'algo': sklearn.neighbors.classification.KNeighborsClassifier,
 'n_neighbors': 6}
```

Internal conditions can be modeled using nested dictionaries. For example, the SVM from last example can have different kernels. The next search space will share the `C` parameter amongst all SVMs, but will branch on the kernel type with their individual parameters.

```
In [2]: s = Space([{"algo": "svm",
                    "C": log(low=-3, high=5, base=10),
                    "kernel": {"linear": None,
                               "rbf": {"gamma": log(low=-2, high=3, base=10)}}},
                   {"algo": "knn",
                        "n_neighbors": quantized_uniform(low=1, high=20, step=1)}
 ↪])

In [3]: len(s)
Out[3]: 5

In [4]: x = [0.1, 0.2, 0.7, 0.4, 0.5]

In [5]: s(x)
Out[5]: {'C': 0.039810717055349734, 'algo': 'svm', 'gamma': 1.0, 'kernel': 'rbf'}
```

names(*unique=True*)

Returns unique sequential names meant to be used as database column names.

Parameters **unique** – Whether or not to return unique mangled names. Subspaces will still be mangled.

Examples

If the length of the space is 2 as follow

```
In [2]: s = Space({"learning_rate": uniform(0.0005, 0.1),
                   "n_estimators" : quantized_uniform(1, 11, 1)})

In [3]: s.names()
Out[3]: ['learning_rate', 'n_estimators']
```

While in conditional spaces, if the length of the space is 5 (one for the choice od subspace and four independent parameters)

```
In [4]: s = Space([{"algo": "svm", "kernel": "linear",
                        "C": log(low=-3, high=5, base=10)},
                   {"algo": "svm", "kernel": "rbf",
                        "C": log(low=-3, high=5, base=10),
                        "gamma": log(low=-2, high=3, base=10)},
                   {"algo": "knn",
                        "n_neighbors": quantized_uniform(low=1, high=20,␣
↪step=1)}])

In [5]: s.names()
Out[5]:
['_subspace',
 'algo_svm_kernel_linear_C',
 'algo_svm_kernel_rbf_C',
 'algo_svm_kernel_rbf_gamma',
 'algo_knn_n_neighbors']
```

When using methods or classes as parameter values for conditional choices the output might be a little bit more verbose, however the names are still there.

```
In [6]: s = Space([{"algo": SVC,
                    "C": log(low=-3, high=5, base=10),
                    "kernel": {"linear": None,
                               "rbf": {"gamma": log(low=-2, high=3, base=10)}
↪}},

                   {"algo": KNeighborsClassifier,
                        "n_neighbors": quantized_uniform(low=1, high=20,␣
↪step=1)}])

In [7]: s.names()
Out[7]:
['_subspace',
 'algo_<class sklearn_svm_classes_SVC>_C',
 'algo_<class sklearn_svm_classes_SVC>_kernel__subspace',
 'algo_<class sklearn_svm_classes_SVC>_kernel_kernel_rbf_gamma',
 'algo_<class sklearn_neighbors_classification_KNeighborsClassifier>_n_
↪neighbors']
```

isactive(x)

Checks within conditional subspaces if, with the given vector, a parameter is active or not.

Parameters **x** – A vector of numbers in the half-open uniform distribution $[0, 1)$.

Returns A list of booleans telling is the parameter is active or not.

Example

When using conditional spaces it is often necessary to assess quickly what dimensions are active according to a given vector. For example, with the following conditional space

```
In [2]: s = Space([{"algo": "svm",
                    "C": log(low=-3, high=5, base=10),
                    "kernel": {"linear": None,
                               "rbf": {"gamma": log(low=-2, high=3, base=10)}
↪}},
                   {"algo": "knn",
                        "n_neighbors": quantized_uniform(low=1, high=20,␣
↪step=1)}])
In [3]: s.names()
Out[3]:
['_subspace',
 'algo_svm_C',
 'algo_svm_kernel__subspace',
 'algo_svm_kernel_kernel_rbf_gamma',
 'algo_knn_n_neighbors']

In [4]: x = [0.1, 0.2, 0.7, 0.4, 0.5]

In [5]: s(x)
Out[5]: {'C': 0.039810717055349734, 'algo': 'svm', 'gamma': 1.0, 'kernel':
↪'rbf'}

In [6]: s.isactive(x)
Out[6]: [True, True, True, True, False]

In [6]: x = [0.6, 0.2, 0.7, 0.4, 0.5]

In [8]: s(x)
Out[8]: {'algo': 'knn', 'n_neighbors': 10}

In [9]: s.isactive(x)
Out[9]: [True, False, False, False, True]
```

steps()
Returns the steps size between each element of the space dimensions. If a variable is continuous the returned stepsize is `None`.

isdiscrete()
Returns whether or not this search space has only discrete dimensions.

subspaces()
Returns every valid combinaition of conditions of the tree- structured search space. Each combinaition is a list of length equal to the total dimensionality of this search space. Active dimensions are either a fixed value for conditions or a *`Distribution`* for optimizable parameters. Inactive dimensions are `None`.

Example

The following search space has 3 possible subspaces

```
In [2]: s = Space([{"algo": "svm",
                    "C": log(low=-3, high=5, base=10),
                    "kernel": {"linear": None,
```

(continues on next page)

(continued from previous page)

```
                                    "rbf": {"gamma": log(low=-2, high=3, base=10)}
↪}},
                    {"algo": "knn",
                         "n_neighbors": quantized_uniform(low=1, high=20,␣
↪step=1)}])

In [3]: s.names()
Out[3]:
['_subspace',
 'algo_svm_C',
 'algo_svm_kernel__subspace',
 'algo_svm_kernel_kernel_rbf_gamma',
 'algo_knn_n_neighbors']

In [4]: s.subspaces()
Out[4]:
[[0.0, log(low=-3, high=5, base=10), 0.0, None, None],
 [0.0, log(low=-3, high=5, base=10), 0.5, log(low=-2, high=3, base=10), None],
 [0.5, None, None, None, quantized_uniform(low=1, high=20, step=1)]]
```

class chocolate.**Distribution**
Base class for every Chocolate distributions.

class chocolate.**ContinuousDistribution**
Base class for every Chocolate continuous distributions.

class chocolate.**QuantizedDistribution**
Base class for every Chocolate quantized distributions.

class chocolate.**uniform**(*low*, *high*)
Uniform continuous distribution.

Representation of the uniform continuous distribution in the half-open interval $[\text{low}, \text{high})$.

Parameters

- **low** – Lower bound of the distribution. All values will be greater or equal than low.
- **high** – Upper bound of the distribution. All values will be lower than high.

__call__(*x*)
Transforms x a uniform number taken from the half-open continuous interval $[0, 1)$ to the represented distribution.

Returns The corresponding number in the half-open interval $[\text{low}, \text{high})$.

class chocolate.**quantized_uniform**(*low*, *high*, *step*)
Uniform discrete distribution.

Representation of the uniform continuous distribution in the half-open interval $[\text{low}, \text{high})$ with regular spacing between samples. If $\left\lceil \frac{\text{high}-\text{low}}{step} \right\rceil \neq \frac{\text{high}-\text{low}}{step}$, the last interval will have a different probability than the others. It is preferable to use $\text{high} = N \times \text{step} + \text{low}$ where N is a whole number.

Parameters

- **low** – Lower bound of the distribution. All values will be greater or equal than low.
- **high** – Upper bound of the distribution. All values will be lower than high.
- **step** – The spacing between each discrete sample.

__call__(*x*)
Transforms *x*, a uniform number taken from the half-open continuous interval $[0, 1)$, to the represented distribution.

Returns The corresponding number in the discrete half-open interval $[\text{low}, \text{high})$ alligned on step size. If the output number is whole, this method returns an `int` otherwise a `float`.

__iter__()
Iterate over all possible values of this discrete distribution in the $[0, 1)$ space. This is the same as

```
numpy.arange(0, 1, step / (high - low))
```

__getitem__(*i*)
Retrieve the `i` th value of this distribution in the $[0, 1)$ space.

__len__()
Get the number of possible values for this distribution.

class chocolate.**log**(*low*, *high*, *base*)
Logarithmic uniform continuous distribution.

Representation of the logarithmic uniform continuous distribution in the half-open interval $[\text{base}^{\text{low}}, \text{base}^{\text{high}})$.

Parameters

- **low** – Lower bound of the distribution. All values will be greater or equal than base^{low}.
- **high** – Upper bound of the distribution. All values will be lower than $\text{base}^{\text{high}}$.
- **base** – Base of the logarithmic function.

__call__(*x*)
Transforms *x*, a uniform number taken from the half-open continuous interval $[0, 1)$, to the represented distribution.

Returns The corresponding number in the discrete half-open interval $[\text{base}^{\text{low}}, \text{base}^{\text{high}})$ alligned on step size. If the output number is whole, this method returns an `int` otherwise a `float`.

class chocolate.**quantized_log**(*low*, *high*, *step*, *base*)
Logarithmic uniform discrete distribution.

Representation of the logarithmic uniform discrete distribution in the half-open interval $[\text{base}^{\text{low}}, \text{base}^{\text{high}})$. with regular spacing between sampled exponents.

Parameters

- **low** – Lower bound of the distribution. All values will be greater or equal than base^{low}.
- **high** – Upper bound of the distribution. All values will be lower than $\text{base}^{\text{high}}$.
- **step** – The spacing between each discrete sample exponent.
- **base** – Base of the logarithmic function.

__call__(*x*)
Transforms *x*, a uniform number taken from the half-open continuous interval $[0, 1)$, to the represented distribution.

Returns The corresponding number in the discrete half-open interval $[\text{base}^{\text{low}}, \text{base}^{\text{high}})$ alligned on step size. If the output number is whole, this method returns an `int` otherwise a `float`.

__iter__()
Iterate over all possible values of this discrete distribution in the $[0, 1)$ space. This is the same as

```
numpy.arange(0, 1, step / (high - low))
```

__getitem__(*i*)
Retrieve the `i` th value of this distribution in the $[0, 1)$ space.

__len__()
Get the number of possible values for this distribution.

class chocolate.**choice**(*values*)
Uniform choice distribution between non-numeric samples.

Parameters **values** – A list of choices to choose uniformly from.

__call__(*x*)
Transforms *x*, a uniform number taken from the half-open continuous interval $[0, 1)$, to the represented distribution.

Returns The corresponding choice from the entered values.

__iter__()
Iterate over all possible values of this discrete distribution in the $[0, 1)$ space. This is the same as

```
numpy.arange(0, 1, step / (high - low))
```

__getitem__(*i*)
Retrieve the `i` th value of this distribution in the $[0, 1)$ space.

__len__()
Get the number of possible values for this distribution.

10.2 Database Connections

class chocolate.**SQLiteConnection**(*url*, *result_table='results'*, *complementary_table='complementary'*, *space_table='space'*)
Connection to a SQLite database.

Before calling any method you must explicitly *lock()* the database since SQLite does not handle well concurrency.

We use dataset under the hood allowing us to manage a SQLite database just like a list of dictionaries. Thus no need to predefine any schema nor maintain it explicitly. You can treat this database just as a list of dictionaries.

Parameters

- **url** (*str*) – Full url to the database, as described in the SQLAlchemy documentation. The url is parsed to find the database path. A lock file will be created in the same directory than the database. In memory databases (`url = "sqlite:///"` or `url = "sqlite://:memory:"`) are not allowed.
- **result_table** (*str*) – Table used to store the experiences and their results.
- **complementary_table** (*str*) – Table used to store complementary information necessary to the optimizer.
- **space_table** (*str*) – Table used to save the optimization *Space*.

Raises `RuntimeError` – When an invalid name is given, see error message for precision.

results_as_dataframe()
Compile all the results and transform them using the space specified in the database. It is safe to use this method while other experiments are still writing to the database.

Returns A `pandas.DataFrame` containing all results with its `"_chocolate_id"` as `"id"`, their parameters and its loss. Pending results have a loss of `None`.

lock(*timeout=-1*, *poll_interval=0.05*)
Context manager that locks the entire database.

Parameters

- **timeout** – If the lock could not be acquired in *timeout* seconds raises a timeout error. If 0 or less, wait forever.
- **poll_interval** – Number of seconds between lock acquisition tryouts.

Raises `TimeoutError` – Raised if the lock could not be acquired.

Example:

```
conn = SQLiteConnection("sqlite:///temp.db")
with conn.lock(timeout=5):
    # The database is locked
    all_ = conn.all_results()
    conn.insert({"new_data" : len(all_)})

# The database is unlocked
```

all_results()
Get a list of all entries of the result table. The order is undefined.

insert_result(*document*)
Insert a new *document* in the result table. The columns must not be defined nor all present. Any new column will be added to the database and any missing column will get value None.

update_result(*filter*, *values*)
Update or add *values* of given rows in the result table.

Parameters

- **filter** – An identifier of the rows to update.
- **values** – A mapping of values to update or add.

count_results()
Get the total number of entries in the result table.

all_complementary()
Get all entries of the complementary information table as a list. The order is undefined.

insert_complementary(*document*)
Insert a new document (row) in the complementary information table.

find_complementary(*filter*)
Find a document (row) from the complementary information table.

get_space()
Returns the space used for previous experiments.

Raises `AssertionError` – If there are more than one space in the database.

insert_space(*space*)
Insert a space in the database.

Raises AssertionError – If a space is already present in the database.

clear()
Clear all data from the database.

class chocolate.**MongoDBConnection**(*url*, *database='chocolate'*, *result_col='results'*, *complementary_col='complementary'*, *space_col='space'*)
Connection to a MongoDB database.

Parameters

- **url** (*str*) – Full url to the database including credentials but omitting the database and the collection. When using authenticated databases, the url must contain the database and match the database argument.
- **database** (*str*) – The database name in the MongoDB engine.
- **result_col** (*str*) – Collection used to store the experiences and their results.
- **complementary_col** (*str*) – Collection used to store complementary information necessary to the optimizer.
- **space_table** (*str*) – Collection used to save the optimization *Space*.

results_as_dataframe()
Compile all the results and transform them using the space specified in the database. It is safe to use this method while other experiments are still writing to the database.

Returns A pandas.DataFrame containing all results with its "_chocolate_id" as "id", their parameters and its loss. Pending results have a loss of None.

lock(*timeout=-1*, *poll_interval=0.05*)
Context manager that locks the entire database.

```
conn = MongoDBConnection("mongodb://localhost:27017/")
with conn.lock(timeout=5):
    # The database is lock
    all_ = conn.all_results()
    conn.insert({"new_data" : len(all_)})

# The database is unlocked
```

Parameters

- **timeout** – If the lock could not be acquired in *timeout* seconds raises a timeout error. If 0 or less, wait forever.
- **poll_interval** – Number of seconds between lock acquisition tryouts.

Raises TimeoutError – Raised if the lock could not be acquired.

all_results()
Get all entries of the result table as a list. The order is undefined.

insert_result(*document*)
Insert a new *document* in the result table.

update_result(*token*, *values*)
Update or add *values* to given documents in the result table.

Parameters

- **token** – An identifier of the documents to update.

- **value** – A mapping of values to update or add.

count_results()
Get the total number of entries in the result table.

all_complementary()
Get all entries of the complementary information table as a list. The order is undefined.

insert_complementary(*document*)
Insert a new document in the complementary information table.

find_complementary(*filter*)
Find a document from the complementary information table.

get_space()
Returns the space used for previous experiments.

Raises AssertionError – If there are more than one space in the database.

insert_space(*space*)
Insert a space in the database.

Raises AssertionError – If a space is already present in the database.

clear()
Clear all data from the database.

class chocolate.**DataFrameConnection**(*from_file=None*)
Connection to a pandas DataFrame.

This connection is meant when it is not possible to use the file system or other type of traditional database (e.g. a Kaggle scripts) and absolutely not in concurrent processes. In fact, using this connection in different processes will result in two independent searches **not** sharing any information.

Parameters **from_file** – The name of a file containing a pickled data frame connection.

Using this connection requires small adjustments to the proposed main script. When the main process finishes, all data will vanish if not explicitly writen to disk. Thus, instead of doing a single evaluation, the main process will incorporate a loop calling the search/sample `next` method multiple times. Additionally, at the end of the experiment, either extract the best configuration using *`results_as_dataframe()`* or write all the data using `pickle`.

results_as_dataframe()
Compile all the results and transform them using the space specified in the database. It is safe to use this method while other experiments are still writing to the database.

Returns A `pandas.DataFrame` containing all results with its `"_chocolate_id"` as `"id"`, their parameters and its loss. Pending results have a loss of `None`.

lock(**args*, ***kwargs*)
This function does not lock anything. Do not use in concurrent processes.

all_results()
Get a list of all entries of the result table. The order is undefined.

insert_result(*document*)
Insert a new *document* in the result data frame. The columns does not need to be defined nor all present. Any new column will be added to the database and any missing column will get value None.

update_result(*document*, *value*)
Update or add *value* of given rows in the result data frame.

Parameters

- **document** – An identifier of the rows to update.
- **value** – A mapping of values to update or add.

count_results()
Get the total number of entries in the result table.

all_complementary()
Get all entries of the complementary information table as a list. The order is undefined.

insert_complementary(*document*)
Insert a new document (row) in the complementary information data frame.

find_complementary(*filter*)
Find a document (row) from the complementary information data frame.

get_space()
Returns the space used for previous experiments.

insert_space(*space*)
Insert a space in the database.

Raises `AssertionError` – If a space is already present.

clear()
Clear all data.

10.3 Sampling Algorithms

class chocolate.**Grid**(*connection*, *space*, *crossvalidation=None*, *clear_db=False*)
Regular cartesian grid sampler.

Samples the search space at every point of the grid formed by all dimensions. It requires every dimension to be a discrete distribution.

Parameters

- **connection** – A database connection object.
- **space** – The search space to explore with only discrete dimensions.
- **crossvalidation** – A cross-validation object that handles experiment repetition.
- **clear_db** – If set to `True` and a conflict arise between the provided space and the space in the database, completely clear the database and set the space to the provided one.

next()
Retrieve the next point to evaluate based on available data in the database.

Returns A tuple containing a unique token and a fully qualified parameter set.

update(*token*, *values*)
Update the loss of the parameters associated with *token*.

Parameters

- **token** – A token generated by the sampling algorithm for the current parameters
- **values** – The loss of the current parameter set. The values can be a single `Number`, a `Sequence` or a `Mapping`. When a sequence is given, the column name is set to "_loss_i" where "i" is the index of the value. When a mapping is given, each key is prefixed with the string "_loss_".

class chocolate.**Random**(*connection*, *space*, *crossvalidation=None*, *clear_db=False*, *random_state=None*)

Random sampler.

Samples the search space randomly. This sampler will draw random numbers for each entry in the database in order to restore the random state for reproductibility when used concurrently with other random samplers.

If all parameters are discrete, the sampling is made without replacement. Otherwise, the exploration is conducted independently of conditional search space, meaning that each subspace will receive approximately the same number of samples.

Parameters

- **connection** – A database connection object.
- **space** – The search space to explore with only discrete dimensions. The search space can be either a dictionary or a *chocolate.Space* instance.
- **crossvalidation** – A cross-validation object that handles experiment repetition.
- **clear_db** – If set to `True` and a conflict arise between the provided space and the space in the database, completely clear the database and set the space to the provided one.
- **random_state** – Either a `numpy.random.RandomState` instance, an object to initialize the random state with or `None` in which case the global state is used.

next()

Retrieve the next point to evaluate based on available data in the database.

Returns A tuple containing a unique token and a fully qualified parameter set.

update(*token*, *values*)

Update the loss of the parameters associated with *token*.

Parameters

- **token** – A token generated by the sampling algorithm for the current parameters
- **values** – The loss of the current parameter set. The values can be a single `Number`, a `Sequence` or a `Mapping`. When a sequence is given, the column name is set to "_loss_i" where "i" is the index of the value. When a mapping is given, each key is prefixed with the string "_loss_".

class chocolate.**QuasiRandom**(*connection*, *space*, *crossvalidation=None*, *clear_db=False*, *seed=None*, *permutations=None*, *skip=0*)

Quasi-Random sampler.

Samples the search space using the generalized Halton low-discrepancy sequence. The underlying sequencer is the ghalton package, it must be installed separatly. The exploration is conducted independently of conditional search space, meaning that each subspace will receive approximately the same number of samples.

This sampler will draw random numbers for each entry in the database to restore the random state for reproductibility when used concurrently with other random samplers.

Parameters

- **connection** – A database connection object.
- **space** – The search space to explore with only discrete dimensions. The search space can be either a dictionary or a *chocolate.Space* instance.
- **crossvalidation** – A cross-validation object that handles experiment repetition.
- **clear_db** – If set to `True` and a conflict arise between the provided space and the space in the database, completely clear the database and set the space to the provided one.

- **seed** – An integer used as seed to initialize the sequencer with or `None` in which case the global state is used. This argument is ignored if `permutations` if provided.
- **permutations** – Either, the string `"ea"` in which case the `ghalton.EA_PERMS` are used or a valid list of permutations as desbribed in the ghalton package.
- **skip** – The number of points to skip in the sequence before the first point is sampled.

next()
Retrieve the next point to evaluate based on available data in the database.

Returns A tuple containing a unique token and a fully qualified parameter set.

update(*token*, *values*)
Update the loss of the parameters associated with *token*.

Parameters

- **token** – A token generated by the sampling algorithm for the current parameters
- **values** – The loss of the current parameter set. The values can be a single `Number`, a `Sequence` or a `Mapping`. When a sequence is given, the column name is set to "_loss_i" where "i" is the index of the value. When a mapping is given, each key is prefixed with the string "_loss_".

10.4 Search Algorithms

class chocolate.**Bayes**(*connection*, *space*, *crossvalidation=None*, *clear_db=False*, *n_bootstrap=10*, *utility_function='ucb'*, *kappa=2.756*, *xi=0.1*)
Bayesian minimization method with gaussian process regressor.

This method uses scikit-learn's implementation of gaussian processes with the addition of a conditional kernel when the provided space is conditional [Lévesque2017]. Two acquisition functions are made available, the Upper Confidence Bound (UCB) and the Expected Improvement (EI).

Parameters

- **connection** – A database connection object.
- **space** – the search space to explore with only discrete dimensions.
- **crossvalidation** – A cross-validation object that handles experiment repetition.
- **clear_db** – If set to `True` and a conflict arise between the provided space and the space in the database, completely clear the database and set set the space to the provided one.
- **n_bootstrap** – The number of random iteration done before using gaussian processes.
- **utility_function** (*str*) – The acquisition function used for the bayesian optimization. Two functions are implemented: "ucb" and "ei".
- **kappa** – Kappa parameter for the UCB acquisition function.
- **xi** – xi parameter for the EI acquisition function.

next()
Retrieve the next point to evaluate based on available data in the database.

Returns A tuple containing a unique token and a fully qualified parameter set.

update(*token*, *values*)
Update the loss of the parameters associated with *token*.

Parameters

- **token** – A token generated by the sampling algorithm for the current parameters
- **values** – The loss of the current parameter set. The values can be a single `Number`, a `Sequence` or a `Mapping`. When a sequence is given, the column name is set to "_loss_i" where "i" is the index of the value. When a mapping is given, each key is prefixed with the string "_loss_".

class chocolate.**CMAES**(*connection*, *space*, *crossvalidation=None*, *clear_db=False*, ***params*)

Covariance Matrix Adaptation Evolution Strategy minimization method.

A CMA-ES strategy that combines the $(1 + \lambda)$ paradigm [Igel2007], the mixed integer modification [Hansen2011] and active covariance update [Arnold2010]. It generates a single new point per iteration and adds a random step mutation to dimensions that undergoes a too small modification. Even if it includes the mixed integer modification, CMA-ES does not handle well dimensions without variance and thus it should be used with care on search spaces with conditional dimensions.

Parameters

- **connection** – A database connection object.
- **space** – The search space to explore.
- **crossvalidation** – A cross-validation object that handles experiment repetition.
- **clear_db** – If set to `True` and a conflict arise between the provided space and the space in the database, completely clear the database and set the space to the provided one.
- ****params** – Additional parameters to pass to the strategy as described in the following table, along with default values.

Parameter	Default value	Details
`d`	`1 + ndim / 2`	Damping for step-size.
`ptarg`	`1 / 3`	Taget success rate.
`cp`	`ptarg / (2 + ptarg)`	Step size learning rate.
`cc`	`2 / (ndim + 2)`	Cumulation time horizon.
`ccovp`	`2 / (ndim**2 + 6)`	Covariance matrix positive learning rate.
`ccovn`	`0.4 / (ndim**1.6 + 1)`	Covariance matrix negative learning rate.
`pthresh`	`0.44`	Threshold success rate.

Note: To reduce sampling, the constraint to the search space bounding box is enforced by repairing the individuals and adjusting the taken step. This will lead to a slight over sampling of the boundaries when local optimums are close to them.

next()

Retrieve the next point to evaluate based on available data in the database.

Returns A tuple containing a unique token and a fully qualified parameter set.

update(*token*, *values*)

Update the loss of the parameters associated with *token*.

Parameters

- **token** – A token generated by the sampling algorithm for the current parameters

- **values** – The loss of the current parameter set. The values can be a single `Number`, a `Sequence` or a `Mapping`. When a sequence is given, the column name is set to "_loss_i" where "i" is the index of the value. When a mapping is given, each key is prefixed with the string "_loss_".

class chocolate.**MOCMAES**(*connection*, *space*, *mu*, *crossvalidation=None*, *clear_db=False*, ***params*)

Multi-Objective Covariance Matrix Adaptation Evolution Strategy.

A CMA-ES strategy for multi-objective optimization. It combines the improved step size adaptation [Voss2010] and the mixed integer modification [Hansen2011]. It generates a single new point per iteration and adds a random step mutation to dimensions that undergoes a too small modification. Even if it includes the mixed integer modification, MO-CMA-ES does not handle well dimensions without variance and thus it should be used with care on search spaces with conditional dimensions.

Parameters

- **connection** – A database connection object.
- **space** – The search space to explore.
- **crossvalidation** – A cross-validation object that handles experiment repetition.
- **mu** – The number of parents used to generate the candidates. The higher this number is the better the Parato front coverage will be, but the longer it will take to converge.
- **clear_db** – If set to `True` and a conflict arise between the provided space and the space in the database, completely clear the database and set the space to the provided one.
- ****params** – Additional parameters to pass to the strategy as described in the following table, along with default values.

Parameter	Default value	Details
d	`1 + ndim / 2`	Damping for step-size.
ptarg	`1 / 3`	Taget success rate.
cp	`ptarg / (2 + ptarg)`	Step size learning rate.
cc	`2 / (ndim + 2)`	Cumulation time horizon.
ccov	`2 / (ndim**2 + 6)`	Covariance matrix learning rate.
pthresh	`0.44`	Threshold success rate.
indicator	`mo.hypervolume_indicator`	Indicator function used for ranking candidates

Note: To reduce sampling, the constraint to the search space bounding box is enforced by repairing the individuals and adjusting the taken step. This will lead to a slight over sampling of the boundaries when local optimums are close to them.

next()

Retrieve the next point to evaluate based on available data in the database.

Returns A tuple containing a unique token and a fully qualified parameter set.

update(*token*, *values*)

Update the loss of the parameters associated with *token*.

Parameters

- **token** – A token generated by the sampling algorithm for the current parameters
- **values** – The loss of the current parameter set. The values can be a single `Number`, a `Sequence` or a `Mapping`. When a sequence is given, the column name is set to "_loss_i"

where "i" is the index of the value. When a mapping is given, each key is prefixed with the string "_loss_".

10.5 Conditional Exploration

class chocolate.**ThompsonSampling**(*algo*, *connection*, *space*, *crossvalidation=None*, *clear_db=False*, *random_state=None*, *gamma=0.9*, *epsilon=0.05*, *algo_params=None*)

Conditional subspaces exploration strategy.

Thompson sampling wrapper to sample subspaces proportionally to their estimated quality. Each subspace of a conditional search space will be treated independently. This version uses an estimated moving average for the reward and forgets the reward of unselected subspaces allowing to model the dynamics of the underlying optimizers. Thompson sampling for Bernoulli bandit is described in [Chapelle2011].

Parameters

- **algo** – An algorithm to sample/search each subspace.
- **connection** – A database connection object.
- **space** – The conditional search space to explore.
- **crossvalidation** – A cross-validation object that handles experiment repetition.
- **clear_db** – If set to `True` and a conflict arise between the provided space and the space in the database, completely clear the database and insert set the space to the provided one.
- **random_state** – An instance of `RandomState`, an object to initialize the internal random state with, or None, in which case the global numpy random state is used.
- **gamma** – Estimated moving average learning rate. The higher, the faster will react the bandit to a change of best arm. Should be in [0, 1].
- **epsilon** – Forget rate for unselected arms. Th higher, the faster unselected arms will fallback to a symmetric distribution. Should be in [0, 1].
- **algo_params** – A dictionary of the parameters to pass to the algorithm.

next()

Retrieve the next point to evaluate based on available data in the database.

Returns A tuple containing a unique token and a fully qualified parameter set.

update(*token*, *values*)

Update the loss of the parameters associated with *token*.

Parameters

- **token** – A token generated by the sampling algorithm for the current parameters
- **values** – The loss of the current parameter set. The values can be a single `Number`, a `Sequence` or a `Mapping`. When a sequence is given, the column name is set to "_loss_i" where "i" is the index of the value. When a mapping is given, each key is prefixed with the string "_loss_".

10.6 Cross-Validation

class chocolate.**Repeat**(*repetitions*, *reduce=<function mean>*, *rep_col='_repetition_id'*)

Repeats each experiment a given number of times and reduces the losses for the algorithms.

The repetition cross-validation wraps the connection to handle repetition of experiments in the database. It is transparent to algorithms as it reduces the loss of repeated parameters and returns a list of results containing a single instance of each parameter set when `all_results()` is called. If not all repetitions values are entered in the database before the next point is generated by the algorithm, the algorithm will see the reduced loss of the parameters that are completely evaluated only. Alternatively, if no repetition has finished its evaluation, the algorithm will see a `None` as loss. *`Repeat`* also handles assigning a repetition number to the tokens since the `_chocolate_id` will be repeated. Other token values, such as *`ThompsonSampling`*'s `_arm_id`, are also preserved.

Parameters

- **repetitions** – The number of repetitions to do for each experiment.
- **reduce** – The function to reduce the valid losses, usually average or median.
- **rep_col** – The database column name for the repetition number, it has to be unique.

10.7 Multi-objective Tools

chocolate.mo.**argsortNondominated**(*losses*, *k*, *first_front_only=False*)

Sort input in Pareto-equal groups.

Sort the first *k losses* into different nondomination levels using the "Fast Nondominated Sorting Approach" proposed by Deb et al., see [Deb2002]. This algorithm has a time complexity of $O(MN^2)$, where M is the number of objectives and N the number of losses.

Parameters

- **losses** – A list of losses to select from.
- **k** – The number of elements to select.
- **first_front_only** – If `True` sort only the first front and exit.

Returns A list of Pareto fronts (lists) containing the losses index.

chocolate.mo.**hypervolume_indicator**(*front*, ***kargs*)

Indicator function using the hypervolume value.

Computes the contribution of each of the front candidates to the front hypervolume. The hypervolume indicator assumes minimization.

Parameters

- **front** – A list of Pareto equal candidate solutions.
- **ref** – The origin from which to compute the hypervolume (optional). If not given, ref is set to the maximum value in each dimension + 1.

Returns The index of the least contributing candidate.

chocolate.mo.**hypervolume**(*pointset*, *ref*)

Computes the hypervolume of a point set.

Parameters

- **pointset** – A list of points.

- **ref** – The origin from which to comute the hypervolume. This value should be larger than all values in the point set.

Returns The hypervolume of this point set.

Search Space Representation

Space	Representation of the search space.
Distribution	Base class for every Chocolate distributions.
ContinuousDistribution	Base class for every Chocolate continuous distributions.
QuantizedDistribution	Base class for every Chocolate quantized distributions.
uniform	Uniform continuous distribution.
quantized_uniform	Uniform discrete distribution.
log	Logarithmic uniform continuous distribution.
quantized_log	Logarithmic uniform discrete distribution.
choice	Uniform choice distribution between non-numeric samples.

Database Connections

SQLiteConnection	Connection to a SQLite database.
MongoDBConnection	Connection to a MongoDB database.
DataFrameConnection	Connection to a pandas DataFrame.

Sampling Algorithms

Grid	Regular cartesian grid sampler.
Random	Random sampler.
QuasiRandom	Quasi-Random sampler.

Search Algorithms

Bayes	Bayesian minimization method with gaussian process regressor.
CMAES	Covariance Matrix Adaptation Evolution Strategy minimization method.
MOCMAES	Multi-Objective Covariance Matrix Adaptation Evolution Strategy.

Conditional Exploration

ThompsonSampling	Conditional subspaces exploration strategy.

Cross-Validation

Repeat	Repeats each experiment a given number of times and reduces the losses for the algorithms.

Multi-objective Tools

mo.argsortNondominated	Sort input in Pareto-equal groups.
mo.hypervolume_indicator	Indicator function using the hypervolume value.
mo.hypervolume	Computes the hypervolume of a point set.

CHAPTER 11

Release Notes

CHAPTER 12

About

Chocolate is developped at NovaSyst.

Bibliography

[Lévesque2017] Lévesque, Durand, Gagné and Sabourin. Bayesian Optimization for Conditional Hyperparameter Spaces. 2017

[Igel2007] Igel, Hansen, Roth. Covariance matrix adaptation for multi-objective optimization. 2007

[Arnold2010] Arnold and Hansen. Active covariance matrix adaptation for the (1 + 1)-CMA-ES. 2010.

[Hansen2011] Hansen. A CMA-ES for Mixed-Integer Nonlinear Optimization. Research Report] RR-7751, INRIA. 2011

[Voss2010] Voss, Hansen, Igel. Improved Step Size Adaptation for the MO-CMA-ES. In proc. GECCO‘10, 2010.

[Hansen2011] Hansen. A CMA-ES for Mixed-Integer Nonlinear Optimization. [Research Report] RR-7751, INRIA. 2011

[Chapelle2011] O. Chapelle and L. Li, “An empirical evaluation of Thompson sampling”, in Advances in Neural Information Processing Systems 24 (NIPS), 2011.

[Deb2002] Deb, Pratab, Agarwal, and Meyarivan, “A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II”, 2002.

Python Module Index

C

Index

Symbols

A

B

C

D

F

G

H

I

L

M

N

Q

R

S

T

U